Logic Programming with Linear Logic

Michael David Winikoff

1997

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

Department of Computer Science

School of Electrical Engineering and Computer Science

The University of Melbourne

Parkville, Victoria 3052

AUSTRALIA

Abstract

Programming languages are the basic tools of computer science. The design of a good programming language is a trade-off between many factors. Perhaps the most important and difficult trade-off is between execution efficiency and programmer efficiency. Higher level languages reduce the amount of work the programmer has to do; however they have, to date, been less efficient than lower level languages. That lower level languages are *necessarily* more efficient is a piece of folklore which is under attack – higher level languages are constantly coming closer to the performance of the lower level languages. A consequence of this constantly closing performance gap is that the abstraction level of programming languages is slowly but inevitably rising.

A class of programming languages which has been described as "very high level" is *declarative programming languages*. Declarative programming languages have simple formal semantics and are easier to reason about and to construct tools for than more traditional programming languages. However these languages do suffer from a number of problems. They are weak at expressing side effects and concurrency. Side effects are generally used to perform I/O and as a result declarative languages have been weak at expressing I/O. Declarative languages are also weak at expressing concurrency without compromising their semantic purity and as a result tend to be weak at expressing graphical user interfaces.

Girard's *linear logic* promises solutions to some of these problems – linear logic is capable of modelling updates, it has inspired linear types which enable side effects to be safely introduced for efficiency reasons, and linear logic can model concurrent behavior cleanly.

This thesis focuses on the derivation of a logic programming language based on lin-

ear (rather than classical) logic. Our hypothesis is that it is possible to derive a logic programming language from linear logic. This language can be effectively implemented and is an expressive language, allowing updates, concurrency and other idioms to be cleanly expressed.

We investigate the systematic derivation of logic programming languages from logics and propose a taxonomy of "litmus tests" which can determine whether a subset of a logic can be viewed as a logic programming language. One of the tests developed is applied in order to derive a logic programming language based on the full multiple conclusion linear logic. The language is named *Lygon*.

We derive an efficient set of rules for managing resources in linear logic proofs and illustrate how the selection of a goal formula can be made more deterministic using heuristics derived from known properties of linear logic proofs.

Finally we investigate programming idioms in Lygon and present a range of programs which illustrate the language's expressiveness.

Acknowledgements

I would like to thank my two supervisors – James Harland and Harald Søndergaard, for their support and guidance. Without their fast and accurate proof-reading this thesis would undoubtedly have the odd (additional) inconsistency or two.

The other half of the original Lygon duo, David Pym, has continued to offer comments from afar. I would also like to thank David for interesting and stimulating discussion.

In the last few years the Lygon team has grown at RMIT and included a number of students whose work has advanced the cause of Lygon. In particular, the Lygon debugger [153] is the work of Yi Xiao Xu. Yi Xiao is also the author of programs 14 and 15.

I would like to thank an anonymous referee (of a paper based on chapter 4) for pointing us to Hodas' thesis and for suggesting an encoding of Lygon in Lolli.

I would like to thank the examiners of this thesis for their careful reading and useful comments. In particular, chapter 3 benefited from the detailed critique provided.

Thank go to the Australian Research Council, the Collaborative Information Technology Research Institute, the Centre for Intelligent Decision Systems, the Machine Intelligence Project, the Department of Computer Science and the School of Graduate Studies for financial support. Thanks also go to the administrative and system support staff at the Department of Computer Science for providing a working environment and invisibly keeping things working behind the scenes.

Last but not least, I would like to thank my family: My parents for countless small and many large things, my fiancé, Leanne Veitch, for support above and beyond the call of duty through the last few weeks, and my sister, Yael, for no apparent reason (she wanted to be here!).

Thanks to you all!

Dedication

To the memory of my grandfather, Paul Feher (3.1.1907-8.10.1995).

Contents

List of Figures

List of Programs

Algorithms

Preface

Early work on the implementation of Lygon was reported on at the Australasian Computer Science Conference in 1995 [149]. A (rather abridged) version of chapter 4 appeared at the International Logic Programming Symposium later that year [150]. Some preliminary work on Lygon programming was presented as a poster at that conference [58]. Other material on which chapter 5 is based was presented at the 1996 Australasian Computer Science Conference [152] and at the 1996 conference on Algebraic Methodology and Software Technology [60] where the Lygon system was demonstrated [59]. A preliminary version of chapter 4 appeared as a technical report [148] as did an early version of chapter 3 [151]. Section 5.1 is based on the Lygon 0.7 reference manual [146]. The reference manual also contains a section on Lygon programming.

This thesis is less than 100,000 words in length, exclusive of tables, bibliographies, footnotes and appendices. The work in this thesis has not been published elsewhere, except as noted above.

Michael Winikoff

Melbourne, Australia, March 1997

Chapter 1

Introduction

Programming languages are the basic tools of computer science. Although from a theoretical perspective any (reasonable) programming language is equivalent to any other, there are many reasons why certain programming notations may be better than others. The design of a good programming language is a trade-off between factors such as efficient execution, fast compilation, programmer efficiency, tool support, safety[1], portability and mobility.

Perhaps the most important and difficult trade-off is between execution efficiency and programmer efficiency. Higher level languages reduce the amount of work the programmer has to do; however they have, to date, been less efficient than lower level languages. That lower level languages are *necessarily* more efficient is a piece of folklore which is under attack – higher level languages are constantly coming closer to the performance of the lower level languages. A consequence of this constantly closing performance gap is that the abstraction level of programming languages is slowly but inevitably rising. Research on the design, application and implementation of higher level languages yields ideas which are eventually incorporated into mainstream programming languages[2]. Additionally, higher level languages are increasingly being applied directly with promising

[1] Once this was defined as "can programs written in the language crash the machine?". With the proliferation of memory protection the question becomes "does the program need to be debugged at the conceptual level of byte arrays?".

[2] It has been said that the time it takes for a programming construct to become accepted and move from an experimental language to a mainstream one is twenty years.

results [23, 26, 37, 72, 100, 119, 129].

A class of programming languages which has been described as "very high level" is *declarative programming languages*. One definition of the class of declarative programming languages proposed by Lloyd [96, 97] is that a declarative programming language equates a program with a theory in a logic and a computation with a deduction in the logic. A logic is defined as a formal system with

1. A proof theory,
2. A model theory,
3. A soundness theorem, and
4. (hopefully!) a completeness theorem.

There are two major sub-classes of declarative languages: *functional programming languages* [74] which are based on Church's λ-calculus, and *logic programming languages* [134] which view computation as proof search in a logic. Declarative programming languages are predicated on the assumption that a clean and simple formal semantics is important and useful. Important, since it provides an explanation of the language to programmers, serves as an implementation-neutral contract for implementors and allows the collected knowledge of nearly a hundred years of intensive research into mathematical logic to be leveraged. A simple formal semantics is also useful in that it simplifies proofs of correctness and aids in the construction of programming tools. It is no coincidence that powerful programming tools such as partial evaluators, abstract interpreters and declarative debuggers were first developed for (pure) declarative languages.

Declarative programming languages have a number of benefits when compared to more traditional programming languages — they are higher level, have simple formal semantics and are easier to reason about and to construct tools for. However they do suffer from a number of problems. Current implementations are inefficient (although, as noted above, the performance of these languages is constantly improving [62, 63]) and understanding the performance of programs written in these languages can occasionally be difficult. Declarative languages also tend to lack expressiveness in certain areas — they are not good at expressing I/O or concurrency for example. Since graphical user

interfaces (GUIs) generally involve concurrent behavior, expressing GUIs in declarative languages is still a subject of research.

Girard's *linear logic* [46] promises solutions to some of these problems – linear logic is capable of modelling updates, it has inspired linear types [33, 143] which enable side effects to be safely introduced for efficiency reasons and linear logic can model concurrent behavior cleanly.

This thesis focuses on the derivation of a logic programming language based on linear logic. Our hypothesis is that:

> *It is possible to derive a logic programming language from linear logic. This language can be effectively implemented and is an expressive language, allowing updates, concurrency and other idioms to be cleanly expressed.*

The contributions of this thesis are threefold:

1. We investigate the systematic derivation of logic programming languages from logics. We show that logical equivalences which can be proven in a logic do not necessarily hold in a fragment of the logic (section 3.1). The main contribution of this part of the thesis is to propose a taxonomy of "litmus tests" which can determine whether a subset of a logic can be viewed as a logic programming language. The tests are not specific to linear logic and could be applied to a range of logics (e.g. modal logic, relevant logic, etc.) This work extends our understanding of the essence of logic programming. One of the tests developed is applied in order to derive a logic programming language based on the full multiple conclusion linear logic. The language is named *Lygon*.

2. We derive an efficient set of rules for managing resources in linear logic proofs. These rules are applicable both to the implementation of linear logic programming languages and to linear logic theorem proving. We also illustrate how the selection of a goal formula (when there are multiple formulae in a goal) can be made more deterministic using heuristics derived from known properties of linear logic proofs.

3. We investigate programming idioms in Lygon and present a range of programs which illustrate the language's expressiveness.

4. We compare Lygon to a number of related languages including other logic programming languages based on linear logic and concurrent logic programming languages. We show that Lygon subsumes many of these.

The thesis begins with some background (chapter 2). We then investigate the essence of logic programming and the systematic derivation of logic programming languages (chapter 3). In chapter 4 we look at the implementation of Lygon and in chapter 5 we look at the language's applications and programming methodology. We compare Lygon to other work in chapter 6 and conclude with a discussion and a brief look at further work (chapter 7).

Chapter 2

Background

2.1 Sequent Calculus

The *sequent calculus* is a formalism due to Gentzen [45] for representing inferences and proofs. It is the standard notation used in the proof theoretical analysis of logic programming since it distinguishes naturally between programs and goals. Additionally the sequent calculus rules construct a proof *locally* (as opposed to natural deduction [82]) and allow short direct proofs (as opposed to Hilbert-type systems [82]). This makes the sequent calculus appropriate for *systematic* (and hence automatable) proof search.

A *sequent* is a construct of the form $\Gamma \vdash \Delta$ where Γ and Δ are sequences of formulae. Γ is the *antecedent* and Δ is the *succedent*. A sequent is generally read as *"if all of the* Γ *are true then at least one of the* Δ *is true"*. Note that Γ and Δ may be empty. For example the sequent $p, q \vdash p$ is provable classically and can be read as "p follows from the assumptions p and q".

An inference is a construct of the form

$$\frac{\Pi_1 \quad \ldots \quad \Pi_n}{\Pi_o}\ L$$

where L is a name identifying the inference and the Π are sequents. The inference should be read as *"The conclusion* Π_0 *is derived from the premises* $\Pi_1 \ \ldots \ \Pi_n$*"*. We shall see an example shortly.

Usually for each connective there are two rules - one for the left side and one for the right. For example the classical logic rules for $\wedge$ are

$$\frac{\Gamma, F_1, F_2 \vdash \Delta}{\Gamma, F_1 \wedge F_2 \vdash \Delta} \wedge L \qquad \frac{\Gamma \vdash F_1, \Delta \quad \Gamma \vdash F_2, \Delta}{\Gamma \vdash F_1 \wedge F_2, \Delta} \wedge R$$

Rules that deal with logical connectives (known as *logical rules*) generally leave most of the formulae unchanged and have a single formula in the conclusion which has its top-most connective removed and the resulting subformulae appropriately distributed. The formula which is decomposed ($F_1 \wedge F_2$ above) is known as the *principal* formula. The subformulae of the principal formula (F_1 and F_2 above) are known as *active* or *side* formulae. The unchanged formulae (Γ and Δ above) are the *context*.

In addition to the logical rules there is an axiom rule which states that any atom follows from its assumption (we use p throughout to denote an atomic formulae such as *append([1],[2],[1,2])* or *q(1)*):

$$\frac{}{p \vdash p} Ax$$

Note that for any logic $F \vdash F$ will be derivable for any formula F. In addition to the axiom rule there are also the *structural* rules and the *cut* rule. The structural rules apply to any formula and (in classical logic) allow the order of formulae in a sequent to be changed (**E**xchange), formulae to be deleted[1] (**W**eakening) and additional copies of an existing formula created (**C**ontraction). Often it is convenient to ignore the structural rules by treating Γ as being sets (for classical and intuitionistic logic) and multisets (for linear logic).

Inference rules are given in figure 2.1 for classical logic, figure 2.2 for intuitionistic logic and figures 2.4 and 2.5 for linear logic.

In the linear logic sequent calculus rules, the rule

$$\frac{!\Gamma \vdash F, ?\Delta}{!\Gamma \vdash !F, ?\Delta} !R$$

is applicable only if every formulae in the antecedent is of the form $!G$ and every formulae (other than $!F$) in the succedent is of the form $?G$.

[1] When read bottom up – for a top down reading the opposite behavior occurs.

Figure 2.1 Classical Logic (LK)

$$\frac{}{p \vdash p}\ \text{axiom} \qquad \frac{\Gamma \vdash F, \Delta \qquad \Gamma, F \vdash \Delta}{\Gamma \vdash \Delta}\ \text{cut}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, F \vdash \Delta}\ \text{W-L} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash F, \Delta}\ \text{W-R}$$

$$\frac{\Gamma, F, F \vdash \Delta}{\Gamma, F \vdash \Delta}\ \text{C-L} \qquad \frac{\Gamma \vdash F, F, \Delta}{\Gamma \vdash F, \Delta}\ \text{C-R}$$

$$\frac{\Gamma, F, G \vdash \Delta}{\Gamma, G, F \vdash \Delta}\ \text{E-L} \qquad \frac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash G, F, \Delta}\ \text{E-R}$$

$$\frac{\Gamma, F, G \vdash \Delta}{\Gamma, F \wedge G \vdash \Delta}\ \wedge\text{-L} \qquad \frac{\Gamma \vdash F, \Delta \qquad \Gamma \vdash G, \Delta}{\Gamma \vdash F \wedge G, \Delta}\ \wedge\text{-R}$$

$$\frac{\Gamma, F \vdash \Delta \qquad \Gamma, G \vdash \Delta}{\Gamma, F \vee G \vdash \Delta}\ \vee\text{-L} \qquad \frac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash F \vee G, \Delta}\ \vee\text{-R}$$

$$\frac{\Gamma \vdash F, \Delta \qquad \Gamma', G \vdash \Delta'}{\Gamma, \Gamma', F \to G \vdash \Delta, \Delta'}\ \to\text{-L} \qquad \frac{\Gamma, F \vdash G, \Delta}{\Gamma \vdash F \to G, \Delta}\ \to\text{-R}$$

$$\frac{\Gamma, F[t/x] \vdash \Delta}{\Gamma, \forall x F \vdash \Delta}\ \forall\text{-L} \qquad \frac{\Gamma \vdash F[y/x], \Delta}{\Gamma \vdash \forall x F, \Delta}\ \forall\text{-R}$$

$$\frac{\Gamma, F[y/x] \vdash \Delta}{\Gamma, \exists x F \vdash \Delta}\ \exists\text{-L} \qquad \frac{\Gamma \vdash F[t/x], \Delta}{\Gamma \vdash \exists x F, \Delta}\ \exists\text{-R}$$

$$\frac{\Gamma \vdash F, \Delta}{\Gamma, \neg F \vdash \Delta}\ \neg\text{-L} \qquad \frac{\Gamma, F \vdash \Delta}{\Gamma \vdash \neg F, \Delta}\ \neg\text{-R}$$

The rules $\forall$-R and $\exists$-L have the side condition that y is not free in Γ, F or Δ.

Figure 2.2 Intuitionistic Logic (LJ)

$$\frac{}{p \vdash p}\ \text{axiom} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash G}{\Gamma \vdash G}\ \text{cut}$$

$$\frac{\Gamma \vdash F'}{\Gamma, F \vdash F'}\ \text{W-L} \qquad \frac{\Gamma \vdash}{\Gamma \vdash F}\ \text{W-R}$$

$$\frac{\Gamma, F, F \vdash F'}{\Gamma, F \vdash F'}\ \text{C-L}$$

$$\frac{\Gamma, F, G \vdash F'}{\Gamma, G, F \vdash F'}\ \text{E-L}$$

$$\frac{\Gamma, F, G \vdash F}{\Gamma, F \wedge G \vdash F}\ \wedge\text{-L} \qquad \frac{\Gamma \vdash F \qquad \Gamma \vdash G}{\Gamma \vdash F \wedge G}\ \wedge\text{-R}$$

$$\frac{\Gamma, F \vdash F \qquad \Gamma, G \vdash F}{\Gamma, F \vee G \vdash F}\ \vee\text{-L} \qquad \frac{\Gamma \vdash F_i}{\Gamma \vdash F_1 \vee F_2}\ \vee\text{-R}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, G \vdash F}{\Gamma, F \to G \vdash F}\ \to\text{-L} \qquad \frac{\Gamma, F \vdash G}{\Gamma \vdash F \to G}\ \to\text{-R}$$

$$\frac{\Gamma, F'[t/x] \vdash F}{\Gamma, \forall x F' \vdash F}\ \forall\text{-L} \qquad \frac{\Gamma \vdash F[y/x]}{\Gamma \vdash \forall x F}\ \forall\text{-R}$$

$$\frac{\Gamma, F'[y/x] \vdash F}{\Gamma, \exists x F' \vdash F}\ \exists\text{-L} \qquad \frac{\Gamma \vdash F[t/x]}{\Gamma \vdash \exists x F}\ \exists\text{-R}$$

$$\frac{\Gamma \vdash F}{\Gamma, \neg F \vdash}\ \neg\text{-L} \qquad \frac{\Gamma, F \vdash}{\Gamma \vdash \neg F}\ \neg\text{-R}$$

The rules $\forall$-R and $\exists$-L have the side condition that y is not free in Γ, F or Δ.

Figure 2.3 Multiple Conclusioned Propositional Intuitionistic Sequent Calculus (LJM)
Γ and Δ are multisets.

$$\frac{}{\Gamma, p \vdash p, \Delta}\ Ax \qquad \frac{}{\Gamma, \bot \vdash \Delta}\ \bot - L$$

$$\frac{\Gamma, F, G \vdash \Delta}{\Gamma, F \wedge G \vdash \Delta} \wedge - L \qquad \frac{\Gamma \vdash F, \Delta \quad \Gamma \vdash G, \Delta}{\Gamma \vdash F \wedge G, \Delta} \wedge - R$$

$$\frac{\Gamma, F \vdash \Delta \quad \Gamma, G \vdash \Delta}{\Gamma, F \vee G, \Gamma \vdash \Delta} \vee - L \qquad \frac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash F \vee G, \Delta} \vee - R$$

$$\frac{\Gamma, F \to G \vdash F \quad \Gamma, G \vdash \Delta}{\Gamma, F \to G \vdash \Delta} \to -L \qquad \frac{\Gamma, F \vdash G}{\Gamma \vdash F \to G, \Delta} \to -R$$

[41] presents a group of rules which replace $\to -L$:

$$\frac{\Gamma, G, F \vdash \Delta}{\Gamma, F \to G, F \vdash \Delta} \to -L_1 \qquad \frac{\Gamma, E \to (H \to G) \vdash \Delta}{\Gamma, (E \wedge H) \to G \vdash \Delta} \to -L_2$$

$$\frac{\Gamma, E \to G, H \to G \vdash \Delta}{\Gamma, (E \vee H) \to G \vdash \Delta} \to -L_3 \qquad \frac{\Gamma, H \to G \vdash E \to H \quad \Gamma, G \vdash \Delta}{\Gamma, (E \to H) \to G \vdash \Delta} \to -L_4$$

A *proof* is a tree where the root is the sequent proven and the leaves are instances of the axiom rule. As an example consider proving that in classical logic $B \to ((B \to A) \to A)$ is always true

$$\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\ Ax \quad \dfrac{}{A \vdash A}\ Ax}{B, (B \to A) \vdash A} \to L}{B \vdash (B \to A) \to A} \to R}{\vdash B \to ((B \to A) \to A)} \to R$$

Note that when used for logic programming the inference rules are applied *bottom-up*.[2] We apply the inference rules to go from conclusions to premises. We are given a query and seek to prove or disprove it.

We use *left* (*right*) to refer to all rules (both structural and logical) which operate on

[2] Since proof trees are upside down to a computer scientist "bottom-up" here corresponds to what logic programmers normally call "top-down", i.e. the standard Prolog mechanism.

Figure 2.4 Linear Logic (LL)

$$\frac{}{p \vdash p}\ \text{axiom}$$

$$\frac{\Gamma \vdash F, \Delta \qquad \Gamma', F \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \text{cut}$$

$$\frac{\Gamma, F, G, \Gamma' \vdash \Delta}{\Gamma, G, F, \Gamma' \vdash \Delta}\ \text{E-L} \qquad \frac{\Gamma \vdash \Delta, F, G, \Delta'}{\Gamma \vdash \Delta, G, F, \Delta'}\ \text{E-R}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, \mathbf{1} \vdash \Delta}\ \mathbf{1}\text{-L} \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1}\text{-R}$$

$$\frac{}{\bot \vdash}\ \bot\text{-L} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta}\ \bot\text{-R}$$

$$\frac{}{\Gamma \vdash \top, \Delta}\ \top\text{-R}$$

$$\frac{}{\Gamma, \mathbf{0} \vdash \Delta}\ \mathbf{0}\text{-L}$$

$$\frac{\Gamma \vdash F, \Delta}{\Gamma, F^\bot \vdash \Delta}\ -^\bot\text{-L} \qquad \frac{\Gamma, F \vdash \Delta}{\Gamma \vdash F^\bot, \Delta}\ -^\bot\text{-R}$$

$$\frac{\Gamma, F, G \vdash \Delta}{\Gamma, F \otimes G \vdash \Delta}\ \otimes\text{-L} \qquad \frac{\Gamma \vdash F, \Delta \qquad \Gamma' \vdash G, \Delta'}{\Gamma, \Gamma' \vdash F \otimes G, \Delta, \Delta'}\ \otimes\text{-R}$$

$$\frac{\Gamma, F \vdash \Delta}{\Gamma, F \,\&\, G \vdash \Delta} \qquad \frac{\Gamma, G \vdash \Delta}{\Gamma, F \,\&\, G \vdash \Delta}\ \&\text{-L} \qquad \frac{\Gamma \vdash F, \Delta \qquad \Gamma \vdash G, \Delta}{\Gamma \vdash F \,\&\, G, \Delta}\ \&\text{-R}$$

$$\frac{\Gamma, F \vdash \Delta \qquad \Gamma, G \vdash \Delta}{\Gamma, F \oplus G \vdash \Delta}\ \oplus\text{-L} \qquad \frac{\Gamma \vdash F, \Delta}{\Gamma \vdash F \oplus G, \Delta} \qquad \frac{\Gamma \vdash G, \Delta}{\Gamma \vdash F \oplus G, \Delta}\ \oplus\text{-R}$$

$$\frac{\Gamma, F \vdash \Delta \qquad \Gamma', G \vdash \Delta'}{\Gamma, \Gamma', F \parr G \vdash \Delta, \Delta'}\ \parr\text{-L} \qquad \frac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash F \parr G, \Delta}\ \parr\text{-R}$$

$$\frac{\Gamma \vdash F, \Delta \qquad \Gamma', G \vdash \Delta'}{\Gamma, \Gamma', F \multimap G \vdash \Delta, \Delta'}\ \multimap\text{-L} \qquad \frac{\Gamma, F \vdash G, \Delta}{\Gamma \vdash F \multimap G, \Delta}\ \multimap\text{-R}$$

$$\frac{\Gamma, F \vdash \Delta}{\Gamma, !F \vdash \Delta}\ !\text{-L} \qquad \frac{!\Gamma \vdash F, ?\Delta}{!\Gamma \vdash !F, ?\Delta}\ !\text{-R}$$

$$\frac{!\Gamma, F \vdash ?\Delta}{!\Gamma, ?F \vdash ?\Delta}\ ?\text{-L} \qquad \frac{\Gamma \vdash F, \Delta}{\Gamma \vdash ?F, \Delta}\ ?\text{-R}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, !F \vdash \Delta}\ W!\text{-L} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash ?F, \Delta}\ W?\text{-R}$$

$$\frac{\Gamma, !F, !F \vdash \Delta}{\Gamma, !F \vdash \Delta}\ C!\text{-L} \qquad \frac{\Gamma \vdash ?F, ?F, \Delta}{\Gamma \vdash ?F, \Delta}\ C?\text{-R}$$

$$\frac{\Gamma, F[t/x] \vdash \Delta}{\Gamma, \forall x\,.\,F \vdash \Delta}\ \forall\text{-L} \qquad \frac{\Gamma \vdash F[y/x], \Delta}{\Gamma \vdash \forall x\,.\,F, \Delta}\ \forall\text{-R}$$

$$\frac{\Gamma, F[y/x] \vdash \Delta}{\Gamma, \exists x\,.\,F \vdash \Delta}\ \exists\text{-L} \qquad \frac{\Gamma \vdash F[t/x], \Delta}{\Gamma \vdash \exists x\,.\,F, \Delta}\ \exists\text{-R}$$

where y is not free in Γ, Δ.

Figure 2.5 Intuitionistic Linear Logic (ILL)

$$\frac{}{p \vdash p}\ \text{axiom}$$

$$\frac{\Gamma \vdash F \qquad \Gamma', F \vdash G}{\Gamma, \Gamma' \vdash G}\ \text{cut}$$

$$\frac{\Gamma, F, G, \Gamma' \vdash F}{\Gamma, G, F, \Gamma' \vdash F}\ \text{E-L}$$

$$\frac{\Gamma \vdash F}{\Gamma, \mathbf{1} \vdash F}\ \mathbf{1}\text{-L} \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1}\text{-R}$$

$$\frac{}{\bot \vdash}\ \bot\text{-L} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \bot}\ \bot\text{-R}$$

$$\frac{}{\Gamma \vdash \top}\ \top\text{-R}$$

$$\frac{}{\Gamma, \mathbf{0} \vdash \Delta}\ \mathbf{0}\text{-L}$$

$$\frac{\Gamma \vdash F}{\Gamma, F^{\bot} \vdash}\ {-^{\bot}}\text{-L} \qquad \frac{\Gamma, F \vdash}{\Gamma \vdash F^{\bot}}\ {-^{\bot}}\text{-R}$$

$$\frac{\Gamma, F, G \vdash F}{\Gamma, F \otimes G \vdash F}\ \otimes\text{-L} \qquad \frac{\Gamma \vdash F \qquad \Gamma' \vdash G}{\Gamma, \Gamma' \vdash F \otimes G}\ \otimes\text{-R}$$

$$\frac{\Gamma, F \vdash F}{\Gamma, F \,\&\, G \vdash F} \qquad \frac{\Gamma, G \vdash F}{\Gamma, F \,\&\, G \vdash F}\ \&\text{-L} \qquad \frac{\Gamma \vdash F \qquad \Gamma \vdash G}{\Gamma \vdash F \,\&\, G}\ \&\text{-R}$$

$$\frac{\Gamma, F \vdash F \qquad \Gamma, G \vdash F}{\Gamma, F \oplus G \vdash F}\ \oplus\text{-L} \qquad \frac{\Gamma \vdash F}{\Gamma \vdash F \oplus G} \qquad \frac{\Gamma \vdash G}{\Gamma \vdash F \oplus G}\ \oplus\text{-R}$$

$$\frac{\Gamma, F \vdash F \qquad \Gamma, G \vdash}{\Gamma, F \parr G \vdash F} \qquad \frac{\Gamma, F \vdash \qquad \Gamma, G \vdash F}{\Gamma, F \parr G \vdash F}\ \parr\text{-L}$$

$$\frac{\Gamma \vdash F \qquad \Gamma', G \vdash F}{\Gamma, \Gamma', F \multimap G \vdash F}\ \multimap\text{-L} \qquad \frac{\Gamma, F \vdash G}{\Gamma \vdash F \multimap G}\ \multimap\text{-R}$$

$$\frac{\Gamma, F \vdash F}{\Gamma, !F \vdash F}\ !\text{-L} \qquad \frac{!\Gamma \vdash F}{!\Gamma \vdash !F}\ !\text{-R}$$

$$\frac{!\Gamma, F \vdash ?F}{!\Gamma, ?F \vdash ?F}\ ?\text{-L} \qquad \frac{\Gamma \vdash F}{\Gamma \vdash ?F}\ ?\text{-R}$$

$$\frac{\Gamma \vdash F}{\Gamma, !F \vdash F}\ W!\text{-L} \qquad \frac{\Gamma \vdash}{\Gamma \vdash ?F}\ W?\text{-R}$$

$$\frac{\Gamma, !F, !F \vdash F}{\Gamma, !F \vdash F}\ C!\text{-L}$$

$$\frac{\Gamma, F[t/x] \vdash F}{\Gamma, \forall x\,.\,F \vdash F}\ \forall\text{-L} \qquad \frac{\Gamma \vdash F[y/x]}{\Gamma \vdash \forall x\,.\,F}\ \forall\text{-R}$$

$$\frac{\Gamma, F[y/x] \vdash F}{\Gamma, \exists x\,.\,F \vdash F}\ \exists\text{-L} \qquad \frac{\Gamma \vdash F[t/x]}{\Gamma \vdash \exists x\,.\,F}\ \exists\text{-R}$$

where y is not free in Γ, Δ.

formulae on the left (right) of the turnstile ($\vdash$). The rules Ax and Cut are neither left nor right.

A sequent calculus is *single conclusioned* if there is at most a single formula on the right of the turnstile, otherwise it is *multiple conclusioned*. Single conclusioned calculi do not possess the right contraction and right exchange rules. One of the important early results in the area is that limiting the standard sequent calculus for classical logic to a single conclusion gives a sequent calculus for intuitionistic logic [45]. Note that there exists a multiple conclusioned sequent calculus for intuitionistic logic (given in figure 2.3).

A sequent calculus is *one sided* if all succedents (or all antecedents) are empty. This can be achieved by shifting formulae across the turnstile by negating them and then using de Morgan rules to push negations inwards. Linear logic was first presented [46] as a one sided system since there are half as many rules involved.

For example, to construct a one sided sequent calculus for classical logic we can replace sequents $\Gamma \vdash \Delta$ with sequents $\vdash \neg\Gamma, \Delta$ and push negations inwards. The axiom rule becomes

$$\frac{}{\vdash p, \neg p}\ Ax$$

In figure 2.6 a one sided presentation of the rules of linear logic is given. The system ($\mathcal{L}$) also introduces a non-linear region (δ) which is duplicated by the $\otimes$ rule. This makes the $\otimes$ rule permutable with contraction (see section 2.4). Additionally, the axiom rule encodes applications of weakening. This system is used as a stepping stone in the completeness proof of the deterministic resource management rules presented in chapter 4. Showing the equivalence of this system to the standard rules in figure 2.4 is straightforward. The sequent $\delta : \Gamma$ is provable in $\mathcal{L}$ iff the sequent $\vdash \Gamma, ?\delta$ has an LL proof.

Two formulae, A and B, are *logically equivalent* if both $A \vdash B$ and $B \vdash A$ are provable. For example, in classical logic $A \to B$ is logically equivalent to $(\neg A) \vee B$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax \quad \dfrac{}{B \vdash B}\ Ax}{A, (A \to B) \vdash B}\ \to -L}{A \to B \vdash \neg A, B}\ \neg - R}{A \to B \vdash (\neg A) \vee B}\ \vee - R \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax}{A \vdash B, A}\ W - R \quad \dfrac{}{B \vdash B}\ Ax}{\neg A, A \vdash B}\ \neg - L \quad \dfrac{}{B, A \vdash B}\ W - L}{(\neg A) \vee B, A \vdash B}\ \vee - L}{(\neg A) \vee B \vdash A \to B}\ \to -R$$

Figure 2.6 One Sided Linear Logic ($\mathcal{L}$)

$$\frac{}{\delta : p, p^{\perp}}\ Ax \qquad \frac{}{\delta : \mathbf{1}}\ \mathbf{1}$$

$$\frac{}{\delta : \top, \Gamma}\ \top \qquad \frac{\delta : \Gamma}{\delta : \perp, \Gamma}\ \perp$$

$$\frac{\delta : F, \Gamma \quad \delta : G, \Delta}{\delta : F \otimes G, \Gamma, \Delta}\ \otimes \qquad \frac{\delta : F, \Gamma \quad \delta : G, \Gamma}{\delta : F \,\&\, G, \Gamma}\ \&$$

$$\frac{\delta : F, \Gamma}{\delta : F \oplus G, \Gamma}\ \oplus_1 \qquad \frac{\delta : F, G, \Gamma}{\delta : F \parr G, \Gamma}\ \parr$$

$$\frac{\delta : G, \Gamma}{\delta : F \oplus G, \Gamma}\ \oplus_2 \qquad \frac{\delta : F}{\delta :!F}\ !$$

$$\frac{F, \delta : \Gamma}{\delta :?F, \Gamma}\ ? \qquad \frac{F, \delta : F, \Gamma}{F, \delta : \Gamma}\ ?D$$

$$\frac{\delta : F[t/x], \Gamma}{\delta : \exists x F, \Gamma}\ \exists \qquad \frac{\delta : F[y/x], \Gamma}{\delta : \forall x F, \Gamma}\ \forall$$

where y is not free in Γ, Δ.

Figure 2.7 The Cut Rule

$$\frac{\Gamma, F \vdash \Delta \quad \Gamma \vdash F, \Delta}{\Gamma \vdash \Delta}\, Cut$$

Cut

The *cut* rule (figure 2.7[3]) occupies a special place in a number of respects. It is a structural rule in that can be applied to any formula. It is neither a left nor a right rule. Additionally, and most importantly, it is the only rule (in the logics which we will be considering) which does not satisfy the *sub-formula property*. The sub-formula property is satisfied by a sequent calculus rule if all formulae in the premis(es) of the rule are subformulae of formulae in the conclusion. The absence of this property for the cut rule is a severe impediment to bottom-up proof search since it requires that we "guess" the formula F.

The property of *cut elimination* states that if there exists a proof in some system with conclusion $\Gamma \vdash \Delta$ which makes use of the cut rule then there exists another proof with the same conclusion which does not make use of the cut rule. The cut elimination theorem is also known as Gentzen's Hauptsatz (main theorem) [45]. The cut elimination property has been shown to hold for all of the logics we shall be considering. That cut elimination holds is essential for bottom up proof search to be practical.

From cut elimination we can derive a number of useful properties. For example, modus ponens as a meta-level inference step can be shown to hold using cut. If we have proofs of $\vdash A$ and $\vdash A \to B$ then we can derive $\vdash B$:

$$\dfrac{\vdots}{\vdash A} \qquad \dfrac{\dfrac{\vdots}{\vdash A \to B} \qquad \dfrac{\dfrac{}{A \vdash A} \quad \dfrac{}{B \vdash B}}{A, A \to B \vdash B} \to -L}{A \vdash B}\, Cut}{\vdash B}\, Cut$$

For more details on the sequent calculus we refer the reader to [81].

[3] Note that the rule given in figure 2.7 corresponds to an *additive* presentation of the cut rule. In linear logic it is more natural to consider a *multiplicative* presentation where the formulae in the conclusion are split between the two premises. In classical and intuitionistic logic these two presentations are equivalent.

2.2 Intuitionistic Logic

Intuitionistic logic was developed early this century. It shares its syntax with classical logic but differs semantically in that it rejects the law of the excluded middle which states that $\vdash A \vee \neg A$ is true, *i.e.*, that predicates must be either true or false. A number of consequences follow, for example although $A \rightarrow \neg\neg A$, it is not the case that $\neg\neg A \rightarrow A$, thus A and $\neg\neg A$ are distinct. For our purposes it is sufficient to know that it can be specified by taking the standard sequent calculus for classical logic and limiting it to single conclusions. For more details we refer the reader to [81].

2.3 Linear Logic

Linear logic was introduced in a seminal 1987 paper by Girard [46] and has since inspired much work in the computer science and mathematical communities.

Whereas classical logic can be said to be based on the intuitive notion of truth, linear logic is intuitively based on the notion of *resources*. A predicate in linear logic is a resource. Resources can neither be duplicated nor discarded. Thus for example the sequent $\mathit{dollar}, \mathit{dollar} \vdash \mathit{dollar}$ which is provable classically does *not* hold in linear logic.

Linear logic provides connectives that allow *controlled* duplication and deletion of resources. This accounts in part for linear logic's richness and versatility.

As an example consider the familiar resource of pizza slices. Let us use *cap* to represent the resource of a single slice of capricciosa pizza and let *veg* represent a single slice of vegetarian pizza.

Resources can be supplied and consumed. These dual notions are entirely symmetrical in linear logic. If *cap* represents the consumption of a slice of pizza then its linear negation $\mathit{cap}^{\perp}$ ("neg cap" or "perp[4] cap") denotes the act of supplying a slice of pizza.

Due to the linearity of resources the familiar conjunction and disjunction are split into two connectives each. There are two conjunctions: $a \otimes b$ ("a cross b" or sometimes due to the visual appearance of the connective "a pizza b") splits the resources between the sub-proofs of a and b. $a \,\&\, b$ ("a with b") uses all of the resources to prove a and all of the

[4] from "perpendicular" - at right angles

resources to prove b. There are also two disjunctions: $a \oplus b$ ("a or b") has a fairly standard intuitive behaviour – it is provable if either a or b are. $a \parr b$ ("a par b" or "a tensum b") is the dual of $\otimes$. It places both a and b into the context. If one thinks as $\otimes$ as enabling multiple consumptions by splitting the resources between a and b then $\parr$ enables multiple supplies.

Linear logic has four logical constants:

$\top$**:** Is a version of truth which is provable in any context. It is the unit of $\&$ (i.e., $\top \& F = F$)

1: Is provable, but only in an empty context. It is the unit of $\otimes$.

$\bot$**:** Is the unit of $\parr$. It cannot be proved, but can be weakened away.

0: Is the unit of $\oplus$.

The formula

$$(\mathit{veg} \otimes \mathit{veg} \otimes \mathit{veg} \otimes \mathit{veg}) \oplus (\mathit{veg} \otimes \mathit{veg} \otimes \mathit{veg})$$

expresses the consumption of either three or four slices of vegetarian pizza. Using the constant **1** this could also be written as

$$\mathit{veg} \otimes \mathit{veg} \otimes \mathit{veg} \otimes (\mathit{veg} \oplus \mathbf{1})$$

The constant $\top$ can consume any amount of resources:

$$\frac{}{\vdash \top, \mathit{veg}, \mathit{cap}^{\bot}, \mathit{cap}}\ \top$$

The two *exponentials* "!" and "?" represent different notions of infinite resources. "?" represents a possibility for endless resources. ?*veg* ("why not veg") can be interpreted as someone who is content with any number of vegetarian pizzas including zero. On the other hand !*veg* ("bang veg" or "of course veg") represents someone who is only happy if given an infinite number of vegetarian pizzas. Conversely, $?(\mathit{veg}^{\bot})$ is a restaurant capable of producing any number of vegetarian pizzas and $!(\mathit{cap}^{\bot})$ is a strange establishment that insists on feeding its customers with an infinite amount of capricciosa pizza!

A certain very hungry person is happy only if given an infinite amount of pizzas but is not fussy about which type they are. We can represent this person using the following formula:

$$!(\mathit{veg} \oplus \mathit{cap})$$

The following formula represents an infinitary *supply* of vegetarian pizzas.

$$?(\mathit{veg}^{\perp})$$

Using the rules in figure 2.4 we can prove that an infinite supply of vegetarian pizzas satisfies *very hungry*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathit{veg} \vdash \mathit{veg}}\, Ax}{\vdash \mathit{veg}^{\perp}, \mathit{veg}}\, -^{\perp} - R}{\vdash ?(\mathit{veg}^{\perp}), \mathit{veg}}\, ? - R}{\vdash ?(\mathit{veg}^{\perp}), \mathit{veg} \oplus \mathit{cap}}\, \oplus - R}{\vdash ?(\mathit{veg}^{\perp}), !(\mathit{veg} \oplus \mathit{cap})}\, ! - R$$

The quantifiers are written as they are in classical logic and behave identically.

Linear logic possesses the following de Morgan rules:

$$(F_1 \otimes F_2)^{\perp} \equiv (F_1)^{\perp} \parr (F_2)^{\perp}$$
$$(F_1 \parr F_2)^{\perp} \equiv (F_1)^{\perp} \otimes (F_2)^{\perp}$$
$$(F_1 \oplus F_2)^{\perp} \equiv (F_1)^{\perp} \,\&\, (F_2)^{\perp}$$
$$(F_1 \,\&\, F_2)^{\perp} \equiv (F_1)^{\perp} \oplus (F_2)^{\perp}$$
$$(!F)^{\perp} \equiv ?(F)^{\perp}$$
$$(?F)^{\perp} \equiv !(F)^{\perp}$$
$$(\exists x F)^{\perp} \equiv \forall x (F)^{\perp}$$
$$(\forall x F)^{\perp} \equiv \exists x (F)^{\perp}$$
$$(\mathbf{1})^{\perp} \equiv \perp$$
$$(\perp)^{\perp} \equiv \mathbf{1}$$
$$(\mathbf{0})^{\perp} \equiv \top$$
$$(\top)^{\perp} \equiv \mathbf{0}$$

Good tutorial introductions to linear logic and its applications to computer science can be found in [3, 4, 123, 124].

2.4 Permutabilities

Permutabilities [83] play an important role in the proof theoretical analysis of logic programming languages. We say that (e.g.) $\otimes$-R permutes down over $\&$-L if whenever a proof contains an occurrence of $\otimes$-R immediately above[5] an occurrence of $\&$-L we can swap the order and retain a valid proof. For example in any context, the inference

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots\\ \Gamma, F_1 \vdash \Delta, F_3 & \Gamma' \vdash \Delta', F_4\end{array}}{\Gamma, \Gamma', F_1 \vdash \Delta, \Delta', F_3 \otimes F_4}\otimes - R}{\Gamma, \Gamma', F_1 \,\&\, F_2 \vdash \Delta, \Delta', F_3 \otimes F_4}\,\& - L$$

can be replaced by

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, F_1 \vdash \Delta, F_3\end{array}}{\Gamma, F_1 \,\&\, F_2 \vdash \Delta, F_3}\,\& - L \quad \begin{array}{c}\vdots\\ \Gamma' \vdash \Delta', F_4\end{array}}{\Gamma, \Gamma', F_1 \,\&\, F_2 \vdash \Delta, \Delta', F_3 \otimes F_4}\otimes - R$$

So, $\otimes$-R does indeed permute down over $\&$-L. Alternatively, $\&$-L permutes *up* over $\otimes$-R.

In the case where the two rules in question are both right or both left rules we need to modify the definition slightly. Where the upper rule operates on a formula which is introduced by the bottom rule permuting the rules is impossible. We modify the definition to exclude this situation when testing for permutability. For example in the following proof we cannot exchange the order of $\otimes - R$ and $\oplus - R$; however they do permute over each other when $\oplus$ is not in a sub-formula of $\otimes$.

$$\dfrac{\dfrac{}{F \vdash F} \quad \dfrac{\dfrac{}{H \vdash H}}{H \vdash G \oplus H}\oplus R}{F, H \vdash F \otimes (G \oplus H)}\otimes R$$

[5] By "above" we mean "closer to the leaves"

As a negative example consider $\otimes$-R and $\&$-R. The following proof is derivable

$$\dfrac{\dfrac{\dfrac{}{G \vdash \top}\top \quad \dfrac{\dfrac{}{F \vdash F}Ax}{F \vdash F, \bot}\bot}{F, G \vdash F, \bot \otimes \top}\otimes - R \quad \dfrac{\dfrac{}{F \vdash \top}\top \quad \dfrac{\dfrac{}{G \vdash G}Ax}{G \vdash G, \bot}\bot}{F, G \vdash G, \bot \otimes \top}\otimes - R}{\begin{array}{c} F, G \vdash F \,\&\, G, \bot \otimes \top \\ \vdots \end{array}}\& - R$$

Consider now a proof of the same sequent which begins with an application of $\otimes$-R. It is clear that the formula $F \,\&\, G$ will have to go to the branch containing $\bot$ since none of $\vdash \bot$, $F \vdash \bot$, $G \vdash \bot$ or $F, G \vdash \bot$ are provable. Unfortunately, neither are any of $\vdash F \,\&\, G, \bot$ or $F \vdash F \,\&\, G, \bot$ or $F, G \vdash F \,\&\, G, \bot$. Thus there is no proof of the sequent which begins by applying $\otimes$-R and hence $\otimes$-R does not permute down over $\&$-R.

A rule is *reversible* (the term *invertible* is also sometimes used) if it permutes down over all other rules and *relatively reversible* if it permutes down over all the rules that can occur in the fragment being considered, but not necessarily over all of the rules. For example, in linear logic the right rule for $\parr$ is reversible. The right rule for $\oplus$ is relatively reversible if the proof cannot contain $\&$-R (or the equivalent $\oplus$-L).

A connective is *asynchronous* [6] if the corresponding right rule is reversible and is *relatively asynchronous* if the corresponding right rule is relatively reversible. Non asynchronous connectives are referred to as *synchronous*. A connective occurring on the left of the turnstile is synchronous (respectively, asynchronous) iff it is asynchronous (respectively, synchronous) on the right of the turnstile.

If there is a proof of a sequent containing a formula whose topmost connective is asynchronous then there is a proof of the sequent where that formula is the active formula.

As an example, consider the proof (on the left) of $(a(1)^{\perp} \parr \bot), \exists x a(x)$. Since $\parr$ is asynchronous there exists a proof (on the right) where the first inference applied is $\parr$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a(1)^{\perp}, a(1)}Ax}{\vdash a(1)^{\perp}, \bot, a(1)}\bot}{\vdash a(1)^{\perp} \parr \bot, a(1)}\parr}{\vdash a(1)^{\perp} \parr \bot, \exists x a(x)}\exists \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a(1)^{\perp}, a(1)}Ax}{\vdash a(1)^{\perp}, \exists x a(x)}\exists}{\vdash a(1)^{\perp}, \bot, \exists x a(x)}\bot}{\vdash a(1)^{\perp} \parr \bot, \exists x a(x)}\parr$$

In linear logic, the connectives $\top$, $\bot$, $\parr$[6], $\&$, $?$ and $\forall$ are asynchronous and the connectives **1** , **0** , $\otimes$, $\oplus$, $!$ and $\exists$ are synchronous.

In [6] Andreoli observes that if there is a proof which begins by applying a synchronous rule and a side formula is itself synchronous then there exists a proof of the resulting sequent if and only if there is a proof which begins by applying a rule to the synchronous side formula. This property is known as *focusing*. Note that an atom on the left of the turnstile can be considered to be synchronous with respect to focusing.

The permutability properties for classical and intuitionistic logic below are from Kleene [83]. The permutability properties for linear logic are from Lincoln [94]. Most of the pairs of inference rules permute for classical and intuitionistic logic. The exceptions are outlined below.

Classical Logic

The only pairs of rules which do not commute involve $\exists$ and $\forall$.

1. $\forall - L$ above $\forall - R$: $\forall x p(x) \vdash \forall x p(x)$
2. $\forall - L$ above $\exists - L$: $\forall x p(x), \exists x \neg p(x) \vdash$
3. $\exists - R$ above $\exists - L$: $\exists x p(x) \vdash \exists x p(x)$
4. $\exists - R$ above $\forall - R$: $\vdash \exists x p(x), \forall x \neg p(x)$
5. $\exists - R$ above $\vee - L$: $p(a) \vee p(b) \vdash \exists x p(x)$
6. $\forall - L$ above $\vee - L$: $\forall \neg p(x), p(1) \vee p(2) \vdash$

These can be summarised in a table where a number means that the rule in that column cannot be permuted below the rule of the row. For example, in the following table, the occurrence of 5 indicates that $\exists - R$ cannot be permuted down past $\vee - L$.

[6] And $\multimap$

	$\forall - L$	$\exists - R$
$\vee - L$	6	5
$\forall - R$	1	4
$\exists - L$	2	3

Rules which do not appear as a column heading can always be permuted down. Rules which do not appear as row headings can always be permuted up. Rules which do not appear at all can be permuted arbitrarily.

Uniformity requires that right rules can be permuted down past left rules. Thus for the purposes of uniformity the problematic cases are those where a right rule above a left rule does not permute. For classical logic only the third and fifth cases above present a problem to uniformity. Intuitively, we can conclude that a logic programming language based on the application of uniformity to classical logic cannot allow the application of both $\exists - R$ and $\exists - L$ or of both $\exists - R$ and $\vee - L$.

Intuitionistic Logic

The following pairs of rules do not commute:

1. $\forall - L$ above $\forall - R$: $\forall x p(x) \vdash \forall x p(x)$
2. $\forall - L$ above $\exists - L$: $\forall x p(x), \exists x \neg p(x) \vdash$
3. $\exists - R$ above $\exists - L$: $\exists x p(x) \vdash \exists x p(x)$
4. $\rightarrow -L$ above $\rightarrow -R$: $p \rightarrow \neg p \vdash p \rightarrow q$
5. $\neg - L$ above $\rightarrow -R$: $\neg p \vdash p \rightarrow q$
6. $\rightarrow -L$ above $\neg - R$: $p \rightarrow \neg p \vdash \neg p$
7. $\neg - L$ above $\neg - R$: $\neg p \vdash \neg(p \wedge p)$
8. $\rightarrow -L$ above $\vee - L$: $p \vee q, p \rightarrow q \vdash q$

9. $\vee - R$ above $\vee - L : p \vee q \vdash p \vee q$
10. $\neg - L$ above $\vee - L : p \vee q, \neg p \vdash q$
11. $\exists - R$ above $\vee - L : p(a) \vee p(b) \vdash \exists x p(x)$
12. $\forall - L$ above $\vee - L : \forall \neg p(x), p(1) \vee p(2) \vdash$

These can be summarised in a table where a number means that the rule in that column cannot be permuted below the rule of the row. Note that since intuitionistic logic is single conclusioned it is not possible to have two consecutive right rules where the upper rule does not operate on a side formula of the lower rule. Hence all pairs of right rules are automatically not candidates for permutability and we mark them as N/A.

	$\vee - R$	$\forall - L$	$\exists - R$	$\rightarrow - L$	$\neg - L$
$\vee - L$	9	12	11	8	10
$\forall - R$	N/A	1	N/A		
$\exists - L$		2	3		
$\rightarrow - R$	N/A		N/A	4	5
$\neg - R$	N/A		N/A	6	7

Rules which do not appear as a column heading can always be permuted down. Rules which do not appear as row headings can always be permuted up. Rules which do not appear at all can be permuted arbitrarily.

Linear Logic

Linear logic has more impermutabilities than either classical or intuitionistic logic. In the following table (taken from Lincoln's thesis [94]) a number means that the right rule of the connective in that column cannot be permuted below the rule of the row. For example, the entry 1 indicates that $\otimes$ does not permute down past $\parr$:

$$\dfrac{\dfrac{\dfrac{}{\vdash A, A^{\perp}}\, Ax \quad \dfrac{}{\vdash B, B^{\perp}}\, Ax}{\vdash A, B, A^{\perp} \otimes B^{\perp}}\, \otimes}{\vdash A \parr B, A^{\perp} \otimes B^{\perp}}\, \parr \qquad \dfrac{\vdash A \parr B, A^{\perp} \quad \vdash B^{\perp}}{\vdash A \parr B, A^{\perp} \otimes B^{\perp}}\, \otimes$$

The table only summarises right rules (and omits **1** and $\top$ since they have no premises). To obtain the permutability properties for left rules use the de Morgan dual. For example, the left $\parr$ rule permutability properties are identical to the right $\otimes$ rule. Rules which do not appear as a column heading can always be permuted down. Rules which do not appear as row headings can always be permuted up. Rules which do not appear at all can be permuted arbitrarily. The !-R rule requires that there be no linear formulae and thus it does not permute with any rule not involving ? since the presence of a formula with a topmost connective other than ? prevents the ! rule from being applicable. For example, the sequent $\vdash (?p) \,\&\, (?q), !\mathbf{1}$ is provable but the ! rule is not applicable before the $\&$ rule.

	$\otimes$	$\oplus$	$?W$	?	!	$\exists$
$\otimes$					0	
$\parr$	1				0	
$\&$	2	3	4	5	0	6
$\oplus$					0	
$?C$	7					
!				8		
$\bot$					0	
$\forall$					0	9
$\exists$					0	

0 various examples
1 $\vdash (A \parr B), (A^{\bot} \otimes B^{\bot})$
2 $\vdash (A \,\&\, B), (\bot \otimes \top), A^{\bot}, B^{\bot}$
3 $\vdash (A \,\&\, B), (A^{\bot} \oplus B^{\bot})$
4 $\vdash (\mathbf{1} \,\&\, A), ?A^{\bot}$
5 $\vdash ((!A) \,\&\, A), ?A^{\bot}$
6 $\vdash (A(t)^{\bot} \,\&\, A(u)^{\bot}), \exists x.A(x)$
7 $\vdash ?A, (A^{\bot} \otimes A^{\bot})$
8 $\vdash !(A^{\bot} \oplus B), ?A$
9 $\vdash \forall y.(A(y) \oplus B), \exists x.A(x)^{\bot}$

2.5 Linear Logic Programming Languages

A linear logic programming language can be defined by giving a set of allowed program and goal formulae. Grammars defining the classes of goal and program formulae for a number of linear logic programming languages can be found in figure 2.8.

The semantics of a linear logic programming language can be obtained using the standard linear logic sequent calculus inference rules. However, a more deterministic set of rules is generally possible. For example, if a language requires that all program formulae are of the form $!\forall\overline{x}(\mathcal{G} \multimap A)$ then the following rule is a sound and complete replacement

Figure 2.8 Linear Logic Programming Languages

Lolli [70]

$$\mathcal{D} ::= \top \mid A \mid \mathcal{D} \,\&\, \mathcal{D} \mid \mathcal{G} \multimap \mathcal{D} \mid \mathcal{G} \Rightarrow \mathcal{D} \mid \forall x.\mathcal{D}$$

$$\mathcal{G} ::= \top \mid A \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D} \Rightarrow {}^{?}\mathcal{G} \mid \forall x.\mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathbf{1} \mid \mathcal{G} \otimes \mathcal{G} \mid !\mathcal{G} \mid \exists x\mathcal{G}$$

ACL [85]

$$\mathcal{D} ::= !\forall \overline{x}(\mathcal{G} \multimap A_p)$$

$$\mathcal{G} ::= \bot \mid \top \mid A_m \mid ?A_m \mid A_p \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \forall x\mathcal{G} \mid \mathcal{R}$$

$$\mathcal{R} ::= \exists \overline{x}(A_m^{\perp} \otimes \ldots \otimes A_m^{\perp} \otimes \mathcal{G}) \mid \mathcal{R} \oplus \mathcal{R}$$

LO [10]

$$\mathcal{D} ::= !\forall \overline{x}(\mathcal{G} \multimap A_1 \parr \ldots \parr A_n)$$

$$\mathcal{G} ::= A \mid \top \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \parr \mathcal{G}$$

Forum [109]

$$\mathcal{D} ::= \mathcal{G}$$

$$\mathcal{G} ::= A \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \multimap \mathcal{G} \mid \mathcal{G} \Rightarrow \mathcal{G} \mid \top \mid \bot \mid \forall x\mathcal{G}$$

$\mathcal{LC}$ [140]

$$\mathcal{D} ::= !\forall \overline{x}(\mathcal{G} \multimap A_1 \parr \ldots \parr A_n)$$

$$\mathcal{G} ::= A \mid \mathbf{1} \oplus \bot \mid \top \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \exists x\mathcal{G}$$

Lygon [122]

$$\mathcal{D} ::= A \mid \mathbf{1} \mid \bot \mid \mathcal{D} \,\&\, \mathcal{D} \mid \mathcal{D} \otimes \mathcal{D} \mid \mathcal{G} \multimap A \mid \mathcal{G}^{\perp} \mid \forall x\mathcal{D} \mid !\mathcal{D} \mid \mathcal{D} \parr \mathcal{D}$$

$$\mathcal{G} ::= A \mid \mathbf{1} \mid \bot \mid \top \mid \mathcal{G} \otimes \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D}^{\perp} \mid \forall x\mathcal{G} \mid \exists x\mathcal{G} \mid !G \mid ?G$$

for the axiom and left rules:

$$\frac{\Gamma, !\forall \overline{x}(G \multimap A') \vdash G[\overline{t}/\overline{x}], \Delta}{\Gamma, !\forall \overline{x}(G \multimap A') \vdash A, \Delta}\ Prog$$

where $A'[\overline{t}/\overline{x}] = A$.

A more detailed introduction to the individual languages, as well as a comparison between them, can be found in section 6.1. A more detailed introduction to Lygon as a programming language can be found in section 5.1. Note that the grammar given in figure 2.8 for Lygon is the one developed in [122]. In section 3.11 we derive a different version of the language (which we called Lygon$_2$) which we propose as a replacement for the earlier language design.

Chapter 3

Deriving Logic Programming Languages

This chapter is concerned with the derivation of logic programming languages from logics. The basic motivation is one of language design: given a logic, how should we go about creating a logic programming language based upon the logic? In investigating the question we need to take a close look at what logic programming is and how logic programming differs from, say, theorem proving, to which it is closely related.

In order to derive logic programming languages we need to have a notion of a *characteriser*. A characteriser is simply a test that tells us whether a set of permissible program and goal formulae constitutes a logic programming language. The derivation of logic programming languages can be reduced to the search for properties of proofs which can be used as characterisers and which capture the essence of logic programming. In this chapter we investigate a number of properties of proofs which capture various intuitions regarding the essence of logic programming.

Consider as an example logic programming in classical logic. It is well known that the Horn clause fragment of the logic forms a logic programming language (namely pure Prolog [134]). On the other hand, allowing arbitrary classical logic formulae as goals and programs does not result in a logic programming language since the resulting proof system lacks a number of desirable properties associated with logic programming. One such property is the ability of the system to return appropriate values for variables. When

we ask a goal such as `plus(1,3,X)` we expect the system to reply with `X=4` rather than with `yes`.

The ability of the system to do this relies on the property that a query of the form $\exists x F$ is provable if *and only if* there is a term t such that $F[t/x]$ is provable. This property holds for the Horn clause logic fragment but not for the full logic. For example, the goal $\exists x p(x)$ is provable from the program $p(a) \vee p(b)$; however, neither $p(a)$ nor $p(b)$ are provable from the program.

One property which seems essential to our intuition of what constitutes logic programming is *goal directedness*. Uniformity [112] is a formalisation of the intuitive notion of *goal-directedness*. The sequent $\Gamma \vdash \Delta$ is seen as comprising a program Γ and a goal Δ. A proof is goal directed if its "shape" – that is the choice of rule to be applied at each step in the derivation of a proof – is determined by the goal and not by the program.

DEFINITION 1 (UNIFORMITY [112])
A proof in the intuitionistic sequent calculus is uniform *if each occurrence of a sequent whose succedent contains a non-atomic formula is the conclusion of the inference rule which introduces its* [the non-atomic formula's] *top-level connective.*

For example, a proof of the sequent $\Gamma \vdash a \wedge c$ is non-uniform if it begins with a rule other than $\wedge - R$. The proof is uniform if it begins with $\wedge - R$ and if all sequents in the proof satisfy the uniformity condition. The first of the following proofs (in intuitionistic logic) is uniform, but the second is not:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{a \vdash a}\,W}{c, a \vdash a}\,W}{c, a, b \to a \vdash a}\,W \quad \dfrac{\dfrac{\dfrac{\overline{b \vdash b}\,W}{b \vdash b, a}\,W \quad \dfrac{\overline{a \vdash a}\,W}{b, a \vdash a}\,W}{c, b \vdash b, a}\,W \quad \dfrac{\overline{a \vdash a}\,W}{c, b, a \vdash a}\,W}{c, b, b \to a \vdash a}\,\to -L}{c, a \vee b, b \to a \vdash a}\,\vee - L \quad \dfrac{\dfrac{\overline{c \vdash c}\,W}{c, b \to a \vdash c}\,W}{c, a \vee b, b \to a \vdash c}\,W}{c, a \vee b, b \to a \vdash a \wedge c}\,\wedge - R$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{a \vdash a}\,W}{a, b \to a \vdash a}\,W}{c, a, b \to a \vdash a}\,W \quad \dfrac{\dfrac{\overline{c \vdash c}\,W}{c, b \to a \vdash c}\,W}{c, a, b \to a \vdash c}\,W}{c, a, b \to a \vdash a \wedge c}\,\wedge - R \quad \dfrac{\dfrac{\dfrac{\dfrac{\overline{b \vdash b}\,W}{b \vdash a, b}\,W \quad \dfrac{\overline{a \vdash a}\,W}{b, a \vdash a}\,W}{c, b \vdash a, b}\,W \quad \dfrac{\overline{a \vdash a}\,W}{c, b, a \vdash a}\,W}{c, b, b \to a \vdash a}\,\to -L \quad \dfrac{\dfrac{\overline{c \vdash c}\,W}{c, b \to a \vdash c}\,W}{c, b, b \to a \vdash c}\,W}{c, b, b \to a \vdash a \wedge c}\,\wedge - R}{c, a \vee b, b \to a \vdash a \wedge c}\,\vee - L$$

Miller et al. [112] define an idealised abstract interpreter $\vdash_O$ which corresponds to the operational viewpoint. This is linked to logic through the definition of a uniform proof. They then define an abstract logic programming language as a triple $< \mathcal{D}, \mathcal{G}, \vdash >$ such that for any P a finite subset of $\mathcal{D}$, and for any G in $\mathcal{G}$, $P \vdash G$ if and only if $P \longrightarrow G$ has a uniform proof.

Unfortunately uniformity is defined in the context of single conclusioned sequent calculi systems. This is a problem since we are interested in a general criteria applicable to a range of both single and multiple conclusion logics including relevant logic [24], temporal and modal logics [118] and linear logic [6, 10, 70, 85, 109, 140, 147].

This chapter continues the investigation into the derivation of logic programming languages. We define a number of formal characterisers (including uniformity) and explore the relationships amongst them. We begin with single conclusion logics and then generalise to the multiple conclusion setting.

Uniformity turns out to subsume most of the characterisers we investigate which is evidence that it is the right direction. We find however, that it is necessary to introduce a mechanism by which we can determine whether a proof is being guided by *atomic* goals. This mechanism is crucial to the proper generalisation of uniformity to multiple conclusion logics which yields two versions of uniformity. We propose these and an extended version of the single conclusion case as characterisers for the derivation of logic programming languages.

We begin by discussing the problems with uniformity and the pitfalls that arise in the use of logical equivalence (section 3.1). After discussing some intuitions about the essence of logic programming (section 3.2) we examine the single conclusion case (section 3.3) and look at some examples (section 3.4). We examine the multiple conclusion case (section 3.5) and look at some more examples (section 3.6). In particular, we look at Forum (section 3.7) and Lygon (section 3.8) and how they are judged by the various criteria presented. After applying our criteria to classical logic (section 3.9) we compare our work with other related work in section 3.10. We then re-derive the language Lygon in light of our results (section 3.11) and finish with a (single) conclusion and a brief look at further work.

Two items of terminology we shall use are *open nodes* and *proof steps*. Both of these relate to the process of proof construction. The process of proof search (as used in logic programming) begins with a single sequent at the root of the proof and progressively expands the proof upwards by applying inferences. The first inference applied has the root of the tree as its conclusion. At each step of the proof construction process there are a number of sequents which still need to be proved; these are *open nodes*. For example, in the following incomplete (intuitionistic logic) proof the sequents $p \vdash p \vee q$ and $p, p \vdash p$ are both open nodes.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\, Ax \quad p \vdash p \vee q}{p \vdash p \wedge (p \vee q)}\, \wedge -R}{\vdash p \to (p \wedge (p \vee q))}\, \to -R \quad \dfrac{\dfrac{p, p \vdash p}{p \wedge p \vdash p}\, \wedge -L}{\vdash (p \wedge p) \to p}\, \to -R}{\vdash (p \to (p \wedge (p \vee q))) \wedge ((p \wedge p) \to p)}\, \wedge -R$$

A *proof step* is the extension of an incomplete proof by the consecutive application of one or more inference rules. By "consecutive" we mean that the conclusion of an inference step is a premise of an earlier inference in the proof step. Intuitively a proof step extends a single open node with a number of inferences.

3.1 Problems with Uniformity

Uniformity is the main existing method for characterising logic programming languages. Unfortunately, it suffers from a number of problems.

Firstly, uniformity is only defined for single conclusion sequent systems[1]. A straightforward extension to a multiple conclusion setting is fairly obvious [55, 106, 109, 140] but it requires that all connectives occurring in goals permute over each other – a restriction which we shall see is unnecessary.

Another problem with a simple generalisation of uniformity to multiple conclusion sequent systems involves one sided presentations. Most[2] multiple conclusion presentations of a logic can be simply transformed into a one sided presentation. The problem

[1] From [112]: "A **C**-proof in which each sequent occurrence has a singleton set for its succedent is also called an **I**-proof". And later: "A *uniform proof* is an **I**-proof in which . . . "

[2] Sufficient de Morgan rules are required.

with one sided presentations is that they blur the distinction between programs and goals and thus cause problems with the application of uniformity. For example consider the one sided presentation derived by replacing the sequent $\Gamma \vdash \Delta$ with the sequent $\Gamma, (\neg\Delta) \vdash$ using de Morgan rules to push negation inwards to atoms and adding the following rule

$$\frac{}{p, \neg p \vdash} \; Ax'$$

Note that this rule is derivable:

$$\frac{\dfrac{}{p \vdash p}\, Ax}{p, \neg p \vdash} \neg - L$$

Since proofs in the given one-sided presentation do not contain any goals, all proofs are trivially uniform! As a direct consequence, the entire logic is trivially considered to be a logic programming language by uniformity.

Unless we are careful with transforming the presentation of the logic it becomes easy to conclude that uniformity considers the full logic of most multiple conclusion logics (including classical and linear) to be a logic programming language.

The conclusion that may be drawn from this is that only "sensible" presentations of logics should be considered. At a minimum, a sensible presentation requires that the notion of a goal and a program be present.

Note that having an empty program and a number of goals is a perfectly reasonable special case in a two sided presentation which we should expect a characteriser to handle.

This is a special case of the second problem with uniformity – it is reliant on a particular formulation of intuitionistic logic. This is a general problem – if we judge whether a fragment of a logic constitutes a logic programming language by looking at proofs and seeing if they have a certain property (such as uniformity) then changes in the system which is used to construct these proofs can effect the properties of the proofs. As a consequence, changing the presentation of a logic can change the class of formulae which are considered to be a logic programming language.

For example, in intuitionistic logic, program formulae of the form $F_1 \vee F_2$ are generally not permitted since the sequent $p \vee q \vdash p \vee q$ is provable but there is no proof which concludes with an application of the standard $\vee - R$ rule. However, if we replace the

standard $\vee - R$ rule with the rule:

$$\frac{\Gamma \vdash F_1, F_2, \Delta}{\Gamma \vdash F_1 \vee F_2, \Delta} \vee - R$$

and modify the rest of the calculus accordingly (see [41]) then the sequent has a uniform proof and programs can contain top-level disjunctions.

Thirdly, uniformity does not constrain the search for the proof of a sequent with (only) atomic goals. This is a problem in that a uniform proof search mechanism must apply the appropriate right rule while the goal is compound but can do almost anything once the goal becomes atomic. An example where this can lead to undesirable behaviour is in the proof of $q, \neg q \vdash \exists x p(x)$. This sequent is provable; however the proof weakens away $p(x)$ – so the goal is *irrelevant* to the proof. As we shall see this problem needs to be fixed in order for one of the two multiple conclusion generalisations of uniformity to work.

Finally, uniformity possesses a loophole. This allows designs to satisfy the letter of the rule but violate its spirit. We devote the rest of this section to this problem. It is worth stressing that this problem lies in the *application* of uniformity and not in uniformity itself.

Abusing Logical Equivalence

Recall that two formulae F and G are logically equivalent if each implies the other, that is, if we can prove both $F \vdash G$ and $G \vdash F$. Given that F and G are logically equivalent we can use the cut rule to replace F with G in a derivation and vice versa:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ F \vdash G & \Gamma \vdash \Delta, F\end{array}}{\begin{array}{c}\Gamma \vdash \Delta, G \\ \vdots\end{array}} Cut \qquad \frac{\begin{array}{cc}\vdots & \vdots \\ G \vdash F & \Gamma \vdash \Delta, G\end{array}}{\begin{array}{c}\Gamma \vdash \Delta, F \\ \vdots\end{array}} Cut$$

Since the cut rule is eliminable one concludes that F and G can be freely substituted for one another.

What has this to do with uniformity? The careless use of logical equivalence arguments allows for a loophole in the characterisation of logic programming languages. We

can (ab)use logical equivalence to design languages which are uniform in letter but which violate a number of properties generally associated with goal directed proofs and logic programming.

Note that, as an extreme case, the application of standard linear logical equivalences enables one to encode the entirety of linear logic into a subset that can be shown to be uniform [6, 109]. Since full linear logic is not uniform this suggests that the use of logical equivalences to extend a logic programming language might not always be appropriate. To see that full linear logic is not uniform consider the sequent $p \oplus q \vdash p \oplus q$. The sequent is provable; however the only possible proof uses a left rule when the goal is non-atomic and thus the proof is not uniform. Hence limiting ourselves to considering only uniform proofs is not complete for the *full* linear logic.

The argument to watch out for is *"these two formulae are logically equivalent and hence can replace each other"*. The point is that this only holds in the full logic. If the proof system we are working within is a limited version of the full one, then it may be incapable of proving that $F \vdash G$ and hence incapable of performing the replacement. Thus in the context of uniformity the argument above is *not* valid if we can not derive $F \vdash G$ in a *uniform* way. Indeed, the proof system may be able to differentiate between "logically equivalent" F and G.

For example, it is easy to show that the two formulae $p \,\&\, q$ and $(p^\perp \oplus q^\perp) \multimap \perp$ are logically equivalent. However the equivalence proof is not goal directed, and hence we cannot replace the first by the second in a goal directed proof. Given the program $p \,\&\, q$ only the first has a proof which we would intuitively consider to be "goal directed":

$$\dfrac{\dfrac{\dfrac{}{p \vdash p}\,Ax}{p \,\&\, q \vdash p}\,\&-L \qquad \dfrac{\dfrac{}{q \vdash q}\,Ax}{p \,\&\, q \vdash q}\,\&-L}{p \,\&\, q \vdash p \,\&\, q}\,\&-R$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\,Ax}{p \,\&\, q \vdash p}\,\&-L}{p \,\&\, q, p^\perp \vdash}\,-^\perp-L \qquad \dfrac{\dfrac{\dfrac{}{q \vdash q}\,Ax}{p \,\&\, q \vdash q}\,\&-L}{p \,\&\, q, q^\perp \vdash}\,-^\perp-L}{p \,\&\, q, p^\perp \oplus q^\perp \vdash}\,\oplus-L}{p \,\&\, q, p^\perp \oplus q^\perp \vdash \perp}\,\perp-R}{p \,\&\, q \vdash (p^\perp \oplus q^\perp) \multimap \perp}\,\multimap-R$$

The second proof is uniform, but it is clearly less "goal directed" than the first. This indicates that uniformity is not constraining enough. Another, more compelling, example

is the logical equivalence of $\exists x p(x)$ and $(\forall x(p(x)^\perp))^\perp$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p(y) \vdash p(y)}\, Ax}{p(y), p(y)^\perp \vdash}\, -^\perp - L}{p(y), \forall x(p(x)^\perp) \vdash}\, \forall - L}{\exists x p(x), \forall x(p(x)^\perp) \vdash}\, \exists - L}{\exists x p(x) \vdash (\forall x(p(x)^\perp))^\perp}\, -^\perp - R \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p(y) \vdash p(y)}\, Ax}{\vdash p(y)^\perp, p(y)}\, -^\perp - R}{\vdash p(y)^\perp, \exists x p(x)}\, \exists - R}{\vdash \forall x(p(x)^\perp), \exists x p(x)}\, \forall - R}{(\forall x(p(x)^\perp))^\perp \vdash \exists x p(x)}\, -^\perp - L$$

The second proof is *not* uniform. Moreover there does not exist a uniform proof of the sequent. Attempts to use this logical equivalence in a programming language can cause problems.

For example, consider the *Forum* [106, 109] fragment of linear logic. As will be discussed in section 3.7, Forum was designed using a multiple conclusion extension of Uniformity. However, for a number of reasons the language is viewed more as a specification language and less as a logic programming language. Forum makes extensive use of logical equivalence to encode the whole of linear logic into a fragment of the logic which can be argued to be uniform. For example, in Forum there is no $\exists$ connective. It is defined using the equivalence

$$\exists x F \quad = \quad (\forall x F^\perp)^\perp$$

Unfortunately, there *is* a difference between these two formulae. If we are told that the proof system is complete for uniform proofs then we can deduce that $\Gamma \vdash \exists x F$ is provable if and only if $\Gamma \vdash F[t/x]$ is provable for some term t. By encoding the existential quantifier as a doubly negated universal quantifier this property is lost.

For example, consider the following (legal) Forum program

$$((p(a) \multimap \perp) \,\&\, (p(b) \multimap \perp)) \multimap \perp$$

which we shall denote by P. The user now asks whether $\exists x p(x)$ holds. The system replaces the query with the logically equivalent[3] formula:

$$(\forall x(p(x) \multimap \perp)) \multimap \perp$$

[3] $F^\perp$ is encoded as $F \multimap \perp$ in Forum

The proof below is the only one (modulo trivial permutabilities). Note that no single value for x can be extracted from it.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bot \vdash}\bot - L \quad \dfrac{}{p(a) \vdash p(a)}Ax}{p(a), (p(a) \multimap \bot) \vdash}\multimap - L}{p(a), (\forall x(p(x) \multimap \bot)) \vdash}\forall - L}{p(a), (\forall x(p(x) \multimap \bot)) \vdash \bot}\bot}{\forall x(p(x) \multimap \bot) \vdash p(a) \multimap \bot}\multimap - R \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bot \vdash}\bot - L \quad \dfrac{}{p(b) \vdash p(b)}Ax}{p(b), (p(b) \multimap \bot) \vdash}\multimap - L}{p(b), (\forall x(p(x) \multimap \bot)) \vdash}\forall - L}{p(b), (\forall x(p(x) \multimap \bot)) \vdash \bot}\bot}{\forall x(p(x) \multimap \bot) \vdash p(b) \multimap \bot}\multimap - R}{\forall x(p(x) \multimap \bot) \vdash (p(a) \multimap \bot) \,\&\, (p(b) \multimap \bot)}\,\&}{P, (\forall x(p(x) \multimap \bot)) \vdash}\multimap - L}{P, (\forall x(p(x) \multimap \bot)) \vdash \bot}\bot}{P \vdash (\forall x(p(x) \multimap \bot)) \multimap \bot}\multimap - R$$

Although this proof may appear complex, it actually represents the Forum encoding of the simpler (and non-uniform!) proof:

$$\dfrac{\dfrac{\dfrac{}{p(a) \vdash p(a)}Ax}{p(a) \vdash \exists x p(x)}\exists \quad \dfrac{\dfrac{}{p(b) \vdash p(b)}Ax}{p(b) \vdash \exists x p(x)}\exists}{p(a) \oplus p(b) \vdash \exists x p(x)}\oplus - L$$

The Forum implementation[4] actually loops when given the above program and goal.

In summary, it may be invalid to apply logical equivalences to a logic programming language. If the equivalence can not be proven by the limited notion of proof used by the language then the application of the equivalence is likely to violate desirable properties of the language.

More generally, when a characteriser is fed with a set of valid goals ($\mathcal{G}$) and a set of valid program formulae ($\mathcal{D}$) for a logic and answers "yes, this is a logic programming language", then $\mathcal{G}$ and $\mathcal{D}$ can be considered to be a logic programming language. However, the use of logical equivalences to extend, say, $\mathcal{G}$ to $\mathcal{G}'$ is not necessarily valid. That $\mathcal{G}'$ can be pre-processed into $\mathcal{G}$ using logical equivalences does not mean that $\mathcal{G}'$ is a logic programming language. As an analogy, we do not consider `C` to be a logic programming language even though logic programming languages can be transformed into it. The intuition here is that when debugging and programming in `C` we need to think at *its* level,

[4] `ftp://ftp.cs.hmc.edu/pub/hodas/Forum/forum.tar.Z`

we cannot think at the higher level since the behaviour of the program can violate the high level semantics.

The essential point is that the level at which the programmer visualises what the program is doing is important. An encoding is valid if the programmer can work completely at the original un-encoded level. We shall return to this issue in section 3.12.

Thus, to summarise this section, uniformity suffers from a number of problems:

- It is defined for single conclusion systems. Furthermore, the obvious extensions to multiple conclusioned systems suffer from problems.
- Uniformity is reliant on a particular presentation of the logic – changes to the presentation can affect the resulting logic programming language.
- It does not constrain the proof of sequents with atomic goals.
- It allows for the abuse of logical equivalence.

3.2 What Is Logic Programming?

In our search for alternatives to uniformity as a characteriser of logic programming languages our ultimate goal is a formal characteriser appropriate for a multiple conclusion setting.

Before we can begin working with formal definitions however, we need to have an *informal* notion of what constitutes a logic programming language. What is the essential difference (or differences) between a logic programming language implementation and a theorem prover?

We feel that one of the key differences relates to the notion of *auditing*. When a program fails to work it is a problem to be resolved by the programmer (not the language implementor). In order to debug the program the programmer needs to be able to *audit* the execution, that is to be able to ask "how was this derived?" and get a useful answer. With a theorem prover there is no guarantee that the system will be able to explain to a user how the answer was derived. On the other hand one of the defining characteristics

of a logic programming language is that there exists a simple explanation of the proof search process.

Note that the audit need not be too detailed. Details can be left out where they are irrelevant. Details are irrelevant when we can trust the system to never fail. For example, consider register allocation in high level languages. and garbage collection. In both cases we have enough faith in the technology to leave it entirely up to the implementation.

We now proceed to examine several informal notions of what makes a logic programming language. These notions will be formalised in later sections. There are a number of properties that are generally associated with logic programming:

1. **Active goals, passive programs**

   The first basic intuition we have is that there is a fundamental difference between a program and a goal. The goal is "active" whereas the program is "passive" and provides a context within which the goal executes. A slightly stronger intuition is summarised by the slogan:

2. **Connectives as Instructions**

   This slogan suggests that we view the logical connectives as instructions that when executed result in the appropriate proof search steps being carried out. With this intuition we can think of a goal as a thread of control and a program as a set of procedures.

3. **Answer substitutions**

   In logic programming an important role is played by logical variables. When we ask a query such as `factorial(4,X)` we expect not only that the system say `yes` but also that it provide us with the binding `X = 24`. As we have seen this property is not generally satisfied by proof systems.

4. **The operational behaviour should be simple**

   This is related to the notion of auditing. The proof search process should be simple and understandable. Note that we do not insist that the *detailed* operational semantics be simple but rather that at *a suitable level of abstraction* the proof search process is simple. In many logic programming languages, features such as constraint

solving, co-routining and lazy resource allocation make the detailed proof search mechanism rather complex. In all of these cases however, there is an appropriate level of abstraction where the proof search mechanism becomes simple. We shall return to this issue in section 3.12.

5. **The language should be efficient**
   This is certainly desirable but is not a useful guideline for determining what is a logic programming language. The basic problem with using efficiency as a guideline is that certain inference rules may appear to be intractable but may actually have an efficient implementation. An obvious example is the use of logical variables and unification to delay the choice of a term in the $\exists - R$ and $\forall - L$ rules. Another example is the handling of the $\otimes - R$ rule in linear logic based programming languages [70, 150] (see also chapter 4). An additional problem with the use of efficiency as a characteriser is that the vast majority of logic programming languages and systems omit the occur check. This yields efficiency but costs soundness.

We formalise some of these intuitions in the simpler single conclusioned setting before proceeding to the multiple conclusioned setting. Section 3.3 covers the single conclusioned case and section 3.5 the multiple conclusioned case. In each of the two sections we will present a series of possible characterisers interspersed with discussion, examples and comments on the relationship between the characterisations.

3.3 The Single Conclusion Case

In this section we formalise the notion of what constitutes a logic programming language. In our presentation we strive to be general. We shall mostly assume only that the logic in question has a cut-elimination theorem[5].

A *criterion* is a formalisation of the (informal) notion of a characteriser. A criterion is (usually) a limitation on the structure of proofs. A subset of the logic is deemed a programming language by a given criterion if the application of the criterion preserves completeness, that is, if a proof search mechanism limited by the criterion can prove all

[5] "A logic without cut-elimination is like a car without engine" [47]

consequences for the subset of the logic. For example, the Horn clause subset of intuitionistic logic is complete under uniformity and is deemed to be a logic programming language by the uniformity criterion.

We write $\Gamma \vdash_x F$ to indicate that the sequent $\Gamma \vdash F$ is provable and can be proven in a way which satisfies criterion x. We view $\mathcal{D}$ and $\mathcal{G}$ as (generally infinite) sets of formulae in order to simplify the notation. Γ and Δ are finite throughout.

DEFINITION 2 (CRITERION)
A criterion *is a decision procedure which takes a sequent $\Gamma \vdash F$ and returns true or false. We write $\Gamma \vdash_x F$ to indicate that the sequent $\Gamma \vdash F$ is assigned true by criterion x. We require that criteria be sound, that is, if $\Gamma \vdash_x F$ holds then $\Gamma \vdash F$ must be provable.*

In general $\Gamma \vdash_x F$ is defined to hold if $\Gamma \vdash F$ holds and there is a proof which satisfies some additional constraints. This trivially ensures soundness of the criterion.

The next definition is based on the one in [112]. It provides the link between criteria and logic programming languages.

DEFINITION 3 (ABSTRACT LOGIC PROGRAMMING LANGUAGE (ALPL))
Let $\mathcal{D}$ be a set of legal program formulae, $\mathcal{G}$ a set of legal goal formulae and $\vdash$ some notion of provability. Then the triple $< \mathcal{D}, \mathcal{G}, \vdash >$ is deemed to be an abstract logic programming language *(ALPL) according to criterion x iff for any finite subset P of $\mathcal{D}$ and for any G in $\mathcal{G}$*

$$P \vdash G \iff P \vdash_x G$$

DEFINITION 4
Criterion x is stronger *than criterion y (written $x \rightsquigarrow y$) if we have that*

$$\forall \Gamma \ \forall F \quad \Gamma \vdash_x F \ \Rightarrow \ \Gamma \vdash_y F$$

Let $x \rightsquigarrow y$. If $< \mathcal{D}, \mathcal{G}, \vdash >$ is deemed to be an ALPL by criterion x then it is also deemed to be an ALPL by criterion y.

We write $x \not\rightsquigarrow y$ when x is not stronger than y.

LEMMA 1
$\rightsquigarrow$ is a partial order.
Proof: *$\rightsquigarrow$ is reflexive, transitive and antisymmetric (obvious).* ■

We now proceed to define the various criteria and to investigate the relationships between them. This is an exploratory process and (as we shall see) not all of the criteria defined are useful – some of the criteria (for example criterion A defined below) are insufficient to capture the essence of logic programming. However, for example, criterion A *is* necessary, and the fact that it is implied by criterion F (but not by criterion D_{strong}) lends weight to the argument that criterion F is a better test than uniformity.

Our main interest is to build up a global picture of the relationship between the various criteria (see figure 3.2) and so the detailed proofs of the (many) propositions are not of direct interest and we relegate them to appendix A. The definitions of the various criteria are summarised in figure 3.1 on page 50.

DEFINITION 5 (CRITERION A)
$\Gamma \vdash_A F$ *if there is a proof of* $\Gamma \vdash F$ *which does not use the contraction-right and weakening-right rules.*

Contraction right and weakening right are respectively the rules:

$$\frac{\Gamma \vdash F, F, \Delta}{\Gamma \vdash F, \Delta}\, C-R \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash F, \Delta}\, W-R$$

Their important property is that they are applicable to *any* formula F. Note that in a single conclusion setting $C-R$ is never applicable and $W-R$ has an empty Δ.

The intuition behind this criterion is that the goals (being in some sense threads of control) can not be freely cloned and deleted.

In the definition above, the phrase "if there is a proof" is important. A sequent may have a proof which violates the criterion in question. However, as long as there is also another proof which satisfies the criterion we retain completeness since a proof search process limited by the criterion will still be able to find a proof of the sequent.

When giving examples it is often more illuminating to look at classes of formulae which are *excluded* by the criterion. Consider the program $p, \neg p$. This program allows the goal q to be provable. This is not desirable in that the goal is irrelevant. The proof makes essential use of weakening-right and is excluded by criterion A. Thus criterion A does not deem the language

$$\mathcal{D} ::= A \mid \neg A \qquad \mathcal{G} ::= A$$

to be a logic programming language since the sequent $p, \neg p \vdash q$ can only be proved using $W-R$:

$$\dfrac{\dfrac{\dfrac{}{p \vdash p}\, Ax}{p, \neg p \vdash}\, \neg - L}{p, \neg p \vdash q}\, W - R$$

Criterion A assumes that the logic is similar enough to certain standard logics so that it possesses the standard structural rules. Linear logic, for example, does not have these structural rules and hence trivially satisfies this criterion.

DEFINITION 6 (CRITERION B)
$\Gamma \vdash_B F$ *if there exists a proof of* $\Gamma \vdash F$ *which does not contain a sub-proof of a sequent of the form* $\Xi \vdash$ *for some* Ξ.

This is based on the intuitive notion that the program is passive – a program without a goal should not be able to do anything.

An example of a language that does not satisfy criteria B is Forum where the program $p, (p \multimap \bot)$ is legal yielding the derivation:

$$\dfrac{\dfrac{}{\bot \vdash} \quad \dfrac{}{p \vdash p}}{p, p \multimap \bot \vdash} \multimap - L$$

If a linear logic programming language satisfies criterion B then whenever the proof search process finds itself searching for a proof of a sequent of the form $\Gamma \vdash \bot$ (or equivalently $\Gamma \vdash$) the search can be immediately terminated with failure.

PROPOSITION 2
$B \rightsquigarrow A$

PROPOSITION 3
$A \not\rightsquigarrow B$

DEFINITION 7 (CRITERION C)
$\Gamma \vdash_C F$ *if there exists a proof of* $\Gamma \vdash F$ *where all sequents of the form* $\Gamma \vdash \exists x F$ *are the conclusion of an application of the* $\exists$*-right rule.*

This criterion excludes systems where the notion of a unique answer substitution does not make sense. Specifically it excludes situations such as $p(a) \vee p(b) \vdash \exists x p(x)$ and $\exists x p(x) \vdash \exists x p(x)$.

Note that there actually exists a weaker criterion which only insists that a sequent of the form $\Gamma \vdash \exists x F$ is the conclusion of the $\exists$-right rule when it occurs at the root of the proof. Since criterion C is quite weak – it is implied by all variants of uniformity – weakening it further does not appear to be useful.

At first glance we might expect that $C \rightsquigarrow A$ since criterion C prohibits the weakening of formulae of the form $\exists x F$. However this is not sufficient since it is possible to apply the $\exists - R$ rule and then weaken the result.

PROPOSITION 4
$C \not\rightsquigarrow A$

PROPOSITION 5
$C \not\rightsquigarrow B$

PROPOSITION 6
$A \not\rightsquigarrow C$, $B \not\rightsquigarrow C$

The next criterion is uniformity, introduced in [112]. Uniformity restricts sequents with non-atomic goals to be the conclusions of right rules. There is some scope for variation though, as we have a choice as to *which* right rule. Even for a single conclusion system there is still a choice between a structural and a logical right rule.

The wording in [112] requires that a non-atomic goal be the conclusion of *the right rule which introduces its topmost connective* – i.e. the appropriate logical rule. Other papers (for example [70]) do not specify which right rule. Note that for a single conclusion setting with no structural rules (for example Lolli [69, 70]) these two definitions coincide.

Our definition of D_{weak} allows for *any* right rule to be used and D_{strong} requires that the right rule introduce the goal's topmost connective.

DEFINITION 8 (CRITERION D_{weak})
$\Gamma \vdash_{D_{weak}} F$ *if there exists a proof of* $\Gamma \vdash F$ *where all sequents of the form* $\Gamma \vdash G$ *(for non-atomic* G*) are the conclusion of some right rule.*

DEFINITION 9 (CRITERION D_{strong})

$\Gamma \vdash_{D_{strong}} F$ *if there exists a proof of* $\Gamma \vdash F$ *where all sequents of the form* $\Gamma \vdash G$ *(for non-atomic* G*) are the conclusion of the right rule which introduces the topmost connective in* F*.*

We use D to denote both D_{weak} and D_{strong}.

Examples of proofs that are excluded by criterion D but not by criterion C include

$$\dfrac{\dfrac{\overline{p \vdash p}\ Ax \quad \overline{q \vdash q}\ Ax}{p, q \vdash p \otimes q} \otimes - R}{p \otimes q \vdash p \otimes q} \otimes - L \qquad \dfrac{\dfrac{\overline{p \vdash p}\ Ax}{p \vdash ?p}\ ? - R}{?p \vdash ?p}\ ? - L$$

In both cases the only possible proof begins with a left rule.

Note that strictly speaking, criterion D_{strong} does not allow the rule

$$\dfrac{\Gamma \vdash}{\Gamma \vdash ?F}\ W? - R$$

since it does not add the desired connective to an existing formula. For intuitionistic linear logic the use of $?$ in goal formulae is of limited use since it only allows weakening and not contraction. Note that a similar effect can be encoded as $F \oplus \bot$.

PROPOSITION 7

$D_{strong} \rightsquigarrow D_{weak}$

PROPOSITION 8

$D_{weak} \rightsquigarrow C$

PROPOSITION 9

$D_{strong} \rightsquigarrow C$

PROPOSITION 10

$C \not\rightsquigarrow D$

PROPOSITION 11

$D_{weak} \not\rightsquigarrow D_{strong}$

PROPOSITION 12

$A \not\rightsquigarrow D$, $B \not\rightsquigarrow D$

We might expect that $D \rightsquigarrow A$; however as in the case for criterion C we can apply right rules and then weaken atoms.

PROPOSITION 13
$D \not\rightsquigarrow A$

PROPOSITION 14
$D \not\rightsquigarrow B$

3.3.1 Extending Uniformity to Deal with Atomic Goals

Neither version of uniformity places any limitation on the proof of sequents of the form $\Gamma \vdash A$ where A is atomic. This is a problem since an atom is just as much a thread of control as is a compound goal. It differs from the logical connectives in that where the semantics of $\wedge$ and $\exists$ are fixed by the logic, the semantics of p or *append* is defined by the program.

We argue that limiting in some way the search for proofs of sequents with atomic goals is desirable.

Firstly, note that criterion D is not stronger than criteria A and B. The reason is essentially that although the conditions of criterion D do imply criteria A and B these conditions are not applied when the goal is atomic and hence violations of criteria A and B can occur when the goal is atomic. As we shall see applying an appropriate constraint to the search for proofs of sequents with atomic goals yields a criterion which is stronger than criteria A and B.

Secondly criterion D allows some rather bizarre logic subsets as programming languages. Consider the language

$$\mathcal{D} ::= \mathcal{D} \wedge \mathcal{D} \mid \mathcal{D} \vee \mathcal{D} \mid \exists x \mathcal{D} \mid \forall x \mathcal{D} \mid A \mid \neg A \qquad \mathcal{G} ::= A$$

Clearly this language satisfies criterion D since the goal cannot be non-atomic. However it is hard to consider it to be a programming language since there is no limitation on the structure of the proof. Other evidence against it being considered a logic programming language is that goals lack variables and thus a notion of answer substitutions. Furthermore it fails criteria A and B.

There are a number of possible approaches to limiting the proof search process for sequents with atomic goals:

1. Simplicity
2. Static Clausal Form
3. Dynamic Clausal Form

The first, the notion of *simplicity*, was introduced in [107]. Simplicity restricts occurrences of the $\rightarrow -L$ rule to have axiomatic right hand premises. This is related to [93]. Note that this assumes that the logic has the rule:

$$\frac{\Gamma \vdash F \quad \Gamma, G \vdash H}{\Gamma, F \rightarrow G \vdash H} \rightarrow -L$$

The advantage of simplicity is that it is simple to reason about and to compare to other criteria. The disadvantage is that it is not general. The other two approaches use the notion of *clausal form*. We view a program as a set of clauses of the form $\mathcal{G} \rightarrow A$ with the single left rule:

$$\frac{\Gamma \vdash \mathcal{G}}{\Gamma \vdash A}\ Resolution \qquad \text{where } (\mathcal{G} \rightarrow A) \in \Gamma$$

This view can be used as a criterion in two different ways. *Static clausal form* insists that the program is given in clausal form. *Dynamic clausal form* insists that the logical presentation bundle all of the left rules into a single rule that is applied whenever the goal is atomic. This can be seen as deriving clauses at runtime. The single rule can be quite complex but it yields a system that, in conjunction with uniformity can truly be said to be goal directed — for *any* sequent the choice of which rule to apply is determined entirely by the goal.

Static clausal form is usable although it does tend to overly restrict the program. We shall consider dynamic clausal form since it generalises both simplicity and static clausal form.

A problem with dynamic clausal form is that formulating an equivalent logical system which combines the left rules into a single rule can involve considerable ingenuity (see

for example [122]) and as a result it is generally hard to show that it is *not* possible to combine the left rules into a single rule. Thus rather than modifying the proof system we retain the original system and require that a sequence of applications of left rules be able to be treated as an atomic entity. That applications of left rules are "bundle-able" implies that a single left rule can be devised (namely the one that does the appropriate bundling); however, the converse does not hold since even though bundling sequences of left rules may not be possible without a loss of completeness it may be possible in a different presentation of the logic.

We bundle up a sequence of left rule applications into a *left-focused proof step*. We then consider what languages are complete when left rules can only be applied as part of a left-focused proof step.

DEFINITION 10 (LEFT FOCUSING)
A proof of a sequent is left-focused *if one of the following hold:*

1. *It consists of a single application of the Ax rule.*

2. *It begins with a sequence of left rules resulting in a proof tree with open nodes $\Gamma_i \vdash F_i$:*

$$\frac{\Gamma_1 \vdash F_1 \quad \ldots \quad \Gamma_n \vdash F_n}{\begin{array}{c}\vdots \\ \Pi \\ \vdots \\ D, \Gamma \vdash A\end{array}\ \sharp - L}$$

   such that:

   (a) *The goal A is eliminated, that is, it is not any of the F_i.*

   (b) *Π consists only of left rules or Ax.*

   (c) *All of the principal formulae in Π are sub-formulae of D (or D itself).*

We have to take care with multiple copies of atoms. For example the proof

$$\frac{\Gamma \vdash p \quad \overline{p \vdash p}}{\Gamma, p \to p \vdash p} \to -L$$

has a single open node $\Gamma \vdash p$ where $F_1 = A = p$; however by labelling distinct occurrences of p we can see that the goal p_1 is actually eliminated:

$$\frac{\Gamma \vdash p_2 \quad \overline{p_3 \vdash p_1}}{\Gamma, p_2 \to p_3 \vdash p_1} \to -L$$

The intuition is that a left-focused proof of an atomic goal selects a program formula and reduces it entirely. In the process, the goal is satisfied and new goals are possibly created. For example the following proof fragment is left-focused:

$$\frac{\dfrac{\dfrac{r, \Gamma \vdash q \quad \overline{p \vdash p}\, Ax}{q \multimap p, r, \Gamma \vdash p} \multimap -L}{(q \multimap p) \otimes r, \Gamma \vdash p} \otimes -L}{s \,\&\, ((q \multimap p) \otimes r), \Gamma \vdash p} \,\&\, -L$$

This is related to the notion of resolution developed in [122].

DEFINITION 11 (CRITERION E)
$\Gamma \vdash_E F$ *if there exists a proof of* $\Gamma \vdash F$ *where all (sub)proofs of sequents with atomic goals are left-focused.*

As an example consider the language

$$\mathcal{D} ::= \mathcal{D} \vee \mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \mid \exists x \mathcal{D} \mid \forall x D \mid A \mid \neg A \qquad \mathcal{G} ::= A$$

As we have seen this language satisfies criterion D but it is rejected by criterion E since the sequent $p(1) \vee p(2), \forall x \neg p(x) \vdash q$ cannot be proved in an appropriate manner. The proof cannot begin with $\forall - L$ since there is no term t such that both $p(1) \vdash p(t)$ and $p(2) \vdash p(t)$ are provable. Hence the proof must begin with $\vee - L$:

$$\frac{\dfrac{\dfrac{\dfrac{\overline{p(1) \vdash p(1)}\, Ax}{p(1), \neg p(1) \vdash} \neg -L}{p(1), \neg p(1) \vdash q} W-R}{p(1), \forall x \neg p(x) \vdash q} \forall -L \qquad \frac{\dfrac{\dfrac{\overline{p(2) \vdash p(2)}\, Ax}{p(2), \neg p(2) \vdash} \neg -L}{p(2), \neg p(2) \vdash q} W-R}{p(2), \forall x \neg p(x) \vdash q} \forall -L}{p(1) \vee p(2), \forall x \neg p(x) \vdash q} \vee -L$$

Consider now whether this proof is left-focused. It does not begin with an axiom rule and hence must begin with a sequence of left rules. If we take this sequence of left rules

to be just the first rule ($\vee - L$) then we fail the first sub-condition of definition 10 since the goal (q) is not eliminated. If we take the sequence of left rules to be more than just the first rule then we fail the third sub-condition since principal formulae are from both program clauses and not from a single D.

Putting criteria D_{strong} and E together gives us criterion F:

DEFINITION 12 (CRITERION F)
$\Gamma \vdash_F F$ *if there is a proof of* $\Gamma \vdash F$ *where any sequent of the form* $\Delta \vdash A$ *(where* A *is atomic) is the conclusion of a left-focused (sub-)proof and any sequent of the form* $\Delta \vdash G$ *(*G *non-atomic) is the conclusion of the right rule which introduces the topmost connective in* G*.*

PROPOSITION 15
$F \rightsquigarrow D_{strong} \rightsquigarrow D_{weak}$

PROPOSITION 16
$D \not\rightsquigarrow E$

PROPOSITION 17
$F \rightsquigarrow C$

PROPOSITION 18
$C \not\rightsquigarrow F$

PROPOSITION 19
$A \not\rightsquigarrow F$

PROPOSITION 20
$F \rightsquigarrow A$

PROPOSITION 21
$F \not\rightsquigarrow B$

PROPOSITION 22
$B \not\rightsquigarrow F$

PROPOSITION 23

$E \not\rightarrowtail C$

COROLLARY:$E \not\rightarrowtail F$, $E \not\rightarrowtail D$ *(since* $F \rightsquigarrow C$ *and* $D \rightsquigarrow C$*)*

PROPOSITION 24

$E \not\rightarrowtail B$

PROPOSITION 25

$E \not\rightarrowtail A$

PROPOSITION 26

$B \not\rightarrowtail E$

COROLLARY:$A \not\rightarrowtail E$ *(since* $B \rightsquigarrow A$*)*

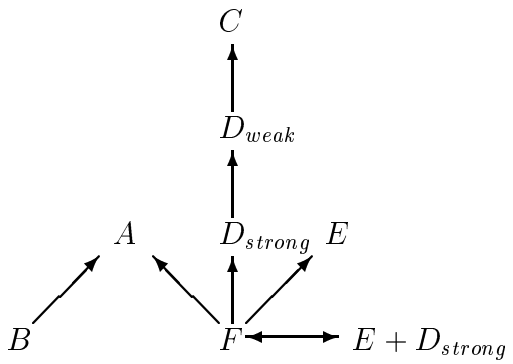
PROPOSITION 29

Let $\Gamma \vdash \Delta$ *be a sequent in a logic subset which satisfies criteria* E *and* D_{strong} *and such that* $\Gamma \vdash \Delta$ *is provable. Then there exists a proof of* $\Gamma \vdash \Delta$ *which satisfies criterion* F*.*

Figure 3.1 Single Conclusion Criteria Definitions

Criteria	Definition
A	No $W - R$
B	No sub-proofs of $\Gamma \vdash$
C	$\Gamma \vdash \exists F$ is conclusion of $\exists - R$ rule
D_{weak}	$\Gamma \vdash F$ (F non-atomic) are conclusion of a right rule
D_{strong}	$\Gamma \vdash F$ (F non-atomic) are conclusion of *the* right rule
E	Atomic goals are conclusion of left-focused proof step
F	$E + D_{strong}$

Figure 3.2 Relationships for Single Conclusion Criteria



3.4 Examples

We now consider a number of languages and relate them to the various criteria. Note that most of the languages in figure 2.8 are multiple conclusion and are covered in section 3.6. Both languages in this section have been shown (in [112] and [70] respectively) to satisfy uniformity. The contribution here is to show that they also satisfy criterion F. Since criterion F is stronger than uniformity this is a new result.

3.4.1 Pure Prolog

Prolog is often presented as a language based on *classical* logic which is multiple conclusion. Actually, the language does not make use of multiple conclusions and indeed, intuitionistic and classical logic agree where the Prolog subset is concerned. The Prolog syntax is given by the following BNF. Note that whenever the user's query is non-ground there is an implicit existential quantifier involved. The following grammar makes this explicit.

$$\mathcal{D} ::= [\forall \overline{x}](\mathcal{G}' \rightarrow A) \qquad \mathcal{G} ::= \exists \overline{x}(A_1 \wedge \ldots \wedge A_n)$$

Prolog satisfies all of the criteria.

PROPOSITION 30

Prolog satisfies criterion B

Proof: *Proof by induction. The only rules applicable to a sequent of the form $\Gamma \vdash$ are $\forall$-L, $\rightarrow$-L, W-L and C-L. It is easy to verify that whenever the conclusion of one of these rules has an empty goal so does at least one of its premises. Note that the axiom rule needs a non-empty goal. Hence no proof of a sequent with an empty goal can succeed.* ■

LEMMA 31

Let $\Gamma \subset \mathcal{D}$ and $F \in \mathcal{G}$. Then for any proof of $\Gamma \vdash F$ all sequents in the proof are of the form $\Gamma' \vdash F'$ such that $\Gamma' \subset \mathcal{D}$ and $F' \in \mathcal{G}$.

Proof: *Proof by induction. The only rules which can be applied in the proof are Ax, $\wedge$-R, $\exists$-R, $\forall$-L, $\rightarrow$-L, W-L and C-L. Note that the previous lemma precludes the use of W-R. It is easy to show that if the conclusion of one of these rules satisfies the condition then so do its premises.* ■

PROPOSITION 32

Prolog satisfies criterion D_{strong}

Proof: *To show that Prolog satisfies criteria D_{strong} we need to show that for non-atomic goals the appropriate right rule can be applied immediately without loss of completeness. That is that*

1. $\Gamma \vdash \exists x F$ *iff* $\Gamma \vdash F[t/x]$ *for some term* t.
2. $\Gamma \vdash F_1 \wedge F_2$ *iff* $\Gamma \vdash F_1$ *and* $\Gamma \vdash F_2$

Showing this involves a permutability *argument. From the permutability properties of classical logic (see section 2.4) it is evident that* $\wedge - R$ *permutes down past any rule and that* $\exists - R$ *permutes down past both* $\forall - L$ *and* $\rightarrow - L$*. Hence, if a sequent is provable then by permuting occurrences of right rules down we can obtain a proof where the two properties above are satisfied.* ■

COROLLARY:*Prolog satisfies criteria* D_{weak}*,* C *and* A*.*

DEFINITION 13 (DECOMPOSITION)

A sequence of left rules is said to decompose *a program formula* D *if their principal formulae are sub-formulae of* D *(or* D *itself) and no more left rules can be applied to sub-formulae of* D*.*

Intuitively decomposition involves selecting a program formula and applying left rules exclusively to it until one is left only with atoms.

PROPOSITION 33

Prolog satisfies criterion E

Proof: *Consider a proof of a sequent of the form* $D, \Gamma \vdash A$*. We know that* D *must have the form* $[\forall x](\mathcal{G} \rightarrow A)$*. Note that the only relevant left rules (*$\forall$*,* $\rightarrow$ *and* C*) permute with each other. Thus we can commit to decomposing a clause as an atomic proof step. This proof step looks like:*

$$\dfrac{\dfrac{\dfrac{\overline{A \vdash A} \quad \begin{array}{c}\vdots\\ \Gamma, \forall x\mathcal{G} \rightarrow A \vdash \mathcal{G}\end{array}}{\Gamma, \forall x\mathcal{G} \rightarrow A, \mathcal{G} \rightarrow A \vdash A} \rightarrow -L}{\Gamma, \forall x\mathcal{G} \rightarrow A, \forall x\mathcal{G} \rightarrow A \vdash A} \forall - L}{\Gamma, \forall x\mathcal{G} \rightarrow A \vdash A} C - L$$

Observe that the proof step is left-focused and that it succeeds if and only if the goal matches the head of the clause. ■

PROPOSITION 34

Prolog satisfies criterion F

Proof: *Prolog satisfies criteria* D_{strong} *and* E*. According to proposition 29 it therefore satisfies criterion* F*.* ■

3.4.2 Lolli

Lolli was introduced in [69]. For our purposes it can be viewed essentially as a single conclusion version of Lygon[6]. Lolli's syntax is given by the BNF:

$$\mathcal{D} ::= \top \mid A \mid \mathcal{D} \,\&\, \mathcal{D} \mid \mathcal{G} \multimap \mathcal{D} \mid \mathcal{G} \Rightarrow {}^{7}\mathcal{D} \mid \forall x.\mathcal{D}$$

$$\mathcal{G} ::= \top \mid A \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D} \Rightarrow \mathcal{G} \mid \forall x.\mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathbf{1} \mid \mathcal{G} \otimes \mathcal{G} \mid !\mathcal{G} \mid \exists x\mathcal{G}$$

PROPOSITION 35

Lolli satisfies criterion A

Proof: *Obvious since linear logic does not possess the weakening right rule.* ■

PROPOSITION 36

Lolli satisfies criterion B

Proof: *Proof by induction.*

Base case: *No sequent with an empty goal can succeed since the program cannot contain the constants* **1** *and* $\bot$ *and the* Ax *rule requires a non-empty goal.*

Induction: *If the conclusion of a left rule has an empty goal then so does at least one premise of the rule. Hence no sequent with an empty goal can be proven.* ■

PROPOSITION 37

Lolli satisfies criterion D_{strong}

Proof: *See [66]. The proof basically involves the use of permutability properties to transform a given proof into one satisfying D_{strong}.* ■

COROLLARY:*Lolli satisfies criteria D_{weak} and C.*

We now look at showing that Lolli satisfies criterion E. The steps involved are general and remain the same when showing that other languages satisfy criterion E.

1. Show that when the goal is atomic we can decompose a single program formula before considering other program formulae without losing completeness.

2. Show that if such a proof step is provable then there exists a proof which consumes the (atomic) goal.

[6] Conversely, Lygon could be viewed as a multiple conclusion generalisation of Lolli.

[7] $a \Rightarrow b \equiv (!a) \multimap b$

If these two requirements are met then when searching for a proof it is sufficient to work on a single program clause at a time and the resulting proof is guaranteed to be left-focused.

For example consider proving the sequent:

$$\forall x(p(x) \mathbin{\&} r), q \mathbin{\&} r, \forall x((q \otimes r) \multimap p(x)) \vdash p(a)$$

Since this sequent represents a valid Lolli program and goal we are guaranteed that it is provable if and only if there is a proof which manipulates program clauses one at a time and which is left-focused. One such proof begins by applying a left-focused proof step to $\forall x((q \otimes r) \multimap p(x))$ yielding the following:

$$\dfrac{\dfrac{\forall x(p(x) \mathbin{\&} r), q \mathbin{\&} r, \vdash q \otimes r \quad \overline{p(a) \vdash p(a)}}{\forall x(p(x) \mathbin{\&} r), q \mathbin{\&} r, ((q \otimes r) \multimap p(a)) \vdash p(a)} \multimap -L}{\forall x(p(x) \mathbin{\&} r), q \mathbin{\&} r, \forall x((q \otimes r) \multimap p(x)) \vdash p(a)} \forall -L$$

This is left-focused. Since the goal is compound we can apply a right rule:

$$\dfrac{q \mathbin{\&} r \vdash q \quad \forall x(p(x) \mathbin{\&} r) \vdash r}{\forall x(p(x) \mathbin{\&} r), q \mathbin{\&} r \vdash q \otimes r} \otimes -R$$
$$\vdots$$

We can then finish the proof by applying a left-focused proof step to each of the two open nodes:

$$\dfrac{\dfrac{\overline{q \vdash q}}{q \mathbin{\&} r \vdash q} \mathbin{\&} -L \quad \dfrac{\dfrac{\overline{r \vdash r}}{p(b) \mathbin{\&} r \vdash r} \mathbin{\&} -L}{\forall x(p(x) \mathbin{\&} r) \vdash r} \forall -L}{\forall x(p(x) \mathbin{\&} r), q \mathbin{\&} r \vdash q \otimes r} \otimes -R$$
$$\vdots$$

Note that the only place in the proof search process where there was any choice to be made as to which inference rule to apply (and to which formula) was at the very start when we could have selected to decompose an alternative program clause; in the rest of the proof search process the selection of the inference rule and principal formula was deterministic.

We now apply the procedure outlined to showing that Lolli satisfies criterion E.

LEMMA 38

Let Γ be a multiset of Lolli program formulae and let F be a Lolli goal formula such that $\Gamma \vdash F$ is provable. Then there is a proof of $\Gamma \vdash F$ where all left rules are part of a sequence which decomposes a single clause.

Proof: *Observe that all of the left inference rules which can be applied in a Lolli derivation are synchronous. Hence according to the focusing property [6] once we have applied a left rule to a program formula we can continue to decompose that formula without a loss of completeness. Furthermore, as observed in [6]:*

> *When a* negative *atom $A^{\perp}$ is reached at the end of a critical focusing section, the Identity must be used, so that A must be found in the rest of the sequent, either as a restricted resource . . . or as an unrestricted resource . . .*

Thus, once a program clause is decomposed to an atom on the left there is no loss of completeness in requiring that the next rule be the axiom rule. ■

We now need to show that if the decomposing proof step is provable then there exists a proof which consumes the (atomic) goal. In the following, when we say that *"the proof fails"* we mean that it can not be completed with an axiom rule. In general, it may be possible to complete the proof step using other program clauses. However, once we have chosen to decompose a given program clause, lemma 38 allows us to insist that the decomposing proof step terminate with an axiom rule where the program clause reduces to an atom. Hence we can ignore possible completions of the proof step which involve other program clauses without a loss of completeness.

We begin by defining a notion of an atom *matching* a clause. We then argue case-wise that:

(a) If the atom does not match the clause then a proof step focusing on the clause must fail.

(b) If the atom matches the clause then a proof step focusing on the clause either fails (see the note above) or eliminates the (atomic) goal.

In the following we let F be the selected program clause and A the (atomic) goal.

DEFINITION 14

A matches *the Lolli program clause* F *iff*

- F *is atomic is equal*[8] *to* A*.*
- $F = F_1 \,\&\, F_2$ *and* A *matches at least one of the* F_i
- $F = G \multimap F'$ *and* A *matches* F'
- $F = G \Rightarrow F'$ *and* A *matches* F'
- $F = \forall x F'$ *and* A *matches* F'

LEMMA 39

Consider a proof step which decomposes the Lolli program clause F *in the sequent* $\Delta, F \vdash A$ *where* A *is atomic. Then if* A *matches* F *then the proof step either eliminates* A *or fails; otherwise, if* F *does not match* A *then the proof step fails.*

Proof: *Induction on the structure of* F*.*

- F *is atomic: Without a loss of completeness (lemma 38) the sequent must be the conclusion of an axiom rule. If* F *matches* A *then we have* $\Gamma, A \vdash A$ *which either eliminates* A *or fails (if* Γ *contains linear formulae). If* F *does not match with* A *then the axiom rule cannot be applied and the proof fails.*
- $F = \top$*: There is no left rule so the proof fails.*
- $F = F_1 \,\&\, F_2$*: If* F *matches* A *then without loss of generality let* A *match* F_1 *and not match* F_2*. By the induction hypothesis the proof which uses the left* $\&$ *rule to select* F_1 *either eliminates* A *or fails. The proof which uses the left* $\&$ *rule to select* F_2 *fails. If* F *does not match* A *then* A *matches neither* F_1 *nor* F_2 *and by induction the premise of the* $\& - L$ *rule fails.*
- $F = G \multimap F'$*: The relevant rule is:*

$$\frac{\Gamma', F' \vdash A \quad \Gamma \vdash G}{\Gamma, \Gamma', G \multimap F' \vdash A} \multimap -L$$

[8] At the implementation level read as "is unifiable with A"

Figure 3.3 Summary of Single Conclusion Languages and Criteria

Language	A	B	C	D_{weak}	D_{strong}	E	F
Prolog	✓	✓	✓	✓	✓	✓	✓
Lolli	✓	✓	✓	✓	✓	✓	✓

Since A matches F iff it matches F', according to the induction hypothesis, the premise of the inference fails if A does not match F' and hence the conclusion fails. Likewise, if A matches F' then the sequent is provable only if the proof eliminates A and this property carries through to the conclusion of the inference.

The remaining two cases are analogous. ■

PROPOSITION 40

Lolli satisfies criterion E

Proof: *Let $\Gamma \vdash A$ be provable. The sequent must be the conclusion of either an axiom rule or a left rule. If it the conclusion of an axiom inference then we are done. Otherwise, the left rule is being applied to $F \in \Gamma$. According to lemma 38 there is a proof which focuses on F. According to lemma 39 this proof eliminates A.* ■

PROPOSITION 41

Lolli satisfies criterion F

Proof: *Lolli satisfies criteria E and D_{strong} and by proposition 29 it therefore satisfies criterion F.* ■

3.5 The Multiple Conclusioned Case

We now generalise our criteria to the multiple conclusion case. Most of the commentary carries across unchanged. As a result this section contains mostly definitions and theorems.

DEFINITION 15 (CRITERION)

A criterion *is a decision procedure which takes a sequent $\Gamma \vdash \Delta$ and returns true or false. We write $\Gamma \vdash_x \Delta$ to indicate that the sequent $\Gamma \vdash \Delta$ is assigned true by criterion x. We*

require that criteria be sound, that is, if $\Gamma \vdash_x \Delta$ *holds then* $\Gamma \vdash \Delta$ *must be provable.*

In general $\Gamma \vdash_x \Delta$ is defined to hold if $\Gamma \vdash \Delta$ holds and there is a proof which satisfies some additional constraints. This trivially ensures soundness of the criterion.

The next definition is based on the one in [112]. It provides the link between criteria and logic programming languages.

DEFINITION 16 (ABSTRACT LOGIC PROGRAMMING LANGUAGE (ALPL))
Let $\mathcal{D}$ *be a set of legal program formulae,* $\mathcal{G}$ *a set of legal goal formulae and* $\vdash$ *some notion of provability. Then the triple* $< \mathcal{D}, \mathcal{G}, \vdash >$ *is deemed to be an* abstract logic programming language *(ALPL) according to criterion* x *iff for any finite subset* P *of* $\mathcal{D}$ *and for any finite subset* G *of* $\mathcal{G}$

$$P \vdash G \iff P \vdash_x G$$

DEFINITION 17
Criterion x *is* stronger *than criterion* y *(written* $x \rightsquigarrow y$*) if we have that*

$$\forall\Gamma\ \forall\Delta \quad \Gamma \vdash_x \Delta \ \Rightarrow\ \Gamma \vdash_y \Delta$$

Let $x \rightsquigarrow y$. If $< \mathcal{D}, \mathcal{G}, \vdash >$ is deemed to be an ALPL by criterion x then it is also deemed to be an ALPL by criterion y.

We write $x \not\rightsquigarrow y$ when x is not stronger than y.

LEMMA 42
$\rightsquigarrow$ *is a partial order.*
Proof: $\rightsquigarrow$ *is reflexive (obvious), transitive (obvious) and antisymmetric (obvious).* ■

The definitions of the various criteria are summarised in figure 3.4 on page 68.

DEFINITION 18 (CRITERION A)
$\Gamma \vdash_A \Delta$ *if there is a proof of* $\Gamma \vdash \Delta$ *which does not use the contraction-right and weakening-right rules.*

As in the single conclusion case the intuition is that program threads do not spontaneously die or clone themselves. Interestingly this criterion is weaker here than in the

single conclusion setting. Consider the following proof

$$\dfrac{\dfrac{\dfrac{}{q \vdash q}\, Ax}{q, \neg q \vdash}\, \neg - L}{q, \neg q \vdash p}\, W - R$$

In a single conclusion setting the weakening step must take place before the $\neg q$ can be moved across the turnstile. On the other hand in a multiple conclusion system there is no requirement for there to be at most one conclusion and the following proof becomes possible

$$\dfrac{\dfrac{\dfrac{}{q \vdash q}\, Ax}{q \vdash q, p}\, W - R}{q, \neg q \vdash p}\, \neg - L$$

This suggests that for classical logic limiting occurrences of weakening-right is not as strong a criterion as it is for intuitionistic logic. Note that it is possible to eliminate weakening altogether by combining it with the axiom rule. This can be done by introducing the composite rule Ax':

$$\dfrac{}{\Gamma \vdash \Delta}\, Ax' \qquad \Gamma \cap \Delta \neq \emptyset$$

Thus, placing conditions on occurrences of weakening-right does not seem to limit the class of formulae that are judged to be logic programming languages since weakening can be eliminated. On the other hand, the absence of contraction-right *is* a limitation and thus criterion A is not trivial for classical logic.

DEFINITION 19 (CRITERION B)
$\Gamma \vdash_B \Delta$ *if there is a proof of* $\Gamma \vdash \Delta$ *which does not contain a sub-proof of a sequent of the form* $\Xi \vdash$ *for some* Ξ.

This is the same as the single conclusion version. As a result of the reduction in power of criterion A we have that for multiple conclusion logics criteria A and B are not related.

PROPOSITION 43
$B \not\rightarrow A$

PROPOSITION 44
$A \not\rightarrow B$

The next criterion captures the importance of answer substitutions. The single conclusion definition covers the case $\Gamma \vdash \exists x F$; the question remains however as to what requirement we place on sequents of the form $\Gamma \vdash \exists x F, \Delta$? There are two possibilities; we can either just retain the single conclusion definition and not impose any constraints on the case where there are multiple goal formulae or we can insist that the sequent is the conclusion of an $\exists$-R rule whenever it contains a formula whose topmost connective is $\exists$, *regardless* of what other formulae are present. This second possibility is the stronger of the two and corresponds to C_{strong}.

DEFINITION 20 (CRITERION C)
$\Gamma \vdash_C \Delta$ *if there exists a proof of* $\Gamma \vdash \Delta$ *where all sequents of the form* $\Gamma \vdash \exists x F$ *are the conclusion of an application of the* $\exists$*-right rule.*

Note that criterion C is weak in that it is only applicable to sequents where the conclusion contains the *single* goal $\exists x F$.

DEFINITION 21 (CRITERION C_{strong})
$\Gamma \vdash_{C_{strong}} \Delta$ *if there exists a proof of* $\Gamma \vdash \Delta$ *where all sequents of the form* $\Gamma \vdash \exists x F, \Delta$ *are the conclusion of an application of the* $\exists$*-right rule.*

PROPOSITION 45
$C_{strong} \rightsquigarrow C$

We might expect that $C \rightsquigarrow A$ since criterion C prohibits the weakening of formulae of the form $\exists x F$. However, as in the single conclusion case, this is not sufficient - one can always apply the $\exists - R$ rule and then weaken the result.

PROPOSITION 46
$C \not\rightsquigarrow A$, $C_{strong} \not\rightsquigarrow A$

PROPOSITION 47
$C \not\rightsquigarrow B$, $C_{strong} \not\rightsquigarrow B$

PROPOSITION 48
$A \not\rightsquigarrow C$, $B \not\rightsquigarrow C$, $A \not\rightsquigarrow C_{strong}$, $B \not\rightsquigarrow C_{strong}$

3.5.1 Generalising Uniformity to the Multiple Conclusion Case

Throughout the next section we shall use $\mathcal{A}$ to denote a multiset of atomic formulae and $\mathcal{C}$ to denote a multiset of compound (i.e. non-atomic) formulae.

In generalising uniformity to a multiple conclusion setting we have a choice to make. There are two possibilities:

(1) All sequents in the proof of the form $\Gamma \vdash \mathcal{C}$ are the conclusion of a right rule. That is, if the succedent consists *entirely* of compound goals then the sequent must be a conclusion of a right rule. If the succedent contains both atomic and compound goals then no restrictions are applied. This is analogous to criterion C.

(2) All sequents in the proof of the form $\Gamma \vdash \mathcal{C}, \mathcal{A}$ are the conclusion of a right rule. That is, if the succedent contains *any* compound goals then the sequent is the conclusion of a right rule. This is analogous to criterion C_{strong}.

In addition to requiring the proof to be guided by compound goals we can also require that the proof process be guided by atomic goals. There is a single mechanism for doing this and the choice we face is whether to apply it or not. Our choices are:

(a) All sequents are either the conclusion of a right rule or the conclusion of a left-focused proof step.

(b) There is no use of atomic goals to guide the proof search, so for example there is no restriction on the proof of sequents of the form $\Gamma \vdash \mathcal{A}$.

Since these two choices are independent we can consider various combinations. As we shall see some combinations do not make sense and some combinations coincide. There are four combinations of choices:

(1b) We choose to restrict ourselves to right rules only when there are no atomic goals (1) and atoms are not used to guide the proof (b). In this situation we are obtaining only partial guidance from compound goals. Since we do not use atoms to guide the proof there is insufficient guidance to be able to consider this choice goal directed. For instance there is no restriction whatsoever on the proof of sequents of the form $\Gamma \vdash \mathcal{A}, \mathcal{C}$.

(1a) By making use of atomic goals we obtain a characteriser that is useful. Note that it is essential that we have a notion of atomic goals directing the proof search process. We term this characteriser *synchronous uniformity*. This notion is similar to the notion of locally LR proof search which was used in the design of *Lygon* [122].

(2) By choosing to be guided by compound goals even if there are atomic goals we obtain sufficient guidance. The only place where the proof search process has no restrictions is in the proof of sequents of the form $\Gamma \vdash \mathcal{A}$. As in the single conclusion case we argue that making use of atomic goals to direct the proof search process is desirable. Note that the need to use atomic goals to guide the proof is not as vital here as it is for the previous case. We term this characteriser *asynchronous uniformity*. This is the notion that is used in (for example) the design of *Forum* [106].

3.5.2 Asynchronous Uniformity

Given that we can apply right rules whenever there is a compound goal formulae it is easy to show that we cannot have impermutabilities on the right. Consider two connectives whose right rules do not permute, say $\otimes$ and $\parr$, where $\parr$ may need to be done first. Consider now a sequent of the form

$$!((q^{\perp} \parr r) \multimap p) \vdash p, q \otimes r^{\perp}$$

This sequent is provable, but the proof violates synchronous uniformity since we need to replace the p with $q^{\perp} \parr r$ *before* we can apply the $\otimes$ rule; that is, although the goal contains a non-atomic formula we have to apply left rules before we can apply a right rule. Thus asynchronous uniformity implies that the right rules permute over each other[9]. Note that this is independent of whether atomic goals are used to guide the proof.

In order to define this formally (as criterion D_A) we shall need a notion of left-focusing for multiple conclusion logics.

[9] This assumes that program formulae are permitted to have the form $\mathcal{D} ::= !(\mathcal{G} \multimap A)$. This assumption holds for all languages considered in this thesis.

DEFINITION 22 (LEFT FOCUSING)
A proof of a sequent is left-focused *if one of the following hold:*

1. *It consists of a single application of the Ax rule.*

2. *It begins with a sequence of left rules resulting in a proof tree with open nodes $\Gamma_i \vdash \Delta_i$:*

$$\dfrac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\begin{array}{c}\vdots \\ \Pi \\ \vdots \\ D, \Gamma \vdash \mathcal{A}, \Delta\end{array}} \sharp - L$$

   such that:

   (a) *The goals $\mathcal{A}$ are eliminated, that is, they are not in any of the Δ_i.*

   (b) *Π consists only of left rules or Ax.*

   (c) *All of the active formulae in Π are sub-formulae of D (or D itself).*

As in the single conclusion case we have to take care with multiple copies of atoms. For example the proof

$$\dfrac{\Gamma \vdash p \quad \overline{p \vdash p}}{\Gamma, !(p \multimap p) \vdash p} \rightarrow -L$$

has $F_1 = A = p$. However by labelling distinct occurrences of p we can see that the goal p_1 is actually eliminated:

$$\dfrac{\Gamma \vdash p_2 \quad \overline{p_3 \vdash p_1}}{\Gamma, !(p_2 \multimap p_3) \vdash p_1} \rightarrow -L$$

The intuition is that a left-focused proof of an atomic goal selects a program formula and reduces it entirely. In the process, the goal is satisfied and new goals are possibly created.

Note that for Forum there is a minor technicality – $\mathcal{A}$ could be empty. This only occurs with $\perp$ in clauses. Such clauses can be resolved against at any time. It is possible to insist in the definition of left-focusing that $\mathcal{A}$ be non-empty. If this is done then

Forum$^-$ does not satisfy criterion D_A since the sequent $\top \multimap \bot \vdash p$ is provable but the only proof possible begins with a left-focused proof step which does not eliminate p:

$$\dfrac{\dfrac{}{\bot \vdash} \quad \dfrac{}{\vdash \top, p}}{\top \multimap \bot \vdash p} \multimap -L$$

Note that the use of criterion D_S precludes clauses of the form $\mathcal{G} \multimap \bot$ even without limiting $\mathcal{A}$ to be non-empty (although we need to assume that the language in question is sufficiently rich). Assuming that $\top$ and $p \otimes q$ are both valid goals we have that the sequent $\top \multimap \bot \vdash p \otimes q$ is provable (the left proof below). However, since the goal consists of a single compound formula criteria D_S requires that there exists a proof which begins with $\otimes - R$ and as the right proof below demonstrates no such proof exists.

$$\dfrac{\dfrac{}{\bot \vdash} \bot - L \quad \dfrac{}{\vdash \top, p \otimes q} \top - R}{\top \multimap \bot \vdash p \otimes q} \multimap -L \qquad \dfrac{\dfrac{\dfrac{}{\bot \vdash} \quad \dfrac{}{\vdash \top, p}}{\top \multimap \bot \vdash p} \multimap -L \quad \vdash q}{\top \multimap \bot \vdash p \otimes q} \otimes - R$$

The example used does not apply to D_A since it makes use of the goal formula $p \otimes q$ which is not asynchronous and which would not be permitted to occur in a language derived using D_A. The assumption of "sufficiently rich" needed above includes the presence of synchronous goal connectives which prevent the language from being accepted as a logic programming language by criterion D_A.

DEFINITION 23 (CRITERION D_A)
$\Gamma \vdash_{D_A} \Delta$ *if there exists a proof of* $\Gamma \vdash \Delta$ *such that*

- *For any sequent of the form* $\Gamma \vdash \Delta, \mathcal{C}$ *and for any (compound) formulae* $F \in \mathcal{C}$ *there is a proof where the sequent is the conclusion of the right rule which introduces the topmost connective in* F*.*
- *The proof of any sequent of the form* $\Gamma \vdash \mathcal{A}$ *is left-focused.*

3.5.3 Synchronous Uniformity

Synchronous uniformity relaxes the strong requirement that compound goals direct the proof search process whenever they are present. Doing this is necessary in order to allow impermutabilities on the right to exist within a language. To be able to obtain less

guidance from compound goals we need to compensate by obtaining more guidance from atomic goals.

Note that for a sequent of the form $\Gamma \vdash \mathcal{A}, \mathcal{C}$ we *allow* the proof to be guided by an atom even though there are compound goals present. We cannot *require* that such sequents be guided by atoms in all cases (thus giving priority to resolution over decomposition) since in some cases decomposing compound goals will need to be done before resolution. A simple example is the proof of the sequent

$$p \parr (q \parr r) \vdash p \parr q, r$$

where we need to decompose the formula $p \parr q$ so that the p is available to the resolution of r. Another example is the proof of the sequent[10]

$$(r \parr p) \,\&\, (r \parr q) \vdash p \,\&\, q, r$$

where $p \,\&\, q$ needs to be done first so that we can choose different program clauses for the two sub-proofs:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{r \vdash r} \quad \dfrac{}{p \vdash p}}{r \parr p \vdash p, r}\parr - L}{(r \parr p) \,\&\, (r \parr q) \vdash p, r}\,\& - L \qquad \dfrac{\dfrac{\dfrac{}{r \vdash r} \quad \dfrac{}{q \vdash q}}{r \parr q \vdash q, r}\parr - L}{(r \parr p) \,\&\, (r \parr q) \vdash q, r}\,\& - L}{(r \parr p) \,\&\, (r \parr q) \vdash p \,\&\, q, r}\,\& - R$$

DEFINITION 24 (CRITERION D_S)
$\Gamma \vdash_{D_S} \Delta$ *if there exists a proof of* $\Gamma \vdash \Delta$ *such that*

- *Any sequent of the form* $\Gamma \vdash \mathcal{C}$ *is the conclusion of a right rule which introduces the topmost connective of a formula* $F \in \mathcal{C}$.

- *Any sequent in the proof is either the conclusion of a right rule which introduces the topmost connective of a formula in the goal or is the conclusion of a left-focused (sub-)proof.*

We shall use D to refer to either of D_A and D_S.

[10] Note that this could also be written as: $(p^{\perp} \multimap r) \,\&\, (q^{\perp} \multimap r) \vdash p \,\&\, q, r$

Both D_A and D_S require that a right rule introduces the topmost connective of a formula. Strictly speaking, this means that the following rules are not acceptable in a proof:

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash ?F, \Delta}\ W? - R \qquad \frac{\Gamma \vdash ?F, ?F, \Delta}{\Gamma \vdash ?F, \Delta}\ C? - R$$

There are two solutions

1. View weakening and contraction of goals as undesirable even if they are limited to certain goals (i.e. it's a feature, not a bug)

2. Relax the definition to require that the right rule be specific to the top level connective - this rules out weakening and contraction.

Of the languages considered in figure 2.8, the difference only affects ACL and it can be argued that the language still satisfies criterion D_S even though it allows formulae of the form $?A_m$. The reason for this is that the formulae which can be replicated by contraction are of a very limited form – they must be "message predicates" – and the contraction can be delayed until a "message receive" ($A_m \otimes G$) is executed.

Note that the current Lygon implementation does not allow goals of the form $?F$ since they do cause a problem. Specifically, it becomes hard to determine that a sequent of the form $\Gamma \vdash G, ?F$ has no proof since the proof search space is infinite.

In general an application of right contraction cannot be delayed indefinitely since the formula may be needed. In the proof of the sequent

$$!(\mathbf{1} \multimap (r \parr q)), !(q \multimap (p \parr p)) \vdash r, ?p$$

we need to begin by applying $C? - R$ twice so that we can resolve against the second clause.

Finally, note that it is possible to avoid the use of $?$ in goals by replacing $?F$ with a new constant p and adding the program clause $!(((F \parr p) \oplus \bot) \multimap p)$. Similar behavior to $?$ can be captured as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash F, p, \Delta}}{\Gamma \vdash F \parr p, \Delta}}{\dfrac{\Gamma \vdash (F \parr p) \oplus \bot, \Delta}{\Gamma \vdash p, \Delta}} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \Delta}}{\Gamma \vdash \bot, \Delta}}{\dfrac{\Gamma \vdash (F \parr p) \oplus \bot, \Delta}{\Gamma \vdash p, \Delta}}$$

For example, the sequent above could be written as

$$!(\mathbf{1} \multimap (r \parr q)), !(q \multimap (p \otimes p)), !(((p \otimes p') \oplus \bot) \multimap p') \vdash r, p'$$

This technique is not perfect since the right ! rule behaves differently but it does cover common logic programming uses of ? (as opposed to theorem proving applications).

PROPOSITION 49
$B \not\rightsquigarrow D$, $A \not\rightsquigarrow D$

PROPOSITION 50
$D_A \rightsquigarrow A$

PROPOSITION 51
$D_S \rightsquigarrow A$

PROPOSITION 52
$D \rightsquigarrow C$

PROPOSITION 53
$D_A \rightsquigarrow C_{strong}$

PROPOSITION 54
$D_S \not\rightsquigarrow C_{strong}$

PROPOSITION 55
$D \not\rightsquigarrow B$

PROPOSITION 56
$C \not\rightsquigarrow D$, $C_{strong} \not\rightsquigarrow D$

PROPOSITION 57
$D_S \not\rightsquigarrow D_A$

PROPOSITION 58
$D_A \rightsquigarrow D_S$

Figure 3.4 Multiple Conclusioned Criteria Definitions

Criteria	Definition
A	No $C - R$ or $W - R$
B	No sub-proofs of $\Gamma \vdash$
C	$\Gamma \vdash \exists F$ is conclusion of $\exists - R$
C_{strong}	$\Gamma \vdash \exists F, \Delta$ is conclusion of $\exists - R$
D_A	$\Gamma \vdash \Delta, \mathcal{C}$ is the conclusion of a right rule
	$\Gamma \vdash \mathcal{A}$ is left-focused
D_S	$\Gamma \vdash \mathcal{C}$ is the conclusion of a right rule
	$\Gamma \vdash \Delta, \mathcal{A}$ is either left-focused or the conclusion of a right rule

The previous proposition implies that any language that satisfies criterion D_A also satisfies criterion D_S. However it does not tell us anything about the properties of languages which satisfy D_S but not D_A.

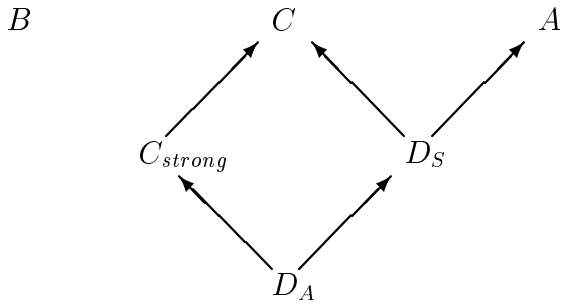
Note that for sequents of the form $\Gamma \vdash \mathcal{C}$ criterion D_S requires that the sequent be the conclusion of a right rule which introduces the topmost connective of *some* formula $F \in \mathcal{C}$ whereas D_A requires that for *every* formula $F \in \mathcal{C}$ there exist a proof which begins by introducing that formula's top level connective.

It appears feasible to consider a criterion lying between D_A and D_S that modifies D_S by requiring that sequents of the form $\Gamma \vdash \mathcal{C}$ have a proof which introduces the topmost connective of F for *all* $F \in \mathcal{C}$. This differs from criterion D_A in that there is no requirement that sequents of the form $\Gamma \vdash \mathcal{C}, \mathcal{A}$ be the conclusion of a right rule.

PROPOSITION 59

Let D_{S+} be a criterion that modifies D_S by requiring that sequents of the form $\Gamma \vdash \mathcal{C}$ have a proof which introduces the topmost connective of F for all *$F \in \mathcal{C}$. Then D_{S+} is equivalent to D_A.*

Figure 3.5 Relationships for Multiple Conclusion Criteria



3.6 Examples

Before we proceed to consider specific example languages, let us establish a few more results.

LEMMA 60

Let $\Gamma, A \vdash \Delta$ *have a linear logic proof. Then there is an incomplete proof of* $\Gamma', \Gamma \vdash \Delta, \Delta'$ *which has an identical structure and where some of the sequents of the form*

$$\frac{}{A \vdash A}\ Ax$$

have been replaced by the open node

$$\Gamma' \vdash A, \Delta'$$

Proof: *We use induction. Observe that the "axiomatic" rules (*Ax*,* $\mathbf{1} - R$ *,*$\top - R$*) remain unchanged except that* Ax *rules are changed as specified. All of the unary rules except for* $! - R$ *allow the induction to proceed smoothly. Observe that the two binary rules -* $\otimes - R$ *and* $\& - R$ *- also satisfy the induction. In the case of the* $! - R$ *rule the conclusion cannot contain any atoms; thus even if it does occur in a proof of* $\Gamma, A \vdash \Delta$ *it remains unaffected by the replacement of* A *at the root since the change gets shunted along with occurrences of the root sequent's formula* A*.* ■

For example let $\Gamma = p \multimap q$, let $\Delta = q \otimes !(p \multimap p)$ and let $A = p$. Then the sequent $\Gamma, A \vdash \Delta$ is provable:

$$\dfrac{\dfrac{\dfrac{}{p \vdash p}\,Ax \quad \dfrac{}{q \vdash q}\,Ax}{p \multimap q, p \vdash q}\multimap -L \quad \dfrac{\dfrac{\dfrac{}{p \vdash p}\,Ax}{\vdash p \multimap p}\multimap -R}{\vdash !(p \multimap p)}\,! - R}{p \multimap q, p \vdash q \otimes !(p \multimap p)}\otimes - R$$

By the previous lemma there exists a proof of $\Gamma, \Gamma' \vdash \Delta, \Delta'$ that is a proof of $p \multimap q, \Gamma' \vdash q \otimes !(p \multimap p), \Delta'$ where some of the axiom inferences are replaced accordingly and where the structure of the proof is preserved:

$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash p, \Delta'} \quad \dfrac{}{q \vdash q}\,Ax}{p \multimap q, \Gamma' \vdash q, \Delta'}\multimap -L \quad \dfrac{\dfrac{\dfrac{}{p \vdash p}\,Ax}{\vdash p \multimap p}\multimap -R}{\vdash !(p \multimap p)}\,! - R}{p \multimap q, \Gamma' \vdash q \otimes !(p \multimap p), \Delta'}\otimes - R$$

The next theorem asserts that when the program formulae have a particular form then limiting the proof search process to only perform left-focused proof steps when the relevant atomic goals are present preserves completeness. A problem we shall encounter in the proof of the theorem is that it is possible to begin applying a left-focused proof step even if the guiding atom is not yet present and perform the appropriate right rules to introduce the needed atom only when it is needed for the application of axiom rules. The previous lemma is used to argue that it is complete to ignore proofs which do this, as there will always exist a proof of the same sequent which introduces atoms before decomposing program clauses.

The following proof is an example of this. Note that the proof decomposes a program clause even though both atoms in its head are absent as goals.

$$\dfrac{\dfrac{\dfrac{}{q \vdash q}}{q \vdash q \oplus r}\oplus - R \quad \begin{array}{c}\vdots\\ \vdash G\end{array}}{G \multimap q \vdash q \oplus r}\multimap -L$$

However there exists a proof which begins by introducing the atom q and only then (when both atoms in the clause's head are present as goals) applies left rules to the program

clause.

$$\dfrac{\dfrac{\dfrac{}{q \vdash q} \quad \begin{array}{c}\vdots\\ \vdash G\end{array}}{G \multimap q \vdash q} \multimap -L}{G \multimap q \vdash q \oplus r} \oplus -R$$

DEFINITION 25

The formula F occurs negatively in $F \multimap G$ and in $F^{\perp}$. Intuitively a sub-formula occurs negatively if it can be transferred across the turnstile in the course of a proof construction.

For example, in the sequent $p \multimap q \vdash q \,\&\, (r \oplus s)^{\perp}$ the formulae p and $r \oplus s$ occur negatively.

THEOREM 61

Let Γ be a multiset of formulae of the form $!\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n))$. Let Δ be a multiset of formulae where all negatively occurring formulae are of the appropriate form ($!\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n))$). Then the sequent $\Gamma \vdash \Delta$ has a proof where each sequent in the proof is either the conclusion of a right rule or is the conclusion of a left-focused proof.

Proof: *Observe that the connectives $\forall$, $!$, $\multimap$ and $\parr$ are synchronous when they occur on the left. Thus by an application of the focusing property [6] we can apply the left rules decomposing a program clause as an indivisible group of inferences without any loss of completeness. We shall refer to this sequence of inferences as a left–focused proof step or proof step.*

A left-focused proof step satisfies most of the requirements of a left-focused proof. We only need to show that it either fails or eliminates the relevant goals.

Firstly, we show that if the A_i are present then the proof step eliminates them. Given the sequent $\Gamma \vdash \Delta$, if $\{A_1 \ldots A_n\} \subseteq \Delta$ then the proof step using the program clause $!\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n))$ is left-focused - that is, it eliminates the A_i. The proof involved is the following where we let $\mathcal{P}$ represent $!\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n)), \Gamma$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A_1 \vdash A_1} \quad \ldots \quad \dfrac{}{A_n \vdash A_n}}{A_1 \parr \ldots \parr A_n \vdash A_1, \ldots, A_n} \parr -L \quad \begin{array}{c}\vdots\\ \mathcal{P} \vdash \mathcal{G}, \Delta\end{array}}{\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n), \mathcal{P} \vdash A_1, \ldots, A_n, \Delta} \multimap -L}{\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n)), \mathcal{P} \vdash A_1, \ldots, A_n, \Delta} \forall -L}{!\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n)), \mathcal{P} \vdash A_1, \ldots, A_n, \Delta} ! -L}{\mathcal{P} \vdash A_1, \ldots, A_n, \Delta} C! -L$$

We now need to show that without a loss of completeness we can limit the proof search process so left rules are only applied when the appropriate atomic goals are present. That is, if there is a proof of a sequent which begins with a left-focused proof step where the relevant atomic goals are not all present then there exists another proof where the relevant atomic goals are introduced before the left-focused proof step is performed.

We call an application of a left-focused proof step premature *when some of the relevant atomic goals are absent. The general case for a premature application of a proof step is the following (Π_0), where we assume without loss of generality that the atom missing is A_1:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ A_1 \vdash \Delta'\end{array} \quad \dfrac{}{A_2 \vdash A_2} \quad \ldots \quad \dfrac{}{A_n \vdash A_n}}{A_1 \parr \ldots \parr A_n \vdash \Delta', A_2, \ldots, A_n} \parr -L \quad \begin{array}{c}\vdots\\ \mathcal{P} \vdash \mathcal{G}, \Delta\end{array}}{\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n), \mathcal{P} \vdash A_2, \ldots, A_n, \Delta} \multimap -L}{\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n)), \mathcal{P} \vdash \Delta', A_2, \ldots, A_n, \Delta} \forall -L}{!\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n)), \mathcal{P} \vdash \Delta', A_2, \ldots, A_n, \Delta} ! -L}{\mathcal{P} \vdash \Delta', A_2, \ldots, A_n, \Delta} C! -L}{\begin{array}{c}\vdots\\ \mathcal{P} \vdash \Delta', A_2, \ldots, A_n, \Delta\end{array}}$$

Observe that $A_1 \vdash \Delta'$ must be provable. According to lemma 60, if it is provable then there exists an incomplete proof of the sequent $\mathcal{P} \vdash \Delta', A_2, \ldots, A_n, \Delta$ of the following form

$$\dfrac{\begin{array}{c}\overline{\ldots}\\ \vdots\end{array} \quad \ldots \mathcal{P} \vdash A_1, A_2, \ldots, A_n, \Delta \ldots \quad \begin{array}{c}\overline{\ldots}\\ \vdots\end{array}}{\begin{array}{c}\vdots\\ \mathcal{P} \vdash \Delta', A_2, \ldots, A_n, \Delta\end{array}}$$

However the open node of this proof is the root of the desired proof. Hence we can extend this proof as follows

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{A_1 \vdash A_1} \quad \ldots \quad \overline{A_n \vdash A_n}}{A_1 \parr \ldots \parr A_n \vdash A_1, \ldots, A_n} \otimes - L \quad \vdots\ \mathcal{P} \vdash \mathcal{G}, \Delta}{\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n), \mathcal{P} \vdash A_1, \ldots, A_n, \Delta} \multimap - L}{\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n)), \mathcal{P} \vdash A_1, \ldots, A_n, \Delta} \forall - L}{!\forall x(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n)), \mathcal{P} \vdash A_1, \ldots, A_n, \Delta} ! - L}{\mathcal{P} \vdash A_1, \ldots, A_n, \Delta} C! - L}{\mathcal{P} \vdash \Delta', A_2, \ldots, A_n, \Delta}$$

Thus delaying the application of left-focused proof steps until all of the required atoms are present preserves completeness. Furthermore, the proof step produces a left-focused proof as desired. ■

3.6.1 ACL

ACL (Asynchronous Communication based on Linear logic) [85] has the flavour of a concurrent extension to an ML-like functional language. The concurrent extensions are based on linear logic. From our point of view we can view the language as having the logical syntax[11]

$$\mathcal{D} ::= !\forall \overline{x}(\mathcal{G} \multimap A_p)$$

$$\mathcal{G} ::= \bot \mid \top \mid A_m \mid ?A_m \mid A_p \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \forall x \mathcal{G} \mid \mathcal{R}$$

$$\mathcal{R} ::= \exists \overline{x}(A_m^{\perp} \otimes \ldots \otimes A_m^{\perp} \otimes \mathcal{G}) \mid \mathcal{R} \oplus \mathcal{R}$$

where A_m is a message predicate and A_p a program predicate. ACL satisfies some, but not all of the criteria.

[11] Note that, unlike the technical report version of this chapter [151] we are using the extended version of ACL on page 285 of [85] which includes quantifiers.

PROPOSITION 62

ACL fails to satisfy criterion C_{strong}.

Proof: *Consider the sequent* $\vdash \exists x(p(x)^{\perp} \otimes \top), p(1) \,\&\, p(2)$. *This sequent is provable:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash p(1)^{\perp}, p(1)}\, Ax \quad \dfrac{}{\vdash \top}\, \top - R}{\vdash p(1)^{\perp} \otimes \top, p(1)}\, \otimes - R}{\vdash \exists x(p(x)^{\perp} \otimes \top), p(1)}\, \exists - R \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash p(2)^{\perp}, p(2)}\, Ax \quad \dfrac{}{\vdash \top}\, \top - R}{\vdash p(2)^{\perp} \otimes \top, p(2)}\, \otimes - R}{\vdash \exists x(p(x)^{\perp} \otimes \top), p(2)}\, \exists - R}{\vdash \exists x(p(x)^{\perp} \otimes \top), p(1) \,\&\, p(2)}\, \& - R$$

However, there is no proof which begins with an $\exists - R$ *rule since* t *cannot be simultaneously both* 1 *and* 2*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash p(t)^{\perp}, p(1) \quad \dfrac{}{\vdash \top}\, \top - R}{\vdash p(t)^{\perp} \otimes \top, p(1)}\, \otimes - R}{\vdash p(t)^{\perp} \otimes \top, p(1) \,\&\, p(2)}\, \& - R \quad \dfrac{\dfrac{\vdash p(t)^{\perp}, p(2) \quad \dfrac{}{\vdash \top}\, \top - R}{\vdash p(t)^{\perp} \otimes \top, p(2)}\, \otimes - R}{\vdash \exists x(p(x)^{\perp} \otimes \top), p(1) \,\&\, p(2)}\, \exists - R$$

Thus ACL fails to satisfy criterion C_{strong}. ■

COROLLARY:*ACL fails to satisfy criterion* D_A.

PROPOSITION 63

ACL satisfies criterion B.

Proof: *Simple induction argument. The class of program formulae is too limited to allow for a proof of* $\Gamma \vdash$. ■

PROPOSITION 64

ACL satisfies criterion D_S.

Proof: *According to theorem 61, ACL proofs can be restricted to left-focused proofs without a loss of completeness. Consider now a proof of a sequent of the form* $\Gamma \vdash \mathcal{C}$. *Since there are no atoms, we know that the sequent cannot be the result of left rules or the axiom rule and thus it must be the result of a right rule.* ■

COROLLARY:*ACL satisfies criteria* A *and* C.

3.6.2 LO

LO (Linear Objects) [10] is one of the earlier languages based on linear logic. It is motivated by a desire to add concurrency and object-oriented features to logic programming. Its syntax is given by the BNF:

$$\mathcal{D} ::= !\forall \overline{x}(\mathcal{G} \multimap A_1 \parr \ldots \parr A_n)$$

$$\mathcal{G} ::= A \mid \top \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \parr \mathcal{G}$$

PROPOSITION 65

LO satisfies D_A.

Proof: *All of the connectives used in goals are asynchronous, and so given any goal in a sequent we can permute the rule decomposing that goal to the bottom of the proof, hence the first condition of criterion D_A is satisfied. Furthermore according to theorem 61, LO proofs satisfy the left-focusing condition.* ■

COROLLARY:*LO satisfies D_S, C , A and C_{strong}.*

PROPOSITION 66

LO satisfies B.

Proof: *Simple induction as for ACL.* ■

3.6.3 $\mathcal{LC}$

The language $\mathcal{LC}$ (Linear Chemistry) [140] is based on a similar proof theoretical analysis to Lolli and Lygon. It is designed to satisfy criterion D_A. Its choice of connectives is interesting. Note that $\mathcal{LC}$ does not use any binary rules and thus its proofs are "sticks" rather than trees.

$$\mathcal{D} ::= !\forall \overline{x}(\mathcal{G} \multimap A_1 \parr \ldots \parr A_n)$$

$$\mathcal{G} ::= A \mid \mathbf{1} \oplus \bot \mid \top \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \exists x \mathcal{G}$$

PROPOSITION 67

$\mathcal{LC}$ satisfies criteria B and A.

Proof: *Obvious for criterion A and simple induction for criterion B.* ■

PROPOSITION 68

$\mathcal{LC}$ satisfies criterion D_A.

Proof: *$\mathcal{LC}$ satisfies the left-focusing condition according to theorem 61. Observe that all of the connectives used in goals are* relatively *asynchronous.* ■

COROLLARY:*$\mathcal{LC}$ satisfies criteria D_S , C and C_{strong}.*

As figure 3.6 indicates, the languages proposed fall into three groups:

1. Those which satisfy criterion D_A: LO and $\mathcal{LC}$.

2. Those which satisfy criterion D_S but not D_A: ACL.

3. Those languages which allow program clauses of the form $\mathcal{G} \multimap \bot$ and hence fail most of the criteria: Forum and Lygon.

In figure 3.6 the entries in brackets indicate that the result is due to a technicality and would otherwise be different.

Figure 3.6 Summary of Multiple Conclusioned Languages and Criteria

Language	A	B	C	C_{strong}	D_A	D_S
ACL	✓	✓	✓	✗	✗	✓
LO	✓	✓	✓	✓	✓	✓
$\mathcal{LC}$	✓	✓	✓	✓	✓	✓
Forum	✓	✗	✗	✗	✗	✗
Forum$^-$	✓	✗	(✓)	(✓)	✓	✓
Lygon	✓	✗	✗	✗	✗	✗
Lygon$^\perp$	✓	✓	✓	✗	✗	(✗)
Lygon$^-$	✓	✓	✓	✗	✗	✓
Lygon$_2$ (section 3.11)	✓	✓	✓	✗	✗	✓

3.7 Forum

Forum [109] is intended more as a specification language than as a programming language. It is interesting in that it consists entirely of asynchronous connectives (in goals) and the *same* class of formulae in programs.

$$\mathcal{D} ::= \mathcal{G}$$

$$\mathcal{G} ::= A \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \multimap \mathcal{G} \mid \mathcal{G} \Rightarrow {}^{12}\mathcal{G} \mid \top \mid \bot \mid \forall x \mathcal{G}$$

We begin by considering Forum as the class of formulae above extended using logical equivalences to cover all of linear logic as is done in [106].

PROPOSITION 69

Forum satisfies criteria A.

Proof: *Linear logic does not have generally applicable right structural rules.* ■

PROPOSITION 70

Forum fails to satisfy criterion B.

Proof: *The proof of the sequent $\bot \vdash$ violates criterion B.* ■

PROPOSITION 71

Forum fails to satisfy criterion C.

Proof: *Consider the following derivation:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{p(a) \vdash p(a)}}{p(a) \vdash \exists x p(x)}\,\exists - R}{\vdash p(a)^{\perp}, \exists x p(x)}\,-^{\perp} - R \quad \dfrac{\dfrac{\overline{p(b) \vdash p(b)}}{p(b) \vdash \exists x p(x)}\,\exists - R}{\vdash p(b)^{\perp}, \exists x p(x)}\,-^{\perp} - R}{\vdash p(a)^{\perp} \,\&\, p(b)^{\perp}, \exists x p(x)}\,\& - R}{(p(a)^{\perp} \,\&\, p(b)^{\perp})^{\perp} \vdash \exists x p(x)}\,-^{\perp} - L$$

(this is the example given in section 3.1). ■

COROLLARY:*Forum fails to satisfy criteria D_A , D_S and C_{strong}*

Forum fails most of the tests *when considered in conjunction with logical equivalences*. We consider Forum in conjunction with logical equivalences because that is the

[12] $a \Rightarrow b \equiv (!a) \multimap b$

approach followed by Forum's designers in [106, 109, 111]. We now consider Forum as it stands, without extension. We shall call this language Forum$^-$.

LEMMA 72

Forum$^-$ satisfies criterion A.

Proof: *Trivial, since linear logic does not provide generally applicable right structural rules.* ■

PROPOSITION 73

Forum$^-$ fails to satisfy criterion B.

Proof: *$\perp$ is a legal Forum$^-$ program which when given the goal $\perp$ has the following proof:*

$$\dfrac{\dfrac{}{\perp \vdash}\ \perp - L}{\perp \vdash \perp}\ \perp - R$$

which violates criterion B. ■

In the following, when we say that *"the proof fails"* we mean that it can not be completed with an axiom rule. In general, it may be possible to complete the proof step using other program clauses. However, once we have chosen to decompose a given program clause, lemma 74 (below) allows us to insist that the decomposing proof step terminate with an axiom rule where the program clause reduces to an atom. Hence we can ignore possible completions of the proof step which involve other program clauses without a loss of completeness.

In order to prove the next proposition we shall need to define a notion of *matching*. We can then reason as follows:

1. If the atoms do not match the clause then a proof step focusing on the clause must fail.

2. If the atoms match the clause then a proof step focusing on the clause either fails or eliminates the relevant atoms.

However we begin with a useful lemma.

LEMMA 74

Let Γ and Δ be respectively multisets of Forum$^-$ program clauses and goal formulae such that $\Gamma \vdash \Delta$ is provable. Then there exists a proof of $\Gamma \vdash \Delta$ where all left rules are part of a sequence which decomposes a single clause.

Proof: *Observe that all of the left inference rules which are applicable in Forum$^-$ derivations are synchronous. Hence according to the focusing property [6] once we have applied a left rule to a program formula we can continue to decompose that formula without a loss of completeness. Furthermore, as observed in [6]:*

> *When a* negative *atom $A^\perp$ is reached at the end of a critical focusing section, the Identity must be used, so that A must be found in the rest of the sequent, either as a restricted resource . . . or as an unrestricted resource . . .*

Thus, once a program clause is decomposed to an atom on the left there is no loss of completeness in requiring that the next rule be the axiom rule. ■

In the following we let F be the selected program clause and $\mathcal{A}$ a multiset of atomic goals. Note that here we are matching a program clause with a number of atoms.

DEFINITION 26

The multiset of atoms $\mathcal{A}$ matches *the Forum$^-$ program clause F iff*

- *F is atomic, $\mathcal{A}$ consists of the single formula A and $A = F$.*
- *$F = \perp$ and $\mathcal{A}$ is empty.*
- *$F = F_1 \parr F_2$, $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ and we have that $\mathcal{A}_1$ matches F_1 and $\mathcal{A}_2$ matches F_2.*
- *$F = F_1 \,\&\, F_2$ and $\mathcal{A}$ matches at least one of the F_i*
- *$F = G \multimap F'$ and $\mathcal{A}$ matches F'*
- *$F = G \Rightarrow F'$ and $\mathcal{A}$ matches F'*
- *$F = \forall x F'$ and $\mathcal{A}$ matches F'*

LEMMA 75

Consider a proof step which decomposes the Forum$^-$ program clause F in the sequent $\Gamma, F \vdash \mathcal{A}, \Delta$ where $\mathcal{A}$ is a multiset of atomic formulae. If $\mathcal{A}$ matches F then the proof step either eliminates $\mathcal{A}$ or fails; otherwise, if F does not match $\mathcal{A}$ then the proof step fails.

Proof: *We proceed by induction on the structure of F. There are a number of possible cases:*

- *F is atomic: Without a loss of completeness (lemma 74) the sequent must be the conclusion of an axiom rule. If F matches $\mathcal{A}$ then the multiset actually contains a single atom A and the axiom rule will eliminate A or fail (if there are excess formulae). If F does not match $\mathcal{A}$ then either F does not equal A or $\mathcal{A}$ contains the wrong number of atoms. In either case the axiom rule cannot be applied.*

- *$F = \top$: F can not match $\mathcal{A}$ and since there is no left rule, the proof fails.*

- *$F = \bot$: If F matches $\mathcal{A}$ then $\mathcal{A}$ is empty and it can be considered as consumed so this case is trivially satisfied. If F does not match $\mathcal{A}$ then it must be non empty and hence the $\bot - L$ rule cannot be applied and the proof fails.*

- *$F = F_1 \,\&\, F_2$: If F matches $\mathcal{A}$ then without loss of generality let $\mathcal{A}$ match F_1 and not match F_2. Then by the induction hypothesis the proof which uses the left $\&$ rule to select F_1 either eliminates $\mathcal{A}$ or fails. Since $\mathcal{A}$ does not match F_2 the proof which uses $\&$ to select F_2 fails. If it fails then so does the whole proof, if it consumes $\mathcal{A}$ then so does the conclusion. If F does not match $\mathcal{A}$ then $\mathcal{A}$ matches neither F_1 nor F_2 and by induction the premise of the $\& - L$ rule fails.*

- *$F = G \multimap F'$: The relevant rule is:*

$$\frac{\Gamma', F' \vdash \mathcal{A} \quad \Gamma \vdash G, \Delta}{\Gamma, \Gamma', G \multimap F' \vdash \mathcal{A}, \Delta} \multimap -L$$

  Note that F matches $\mathcal{A}$ iff F' matches $\mathcal{A}$. By the induction hypothesis, the left premise of the proof either fails or consumes $\mathcal{A}$ if F matches $\mathcal{A}$. By the induction hypothesis, the left premise fails if F does not match $\mathcal{A}$.

- $F = F_1 \parr F_2$: *The relevant rule is*

$$\frac{\Gamma, F_1 \vdash \mathcal{A}_1, \Delta \quad \Gamma', F_2 \vdash \mathcal{A}_2, \Delta'}{\Gamma, \Gamma', F_1 \parr F_2 \vdash \mathcal{A}_1, \mathcal{A}_2, \Delta, \Delta'} \parr -L$$

  F matches $\mathcal{A}$:
  By the induction hypothesis both of the premises match and either eliminate their $\mathcal{A}_i$ or fail. If both eliminate their $\mathcal{A}_i$ then so does the whole proof. If either fails then so does the whole proof. If we choose to split $\mathcal{A}$ such that an $\mathcal{A}_i$ does not match F_i then the proof will fail.
  F does not match $\mathcal{A}$:
  By the induction hypothesis at least one of the premises fails to match and hence the proof fails.

The remaining cases are analogous. ■

PROPOSITION 76
Forum$^-$ satisfies criterion D_A.
Proof: *The first condition follows since all of the right rules that are applicable are reversible. Consider now the proof of sequents of the form $\Gamma \vdash \mathcal{A}$. We assume the sequent has a proof. If this proof is simply an application of the axiom inference then the proof satisfies D_A and we are done. Otherwise the proof begins by applying a left rule to some formulae $F \in \Gamma$. According to lemma 74 there exists a proof which begins by completely decomposing F. Since we know that this proof is successful we have by lemma 75 that F must match some sub-multiset of $\mathcal{A}$ and furthermore that the left-focused proof step which decomposes F must eliminate this sub-multiset. Thus Forum$^-$ satisfies the left-focusing condition and hence satisfies D_A.* ■
COROLLARY:*Forum$^-$ satisfies criteria D_S , C and C_{strong}.*

So Forum$^-$ satisfies most of the criteria. As we shall see in section 6.1 there are still problems associated with allowing $\perp$ to occur in the head of a clause.

3.8 Lygon

Lygon is derived by a systematic analysis. In terms of the class of formulae permitted

it is the most general of the languages considered:

$$\mathcal{D} ::= A \mid \mathbf{1} \mid \bot \mid \mathcal{D} \,\&\, \mathcal{D} \mid \mathcal{D} \otimes \mathcal{D} \mid \mathcal{G} \multimap A \mid \mathcal{G}^{\bot} \mid \forall x \mathcal{D} \mid !\mathcal{D} \mid \mathcal{D} \parr \mathcal{D}$$

$$\mathcal{G} ::= A \mid \mathbf{1} \mid \bot \mid \top \mid \mathcal{G} \otimes \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D}^{\bot} \mid \forall x \mathcal{G} \mid \exists x \mathcal{G} \mid !G \mid ?G$$

PROPOSITION 77

Lygon satisfies criterion A.

Proof: *Obvious, since linear logic does not have weakening or contraction right.* ■

PROPOSITION 78

Lygon fails to satisfy criterion B.

Proof: *The sequent $\bot \vdash \bot$ consists of a valid Lygon program and goal and its proof necessarily involves a sub-proof with an empty goal.* ■

PROPOSITION 79

Lygon fails to satisfy criterion C.

Proof: *Consider the Lygon program $(p(1)^{\bot} \,\&\, p(2)^{\bot})^{\bot}$ and the goal $\exists x p(x)$. According to criterion C we can begin the proof by applying $\exists$-R; however, this fails since t cannot be both 1 and 2 simultaneously.*

$$\dfrac{\dfrac{\dfrac{\dfrac{p(1) \vdash p(t)}{\vdash p(1)^{\bot}, p(t)} \quad \dfrac{p(2) \vdash p(t)}{\vdash p(2)^{\bot}, p(t)}}{\vdash p(1)^{\bot} \,\&\, p(2)^{\bot}, p(t)}}{(p(1)^{\bot} \,\&\, p(2)^{\bot})^{\bot} \vdash p(t)}}{(p(1)^{\bot} \,\&\, p(2)^{\bot})^{\bot} \vdash \exists x p(x)} \exists - R$$

This sequent is *provable however:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p(1) \vdash p(1)}}{\vdash p(1)^{\bot}, p(1)}}{\vdash p(1)^{\bot}, \exists x p(x)} \quad \dfrac{\dfrac{\dfrac{}{p(2) \vdash p(2)}}{\vdash p(2)^{\bot}, p(2)}}{\vdash p(2)^{\bot}, \exists x p(x)}}{\vdash p(1)^{\bot} \,\&\, p(2)^{\bot}, \exists x p(x)}}{(p(1)^{\bot} \,\&\, p(2)^{\bot})^{\bot} \vdash \exists x p(x)}$$

Hence Lygon violates criterion C. ■

COROLLARY:*Lygon violates criteria C_{strong}, D_S and D_A.*

Thus Lygon fails most of our criteria! Lygon allows program clauses to be of the form $\mathcal{G}^{\perp}$ (encoded as $\mathcal{G} \multimap \perp$). The problem is that such clauses can be resolved against at anytime; there is no requirement that a particular goal be present. As is pointed out on page 34, allowing program clauses that can be resolved against without being invoked explicitly by a goal causes problems. Specifically, a number of desirable properties of logic programming languages (for instance that $\Gamma \vdash \exists x F$ is provable if and only if $\Gamma \vdash F[t/x]$ is provable for some t) are violated as a result.

We shall therefore consider a variant of Lygon - Lygon$^{\perp}$ - which differs from Lygon in that it removes the production $\mathcal{D} ::= \perp \mid \mathcal{G}^{\perp}$. We will also consider Lygon^{-} which has a limited notion of program clause corresponding to the BNF:

$$\mathcal{D} ::= !\forall \overline{x}(\mathcal{G} \multimap A) \mid A$$

Lygon^{-} also omits the possibility for controlled weakening and contraction of goals which is afforded by formulae of the form $?G$.

$$\mathcal{G} ::= A \mid \mathbf{1} \mid \perp \mid \top \mid \mathcal{G} \otimes \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D}^{\perp} \mid \forall x \mathcal{G} \mid \exists x \mathcal{G} \mid !G$$

As discussed earlier it is possible to make proofs containing replication and deletion of such formulae acceptable with a slight modification to the definition of D_S. Lygon^{-} is closest to the current Lygon implementation [146, 147] (see also section 5.1).

PROPOSITION 80

Lygon$^{\perp}$ and Lygon^{-} satisfy criteria A.

Proof: *Obvious.* ■

PROPOSITION 81

Lygon$^{\perp}$ satisfies criterion B.

Proof: *Proof by induction. In order to be a provable axiom, a sequent must have a non-empty goal - since Lygon$^{\perp}$ programs cannot contain* **0** *or* $\perp$*. Each of the applicable left rules satisfies the condition that if the conclusion has an empty goal then so does a premise.* ■

PROPOSITION 82

Lygon^{-} satisfies criterion B.

Proof: *Simple induction.* ■

PROPOSITION 83

Lygon$^{\perp}$ and Lygon^{-} fail to satisfy criterion C_{strong}.

Proof: *Consider the sequent $\vdash \exists x p(x), \forall x(p(x) \multimap \perp)$.* ■

COROLLARY:*Lygon$^{\perp}$ and Lygon^{-} fail to satisfy criterion D_A.*

PROPOSITION 84

Lygon^{-} satisfies criterion D_S.

Proof: *According to theorem 61, Lygon^{-} satisfies the left-focusing restriction. Observe that the proof of a sequent of the form $\Gamma \vdash \mathcal{C}$ cannot begin with an Ax or left rule since there are no atoms to guide the proof step. Hence, all sequents of the form $\Gamma \vdash \mathcal{C}$ are the conclusion of a right rule.* ■

COROLLARY:*Lygon^{-} satisfies criterion C.*

PROPOSITION 85

Lygon$^{\perp}$ satisfies criterion C.

Proof: *We need to show that for Γ consisting of valid Lygon program clauses and $\exists x F$ a valid Lygon goal the sequent $\Gamma \vdash \exists x F$ is provable if and only if the sequent $\Gamma \vdash F[t/x]$ is provable.*

The only rules which can occur in a Lygon derivation and which do not permute up past $\exists - R$ are $\& - R$ and $\forall - R$. Since the sequent we are considering for criterion C does not have any other goals (that is, the succedent is a singleton multiset) we only need to argue that the following scenarios cannot occur

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash p \,\&\, q, \exists x F & \Gamma', r \vdash\end{array}}{(p \,\&\, q) \multimap r, \Gamma, \Gamma' \vdash \exists x F} \multimap -L \qquad \frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash \forall x F', \exists x F & \Gamma', r \vdash\end{array}}{(\forall F') \multimap r, \Gamma, \Gamma' \vdash \exists x F} \multimap -L$$

Since Lygon$^{\perp}$ satisfies criterion B the right premise cannot be proven and hence this scenario is impossible. Therefore, we can permute the $\exists - R$ rule down to the bottom of the proof and thus $\Gamma \vdash \exists x F$ is provable if and only if $\Gamma \vdash F[t/x]$ is provable as desired.

■

PROPOSITION 86

Lygon$^{\perp}$ fails criterion D_S.

Proof: *The sequent $a \otimes b \vdash a \otimes b$ comprises a valid Lygon$^{\perp}$ program and a valid*

Lygon$^\perp$ goal. It can only be proven by applying $\otimes - L$ first. Since the goal is compound this violates criterion D_S. ■

This is symptomatic of the fact that D_S is a generalisation of uniformity and that the notion of simple locally LR used in Lygon (see [122] is *not* uniformity. The differences are minor and relate to the following impermutabilities

- $\otimes - R$ above $\otimes - L$
- $\otimes - R$ above $C! - L$
- $\mathbf{1} - R$ above $\mathbf{1} - L$ or $W! - L$
- $! - R$ above most left rules

The second and third of these are standard and are easily solved by introducing a "non-linear" region into the rules as is done in $\mathcal{L}$ (figure 2.6 on page 13).

Thus with respect to the sequent rules for linear logic given in figure 2.4, Lygon is *not uniform*; however, if as programmers we are prepared to think in terms of a slightly different set of rules where, for example, $\otimes$-R can apply simple preprocessing to the left-hand side of the sequent then Lygon regains its uniformity.

We invite the reader to consult [122] for the details of the derivation and for the logical rules involved.

3.9 Applying D_A and D_S to Classical Logic

All of the languages in the previous few sections have been based on linear logic. It is interesting to consider applying our criteria (specifically D_A and D_S) to *classical* logic.

We begin by reviewing existing relevant work.

As far as we are aware there is little existing work which applies the idea of uniformity to determine (in a proof theoretical manner) a logic programming language based on classical logic.

In [55] the criteria D_{all} and D_{some} (introduced in the next section) are used to derive logic programming languages based on classical logic. The conclusions drawn in the paper are that:

1. D_{some} does not produce any non-trivial languages that do not also satisfy D_{all}.

2. The following languages satisfy criterion D_{all}:

   A:

   $$\mathcal{D} ::= A \mid \forall x\mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \mid \mathcal{D} \vee \mathcal{D} \mid \neg\mathcal{G} \mid \mathcal{G} \rightarrow \mathcal{D}$$

   $$\mathcal{G} ::= A \mid \exists x\mathcal{G} \mid \mathcal{G} \wedge \mathcal{G} \mid \mathcal{D} \vee \mathcal{G} \mid \neg\mathcal{D} \mid \mathcal{D} \rightarrow \mathcal{G}$$

   B:

   $$\mathcal{D} ::= A \mid \forall x\mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \mid \neg\mathcal{G} \mid \mathcal{G} \rightarrow \mathcal{D}$$

   $$\mathcal{G} ::= A \mid \exists x\mathcal{G} \mid \forall x\mathcal{G} \mid \mathcal{G} \wedge \mathcal{G} \mid \mathcal{D} \vee \mathcal{G}$$

3. The languages **A** and **B** violate goal directedness in that they permit proofs where the (atomic) goal is irrelevant.

   > *"Thus it seems that the completeness of goal-directed provability is a signpost rather than a definitive criterion in classical logic. For example, when negations are allowed in programs, as in both existential-free formulae and flat definite formulae, the goal may not be actually "relevant" to the proof, such as in the sequent* $p, \neg p \vdash q$. *Whilst it is true that this sequent has a right-reductive* [i.e. $\Gamma \vdash \Delta, \mathcal{C}$ is the conclusion of a right rule] *proof, it is clear that there is nothing about this proof that is peculiar to* q*, and so it seems philosophically difficult to describe this proof as goal-directed. Hence it would seem that stronger restrictions than goal-directed* [actually uniform] *provability need to be placed on the class of proofs in order to determine logic programming languages"* ([55, last paragraph, section 5]).

A consequence of observation (3) is that **A** and **B** both fail to satisfy the left-focusing condition and hence the languages fail to satisfy D_A and D_S.

In [114, 115] Nadathur and Loveland examine the application of uniformity to disjunctive logic programming. They conclude that for the language

C:

$$\mathcal{D} ::= A \mid \exists x\mathcal{D} \mid \forall x\mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \mid \mathcal{D} \vee \mathcal{D} \mid \mathcal{G} \rightarrow \mathcal{D}$$

$$\mathcal{G} ::= A \mid \exists x\mathcal{G} \mid \mathcal{G} \wedge \mathcal{G} \mid \mathcal{G} \vee \mathcal{G}$$

the sequent $\Gamma \vdash G$ has a classical proof if and only if the sequent $\neg G, \Gamma \vdash G$ has a proof in *intuitionistic* logic which is *uniform*.

It is simple to show that **C** fails most of our criteria. The proof of the sequent $p(a) \vee p(b) \vdash \exists x p(x)$ violates all of our criteria except for criterion B.

The problem is that although there exists an efficient method of proof search for **C**, it does not confirm to a uniform (as in goal directed) view of classical logic. As in the case of Lygon we can view **C** as being uniform *with respect to a different set of rules*. In this case the logic would require the rule

$$\frac{\Gamma \vdash F[t_1/x], \ldots, F[t_n/x], \Delta}{\Gamma \vdash \exists x F, \Delta}\ \exists' - R$$

We shall return to the issue of modifying the underlying logic in section 3.12.

We now look at deriving a logic programming language based on classical logic using criteria D_A and D_S.

Firstly consider D_A. We require that all connectives that can occur topmost on the right side of the sequent permute over each other. The only impermutability is $\exists - R$ which does not permute down past $\forall - R$, $\exists - L$ and $\vee - L$. Thus our language cannot make use of these three rules if we retain $\exists - R$. Additionally we have seen that allowing $\mathcal{D} ::= \neg\mathcal{G}$ causes problems. This gives us the following language, which we dub **ALK** (**A**synchronous language based on **LK**) (pronounced "elk")

$$\mathcal{D} ::= A \mid \mathcal{D} \wedge \mathcal{D} \mid \forall x\mathcal{D} \mid \mathcal{G} \rightarrow \mathcal{D}$$

$$\mathcal{G} ::= A \mid \mathcal{G} \wedge \mathcal{G} \mid \mathcal{G} \vee \mathcal{G} \mid \exists x\mathcal{G} \mid \mathcal{D} \rightarrow \mathcal{G} \mid \neg\mathcal{D}$$

Note that the connectives which move formulae between sides have sub-formulae which are appropriate for "other" side (e.g. $\mathcal{G} ::= \mathcal{D} \rightarrow \mathcal{G} \mid \neg\mathcal{D}$).

Note that although the syntax is similar to hereditary Harrop formulae [54, 105] the semantics is different since the proof rules are those of classical logic. For example, intuitionistically, $p \vee (p \rightarrow q)$ is unprovable but classically we have a proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\, Ax}{p \vdash p, q}\, W-R}{\vdash p, p \rightarrow q}\, \rightarrow -R}{\vdash p \vee (p \rightarrow q)}\, \vee -R$$

Although the goal q is irrelevant the goal p is used to guide the proof. Thus the proof is goal-directed at all times.

Observe that **ALK** is just **A** with the problematic production ($\mathcal{D} ::= \neg\mathcal{G}$) removed.

We begin by defining a notion of matching. In the following we let F be the selected program clause and A an atomic goal.

DEFINITION 27
The atom A matches *the* **ALK** *program clause* F *iff*

- F *is atomic and is equal to* A.
- $F = F_1 \wedge F_2$ *and* A *matches at least one of the* F_i
- $F = G \rightarrow F'$ *and* A *matches* F'
- $F = \forall x F'$ *and* A *matches* F'

LEMMA 87
Consider a proof step which decomposes the **ALK** *program clause* F *in the sequent* $\Gamma, F \vdash A, \Delta$ *where* A *is atomic. If* A *matches* F *then the proof step succeeds, eliminating* A*; otherwise, if* F *does not match* A *then the proof step fails.*
Proof: *Induction on the structure of* F.

- F *is atomic: If* F *matches* A *then the only relevant rule is the axiom rule which succeeds, consuming* A. *If* F *does not match with* A *then the axiom rule cannot be applied and the proof fails.*

- *$F = F_1 \wedge F_2$: If F matches A then without loss of generality let A match F_1 and not match F_2. Then the premise of the $\wedge$ inference is $\Gamma, F_1, F_2 \vdash A, \Delta$ and by the induction hypothesis its proof eliminates A. If F does not match A then A matches neither F_1 nor F_2 and by induction the premise of the $\wedge - L$ rule fails.*

- *$F = G \rightarrow F'$: The relevant rule is:*

$$\frac{\Gamma', F' \vdash A \quad \Gamma \vdash G}{\Gamma, \Gamma', G \rightarrow F' \vdash A} \rightarrow -L$$

  If A matches F then it must match F' and hence by the induction hypothesis the left premise is provable and hence A is eliminated. If A does not match F then it also fails to match F' and hence by the induction hypothesis the left premise of the inference fails.

The remaining case is analogous. ■

PROPOSITION 88

The language **ALK** *satisfies criterion D_A.*

Proof: *All of the right connectives permute down, thus the first condition (that there exist a proof where any sequent of the form $\Gamma \vdash \Delta, \mathcal{C}$ (where $\mathcal{C}$ consists of compound formulae) is the conclusion of a right rule) holds. The second condition of criterion D_A follows from lemma 87.* ■

COROLLARY:**ALK** *also satisfies criteria D_S, C, C_{strong} and A.*

We now consider applying D_S in order to determine a logic programming language based on classical logic. Obviously **ALK** satisfies D_S. The interesting languages are those which satisfy D_S but not D_A. Consider a language which satisfies D_S. According to proposition 59 if all the right connectives permute then the language also satisfies criterion D_A. Hence, in order for the language to satisfy D_S but not D_A it must contain an impermutable pair of right connectives. For classical logic the only non-permutable pair of right connectives are $\exists$ and $\forall$. Additionally, we would like a useful language to generalise Horn clauses. Thus, a minimal language which generalises Horn clauses and satisfies D_S but not D_A is:

$$\mathcal{D} ::= A \mid \mathcal{G} \rightarrow A \mid \forall x \mathcal{D} \qquad \mathcal{G} ::= A \mid \mathcal{G} \wedge \mathcal{G} \mid \exists x \mathcal{G} \mid \forall x \mathcal{G}$$

However, as the following proof demonstrates, even this language violates criteria D_A and D_S

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p(c) \vdash p(c)}}{p(c) \vdash p(c), q}\, W-R}{p(c) \vdash \exists x p(x), q}\, \exists - R}{\vdash \exists x p(x), p(c) \to q}\, \to -R}{\vdash \exists x p(x), \forall x (p(x) \to q)}\, \forall - R \qquad \dfrac{\dfrac{}{p(3) \vdash p(3)}}{p(3) \vdash \exists x p(x)}\, \exists - R}{(\forall x (p(x) \to q)) \to p(3) \vdash \exists x p(x)}\, \to -L$$

The sequent $(\forall x(p(x) \to q)) \to p(3) \vdash \exists x p(x)$ is provable but there is no single value of x for which it is provable. This violates criterion C (and hence criteria D_A and D_S since $D_A \rightsquigarrow D_S \rightsquigarrow C$). By observation the language used in the proof is

$$\mathcal{D} ::= A \mid \mathcal{G} \to A \qquad \mathcal{G} ::= A \mid \exists x A \mid \forall x A \mid A \to A$$

Since we would like a logic programming language to generalise Horn clauses it follows that we cannot have both $\forall - R$ and $\to -R$ in the language and thus, since we desire to have $\forall - R$, we must omit $\to -R$ from the language.

For the same reason we must also omit $\mathcal{G} ::= \neg\mathcal{D}$. As was the case for **ALK** we cannot allow the proof to make use of the rules $\exists - L$, $\vee - L$ and $\neg - L$ and hence we have the following language which we dub **SLK** (**S**ynchronous language based on **LK**) (pronounced "silk")

$$\mathcal{D} ::= A \mid \mathcal{D} \wedge \mathcal{D} \mid \forall x \mathcal{D} \mid \mathcal{G} \to \mathcal{D}$$

$$\mathcal{G} ::= A \mid \mathcal{G} \wedge \mathcal{G} \mid \mathcal{G} \vee \mathcal{G} \mid \exists x \mathcal{G} \mid \forall x \mathcal{G}$$

This language however satisfies criterion D_A!

That this should be case despite the possibility of applying both $\exists - R$ and $\forall - R$ is explained by the following lemma.

LEMMA 89 (LEMMA 3 OF [55])
Let Γ and Δ be respectively a program and goal in **SLK**. *Then $\Gamma \vdash \Delta$ is provable iff $\Gamma \vdash F$ is provable for some $F \in \Delta$.*

What is happening is that the omission of $\to -R$ and $\neg - R$ prevents the multiple goals from interacting.

Thus our results confirm those of [55] - there do not seem to be any interesting languages based on classical logic which satisfy D_S but not D_A. This should not be surprising - essentially what this says is that classical logic does not have any connectives which are useful for a logic programmer and whose use is made difficult by impermutabilities. More precisely, allowing $\forall$ in goals and $\vee$ and $\exists$ in programs does not appear to be useful to a logic programmer.

3.10 Other Work

In this section we look at how this work ties in with previous work in uniformity and the characterisation of logic programming languages. An important goal is to suggest a common terminology.

As mentioned earlier the seminal work in this area is [112] where the original definition of uniformity is presented. Since uniformity works reasonably well in a single conclusion setting there has been little research on extending it until the need arose for a multiple conclusion generalisation.

In the absence of a notion of atom-guided proof search, the options that have been considered in order to extend uniformity to a multiple conclusion setting [55, 106, 140] are:

D_{all} : For a sequent of the form $\Gamma \vdash \mathcal{C}, \mathcal{A}$ or $\Gamma \vdash \mathcal{C}$ there exist proofs where the first step introduces the topmost connective of F for *all* $F \in \mathcal{C}$.

D_{some} : For a sequent of the form $\Gamma \vdash \mathcal{C}, \mathcal{A}$ or $\Gamma \vdash \mathcal{C}$ there exist proofs where the first step introduces the topmost connective of F for *some* $F \in \mathcal{C}$.

In the absence of a notion of atom-directed proof search it is not possible to allow resolution to occur while there are compound goals, and thus D_{some} is not useful since, as we have seen, allowing impermutable rules requires that we allow resolution to occur before decomposition in certain cases. For example, the following proof necessarily vi-

olates D_{some}:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{q(c) \vdash q(c)} \quad \overline{r(c) \vdash r(c)}}{q(c) \parr r(c) \vdash q(c), r(c)} \parr - L}{\forall x(q(x) \parr r(x)) \vdash q(c), r(c)} \forall - L}{\forall x(q(x) \parr r(x)) \vdash q(c), \exists x r(x)} \exists - R}{\dfrac{\overline{p \vdash p} \quad \forall x(q(x) \parr r(x)) \vdash \forall x q(x), \exists x r(x)}{\forall x(q(x) \parr r(x)), (\forall x q(x)) \multimap p \vdash p, \exists x r(x)} \multimap - L} \forall - R$$

One solution is to disallow $\multimap -L$; however this does not yield a useful logic programming language. The obvious conclusion is that criterion D_{some} is not useful. Indeed, in the absence of any notion of atom-guided proof search, the only generalisation of uniformity to the multiple conclusioned setting is D_{all}. As we have seen, however, introducing a notion of atom-guided proof search allows a variant of D_{some} – D_S – to be useful.

PROPOSITION 90

$D_S \not\rightarrowtail D_{all}$

Proof: *Obvious from definition.* ■

PROPOSITION 91

$D_A \rightsquigarrow D_{all}$

Proof: *Obvious from definition.* ■

PROPOSITION 92

$D_{all} \not\rightarrowtail D_S$, $D_{all} \not\rightarrowtail D_A$

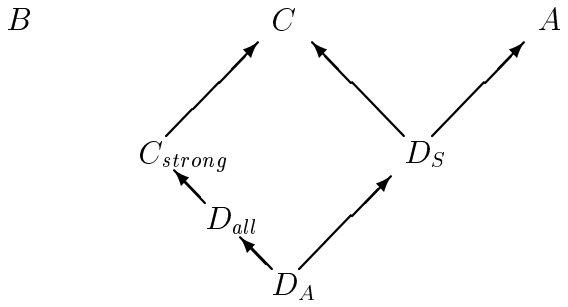
Proof: *The following proof satisfies D_{all} but violates D_S and D_A*

$$\dfrac{\dfrac{\overline{\vdash \top, p} \quad \overline{q \vdash q}}{q \vdash q \otimes \top, p}}{q, (q \otimes \top)^{\perp} \vdash p}$$

Note that if we remove the second condition from D_A (which simply states that D_A is guided by atoms where there are no compound goals) then we are left with D_{all}; that is

$$D_A \;=\; D_{all} + \text{left-focusing}$$

Figure 3.7 Relationships for Multiple Conclusion Criteria



3.10.1 Proposed Terminology

There has not been much work addressing the question of determining logic programming languages. As a result terminology is not standard. In particular the word "Uniform" has been used for different definitions and this overloading can and has caused confusion and mis-communication.

In this section we propose some terminology which avoids confusion. We would like to suggest this terminology as a possible standard. One feature of this terminology is that it distinguishes between single and multiple conclusion logics - the term "uniform" is defined as a property of single conclusion systems only.

Goal Directed - This is an intuitive, informal notion that requires that the proof search process be guided by the goal.

Uniform - This is the original definition in [112] which applies to single conclusion systems. It corresponds to criterion D_{strong}.

Fully Uniform - Uniformity with the addition of left-focusing. Applies to single conclusion systems and corresponds to criterion F.

Simple - This is one method of insisting that atoms guide the proof search process. It is

used in [107] and requires that the right premise of the $\multimap -L$ rule be the conclusion of an axiom rule.

Left Focussed - This is another method of allowing atoms to guide the proof search process.

Locally LR - This is the criterion used in [122] for defining Lygon. Although the paper defines uniform as simple locally LR we feel that this overloading is undesirable since the two concepts are distinct. The locally LR criterion is similar to synchronous uniformity (see below) extended to handle a number of other impermutabilities in linear logic (for example $\otimes$-L below $\otimes$-R).

Synchronous Uniformity - This criterion is one of the two generalisations of uniformity to the multiple conclusion setting. It allows resolution to occur before decomposition thus allowing impermutable goal connectives to co-exist in a logic programming language. It corresponds to criterion D_S.

Asynchronous Uniformity - This criterion is one of the two generalisations of uniformity to the multiple conclusion setting. It insists that decomposition be given priority over resolution and as a consequence limits the language to use (relatively) reversible connectives in goals. It corresponds to criterion D_A.

3.11 Re-Deriving Lygon

As we have seen, the full Lygon language as presented in [122] fails most of the criteria developed. We have introduced two subsets of Lygon – Lygon$^{\perp}$ and Lygon^{-}. The first still fails criterion D_S; although the failure is not as severe. Lygon^{-} satisfies all of the appropriate criteria; however the class of program formulae permitted is rather impoverished.

In this section we informally but systematically derive an intermediate subset – Lygon$_2$ – which satisfies criteria B and D_S and has a richer clause structure than Lygon^{-} .

We shall assume that goals can contain, as a minimum, $\otimes$, !, $\parr$, **1** and $\exists$. As we shall see, the presence of $\otimes$ and ! in goals allows us to simplify the structure of program

clauses. The connective $\parr$ is essential for concurrency applications and $\exists$ is basic to logic programming.

We can rule out at the outset the use of the following left inference rules:[13]

- $\exists$ and $\oplus$. Both of the following sequents only have a proof which begins with a left rule, this violates criterion D_S.

$$p(1) \oplus p(2) \vdash \exists x p(x) \qquad \exists x p(x) \vdash \exists x p(x)$$

- $\otimes$. As we have seen, the sequent $p \otimes q \vdash p \otimes q$ presents a problem to uniform provability.

- $?$. There are two cases here. If $?$ is allowed in goals then the sequent $?p \vdash ?p$ violates criterion D_S. Otherwise observe that the left $?$ rule can only be applied when the goal is empty so the formula is either useless or involves an empty goal which violates criterion B.

- $\mathbf{0}$. The sequent $\mathbf{0} \vdash \mathbf{1}$ has a single proof which applies a left rule. Since $\mathbf{1}$ is not an atom this proof is (technically) not uniform. In general allowing $\mathbf{0} - L$ to be used seems to go against the idea of logic programming in that the goal becomes irrelevant – any goal can be proven using $\mathbf{0} - L$.

- $\bot$ directly violates criterion B.

- $\mathbf{1}$. In the sequent $p \,\&\, \mathbf{1} \vdash !\mathbf{1}$ the $!$ rule on the right cannot be applied until the $\mathbf{1}$ on the left is eliminated using its rule.

Additionally, we cannot have nested occurrences of $!$ in programs – occurrences of $!$ must be at the top level. The problem with nested occurrences of $!$ is that the right $!$ and $\otimes$ rules require that all occurrences of $!$ be at the top level (in the case of the $\otimes$ rule we need to copy all non-linear formulae). If an occurrence of $!$ on the left is in a sub-formula then we may need to apply left rules to expose it before being able to apply a right rule.

[13] Note that we are ruling out left *inference rules*. The connectives in question can occur in programs, but they must occur in a negative context. For example, the program clause $p \leftarrow \exists x(q \oplus r)$ offers no difficulties to uniform provability.

For example, observe that a proof of the sequent $q\&!p \vdash p \otimes p$ exists (the right proof below) but that if we begin by applying the right rule – as required by criterion D_S – then no proof can be found.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\,Ax}{!p \vdash p}\,! - L}{q\&!p \vdash p}\,\& - L \qquad p}{q\&!p \vdash p \otimes p}\,\otimes - R \qquad \dfrac{\dfrac{\dfrac{\vdots \quad \vdots}{!p \vdash p \quad !p \vdash p}\,\otimes - R}{!p, !p \vdash p \otimes p}\,C! - L}{!p \vdash p \otimes p}\,\& - L}{q\&!p \vdash p \otimes p}$$

Note that we could have used almost any left connective in place of $\&$.

This yields the following definition for Lygon$_2$:

$$\mathcal{G} ::= A \mid \mathbf{1} \mid \bot \mid \top \mid \mathcal{G} \otimes \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \& \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D}^{\perp}{}^{14} \mid \forall x \mathcal{G} \mid \exists x \mathcal{G} \mid !G \mid ?G$$

$$\mathcal{D} ::= !\mathcal{D}_n \mid \mathcal{D}_l$$

$$\mathcal{D}_n ::= A \mid \mathcal{D}_n \parr \mathcal{D}_n \mid \mathcal{D}_n \,\&\, \mathcal{D}_n \mid \forall \mathcal{D}_n \mid \mathcal{G} \multimap \mathcal{D}_n$$

$$\mathcal{D}_l ::= \top \mid A \mid \mathcal{D}_l \parr \mathcal{D}_l \mid \mathcal{D}_l \,\&\, \mathcal{D}_l \mid \forall x \mathcal{D}_l \mid \mathcal{G} \multimap \mathcal{D}_l$$

We now proceed to demonstrate how the form of non-linear clauses can be simplified without a loss of expressiveness.

Observe that the program clauses allowed in Lygon$_2$ are a subset of the Forum class of program clauses[15]. As is observed in [106] there exists the following normal form for Forum program clauses:

$$\mathcal{D} ::= !(C_1 \,\&\, \ldots \,\&\, C_n) \mid (C_1 \,\&\, \ldots \,\&\, C_n)$$

$$C ::= \forall \overline{x}(\mathcal{G} \leftarrow \ldots \leftarrow (\mathcal{G} \leftarrow (A_1 \parr \ldots \parr A_n)))$$

where $\leftarrow$ is one of $\Rightarrow$ or $\multimap$. We can rewrite this by expanding $F \Rightarrow G$ as $(!F) \multimap G$ to

$$C ::= \forall \overline{x}(\mathcal{G}' \multimap \ldots \multimap (\mathcal{G}' \multimap (A_1 \parr \ldots \parr A_n)))$$

[14] This is a special case of $\mathcal{D} \multimap \mathcal{G}$.

[15] The two classes of formula are almost identical – the main difference is the absence of $\bot$ in the Lygon$_2$ class.

$$\mathcal{G}' ::= \mathcal{G} \mid !\mathcal{G}$$

Since Lygon$_2$ goals allow both $\otimes$ and ! we can use the following equivalence to replace nested implications with occurrences of $\otimes$ in goals.

$$\mathcal{G}' \multimap \ldots \multimap (\mathcal{G}' \multimap (A_1 \parr \ldots \parr A_n)) \equiv (\mathcal{G}' \otimes \ldots \otimes \mathcal{G}') \multimap (A_1 \parr \ldots \parr A_n)$$

Thus we have the following normal form for Lygon$_2$ program clauses

$$\mathcal{D} ::= !(C_1 \,\&\, \ldots \,\&\, C_n) \mid (C_1 \,\&\, \ldots \,\&\, C_n)$$

$$C ::= \forall \overline{x}(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n))$$

Since $!(C_1 \,\&\, \ldots \,\&\, C_n) \equiv (!C_1) \otimes \ldots \otimes (!C_n)$ we can replace nonlinear occurrences of $C_1 \,\&\, \ldots \,\&\, C_n$ with n separate clauses.

$$\mathcal{D} ::= (C_1 \,\&\, \ldots \,\&\, C_n) \mid !C$$

$$C ::= \forall \overline{x}(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n))$$

Note that unlike Forum we insist that $n > 0$, that is we do not allow program clauses of the form $\mathcal{G} \multimap \bot$.

So, we can define Lygon$_2$ as consisting of the following class of formulae:

$$\mathcal{D} ::= (C_1 \,\&\, \ldots \,\&\, C_n) \mid !C$$

$$C ::= \forall \overline{x}(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n))$$

$$\mathcal{G} ::= A \mid \mathbf{1} \mid \bot \mid \top \mid \mathcal{G} \otimes \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D}^{\perp}{}^{16} \mid \forall x \mathcal{G} \mid \exists x \mathcal{G} \mid !G \mid ?G$$

We now proceed to prove that Lygon$_2$ satisfies criteria D_S and B.

LEMMA 93

Let Γ and Δ be respectively multisets of Lygon$_2$ program clauses and goal formulae such that $\Gamma \vdash \Delta$ is provable. Then there exists a proof of $\Gamma \vdash \Delta$ where all left rules are part of a sequence which decomposes a single clause.

Proof: *Observe that all of the left inference rules which are applicable in Lygon$_2$ derivations are synchronous. Hence according to the focusing property [6] once we have applied a left rule to a program formula we can continue to decompose that formula without a loss of completeness. Furthermore, as observed in [6]:*

[16] This is a special case of $\mathcal{D} \multimap \mathcal{G}$.

> *When a* negative *atom $A^{\perp}$ is reached at the end of a critical focusing section, the Identity must be used, so that A must be found in the rest of the sequent, either as a restricted resource . . . or as an unrestricted resource . . .*

Thus, once a program clause is decomposed to an atom on the left there is no loss of completeness in requiring that the next rule be the axiom rule. ■

In the following lemma we let F be the selected program clause and $\mathcal{A}$ a multiset of atomic goals. Note that here we are matching a program clause with a number of atoms.

DEFINITION 28

The multiset of atoms $\mathcal{A}$ matches *the Lygon$_2$ program clause F iff*

- *F is atomic, $\mathcal{A}$ consists of the single formula A and $A = F$.*
- *$F = F_1 \parr F_2$, $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ and we have that $\mathcal{A}_1$ matches F_1 and $\mathcal{A}_2$ matches F_2.*
- *$F = F_1 \,\&\, F_2$ and $\mathcal{A}$ matches at least one of the F_i*
- *$F = G \multimap F'$ and $\mathcal{A}$ matches F'*
- *$F = \forall x F'$ and A matches F'*
- *$F = !F'$ and A matches F'*

For example the program clause $!\forall \overline{x}(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n))$ is matched precisely by the multiset $\{A_1, \ldots, A_n\}$.

In the following, when we say that *"the proof fails"* we mean that it can not be completed with an axiom rule. In general, it may be possible to complete the proof step using other program clauses. However, once we have chosen to decompose a given program clause, lemma 93 allows us to insist that the decomposing proof step terminate with an axiom rule where the program clause reduces to an atom. Hence we can ignore possible completions of the proof step which involve other program clauses without a loss of completeness.

LEMMA 94

Consider a proof step which decomposes the Lygon$_2$ program clause F in the sequent $\Gamma, F \vdash \mathcal{A}, \Delta$ where $\mathcal{A}$ consists of atomic formulae. If $\mathcal{A}$ matches F then the proof step

either eliminates $\mathcal{A}$ or fails; otherwise, if F does not match $\mathcal{A}$ then the proof step fails.
Proof: *We proceed by induction on the structure of F. There are a number of possible cases:*

- *F is atomic: Without a loss of completeness (lemma 93) the sequent must be the conclusion of an axiom rule. If F matches $\mathcal{A}$ then the multiset actually contains a single atom A and the axiom rule either fails due to extraneous formulae or eliminates A. If F does not match $\mathcal{A}$ then either F does not equal A or $\mathcal{A}$ contains the wrong number of atoms. In either case the axiom rule cannot be applied.*

- *$F = F_1 \,\&\, F_2$: If F matches $\mathcal{A}$ then without loss of generality let $\mathcal{A}$ match F_1 and not match F_2. Then by the induction hypothesis, the proof which uses the left $\&$ rule to select F_1 either eliminates $\mathcal{A}$ or fails. Since $\mathcal{A}$ does not match F_2 the proof which uses $\&$ to select F_2 fails. If it fails then so does the whole proof, if it consumes $\mathcal{A}$ then so does the conclusion. If F does not match $\mathcal{A}$ then $\mathcal{A}$ matches neither F_1 nor F_2 and by induction the premise of the $\& - L$ rule fails.*

- *$F = G \multimap F'$: The relevant rule is:*

$$\frac{\Gamma', F' \vdash \mathcal{A} \quad \Gamma \vdash G, \Delta}{\Gamma, \Gamma', G \multimap F' \vdash \mathcal{A}, \Delta} \multimap -L$$

  Note that F matches $\mathcal{A}$ iff F' matches $\mathcal{A}$. By the induction hypothesis the left premise of the proof either fails or consumes $\mathcal{A}$ if F matches $\mathcal{A}$. By the induction hypothesis the left premise fails if F does not match $\mathcal{A}$.

- *$F = F_1 \parr F_2$: The relevant rule is*

$$\frac{\Gamma, F_1 \vdash \mathcal{A}_1, \Delta \quad \Gamma', F_2 \vdash \mathcal{A}_2, \Delta'}{\Gamma, \Gamma', F_1 \parr F_2 \vdash \mathcal{A}_1, \mathcal{A}_2, \Delta, \Delta'} \parr - L$$

  F matches $\mathcal{A}$:
  By the induction hypothesis both of the premises match and either eliminate their $\mathcal{A}_i$ or fail. If both eliminate their $\mathcal{A}_i$ then so does the whole proof. If either fails then so does the whole proof. If we choose to split $\mathcal{A}$ such that an $\mathcal{A}_i$ does not match F_i then the proof will fail.

F does not match $\mathcal{A}$:

By the induction hypothesis at least one of the premises fails to match and hence the proof fails.

The remaining cases are analogous. ■

THEOREM 95

Lygon$_2$ satisfies criterion D_S.

Proof: *Let Γ and Δ be respectively multisets of Lygon$_2$ program clauses and goal formulae such that $\Gamma \vdash \Delta$ is provable. Consider the proof of the sequent $\Gamma \vdash \Delta$. There are three cases depending on which inference rule the sequent is the conclusion of.*

Case 1: *The sequent is the conclusion of an axiom rule. In this case the proof satisfies criterion D_S.*

Case 2: *The sequent is the conclusion of a right rule. In this case the proof satisfies criterion D_S.*

Case 3: *The sequent is the conclusion of a left rule. The left rule is applied to some formula $F \in \Gamma$. According to lemma 93 there exists a proof which decomposes F using a left-focused proof step. Since the sequent is provable we have by lemma 94 that F must match some sub-multiset of atoms $\mathcal{A} \subseteq \Delta$ and furthermore that the left-focused proof step decomposing F eliminates $\mathcal{A}$. Thus the second condition of D_S is satisfied. Consider now the proof of a sequent of the form $\Gamma \vdash \mathcal{C}$ where $\mathcal{C}$ contains only compound formulae. We know that limiting the proof search process to using left-focussed proof steps does not lose completeness. Since a left-focused proof step cannot be successfully performed when the goal does not contain any atoms, we have that a Lygon$_2$ goal not containing any atomic formulae is provable if and only if there is a proof which begins with a right rule.* ■

COROLLARY:*Lygon$_2$ satisfies criteria A and C.*

THEOREM 96

Lygon$_2$ satisfies criterion B.

Proof: *The only left rules that can occur in a proof of a Lygon$_2$ program and goal are: $!$, $\forall$, $\parr$, $\multimap$ and $\&$. In each of these rules, if the premise of the rule has an empty goal then so does at least one premise. The only rules with no premises which allow an empty goal*

are $\bot - L$ *and* $\mathbf{0} - L$*, and neither of these can occur in a Lygon*$_2$ *program.* ■

We compare Lygon$_2$ to other linear logic programming languages in chapter 6.

3.12 Discussion

We have seen how a variety of criteria for characterising logic programming languages relate to each other and how they apply to a range of logic programming languages which have been proposed.

The four major conclusions that can be drawn from this chapter are that:

1. Logical equivalence can be abused.
2. Uniformity is a promising start, but it suffers (particularly in the multiple conclusion setting) from not having atomic goals guide the proof search process.
3. For the single conclusion setting, *full uniformity* presents an improvement over uniformity.
4. For multiple conclusion logics both *synchronous* and *asynchronous* uniformity are useful as characterisers. The former yields richer languages and the latter yields a permutable set of connectives.

The recognition of the need for atomic goals to guide the proof search and the introduction of *left-focused proof steps* as a mechanism for achieving this are the key contributions of this chapter. The introduction of left-focused proof steps is a crucial step which allows the definition of a working version of synchronous uniformity.

In section 3.1 we identified four problems which uniformity suffers from:

1. It is defined for single conclusioned systems.
2. It is sensitive to the presentation of the logic.
3. It does not constrain the proof of sequents with atomic goals.
4. It allows for the abuse of logical equivalence.

Our proposed criteria solve the first and third problems. The abuse of logical equivalences remains a problem – since it is applied *after* a criterion has been used to derive a logic programming language we need to consider the larger picture in order to be able avoid this problem.

Sensitivity to the presentation of the logic is in general a problem (and we discuss it further in section 3.12). However, the extreme case of one sided presentations can be handled. If we do not allow $\bot$ headed clauses (and note that criteria D_S excludes $\bot$ headed clauses (see page 64)) then the definition of left focusing can insist that $\mathcal{A}$ be non-empty. This prevents the use of a one sided presentation which only uses left rules.

Additionally, criteria D_A does not allow the use of a one sided presentation which only uses right rules. For example, consider the program and goal $q \multimap p, \mathbf{1} \multimap (q \parr r) \vdash p \parr r$. This sequent is provable in a way which satisfies D_A:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\,Ax \qquad \dfrac{\dfrac{\dfrac{}{q \vdash q}\,Ax \quad \dfrac{}{r \vdash r}\,Ax}{q \parr r \vdash q, r}\,\parr - L \quad \dfrac{}{\vdash \mathbf{1}}\,\mathbf{1}}{\mathbf{1} \multimap (q \parr r) \vdash q, r}\,\multimap - L}{q \multimap p, \mathbf{1} \multimap (q \parr r) \vdash p, r}\,\multimap - L}{q \multimap p, \mathbf{1} \multimap (q \parr r) \vdash p \parr r}\,\parr - R$$

Consider now the one-sided translation of the sequent which is $\vdash p \parr r, p^\bot \otimes q, \mathbf{1} \otimes r^\bot \otimes q^\bot$. This sequent is provable, but criterion D_A requires that for any goal formula F there is a proof which begins by introducing the topmost connective of F. This is not the case:

$$\dfrac{\vdash p^\bot, p \parr r \quad \vdash q, \mathbf{1} \otimes r^\bot \otimes q^\bot}{\vdash p \parr r, p^\bot \otimes q, \mathbf{1} \otimes r^\bot \otimes q^\bot}\,\otimes$$

No matter how we distribute the formulae between the two premises of the $\otimes$ rule, a proof is not possible. Thus criterion D_A avoids the problem of one sided presentations being trivially logic programming languages.

There is another more general approach to dealing with one sided presentations. Both criteria D_A and D_S require that proofs of atomic goals are left-focused. This is essentially just resolution and in order to apply it we need to be able to distinguish between programs and goals. Note that D_S allows impermutabilities between goals, and so it is possible to view a one sided presentation as consisting entirely of goals. This implies that there is *no*

program - the notion of a program is coded in terms of goals. From an operational point of view a program clause is passive and is only "activated" when the appropriate atomic goal is present. Encoding program clauses as goals makes them active and allows for more proofs. In the example above the two sided proof given is the only one which satisfies D_A (or D_S). On the other hand the one sided presentation which encodes programs as goals allows for a number of different proofs which all satisfy D_S (but not D_A):

$$\dfrac{\dfrac{\dfrac{}{\vdash p, p^\perp}\,Ax \quad \dfrac{\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1} \quad \dfrac{\dfrac{}{\vdash r, r^\perp}\,Ax \quad \dfrac{}{\vdash q, q^\perp}\,Ax}{\vdash q, r, r^\perp \otimes q^\perp}\otimes}{\vdash r, q, \mathbf{1} \otimes r^\perp \otimes q^\perp}\otimes}{\vdash p, r, p^\perp \otimes q, \mathbf{1} \otimes r^\perp \otimes q^\perp}\otimes}{\vdash p \parr r, p^\perp \otimes q, \mathbf{1} \otimes r^\perp \otimes q^\perp}\parr$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1} \quad \dfrac{\dfrac{}{\vdash p, p^\perp}\,Ax \quad \dfrac{\dfrac{}{\vdash r, r^\perp}\,Ax \quad \dfrac{}{\vdash q, q^\perp}\,Ax}{\vdash r, q, r^\perp \otimes q^\perp}\otimes}{\vdash p, r, p^\perp \otimes q, r^\perp \otimes q^\perp}\otimes}{\vdash p, r, p^\perp \otimes q, \mathbf{1} \otimes r^\perp \otimes q^\perp}\otimes}{\vdash p \parr r, p^\perp \otimes q, \mathbf{1} \otimes r^\perp \otimes q^\perp}\parr$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1} \quad \dfrac{\dfrac{}{\vdash r, r^\perp}\,Ax \quad \dfrac{\dfrac{}{\vdash p, p^\perp}\,Ax \quad \dfrac{}{\vdash q, q^\perp}\,Ax}{\vdash p, p^\perp \otimes q, q^\perp}\otimes}{\vdash p, r, p^\perp \otimes q, r^\perp \otimes q^\perp}\otimes}{\vdash p, r, p^\perp \otimes q, \mathbf{1} \otimes r^\perp \otimes q^\perp}\otimes}{\vdash p \parr r, p^\perp \otimes q, \mathbf{1} \otimes r^\perp \otimes q^\perp}\parr$$

In a sense having only goals is an explicit request from the programmer for a different semantics. Having the system transform the program and goal to a one-sided presentation is akin to the abuse of logical equivalences in that the semantics of the program can be changed – in this case the search space becomes larger which can lead to reduced efficiency, certain solutions being found later (and possibly not at all) etc.

As we have seen, a language is uniform or not *with respect to a given set of rules*. In some cases (for example disjunctive logic programming) changes to the logic affect whether a language is considered to be a logic programming language. As with the use of logical equivalences this raises the question: when is it "sound" to "fiddle" with a logic?

An important question is to what extent the presentation of the logic can affect the derivation of programming languages. It would be desirable for different presentations of the same logic to yield the same language. This is not the case in general (for example D_A seems to be more restrictive for one sided presentations) but in the examples we have seen

that changes to the language were fairly minor. We now show that in general, significant changes to the language derived can be made by changing the presentation of a logic.

Let us consider as an example the multiple conclusioned presentation of intuitionistic logic presented in figure 2.3 on page 9. Since intuitionistic logic has seen a fair amount of work and some definite conclusions [56] this is a good test.

Consider the proof of the sequent $p \vee q \vdash p \vee q$. Using the standard (single conclusioned) rules for intuitionistic logic this sequent has a proof but the proof must begin by applying a left rule. Thus uniformity (and any of our extensions) will not deem the sequent to be legal in a logic programming language.

On the other hand, using the multiple conclusioned rules we have a uniform proof:

$$\dfrac{\dfrac{\dfrac{}{p \vdash p}\,Ax \quad \dfrac{}{q \vdash q}\,Ax}{p \vee q \vdash p, q}\,\vee - L}{p \vee q \vdash p \vee q}\,\vee - R$$

This indicates that the presentation of a logic can, in general, completely change the logic programming language derived. There is room for further work into the derivation of logic programming languages. This work is likely to go beyond uniformity and look at the big picture.

The Big Picture

Let us take a step back and consider the global picture. In programming there are a number of different levels at which a program can be viewed. These are (i) the formal semantics level, (ii) the implementation level, and (iii) a level in between which the programmer uses to visualise the execution.

In some cases the third logic is just one of the previous two but in general this is not the case. When the implementation makes use of sophisticated algorithms (for example, lazy evaluation for functional languages [74, 75], co-routining [116] or constraint solving [79] for logic programming languages, or indeed, lazy resource allocation for linear logic programming languages) it becomes infeasible to use the implementation rules as a way of visualising execution. The formal semantics can be too high level to visualise execution – for example issues of efficiency – both time and space – are often ignored.

When we consider designing a logic programming language, we should consider these three levels. In the context of logic programming these are expressed as logics:

1. A *semantic* logic,
2. A *visualisation* logic, and
3. An *execution* logic.

The semantic logic is typically the standard sequent calculus rules for the logic. It is used to give a declarative semantics to the language. This logic is used to prove properties of programs and to construct tools such as program analysers, partial evaluators etc.

The execution logic must be "tolerably efficient". This includes for example the handling of $\exists - R$ by unification and the handling of $\otimes - R$ by a lazy resource allocation mechanism. As a rough guide the application of a rule should not require an infinite ($\exists - R$) or exponential ($\otimes - R$) choice.

The visualisation logic is used by the programmer to picture the execution of the program. When the programmer asks the system "how was this answer derived?" the answer is expressed in terms of the visualisation logic. The essential property of this logic is its *simplicity*. I would like to suggest that *the existence of the visualisation logic is what distinguishes logic programming from theorem proving.* Note that the visualisation logic does not need to be efficient – for example when visualising the execution of Lygon programs it is natural to work with the naïve version of $\otimes$.

The design of characterisers which can be used to judge whether a logic subset can be considered a programming language amounts to an investigation of the required relationship between these three logics.

For example, uniformity requires that the visualisation logic be the semantic logic limited to a goal-directed proof search strategy. There does not seem to be a good reason why more general visualisation logics are not feasible. Indeed, for certain languages (e.g. Lygon and disjunctive logic programming) they are highly desirable.

We would like to suggest that the execution logic should be related to the visualisation logic at the *inference* level. That is, if

$$\frac{\Gamma \vdash \Delta}{\Gamma' \vdash \Delta'}$$

is an inference rule in the visualisation logic then there is an essentially equivalent set of inference steps in the execution logic. Conversely, any proof in the execution logic should be able to be viewed as a collection of proof steps each of which corresponds to an inference at the visualisation level.

This strong relationship between the execution and visualisation logics is required so that the programmer can use the visualisation logic to debug programs being executed by the execution logic. This requirement prevents arbitrary theorem provers from being considered as logic programming languages. On the other hand, we see no obvious reason why the relationship between the semantic and visualisation logics needs to be any stronger than the theorem level, that is, that there is a proof of $\Gamma \vdash \Delta$ using the semantic logic if and only if there is a proof of the same sequent using the visualisation logic.

There is room for further work in this area. The fundamental question of what constitutes a logic programming languages is still not well understood.

Chapter 4

Implementation Issues

Implementing Lygon is nontrivial. In addition to the usual issues for logic programming languages there are a number of new ones. Two particular problems stand out:

1. In searching for a proof of a goal involving the connective $\otimes$ we need to split the context between the two subproofs. This is done in the reduction of the $\otimes$ rule. Since logic programming languages search for proofs in a bottom up fashion this splitting is nondeterministic and inefficient if done naïvely.

2. Consider searching for a proof of a multiple conclusioned goal. Each time an inference rule is to be applied we must first select the formula that is to be reduced. This formula is the *active formula*. If done naïvely this selection process implies that any proof involving multiple conclusions has significantly more non-determinism than necessary.

This chapter discusses these two issues and presents solutions to them. The solutions presented have been incorporated in the current implementation of Lygon.

We begin by tackling the first problem. We show how the lazy implementation of the multiplicative connectives (specifically $\otimes$) cannot be done simply by altering the appropriate rule for $\otimes$, but requires the entire system to be redesigned, and particular care taken with the rules for $\&$ and $\top$. Essentially this is done by adding some new markers to formulae, to indicate whether the formula has been used in another part of the proof or not, and hence determine what resources are available to the current branch of the proof.

This process may be thought of as a problem of resource management, in that one of the key requirements of the proof search process is to allocate each formula to a branch of the proof. Clearly any implementation of Lygon will need to do this in a deterministic manner, i.e., follow a particular allocation strategy. It is the technicalities associated with this problem that is the main contribution of this chapter.

This chapter is organized as follows. In section 4.1 we discuss the problems of using a lazy approach to the multiplicative connectives, and how this affects other parts of the system. In the following two sections we prove that the resulting lazy system is sound and complete with respect to the standard one sided sequent calculus for linear logic. In section 4.4 we discuss the second problem and note that the observations in [6, 44] can be used to provide a heuristic that (partially) solves the second problem. We conclude the chapter with a brief discussion.

4.1 The Challenge of Being Lazy

The standard formulation of the inference rules for linear logic have a significant amount of nondeterminism. Some of this nondeterminism is unavoidable and does not create efficiency problems; for example the $\oplus$ rule. Consider however the $\otimes$ rule:

$$\frac{\Gamma \vdash F, \Delta \quad \Sigma \vdash G, \Xi}{\Gamma, \Sigma \vdash F \otimes G, \Delta, \Xi} \otimes$$

When this rule is applied bottom up, that is from root to leaves, we need to divide the formulae in the sequent between the two sub-branches. This division can be done in a number of ways which is exponential in the number of formulae. Hence a naïve implementation which backtracks through all possibilities is not feasible.

However it is possible to modify the $\otimes$ rule so the division of formulae between sub-branches is done efficiently. The key idea is that all formulae are passed to the first sub-branch. Unused formulae are returned and then passed to the second sub-branch. We refer to this mechanism as *lazy splitting* ([69] refer to this mechanism as an input output model of resource consumption). Consider as an example the proof of $p \parr (\mathbf{1} \otimes p^{\perp})$. We begin by using the $\parr$ rule to break off the p yielding $p, \mathbf{1} \otimes p^{\perp}$. We then process the left

side of the $\otimes$ rule – we pass it the rest of the context (i.e. the p). We are now trying to prove $p, \mathbf{1}$. This succeeds with the p as unused residue. The p is then passed to the right branch of the $\otimes$ rule: $p, p^\perp$ which is just an instance of the axiom rule.

The details of this solution however are not without a certain amount of subtlety – if care is not taken, soundness can be compromised. We begin by considering a fragment of Lygon excluding $\top$. Lazy splitting for this fragment is relatively straightforward; the real subtlety arises when $\top$ is re-introduced.

Consider the formula $(p \parr \mathbf{1}) \otimes p^\perp$. Clearly it is not provable as the following attempt shows

$$\dfrac{\dfrac{\vdash \mathbf{1}, p}{\vdash \mathbf{1} \parr p}\ \parr \qquad \vdash p^\perp}{\vdash (\mathbf{1} \parr p) \otimes p^\perp}\ \otimes$$

Consider now a naïve formulation of lazy splitting. Instead of a sequent of the form $\vdash \Gamma$ we use the notation $\Gamma \Rightarrow \Delta$ with the intention that the Δ are the excess formulae being returned unused. A successful proof cannot have excess resources and hence we require that its root be of the form $\Gamma \Rightarrow \emptyset$.

The standard sequent rules are modified as follows. The axiom rules are modified to return unused formulae. Note that since the treatment of the nonlinear region (δ) is unchanged from $\mathcal{L}$ we elide the "δ :" from the rules.

$$\dfrac{}{p, p^\perp, \Gamma \Rightarrow \Gamma}\ Ax \qquad \dfrac{}{\mathbf{1}, \Gamma \Rightarrow \Gamma}\ \mathbf{1}$$

The unary logical rules are modified to pass on returned formulae

$$\dfrac{F, G, \Gamma \Rightarrow \Delta}{F \parr G, \Gamma \Rightarrow \Delta}\ \parr$$

Finally, the $\otimes$ rule passes the excess from the left sub-branch into the right sub-branch.

$$\dfrac{F, \Gamma \Rightarrow \Delta \qquad G, \Delta \Rightarrow \Lambda}{F \otimes G, \Gamma \Rightarrow \Lambda}\ \otimes$$

Using these rules we find however that $(p \parr \mathbf{1}) \otimes p^\perp$ is derivable!

$$\dfrac{\dfrac{\dfrac{}{\mathbf{1}, p \Rightarrow p}\ \mathbf{1}}{\mathbf{1} \parr p \Rightarrow p}\ \parr \qquad \dfrac{}{p, p^\perp \Rightarrow \emptyset}\ Ax}{(\mathbf{1} \parr p) \otimes p^\perp \Rightarrow \emptyset}\ \otimes$$

The reason we have lost soundness is that not all formulae should be returnable. For lazy splitting to be valid we must not return formulae which were not present in the conclusion of the $\otimes$ rule. In the unsound proof above, p should not be returned by the **1** rule since it was introduced above the $\otimes$ rule.

To prevent this problem we must keep track of which formulae are returnable. We do this by tagging a returnable formula with a subscript 1. Untagged formulae must be consumed in the current branch of the proof. Tagged formulae may be returned from the current branch. The revised rules can only pass on unused formulae if they are tagged:

$$\frac{}{p, p^{\perp}, \Gamma_1 \Rightarrow \Gamma_1}\ Ax \qquad \frac{}{\mathbf{1}, \Gamma_1 \Rightarrow \Gamma_1}\ \mathbf{1}$$

The revised $\otimes$ rule

$$\frac{F, \Gamma_1 \Rightarrow \Lambda_1 \quad G, \Lambda \Rightarrow \Delta}{F \otimes G, \Gamma \Rightarrow \Delta}\ \otimes$$

marks all existing formulae as returnable and then passes them to the first sub-branch. Note that the formulae returned from the first sub-branch must have their tags removed before being passed to the second sub-branch. This ensures that formulae which are untagged in the conclusion of the $\otimes$ rule cannot be returned unused from the right premise and hence from the conclusion. As an example consider the formula $(p \parr (\mathbf{1} \otimes \mathbf{1})) \otimes p^{\perp}$ which is clearly unprovable in $\mathcal{L}$. If we neglect to have our lazy $\otimes$ rule strip away the tags before passing formulae to the right premise we lose soundness since the above formula has a derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p_1, \mathbf{1} \Rightarrow p_1}\ \mathbf{1} \quad \dfrac{}{p_1, \mathbf{1} \Rightarrow p_1}\ \mathbf{1}}{p, \mathbf{1} \otimes \mathbf{1} \Rightarrow p_1}\ \otimes}{p \parr (\mathbf{1} \otimes \mathbf{1}) \Rightarrow p_1}\ \parr \qquad \begin{array}{c}\vdots \\ p_1, p^{\perp} \Rightarrow\end{array}}{(p \parr (\mathbf{1} \otimes \mathbf{1})) \otimes p^{\perp} \Rightarrow}\ \otimes$$

Aside: A seemingly plausible alternative to having the $\otimes$ rule do the checking is to have the $\parr$ rule refuse to construct $F \parr G$ if either F or G occur in the residue. To see that this is unworkable consider the derivation of $(p \parr (\mathbf{1} \otimes \mathbf{1})) \otimes q^{\perp})$ where the program contains the single clause $p \leftarrow q$ (which could be encoded on the right hand side of the

sequent as $?(p^{\perp} \otimes q)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q_1, \mathbf{1} \Rightarrow q_1}\,\mathbf{1} \quad \dfrac{}{q_1, \mathbf{1} \Rightarrow q_1}\,\mathbf{1}}{q, \mathbf{1} \otimes \mathbf{1} \Rightarrow q_1}\,\otimes}{p, \mathbf{1} \otimes \mathbf{1} \Rightarrow q_1}\,Program}{p \parr (\mathbf{1} \otimes \mathbf{1}) \Rightarrow q_1}\,\parr \qquad \begin{array}{c}\vdots\\ q_1, q^{\perp} \Rightarrow\end{array}}{(p \parr (\mathbf{1} \otimes \mathbf{1})) \otimes q^{\perp} \Rightarrow}\,\otimes$$

Note that the $\parr$ rule has no way to detect that anything untoward is happening. This derivation is essentially the same as the previous one and is equally invalid. **End Aside.**

We now introduce an additional rule. The *Use* rule is used to claim a formula for use in the current sub-branch by stripping off the tags that allow it to be passed out of the current sub-branch.

$$\dfrac{F', \Gamma \Rightarrow \Delta}{F, \Gamma \Rightarrow \Delta}\,Use$$

where F' represents F with all tags removed.

EXAMPLE 1

$$\dfrac{\dfrac{\dfrac{}{p, p^{\perp} \Rightarrow}\,Ax}{p_1, p^{\perp} \Rightarrow}\,Use \quad \dfrac{}{\mathbf{1} \Rightarrow}\,\mathbf{1}}{p, p^{\perp} \otimes \mathbf{1} \Rightarrow}\,\otimes$$

Note that we have to allow nestable tags in order to handle nested occurrences of $\otimes$. The $\otimes$ rule adds and removes a single tag. We generalise the notation Γ_1 to any natural number.

EXAMPLE 2

The formula $((p \parr (\mathbf{1} \otimes p^{\perp})) \otimes q) \parr q^{\perp}$ *is provable*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1} \quad \dfrac{}{\vdash p, p^{\perp}}\,Ax}{\vdash p, \mathbf{1} \otimes p^{\perp}}\,\otimes}{\vdash p \parr (\mathbf{1} \otimes p^{\perp})}\,\parr \qquad \dfrac{}{\vdash q, q^{\perp}}\,Ax}{\vdash (p \parr (\mathbf{1} \otimes p^{\perp})) \otimes q, q^{\perp}}\,\otimes}{\vdash ((p \parr (\mathbf{1} \otimes p^{\perp})) \otimes q) \parr q^{\perp}}\,\parr$$

Using lazy splitting we obtain the following proof which uses "nested" tags.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{1}, p_1, q_2^\perp \Rightarrow p_1, q_2^\perp}\,\mathbf{1} \quad \dfrac{}{p, p^\perp, q_1^\perp \Rightarrow q_1^\perp}\,Ax}{p, \mathbf{1} \otimes p^\perp, q_1^\perp \Rightarrow q_1^\perp}\,\otimes}{p \parr (\mathbf{1} \otimes p^\perp), q_1^\perp \Rightarrow q_1^\perp}\,\parr \quad \dfrac{}{q, q^\perp \Rightarrow}\,Ax}{(p \parr (\mathbf{1} \otimes p^\perp)) \otimes q, q^\perp \Rightarrow}\,\otimes}{((p \parr (\mathbf{1} \otimes p^\perp)) \otimes q) \parr q^\perp \Rightarrow}\,\parr$$

EXAMPLE 3

The similar formula $((q^\perp \parr (\mathbf{1} \otimes p^\perp)) \otimes q) \parr p$ is not provable. That the lazy splitting proof fails is dependent on the use of nested tags.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q_1^\perp, \mathbf{1}, p_2 \Rightarrow q_1^\perp, p_2}\,\mathbf{1} \quad p^\perp, p_1, q^\perp \Rightarrow ?}{q^\perp, \mathbf{1} \otimes p^\perp, p_1 \Rightarrow ?}\,\otimes}{q^\perp \parr (\mathbf{1} \otimes p^\perp), p_1 \Rightarrow ?}\,\parr \quad q, ? \Rightarrow \Gamma}{(q^\perp \parr (\mathbf{1} \otimes p^\perp)) \otimes q, p \Rightarrow \Gamma}\,\otimes}{((q^\perp \parr (\mathbf{1} \otimes p^\perp)) \otimes q) \parr p \Rightarrow \Gamma}\,\parr$$

Note that in the leaf $p^\perp, p_1, q^\perp$ we cannot return the q since it is not tagged.

We have seen the rules for $\otimes$, $\parr$ and $\mathbf{1}$. The rules for $\oplus$, $?$, $\perp$, $\forall$, $\exists$ are similar to $\parr$. The Ax rule is similar to $\mathbf{1}$. When we apply ! we must ensure that there are no other (linear) formulae. Thus we force all excess formulae to be returned. In a way ! is similar to the Ax and $\mathbf{1}$ rules.

$$\dfrac{F \Rightarrow}{!F, \Gamma_n \Rightarrow \Gamma_n}\,!$$

EXAMPLE 4

The formula $((!p) \otimes (p \oplus \mathbf{1})) \parr p^\perp$ is unprovable:

$$\dfrac{\dfrac{\dfrac{p \Rightarrow}{!p, p_1^\perp \Rightarrow p_1^\perp}\,! \quad \dfrac{\dfrac{}{p, p^\perp}\,Ax}{p \oplus \mathbf{1}, p^\perp \Rightarrow}\,\oplus}{(!p) \otimes (p \oplus \mathbf{1}), p^\perp \Rightarrow}\,\otimes}{((!p) \otimes (p \oplus \mathbf{1})) \parr p^\perp \Rightarrow}\,\parr$$

The other logical rule which is affected by the lazy splitting mechanism is the $\&$ rule:

$$\dfrac{F, \Gamma \Rightarrow \Xi \quad G, \Gamma \Rightarrow \Sigma}{F \,\&\, G, \Gamma \Rightarrow \Delta}\,\&$$

This rule has the constraint that $\Xi = \Sigma = \Delta$. This enforces the constraint in the non-lazy $\&$ rule that the two sub-branches have the same context. Note that we prefer to have an explicit constraint since in the next section this constraint will need to be modified.

EXAMPLE 5

We now consider a slightly larger example. Consider the program $p \leftarrow q$. This is translated as $!(q \multimap p)$ on the left hand side. Expressing $\multimap$ in terms of $\parr$ we obtain $!(q^\perp \parr p)$. Since we work with a single sided sequent calculus we negate the formulae so we can put it on the right. Negating the formula yields $?(p^\perp \otimes q)$. Our query is $(p \,\&\, \mathbf{1}) \parr (q^\perp \oplus \perp)$.

The proof illustrates the handling of nonlinear formulae and the $\&$ rule. Recall that the notation $\delta : \Gamma \Rightarrow \Delta$ represents $?\delta, \Gamma \Rightarrow \Delta$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p^\perp \otimes q : p^\perp, p, (q^\perp \oplus \perp)_1 \Rightarrow (q^\perp \oplus \perp)_1}\, Ax}{p^\perp \otimes q : p^\perp, p_1, (q^\perp \oplus \perp)_1 \Rightarrow (q^\perp \oplus \perp)_1}\, Use \quad \dfrac{\dfrac{}{p^\perp \otimes q : q, q^\perp \Rightarrow}\, Ax}{p^\perp \otimes q : q, q^\perp \oplus \perp \Rightarrow}\, \oplus}{p^\perp \otimes q : p^\perp \otimes q, p, q^\perp \oplus \perp \Rightarrow}\, \otimes}{p^\perp \otimes q : p, q^\perp \oplus \perp \Rightarrow}\, ?D \qquad \dfrac{\dfrac{\dfrac{}{p^\perp \otimes q : \mathbf{1} \Rightarrow}\, \mathbf{1}}{p^\perp \otimes q : \mathbf{1}, \perp \Rightarrow}\, \perp}{p^\perp \otimes q : \mathbf{1}, q^\perp \oplus \perp \Rightarrow}\, \oplus}{p^\perp \otimes q : p \,\&\, \mathbf{1}, q^\perp \oplus \perp \Rightarrow}\, \&$$

$$\dfrac{:?(p^\perp \otimes q), p \,\&\, \mathbf{1}, q^\perp \oplus \perp \Rightarrow}{:?(p^\perp \otimes q), (p \,\&\, \mathbf{1}) \parr (q^\perp \oplus \perp) \Rightarrow}\, \parr$$

The rules we have seen form the working core of the lazy splitting inference rules for the fragment of the logic excluding $\top$. Looking at a complete proof and seeing sequents of the form $a_1^\perp, b_1, a \Rightarrow b_1$ the reader may be left with the feeling that there is still some magic at work. This is not so; when a proof is being constructed bottom up, the right-hand side of the arrow ($\Rightarrow$) is left unbound on the way up and determined at the leaves.

EXAMPLE 6

The construction of a proof for $p^\perp, q^\perp, p \otimes q$ goes through the following steps (where capital letters represent meta-variables)

1. *The root of the proof is*
$$p^\perp, q^\perp, p \otimes q \Rightarrow Y$$
2. *We apply the $\otimes$ rule to obtain*
$$\dfrac{p_1^\perp, q_1^\perp, p \Rightarrow X_1 \quad q, X \Rightarrow Y}{p^\perp, q^\perp, p \otimes q \Rightarrow Y}\, \otimes$$

3. *We realise that we need $p^\perp$ in order to apply the axiom rule and so decide to Use it.*

$$\dfrac{\dfrac{p^\perp, q_1^\perp, p \Rightarrow X_1}{p_1^\perp, q_1^\perp, p \Rightarrow X_1}\, Use \qquad q, X \Rightarrow Y}{p^\perp, q^\perp, p \otimes q \Rightarrow Y}\, \otimes$$

4. *We now apply the axiom rule. The formula $q_1^\perp$ is returnable excess so we return it. This binds X.*

$$\dfrac{\dfrac{\dfrac{}{p^\perp, q_1^\perp, p \Rightarrow q_1^\perp}\, Ax}{p_1^\perp, q_1^\perp, p \Rightarrow q_1^\perp}\, Use \qquad q, q^\perp \Rightarrow Y}{p^\perp, q^\perp, p \otimes q \Rightarrow Y}\, \otimes$$

5. *We have finished the left sub-branch of the $\otimes$ rule and now look at the right sub-branch. We realise that we can immediately apply the axiom rule. Since there are no excess formulae, Y is bound to empty.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p^\perp, q_1^\perp, p \Rightarrow q_1^\perp}\, Ax}{p_1^\perp, q_1^\perp, p \Rightarrow q_1^\perp}\, Use \qquad \dfrac{}{q, q^\perp \Rightarrow}\, Ax}{p^\perp, q^\perp, p \otimes q \Rightarrow}\, \otimes$$

Adding $\top$

The rules presented so far form a sound and complete collection of inference rules for the fragment of the logic excluding $\top$. These rules manage resources deterministically.

The lazy splitting version of $\top$ involves a significant amount of subtlety and has implications for the $\&$ rule which ends up becoming fairly complex.

Consider the standard inference

$$\dfrac{\dfrac{}{\vdash \top, \Gamma}\, \top \qquad \begin{array}{c}\vdots\\ \vdash G, \Sigma\end{array}}{\begin{array}{c}\vdash \top \otimes G, \Gamma, \Sigma\\ \vdots\end{array}}\, \otimes$$

The rule for $\top$ simply consumes all formulae. Consider however the lazy splitting ver-

sion of the above inference

$$\dfrac{\dfrac{}{\top, \Gamma_1, \Sigma_1 \Rightarrow \Sigma_1}\ \top \qquad \begin{array}{c}\vdots\\ G, \Sigma \Rightarrow\end{array}}{\begin{array}{c}\top \otimes G, \Gamma, \Sigma \Rightarrow\\ \vdots\end{array}}\ \otimes$$

The application of the $\top$ rule must somehow know which formulae are not to be consumed since they will be required elsewhere in the proof.

Note that the lazy splitting version of the $\top$ rule has to divide formulae between $\top$ and the rest of the proof. This can be done in a number of ways which is exponential in the number of formulae.

Although this sounds similar to the problem in the original $\otimes$ rule there are two differences. Firstly, it is not possible to nest $\top$s, so a single non-nestable tag will suffice. Secondly and more significantly, the direction is opposite – the first sub-branch of the $\otimes$ rule returns formulae which are unconsumed and which must be consumed by the second sub-branch. The $\top$ rule on the other hand will consume all formulae by tagging them appropriately then passing them on. The formulae passed on *have been consumed* by $\top$ but – in case they should not have been – can be "unconsumed".

We shall use a superscript question mark (e.g. $\Gamma^?$) to tag formulae which have been consumed by $\top$ and which can be "unconsumed" if necessary. Note that untagged formulae are simply consumed by the $\top$ rule since they cannot be returned for use elsewhere.

$$\dfrac{}{\top, \Delta, \Gamma_1 \Rightarrow \Gamma_1^?}\ \top$$

One might be tempted to define $-^?$ in terms of existing connectives, *viz.* $F^? = F \oplus \bot$. There is however a subtle but important difference between the two. $F^?$ represents a formula which may have been consumed by $\top$, thus F is either consumed — in which case it must not be used elsewhere — or unconsumed — in which case it must be consistently accounted for.

$F \oplus \bot$ on the other hand allows for some parts of the proof to choose F and other parts to choose $\bot$. Consider $((\top \otimes (p \,\&\, \mathbf{1})) \parr p^\bot)$ which is not provable in the standard

system as shown below

$$\dfrac{\dfrac{}{\vdash \top}\top \quad \dfrac{\dfrac{\dfrac{}{\vdash p, p^\perp}Ax \quad \vdash \mathbf{1}, p^\perp}{\vdash p \,\&\, \mathbf{1}, p^\perp}\,\&}{\vdash \top \otimes (p \,\&\, \mathbf{1}), p^\perp}\otimes}{\vdash (\top \otimes (p \,\&\, \mathbf{1})) \parr p^\perp}\parr \qquad \dfrac{\dfrac{}{\vdash \top, p^\perp}\top \quad \dfrac{\vdash p \quad \dfrac{}{\vdash \mathbf{1}}\mathbf{1}}{\vdash p \,\&\, \mathbf{1}}\,\&}{\vdash \top \otimes (p \,\&\, \mathbf{1}), p^\perp}\otimes}{\vdash (\top \otimes (p \,\&\, \mathbf{1})) \parr p^\perp}\parr$$

Yet if we use $p \oplus \perp$ in place of $p^?$ it has a derivation:

$$\dfrac{\dfrac{}{\top, p^\perp{}_1 \Rightarrow (p^\perp \oplus \perp)_1}\top \quad \dfrac{\dfrac{\dfrac{}{p, p^\perp \Rightarrow}Ax}{p, p^\perp \oplus \perp \Rightarrow}\oplus \quad \dfrac{\dfrac{\dfrac{}{\mathbf{1} \Rightarrow}\mathbf{1}}{\mathbf{1}, \perp \Rightarrow}\perp}{\mathbf{1}, p^\perp \oplus \perp \Rightarrow}\oplus}{p \,\&\, \mathbf{1}, p^\perp \oplus \perp \Rightarrow}\,\&}{\top \otimes (p \,\&\, \mathbf{1}), p^\perp \Rightarrow}\otimes}{(\top \otimes (p \,\&\, \mathbf{1})) \parr p^\perp \Rightarrow}\parr$$

In addition to pointing out the difference between $F^?$ and $F \oplus \perp$ this suggests that the $\top$ rule cannot be simply added to the system since maintaining the consistent usage of formulae of the form $F^?$ requires a measure of global information. Thus, rather than having the axiom rules delete consumed formulae which have turned out to be unneeded

$$\dfrac{}{p, p^\perp, \Delta^?, \Gamma_1 \Rightarrow \Gamma_1}\,Ax'$$

we must return the consumed formulae which were unneeded, so that we can check that different additive branches agree on which consumed formulae have had to be unconsumed.

$$\dfrac{}{p, p^\perp, \Delta^?, \Gamma_1 \Rightarrow \Gamma_1, \Delta^?}\,Ax$$

If we do not return the unused formulae then we lose soundness, and we can derive unprovable sequents such as $p, q, \top \otimes (p^\perp \,\&\, q^\perp)$

EXAMPLE 7

$$\dfrac{\dfrac{}{p_1, q_1, \top \Rightarrow p_1^?, q_1^?}\top \quad \dfrac{\dfrac{\dfrac{}{p, q^?, p^\perp \Rightarrow}Ax'}{p^?, q^?, p^\perp \Rightarrow}Use \quad \dfrac{\dfrac{}{p^?, q, q^\perp \Rightarrow}Ax'}{p^?, q^?, q^\perp \Rightarrow}Use}{p^?, q^?, p^\perp \,\&\, q^\perp \Rightarrow}\,\&}{p, q, \top \otimes (p^\perp \,\&\, q^\perp) \Rightarrow}\otimes$$

We thus use Ax rather than Ax'. In order to do this we must modify the unary rules to pass on the returned formulae of the form $F^?$. We also need to have a look at the two binary rules — $\otimes$ and $\&$.

The $\otimes$ rule does the usual passing from the left sub-branch to the right. The only interesting point is that it is possible for the left sub-branch to return formulae which do not have a $-_n$ tag. These formulae are of the form $F^?$ and they must be returned by the conclusion of the $\otimes$ inference without being passed to the right sub-branch. If this is not done, soundness is compromised. Consider the formula $((\top \otimes \mathbf{1}) \parr p) \otimes p^\perp$ which is clearly unprovable. If we use a lazy $\otimes$ rule which passes these formulae to the right premise before returning them then there is a derivation of this formula:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\top, p_1 \Rightarrow p_1^?}\top \quad \dfrac{}{\mathbf{1}, p^? \Rightarrow p^?}\mathbf{1}}{\top \otimes \mathbf{1}, p \Rightarrow p^?}\otimes}{(\top \otimes \mathbf{1}) \parr p \Rightarrow p^?}\parr \qquad \dfrac{\dfrac{}{p, p^\perp \Rightarrow}Ax}{p^?, p^\perp \Rightarrow}Use}{((\top \otimes \mathbf{1}) \parr p) \otimes p^\perp \Rightarrow}\otimes$$

The intuition behind this is that a formula of the form F_1 can be returned from the current branch for use elsewhere. The same is not true of formulae of the form $F^?$ - we cannot pass them on for use elsewhere, they must be propagated down the proof. These formulae ($F^?$) are returned only to enable the $\&$ rule to enforce consistent consumption of formulae by $\top$ rules in both subproofs.

We now consider the $\&$ rule. As we shall see, it is possible for the residues of the two premises of a $\&$ inference to be different. We need to determine when such a disagreement is harmless and what the residue of the inference's conclusion should be in these situations.

The presence of a formula of the form $p_n^?$ in the residue of a sequent indicates that the proof of the sequent has consumed p but that it could equally well be unconsumed. For example if $p_1, \Gamma \Rightarrow p_1^?$ is provable then $\vdash \Gamma$ and $\vdash \Gamma, p$ are both provable.

Consider now a premise of the $\&$ rule which has this property. The conclusion of the $\&$ inference will have the property of being provable with or without p only if *both* premises have the property.

In the following example the right premise has the property that both $\vdash \top$ and $\vdash \top, p$ are provable. However the left premise does not have this property as $\vdash p, p^\perp$ is provable

but $\vdash p^{\perp}$ is not. As a result, the conclusion of the $\&$ inference does not have this property as $\vdash p^{\perp} \,\&\, \top, p$ is provable but $\vdash p^{\perp} \,\&\, \top$ is not.

EXAMPLE 8

$$\dfrac{\dfrac{\dfrac{}{p^{\perp}, p \Rightarrow}\, Ax}{p^{\perp}, p_1 \Rightarrow}\, Use \qquad \dfrac{}{\top, p_1 \Rightarrow p_1^{?}}\, \top}{p^{\perp} \,\&\, \top, p_1 \Rightarrow X}\, \&$$

This sub-proof occurs in the proof of $((p^{\perp} \,\&\, \top) \otimes \mathbf{1}) \parr p$ and hence needs to be derivable. Intuitively, since the conclusion of the inference needs the p its residue should be empty – this prevents a different proof branch from attempting to use p. Note that if the right premise did not have the property discussed then the inference would be invalid.

Consider now the following example, which illustrates a $\&$ inference where the residues of the premises disagree on a formula of the form $F^{?}$.

EXAMPLE 9

$$\dfrac{\dfrac{\dfrac{}{\top, p_1 \Rightarrow p_1^{?}}\, \top \qquad \dfrac{}{\mathbf{1}, p^{?} \Rightarrow p^{?}}\, \mathbf{1}}{\top \otimes \mathbf{1}, p \Rightarrow p^{?}}\, \otimes \qquad \dfrac{}{p^{\perp}, p \Rightarrow}\, Ax}{(\top \otimes \mathbf{1}) \,\&\, p^{\perp}, p \Rightarrow}\, \&$$

The presence of $p^{?}$ in $\Gamma \Rightarrow p^{?}$ indicates that $p^{?}$ was not unconsumed. The proof in example 9 is valid since we can force the left premise to consume the p. This can be done by having the residue of the conclusion be empty which rules out the possibility of another proof branch unconsuming the p.

As before, the reconciliation of the differences in the premise's residues requires that we know whether the premise with an excess in its residue is able to accept the excess. The local information present at the point of the $\&$ inference is insufficient to determine this. As examples 8 and 9 indicate, whether a sub-prof can accept extra formulae is determined by its structure.

We term (sub-)proofs that can accept additional formulae $\top$-like (since such proofs must contain an occurrence of the $\top$ inference). In a $\&$ inference, whenever the premise's residues disagree we can reconcile the disagreement *if* the premise with the larger residue

is flexible – that is, if it is the conclusion of a $\top$-like proof. If we have a disagreement and no reconciliation is possible since the premise in questions is not $\top$-like (as occurs in example 7) then the $\&$ inference fails.

In example 8 the disagreement is that the right premise has more in the residue than the left premise. Since the right premise is $\top$-like we can reconcile by forcing the right premise to consume p – that is, it can not offer the rest of the proof the possibility of unconsuming p since the left premise of the $\&$ inference will fail if this is done. The reconciliation can simply be done by having the residue of the $\&$ rule be the *intersection* of the residues of its premises. In example 9 the left premise is $\top$-like and reconciliation is possible. The residue of the conclusion is empty.

To be able to check whether reconciliation is possible we add tags to enable us to track whether a given sequent is the conclusion of a $\top$-like proof. We tag a sequent with $/true$ if the proof is $\top$-like and with $/false$ if it is not. The $\top$ rule is tagged $/true$, the Ax and **1** rules are tagged $/false$. The unary rules pass on the tag. A $\otimes$ rule's conclusion is $\top$-like if at least one of its premises is. So, if the tags on the two premises are $/x$ and $/y$ the tag on the conclusion is $/x \vee y$. The conclusion of a $\&$ inference is $\top$-like if *both* premises are. So, if the tags on the two premises are $/x$ and $/y$ the tag on the conclusion is $/x \wedge y$.

The proof in example 8 is valid since the right premise would be labelled with $/true$. The left premise of the $\&$ inference in example 9 would also be labelled with $/true$.

One minor wrinkle is that while intersection must agree on the formulae, it may not agree on their tags. Specifically, the formulae in one premise may have a $-^?$ tag in addition to some $-_n$ tags whereas those in the other premise may not have the $-^?$ tag:

EXAMPLE 10

What value should we give to X in the following proof?

$$\dfrac{\dfrac{}{p_1, \top \Rightarrow p_1^?}\,\top \qquad \dfrac{}{p_1, \mathbf{1} \Rightarrow p_1}\,\mathbf{1}}{p_1, \top \,\&\, \mathbf{1} \Rightarrow X}\,\&$$

The solution is to pass on the $-^?$ tag if both premises have it and to strip it off if only one of the premises has it. This can be viewed as a sort of unification over tags.

The final $\&$ rule in all its glory may be found in figure 4.1. In calculating the conclusion's residue it uses intersection on multisets of formulae and an operation "$mintag$" which given two tagged formulae removes the $-^?$ tag if exactly one of the formulae has the tag and leaves the formulae unchanged otherwise.

Our final example shows the $\&$ and $\top$ rules in action. Note that the r in the right premise of the $\&$ rule is excess; however the premise in question is tagged with $/true$ so the proof is valid.

EXAMPLE 11

$$\dfrac{\dfrac{}{\top, p_1, q_1, r_1 \Rightarrow p_1^?, q_1^?, r_1^?/true}\top \quad \dfrac{\Pi_1 \quad \Pi_2}{(r^\perp \otimes p^\perp) \,\&\, (\top \otimes p^\perp), p^?, q^?, r^? \Rightarrow q^?/false}\,\&}{\top \otimes ((r^\perp \otimes p^\perp) \,\&\, (\top \otimes p^\perp)), p, q, r \Rightarrow q^?/true}\otimes$$

where Π_1 *is*

$$\dfrac{\dfrac{\dfrac{}{r^\perp, p_1^?, q_1^?, r \Rightarrow p_1^?, q_1^?/false}Ax}{r^\perp, p_1^?, q_1^?, r_1^? \Rightarrow p_1^?, q_1^?/false}Use \quad \dfrac{\dfrac{}{p^\perp, p, q^? \Rightarrow q^?/false}Ax}{p^\perp, p^?, q^? \Rightarrow q^?/false}Use}{r^\perp \otimes p^\perp, p^?, q^?, r^? \Rightarrow q^?/false}\otimes$$

and Π_2 *is*

$$\dfrac{\dfrac{}{\top, p_1^?, q_1^?, r_1^? \Rightarrow p_1^?, q_1^?, r_1^?/true}\top \quad \dfrac{\dfrac{}{p^\perp, p, q^?, r^? \Rightarrow q^?, r^?/false}Ax}{p^\perp, p^?, q^?, r^? \Rightarrow q^?, r^?/false}Use}{\top \otimes p^\perp, p^?, q^?, r^? \Rightarrow q^?, r^?/true}\otimes$$

DEFINITION 29 (NOTATION)

In the remainder of this chapter we shall need to refer to different multisets of formulae based on their tags. The following notation is used:

- *Capital letters (A, B, C, D) are used to represent multisets of formulae.*
- *We use subscripts numbers and a possible superscript "?" to indicate which tags characterise the multiset. For example A_2 consists of all formulae of the form p_2.*
- *We assume that n is the maximum number of tags occurring in a given proof. A common idiom is $A, A_1, \ldots, A_n$ which covers all formulae in a sequent which do not have $-^?$ tags.*

- *We use a t superscript to represent both formulae with a $-^?$ tag and formulae without.* $A^t_j \stackrel{\text{def}}{=} A_j \cup A^?_j$

- *We use an x subscript to represent all subscript tags from 1 upwards.* $A_x \stackrel{\text{def}}{=} A_1 \cup A_2 \cup \ldots \cup A_n$

- *For uniformity we shall sometimes write A_0. This is equivalent to A.*

- *We shall sometimes need to refer to a multiset and modify its tags. A* superscript *modifier is used to denote this. For example if A_2 is the multiset $\{p_2, (q \,\&\, r)_2\}$ then $A_2^{+?}$ is the multiset $\{p_2^?, (q \,\&\, r)_2^?\}$ and A_2^{+1} is the multiset $\{p_3, (q \,\&\, r)_3\}$.*

EXAMPLE 12

Suppose we have a sequent containing the following formulae:

$$p, q_1, r^?, t_2^?, s_1, \mathbf{1}_4$$

then the following hold:

$$A = p \qquad A^{+1} = p_1$$

$$A_1 = q_1, s_1 \qquad A_1^{+1} = q_2, s_2$$

$$A^? = r^? \qquad A^{?+1} = r_1^?$$

$$A_2^? = t_2^? \qquad A_2^{?-1} = t_1^?$$

$$A_1^? = \emptyset \qquad A_1^{?+1} = \emptyset$$

$$A^?_{1+1} = A_2^? = t_2^?$$

$$A_0^t = A^t = A \cup A^? = p, r^?$$

$$A_0^{t+?} = p^?, r^?$$

$$A_x = q_1, s_1, \mathbf{1}_4 \qquad A_x^? = t_2^?$$

$$A_x^t = A_x \cup A_x^? = q_1, s_1, \mathbf{1}_4, t_2^?$$

Figure 4.1 The Final System ($\mathcal{M}$)

We define $-'$ to remove all tags. Thus $(F_x^t)' = F$. We shall also need to define $mintag$:

$mintag(x, y) \stackrel{\text{def}}{=} x$, if x and y are identical formulae with the same tags

$mintag(F_n^?, F_n) \stackrel{\text{def}}{=} F_n$

$mintag(F_n, F_n^?) \stackrel{\text{def}}{=} F_n$

The $\&$ rule is only applicable if the following four side conditions hold:

1. $\Sigma_x^t \stackrel{\text{def}}{=} \{mintag(F, G) | F \in \Pi_x^t \wedge G \in \Xi_x^t \wedge F^{+?} = G^{+?}\}$
2. $\Sigma^? \stackrel{\text{def}}{=} \{F | F \in \Pi^? \cup \Xi^?\}$. We shall sometimes refer to $\Sigma \stackrel{\text{def}}{=} \Sigma_x^t \cup \Sigma^?$.
3. If x is $/false$ then $(\Pi_x^t, \Pi^?) \subseteq (\Sigma_x^t, \Sigma^?)$.
4. If y is $/false$ then $(\Xi_x^t, \Xi^?) \subseteq (\Sigma_x^t, \Sigma^?)$.

$$\frac{}{\delta : p, p^\perp, A_x^t, B^? \Rightarrow A_x^t, B^?/false}\ \text{Ax} \qquad \frac{}{\delta : \mathbf{1}, A_x^t, B^? \Rightarrow A_x^t, B^?/false}\ \mathbf{1}$$

$$\frac{}{\delta : \top, \Gamma, A_x^t, B^? \Rightarrow A_x^{t+?}, B^?/true}\ \top \qquad \frac{\delta : \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : \perp, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}\ \perp$$

$$\frac{\delta : F, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : F \oplus G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}\ \oplus_1 \qquad \frac{\delta : G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : F \oplus G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}\ \oplus_2$$

$$\frac{\delta : F \Rightarrow D^?/x}{\delta : !F, A_x^t, B^? \Rightarrow A_x^t, B^?/false}\ ! \qquad \frac{\delta : F, G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : F \parr G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}\ \parr$$

$$\frac{\delta : F[t/x], \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : \exists x F, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}\ \exists \qquad \frac{\delta : F[y/x], \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : \forall x F, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}\ \forall$$

where y is not free in Γ

$$\frac{\delta, F : \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : ?F, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}\ ? \qquad \frac{F, \delta : F, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{F, \delta : \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}\ ?D$$

$$\frac{\delta : F', \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : F, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}\ Use$$

$$\frac{\delta : F, \Gamma^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1} \Rightarrow \Delta^{+1}, \Pi_x^{t+1}, \Pi^{?+1}, \Xi^?/x \quad \delta : G, \Delta, \Pi_x^t, \Pi^? \Rightarrow \Sigma_x^t, \Sigma^?/y}{\delta : F \otimes G, \Gamma, \Gamma_x^t, \Gamma^? \Rightarrow \Sigma_x^t, \Sigma^?, \Xi^?/x \vee y}\ \otimes$$

$$\frac{\delta : F, \Gamma, \Gamma_x^t, \Gamma^? \Rightarrow \Pi_x^t, \Pi^?/x \quad \delta : G, \Gamma, \Gamma_x^t, \Gamma^? \Rightarrow \Xi_x^t, \Xi^?/y}{\delta : F \,\&\, G, \Gamma, \Gamma_x^t, \Gamma^? \Rightarrow \Sigma_x^t, \Sigma^?/x \wedge y}\ \&$$

The Rules

Having travelled through the evolution of the deterministic rules we are now in a position to collect and categorise the result. This categorisation will be of use in induction proofs in the following sections. By recognising the similarities among groups of rules we can use induction over the rule groups rather than the rules thus reducing the number of cases.

The rules ($\mathcal{M}$, given in figure 4.1) fall broadly into four groups. The first group comprises rules which are essentially unchanged. For example the $\mathcal{L}$ rule for $\parr$ is

$$\frac{\delta : F, G, \Gamma}{\delta : F \parr G, \Gamma} \parr$$

and the $\mathcal{M}$ rule is

$$\frac{\delta : F, G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : F \parr G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x} \parr$$

This rule (and the others in the first group) simply pass along the tagged formulae and the returned residue. In this group are the rules for $\bot$, $\oplus$, $\parr$, $\exists$, $\forall$, $?$ and $?D$.

The second group of rules comprises the axioms. These rules instantiate the residue to the tagged formulae and the tag to $/false$. Note that we include ! as an axiom since it behaves like one.

$$\frac{}{\delta : p, p^{\perp}, A_x^t, B^? \Rightarrow A_x^t, B^?/false} \text{ Ax} \qquad \frac{\delta : a \Rightarrow D^?/x}{\delta : !a, A_x^t, B^? \Rightarrow A_x^t, B^?/false} \; !$$

This group comprises Ax , **1** and !.

Both groups of rules so far have a simple relationship to the $\mathcal{L}$ rules. The third group of rules comprises the remaining rules in $\mathcal{L}$ which are significantly changed in $\mathcal{M}$. These rules are $\top$, $\&$ and $\otimes$.

Finally the *Use* rule does not have a counterpart in $\mathcal{L}$ and we place this rule into a group of its own.

We shall see later on (theorem 119) that this system manages resources deterministically, and as such it can be implemented efficiently.

4.2 Soundness

In this section we prove that our deterministic system ($\mathcal{M}$) is sound with respect to the one-sided sequent calculus for linear logic ($\mathcal{L}$).

The basic idea behind the soundness proof is to treat $C_x^t, D^?$ in the $\mathcal{M}$ sequent $\delta : \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?$ as excess and "subtract" them from $A_x^t, B^?$ to obtain the $\mathcal{L}$ sequent $\delta : \Gamma, ((A, B) - (C, D))$. The core of the soundness proof is an algorithm which converts an $\mathcal{M}$ proof to an $\mathcal{L}$ proof. This algorithm will be shown to map *any* $\mathcal{M}$ proof into an $\mathcal{L}$ proof; thus showing the soundness of $\mathcal{M}$ with respect to $\mathcal{L}$.

This approach is complicated by the fact that the subtraction used is not necessarily valid - although we show that C_x^t is a sub-multiset of A_x^t, this does not hold for $D^?$ and $B^?$. We observe that whenever $D^? \not\subseteq B^?$

1. A $\top$ rule is involved (lemma 100)

2. The sequent is labelled with $/true$ (lemma 101)

We then define a notion of *backpatching*. Intuitively, wherever $D^? \not\subseteq B^?$ we augment the sequent by adding $D^?$ to the left of the $\Rightarrow$ thus fixing the imbalance. In order for the augmented sequent to fit into the proof we need to *propagate*. Propagation simply adds the same formulae to the premises until a $\top$ rule is reached. We show that backpatching and propagation

1. Always succeed (lemma 103)

2. Produce a proof where $D^?$ is a sub-multiset of $B^?$ (lemma 106)

For the rest of this section we will view sequents as consisting of the following groups of formulae:

$$\delta : A_0, A_1, \ldots, A_n, B_0^?, B_1^?, \ldots, B_n^? \Rightarrow C_1, \ldots, C_n, D_1^?, \ldots, D_n^?, D_0^?$$

For the purposes of induction arguments we shall use induction over the rules **1** , !, $\parr$, $\top$, $\otimes$, $\&$ and *Use*. The other rules are similar to either **1** or $\parr$. We write $X \subseteq Y$ to indicate that X is either equal to or is a sub-multiset of Y. This comparison ignores tags; that is $X \subseteq Y$ iff $X' \subseteq Y'$ where $(F_x^t)' \stackrel{\text{def}}{=} F$.

LEMMA 97

In all sequents occurring in $\mathcal{M}$ proofs we have that $C_i, D_i \subseteq A_i, B_i$ and $C_i \subseteq A_i$ where $i \geq 1$.

Proof: *We use induction on the structure of the proof.*

The rules 1 and Ax:

These rules are base cases for the induction. In both cases the C_i are the A_i and the D_i are the B_i. Hence the hypothesis trivially holds.

The $\top$ rule:

We can write the $\top$ inference as follows

$$\frac{}{\delta : \top, \Gamma, A_1, \ldots A_n, B^?, B_1^?, \ldots, B_n^? \Rightarrow A_1^{+?}, \ldots, A_n^{+?}, B^?, B_1^?, \ldots, B_n^?} \top$$

Thus, the C_i are empty, as all formulae in the residue have a $-^?$ tag; and the D_i are the union of the corresponding A_i and B_i. Hence $C_i = \emptyset \subseteq A_i$ and $C_i, D_i = A_i^{+?}, B_i \subseteq A_i, B_i$ as desired.

The $\otimes$ rule:

The desired property holds for the premise by the induction hypothesis. Observe that the differences between the premise and the conclusion do not involve tagged formulae, and hence the desired property holds in the conclusion of the inference.

The *Use* rule:

The Use rule can be written as follows

$$\frac{F, A_1, \ldots A_n, B^?, B_1^?, \ldots, B_n^? \Rightarrow C_1, \ldots, C_n, D^?, D_1^?, \ldots, D_n^?}{F_x^t, A_1, \ldots A_n, B^?, B_1^?, \ldots, B_n^? \Rightarrow C_1, \ldots, C_n, D^?, D_1^?, \ldots, D_n^?} Use$$

There are a number of cases:

1. *The formula being Used is tagged $F^?$. In this case there is no effect on the A_i or B_i and since, by the induction hypothesis, the property holds for the premise, it also holds for the conclusion of the inference.*

2. *The formula being Used is tagged $F_i^?$. In this case the induction hypothesis applied to the premise tells us that $C_i, D_i \subseteq A_i, B_i$ and $C_i \subseteq A_i$. The properties that we wish to show hold in the conclusion are $C_i, D_i \subseteq A_i, B_i, F_i^?$ and $C_i \subseteq A_i$ and these follow directly from the induction hypothesis.*

3. *The formula being Used is tagged F_i. In this case the induction hypothesis applied to the premise tells us that $C_i, D_i \subseteq A_i, B_i$ and $C_i \subseteq A_i$. The properties that we wish to show hold in the conclusion are $C_i \subseteq A_i, F_i$ and $C_i, D_i \subseteq A_i, F_i, B_i$ and these follow directly from the induction hypothesis.*

The $\&$ rule:

The left premise is

$$\delta : A_0, A_1, \ldots, A_n, B_0^?, B_1^?, \ldots, B_n^? \Rightarrow C_1, \ldots, C_n, D_1^?, \ldots, D_n^?, D_0^?$$

and the right premise is

$$\delta : A_0, A_1, \ldots, A_n, B_0^?, B_1^?, \ldots, B_n^? \Rightarrow E_1, \ldots, E_n, F_1^?, \ldots, F_n^?, F_0^?$$

By the induction hypothesis these satisfy the constraints

$$C_i, D_i \subseteq A_i, B_i$$

$$C_i \subseteq A_i$$

$$E_i, F_i \subseteq A_i, B_i$$

$$E_i \subseteq A_i$$

The conclusion of the $\&$ inference is

$$\delta : A_0, A_1, \ldots, A_n, B_0^?, B_1^?, \ldots, B_n^? \Rightarrow G_1, \ldots, G_n, H_1^?, \ldots, H_n^?, H_0^?$$

The first property we wish to show is that $G_i, H_i \subseteq A_i, B_i$. We have that $G_i, H_i \subseteq C_i, D_i$ since – by the definition of the $\&$ rule – G_i, H_i is the intersection of C_i, D_i and E_i, F_i and hence $G_i, H_i \subseteq C_i, D_i \subseteq A_i, B_i$. The second property is that $G_i \subseteq A_i$. Note that $mintag$ will "place" a formula in G_i if and only if it occurs in at least one of E_i or C_i. Hence, since $C_i \subseteq A_i$ and $E_i \subseteq A_i$ (by the induction hypothesis) any formula in G_i must occur in A_i hence $G_i \subseteq A_i$ as required.

The $\otimes$ rule:

On the top left-hand side of the rule we have the sequent

$$\delta : p, A_1, \ldots, A_n, B_1^?, \ldots, B_n^? \Rightarrow C_1, \ldots, C_n, D_1^?, \ldots, D_n^?, D_0^?$$

which satisfies the conditions $C_i, D_i \subseteq A_i, B_i$ *and* $C_i \subseteq A_i$
On the top right hand side of the rule we have the sequent

$$\delta : q, C_1^{-1}, C_2^{-1}, \ldots, C_n^{-1}, D_1^{?-1}, D_2^{?-1}, \ldots, D_n^{?-1} \Rightarrow E_1, \ldots, E_n, F_1^?, \ldots, F_n^?, F_0^?$$

which satisfies the conditions $E_i, F_i \subseteq C_{(i+1)}, D_{(i+1)}$ *and* $E_i \subseteq C_{(i+1)}$.
The conclusion of the inference is

$$\delta : p \otimes q, A_1^{-1}, A_2^{-1}, \ldots, A_n^{-1}, B_1^{?-1}, B_2^{?-1}, \ldots, B_n^{?-1} \Rightarrow D_0^?, E_1, \ldots, E_n, F_1^?, \ldots, F_n^?, F_0^?$$

We wish to show that this sequent satisfies the properties $E_i, F_i \subseteq A_{(i+1)}, B_{(i+1)}$ *and* $E_i \subseteq A_{(i+1)}$. *We have that* $E_i \subseteq C_{(i+1)} \subseteq A_{(i+1)}$ *and that* $E_i, F_i \subseteq C_{(i+1)}, D_{(i+1)} \subseteq A_{(i+1)}, B_{(i+1)}$ ■

LEMMA 98
In the $\otimes$ *rule we have that* $\Delta \subseteq \Gamma$.
Proof: *This follows from lemma 97 since* Δ *corresponds to* C_1 *and* Γ *to* A_1. ■

We now define *backpatching*. Backpatching is used to ensure that $D^? \subseteq B^?$. Intuitively, the role of backpatching and propagation is to take formulae which have been consumed by a $\top$ rule and which have not been "unconsumed" and place them at the appropriate $\top$ rule. Note that the result of backpatching and propagation may not be a proper proof under $\mathcal{M}$. However under the mapping defined by the third step of Algorithm 1 (below) the result will be a well formed $\mathcal{L}$ proof. To avoid confusion we shall refer to the result of backpatching and propagating an $\mathcal{M}$ proof as a *quasi-proof*. Note that in some cases backpatching may not have any effect; in these cases the quasi-proof is just an $\mathcal{M}$ proof. An example of such a proof is

$$\frac{\dfrac{}{\mathbf{1} \Rightarrow} \quad \dfrac{}{\top \Rightarrow}}{\mathbf{1} \otimes \top \Rightarrow}$$

DEFINITION 30 (QUASI-PROOF)
A quasi-proof is a backpatched and propagated $\mathcal{M}$ *proof.*

DEFINITION 31 (BACKPATCHING)
To backpatch an $\mathcal{M}$ *proof we modify certain sequents occurring in the proof as follows. The structure of the proof remains unchanged.*

- *We replace a* ! *inference of the form*

$$\frac{\delta : F \Rightarrow D^?/x}{\cdots}\ !$$

with the inference

$$\frac{\delta : F, D^? \Rightarrow D^?/x}{\cdots}\ !$$

- *We replace the inference*

$$\frac{\ldots \Rightarrow \ldots \Pi_x^?, \Pi^? \quad \ldots \Rightarrow \ldots \Xi_x^?, \Xi^?}{\ldots \Rightarrow \ldots \Sigma_x^?, \Sigma^?}\ \&$$

with the inference

$$\frac{\ldots (\Pi_x^?, \Pi^?) - (\Sigma_x^?, \Sigma^?) \Rightarrow \ldots \Pi_x^?, \Pi^? \quad \ldots (\Xi_x^?, \Xi^?) - (\Sigma_x^?, \Sigma^?) \Rightarrow \ldots \Xi_x^?, \Xi^?}{\ldots \Rightarrow \ldots \Sigma_x^?, \Sigma^?}\ \&$$

- *We transform the root of the proof from*

$$\begin{array}{c} \vdots \\ \Gamma \Rightarrow D^? \end{array}$$

to

$$\begin{array}{c} \vdots \\ \Gamma, D^? \Rightarrow D^? \end{array}$$

EXAMPLE 13

We backpatch the proof

$$\frac{\dfrac{}{\top, p_1, q_1 \Rightarrow p_1^?, q_1^?/true}\ \top \quad \dfrac{\dfrac{}{q^\perp, p^?, q \Rightarrow p^?/false}\ Ax}{q^\perp, p^?, q^? \Rightarrow p^?/false}\ Use}{\top \otimes q^\perp, p, q \Rightarrow p^?/true}\ \otimes$$

In this case, there are no occurrences of ! *or* & *inferences and so the only sequent that is changed is the root of the proof which has* $p^?$ *added to it yielding*

$$\frac{\dfrac{}{\top, p_1, q_1 \Rightarrow p_1^?, q_1^?/true}\ \top \quad \dfrac{\dfrac{}{q^\perp, p^?, q \Rightarrow p^?/false}\ Ax}{q^\perp, p^?, q^? \Rightarrow p^?/false}\ Use}{p^?, \top \otimes q^\perp, p, q \Rightarrow p^?/true}\ \otimes$$

Propagation fixes the invalid inferences that are temporarily introduced by backpatching. We propagate from each point where backpatching was done. Note that the propagated formulae are of the form $p^?$. When propagating into the left premise of a $\otimes$ rule we do *not* add a tag; that is, the propagated formulae remain in the form $p^?$ and not the form $p^?_i$. The reason for this is that backpatching introduces formulae in order to balance residue of the form $p^?$. We therefore desire the introduced formulae to remain associated with the appropriate residue.

DEFINITION 32 (PROPAGATION)
To propagate from a sequent we take the formulae which were added by backpatching and add them to the sequent's premises using the following rules. Propagation is applied to each sequent which was backpatched.

- *If the sequent is the conclusion of one of* $\parr, \bot, \oplus, \forall, \exists, ?D, ?$ *or Use then the formulae are added to the premise of the inference and we propagate from the premise.*
- *If the sequent is the conclusion of one of* $\mathbf{1}, Ax$ *or* $!$ *then propagation fails.*
- *If the sequent is the conclusion of a* $\top$ *inference then propagation succeeds.*
- *If the sequent is the conclusion of a* $\&$ *inference then the formulae are added to* both *premises and we propagate from both premises.*
- *If the sequent is the conclusion of a* $\otimes$ *inference then there are four cases:*
  1. *Both premises are tagged* $/false$ *– propagation fails.*
  2. *The left premise is tagged* $/true$ *and the right* $/false$ *– we add all of the relevant formulae to the left premise and propagate from the left premise.*
  3. *The right premise is tagged* $/true$ *and the left* $/false$ *– we add all of the relevant formulae to the right premise and propagate from the right premise.*
  4. *Both premises are tagged* $/true$ *– We add the formulae corresponding to* $\Xi^?$ *to the right premise and those corresponding to* Σ^t_x *or* $\Sigma^?$ *to the left premise. We then propagate from both premises.*

EXAMPLE 14

We apply propagation to the root of the following backpatched proof.

$$\dfrac{\dfrac{}{\top, p_1, q_1 \Rightarrow p_1^?, q_1^?/true}\ \top \quad \dfrac{\dfrac{}{q^\perp, p^?, q \Rightarrow p^?/false}\ Ax}{q^\perp, p^?, q^? \Rightarrow p^?/false}\ Use}{p^?, \top \otimes q^\perp, p, q \Rightarrow p^?/true}\ \otimes$$

In this case backpatching has added the formula $p^?$ to the root of the proof. The sequent is the conclusion of a $\otimes$ inference with one premise tagged $/true$ and the other tagged $/false$. We add the formula to the left premise (which is tagged $/true$) and propagate from that premise. Propagation from the conclusion of a $\top$ inference simply succeeds. This gives us the following quasi-proof:

$$\dfrac{\dfrac{}{p^?, \top, p_1, q_1 \Rightarrow p_1^?, q_1^?/true}\ \top \quad \dfrac{\dfrac{}{q^\perp, p^?, q \Rightarrow p^?/false}\ Ax}{q^\perp, p^?, q^? \Rightarrow p^?/false}\ Use}{p^?, \top \otimes q^\perp, p, q \Rightarrow p^?/true}\ \otimes$$

Note that the quasi-proof's $\top$ inference is not actually an instance of the $\top$ rule since $p^?$ is not returned as residue. Applying the translation in Algorithm 1 (see below) to this quasi-proof yields the following $\mathcal{L}$ proof:

$$\dfrac{\dfrac{}{: \top, p}\ \top \quad \dfrac{}{: q^\perp, q}\ Ax}{: \top \otimes q^\perp, p, q}\ \otimes$$

By applying backpatching and propagation we obtain a proof of the same form where at every sequent $D_0^?$ is a sub-multiset of $B_0^?$. The next few lemmas show that backpatching and propagation will always succeed. We begin by showing that whenever backpatching is needed there is a $\top$ rule in the proof.

DEFINITION 33

We say that a $\top$ inference occurs above a sequent if the sequent is either the conclusion of a $\top$ inference or a $\top$ inference is above one of the sequent's premises.

LEMMA 99

If there is no $\top$ inference above a sequent then $C_i \subseteq A_i$ and $D_i \subseteq B_i$ for $i \geq 1$.

Proof: *The first property has already been proven in lemma 97 for the more general case*

including $\top$ inferences. Hence we only need to show that in the absence of $\top$ inferences $D_i \subseteq B_i$. We use induction.

The 1 rule: *$D_i = B_i$ so the property holds.*

The $\parr$ rule: *B_i and D_i are the same in both the premise and the conclusion. By the induction hypothesis $D_i \subseteq B_i$ in the premise and hence this holds for the conclusion.*

The ! rule: *By the induction hypothesis the desired property holds for the premise of the inference. In the conclusion of the inference $D_i = B_i$ (as for* **1***) and the property follows.*

The $\top$ rule: *Cannot occur by definition.*

The *Use* **rule:** *By the induction hypothesis $D_i \subseteq B_i$ in the premise. The conclusion differs from the premise in that one of the B_i might be augmented with F hence $D_i \subseteq B_i$ holds in the conclusion.*

The $\otimes$ rule: *On the top left hand side of the rule we have the sequent*

$$\delta : p, A_1, \ldots, A_n, B_1^?, \ldots, B_n^? \Rightarrow C_1, \ldots, C_n, D_1^?, \ldots, D_n^?, D_0^?$$

which satisfies the condition $D_i \subseteq B_i$.

On the top right hand side of the rule we have the sequent

$$\delta : q, C_1^{-1}, C_2^{-1}, \ldots, C_n^{-1}, D_1^{?-1}, D_2^{?-1}, \ldots, D_n^{?-1} \Rightarrow E_1, \ldots, E_n, F_1^?, \ldots, F_n^?, F_0^?$$

which satisfies the condition $F_i \subseteq D_{(i+1)}$.

The inference's conclusion is

$$\delta : p \otimes q, A_1^{-1}, A_2^{-1}, \ldots, A_n^{-1}, B_1^{?-1}, B_2^{?-1}, \ldots, B_n^{?-1} \Rightarrow D_0^?, E_1, \ldots, E_n, F_1^?, \ldots, F_n^?, F_0^?$$

We have that $F_i \subseteq D_{i+1} \subseteq B_{i+1}$.

The $\&$ rule: *The left premise is*

$$\delta : A_0, A_1, \ldots, A_n, B_1^?, \ldots, B_n^? \Rightarrow C_1, \ldots, C_n, D_1^?, \ldots, D_n^?, D_0^?$$

and the right premise is

$$\delta : A_0, A_1, \ldots, A_n, B_1^?, \ldots, B_n^? \Rightarrow E_1, \ldots, E_n, F_1^?, \ldots, F_n^?, F_0^?$$

These satisfy the constraints $D_i \subseteq B_i$ and $F_i \subseteq B_i$.
The conclusion of the inference is

$$\delta : A_0, A_1, \ldots, A_n, B_1^?, \ldots, B_n^? \Rightarrow G_1, \ldots, G_n, H_1^?, \ldots, H_n^?, H_0^?$$

We wish to show that $H_i \subseteq B_i$. For a formula to be in H_i it must be in both *F_i and D_i, hence it must be in B_i and $H_i \subseteq B_i$ as required.* ■

LEMMA 100
If there is no $\top$ inference above a sequent then $D_0 \subseteq B_0$.
Proof: *Induction over the height of the proof. The only non-trivial rules are $\otimes$ and $\&$.*
The $\&$ rule:
The induction hypothesis applied to the right premise give us that $\Xi^? \subseteq \Gamma^?$. Since $\Sigma^? = \Xi^? \cap \ldots$ we have that $\Sigma^? \subseteq \Xi^? \subseteq \Gamma^?$ as required.
The $\otimes$ rule:
Using the notation of the $\otimes$ rule we wish to show that $\Sigma^?, \Xi^? \subseteq \Gamma^?$. Applying the induction hypothesis to the left premise tells us that $\Xi^? \subseteq \emptyset$. Applying the induction hypothesis to the right premise tells us that $\Sigma^? \subseteq \Pi^?$. Finally, applying lemma 99 to the left premise provides us with the information that $\Pi^{?+1} \subseteq \Gamma^{?+1}$. Putting all this together we have that $\Xi^?, \Sigma^? \subseteq \Pi^? \subseteq \Gamma^?$ as desired. ■

We now show that propagation always succeeds.

LEMMA 101
If a sequent is tagged $/false$ then for all $i \geq 0$ we have that $D_i \subseteq B_i$.
Proof: *By induction over the structure of the proof. The two non-obvious cases are the binary rules.*

- *$\&$: There are three cases. Note that the conclusion of a $\&$ inference is tagged with $/true$ iff both premises are tagged with $/true$.*
  1. *Both premises are tagged $/false$: In this case both premises satisfy the condition and it follows from examination of the rule that so does the conclusion.*
  2. *Exactly one of the premises is tagged $/false$: The B_i in the premise and the conclusion are identical. Since $D_i^{conclusion} = D_i^{leftpremise} \cap D_i^{rightpremise}$*

we have that

$$D_i^{conclusion} \subseteq D_i^{leftpremise}$$

$$D_i^{conclusion} \subseteq D_i^{rightpremise}$$

Without loss of generality assume that the left premise is tagged $/false$*. Hence* $D_i^{leftpremise} \subseteq B_i$ *and* $D_i^{conclusion} \subseteq D_i^{leftpremise} \subseteq B_i$ *as required.*

3. *Both premises are tagged* $/true$*: The conclusion of the inference is tagged* $/true$ *and the induction trivially holds.*

$\otimes$*: There are two cases*

1. *Both premises are tagged* $/false$*: In this case for* $i > 0$ *we apply lemma 99 and for* $i = 0$ *we apply lemma 100.*
2. *One or more of the premises are tagged* $/true$*: The conclusion of the inference must be tagged* $/true$ *and the induction trivially holds.*

COROLLARY:*If* $D_i \not\subseteq B_i$ *for some* $i \geq 0$ *then the sequent must be tagged* $/true$*.* ■

We have shown that $D_i \not\subseteq B_i$ can not be the case for sequents which are tagged $/false$. We now argue that (a) propagation will always succeed when it is applied to sequents which are tagged $/true$, and (b) backpatching is only applied to sequents tagged with $/true$. It follows that backpatching and propagation cannot fail.

LEMMA 102

If a sequent is tagged with $/true$ *then we can add formulae to the sequent and successfully propagate them.*

Proof: *By induction on the height of the sequent calculus proof. The base case is the* $\top$ *rule which trivially satisfies the desired property. The only cases where propagation fails are a* $\otimes$ *inference with both premises tagged* $/false$ *and* Ax*,* **1** *and* ! *inferences. We therefore need to show that if a sequent is tagged with* $/true$ *then*

- *If it is a unary rule its premise must also be tagged* $/true$ *and hence by the induction hypothesis we can add formulae to the premise of the inference and successfully propagate them.*

- *If it is a $\&$ inference both premises must be $/true$ and hence propagation succeeds.*

- *If it is a $\otimes$ inference then at least one premise must be $/true$.*

From examination of the algorithm for propagation it is obvious that only sequents tagged $/true$ are ever propagated into, providing *the initial sequent was tagged with $/true$. Thus propagation from a $/true$ sequent cannot fail.* ■

LEMMA 103
Backpatching and propagation succeed on any $\mathcal{M}$ proof.
Proof: *We show that backpatching is only applied at sequents labelled $/true$ and invoke lemma 102. There are three places where backpatching may be applied:*

1. *A $\&$ inference: The side conditions on the rule ensure that if backpatching adds a nonempty multiset of formulae to a sequent then the sequent will be tagged $/true$.*

2. *A ! inference: For backpatching to apply $D^?$ must be nonempty in the premise. Since the premise has $B^? = \emptyset$ this implies that $D^? \not\subseteq B^?$ and hence by lemma 101 the sequent must be tagged $/true$.*

3. *The root of the proof: For backpatching to apply $D^?$ must be nonempty. Since the root of the proof has $B^? = \emptyset$ we can apply lemma 101 to conclude that if backpatching is needed the sequent must be tagged $/true$.* ■

We are now in a position to define an algorithm mapping from $\mathcal{M}$ to $\mathcal{L}$ proofs and to use it to prove the soundness of $\mathcal{M}$ with respect to $\mathcal{L}$.

Our first task is to show that the algorithm produces an output for any input $\mathcal{M}$ proof. We then show that the output is a well formed $\mathcal{L}$ proof. Soundness follows from this.

In proving successful termination the only questionable step of the algorithm is the subtraction

$$(A_x^{t\prime}, B^{?\prime}) - (C_x^{t\prime}, D^{?\prime})$$

We have shown that $C_x^t \subseteq A_x^t$ but there are cases where $D^? \not\subseteq B^?$.

Algorithm 1 Translating $\mathcal{M}$ to $\mathcal{L}$ proofs

Input: An $\mathcal{M}$ proof with root $\Gamma \Rightarrow D^?$

Procedure:

1. Backpatch (Definition 31 on page 127)
2. Propagate (Definition 32 on page 129)
3. Apply the translation:

$$\delta : \Gamma, A^t_x, B^? \Rightarrow C^t_x, D^?$$

$$\Downarrow$$

$$\delta : \Gamma,\ ((A^{t\prime}_x, B^{?\prime}) - (C^{t\prime}_x, D^{?\prime}))$$

Output: An $\mathcal{L}$ proof of Γ

EXAMPLE 15

The following proof has $B^? = \emptyset$ and $D^? = p^?$.

$$\dfrac{\dfrac{}{p_1, \top \Rightarrow p_1^?/true}\,\top \quad \dfrac{}{p^?, \mathbf{1} \Rightarrow p^?/false}\,\mathbf{1}}{p, \top \otimes \mathbf{1} \Rightarrow p^?/true}\,\otimes$$

We thus need to show that the result of backpatching and propagation has $D^? \subseteq B^?$.

LEMMA 104

If a sequent has the form $\ldots \Rightarrow \ldots D^? \ldots$ then Algorithm 1 will backpatch $D^?$ into the proof somewhere below the sequent.

Proof: *We argue that if a premise of an inference has $D^? \neq \emptyset$ then either*

1. *$D^?$ is backpatched (if the rule is $!$ or in certain cases for a $\&$ inference)*
2. *The conclusion of the inference contains $D^?$ in its residue.*

The only rules where the $D^?$ in the premise and the conclusion are not identical are $!$, $\otimes$ and $\&$. The conclusion of the cross rule has all formulae of the form $D^?$ which appear

in either premise. The conclusion of the $\&$ rule contains some of the formulae of the appropriate form which appear in the rule's premises. The formulae which are not in the conclusion of the inference are backpatched. All of the formulae of the form $D^?$ in the premise of the $!$ rule are backpatched. Hence if the formulae of the form $D^?$ in a particular sequent are not backpatched at that inference, they are present in the residue of the root and hence are backpatched there. ■

Note that the backpatching of a multiset $D^?$ need not happen at the same place - it is possible for a $\&$ rule to backpatch part of a multiset and pass the rest downwards.

EXAMPLE 16

A proof of the sequent $\top \otimes \mathbf{1}, p, q \,\&\, r$ begins with the inference

$$\frac{\top \otimes \mathbf{1}, p, q \Rightarrow p^?, q^?/true \quad \top \otimes \mathbf{1}, p, r \Rightarrow p^?, r^?/true}{\top \otimes \mathbf{1}, p, q \,\&\, r \Rightarrow p^?/true}\ \&$$

This inference is backpatched and propagated to yield the quasi-proof

$$\frac{\top \otimes \mathbf{1}, p, q, p^?, q^? \Rightarrow p^?, q^?/true \quad \top \otimes \mathbf{1}, p, r, p^?, r^? \Rightarrow p^?, r^?/true}{\top \otimes \mathbf{1}, p, q \,\&\, r, p^? \Rightarrow p^?/true}\ \&$$

which is then transformed to the following $\mathcal{L}$ proof

$$\frac{: \top \otimes \mathbf{1}, p, q \quad : \top \otimes \mathbf{1}, p, r}{: \top \otimes \mathbf{1}, p, q \,\&\, r}\ \&$$

LEMMA 105

Any sequent of the form $\ldots \Rightarrow \ldots D^?/true$ will have (at least) $D^?$ propagated into it by Algorithm 1.

Proof: *By lemma 104 we have that $D^?$ must have been backpatched below the sequent. We then use induction on the structure of the proof from the root to the leaves, to show that it will be propagated into the sequent. The base case of the induction is the point where $D^?$ was backpatched. The unary rules are obvious, the nullary rules are irrelevant. The $\&$ rule works by definition – if the conclusion of the inference is of the appropriate form then it has $D^?$ propagated into it which also propagates $D^?$ into the premises. If the conclusion of a $\otimes$ inference is of the appropriate form then if either premise is of the appropriate form we can propagate the appropriate formulae into it.*

If one of the premises is false the other may get more than $D^?$ propagated into it. For instance, in the previous example the $\top$ inference has an empty $D^?$ but has $p^?$ propagated into it.

LEMMA 106

Any sequent occurring in a quasi-proof satisfies the condition $C_x^{t\prime}, D^{?\prime} \subseteq A_x^{t\prime}, B^{?\prime}$.

Proof: *From lemma 97 we have that* $C_x^{t\prime} \subseteq A_x^{t\prime}$. *If the original sequent (that is before backpatching and propagating) violated the condition then by the corollary of lemma 101 we have that the sequent must be tagged with* $/true$ *and hence by lemma 105 we have that the violation will be fixed by backpatching and propagating.* ■

We now need only to show that the result of the algorithm is a proper $\mathcal{L}$ proof.

THEOREM 107 (SOUNDNESS)

If $\Gamma \Rightarrow D^?$ *is provable in* $\mathcal{M}$ *then there exists an* $\mathcal{L}$ *proof of* $\emptyset : \Gamma$.

Proof: *The root of the proof is backpatched and transformed to* $: \Gamma$. *We then show that each of the inference rules of* $\mathcal{M}$ *are backpatched, propagated and transformed into sound* $\mathcal{L}$ *inferences. Hence the algorithm maps the* $\mathcal{M}$ *proof of* $\Gamma \Rightarrow D^?$ *into an* $\mathcal{L}$ *proof of* $: \Gamma$.

The 1 rule:

This rule is not backpatched and is never propagated since it is tagged $/false$. *It transforms to*

$$\frac{}{: \mathbf{1}}\ \mathbf{1}$$

The $\parr$ rule:

This rule is not backpatched. Let $\Sigma^?$ *be propagated into the rule and let* $\Delta = (A_x^{t\prime}, B^{?\prime}, \Sigma^{?\prime}) - (D^{?\prime}, C_x^{t\prime})$ *then the rule is transformed to*

$$\frac{\delta : F, G, \Gamma, \Delta}{\delta : F \parr G, \Gamma, \Delta}\ \parr$$

The $\top$ rule:

This rule is not backpatched. Let $\Sigma^?$ *be propagated into the rule and let* $\Delta = (A_x^{t\prime}, B^{?\prime}, \Sigma^{?\prime}) - (A_x^{t+?\prime}, B^{?\prime})$ *(actually, since* $A_x^{t+?\prime} = A_x^{t\prime}$ *we have that* $\Delta = \Sigma$*) then the transformed*

rule is

$$\frac{}{\delta : \top, \Gamma, \Delta}\ \top$$

The $\&$ rule:

Let $\aleph^?$ be the propagated formulae and let $\Sigma = \Sigma_x^t \cup \Sigma^?$. The result of backpatching and propagating a $\&$ inference is

$$\frac{\delta : F, \Gamma, \Gamma_x^t, \Gamma^?, \aleph^?, ((\Pi_x^t, \Pi^?) - \Sigma) \Rightarrow \Pi_x^t, \Pi^? \quad \delta : G, \Gamma, \Gamma_x^t, \Gamma^?, \aleph^?, ((\Xi_x^t, \Xi^?) - \Sigma) \Rightarrow \Xi_x^t, \Xi^?}{\delta : F \,\&\, G, \Gamma, \Gamma_x^t, \Gamma^?, \aleph^? \Rightarrow \Sigma}\ \&$$

Let $\Delta = (\Gamma_x^t, \Gamma^?, \aleph^?) - \Sigma$. Then the translation of the inference into an $\mathcal{L}$ inference is

$$\frac{\delta : F, \Gamma, ((\Gamma_x^{t\prime}, \Gamma^{?\prime}, \aleph^{?\prime}, ((\Pi_x^{t\prime}, \Pi^{?\prime}) - \Sigma')) - (\Pi_x^{t\prime}, \Pi^{?\prime})) \quad \delta : G, \Gamma, ((\Gamma_x^{t\prime}, \Gamma^{?\prime}, \aleph^{?\prime}, ((\Xi_x^{t\prime}, \Xi^{?\prime}) - \Sigma')) - (\Xi_x^{t\prime}, \Xi^{?\prime}))}{\delta : F \,\&\, G, \Gamma, \Delta}\ \&$$

Observe that

$$\begin{aligned}
& ((\Gamma_x^{t\prime}, \Gamma^{?\prime}, \aleph^{?\prime}, ((\Pi_x^{t\prime}, \Pi^{?\prime}) - \Sigma')) - (\Pi_x^{t\prime}, \Pi^{?\prime})) \\
= \ & ((\Gamma_x^{t\prime}, \Gamma^{?\prime}, \aleph^{?\prime}, \Pi_x^{t\prime}, \Pi^{?\prime}) - (\Pi_x^{t\prime}, \Pi^{?\prime}, \Sigma')) \\
= \ & ((\Gamma_x^{t\prime}, \Gamma^{?\prime}, \aleph^{?\prime}) - \Sigma'), ((\Pi_x^{t\prime}, \Pi^{?\prime}) - (\Pi_x^{t\prime}, \Pi^{?\prime})) \\
= \ & ((\Gamma_x^{t\prime}, \Gamma^{?\prime}, \aleph^{?\prime}) - \Sigma') \\
= \ & \Delta
\end{aligned}$$

So the inference can be rewritten as

$$\frac{\delta : p, \Gamma, \Delta \quad \delta : q, \Gamma, \Delta}{\delta : p \,\&\, q, \Gamma, \Delta}\ \&$$

The *Use* rule:

This rule is not backpatched. Let $\Sigma^?$ be the propagated formulae and let $\Delta = (A_x^{t\prime}, B^{?\prime}, \Sigma^{?\prime}) - (C_x^{t\prime}, D^{?\prime})$ then the rule propagates and transforms to

$$\frac{\delta : F', \Gamma, \Delta}{\delta : F', \Gamma, \Delta}\ Use$$

which is a sound (if useless!) $\mathcal{L}$ inference.

The ! rule:

This rule is not propagated. *It is backpatched to*

$$\frac{\delta : F, D^? \Rightarrow D^?/x}{\delta :!F, A^t_x, B^? \Rightarrow A^t_x, B^?/false}\ !$$

and transforms to

$$\frac{\delta : F}{\delta :!F}\ !$$

The $\otimes$ rule:

$$\frac{\delta : F, \Gamma^{+1}, \Gamma^{t+1}_x, \Gamma^{?+1} \Rightarrow \Delta^{+1}, \Pi^{t+1}_x, \Pi^{?+1}, \Xi^?/x \quad \delta : G, \Delta, \Pi^t_x, \Pi^? \Rightarrow \Sigma^t_x, \Sigma^?/y}{\delta : F \otimes G, \Gamma, \Gamma^t_x, \Gamma^? \Rightarrow \Sigma^t_x, \Sigma^?, \Xi^?/x \vee y}\ \otimes$$

This rule is not backpatched. There are four cases:

1. $x = y = false$ *in which case there is no propagation and the rule is mapped to*

$$\frac{\delta : F,\ ((\Gamma^{+1}, \Gamma^{t+1}_x, \Gamma^{?+1})' - (\Delta^{+1}, \Xi^?, \Pi^{t+1}_x, \Pi^{?+1})') \quad \delta : G, \Delta,\ ((\Pi^t_x, \Pi^?)' - (\Sigma^t_x, \Sigma^?)')}{\delta : F \otimes G, \Gamma,\ ((\Gamma^t_x, \Gamma^?)' - (\Sigma^t_x, \Xi^?, \Sigma^?)')}\ \otimes$$

In order for this to be a valid $\mathcal{L}$ inference we need to show that the formulae in the bottom sequent of the rule are split between the left and right top sequents. Note that we shall take the primes ($-'$) as read in order to avoid notational clutter.

$$\underbrace{((\Gamma, \Gamma^t_x, \Gamma^?) - (\Delta, \Xi, \Pi^t_x, \Pi^?))}_{\textit{Top Left Sequent}},\ \underbrace{\Delta,\ ((\Pi^t_x, \Pi^?) - (\Sigma^t_x, \Sigma^?))}_{\textit{Top Right Sequent}} = \underbrace{\Gamma,\ ((\Gamma^t_x, \Gamma^?) - (\Sigma^t_x, \Sigma^?, \Xi^?))}_{\textit{Bottom Sequent}}$$

We now show that the left hand side of the equation is equal to the right. The left hand side is $((\Gamma, \Gamma^t_x, \Gamma^?) - (\Delta, \Xi, \Pi^t_x, \Pi^?)), \Delta,\ ((\Pi^t_x, \Pi^?) - (\Sigma^t_x, \Sigma^?))$. Rearranging we obtain $\Delta, ((\Gamma, \Gamma^t_x, \Gamma^?, \Pi^t_x, \Pi^?) - (\Delta, \Xi, \Pi^t_x, \Pi^?, \Sigma^t_x, \Sigma^?))$. We can subtract $\Pi^t_x, \Pi^?$ to yield $\Delta, ((\Gamma, \Gamma^t_x, \Gamma^?) - (\Delta, \Xi, \Sigma^t_x, \Sigma^?))$. From lemma 98 we have that $\Delta \subseteq \Gamma$. We let $\Gamma = \Delta, \Delta'$ obtaining $\Delta, ((\Delta, \Delta', \Gamma^t_x, \Gamma^?) - (\Delta, \Xi, \Sigma^t_x, \Sigma^?))$. We can subtract the Δ yielding $\Delta, ((\Delta', \Gamma^t_x, \Gamma^?) - (\Xi, \Sigma^t_x, \Sigma^?))$. Applying lemma 106 to the conclusion of the inference tells us that $\Sigma^t_x, \Sigma^?, \Xi^? \subseteq \Gamma^t_x, \Gamma^?$. This allows us

to lift out the Δ' *giving* $\Delta, \Delta'((\Gamma_x^t, \Gamma^?) - (\Xi, \Sigma_x^t, \Sigma^?))$. *But by definition* $\Gamma = \Delta, \Delta'$ *and so this is just* $\Gamma, ((\Gamma_x^t, \Gamma^?) - (\Xi, \Sigma_x^t, \Sigma^?))$. *But this is just the formulae occurring in the conclusion of the inference so we have a valid* $\mathcal{L}$ *inference.*

2. $x = true$ *and* $y = false$. *Let* $\aleph^?$ *be the propagated formulae, then the rule is propagated and transformed to*

$$\frac{\delta : F,\ ((\aleph^?, \Gamma^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1})' - (\Delta^{+1}, \Xi^?, \Pi_x^{t+1}, \Pi^{?+1})') \quad \delta : G, \Delta,\ ((\Pi_x^t, \Pi^?)' - (\Sigma_x^t, \Sigma^?)')}{\delta : F \otimes G, \Gamma,\ ((\Gamma_x^t, \Gamma^?, \aleph^?)' - (\Sigma_x^t, \Xi^?, \Sigma^?)')} \otimes$$

For this to correspond to the $\mathcal{L}$ *rule for* $\otimes$ *we need to show that*

$$\Gamma,\ ((\Gamma_x^t, \Gamma^?, \aleph^{?\prime})' - (\Sigma_x^t, \Xi^?, \Sigma^?)')$$

$$= ((\aleph^?, \Gamma^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1})' - (\Delta^{+1}, \Xi^?, \Pi_x^{t+1}, \Pi^{?+1})'), \Delta,\ ((\Pi_x^t, \Pi^?)' - (\Sigma_x^t, \Sigma^?)')$$

Consider the right hand side of this equation. We can rearrange the formulae to obtain $\Delta,\ ((\aleph^?, \Gamma^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1}, \Pi_x^t, \Pi^?)' - (\Delta^{+1}, \Xi^?, \Pi_x^{t+1}, \Pi^{?+1}, \Sigma_x^t, \Sigma^?)')$. *By subtracting* $\Pi_x^{t(+1)}, \Pi^{?(+1)}$ *we have* $\Delta,\ (\aleph^?, \Gamma^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1})' - (\Delta^{+1}, \Xi^?, \Sigma_x^t, \Sigma^?)'$. *As before, lemma 98 allows us to define* $\Gamma = \Delta, \Delta'$. *We can subtract the* Δ *and obtain* $\Delta,\ (\aleph^?, \Delta'^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1})' - (\Xi^?, \Sigma_x^t, \Sigma^?)'$. *As before we can apply lemma 106 to the conclusion of the inference and conclude that* $\Sigma_x^t, \Sigma^?, \Xi^? \subseteq \Gamma_x^t, \Gamma^?, \aleph^?$. *This allows us to lift out the* Δ' *yielding* $\Delta, \Delta'\ (\aleph^?, \Gamma_x^{t+1}, \Gamma^{?+1})' - (\Xi^?, \Sigma_x^t, \Sigma^?)'$ *which is by definition* $\Gamma\ (\aleph^?, \Gamma_x^{t+1}, \Gamma^{?+1})' - (\Xi^?, \Sigma_x^t, \Sigma^?)'$. *This shows the desired equality. Hence the resulting inference is valid in* $\mathcal{L}$.

3. $x = false$ *and* $y = true$. *The reasoning here is analogous to the previous case.*

4. $x = y = true$. *Let* $\aleph^?$ *and* $\beth^?$ *be the propagated formulae.* $\aleph^?$ *gets propagated into the left premise.* $\beth^?$ *gets propagated into the right premise. The* $\otimes$ *rule is propagated and transformed into*

$$\frac{\delta : F,\ ((\aleph^?, \Gamma^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1})' - (\Delta^{+1}, \Xi^?, \Pi_x^{t+1}, \Pi^{?+1})') \quad \delta : G, \Delta,\ ((\beth^?, \Pi_x^t, \Pi^?)' - (\Sigma_x^t, \Sigma^?)')}{\delta : F \otimes G, \Gamma,\ ((\Gamma_x^t, \Gamma^?, \aleph^?, \beth^?)' - (\Sigma_x^t, \Xi^?, \Sigma^?)')} \otimes$$

For this inference to be valid in $\mathcal{L}$ *we need to show that*

$$\Gamma,\ ((\Gamma_x^t, \Gamma^?, \aleph^?, \beth^?)' - (\Sigma_x^t, \Xi^?, \Sigma^?)')$$

$= ((\aleph^?, \Gamma^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1})' - (\Delta^{+1}, \Xi^?, \Pi_x^{t+1}, \Pi^{?+1})'), \Delta, ((\beth^?, \Pi_x^t, \Pi^?)' - (\Sigma_x^t, \Sigma^?)')$

Consider the right hand side of the equation.

$((\aleph^?, \Gamma^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1})' - (\Delta^{+1}, \Xi^?, \Pi_x^{t+1}, \Pi^{?+1})'), \Delta, ((\beth^?, \Pi_x^t, \Pi^?)' - (\Sigma_x^t, \Sigma^?)')$

We can rearrange the equation to obtain $\Delta, ((\aleph^?, \Gamma^{+1}, \Gamma_x^{t+1}, \Gamma^{?+1}, \beth^?, \Pi_x^t, \Pi^?) - (\Delta^{+1}, \Xi^?, \Pi_x^{t+1}, \Pi^{?+1}, \Sigma_x^t, \Sigma^?))$. *As before, lemma 98 allows us to define* $\Gamma = \Delta, \Delta'$. *We can then subtract the* Δ *and use lemma 106 to allow us to lift out the* Δ' *yielding* $\Delta, \Delta', (\aleph^?, \Gamma_x^{t+1}, \Gamma^{?+1}, \beth^?) - (\Xi^?, \Sigma_x^t, \Sigma^?)$ *which is just* $\Gamma, (\aleph^?, \Gamma_x^{t+1}, \Gamma^{?+1}, \beth^?) - (\Xi^?, \Sigma_x^t, \Sigma^?)$ *as desired.*

■

4.3 Completeness

Given an $\mathcal{L}$ proof it is fairly trivial to find an equivalent $\mathcal{M}$ proof — one can simply use *Use* inferences after each $\otimes$ rule to ensure resources are allocated correctly. Unfortunately the resulting proof is not one which can be found bottom up using a deterministic search procedure since it applies the *Use* rule in an "omniscient" way.

In order to have a more useful completeness property we need that for any $\mathcal{L}$ proof there is an $\mathcal{M}$ proof which only uses the *Use* rule deterministically. We define a subclass of proofs (*use-valid*) which captures the deterministic use of the *Use* rule. We then give an algorithm which given an $\mathcal{L}$ proof produces a use-valid $\mathcal{M}$ proof.

DEFINITION 34

An application of Use is said to be valid if it either occurs immediately beneath a rule which acts on the side formula of the Use rule or it occurs in a sequence of rules of the form:

$$\dfrac{\dfrac{\dfrac{}{p, p^{\perp}, A_x^t, B^? \Rightarrow A_x^t, B^?/false}\,Ax}{p, {p^{\perp}}^t_x, A_x^t, B^? \Rightarrow A_x^t, B^?/false}\,Use}{p_x^t, {p^{\perp}}^t_x, A_x^t, B^? \Rightarrow A_x^t, B^?/false}\,Use$$

An application of Use is said to be invalid *if it occurs other than in one of these two situations.*

We use the phrase *use-valid* as shorthand for "all occurrences of the *Use* rule are valid".

Our completeness theorem states that for any $\mathcal{L}$ proof there exists a use-valid $\mathcal{M}$ proof. Hence when searching for a proof in $\mathcal{M}$ it is sufficient to only apply *Use* to a formula when we are about to apply another rule to that formula. That is, we can eliminate *Use* by modifying each rule to operate on tagged formulae. For example in addition to

$$\frac{\delta : F, G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : F \parr G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x} \parr$$

we define the rules:

$$\frac{\delta : F, G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : (F \parr G)_n, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x} \parr$$

$$\frac{\delta : F, G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : (F \parr G)^?, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x} \parr$$

$$\frac{\delta : F, G, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x}{\delta : (F \parr G)_n^?, \Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?/x} \parr$$

The Lygon implementation uses this idea and hence does not need an explicit *Use* rule.

Recall that we use the notation $\Gamma^{t+?}$ to indicate that all the formulae in Γ have a $-^?$ tag added. We shall also define the notation $\Gamma^{t(+?)}$ to indicate that *some* (unknown) subset of the formulae in Γ has $-^?$ tags added.

We shall need a number of lemmas in the construction of the algorithm. These enable us to "massage" proofs by adding tags and formulae.

LEMMA 108

If there is a use-valid proof of $\Gamma, \Gamma_x^t, \Gamma^? \Rightarrow \Delta^?, \Delta_x^t$ *then there is a use-valid proof of* $\Gamma, \Gamma_x^t, \Gamma^?, A_x^t, B^? \Rightarrow A_x^{t(+?)}, B^?, \Delta^?, \Delta_x^t$.

Proof: *We use induction on the structure of the proof.* ■

LEMMA 109

If there is a use-valid proof of $\Gamma \Rightarrow \Delta/false$ *then there is a use-valid proof of* $\Gamma, A_1 \Rightarrow A_1, \Delta/false$.

Proof: *By lemma 108 we have a proof of* $\Gamma, A_1 \Rightarrow A_1^{(+?)}, \Delta/false$. *By lemma 101 we*

have that $D_i \subseteq B_i$ (where D_i, B_i are formulae of the form F_n^t and D_i is on the right of the $\Rightarrow$ and B_i on the left). Since Γ, A_1 does not contain any occurrence of the $-^?$ tag — that is B_i is empty - we have that D_i must be empty. Hence no $-^?$ tags are added, that is, $A^{(+?)} = A$. ■

We define $subf\ \Gamma$ to return all the formulae and sub-formulae occurring in Γ. The intuition is that it represents all the formulae that can be generated from Γ and so we shall also extend $subf\ \Gamma$ with all instances of formulae where quantifiers are involved. For example, if $\Gamma = \{p, q \,\&\, (r \oplus s), \exists x t(x)\}$ then $(subf\ \Gamma) = \{p, q, r, s, r \oplus s, q \,\&\, (r \oplus s), t(X)\}$ for any value of X.

LEMMA 110

For any sequent of the form $\Gamma, A_x^t, B^? \Rightarrow C_x^t, D^?$ in an $\mathcal{M}$ proof we have that $D^? \subseteq (subf\ \Gamma) \cup B^?$.

Proof: *Induction over the structure of the proof. The only non-trivial case is the $\otimes$ rule. Using the notation of the $\otimes$ rule we want to show that $\Xi^? \cup \Sigma^? \subseteq (subf\ (\Gamma \cup \{F \otimes G\})) \cup \Gamma^?$. Applying the induction hypothesis to the left and right premises of the inference yield respectively that $\Xi^? \subseteq (subf\ \{F\})$ and $\Sigma^? \subseteq \Pi^? \cup (subf\ (\Delta \cup \{G\}))$. Applying lemma 97 to the left premise tells us that $\Delta \cup \Pi^? \subseteq \Gamma^? \cup \Gamma$ and that $\Delta \subseteq \Gamma$. Hence*

$$\begin{aligned}
\Xi^? \cup \Sigma^? &\subseteq \Pi^? \cup (subf\ (\Delta \cup \{G, F\})) \\
&\subseteq \Pi^? \cup \Delta \cup (subf\ (\Delta \cup \{G \otimes F\})) \\
&\subseteq \Gamma^? \cup \Gamma \cup (subf\ (\Gamma \cup \{G \otimes F\})) \\
&\subseteq \Gamma^? \cup (subf\ (\Gamma \cup \{G \otimes F\}))
\end{aligned}$$

■

In the construction of the algorithm we will find that we need to find proofs with additional tags at the root. As a general intuition, the tags make it *easier* to prove something. Thus if f adds tags and there is a (use-valid) proof of $A \Rightarrow \ldots$ then there will be a (use-valid) proof of $(f\ A) \Rightarrow \ldots$. Note that in general the residue (the "$\ldots$") will be different. The proof is (as always) by induction. To be able to induce over the binary rules we need to know exactly what the effect of adding tags is on the residue.

Determining the effect of adding tags on the residue is slightly tricky since we are dealing with multisets and as a result it is not obvious where a formula in the residue comes from. For example consider the proof

$$\dfrac{\dfrac{}{\top, p_1, p_1^? \Rightarrow p_1^?, p_1^?}\top \quad \dfrac{\dfrac{}{p^?, p, p^\perp \Rightarrow p^?}Ax}{p^?, p^?, p^\perp \Rightarrow p^?}Use}{\top \otimes p^\perp, p, p^? \Rightarrow p^?}\otimes$$

We now add a $-_1$ tag to the p at the root of the proof. Naïvely we reason that p occurs in the residue and hence adding a tag to it should add a tag to the $p^?$ occurring in the residue. This is correct; however there is another alternative – the residue could have come from the initial residue. Thus either of the following proofs are possible. The difference between the two proofs is that the axiom rule uses a different occurrence of p.

$$\dfrac{\dfrac{}{\top, p_2, p_1^? \Rightarrow p_2^?, p_1^?}\top \quad \dfrac{\dfrac{}{p_1^?, p, p^\perp \Rightarrow p_1^?}Ax}{p_1^?, p^?, p^\perp \Rightarrow p_1^?}Use}{\top \otimes p^\perp, p_1, p^? \Rightarrow p_1^?}\otimes \qquad \dfrac{\dfrac{}{\top, p_2, p_1^? \Rightarrow p_2^?, p_1^?}\top \quad \dfrac{\dfrac{}{p, p^?, p^\perp \Rightarrow p^?}Ax}{p_1^?, p^?, p^\perp \Rightarrow p^?}Use}{\top \otimes p^\perp, p_1, p^? \Rightarrow p^?}\otimes$$

This type of situation makes it non-trivial to reason about the effect on the residue of adding tags. Matters can be simplified if we assume that it is possible to distinguish between multiple occurrences of a formula. In particular, this allows us to track where formulae in the residue actually came from. This is a fairly standard assumption and one way of allowing multiple occurrences of formulae to be distinguished is to give atoms numerical tags. This is done by giving each atom at the root of the proof a unique numeric identifier and then propagating the identifiers upwards through the proof. The rules for propagation are mostly obvious with a few exceptions:

- Duplication of formulae ($?D$) requires the creation of new tags for the atoms in the copy.
- In some situations (like the *Use* rule in the above proofs) multiple assignments of tags are possible. It is crucial to note that whenever more than one assignment of tags is possible *any* of the assignments works. This is obvious since multiple tags are only relevant when we are choosing between identical formulae – if the formulae are identical it does not matter which we choose!

- The axioms use the same tags for the residue. For example the $\top$ rule in the above proof might look like:

$$\frac{}{\top, a1^{\perp}, a2, a3^{?} \Rightarrow a3^{?}} \top$$
$$\vdots$$

- The $\&$ rule uses identical tags in the non-residue parts of both its premises.

For the purposes of the next lemma we shall assume that we can distinguish somehow between different occurrences of a formula. In the following lemma, when we say that something *may* be done we mean that the choice is determined by the proof and just from looking at the root sequent we cannot tell which choice will be made.

LEMMA 111
Let A, B, C and D be multisets of (possibly tagged) formulae. Let F be a single (untagged) formula. The following properties hold:

1. *If there is a use-valid proof of $A, F_i^? \Rightarrow B$ then there is a use-valid proof of $A, F_i^{?+1} \Rightarrow C$ where C differs from B in that if F occurs in B as $F_i^?$ we add one to the tag of the occurrence.* i.e. *$F_n^?$ is replaced by $F_n^{?+1}$.*

2. *If there is a use-valid proof of $A, F_i \Rightarrow B$ then there are use-valid proofs of both*

   - *$A, F_i^{+1} \Rightarrow C$ where C differs from B in that if F occurs in B either as F_i or as $F_i^?$ we add one to the occurrence's tag.*
   - *$A, F_i^? \Rightarrow D$ where D differs from B in that if F occurs in B as F_i then we add a $-^?$ tag to the occurrence.*

3. *If there is a use-valid proof of $A, F^? \Rightarrow B$ then there is a use-valid proof of $A, F_1^? \Rightarrow C$ where C differs from B in that $F^?$ occurs in B then we add one to the occurrence's tag.*

4. *If there is a use-valid proof of $A, F \Rightarrow B/false$ then there is a use-valid proof of $A, F_1 \Rightarrow B/false$.*

5. *If there is a use-valid proof of $A, F \Rightarrow B/true$ then there is a use-valid proof of $A, F_1 \Rightarrow C/true$ where C differs from B in that if $F^?$ occurs in B then we add one to the occurrence's tag.* i.e. *$F^?$ becomes $F^?_1$. If $F^?$ does not occur in B then C differs from B in that it may contain $F^?$. The latter case reflects that we do not know whether the formulae in question was consumed by an axiom or a $\top$ rule.*

6. *If there is a use-valid proof of $A, F \Rightarrow B/false$ then there is a use-valid proof of $A, F^? \Rightarrow B/false$.*

7. *If there is a use-valid proof of $A, F \Rightarrow B/true$ then there is a use-valid proof of $A, F^? \Rightarrow C$ where C differs from B in that $F^?$* may *be added to it.*

Proof: *Induction over the structure of the proof. The rules* **1**, $\top$, ! *and Ax are easily verified.*

Unary Rules:

*In the rules Use and $\parr$ we can use induction trivially if F is not the principal formula. If F is the principal formula then we observe that F cannot occur in B by lemma 110 (since the proof is use-valid the active formula in a Use rule must be decomposed as the next step) and note that in all induction cases, if F is not in B then a proof with an unmodified residue is satisfactory (that is, satisfies all clauses of the form "*may *. . . "). Thus a proof is constructed by adding a Use rule immediately beneath a unary inference other than Use.*

The $\&$ rule:

We note that if F is in exactly one of the premise's Bs then that premise must be tagged $/true$ due to the side conditions on the $\&$ rule. Properties 1–3 are trivial – if F occurs in both premise's B then we must be doing exactly the same thing to both residues so the residue in the conclusion is also modified accordingly. If F occurs in exactly one of the premise's Bs then the modified residue does not get propagated to the conclusion of the inference. Properties 4–7 are more interesting. There are three scenarios:

1. *Both premises are tagged $/true$. The properties that apply can be readily verified using a simple application of the induction hypothesis.*

2. *Both premises are tagged $/false$. Again, simple application of the induction hypothesis suffices.*

3. *One premise is tagged $/true$ and the other is tagged $/false$. This is the interesting case. By lemma 101 applied to the $/false$ premise we have that $D^? \subseteq B^?$. Hence, since F is not in $B^?$ (it has no tags) it cannot occur in $D^?$. Therefore we cannot have a situation where adding tags in the $/true$ premise reduces $D^?$ and forces a violation of the side condition in the $/false$ premise. Having verified that illegal situations cannot arise we can apply the induction hypothesis.*

The $\otimes$ rule:

We prove the properties using simultaneous induction. We indicate for the proof of each property which of the seven inductive hypotheses is applied to which of the two premises. The proofs of properties 1, 2 and 3 are straightforward. For property 1 we apply induction hypothesis 1 to the left premise and induction hypothesis 3 to the right premise. For property 2 we apply induction hypothesis 2 to the left premise and depending on whether F occurs in B as F_i or $F_i^?$ we apply induction hypothesis 1 or 2 to the right premise. For property 3 we apply induction hypothesis 1 to the left premise and induction hypothesis 3 to the right premise. To prove properties 4 and 6 we observe that if the conclusion of the inference is tagged $/false$ then both *premises must be tagged $/false$. We apply induction hypothesis 2 to the left premise and induction hypothesis 4 or 6 to the right premise. Lemma 101 tells us that that for property 4, F can only occur as F_i in the residue of the left premise. In proving property 5 we apply induction hypothesis 2 to the left premise. There are three cases depending on whether F occurs in the left premise's residue, and if so, whether it occurs as F_i or $F_i^?$:*

1. *If F does not occur in the left premise's residue then F does not occur in the residue of the right premise or of the conclusion and the proof follows.*

2. *If F occurs as F_i then we simply apply induction hypothesis 5 to the right premise.*

3. *if F occurs in the residue of the left premise as $F_i^?$ then we apply induction hypothesis 7 to the right premise and then apply induction hypothesis 5 to it.*

To prove property 7 we consider two cases:

1. *The right premise is tagged $/true$: In this case we apply induction hypothesis 2 to the left premise and induction hypothesis 7 to the right premise.*

2. *The right premise is tagged $/false$: In this case we apply induction hypothesis 2 to the left premise and induction hypothesis 6 to the right premise. In this case $F^?$ is not added to C.*

■

LEMMA 112

If there is a use-valid proof of $\Gamma, \Delta \Rightarrow \Sigma^?/true$ then there is a use-valid proof of $\Gamma, \Delta_1 \Rightarrow \Pi^?, \Xi_1^?/true$ for some Ξ and Π.

Proof: *Let $\Delta = X \cup Y \cup Z$ where $X^? \subseteq \Sigma^?$ and $(Y \cup Z) \cap \Sigma = \emptyset$ (*i.e. *X is all the formulae in Δ which occur in Σ). Applying part 5 of lemma 111 we have that formulae in X have one added to their tag and formulae in $(Y \cup Z)$ are possibly added to $\Sigma^?$. Assuming without loss of generality that Y is added to $\Sigma^?$ this gives us a proof of $\Gamma, \Delta_1 \Rightarrow ((\Sigma \cup Y) - X), X_1^?$ which can be written as $\Gamma, \Delta_1 \Rightarrow \Pi^?, \Xi_1^?$ for some Ξ and Π.* ■

LEMMA 113

If there is a use-valid proof of $\Gamma, \Delta \Rightarrow /false$ then there is a use-valid proof of $\Gamma, \Delta_1 \Rightarrow /false$.

Proof: *Corollary of part 4 of lemma 111.* ■

LEMMA 114

If there is a use-valid proof of $\Gamma, \Delta \Rightarrow D^?$ then there is a use-valid proof of $\Gamma^{+?}, \Delta \Rightarrow D^?, \Xi^{+?}$ for some $\Xi \subseteq \Gamma$.

Proof: *Corollary of parts 6 and 7 of lemma 111.* ■

We now present an algorithm for mapping proofs in $\mathcal{L}$ to $\mathcal{M}$. The algorithm (defined in figure 2) makes use of the following definitions.

DEFINITION 35 (TRANSLATING NULLARY RULES IN ALGORITHM 2)

An axiom is translated as itself:

$$\frac{}{\delta : p, p^\perp}\ Ax \qquad \Longrightarrow \qquad \frac{}{\delta : p, p^\perp \Rightarrow /false}\ Ax$$

$$\frac{}{\delta : \top, \Gamma}\ \top \qquad \Longrightarrow \qquad \frac{}{\delta : \top, \Gamma \Rightarrow /true}\ \top$$

Algorithm 2 Translating $\mathcal{L}$ to $\mathcal{M}$ proofs

Input: An $\mathcal{L}$ proof of $\delta : \Gamma$

Output: A use-valid $\mathcal{M}$ proof of $\delta : \Gamma \Rightarrow D^?/x$

Procedure: We process the input top down. At each stage the premise(s) of the current inference have already been processed and are $\mathcal{M}$ proofs of the form $\Gamma \Rightarrow D^?/x$. Rules are translated according to definitions 35 - 38.

DEFINITION 36 (TRANSLATING UNARY RULES IN ALGORITHM 2)

$$\frac{\begin{array}{c}\vdots\\ \delta : F, G, \Gamma\end{array}}{\delta : F \parr G, \Gamma}\ \parr$$

We have already translated the proof of $\delta : F, G, \Gamma$ *to an* $\mathcal{M}$ *proof of the form* $\delta : F, G, \Gamma \Rightarrow D^?/x$. *We then translate the above inference as*

$$\frac{\begin{array}{c}\vdots\\ \delta : F, G, \Gamma \Rightarrow D^?/x\end{array}}{\delta : F \parr G, \Gamma \Rightarrow D^?/x}\ \parr$$

DEFINITION 37 (TRANSLATING THE & RULE IN ALGORITHM 2)

The & *rule is translated in the obvious fashion. If the proofs of* $\delta : F, \Gamma$ *and* $\delta : G, \Gamma$ *have been translated to* $\mathcal{M}$ *proofs with respective roots* $\delta : F, \Gamma \Rightarrow \Pi^?/x$ *and* $\delta : G, \Gamma \Rightarrow \Xi^?/y$ *then we translate the* & *inference as*

$$\frac{\begin{array}{c}\vdots\\ \delta : F, \Gamma \Rightarrow \Pi^?/x\end{array} \quad \begin{array}{c}\vdots\\ \delta : G, \Gamma \Rightarrow \Xi^?/y\end{array}}{\delta : F \,\&\, G, \Gamma \Rightarrow \Pi^? \cap \Xi^?/x \wedge y}\ \&$$

DEFINITION 38 (TRANSLATING THE $\otimes$ RULE IN ALGORITHM 2)

The interesting rule (as always!) is the $\otimes$ *rule. Assume that the the proofs of* $\delta : F, \Gamma$ *and* $\delta : G, \Delta$ *have been translated to* $\mathcal{M}$ *proofs with respective roots* $\delta : F, \Gamma \Rightarrow \Xi^?/x$ *and* $\delta : G, \Delta \Rightarrow \Sigma^?/y$. *In translating the* $\otimes$ *inference there are two cases to consider:*

Case 1 — x is $/false$

We have a use-valid proof of $\delta : F, \Gamma \Rightarrow \Xi^? /false$. By lemma 101 Ξ must be empty, and so we have $\delta : F, \Gamma \Rightarrow /false$. By lemma 113 we have a use-valid proof of $\delta : F, \Gamma_1 \Rightarrow /false$. By lemma 109 we have a use-valid proof of $\delta : F, \Gamma_1, \Delta_1 \Rightarrow \Delta_1 /false$.

We can then apply the inference

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \delta : F, \Gamma_1, \Delta_1 \Rightarrow \Delta_1 /false & \delta : G, \Delta \Rightarrow D^? /y\end{array}}{\delta : F \otimes G, \Gamma, \Delta \Rightarrow D^? /y} \otimes$$

which has the desired conclusion and is a valid inference with provable premises.

Case 2 — x is $/true$

We have a use-valid proof of $\delta : F, \Gamma \Rightarrow \Xi /true$. By lemma 112 we have a use-valid proof of $\delta : F, \Gamma_1 \Rightarrow \Sigma^?, \Pi_1^? /true$ for some Σ and Π. By lemma 108 we have a use-valid proof of $\delta : F, \Gamma_1, \Delta_1 \Rightarrow \Delta_1^{(+?)}, \Sigma^?, \Pi_1^? /true$. Turning to the right hand premise, we have a use-valid proof of $\delta : G, \Delta \Rightarrow D^? /y$. By lemma 114 we have a use-valid proof of $\delta : G, \Delta^{(+?)} \Rightarrow D^?, \Theta^? /y$ for some $\Theta \subseteq \Delta$. By lemma 108 we have a use-valid proof of $\delta : G, \Delta^{(+?)}, \Pi^? \Rightarrow \Pi^?, D^?, \Theta^? /y$. We combine the two proofs using a $\otimes$ inference:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \delta : F, \Gamma_1, \Delta_1 \Rightarrow \Sigma^?, \Pi_1^?, \Delta_1^{(+?)} /true & \delta : G, \Delta^{(+?)}, \Pi^? \Rightarrow \Pi^?, D^?, \Theta^? /y\end{array}}{\delta : F \otimes G, \Gamma, \Delta \Rightarrow \Sigma^?, D^?, \Pi^?, \Theta^? /true} \otimes$$

which has the desired conclusion and is a valid inference with provable premises.

Our next few lemmas combine to give us the desired completeness result.

LEMMA 115

Algorithm 2 works for any proof in $\mathcal{L}$.

Proof: *Check that the algorithm covers all cases.* ■

LEMMA 116

In a proof generated by the above algorithm, if a sequent is tagged $/false$ then it must have an empty residue.

Proof: *A simple induction argument on the structure of the proof. Alternatively, this can be derived as a corollary of lemma 101.* ■

LEMMA 117

Algorithm 2 produces valid proofs in $\mathcal{M}$.

Proof: *Each of the rules except $\&$ and $\otimes$ are translated into an obviously valid $\mathcal{M}$ inference. The $\otimes$ rule is shown to generate valid $\mathcal{M}$ inferences in definition 38. The $\&$ rule in $\mathcal{M}$ however, differs from that produced by Algorithm 2 in that there are extra conditions on $\mathcal{M}$'s $\&$ rule. We need to show that the inferences produced by the algorithm never violate these conditions. The $\&$ inferences produced by the algorithm have $\Pi_x^t = \Xi_x^t = \emptyset$ (The introduction of these is done when processing $\otimes$ rules. Correctness for this has been shown) Hence the only condition in which a proof produced by Algorithm 2 can be invalid is a violation of the last two side conditions; that is x is $/false$ and $\Pi^? \not\subseteq \Xi^?$ or y is $/false$ and $\Xi^? \not\subseteq \Pi^?$. By lemma 116 these conditions are never violated and hence Algorithm 2 is guaranteed to produce valid $\mathcal{M}$ proofs.* ■

THEOREM 118 (COMPLETENESS)

If $\delta : \Gamma$ is provable in $\mathcal{L}$ then there exists a proof in $\mathcal{M}$ of $\Gamma \Rightarrow$.

Proof: *By lemmas 115 and 117 we can translate any $\mathcal{L}$ proof to a valid $\mathcal{M}$ proof.* ■

THEOREM 119 (DETERMINISM)

The $\mathcal{M}$ proof produced by Algorithm 2 is use-valid.

Proof: *Algorithm 2 does not introduce Use rules and all of the lemmas used yield use-valid proofs.* ■

COROLLARY:*If a sequent is provable in $\mathcal{M}$ then it has a use-valid proof. (By theorem 107 the sequent has an $\mathcal{L}$ proof and by this theorem it has a use-valid $\mathcal{M}$ proof).*

4.4 Selecting the Active Formula

When designing a language based on a multiple conclusion logic one can at design time impose the constraint that selecting the active formula be done using "don't care" nondeterminism. Doing this yields a more limited class of formulae but simplifies the implementation. This path was taken in the design of $\mathcal{LC}$ [140] and *LO* [8]. In the design of Lygon the opposite choice was made. As a result Lygon has a significantly larger class of formulae but has to contend with the problem that selecting the active formulae may have to be done using "don't know" nondeterminism.

Although in general selecting the active formulae in Lygon may require backtracking, in certain situations we can safely commit to the selection. The situations in which Lygon can safely use "don't care" nondeterminism to select the active formula can be partially detected by using permutability properties. In particular we use the results of Andreoli [6] and Galmiche and Perrier [44]. This section briefly summarises the results of the two papers. For more details we refer the reader to the papers themselves. Note that there is a fair amount of overlap in the results of the two papers although their motivation is different.

There are three classes of formulae that the results of the two papers indicate can be selected using "don't care" nondeterminism:

1. Asynchronous formulae

2. Synchronous sub-formulae of a synchronous formula

3. Occurrences of $!F$ where the rest of the goal is of the form $?\Delta$

If the topmost connective of a Lygon goal is asynchronous then we can select that goal and commit to the selection. For example, the Lygon goal $\bot, p^\bot \otimes q^\bot, p \parr q$ has a number of proofs. Since $\bot$ is asynchronous we can commit to applying $\bot - R$ first without a loss of completeness. Alternatively, we could select and commit to applying $\parr - R$ first and a proof would be possible:

$$\dfrac{\dfrac{\dfrac{\vdots \quad \vdots}{p^\bot, p \quad q^\bot, q}\otimes - R}{p^\bot \otimes q^\bot, p, q}\parr - R}{p^\bot \otimes q^\bot, p \parr q}\bot - R \qquad \dfrac{\dfrac{\dfrac{\vdots \quad \vdots}{p^\bot, p \quad q^\bot, q}\otimes - R}{p^\bot \otimes q^\bot, p, q}\bot - R}{\bot, p^\bot \otimes q^\bot, p, q}\parr - R$$

Note that as long as there are asynchronous goals we can continue to select an arbitrary asynchronous goal and committing to the selection.

The second optimisation is an application of focusing (see section 2.4). If we have just processed a synchronous connective and a sub-formula is itself synchronous then there is a proof if and only if there is a proof which begins by reducing the synchronous sub-formula. For example the Lygon goal $p \otimes (q \oplus r), p^\bot \oplus r, p \oplus r^\bot$ has a number

of proofs. Assume that we select the first formula – which must be done using "don't know" nondeterminism – for reduction. Then we can select the sub-formula $q \oplus r$ and be guaranteed of finding a proof if one exists.

$$\dfrac{\dfrac{\dfrac{\vdots}{r, r^{\perp}}}{r, p \oplus r^{\perp}} \oplus -R}{q \oplus r, p \oplus r^{\perp}} \oplus -R \qquad \dfrac{\vdots}{p, p^{\perp} \oplus r} \otimes -R}{p \otimes (q \oplus r), p^{\perp} \oplus r, p \oplus r^{\perp}}$$

The third class of formulae occurs rarely in real Lygon programs and so we do not discuss it any further.

These observations (which are incorporated in the current version of Lygon) yield a significant reduction in the nondeterminism associated with selecting the active formula. Benchmarks [150] indicate that the overhead of implementing these strategies is not significant.

The benefit of these strategies varies heavily depending on the program. For programs which do not use $\parr$ there is no benefit as formula selection is always trivial. In certain cases – for example a formula of the form

$$(\mathbf{1} \parr \bot \parr \ldots \parr \bot) \otimes \ldots$$

the benefit is considerable and can reduce the number of solutions and the running time from exponential to linear time.

We now proceed to prove the completeness of the optimisations discussed. Since the optimisations are refinements of the standard sequent calculus for linear logic, soundness follows trivially.

THEOREM 120

Given a proof in the standard sequent calculus there is a proof in the lazy-splitting sequent calculus which has the same structure up to the insertion of Use rules immediately before a formula is reduced.

Proof: *Algorithm 2 maps proofs in the standard sequent calculus to proofs in the lazy-splitting sequent calculus. It is easy to verify that the lazy-splitting proof produced conserves the structure of the proof.* ■

THEOREM 121 (COMPLETENESS OF OPTIMISATIONS)
If there is a proof in the lazy-splitting sequent calculus then a proof search incorporating the observations above will find it.

Proof: *Theorem 6.1 of [44] states that if a sequent is provable then there exists a "normal" proof – that is, a proof which satisfies the optimisations discussed. According to theorem 120 there exists a corresponding use-valid lazy-splitting proof.* ■

4.5 Discussion

We have shown how to eliminate the nondeterminism associated with resource allocation in Lygon. We have also shown how to apply known permutability results in order to reduce the nondeterminism associated with selecting the active formula. Both these optimisations are incorporated in the Lygon interpreter.

Since both of these sources of nondeterminism are exponential, avoiding them is essential for a non-toy implementation. Measurements confirm that these optimisations are significant [150].

Whilst the method presented for splitting resources between sub-branches is optimal (in that the $\otimes$ and $\top$ rules are deterministic) the selection of the formula to be reduced is not. In practice the current system is adequate for the programs we have written.

There are a number of logic programming languages based on linear logic. Some of these, like LinLog [6] are based on proof-theoretic analyses, as Lygon is, but, to the best of my knowledge, have not been implemented and hence do not involve the problems of lazy splitting discussed in this thesis.

Others, like ACL [85, 88, 128] and LO [8–10] use linear logic as motivation and a design guide for concurrent logic programming. These languages use a somewhat restricted class of formulae which excludes $\otimes$. As a result the implementation problems are correspondingly simpler, in that neither language needs lazy splitting[1], and so again the problems addressed in this paper do not arise.

The last class of linear logic programming languages comprises Lolli [68–70], Forum [109] and Lygon [57, 122, 149]. These languages attempt to take a large fragment

[1] Actually ACL includes $\otimes$ but only in contexts which prevent lazy splitting from being needed.

of linear logic, implement it and show that the resulting language is expressive and useful.

The notion of lazy splitting was introduced in Lolli [69], and a lazy method of finding uniform proofs (known in [69] as an input-output model of resource consumption) is given. An extension of this system presented in Hodas' thesis [66] handles the $\top$ rule lazily. In [32] the notion of lazy splitting is extended to improve the early detection of failure due to inconsistent resource usage in the two premises of a $\&$ rule. It is not clear to what extent "typical" Lolli or Lygon programs benefit from this optimisation. On the other hand, optimising the behavior of $\top$ was motivated by observed inefficiency in typical programs.

Lolli is based on a single conclusion logic whereas Lygon is based on a multiple conclusion logic. It is actually possible to encode Lygon into Lolli by using a new constant and allowing resolution to select the formula to be reduced. The goal $a \parr b$ is translated to $((a \multimap n) \otimes (b \multimap n)) \multimap n$ where n is a new constant.

We feel however, that a direct approach is more desirable for a number of reasons. Firstly – as discussed in section 3.1 – there are problems associated with the blind use of equivalences derived using full linear logic when the proof search process is goal directed. The relationship between goal directed (i.e. operational) equivalence and logical equivalence is nontrivial and involves intermediate logics [53]. The logical equivalence of two formulae only implies operational equivalence if the two formulae are within the appropriate subset. For example, the two formulae

$$(a \oplus b) \multimap (a \oplus b)$$

and

$$(a \multimap (a \oplus b)) \& (b \multimap (a \oplus b))$$

are logically equivalent; however the first is not a valid goal formula and does not have a uniform proof even though the second does have a uniform proof.

Secondly, and perhaps more importantly, our presentation allows us to simply use the permutability properties explored in [6, 44] to reduce the nondeterminism associated with selecting the formula to be reduced. In the Lolli encoding, selecting the next formula to be reduced is done by the resolution rule. Reducing the nondeterminism under the Lolli

encoding would require adding a special case to the resolution rule which examines the body of the clause and accordingly decides whether the clause can be committed to.

Additionally, our solution handles the Lolli language as a simple case in which all sequents just happen to have a single formula. Our solution is also applicable to Forum [71].

In addition to the application of this work to the implementation of logic programming languages based on linear logic, the lazy rules above presumably have application in theorem provers for linear logic [95, 113, 135, 136], where they eliminate a significant potential source of inefficiency.

Chapter 5

Applications of Lygon

In this chapter we look at applications of Lygon. The first aim of this chapter is to demonstrate that Lygon's linear logic features buy significant expressiveness and allow simpler solutions to a range of problems. The second aim of the chapter is to develop a methodology of Lygon programming.

Finding ways of using new programming constructs can be highly non-trivial. Programming idioms such as monads in functional programming languages [141, 142] have taken a number of years to emerge. Other examples of non-obvious programming idioms include combinator parsers in functional programming languages [78], difference lists in logic programming languages *etc.* [117, 134].

In our examples we develop a number of linear logic programming idioms – particular ways of using linear logic connectives to achieve a particular behavior. After introducing Lygon from the programmer's perspective, we introduce a number of basic idioms. We then illustrate applications of Lygon to Graph problems, Concurrency and Artificial Intelligence. In the process of developing solutions to problems we develop programming idioms and illustrate how they are used. We present a Lygon meta-interpreter and then finish with some miscellaneous examples and discussion.

All of the programs in this chapter are automatically typeset from executable code and have been run and tested under the Lygon interpreter.

History

Lygon grew out of some work on the proof-theoretic foundations of logic programming languages. The fundamental idea behind Lygon is the completeness properties of certain kinds of proofs in the linear sequent calculus, and the initial work in this area was done in the second half of 1990. Over the period of the next two years, the operational model of the language was defined, revised, extended, and, in part, applied to other logics as well, and the language received its name over dinner one night late in 1992.

The first Lygon implementation appeared in the following year, although there were still some technical problems with the operational semantics, which were ironed out in early 1994, and a prototype implementation was completed later that year.

Lygon the programming language is named after Lygon Street. Lygon street is close to Melbourne University and is known for its restaurants and cafes. There is also a Lygon Road in Edinburgh which is familiar to Harland and Pym. Current information about Lygon can be found on the web [147]. Information on the Lygon language and implementation can be found in [146].

5.1 Lygon - A Programmer's Perspective

It is worth emphasising that at this stage of the language's development our primary aim is to experiment with the linear logic features of the language and as a consequence the other aspects of the language are not as important. In the design and implementation of the language we have inherited freely from Prolog. For example, the handling of I/O is impure and the language is not statically typed or moded. A rough slogan that has guided the design to date is *"Lygon = Prolog + Linear Logic"*. Since the linear logic aspects are orthogonal to issues of types, modes and language purity there is no reason for Lygon to be impure – future work on Lygon may well adopt strong types and modes a la Mercury [132, 133].

Syntax

Lygon syntax is similar to Prolog syntax with the main difference that goals and the bodies of clauses are a (subset) of linear logic formulae rather than a sequence of atoms.

Program clauses are assumed to be reusable (i.e. nonlinear). It is possible to specify program clauses that must be used exactly once in each query. This is done by prefixing the clause with the keyword **linear**.

Lygon syntax is described by the following BNF. The notation [*x*] indicates that *x* is optional. As is traditional in logic programming we use a reversed implication ($\leftarrow$) to write clauses. We define $F \leftarrow G$ as shorthand for $!(G \multimap F)$.

Note that the following syntax is a subset of the full Lygon$_2$ syntax derived in section 3.11. This syntax describes what the Lygon interpreter handles. It differs from the full language by the omission of

1. Program clauses of the form $C_1 \mathbin{\&} \ldots \mathbin{\&} C_n$
2. Goals of the form $D \multimap G$ – these can be replaced with $D^{\perp} \parr G$.
3. Goals of the form $?G$
4. Goals of the form $\forall x G$
5. Goals of the form $D^{\perp}$ – two special cases (**neg** A and $?$ **neg** D) are handled.

$$
\begin{array}{rcl}
G & ::= & G \otimes G \mid G \oplus G \mid G \mathbin{\&} G \mid G \parr G \mid\ !\,G \\
 & & \mid\ ?\ \textbf{neg}\ D \mid \mathbf{1} \mid \bot \mid \top \mid A \mid \textbf{neg}\ A \\
D & ::= & [\textbf{linear}]\ (A_1 \parr \ldots \parr A_n \ \leftarrow\ G)
\end{array}
$$

Lexical syntax and comments follow the underlying Prolog system. For example: $p \leftarrow q(X)$ is a valid program clause and $p \otimes (q(1) \parr (a \oplus b))$ is a valid query.

Semantics

The semantics of Lygon from the programmer's perspective can best be explained by recourse to an abstract interpreter. An abstract Lygon interpreter operates in a simple cycle:

1. Select a formula to be reduced

2. Reduce the selected formula

This repeats until there are no goals left.

The selection step does not occur in Prolog. As discussed in section 4.4, a Lygon goal may consist of a number of formulae. At each step, we reduce one of these formulae. In general which formula we choose to reduce first *can* make a difference to whether a proof is found. The Lygon interpreter thus may need to select a formulae, and – if the proof attempt fails – backtrack and select another formula.

The interpreter attempts to reduce this nondeterminism where possible using known properties of linear logic. This is outlined in section 4.4 and is summarised below.

After selecting a formulae the interpreter reduces the selected formula. The reduction used is dependent entirely on the top level connective of the selected formula. In the rules below we denote the linear context (i.e. the rest of the goal and any linear parts of the program) by *C*.

When we say *"look for a proof of ..."* we mean that the goals indicated *replace* the goal being reduced. The reduction of a formula in a Lygon goal is governed by the following rules:

$A \otimes B$: Split C into $C1$ and $C2$ and look for proofs of $(C1, A)$ and $(C2, B)$.

$A \parr B$: Look for a proof of (C, A, B).

$A \,\&\, B$: Look for proofs of (C, A) and (C, B).

$A \oplus B$: Look for either a proof of (C, A) or a proof of (C, B).

$\top$: The goal succeeds.

$\bot$: Look for a proof of (C).

1: The goal succeeds *if* the context C is empty.

A: *resolve* (see below).

? **neg** A: Add A to the program and look for a proof of (C).

$!A$: If C is empty then look for a proof of (A) otherwise fail.

EXAMPLE 17
The goal $((\mathbf{1} \oplus \bot) \parr (\mathbf{1} \with \top))$ *has a number of proofs. One proof involves the following steps:*

1. *There is only one formula so we select it. We then apply the* $\parr$ *rule yielding a goal consisting of the two formulae* $\mathbf{1} \oplus \bot$ *and* $\mathbf{1} \with \top$.
2. *There are now two formulae in the goal. Let us select the first. We apply the* $\oplus$ *rule yielding the goal* $(\bot, (\mathbf{1} \with \top))$.
3. *Assume we now decide to select the second formula. We apply the* $\with$ *rule yielding two goals:* $(\bot, \mathbf{1})$ *and* $(\bot, \top)$.
4. *We now have two goals which need to be proved independently. Let us consider the second goal first since it is the simpler of the two. We can choose the second formula and apply the* $\top$ *rule. The second goal succeeds and we are left with the first goal.*
5. *The remaining goal is* $(\bot, \mathbf{1})$. *We attempt to select the second formula but find that the context is nonempty and we can not apply the* **1** *rule.*
6. *We backtrack and select the first formula. The* $\bot$ *rule yields the goal* $(\mathbf{1})$ *which is provable using the* **1** *rule.*

Resolution

An atom can be proven in a number of ways:

1. It can be a built in predicate. Builtins are equivalent to **1** in that they can only succeed in an otherwise empty context. Builtins differ from **1** only in that they may bind variables and may fail (for example, *append([1],[2],X)* binds *X* and *append([1],[2],[3])* fails).

2. It can match the axiom rule. The axiom rule states that a context of the form $(A,$ **neg** $B)$ where A and B unify is provable. Note that the context must not have any formulae other than A and **neg** B.

3. It can match a program clause. In this case, we nondeterministically select a (unifiable) program clause, create fresh variables for it, unify its head with the atom and replace the atom with its body. This behaviour is identical to Prolog's.

Multiple headed clauses (that is, clauses of the form $p \parr q \leftarrow \ldots$) are a straightforward extension. We create fresh variables for the clause and then unify each of the atoms in its head with a different atom in the goal. If all unifications are successful then the atoms unified against are replaced with the body of the clause.

EXAMPLE 18
Our goal is $(p(1), q(2), q(1))$ *and our program is*

p(X) $\parr$ q(X) $\leftarrow$ **is**(X1,X+1) $\otimes$ p(X1).

We select the program clause and attempt to resolve. We firstly unify $p(X)$ *and* $p(1)$ *yielding the substitution* $X = 1$*. We then attempt to unify* $q(2)$ *and* $q(X)$*. Since* X *has been bound to* 1 *this unification fails and we attempt to unify the other atom –* $q(1)$ *– with* $q(X)$*. This succeeds and the two atoms –* $p(1)$ *and* $q(1)$ *– are removed and replaced with the clause's body yielding the goal* $(q(2),$ **is**$(X1, X + 1) \otimes p(X1))$*.*

EXAMPLE 19
Consider the program:

toggle $\leftarrow$ (off $\otimes$ **neg** on) $\oplus$ (on $\otimes$ **neg** off).
linear on.

The goal toggle $\parr$ *off is provable as follows:*

1. *Firstly, we add the linear parts of the program to the goal. The linear program fact* (**linear** *on*) *is equivalent to adding* (**neg** *on* $\parr$) *to the goal.*

2. *Our goal then is* (**neg** *on* $\parr$ *toggle* $\parr$ *off*)*. Using two applications of the* $\parr$ *rule we obtain a goal consisting of three formulae:* (**neg** *on*, *toggle*, *off*)*.*

3. *There is never any point in selecting formulae of the form* **neg** A *as there is no corresponding reduction rule. Hence we consider the other two formulae. Selecting off will not work - it is not a builtin, there is no program clause and we can not use the axiom rule.*

4. *We therefore select toggle. Using the program clause we obtain the goal* $(\textbf{neg}\ on, off, ((off \otimes \textbf{neg}\ on) \oplus (on \otimes \textbf{neg}\ off)))$.

5. *We select the compound formula and use the* $\oplus$ *rule to reduce the goal to* $(\textbf{neg}\ on, off, (off \otimes \textbf{neg}\ on))$.

6. *We select the compound formula and use the* $\otimes$ *rule. In applying this rule we have a choice as to how we split the context* C*. The Lygon implementation does this splitting lazily and deterministically. In this case it determines that no matter which way we split the linear context a proof is not possible.*

7. *We backtrack and try the other alternative of the* $\oplus$ *rule. The goal is now* $(\textbf{neg}\ on, off, (on \otimes \textbf{neg}\ off))$.

8. *We apply the* $\otimes$ *rule. This gives us the two goals* $(\textbf{neg}\ on, on)$ *and* $(off, \textbf{neg}\ off)$ *both of which are provable using the axiom rule.*

Selecting the Formula to be Reduced

When selecting the formula to reduce in a goal, the Lygon interpreter uses a number of heuristics to reduce nondeterminism (see section 4.4).

1. If there are formulae in the goal whose top level connectives are asynchronous then we can select any one of these formulae and commit to this selection.

2. If we have just processed a synchronous connective and a sub-formula is itself synchronous then we can select the sub-formula for reduction immediately and commit to this selection. This is due to the focusing property.

EXAMPLE 20

In the goal $(a \otimes b, c \parr d, p(X))$ *we can select the second formula and commit to it. In the goal* $(a \otimes b, p(X) \otimes (q(X) \otimes r(X)))$ *if we select the second formula we may need to backtrack and try the first; however after processing the outermost* $\otimes$ *we can safely select the sub-formula* $q(X) \otimes r(X)$ *using focusing.*

Built In Predicates

The Lygon interpreter provides a number of primitive predicates for operations such as arithmetic, meta-programming and I/O. These are:

- **print**/1: prints its argument.
- **nl**/0: prints a newline.
- **input**/1: reads a Prolog term and binds it to its argument. This is not usable under the graphical user interface.
- **system**/1: passes its argument to the system to be executed as a command.
- **is**/2: evaluates its second argument as a mathematical expression and binds the result to the first argument. This predicate is inherited from Prolog.
- **lt**/2: succeeds if both its arguments are numbers and the first is less than the second.
- **builtin**/1: succeeds if its argument matches a builtin.
- **readgoal**/1 and **readprog/1**: read terms and convert them to Lygon goals and programs respectively. The difference from **input**/1 is that atoms are tagged. Example: If the user types `p * q # r.` then the call to **readgoal**/1 returns `atom(p)*atom(q)#atom(r)`. These are not usable under the GUI.
- **prolog**/1: passes the argument for evaluation by the underlying prolog system. This can be used to extend the set of builtins.
- **tcl**/1: passes its argument to TCL/Tk for evaluation. This can be used to do graphics from within Lygon or to extend the user interface.

- **call**/1: **call**(X) is equivalent to X. It is needed since internally Lygon represents the atomic goal *p(X)* as *atom(p(X))*.

In addition to the built in predicates the example programs given will assume the presence of the standard Lygon library shown in program 1. Note that a number of the predicates are impure and typical of Prolog code.

5.1.1 The Lygon Implementation

The Lygon implementation (current version: 0.7.1) consists of two components:

1. The Lygon interpreter, and

2. A user interface.

The Lygon interpreter is structured as a Prolog meta-interpreter. The emphasis is on ease of implementation rather than on performance or robustness. The interpreter is written in BinProlog[1] and consists of some 724 lines of code[2].

The core of the interpreter is essentially "just" an implementation of the rules of $\mathcal{M}$ which were derived in chapter 4 (see figure 4.1 on page 122).

The user interface is written in TCL/Tk[3] and consists of some 591 lines of code. The graphical user interface is depicted in figure 5.1. The user interface and interpreter communicate via Unix pipes.

The system integrates the Lygon four port debugger developed by Yi Xiao Xu [153]. The debugger provides an execution trace which can be filtered according to a number of criteria.

The Lygon programs presented were mostly tested under Lygon 0.7.1. A few (those requiring textual input for example) were run under Lygon 0.4. The implementation is available from the Lygon World Wide Web page [147] at
`http://www.cs.mu.oz.au/~winikoff/lygon`. For more details on the Lygon implementation and its use see [146].

[1] Available from `http://clement.info.umoncton.ca/~tarau/`
[2] As counted by `wc`.
[3] Available from `http://www.sunlabs.com:80/research/tcl/`

Program 1 Standard Lygon Library

Lygon Standard Library

Negation as failure
not(X) ← **once**((**call**(X) ⊗ **eq**(Y,succeed)) ⊕ **eq**(Y,fail)) ⊗ **eq**(Y,fail).

var(X) ← **not**(**not**(**eq**(X,1))) ⊗ **not**(**not**(**eq**(X,2))).

le(X,Y) ← **lt**(X,Y) ⊕ **eq**(X,Y).
gt(X,Y) ← **lt**(Y,X).
ge(X,Y) ← **lt**(Y,X) ⊕ **eq**(X,Y).

output(X) ← (!(**print**(X) ⊗ **nl**)).

repeat ← **1** ⊕ repeat.

sort([],[]).
sort([X],[X]).
sort([X,Y|Xs],R) ← halve([X,Y|Xs],A,B) ⊗ sort(A,R1) ⊗ sort(B,R2) ⊗ merge(R1,R2,R).

merge([],X,X).
merge([X|Xs],[],[X|Xs]).
merge([X|Xs],[Y|Ys],[X|R]) ← le(X,Y) ⊗ merge(Xs,[Y|Ys],R).
merge([X|Xs],[Y|Ys],[Y|R]) ← gt(X,Y) ⊗ merge([X|Xs],Ys,R).

halve([],[],[]).
halve([X],[X],[]).
halve([X,Y|R],[X|A],[Y|B]) ← halve(R,A,B).

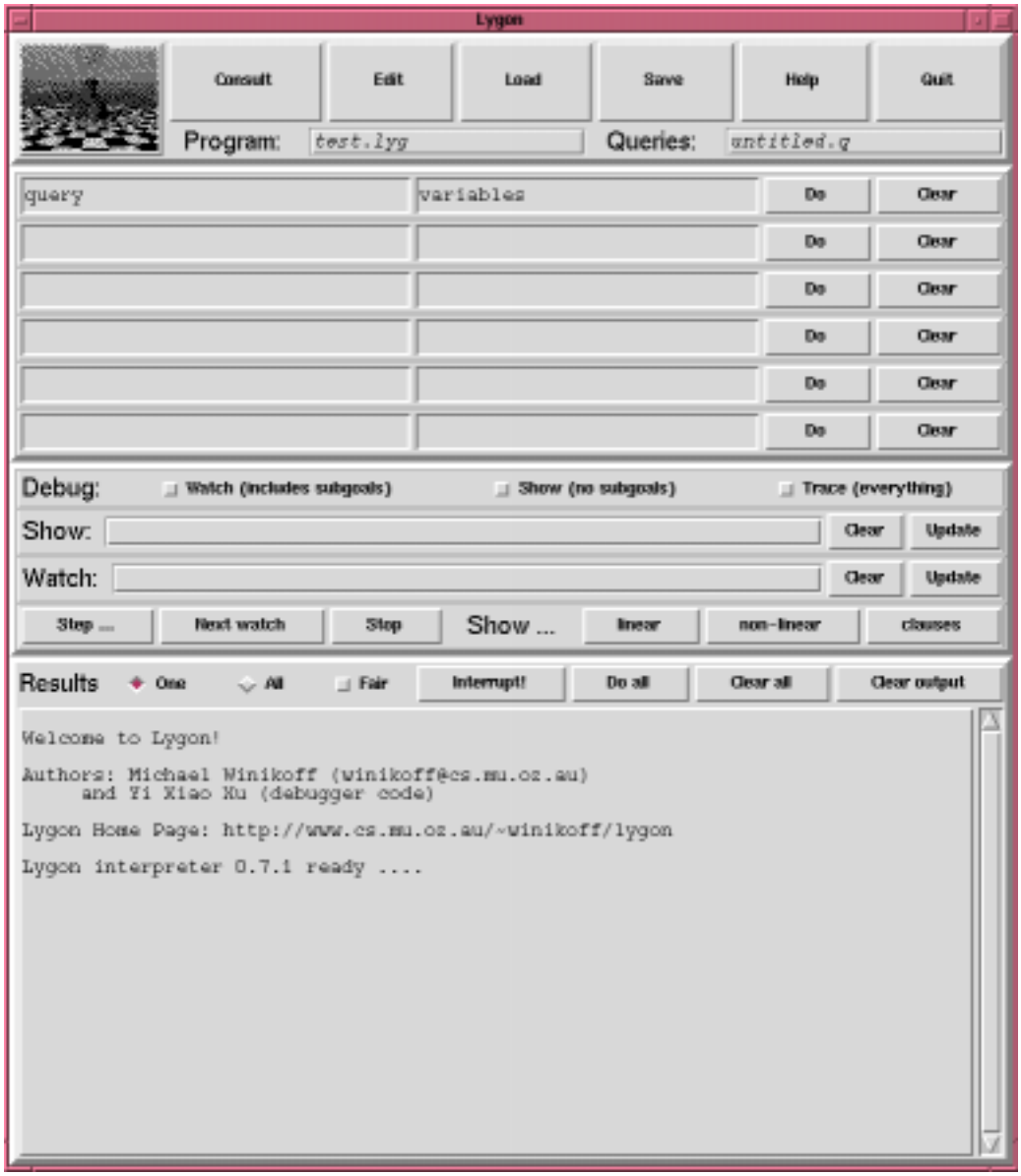
append([],X,X).
append([X|Xs],Y,[X|Z]) ← append(Xs,Y,Z).

reverse(X,Y) ← var(Y) ⊗ rev(X,[],Y).
reverse(X,Y) ← **not**(var(Y)) ⊗ rev(Y,[],X).

rev([],X,X).
rev([X|Xs],Y,R) ← rev(Xs,[X|Y],R).

member(X,[_|Y]) ← member(X,Y).
member(X,[X|_]).

Figure 5.1 The Lygon User Interface



5.2 Basic Techniques

We begin by presenting a number of simple Lygon programs. Some of these demonstrate generally useful programming idioms which will be used in later sections.

The main theme in this section is the simple manipulation of the linear context. The linear context is a multiset of linearly negated atoms and can be thought of as a collection of atomic facts. The two differences between the linear context and a program are that (i) adding facts to the linear context is a logically pure operation, and (ii) linear facts are consumed when they are used. This second difference is crucial in that it allows *replacement* of old information with new. By contrast, classical logic allows pure addition of facts (using implication) but only allows for the *extension* of the program. As we shall see, the linear context can model general state based operations.

We begin with a simple example which uses the linear context to store some state information. Program 2 (taken from [68, 70]) stores a single bit of information and toggles it. The effect of a call to *toggle* in a context containing the fact *off* is to consume the fact and add the fact *on*.

Note that since Lygon programs are implicitly nonlinear we must supply the initial state as part of the goal. Adding *off* as a (non-linear) program clause would mean that even after a toggle we could still prove *off* from the clause. Although it is possible to avoid this by having *off* as a **linear** program clause we choose not to do this. The problem with using linear program clauses is that *every* query gets given an initialised linear context. Using linear clauses can make the program easier to run for a given goal but reduces our ability to make other queries.

The derivation of the goal **neg** *off* $\parr$ *toggle* $\parr$ *on* has been described in example 19. The key point is that once *toggle* has been called it is no longer possible to prove *off*.

Program 2 illustrates a common special case of $\otimes$. As noted in the previous section, the proof of the goal $A \otimes B$ partitions the context between the proof of A and the proof of B. There are two special cases that are common in Lygon programs. These are when A is a builtin and when A is an atom which has corresponding linear facts. In the first case the entire context gets passed to B since builtins are logically equivalent to **1** and can not use any of the context. In the second case a single linear fact matching A is used

Program 2 Toggling State

toggle $\leftarrow$ (off $\otimes$ **neg** on) $\oplus$ (on $\otimes$ **neg** off).

go $\leftarrow$ **neg** off $\parr$ toggle $\parr$ show.

show $\leftarrow$ off $\otimes$ **print**('`off`').
show $\leftarrow$ on $\otimes$ **print**('`on`').

by the axiom rule in the proof of A and the remaining context is used in the proof of B.

For example consider the goal (**neg** p(1) , **neg** q , p(X) $\otimes$ **print**(X) $\otimes$ $\top$) The only formula which can be usefully selected is the third. This gives us the two goals (**neg** p(1), p(X)) and (**neg** q , **print**(X) $\otimes$ $\top$). The first is provable using the axiom rule which unifies X with 1. In proving the second goal we again use the $\otimes$ rule yielding the two new goals (**print**(1)) and (**neg** q , $\top$). The first of these succeeds and as a side effect prints 1. The second succeeds using the $\top$ rule.

The net effect of *p* $\otimes$ *G* (where *p* is not defined by a program clause) is to remove a linear fact matching *p* from the context and pass the rest of the context to *G*.

Another connective in Lygon that is often used in a particular way is $\parr$. Recall that the proof of the goal $A \parr B$ simply looks for a proof of A, B. If A and B are both atoms which have matching program clauses then the two formulae evolve in parallel. A commonly occurring special case is (**neg** A) $\parr$ B which adds the linear fact A to the context and then continues with B. This is equivalent to $A \multimap B$.

A third connective that is often used in a certain way is !. Recall that the goal ! *G* can be proven if (a) *G* can be proven, and (b) the linear context is empty. Thus ! is used to ensure that the goal *G* is executed with an empty linear context. The pattern *H* $\otimes$! *G* forces *H* to consume all of the linear resources. An example is the coding of *output* in program 1 where the goal ! *(print(R)* $\otimes$ *nl)* ensures that the linear context has been consumed. This is important since otherwise it is possible for *print* to succeed and pass linear resources to *nl*. Only when *nl* is executed does it become apparent that the linear resources cannot be consumed and the computation backtracks. For example, the goal *(*$\bot \oplus$ **neg** *p) $\parr$ (print(x)* $\otimes$ *nl)* prints two `x`s followed by a newline whereas the goal *(*$\bot \oplus$ **neg** *p) $\parr$!(print(x)* $\otimes$ *nl)* prints a single `x` followed by a newline.

Before we move on to look at how linear facts can be collected, it is worth noting that since Lygon is an extension of Prolog, any (pure) Prolog program will run under Lygon. In general Lygon's $\otimes$ connective substitutes for Prolog's conjunction ("`,`"). Program 3 is a well known Prolog program transcribed into Lygon.

Program 3 Naïve Reverse

```
append([],X,X).
append([X|Xs],Y,[X|Z]) ← append(Xs,Y,Z).

nrev([],[]).
nrev([X|Xs],R) ← nrev(Xs,R2) ⊗ append(R2,[X],R).
```

The linear context can be used to store an initial set of facts which give the input to the program and after the program has run could contain the resulting output. We need to be able to report on the contents of the linear context. This can be done by *collecting* the linear facts into a data structure and then returning it or printing it.

Consider a linear context which is known to contain any number of unary *p* facts and binary *q* facts. One way of collecting these into a list is the following (see program 4):

```
collect(X,Y) ← p(A) ⊗ collect([p(A)|X],Y).
collect(X,Y) ← q(A,B) ⊗ collect([q(A,B)|X],Y).
collect(X,X).
```

This predicate is called as *collect([],X)*. The linear context is consumed and a list is bound to *X*. An alternative encoding is

```
collect([p(A)|X]) ← p(A) ⊗ collect(X).
collect([q(A,B)|X]) ← q(A,B) ⊗ collect(X).
collect([]).
```

Here, the predicate is called as *collect(X)* and *X* is bound to a list as before. Note that the program clause *collect([])* is equivalent to *collect([])* $\leftarrow$ **1**. This clause can only succeed once the linear context has been emptied.

One problem with both of these definitions concerns backtracking. The elements of a multiset are not in a defined order whereas the elements of a list are. In converting from

a multiset to a list we are adding an order. Both versions of *collect* given are capable of generating any order and will do so upon backtracking. For example, given the context *(p(1),p(2),q(a,b))* both versions of collect will generate all six orderings of the elements. If we would like to be able to find alternative solutions to the Lygon query which generated the relevant linear context then this behavior is a problem – before returning a new solution we will be given permutations on the order of the elements in the multiset.

We solve this problem by using **once** to select an arbitrary ordering of the multiset and prune away other solutions. Program 4 shows two different ways of coding *collect*. These two differ in the order of the solutions returned. Since the ordering is arbitrary anyway this is not important.

The goal *go1(X)* has the single solution *X = [c,b,a,a]* and the goal *go2(X)* has the single solution *X = [a,a,b,c]*.

Program 4 Collecting Linear Facts

go1(X) ← **neg** a ⅋ **neg** b ⅋ **neg** c ⅋ **neg** a ⅋ collect([],X).

First version -- called as collect([],X)
collect(X,Y) ← get(Z) ⊗ collect([Z|X],Y).
collect(X,X).

get(X) ← **once**((**eq**(X,a) ⊗ a) ⊕ (**eq**(X,b) ⊗ b) ⊕ (**eq**(X,c) ⊗ c)).

go2(X) ← **neg** a ⅋ **neg** b ⅋ **neg** c ⅋ **neg** a ⅋ collect(X).
Second version -- called as collect(X)
collect([Z|X]) ← get(Z) ⊗ collect(X).
collect([]).

A related problem which is non-trivial to solve in Prolog – and which has been used to motivate the need for combining embedded implications and negation as failure – involves determining whether a number of clauses of the form *r(1), . . . ,r(n)* contains an odd or even number of clauses.

The following proposed solution [25, 40] uses implication to add a `mark` to facts which have been processed and uses negation as failure to ensure that marked facts are

not processed a second time.

```
even :- not odd.
odd :- select(X), (mark(X) => even).
select(X) :- r(X), not mark(X).
```

The Lygon solution to this problem simply combines programs 2 and 4. We need to collect each program clause; however instead of constructing a list of the collected clauses we simply *toggle* an even/odd indicator for each clause collected, see program 5.

The goal **neg** *count(even)* ⅋ **neg** *r(1)* ⅋ **neg** *r(2)* ⅋ *check(X)* returns the answer *X = even*. Once *check(X)* has consumed all of the $r(i)$ facts it reduces (using the second clause) to *count(X)* and the axiom rule unifies *X* with the odd/even indicator stored in the linear fact.

Program 5 Counting Clauses

check(Y) ← **once**(r(X)) ⊗ (toggle ⅋ check(Y)).
check(X) ← count(X).

toggle ← (count(even) ⊗ **neg** count(odd)) ⊕ (count(odd) ⊗ **neg** count(even)).

An alternative solution in Lygon which is closer to the solution using implications and negations is given in program 6. An empty context (i.e., zero clauses) is even. A context is even if the context with one less fact is odd and vice versa. The goal **neg** *r(1)* ⅋**neg** *r(2)* ⅋ *check(X)* returns *X = even*.

Program 6 Counting Clauses – Alternative Version

check(even) ← even.
check(odd) ← odd.

even.
even ← **once**(r(X)) ⊗ odd.
odd ← **once**(r(X)) ⊗ even.

A property that is used in programs 4 and 5 is that all linear facts must by default be used *exactly* once. In certain situations (for example finding paths as described in the

following section) it is desirable to allow facts to be used at *most* once, that is, we would like to be able to ignore certain facts. This behavior is known as *affine* since it is precisely what affine logic [89] provides.

We can simulate affine facts by using $\top$. Recall that $\top$ succeeds in any context. In some sense it can be thought of as a "cookie monster" which consumes all the linear resources it is given.

If we have a goal G and a linear context Γ then the goal $\Gamma \parr (G \otimes \top)$ allows G to use a *subset* of Γ. Intuitively, Γ is split between G and $\top$ – any linear facts which are not used in G are consumed by $\top$.

Although intuition suggests that $\top$ must occur "after" G, in actual fact the goal $\Gamma \parr (\top \otimes G)$ works equally well. What happens is that the tags described in chapter 4 are used to pre-weaken Γ. The lazy $\top$ rule marks all of the formulae in Γ as affine and then passes them to G.

A useful variation on this allows us to combine affine and linear facts. Suppose Δ contains linear formulae which must be used and Γ contains formulae which we may ignore. Then the formula

$$\Gamma \parr (\top \otimes (\Delta \parr G))$$

captures the desired behavior. For example, in program 7 *go1* and *go2* succeed but *go3* fails. This sort of behavior is used in program 12.

Program 7 Affine Mode

go1 $\leftarrow$ **neg** p $\parr$ ($\top$ $\otimes$ (**neg** q $\parr$ (q))).
go2 $\leftarrow$ **neg** p $\parr$ ($\top$ $\otimes$ (**neg** q $\parr$ (q $\otimes$ p))).
go3 $\leftarrow$ **neg** p $\parr$ ($\top$ $\otimes$ (**neg** q $\parr$ (p))).

We have seen how the linear context can be used to store simple state. We now show how a library of operations can be constructed which uses the linear context to simulate an imperative store.

Consider a binary linear fact *m(A,V)* where *A* represents the address of the memory cell and *V* its value. The goal *m(1,*π*)* $\multimap$ *G* adds memory cell 1 containing a certain well known value and proceeds with *G*. In Lygon this would be written as **neg** *m(1,*π*)* $\parr$ *G*.

The goal *m(1,X)* $\otimes$ *G* binds *X* to the value in memory cell 1 and proceeds with *G*. Note that this read operation is *destructive.* In order to allow *G* to re-read the memory cell we must recreate it using a goal of the form *m(1,X)* $\otimes$ *(***neg** *m(1,X)* $\parr$ *G).*

These goals can be encapsulated into an abstract data type for an imperative store using the operations *newcell/3*, *lookup/3* and *update/3*. Note that we need to be able to specify that goals happen in a certain sequence. We do this by using a continuation passing style.

Program 8 State Abstract Data Type

newcell(Id,Value,Cont) $\leftarrow$ **neg** m(Id,Value) $\parr$ **call**(Cont).
lookup(Id,Value,Cont) $\leftarrow$ m(Id,Value) $\otimes$ (**neg** m(Id,Value) $\parr$ **call**(Cont)).
update(Id,NewValue,Cont) $\leftarrow$ m(Id,_) $\otimes$ (**neg** m(Id,NewValue) $\parr$ **call**(Cont)).

As an example using the above code, consider the following simple imperative code to sum a list of numbers

```
sum  := 0;

while sumlist not empty do
begin
  sum := sum + head sumlist;
  sumlist := tail sumlist;
end

return sum;
```

This can be written in Lygon quite simply (see program 9). Note that we need a $\top$ in the second clause of *sumlist* to delete the memory cells once we are finished with them. This could be added to the state ADT as another operation. The goal *sum([1,5,3,6,7],X)* yields the solution *X = 22*.

Program 9 State Based Sum

sum(List,Result) $\leftarrow$ newcell(sum,0, newcell(sumlist,List, sumlist(Result))).
sumlist(R) $\leftarrow$ lookup(sumlist,[N|Ns], lookup(sum,S,
(**is**(S1,S+N) $\otimes$ update(sum,S1, update(sumlist,Ns, sumlist(R))))))).
sumlist(R) $\leftarrow$ lookup(sumlist,[], lookup(sum,S, **eq**(S,R) $\otimes$ $\top$)).

The final example in this section applies the state based idiom to solve a problem with I/O. Programs in an impure logic programming language such as Prolog do input and output using side effects. One problem with this is that output performed in sub-computations which fail and backtrack is visible. The output produced by a Prolog program which backtracks over I/O operations can be confusing.

In Lygon it is possible to maintain an output list in the linear context and only print the list once the relevant sub-computation has succeeded. This technique is particularly useful in concurrent Lygon programs (see section 5.4) which tend to exhibit a large number of solutions corresponding to all possible interleaved executions.

We begin by creating a linear fact *t([])*. Goals of the form **print***(X)* $\otimes$ *G* are replaced by *pr(X,G)* where *pr* is defined as (see program 10):

pr(X,G) $\leftarrow$ t(R) $\otimes$ (**neg** t([X|R]) $\parr$ **call**(G)).

Note that *pr* is just an imperative style *lookup* followed by an *update*. The query *go* succeeds once and prints 2 and 3. If **print** had been used it would have printed 1 as well.

The role of the $\bot \otimes$ *t(X)* $\otimes \ldots$ is to allow the goal (*backtrack*) to succeed normally and collect the printing to be done using a proof of the following form:

$$\dfrac{\dfrac{\vdots}{\mathbf{1}, \bot} \quad \dfrac{\dfrac{}{\textbf{neg } \text{t([2,3])}, \text{t(X)}}\, Ax, \textit{X=[2,3]} \quad \vdots}{\textbf{neg } \text{t([2,3])}, \text{t(X)} \otimes \ldots}\, \otimes -R}{\mathbf{1}, \textbf{neg } \text{t([2,3])}, \bot \otimes \text{t(X)} \otimes \ldots}\, \otimes -R$$

$$\vdots$$

$$\text{backtrack}, \textbf{neg } \text{t([])}, \bot \otimes \text{t(X)} \otimes \ldots$$

A general idiom that is demonstrated in this program (and in program 9) is the use of a continuation passing style to encode sequentiality. If we wish to make a change to the linear context and then run a goal then we cannot write *makechange* $\otimes$ *goal* since additions made to the context in *makechange* are not visible to the goal. If we write *makechange* $\parr$ *goal* then the changes made are visible to the goal but the goal could also use the old state. The way to obtain correct sequencing is to modify *makechange* to accept an extra argument and to **call** that argument when it has finished changing the context. We then write *makechange(goal)*.

Program 10 Batching Output

go $\leftarrow$ **neg** t([]) $\parr$ backtrack $\parr$ ($\bot \otimes$ t(X) $\otimes$!(reverse(X,RX) $\otimes$ **print**(RX) $\otimes$ **nl**)).
pr(X,G) $\leftarrow$ t(R) $\otimes$ (**neg** t([X|R]) $\parr$ **call**(G)).
backtrack $\leftarrow$ pr(1,fail) $\oplus$ pr(2,pr(3,**1**)).

5.3 Graphs

Graphs are an important data structure in computer science. Indeed, there are many applications of graph problems, such as laying cable networks, evaluating dependencies, designing circuits and optimization problems. The ability of Lygon to naturally state and satisfy constraints, such as that every edge in a graph can be used at most once, means that the solution to these problems in Lygon is generally simpler than in a language such as Prolog.

This observation was made independently by Paul Tarau in the later versions of BinProlog which include a form of linear[4] predicates.

One of the simplest problems involving graphs is finding paths. The standard Prolog program for path finding is the following one, which simply and naturally expresses that the predicate `path` is the transitive closure of the predicate `edge`, in a graph.

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Whilst this is a simple and elegant program, there are some problems with it. For example, the order of the predicates in the recursive rule is important, as due to Prolog's computation rule, if the predicates are in the reverse order, then goals such as `path(a,Y)` will loop forever. This problem can be avoided by using a memoing system such as XSB [144], or a bottom-up system such as Aditi [139]. However, it is common to re-write the program above so that the path found is returned as part of the answer. In such cases, systems such as XSB and Aditi will only work for graphs which are acyclic. For example, consider the program below.

```
path(X,Y,[X,Y]) :- edge(X,Y).
path(X,Y,[X|Path]) :- edge(X,Z), path(Z,Y,Path).
```

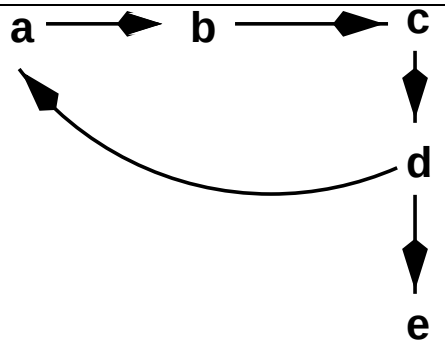
If there are cycles in the graph, then Prolog, XSB and Aditi will all generate an infinite number of paths, many of which will traverse the cycle in the graph more than once.

[4] Actually affine

The problem is that edges in the graph can be used an arbitrary number of times, and hence we cannot mark an edge as used, which is what is done in many imperative solutions to graph problems. However, in a linear logic programming language such as Lygon, we can easily constrain each edge to be used at most once on any path, and hence eliminate the problem with cycles causing an infinite number of paths to be found.

The code is simple; the main change to the above is to load a "linear" copy of the *edge* predicate, and use the code as above, but translated into Lygon. Most of this is mere transliteration, and is given in program 11.

Figure 5.2 Graph 1



The extra predicate *trip* is introduced so that not every path need use every edge in the graph. As written in program 11, *path* will only find paths which use every edge in the graph (and so *path* can be used directly to find Eulerian circuits, ie. circuits which use every edge in the graph exactly once). However, the *trip* predicate can ignore certain edges, provided that it does not visit any edge more than once, and so the *trip* predicate may be considered the affine form of the predicate *path*.

The goal *graph* (which describes figure 5.2) is used to load the linear copy of the graph, and as this is a non-linear rule, we can load as many copies of the graph as we like; the important feature is that within each graph no edge can be used twice. We can then find all paths, cyclic or otherwise, starting at node *a* in the graph with the goal *graph* $\otimes$ *trip(a,_,P)*. This goal yields the solutions below.

P = [a,b,c,d,e]

P = [a,b,c,d,a]

P = [a,b,c,d]

P = [a,b,c]

P = [a,b]

We can also find all cycles in the graph with a query such as *graph ⅋ trip(X,X,P)* which yields the solutions:

X = a, P = [a,b,c,d,a]

X = b, P = [b,c,d,a,b]

X = d, P = [d,a,b,c,d]

X = c, P = [c,d,a,b,c]

Program 11 Path Finding

graph ← **neg** edge(a,b) ⅋ **neg** edge(b,c) ⅋ **neg** edge(c,d) ⅋ **neg** edge(d,a) ⅋ **neg** edge(d,e).

Path: the naïve path predicate. Note that in Lygon this handles cycles.
path(A,B,[A,B]) ← edge(A,B).
path(A,B,[A|R]) ← edge(A,C) ⊗ path(C,B,R).

trip(X,Y,Z) ← ⊤ ⊗ path(X,Y,Z).

This example suggests that Lygon is an appropriate vehicle for finding "interesting" cycles, such as Hamiltonian cycles, ie., those visiting every node in the graph exactly once. We can write such a program in a "generate and test" manner by using the *path* predicate above, and writing a test to see if the cycle is Hamiltonian. The key point to note is that we can delete any edge from a Hamiltonian cycle and we are left with an acyclic path which includes every node in the graph exactly once. Assuming that the cycle is represented as a list, then the test routine will only need to check that the tail of the list of nodes in the cycle is a permutation of the list of nodes in the graph. Hodas and Miller [70] have shown that such permutation problems can be solved simply in linear logic programming languages by "asserting" each element of each list into an appropriately named predicate, such as *list1* and *list2*, and testing that *list1* and *list2* have exactly the same solutions.

The code to do this assertion is below; note that this may be thought of as a data conversion, from a list format into a predicate format.

perm([X|L],L2) $\leftarrow$ **neg** list1(X) $\parr$ perm(L,L2).
perm([], L2) $\leftarrow$ perm1(L2).

perm1([X|l]) $\leftarrow$ **neg** list2(X) $\parr$ perm1(L).
perm1([]) $\leftarrow$ check.

Once both lists are loaded, we call the predicate *check*, which will ensure that the two lists are permutations of each other. This predicate is particularly simple to write, and is given below. Note that if the input is given as predicates (i.e., *[1,2]* would be rendered as the multiset of linear facts *list1(1) , list1(2)*) rather than lists, then this is all the code that we would need to write:

check $\leftarrow$ list1(X) $\otimes$ list2(X) $\otimes$ check.
check.

Note that the second clause is equivalent to *check* $\leftarrow$ **1** and will only succeed if the linear context is empty. The definition of *check* is thus deterministic. The first clause will succeed provided that there is a common solution to *list1* and *list2*; if this is not the case, then the only way to succeed is for both predicates to be exhausted. Hence, the only way for the test to succeed is if the two arguments to *perm* contain the same multiset of elements – that is, they are permutations of each other.

An interesting point raised by this example is that the representation of the data as resources, rather than in the traditional list, is potentially more efficient. The most common data structure used in Prolog programs is a list, and it seems that Prolog programmers have a tendency to overuse lists. It is not uncommon to find that some general collection of objects has been implemented as a list, even when the order in which items are represented is not important. Lists have the property that access to the nth element takes n operations. However, if the same collection is represented as a multiset of formulae, then there is the possibility for more flexible access, as there is no order of access imposed by the framework. Furthermore, by careful utilisation of indexing techniques, it would seem that it is possible to provide constant or near-constant time access to an arbitrary element of the collection, and so Lygon has the potential for a more efficient representation of collections where the sequence of elements is not important. This observation

is borne out by the work in [137, 138] where a speedup factor of 25 is reported for small Lolli programs (and a larger speedup is reported for larger programs).

The only remaining task to complete the code for the Hamiltonian cycle program is to write the code to extract the list of nodes in the graph from its initial representation. As this code is not particularly insightful, we omit it for the sake of brevity and assume that the nodes of the graph are given. The full Lygon program for finding Hamiltonian cycles is given below (program 12).

The role of the $\top$ in *go* is to make the *edge* predicate affine (i.e., not every edge need be used). Note that the nodes remain linear and so *must* be used. Given the query *go(P)*, the program gives the solutions:

P = [c,d,a,b,c]

P = [d,a,b,c,d]

P = [b,c,d,a,b]

P = [a,b,c,d,a]

Program 12 Hamiltonian Cycles

```
go(P) ← graph ⅋ (⊤ ⊗ (nodes ⅋ hamilton(P))).

graph ← neg edge(a,b) ⅋ neg edge(b,c) ⅋ neg edge(c,d) ⅋ neg edge(d,a).
nodes ← neg node(a) ⅋ neg node(b) ⅋ neg node(c) ⅋ neg node(d).

path(X,Y,[X,Y]) ← edge(X,Y).
path(X,Y,[X|P]) ← edge(X,Z) ⊗ path(Z,Y,P).

all_nodes([]).
all_nodes([Node|Rest]) ← node(Node) ⊗ all_nodes(Rest).

hamilton(Path) ← path(X,X,Path) ⊗ eq(Path,[_|P]) ⊗ all_nodes(P).
```

A problem related to the Hamiltonian path is that of the travelling salesman. In the travelling salesman problem we are given a graph as before. However each edge now has an associated *cost*. The solution to the travelling salesman problem is the (or a) Hamiltonian cycle with the minimal total edge cost. Given a facility for finding aggregates,

such as `findall` or `bagof` in Prolog, which will enable all solutions to a given goal to be found, we can use the given program for finding Hamiltonian cycles as the basis for a solution to the travelling salesman problem. This would be done by simply finding a Hamiltonian cycle and computing its cost. This computation would be placed within a `findall`, which would have the effect of finding all the Hamiltonian cycles in the graph, as well as the associated cost of each. We would then simply select the minimum cost and return the associated cycle. Note that as this is an NP-complete problem, there is no better algorithm known than one which exhaustively searches through all possibilities.

In order to directly implement the solution described above, aggregate operators in Lygon are needed. As these are not yet present we do not give the code for this problem here.

Another useful algorithm operating on graphs is the topological sort, which is often applied to the resolution of dependency problems. One application of topological sorting is to find an ordering of a set of dependent actions such that each action is preceded by the actions on which it depends.

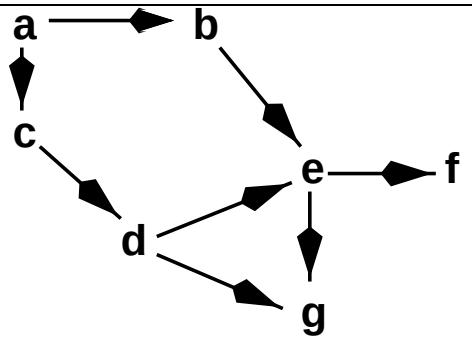
The input to the algorithm is a directed acyclic graph (DAG), and the output is an ordering of the nodes in the graph. The algorithm to topologically sort a DAG is as follows:

1. Select a random node,
2. Recursively topologically sort its descendants,
3. Add the node to the front of the output.

This repeats until all nodes have been processed. Note that any node with multiple parents will be processed more than once. Attempting to process a node for the second or subsequent time should succeed without doing anything. This behavior is given by *getnode* and *getlink*. If the node has already been processed then *getnode* succeeds returning the empty list. The call to *getlink* with an empty list returns an empty list of descendants. The net effect is that topologically sorting a node which has already been processed succeeds quietly. Note the use of the operator **once**; the goal **once***(G)* is operationally the same as the goal *G*, except that **once***(G)* will succeed at most once. In this context the **once** is used to simulate an *if-then-else* construct.

The predicate *sort* selects a random unprocessed node and topologically sorts its descendants (using *tsl*). This repeats until the graph has been completely processed. The predicate *tsl* topologically sorts each node in a list of nodes using *ts*. The code is given in program 13.

Figure 5.3 Graph 2



For a given graph there are many orderings which are valid topological orderings. For example using the graph in figure 5.3 we can apply a topological sort using the query *sort(X)* which returns the first solution *X = [g,f,e,d,c,b,a]*.

If we ask the system to find alternative solutions it comes up with the following distinct solutions:

X = [g,f,e,d,c,b,a]

X = [g,f,e,d,b,c,a]

X = [g,f,e,b,d,c,a]

X = [f,g,e,d,c,b,a]

X = [f,g,e,d,b,c,a]

X = [f,g,e,b,d,c,a]

Many graph search problems make use of either a *depth-first* search or a *breadth-first* search; given a set of alternative paths to follow, a depth-first search will generally explore children before siblings, whereas a breadth-first search will generally explore siblings before children (an analogy is that depth-first search is like an escapee from a jail who tries to get as far away from the jail as possible, whereas breadth-first search is like the police looking for the escapee).

Clearly for either of these processes it is important to detect cycles in the graph, in

Program 13 Topological Sorting

```
linear node(d,[d]).        linear link(a,[b,c]).
linear node(a,[a]).        linear link(b,[e]).
linear node(e,[e]).        linear link(c,[d]).
linear node(b,[b]).        linear link(d,[g,e]).
linear node(f,[f]).        linear link(e,[g,f]).
linear node(c,[c]).        linear link(f,[]).
linear node(g,[g]).        linear link(g,[]).

go ← sort(X) ⊗ output(X).

sort(R) ← node(_,[N]) ⊗ link(N,L) ⊗ tsl(L,R1) ⊗ sort(R2) ⊗ append(R1,[N|R2],R).
sort([]).

ts(Node,R) ← getnode(Node,N) ⊗ getlink(N,L) ⊗ tsl(L,R1) ⊗ append(R1,N,R).
tsl([],[]).
tsl([N|Ns],R) ← ts(N,R1) ⊗ tsl(Ns,Rs) ⊗ append(R1,Rs,R).

getlink([],[]).
getlink([N],L) ← link(N,L).

getnode(N,R) ← once(node(N,R) ⊕ eq(R,[])).
```

order to avoid infinite searches. As above, the ability to specify that each edge be used at most once ensures that the search will always terminate.

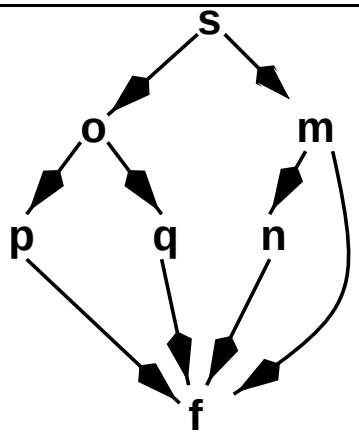
In program 14 the predicate *search* does the searching. Its first argument is a list of nodes to be explored, its second argument is a list of the nodes which have been examined and its third argument is the node being searched for. At each step we take the head node to be explored and (if it is not the desired node) add its descendants to the list of nodes to be explored using *expand*.

The difference between a breadth-first and depth-first (program 15) search is the order in which the new child-nodes are appended with the old list. Note that this program can be easily generalised to other sorting orders – for example, best first search.

As in program 13 we use *getnode* to handle nodes which are explored more than once. The code is given below. Given the query *find(s,f)* the breadth-first version explores the

nodes *[s,o,m,p,q,n,f]* and the depth-first version explores the nodes *[s,o,p,f]*. The graph represented by the collection of linear clauses is given in figure 5.4.

Figure 5.4 Graph 3



Program 14 Breadth First Search

find(S,E,P) ← search([S],P,E).
find(S,E) ← find(S,E,P) ⊗ **print**(P) ⊗ **nl**.

expand([Y|Ys],Nqueue) ← getnode(Y, L) ⊗ append(Ys, L, Nqueue).

The top in the following clause consumes unused nodes.
search([K|Xs], [K], K) ← ⊤.
search([X|Xs], [X|P], K) ← **not**(**eq**(K,X)) ⊗ expand([X|Xs], Nq) ⊗ search(Nq, P,K).

This allows nodes to be referenced multiple times.
getnode(N,R) ← **once**(node(N,R) ⊕ **eq**(R,[])).

linear node(s, [o,m]).
linear node(m, [n,f]).
linear node(n, [f]).
linear node(o, [p,q]).
linear node(p, [f]).
linear node(q, [f]).
linear node(f, []).

Program 15 Depth First Search

find(S,E,P) ← search([S],P,E).
find(S,E) ← find(S,E,P) ⊗ **print**(P) ⊗ **nl**.

expand([Y|Ys],Nqueue) ← getnode(Y, L) ⊗ append(L, Ys, Nqueue).

The top in the following clause consumes unused nodes.
search([K|Xs], [K], K) ← ⊤.
search([X|Xs], [X|P], K) ← **not**(**eq**(K,X)) ⊗ expand([X|Xs], Nq) ⊗ search(Nq, P,K).

This allows nodes to be referenced multiple times.
getnode(N,R) ← **once**(node(N,R) ⊕ **eq**(R,[])).

5.4 Concurrency

It is folklore that linear logic is good at expressing concurrency. Let us begin by distinguishing between *concurrency* and *parallelism* since (in this context) linear logic does not actually appear to offer any advantages to the latter. A *concurrent* program is one where the logic of the program involves multiple, independent threads of activity. Examples include any distributed system, an email system, multi-player games etc. Typically, the computation is affected by timing issues and is non-deterministic. A *parallel* program is one which is run on multiple processors with the aim of attaining better performance. Often, the result of the computation is deterministic.

A number of linear logic programming languages are targeted at concurrent applications (e.g. ACL [85, 88, 128] and LO [7–10, 29, 42]) and it has been shown that much of the π calculus can be mapped into linear logic [108].

In this section we show how various aspects of concurrency can be expressed in Lygon. Some aspects of concurrency that we shall look at are [17]:

1. Specifying multiple processes,
2. Communication, and
3. Synchronisation.

These are demonstrated in programs 16 and 17.

We also show how a range of other paradigms for concurrent programming can be simply and easily embedded in linear logic. Specifically, we embed the Chemical reaction metaphor for concurrent programming [22], the co-ordination language Linda [27, 28], the Actors paradigm [2] and Petri Nets [120]. We finish this section with a solution to the dining philosophers problem.

We begin with a simple example which illustrates how we can program two communicating processes in Lygon. Recall that a clause of the form $a \parr b \leftarrow G$ is applied to a goal of the form a , b , Γ to yield the new goal G, Γ.

Program 16 defines two communicating processes – a consumer and a producer. The first aspect of concurrent programming – specifying multiple processes – is simple in Lygon. A goal of the form $F \parr G$ specifies that F and G evolve concurrently.

Communications between the two processes is achieved by adding and removing goals. It is also possible to communicate using linearly negated atoms which have the advantage of being more obviously passive messages. The disadvantage is primarily a cluttering of the syntax.

The first clause of *produce* states that under appropriate conditions an *ack* message and a *produce(3)* process can evolve to *produce(2)* and a *mesg(1)*.

Note that unlike standard concurrent logic programming languages such as Strand [43], GHC [125, chapter 4], Parlog [50] and [125, chapter 3] and Concurrent Prolog [125, chapters 2 & 5] communication is orthogonal to unification. Unification (the underlying primitive operation of logic programming) is not tampered with.
The goal *go(3)* behaves as follows:

go(3)
$\longrightarrow$ produce(3) $\parr$ consume(0) $\parr$ ack
$\longrightarrow$ produce(3) , consume(0) , ack
$\longrightarrow$ **lt**(0,3) $\otimes$ **is**(N1,3-1) $\otimes$ (mesg(1) $\parr$ produce(N1)) , consume(0)
$\longrightarrow$ mesg(1) $\parr$ produce(2) , consume(0)
$\longrightarrow$ mesg(1) , produce(2) , consume(0)
$\longrightarrow$ **is**(N1,0+1) $\otimes$ (consume(N1) $\parr$ ack) , produce(2)
$\longrightarrow$ consume(1) $\parr$ ack , produce(2)
$\longrightarrow$ consume(1) , ack , produce(2)
$\vdots$
$\longrightarrow$ consume(3) , ack , produce(0)
$\longrightarrow$ consume(3) , finished
$\longrightarrow$ **print**(3) $\otimes$ **nl**.
$\vdots$

Program 16 Communicating Processes

go(N) $\leftarrow$ produce(N) $\parr$ consume(0) $\parr$ ack.

produce(N) $\parr$ ack $\leftarrow$ **lt**(0,N) $\otimes$ **is**(N1, N-1) $\otimes$ (mesg(1) $\parr$ produce(N1)).
produce(0) $\parr$ ack $\leftarrow$ finished.

consume(N) $\parr$ mesg(X) $\leftarrow$ **is**(N1,N+X) $\otimes$ (consume(N1) $\parr$ ack).
consume(N) $\parr$ finished $\leftarrow$ **print**(N) $\otimes$ **nl**.

An important aspect of concurrent programming is *mutual exclusion*. The basic idea is that we have a number of processes. A section of each process is designated as a *critical region*. We need to guarantee that at most one process is in its critical region at a given time. This can be done by using a lock – before entering its critical region a process attempts to acquire the lock. If the lock can be acquired then the process proceeds; otherwise it suspends and waits for the lock to become free.

In program 17 the lock is modelled by the goals *ex* and *noex*. Exactly one of these should be present at any time. The presence of *ex* means that the lock is free and the presence of *noex* that the lock has been acquired.

Processes access the lock using the predicates:

- *ask(C)* which attempts to acquire the lock and calls the continuation *C* if the attempt succeeds, suspending otherwise.

- *rel(C)* which releases the lock and calls the continuation *C*.

Note that the program uses the technique presented in program 10 to batch its output. The predicate *p1* acquires the lock, prints `'p1a'`, prints `'p1b'` and then releases the lock. Note that in order for the lock to actually be useful we need to take two separate printing actions. The predicate *p2* acquires the lock, prints `'p2a'` and then releases the lock.

There are three visible events that occur. Since `p1a` must occur before `p1b` there are three possible executions. The use of the lock prevents `p2a` from occurring between `p1a` and `p1b`. This eliminates one of the possible executions leaving `p1a, p1b, p2a` and `p2a, p1a, p1b` as possible results.

Note that because the Lygon implementation uses "don't know" non-determinism it is possible to backtrack and enumerate all possible executions of the concurrent program. This allows us to verify that the program is correct by automatically exploring every possible execution.

We have seen how Lygon can express the basic mechanisms of concurrency. In the remainder of this section we provide evidence that linear logic in general and Lygon specifically is expressive in this domain. We embed a number of paradigms for concurrency in Lygon. As we shall see, in all cases the embedding is direct and straightforward.

Program 17 Mutual Exclusion

```
ex ⅋ ask(C) ← noex ⅋ call(C).
noex ⅋ rel(C) ← ex ⅋ call(C).

go(X) ← ex ⅋ p1 ⅋ p2 ⅋ neg t([]) ⅋ (neg ex ⊗ t(X)).

p1 ← ask(t(X) ⊗ (neg t(p1a.X) ⅋ (t(X) ⊗ (neg t(p1b.X) ⅋ rel(⊥))))).
p2 ← ask(t(X) ⊗ (neg t(p2a.X) ⅋ rel(⊥))).
```

The chemical reaction paradigm [22] views a concurrent program as a solution of reacting molecules. Chemical reactions between molecules are local. This paradigm forms the basis of the language Gamma [18].

Program 18 gives an example of a simple reaction. Note that since Lygon (like most logic programming languages) is executed using backwards chaining on clauses the implications are reversed. That is, a clause of the form $a \leftarrow b$ indicates that a can be rewritten to b. In the program $o \parr o \parr h2 \parr h2$ is rewritten first to $o2 \parr h2 \parr h2$ and then to $h2o \parr h2o$.

Program 18 Chemical Paradigm

```
o ⅋ o ← o2.
o2 ⅋ h2 ⅋ h2 ← h2o ⅋ h2o.

go ← o ⅋ o ⅋ h2 ⅋ h2 ⅋ (neg h2o ⊗ neg h2o).
```

Another model for concurrency is the co-ordination language Linda [27, 28]. Linda provides four primitive operations which can be added to any language to yield a concurrent programming language. Versions of Linda have been built on top of C, Scheme, Modula-2, Prolog and other languages. Linda's basic abstraction is of a shared distributed *tuple space*. The primitive operations provided are *add*, *read* and *remove* a tuple from the tuple space. The fourth primitive operation (*eval*) creates a new process.

These operations can easily be specified in Lygon; witness program 19. Note that this is a generalisation of the state ADT given in program 8.

Program 19 Linda

in(X,G) ← tup(X) ⊗ **call**(G). *Remove a tuple*
out(X,G) ← **neg** tup(X) ⅋ **call**(G). *Add a tuple*
read(X,G) ← tup(X) ⊗ (**neg** tup(X) ⅋ **call**(G)). *Non-destructively read a tuple*
eval(X,G) ← **call**(X) ⅋ **call**(G).

The Actor model [2] is an abstraction of concurrent processes. An actor is activated when it receives a message. It can respond by

- Sending more messages,
- Creating new actors, and
- Changing its local state

If we encode an actor as *actor(Id,State)* and a message as *mesg(Id,Args)* then a rule describing how an actor responds to a message can be written as

actor(Id,State) ⅋ mesg(Id,Args) ←
  actor(NewId,S) ⅋ . . . ⅋ *Create new actors*
  mesg(NewId,A) ⅋ . . . ⅋ *Send messages*
  actor(Id,NewState). *Update local state*

The example below defines a bank account actor. This actor can respond to three types of messages:

1. A balance query
2. A request to withdraw funds
3. A deposit request

We also add a *shutdown* request which terminates the actor.

Note that the actor model has no natural notion of sequentiality – message receipt is not guaranteed to be ordered. This fits in with the execution semantics of the Lygon realisation of the model. The goal *go* creates an actor with a balance of 0 and sends it a request to withdraw $20, a request to deposit $30 and a balance query. The goal has six solutions which correspond to six different message receipt orders. If the deposit request

is received after the withdraw request then the withdraw request fails and the final balance is $30, otherwise the final balance is $10. The balance query can occur:

1. Before either action (result 0),
2. After both actions (result 10 or 30 depending on the ordering of the other two actions)
3. Between the actions (result 30 or 0 depending on the ordering of the other two actions)

The goal *go* produces the following output:

```
Balance query: 0
Withdraw successful: yes
Final Balance: 10

Balance query: 0
Withdraw successful: no
Final Balance: 30

Balance query: 30
Withdraw successful: yes
Final Balance: 10

Balance query: 10
Withdraw successful: yes
Final Balance: 10

Balance query: 0
Withdraw successful: no
Final Balance: 30

Balance query: 30
Withdraw successful: no
Final Balance: 30
```

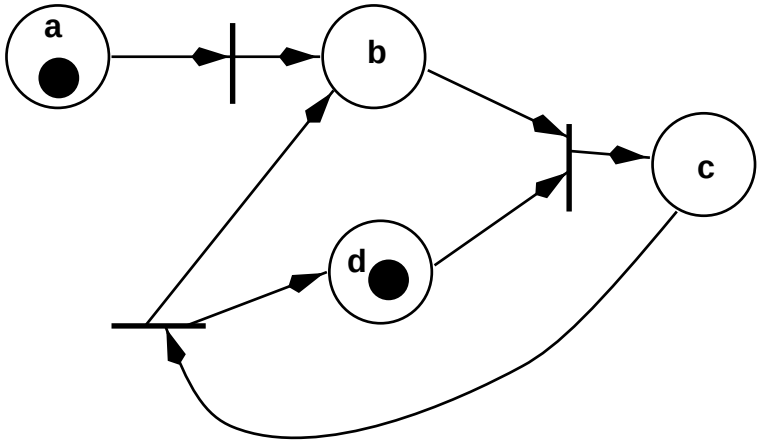
Our final example of embedding a concurrent model in Lygon is Petri nets. A Petri net [120] consists of labelled places (the bigger circles), transitions and tokens (the filled black circles).

Program 20 Actors

actor(Id,Val) ⅋ mesg(Id,balance(Val)) ← actor(Id,Val).
actor(Id,Val) ⅋ mesg(Id,deposit(X)) ← **is**(NV, Val + X) ⊗ actor(Id,NV).
actor(Id, Val) ⅋ mesg(Id,withdraw(X,yes)) ← **lt**(X,Val) ⊗ **is**(NV,Val-X) ⊗ (actor(Id,NV)).
actor(Id, Val) ⅋ mesg(Id,withdraw(X,no)) ← le(X,Val) ⊗ actor(Id,Val).
To shut down ...
actor(Id, Val) ⅋ mesg(Id,terminate(Val)) ← **1**.

Test ...
go ← (actor(ac1,0) ⅋ mesg(ac1,withdraw(20,R)) ⅋ mesg(ac1,deposit(30)) ⅋
mesg(ac1,balance(B)) ⅋ mesg(ac1,terminate(X)))
⊗ **print**('`Balance query: `') ⊗ **print**(B) ⊗ **nl**
⊗ **print**('`Withdraw successful: `') ⊗ **print**(R) ⊗ **nl**
⊗ **print**('`Final Balance: `') ⊗ **print**(X) ⊗ **nl** ⊗ **nl**
⊗ fail.

Figure 5.5 A Petri Net



A transition is *enabled* if all places with incoming arcs have at least one token. An enabled transition *fires* by removing a token from each place with an incoming arc and placing a token on each place with an outgoing arc. In the Petri net above, the only enabled transition is from *a* to *b* and it fires by removing the token from *a* and placing a token on *b*.

Davison [36] investigates modelling Petri nets in the guarded Horn clause language Parlog ([50] and [125, chapter 3]). The Lygon realisation below is simpler and more concise. In Lygon (program 21) the Petri net in figure 5.5 is encoded as:

t(a) $\leftarrow$ t(b).
t(b) $\otimes$ t(d) $\leftarrow$ t(c).
t(c) $\leftarrow$ t(b) $\otimes$ t(d).

This states that (i) a token at place *a* can be transformed to a token at *b*, (ii) if there are tokens at both *b* and *c* then they can be replaced with a token at *d*, and (iii) a token at *d* can be replaced with two tokens at *b* and *c*.

Program 21 uses *tick* to single step the net – a transition cannot fire unless it is given a *tick*. The *control* predicate is quite interesting and demonstrates an application of context copying using $\&$. The predicate takes a single argument – a list, the length of which determines how many steps are to be taken. The first clause of *control* terminates the program (using $\top$) if all transitions have been carried out. The second clause copies the computation; the first copy is collected and printed, the second is given a *tick* token which allows a single transition to take place. This technique allows a concurrent computation to be controlled and terminated.

The query *go* generates the following output:

```
[t(a),t(d)]
[t(b),t(d)]
[t(c)]
[t(b),t(d)]
[t(c)]
[t(b),t(d)]
```

Program 21 Petri Nets

```
tick ⅋ t(a) ← t(b).
tick ⅋ t(b) ⅋ t(d) ← t(c).
tick ⅋ t(c) ← t(b) ⅋ t(d).

go ← control([1,2,3,4,5,6]) ⅋ t(a) ⅋ t(c).

control([]) ← ⊤.
control([A|As]) ← (collect(X) ⊗ output(X)) & (tick ⅋ control(As)).

collect([t(X)|R]) ⅋ t(X) ← collect(R).
collect([]) ← 1.
```

We finish this section with a solution to the classical dining philosophers problem. In this problem there are a number (usually five) of philosophers whose actions are as follows:

1. Get a room ticket and then enter the dining room
2. Get the chopstick on one side
3. Get the chopstick on the other side
4. Eat
5. Return the chopsticks and room ticket
6. Return to thinking then repeat from step 1

A solution to the dining philosophers problem is a (concurrent) program which simulates the actions of the hungry thinkers and implements a strategy which prevents (literal!) starvation. Starvation can occur, for example, if all the philosophers simultaneously pick up their right chopsticks and then wait for the left chopstick to become available.

This particular solution is adapted from [28]. For N philosophers there are $N - 1$ "room tickets". Before entering the room each philosopher must take a roomticket from

a shelf beside the door. This prevents all of the philosophers from being in the room at the same time.

Program 22 combines a number of idioms which we have seen. The first clause of *phil* uses a continuation passing style to sequence operations. The operations being sequenced involve output (using **print**). A lock is used to obtain mutual exclusion, this prevents output from different philosophers from being interleaved.

Program 22 uses a number of linear predicates: *rm* represents a roomticket, *phil(X)* represents the Xth philosopher and *ch(X)* the Xth chopstick. *tok* is a token which enables a single execution step to take place. By supplying a certain number of tokens we prevent the computation from running indefinitely. *lpr* is a lock which is used to guarantee that primitive actions (such as picking up a chopstick or eating) are mutually exclusive.

The main clause defining *phil* is simply the continuation passing encoding of the sequence:

1. Delete a token
2. Write “I’m hacking”
3. Delete a room ticket
4. Write “Entering room”
5. Delete one chopstick
6. Write “Grabbing chopstick”
7. Delete other chopstick
8. Write “Grabbing chopstick”
9. Write “Eating”
10. Write “Returning”
11. Add room ticket and both chopsticks
12. Goto step 1

It is recommended that this program be run with **fair**ness (see [146]) turned on. A typical run produces the following output:

```
phil(a) hacking
phil(a) entering room
phil(c) hacking
phil(b) hacking
phil(d) hacking
phil(a) grabbing a
phil(c) entering room
phil(a) grabbing b
phil(b) entering room
phil(a) eat
phil(a) returning a and b
phil(c) grabbing c
phil(d) entering room
phil(c) grabbing d
phil(b) grabbing b
phil(c) eat
phil(c) returning c and d
phil(b) grabbing c
phil(b) eat
phil(d) grabbing d
phil(d) grabbing e
phil(b) returning b and c
phil(d) eat
phil(d) returning d and e
no tokens
```

Program 22 Dining Philosophers

go ←
phil(a) ⅋ **neg** chop(a) ⅋ **neg** room ⅋
phil(b) ⅋ **neg** chop(b) ⅋ **neg** room ⅋
phil(c) ⅋ **neg** chop(c) ⅋ **neg** room ⅋
phil(d) ⅋ **neg** chop(d) ⅋ **neg** room ⅋
phil(e) ⅋ **neg** chop(e) ⅋
tokens. *Added for termination purposes*

tokens ← **neg** tok ⅋ **neg** tok ⅋ **neg** tok ⅋ **neg** tok ⅋ **neg** lpr.

phil(N) ← tok ⊗ hack(N, (room ⊗ enter(N,
(succmod(N,N1) ⊗ chop(N) ⊗ grab(N,N, (chop(N1) ⊗ grab(N,N1, (
eat(N, (return(N,N1, (**neg** chop(N) ⅋ **neg** chop(N1) ⅋ **neg** room ⅋
phil(N)))))))))))).
phil(N) ← !(**print**('`no tokens`') ⊗ **nl**) ⊗ ⊤.

succmod(a,b). succmod(b,c).
succmod(c,d). succmod(d,e).
succmod(e,a).

enter(N,C) ← lpr ⊗ **print**('`phil(`') ⊗ **print**(N) ⊗ **print**('`) entering room`') ⊗ **nl** ⊗
(**neg** lpr ⅋ **call**(C)).

eat(N,C) ← lpr ⊗ **print**('`phil(`') ⊗ **print**(N) ⊗ **print**('`) eat`') ⊗ **nl** ⊗
(**neg** lpr ⅋ **call**(C)).

hack(N,C) ← lpr ⊗ **print**('`phil(`') ⊗ **print**(N) ⊗ **print**('`) hacking`') ⊗ **nl** ⊗
(**neg** lpr ⅋ **call**(C)).

grab(N,N1,C) ← lpr ⊗ **print**('`phil(`') ⊗ **print**(N) ⊗ **print**('`) grabbing `') ⊗
print(N1) ⊗ **nl** ⊗ (**neg** lpr ⅋ **call**(C)).

return(N,N1,C) ← lpr ⊗ **print**('`phil(`') ⊗ **print**(N) ⊗ **print**('`) returning `') ⊗
print(N) ⊗ **print**(' `and` ') ⊗ **print**(N1) ⊗ **nl** ⊗ (**neg** lpr ⅋ **call**(C)).

5.5 Artificial Intelligence

In this section we are concerned with the *knowledge representation* aspects of Artificial Intelligence (AI). One of the many knowledge representation formalisms that has been commonly used in AI is classical logic. Classical logic suffers from a number of inadequacies – chief among them is that *change* cannot be easily modelled. One aspect of this is the *frame problem* – how to specify what remains *unchanged* by an operation. As we shall see linear logic does not suffer from this problem – state change in linear logic is simple and direct.

We present a range of examples illustrating the application of linear logic and of Lygon to knowledge representation. It is worth noting that the problems we consider have solutions in the classical logic framework. It is also worth emphasising that these solutions are invariably complex and obtuse.

In [5] Vladimir Alexiev looks at Kowalski's *Event Calculus* [90]. The event calculus is a formalisation of events and their effect on states. One of the key features of the Event Calculus is that it is executable – the theory is realised as a Prolog program. The Prolog realisation of the Event Calculus relies heavily on negation as failure. This is inelegant in that negation as failure is global – it deals with non-derivability. On the other hand the notion of change is intuitively a local one. Linear logic allows a realisation of the Event Calculus which is pure, direct and implements change as a local operation. In addition to being simpler, being able to view change as a local operation is also considerably more efficient.

In [145] a more detailed analysis is performed and a Lygon meta-interpreter constructed. The same conclusion – that linear logic is a much more natural (and efficient!) framework for representing change – is reached.

We look at a number of examples:

1. The *Yale shooting problem*: a prototypical example of the frame problem.

2. Blocks world: an example planning problem.

3. Default reasoning.

The Yale shooting problem [52] is a prototypical example of a problem involving actions. The main technical challenge in the Yale shooting problem is to model the appropriate changes of state, subject to certain constraints. In particular:

1. Loading a gun changes its state from unloaded to loaded;

2. Shooting a gun changes its state from loaded to unloaded;

3. Shooting a loaded gun at a turkey changes the turkey's state from alive to dead.

To model this in Lygon, we have predicates *alive*, *dead*, *loaded*, and *unloaded*, representing the given states, and predicates *load* and *shoot*, which, when executed, change the appropriate states. The initial state is to assert *alive* and *unloaded*, as initially the turkey is alive and the gun unloaded. The actions of loading and shooting are governed by the following rules:

load $\leftarrow$ unloaded $\otimes$ **neg** loaded.
shoot $\leftarrow$ alive $\otimes$ loaded $\otimes$ (**neg** dead $\parr$ **neg** unloaded).

Hence given the initial resources *alive* and *unloaded*, the goal *shoot* $\parr$ *load* will cause the state to change first to *alive* and *loaded*, as *shoot* cannot proceed unless *loaded* is true, and then *shoot* changes the state to *dead* and *unloaded*, as required.

Expressing this problem in Lygon is simplicity itself (see program 23). It would be trivial given certain syntactic sugar - see program 28. The query *go* prints out `[unloaded,dead]`.

An alternative way of presenting the first clause is *shoot* $\parr$ *(dead* $\otimes$ *unloaded)* $\leftarrow$ *alive* $\parr$ *loaded*. This is slightly clearer but isn't valid Lygon.

A (slightly) less artificial planning problem is the blocks world. The blocks world consists of a number of blocks sitting either on a table or on other blocks and a robotic arm capable of picking up and moving a single block at a time. We seek to model the state of the world and of operations on it. Our presentation is based on [101–103]. This program was independently written by Alessio Guglielmi [51].

The predicates used to model the world in program 24 are the following:

- *empty:* the robotic arm is empty;

Program 23 Yale Shooting Problem

```
shoot ← alive ⊗ loaded ⊗ (neg dead ⅋ neg unloaded).
load ← unloaded ⊗ neg loaded.
start ← neg alive ⅋ neg unloaded.

go(X) ← collect([],X) ⅋ shoot ⅋ start ⅋ load.
go collects all solutions and displays them.
go ← go(X) ⊗ output(X) ⊗ fail.

collect(X,Y) ← get(Z) ⊗ collect([Z|X],Y).
collect(X,X).

get(X) ← once( (alive ⊗ eq(X,alive)) ⊕
          (dead ⊗ eq(X,dead)) ⊕
          (loaded ⊗ eq(X,loaded)) ⊕
          (unloaded ⊗ eq(X,unloaded))).
```

- *hold(A):* the robotic arm is holding block A;

- *clear(A):* block A does not support another block;

- *ontable(A):* block A is supported by the table;

- *on(A,B):* block A is supported by block B.

There are a number of operations that change the state of the world. We can *take* a block. This transfers a block that does not support another block into the robotic arm. It requires that the arm is empty. We can *remove* a block from the block beneath it, which must be done before picking up the bottom block. We can also *put* a block down on the table or *stack* it on another block. Finally, *initial* describes the initial state of the blocks.

The predicate *showall* allows us to collect the state of the blocks into a list which can be displayed. The goal *initial* ⅋ *go* ⅋ *show* returns the solution *R = [empty, on(a,b), clear(a), clear(c), ontable(c), ontable(b)]*.

The order of the instructions *take*, *put* etc. is not significant: there are actions, specified by the rules, such as *put(c)*, which cannot take place from the initial state, and others,

such as *take(b)* which can. It is the problem of the implementation to find an appropriate order in which to execute the instructions, so giving the final state.

Program 24 Blocks World

```
take(X) ← (empty ⊗ clear(X) ⊗ ontable(X)) ⊗ neg hold(X).
remove(X,Y) ← (empty ⊗ clear(X) ⊗ on(X,Y)) ⊗ (neg hold(X) ⅋ neg clear(Y)).
put(X) ← hold(X) ⊗ (neg empty ⅋ neg clear(X) ⅋ neg ontable(X)).
stack(X,Y) ← (hold(X) ⊗ clear(Y)) ⊗ (neg empty ⅋ neg clear(X) ⅋ neg on(X,Y)).

The initial state of the world ...
initial ← neg ontable(a) ⅋ neg ontable(b) ⅋ neg on(c,a) ⅋ neg clear(b) ⅋ neg clear(c)
      ⅋ neg empty.

The actions to be done ...
go ← remove(c,a) ⅋ put(c) ⅋ take(a) ⅋ stack(a,b).

showall([ontable(X)|R]) ← ontable(X) ⊗ showall(R).
showall([clear(X)|R]) ← clear(X) ⊗ showall(R).
showall([on(X,Y)|R]) ← on(X,Y) ⊗ showall(R).
showall([hold(X)|R]) ← hold(X) ⊗ showall(R).
showall([empty|R]) ← empty ⊗ showall(R).
showall([]).

show ← showall(R) ⊗ output(R). output is defined in program 1.
```

Our final artificial intelligence related example involves non-monotonic reasoning. A logic is non-monotonic if the addition of a fact can cause old facts to become false. Classical logic is monotonic and as a result, attempts at capturing non-monotonic reasoning in classical logic have been complicated. Girard [48] argues that linear logic is a good candidate for capturing non-monotonic reasoning.

A common application for non-monotonic reasoning is reasoning about exceptional situations and taxonomies. For example, as a rule, birds fly and chirp. We also know that penguins are birds but that they do not fly. We seek to find a representation that allows penguins to inherit the chirping behavior from birds while preventing them from inheriting the ability to fly.

In order to use linear logic as the reasoning mechanism we need to represent knowledge as linear predicates. So we have the linear facts *opus*, *tweety*, *bird*, *penguin*, *fly* and *chirp*. Our rules are of the form

$$\textit{class} \leftarrow \textit{properties}$$

For example, *bird* $\leftarrow$ *fly* $\parr$ *chirp*. Using resolution we can derive that a *bird* can *fly* and *chirp* (using the query *(***neg** *fly* $\otimes$ **neg** *chirp)* $\parr$ *bird*).

Since we wish to be able to derive that birds can fly (that is, ignore their ability to chirp where it is not relevant) we allow the derived attributes to be weakened by writing *bird* $\leftarrow$ *(fly* $\oplus\bot$*)* $\parr$ *(chirp* $\oplus\bot$*)*.

We can easily state that Opus has the property of penguin-hood (*opus* $\leftarrow$ *penguin* – note that since Opus only has a single property weakening is not necessary). The interesting part is encoding the knowledge that penguins are non-flying birds. This is where we exploit the non-monotonic properties of linear logic – *bird* $\parr$ **neg** *fly* is provable but, in general, *bird* $\parr$ **neg** *fly* $\parr$ *p* will not be. Thus we write *penguin* $\leftarrow$ *bird* $\parr$ *antifly*. The choice of the formula *antifly* is important. We want flying behavior to be suppressed in penguins but not chirping behavior. In order to achieve this *antifly* must be fly specific. By having *antifly* consume an instance of *fly* the correct behavior occurs.

$$\textit{antifly} \leftarrow \ \textbf{neg}\ \textit{fly} \otimes \bot$$

Girard refers to *antifly* as a "kamikaze" [48].

Below we have a derivation of Opus' chirping behavior (on the left) and a failed attempt to prove that he can fly (on the right).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{fly, \textbf{neg}\ fly} \quad \dfrac{\dfrac{}{chirp, \textbf{neg}\ chirp}}{chirp \oplus \bot, \textbf{neg}\ chirp}}{fly \oplus \bot, \textbf{neg}\ fly \quad \bot, chirp \oplus \bot, \textbf{neg}\ chirp}}{fly \oplus \bot, chirp \oplus \bot, \textbf{neg}\ fly \otimes \bot, \textbf{neg}\ chirp}\otimes}{(fly \oplus \bot) \parr (chirp \oplus \bot), \textbf{neg}\ fly \otimes \bot, \textbf{neg}\ chirp}}{bird, \textbf{neg}\ fly \otimes \bot, \textbf{neg}\ chirp}}{bird \parr (\textbf{neg}\ fly \otimes \bot), \textbf{neg}\ chirp}}{penguin, \textbf{neg}\ chirp}}{opus, \textbf{neg}\ chirp}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{fly, \textbf{neg}\ fly} \quad \dfrac{\dfrac{}{chirp, \textbf{neg}\ fly}}{chirp \oplus \bot, \textbf{neg}\ fly}}{fly \oplus \bot, \textbf{neg}\ fly \quad \bot, chirp \oplus \bot, \textbf{neg}\ fly}}{fly \oplus \bot, chirp \oplus \bot, \textbf{neg}\ fly \otimes \bot, \textbf{neg}\ fly}\otimes}{(fly \oplus \bot) \parr (chirp \oplus \bot), \textbf{neg}\ fly \otimes \bot, \textbf{neg}\ fly}}{bird, \textbf{neg}\ fly \otimes \bot, \textbf{neg}\ fly}}{bird \parr (\textbf{neg}\ fly \otimes \bot), \textbf{neg}\ fly}}{penguin, \textbf{neg}\ fly}}{opus, \textbf{neg}\ fly}$$

Under program 25 we have that the goals *implies(tweety,chirp)*, *implies(tweety,fly)* and *implies(opus,chirp)* are derivable but that *implies(opus,fly)* is not.

Program 25 Exceptional Reasoning

Penguins are birds which don't fly ...
penguin $\leftarrow$ bird $\parr$ (**neg** fly $\otimes \bot$).
Birds fly and chirp ...
bird $\leftarrow$ (fly $\oplus \bot$) $\parr$ (chirp $\oplus \bot$).
Tweety is a bird
tweety $\leftarrow$ bird.
Opus is a penguin
opus $\leftarrow$ penguin.

implies(A,B) $\leftarrow$ **call**(**neg** B) $\parr$ **call**(A).

5.6 Meta-Programming

A *meta-interpreter* is an interpreter for a language L which is itself written in L. For example, a LISP interpreter written in LISP. The usefulness of meta-interpreters as distinct from any other type of interpreter is that in a dynamic language such as Prolog or LISP it is generally possible to delegate aspects of the implementation. For example, the standard Prolog meta-interpreter does not implement unification. This enables aspects of the implementation that are not interesting to be implemented with a minimum of effort.

Meta interpreters have a range of uses. One important area of application is prototyping – the use of meta-interpreters often allows for a quick implementation of a language which is similar to the base language. This allows experimentation with a language implementation while its design is still being developed. A number of languages began their lives as meta-interpreters including Concurrent Prolog [125, chapter 2], Erlang and of course, Lygon. A second area of application involves *non standard executions*. Typical examples are collecting statistics, tracing and debugging. Meta-interpreters are also used in certain areas of Artificial Intelligence.

Meta-interpreters are a part of the Prolog culture [117, 134]. The standard "vanilla" Prolog meta-interpreter handles the resolution mechanism and expresses the essence of Prolog execution in three lines:

```
prove(true) :- true.
prove((A,B)) :- prove(A), prove(B).
prove(A) :- clause(A,B), prove(B).
```

The structure of this program is typical of a meta-interpreter for a logic programming language. To prove a logical constant we simply check that the constant holds. In classical logic *false* will always fail and *true* will always succeed. However, in linear logic some logical constants will either succeed or fail depending on the context. The Prolog clause

```
prove(true) :- true.
```

corresponds to the Lygon clauses:

prove($\bot$) $\leftarrow$ $\bot$.
prove(**1**) $\leftarrow$ **1**.

prove($\top$) $\leftarrow$ $\top$.

To prove a formula which consists of a connective with some sub-formulae we prove the sub-formulae and join the proofs with the appropriate connectives. The Prolog clause

```
prove((A,B)) :- prove(A), prove(B).
```

corresponds to the Lygon clauses:

prove(A $\otimes$ B) $\leftarrow$ prove(A) $\otimes$ prove(B).
prove(A $\&$ B) $\leftarrow$ prove(A) $\&$ prove(B).
prove(A $\parr$ B) $\leftarrow$ prove(A) $\parr$ prove(B).
prove(A $\oplus$ B) $\leftarrow$ prove(A) $\oplus$ prove(B).
prove(exists(X,A)) $\leftarrow$ exists(X,prove(A)).
prove(! A) $\leftarrow$! prove(A).

The final Prolog clause handles atomic goals by resolving against a program clause. The Lygon equivalent is a little more complex since Lygon program clauses have a richer structure (see program 26).

There are a number of aspects of a Lygon meta-interpreter which do not arise in Prolog meta-interpreters:

1. The representation of the context,
2. Resolving against clauses, and
3. Selecting the formula to be reduced.

Note that the last of these is specific to Lygon and does not appear in the Lolli meta-interpreter [30].

We choose to represent a linear formula F as *lin(F)* and a non-linear formula $?F$ as *? nonlin(F)*. This allows us to distinguish between linear and non-linear formula and means that, for instance, the connective ! works as is, since the representation of linear and non-linear formulae are linear and non-linear formulae respectively.

Resolution is slightly more complex than is the case for Prolog but the principle is still straightforward. An atomic goal (*atom(A)*) is provable if either (i) the context contains

the linear negation (*lin(***neg** *A)*), (ii) *A* is a builtin predicate, or (iii) the program contains a clause *R* and resolving eliminates *A*.

The first and second cases are both handled easily by a single clause each. The third case uses the auxiliary predicate *doleft* which implements the relevant left rules.

The following derivation illustrates the use of *doleft*:

prove(atom(append([1],[2],An))) $\otimes$ **print**(An)
$\longrightarrow$ nonlin(R) $\otimes$ doleft(R,atom(append([1],[2],An))) $\otimes$ **print**(An)
R = (atom(append([X|Y],Z,[X|Q])) $\leftarrow$ atom(append(Y,Z,Q)))
$\longrightarrow$ doleft((atom(append([X|Y],Z,[X|Q])) $\leftarrow$ atom(append(Y,Z,Q))) , atom(append([1],[2],An)))
$\otimes$ **print**(An)
X = 1 , Y = [] , Z = [2] , An = [1|Q]
$\longrightarrow$ prove(atom(append([],[2],Q))) $\otimes$ **print**([1|Q])
$\longrightarrow$ nonlin(R) $\otimes$ doleft(R,atom(append([],[2],Q)))$\otimes$ **print**([1|Q])
R = atom(append([],X,X))
$\longrightarrow$ doleft(atom(append([],X,X)) , atom(append([],[2],Q))) $\otimes$ **print**([1|Q])
X = [2] , Q = [2]
$\longrightarrow$ **print**([1,2])

One operation that is specific to Lygon is selecting a formula to be reduced. As we have seen there are a number of heuristics which can be applied to reduce the amount of nondeterminism. Our first meta-interpreter (program 26) side-steps the issue by delegating formula selection to the underlying Lygon implementation. For example, the goal *prove((A* $\otimes$ *B)* $\parr$ *(C* $\&$ *D))* is reduced to *prove(A)* $\otimes$ *prove(B)* , *prove(C)* $\&$ *prove(D)* at which point the Lygon system will select the second formula for reduction and commit to this selection.

The second meta-interpreter (program 27) illustrates how selecting the formula to be reduced can be done by the meta-interpreter. We *select* a formula by looking at what classes of formulae are present. If there are any "async" formulae then we can select an arbitrary "async" formula and commit to it. If all formulae are "sync" then we must select a formula non-deterministically. Note that most of the clauses defining binary connectives are modified to use *select*. By writing *prove(A* $\parr$ *B)* $\leftarrow$ **neg** *lin(A)* $\parr$ **neg** *lin(B)* $\parr$ *select* rather than *prove(A* $\parr$ *B)* $\leftarrow$ *prove(A)* $\parr$ *prove(B)* we ensure that sub-formulae can only run when they have been selected.

The *select* operation copies the linear context (using $\&$). The copy is consumed and

a note is made of which classes of formulae are present. The predicate *get* then selects a formula from the appropriate class. This formula is then passed to *prove*.

For example, the goal[5] *prove((p ⊕ q) ⅋ (***neg** *p & **neg** q))* reduces to the goal **neg** *lin(p ⊕ q)* , **neg** *lin(***neg** *p & **neg** q)* , *select*. The linear context is copied and *type(sync,T)* is called. This goal consumes the two *lin* facts and determines that the first is synchronous and the second asynchronous. The goal returns the result *T = async* since asynchronous goals take precedence over synchronous goals.

The predicate *get(async,A)* is then called. This selects an arbitrary asynchronous goal formula and commits to the selection. The result (*A =* **neg** *p & **neg** q*) is then passed to *prove*.

One of the applications of meta-interpreters is to enable easy experimentation with language variants and extensions. One language extension to Lygon which has been proposed [154] is *rules*. A rule of the form *rule(N,Is ⇒ Os)* is read as stating that in order to prove *N* we must consume *Is* and produce *Os*. Alternatively, the rule rewrites the multiset containing *N* and *Is* to the multiset containing *Os*. The atom *N* is distinguished in that it triggers the rule application. As program 28 demonstrates, extending the Lygon meta-interpreter with rules is quite simple.

We have the following derivation:

init , shoot , load
⟶ unloaded, alive, shoot, load *Using the rule for init*
⟶ loaded, alive, shoot *Using the rule for load*
⟶ unloaded, dead *Using the rule for shoot*

[5] Note that the notation here omits occurrences of *atom(. . .)* to increase readability.

Program 26 Lygon Meta Interpreter I

```
prove(A ⊗ B) ← prove(A) ⊗ prove(B).
prove(A ⅋ B) ← prove(A) ⅋ prove(B).
prove(A ⊕ B) ← prove(A) ⊕ prove(B).
prove(A & B) ← prove(A) & prove(B).
prove(⊥) ← ⊥.
prove(1) ← 1.
prove(⊤) ← ⊤.
prove(exists(X,B)) ← exists(X,prove(B)).
prove(?(neg(F))) ← ? neg nonlin(F).
prove(!(F)) ← ! prove(F).
prove(once(F)) ← once(prove(F)).
prove(atom(A)) ← nonlin(R) ⊗ doleft(R,A). Resolve against a clause.
prove(atom(A)) ← lin(neg(A)).
prove(atom(A)) ← builtin(A) ⊗ call(A).
prove(neg(A)) ← neg lin(neg(A)).

doleft(atom(A),A).
doleft(forall(X,B),A) ← exists(X,doleft(B,A)).
doleft((atom(A) ← G),A) ← prove(G).
A simple Read-Eval-Print loop. Use EOF to exit.
shell ← repeat ⊗ print('Meta1 ') ⊗ readgoal(X) ⊗ print(X) ⊗
      once(
        (eq(X,atom(end_of_file)) ⊗ nl)
       ⊕ (prove(X) ⊗ output(yes))
       ⊕ output(no))
      ⊗ eq(X,atom(end_of_file)).

The program clauses usable from the meta-shell:
(Note that for now we need to manually add the atom wrapper)

nonlin(atom(eq(X,X))).
nonlin(atom(append([],X,X))).
nonlin((atom(append([X|Y],Z,[X|Q])) ← atom(append(Y,Z,Q)))).
```

Program 27 Lygon Meta Interpreter II

```
prove(A ⊗ B) ← prove(A) ⊗ prove(B).
prove(A ⅋ B) ← neg lin(A) ⅋ neg lin(B) ⅋ select.
prove(A ⊕ B) ← (neg lin(A) ⊕ neg lin(B)) ⅋ select.
prove(A & B) ← (neg lin(A) & neg lin(B)) ⅋ select.
prove(⊥) ← ⊥.    prove(1) ← 1.    prove(⊤) ← ⊤.
prove(exists(X,B)) ← exists(X,neg lin(B)) ⅋ select.
prove(?(neg(F))) ← ? neg nonlin(F).
prove(!(F)) ← ! prove(F).
prove(once(F)) ← once(prove(F)).
prove(atom(A)) ← nonlin(R) ⊗ doleft(R,A).
prove(atom(A)) ← lin(neg(A)).
prove(atom(A)) ← builtin(A) ⊗ call(A).
prove(neg(A)) ← neg lin(neg(A)).

doleft(atom(A),A).
doleft(forall(X,B),A) ← exists(X,doleft(B,A)).
doleft((atom(A) ← G),A) ← neg lin(G) ⅋ select.

select ← type(T) & (get(T,A) ⊗ prove(A)).
type(T) ← type(sync,T).

type(C,T) ← once(lin(A)) ⊗ atomtype(A,D) ⊗ lub(C,D,E) ⊗ type(E,T).
type(T,T).

get(async,A) ← once(lin(A) ⊗ atomtype(A,async)).
get(sync,A) ← lin(A) ⊗ atomtype(A,sync).

atomtype(⊗(_,_),sync).          atomtype(⊕(_,_),sync).
atomtype(exists(_,_),sync).     atomtype(1,sync).
atomtype(!(_),sync).            atomtype(once(_),sync).
atomtype(atom(_),sync).         atomtype(neg(_),sync).
atomtype(⅋(_,_),async).         atomtype(&(_,_),async).
atomtype(⊤,async).              atomtype(⊥,async).
atomtype(?(_),async).

lub(sync,X,X).    lub(async,async,async).    lub(async,sync,async).

shell ← repeat ⊗ print('Meta2 ') ⊗ readgoal(X) ⊗
      once(
        (eq(X,atom(end_of_file)) ⊗ nl) ⊕ (prove(X) ⊗ output(yes)) ⊕ output(no))
      ⊗ eq(X,atom(end_of_file)).
```

Program 28 Adding Rules to the Meta Interpreter

```
prove(atom(A)) ← rule(A,In ⇒ Out) ⊗ dorule(In,Out).

dorule([],[A,B|As]) ← (neg lin(neg(A))) ⅋ dorule([],[B|As]).
dorule([],[A]) ← (neg lin(neg(A))).
dorule([A|As],X) ← lin(neg(A)) ⊗ dorule(As,X).
```

We encode the Yale shooting problem (see program 23) using the rules:

```
rule(shoot, [alive,loaded] ⇒ [dead,unloaded]).
rule(load,  [unloaded] ⇒ [loaded]).
rule(init,  [] ⇒ [unloaded, alive]).
```

5.7 Other Programs

This section contains a number of other examples that did not fit into a previous category. These programs have potential for future work on programming idioms in Lygon.

An *exception* is a programming language construct used to handle error situations without having error handling code sprinkled throughout the program. Exceptions are used in a range of languages including SML, Ada, Java and ISO Prolog. Exceptions are manipulated using the two primitives *catch* and *throw*. The catch construct refers to two blocks of code. The first is executed. If an exception is thrown (using "throw") within this block (and is not caught by an enclosed catch block) then execution continues in the second block (which is indicated by the keyword "handle" in the following code). If no exception is generated then the second block is ignored. For example, in the following code, `Hello` is printed and an exception is thrown. The exception is caught by the inner catch and `World` is printed. Since the exception was caught, it is not propagated to the outer catch and its second block is ignored.

```
catch anException in
    catch anException in
        print(Hello);
        throw anException;
        print(There);
    and handle it by doing
        print(World);
    end
and handle it by doing
    print(Exception Raised);
    print(Aborting);
end
```

Program 29 describes a form of exception handling in Lygon. The method relies on the use of $\top$ to consume and thus abort the rest of the computation.

The first clause states that if the goal *raise(Y)* is ever present then we can use the axiom rule, unifying *X* and *Y* and then pass the rest of the linear environment (including

the rest of the computation) to $\top$ for consumption. The *!* around the handler forces the computation to be consumed by the $\top$. In the case that no exception is raised we need to allow *catch* to be ignored which is done by the second clause.

The goal *go1* has a single solution which prints `handle: 3`. The goal *go2* has a single solution which prints `Result: 3`.

Program 29 Exceptions

```
catch ⅋ raise(X) ← ⊤ ⊗ (!handler(X)).
catch ← ⊥.

handler(X) ← !(print('handle: ') ⊗ output(X)).
result(X) ← !(print('Result: ') ⊗ output(X)).

go1 ← catch ⅋ compute1.
go2 ← catch ⅋ compute2.

compute1 ← is(X,1+2) ⊗ (raise(X) ⅋ result(X)).
compute2 ← is(X,1+2) ⊗ result(X).
```

We move now to an application of Lygon to parsing. In [35] *constraint multiset grammars* are used to parse visual programming languages. Since the linear context is just a multiset this fits in quite nicely with Lygon and suggests that a Lygon implementation of a parser for constraint multiset grammars ought to be straightforward.

The idea behind multiset grammars is that basic pictorial elements such as circles, lines and text are terminals in a multiset. Grammar rules operate on *multisets* of symbols rather than on sequences of symbols. This makes sense since a picture does not have the natural notion of a sequence of symbols that is present in a string.

For example given the multiset { *dot(p(30,70)), line(p(10,20), p(30,70)), text(p(10,20), "Some Text")* } the following rule, which states that an arrow is a line which ends at the position of a dot *arrow(S,E)* $\leftarrow$ *line(S,E)* $\otimes$ *dot(P)* $\otimes$ *close(P,E)* can be used to yield the multiset { *text(p(10,20),"Some Text"), arrow(p(10,20),p(30,70))* } containing a terminal and a nonterminal. The predicate *close* takes two locations (of the form *p(x,y)* and succeeds if they are within ϵ of each other.

The example below parses finite state machines. The grammar contains four termi-

nals symbols:

1. *dot(Posn)* which indicates the pointing end of an arrow.
2. *line(Start,End,Middle)*
3. *text(Posn,Text)*
4. *circle(Posn,Radius)*

Figure 5.6 Finite State Machine

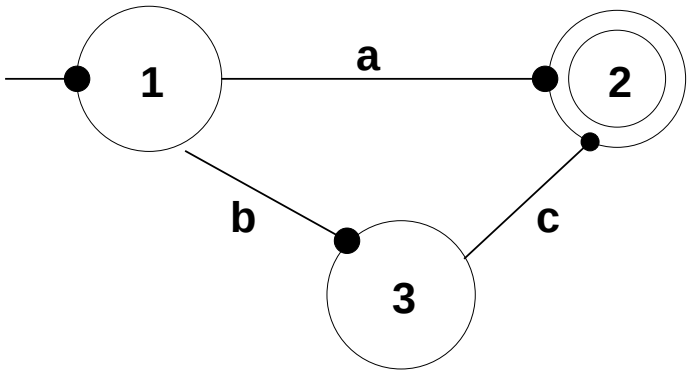


Figure 5.6 can be represented as

figure ←
neg circle(p(20,10),10) ⅋ **neg** circle(p(50,40),10) ⅋ **neg** circle(p(90,10),10) ⅋
neg circle(p(90,10),7) ⅋ **neg** line(p(10,0),p(10,10),p(10,5)) ⅋
neg line(p(20,20),p(40,40),p(30,30)) ⅋ **neg** line(p(70,10),p(30,10),p(50,10)) ⅋
neg line(p(60,40),p(90,30),p(75,30)) ⅋ **neg** dot(p(10,10)) ⅋
neg dot(p(70,10)) ⅋ **neg** dot(p(40,40)) ⅋ **neg** dot(p(90,30)) ⅋
neg text(p(20,10),1) ⅋ **neg** text(p(90,10),2) ⅋ **neg** text(p(50,40),3) ⅋
neg text(p(30,30),b) ⅋ **neg** text(p(50,10),a) ⅋ **neg** text(p(75,30),c).

The grammar also defines a number of non-terminals which we represent using rules which consume the required terminals.

- An *arrow* is formed from a line and a dot at the arrow's end. Arrows have a start, an end and a middle – *arrow(Start,End,Middle)*.
- An *arc* is a labelled arrow. It has a start, an end, a middle and some text *arc(Start,End,Middle,Text)*.

- A *startarc* is an arrow which does not have an attached label. It indicates the initial state and has only an end.
- There are three types of *states*, all have a middle, a radius, a label and a tag identifying their type – *state(Middle,Radius,Label,Type)*. All include a circle and some text.
  - A *final* state consists of two nested circles and a label.
  - A *start* state is a state which is the target of a *startarc*.
  - A state which is not one of the above is *normal*.
- A *trans(ition)* is an arc between two states. The rule given in [35] is awkward in Lygon since it makes use of non-consuming sub-derivations. The basic problem is that a transition needs to be able to reference its source and destination states; however there can be more than one transition to or from a given state. We solve this problem by parsing in two stages – in the first stage the states are derived and a note of the states derived is made in the non-linear predicates *statem*. In the second phase the transitions between the states are parsed. The predicate *statem* implements a form of memoing.

The program operates by starting off with a multiset containing terminal atoms. It then applies rules which combine these primitive picture elements into higher level elements such as arcs and transitions.

Note the use of negation as failure. The **not** predicate is defined in the Lygon standard library (program 1). The semantics of negation as failure mirrors Prolog – the goal **not***(G)* succeeds if **call***(G)* fails and fails if **call***(G)* succeeds.

The goal *figure* $\otimes$ *go(X,Y)* has the single solution
X = [state(p(50,40),10,3,normal), state(p(20,10),10,1,start), state(p(90,10),20,2,final)]
Y = [trans(3,2,c), trans(1,3,b), trans(1,2,a)].

Another Lygon application is the Hamming sequence. This program illustrates how lazy streams can be represented in Lygon. The Hamming sequence consists of numbers of the form $2^i 3^j 5^k$ in ascending order. A simple algorithm for generating these numbers uses streams (see figure 5.7). We seed the input with 1. We can generate further numbers

Program 30 Parsing Visual Grammars

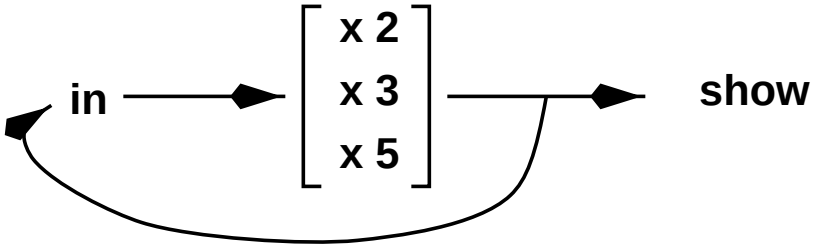
```
arrow(S,E,M) ← (line(S,E,M) ⊕ line(E,S,M)) ⊗ dot(P) ⊗ close(P,E).
arc(S,E,M,T) ← arrow(S,E,M) ⊗ text(P,T) ⊗ close(P,M).
startarc(P) ← arrow(S,P,M) ⊗ not(text(M1,_) ⊗ close(M1,M)).

state(M,R1,N,final) ← circle(M1,R1) ⊗ circle(M2,R2) ⊗
      close(M1,M2) ⊗ lt(R2,R1) ⊗ text(M,N) ⊗ close(M,M1).
state(M,R,N,start) ← startarc(E) ⊗ circle(M,R) ⊗ oncircle(E,M,R) ⊗
      text(P,N) ⊗ close(P,M) ⊗ not(circle(M,_)).
state(M,R,N,normal) ← circle(M,R) ⊗ text(P,N) ⊗ close(P,M) ⊗
      not(circle(M,_)) ⊗ not(startarc(P2) ⊗ oncircle(P2,M,R)).

trans(S,E,L) ← arc(Sa,Ea,Ma,L) ⊗
      ((statem(M1,R1,S,_) ⊗ ⊤) &
       (statem(M2,R2,E,_) ⊗ not(eq(M1,M2)) ⊗ ⊤) &
       (oncircle(Sa,M1,R1) ⊗ oncircle(Ea,M2,R2) ⊗ ⊤)).

states([]) ← not(state(_,_,_,_)).
states([state(A,B,C,D)|X]) ← once(state(A,B,C,D)) ⊗ states(X).

transitions([]) ← not(trans(_,_,_)).
transitions([trans(A,B,C)|X]) ← once(trans(A,B,C)) ⊗ transitions(X).

go(States,Transitions)
go(X,Y) ← states(X) ⊗ (put(X) ⅋ transitions(Y)).

put([]) ← ⊥.
put([state(A,B,C,D)|Xs]) ← (? neg statem(A,B,C,D)) ⅋ put(Xs).

oncircle(P1,P2,R) ← distance(P1,P2,R1) ⊗ closev(R,R1).
close(A,B) ← distance(A,B,X) ⊗ eps(Epsilon) ⊗ lt(X,Epsilon).
closev(A,B) ← is(D,A-B) ⊗ sqr(D,E) ⊗ eps(Eps) ⊗ lt(E,Eps).
eps(1.0).
distance(p(X1,Y1),p(X2,Y2),R) ← is(Dx,X1-X2) ⊗ is(Dy,Y1-Y2) ⊗
      sqr(Dx,Dxs) ⊗ sqr(Dy,Dys) ⊗ is(S,Dxs+Dys) ⊗ sqrt(S,R).
sqrt(X,Y) ← is(Y,pow(X,0.5)).
sqr(X,Y) ← is(Y,pow(X,2.0)).
```

in the sequence by multiplying existing numbers by $2, 3$ or 5. Each number generated is both output and fed back into the process to get further numbers.

In program 31 we represent a stream as a collection of linear facts. The *times* predicate multiplies its arguments. If the result is greater than the limit then it is ignored otherwise it is cycled back into *in* and copied to *show*. Once the sequence has been generated up to a given limit we collect and display the results. The query *go(30)* prints

```
[2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27]
```

Figure 5.7 Diagram of Hamming Processes



Our final example involves bottom up computation. Logic programming languages such as Prolog and Lygon are *top down* – given a clause such as $p \leftarrow q \wedge r$ and a query p they attempt to show that p holds by showing that $q \wedge r$ holds.

The complementary approach – used in deductive databases – is *bottom up*. Here we begin with the knowledge that q and r hold and we then conclude from the rule that p must hold. Since bottom up computation is not goal directed, generalising it to linear logic is an open area of research [61].

Program 32 illustrates how we can emulate simple bottom up processing in Lygon. We have a collection of facts which are known to hold and we apply rules "backwards" to derive new facts. To avoid inefficiency we tag each fact with its "generation number" – initial facts are tagged with 0 and a fact derived from facts tagged n and m is tagged with $\max(m, n) + 1$.

The predicate *do* has an argument which indicates the current generation being processed. To avoid re-generating facts, one of the facts being used must be from this generation. For example, if we are processing generation 2 then we can use a fact from generation 2 and one from generation 1 to generate a new fact which will be of generation 3.

Program 31 Hamming Sequence Generator

```
go(Lim) ← go(Lim,X) ⊗ nodupsort(X,X2) ⊗ output(X2).
go(Lim,L) ← in(1,Lim) ⅋ collect(L).

in(X,Lim) ← times(2,X,Lim) ⅋ times(3,X,Lim) ⅋ times(5,X,Lim).

times(X,Y,Lim) ← is(Z, X * Y) ⊗
    ((lt(Z,Lim) ⊗ (in(Z,Lim) ⅋ neg show(Z))) ⊕
    (ge(Z,Lim) ⊗ ⊥)).

Collects all shows into list ...
collect([Z|X]) ← once(show(Z)) ⊗ collect(X).
collect([]).

nodupsort([],[]).
nodupsort([X],[X]).
nodupsort([X,Y|Xs],R) ← halve([X,Y|Xs],A,B)
      ⊗ nodupsort(A,R1) ⊗ nodupsort(B,R2) ⊗ merge(R1,R2,R).

merge([],X,X).
merge([X|Xs],[],[X|Xs]).
merge([X|Xs],[Y|Ys],[X|R]) ← lt(X,Y) ⊗ merge(Xs,[Y|Ys],R).
merge([X|Xs],[Y|Ys],[Y|R]) ← gt(X,Y) ⊗ merge([X|Xs],Ys,R).
merge([X|Xs],[Y|Ys],R) ← eq(X,Y) ⊗ merge([X|Xs],Ys,R).
```

The particular application is path finding. The predicate *canadd* finds an edge in the current generation and another edge and checks that the new composite edge derived is not present in a previous generation. Note the use of & to avoid consuming the edges being checked.

The clauses for *do* simply find a new edge using *canadd* and add it. If no more edges can be added in the current generation then the generation is incremented and *checkdo* is called. The predicate *checkdo* calls *do* if edges were added to the previous generation and terminates otherwise.

Consider running the goal *go(X)*. The graph described is that of figure 5.2. In the first generation we add composite edges from *a* to *c*, *b* to *d*, *c* to *e*, *c* to *a* and *d* to *b*.

The second generation adds composite edges from *a* to *d*, *b* to *a*, *b* to *c*, *c* to *b* and *d* to *c* using an initial edge and a first generation edge and adds composite edges from *a* to *e*, *a* to *a*, *b* to *b*, *c* to *c* and *d* to *d* using pairs of first generation edges. No more composite edges can be derived and so the goal returns the answer:
X = [e(d,e,0), e(d,a,0), e(c,d,0), e(b,c,0), e(a,b,0), e(a,c,1), e(b,d,1), e(c,a,1), e(c,e,1), e(d,b,1), e(d,d,2), e(d,c,2), e(c,c,2), e(c,b,2), e(b,b,2), e(b,a,2), e(b,e,2), e(a,e,2), e(a,a,2), e(a,d,2)]

Program 32 Bottom Up Computation

```
go(X) ← graph ⅋ neg result(X) ⅋ do(0).

graph ←
 neg edge(a,b,0) ⅋ neg edge(b,c,0) ⅋ neg edge(c,d,0) ⅋ neg edge(d,a,0) ⅋ neg edge(d,e,0).

canadd(N,A,C,NR) ← (edge(A,B,N) ⊗ edge(B,C,N1) ⊗ le(N1,N) ⊗ is(NR,N+1) ⊗ ⊤) &
      (not(edge(A,C,_) ⊗ ⊤) ⊗ ⊤).

do(N) ← canadd(N,A,B,NR) & (neg edge(A,B,NR) ⅋ do(N)).
do(N) ← not(canadd(_,_,_)) ⊗ is(N1,N+1) ⊗ checkdo(N1).

checkdo(N) ← edge(A,B,N) ⊗ (neg edge(A,B,N) ⅋ do(N)).
checkdo(N) ← not(edge(A,B,N)) ⊗ collect([]).

collect(X) ← once(edge(A,B,N)) ⊗ collect([e(A,B,N)|X]).
collect(X) ← result(X).
```

5.8 Discussion

In this chapter we have presented a range of applications of Lygon. These illustrate that the linear logic aspects of the language do indeed gain significant additional expressiveness. Many of the programs could also be written in another linear logic programming language (see section 6.1). However only Lygon and Forum are capable of expressing all of the examples presented.

In the process of developing the programs we have noted a number of idioms that occur frequently in Lygon programs. These programming patterns form the beginnings of a programming methodology for linear logic programming languages. An example is the use of $\top$ to simulate an affine mode where certain resources can be ignored but not duplicated.

A number of the common programming idioms have a clumsy syntax; perhaps the most glaring example is the use of a continuation passing style to enforce a certain sequence of operations. One area for further work is the use of *syntactic sugar* to make common idioms less syntactically clumsy. Rules (see program 28 and [154]) are an example of syntactic sugar.

Another area for further work concerns type systems. Type systems for logic programming languages [121] describe the usage of *terms*. In linear logic programming languages it makes sense to also consider a form of typing which operates at the level of *predicates*. That is, rather than describe the possible structure of the arguments to a predicate, the type system would describe the possible effects of the predicate on the linear context. If, for example, we know which linear predicates are added and removed by each predicate then it is possible to detect at compile time that a goal can not consume a certain linear predicate. This notion of predicate-level typing is similar to the results of the analysis presented in [12] and to the type system of [87].

Chapter 6

Comparison to Other Work

In this chapter we survey and compare the variety of logic programming languages which are based on linear logic. We also briefly relate our work to other uses being made of linear logic in the programming language research community. In particular, we look at functional programming languages based on linear language and at *uniqueness types*.

Uniqueness types apply the concept of linearity to *values* in a (usually declarative) programming language. Uniqueness types exist in Clean [19–21, 130], Mercury [64, 132, 133] and in a version of Lisp developed by Henry Baker [13–16].

Uniqueness types are inspired by linear logic but in general they make use of a very limited subset of the logic. The two main uses of uniqueness types are to guarantee single threading of side effecting I/O operations and to ensure single threading of data structures — in particular arrays — to ensure that in place updating can be done safely. This second application can be seen as a garbage collection issue [33, 143].

A related application of linear logic is its use to derive functional programming languages. This is done using the Curry-Howard isomorphism which states that there is an equivalence between theorems and types in the λ-calculus. This has been applied by Mackie [98, 99], Abramsky [1], Lafont [91, 92] and Lafont and Girard [49] to derive a linear version of the λ-calculus. The resulting languages are rather different to logic programming languages. Formally, the execution model in functional programming is based on the *reduction* of proofs through cut elimination whereas in logic programming the execution model is based on proof search. More intuitively, in a functional programming

language based on linear logic, linearity is applied to *values* - linearity thus *limits* what the programmer can write. In a logic programming language based on linear logic linearity is applied to *predicates* and gives the programmer *more* ways of doing things.

The application of linear logic to values is orthogonal to its application to predicates – it is quite possible to visualise a variant of, say, Lygon which restricts values to being used linearly and thus avoids the need for a garbage collector.

6.1 Linear Logic Programming Languages

In recent years a number of logic programming languages based on linear logic have been proposed. These tend to fall into two classes based on how they use linear logic:

1. Languages which extend Prolog with linear logic features, and

2. Concurrent languages.

Some languages – for instance Lygon and Forum – both extend Prolog *and* use linear logic to add concurrency. In general, the concurrent languages tend to have *ad hoc* designs and allow a fairly limited class of formulae. Languages in this class are LO, ACL and $\mathcal{LC}$.

The first class of languages tend to have been designed using a proof theoretical analysis. These languages generally allow a rich class of linear logic formulae to be used. Languages which fall into this group are Lolli, Lygon$_2$ and Forum.

As we shall see, Lygon$_2$ is a superset of most other linear logic based logic programming languages. The exception is Forum which seems to be roughly equivalent to Lygon (as proposed in [122]) and as a result is harder to implement than Lygon$_2$.

A survey of the languages can be found in [110]. The syntax of the various languages are summarised in figure 6.1. Note when comparing Lygon$_2$ to Forum and Lolli that the Lygon$_2$ set of program clauses is the normal form of the larger set:

$$\mathcal{D} ::= !\mathcal{D}_n \mid \mathcal{D}_l$$

$$\mathcal{D}_n ::= A \mid \mathcal{D}_n \parr \mathcal{D}_n \mid \mathcal{D}_n \,\&\, \mathcal{D}_n \mid \forall \mathcal{D}_n \mid \mathcal{G} \multimap \mathcal{D}_n$$

$$\mathcal{D}_l ::= \top \mid A \mid \mathcal{D}_l \parr \mathcal{D}_l \mid \mathcal{D}_l \,\&\, \mathcal{D}_l \mid \forall x \mathcal{D}_l \mid \mathcal{G} \multimap \mathcal{D}_l$$

The same normal form can also be applied to Forum and Lolli.

LO and LinLog

LO (Linear Objects) [8–10] is one of the earlier languages proposed. The motivation behind its design is to take the object oriented programming paradigm as realised in first generation concurrent logic programming languages (see section 6.2) and realise it in a concurrent language based on linear logic. LO's design extends the ideas of committed clause logic programming languages with multi-headed clauses (i.e., clauses of the form $!(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n))$). However the class of goal formulae permitted is fairly limited. LO has been implemented but the implementation is not publicly available.

A novel mechanism introduced in the language is a *broadcast marker*. Some of the formulae in the head of a clause can be marked for broadcast. These formulae are not removed from the relevant context but *added* to the *initial* context. The initial query is seen as containing an unknown region which is determined as broadcasts are processed.

More recent work with LO involves optimising communication by abstract interpretation [11, 12] and the use of the language as a co-ordination language [29]. In [42] LO is used to co-ordinate a distributed diary system whose components were written in Prolog.

With the exception of the broadcast mechanism the language is a subset of a number of other languages including ACL, Forum and Lygon.

A related language to LO is LinLog. LinLog has not, to the best of my knowledge, been implemented. It is similar to Forum in that it is "complete for the whole of linear logic". In other words, there exists a translation algorithm which can encode an arbitrary linear logic proof into a proof that consists of LinLog goals and programs.

An interesting property of LinLog is that selecting a goal formula needs to be done using "don't know" nondeterminism but once a goal formula is selected it can be completely reduced to atoms before another goal needs to be considered. This property is achieved by requiring LinLog goals to consist of synchronous connectives outside asynchronous connectives. For example, the formula $p \parr (q \otimes r)$ is not a valid LinLog goal since the $\parr$ (asynchronous) occurs outside of a synchronous connective ($\otimes$).

Figure 6.1 Linear Logic Based Logic Programming Languages

LO [10]:

$$\mathcal{D} ::= !\forall\overline{x}(\mathcal{G} \multimap A_1 \parr \ldots \parr A_n) \qquad \mathcal{G} ::= A \mid \top \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \parr \mathcal{G}$$

LinLog **[6]:**

$$\mathcal{D} ::= !\forall\overline{x}(\mathcal{G} \multimap A_1 \parr \ldots \parr A_n)$$
$$\mathcal{G} ::= \mathcal{H} \mid !\mathcal{H} \mid \mathbf{1} \mid \mathcal{G} \otimes \mathcal{G} \mid \mathbf{0} \mid \mathcal{G} \oplus \mathcal{G} \mid \exists x\mathcal{G}$$
$$\mathcal{H} ::= A \mid ?A \mid \bot \mid \mathcal{H} \parr \mathcal{H} \mid \top \mid \mathcal{H} \,\&\, \mathcal{H} \mid \forall x\mathcal{H}$$

ACL [85]:

$$\mathcal{D} ::= !\forall\overline{x}(\mathcal{G} \multimap A_p)$$
$$\mathcal{G} ::= \bot \mid \top \mid A_m \mid ?A_m \mid A_p \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \forall x\mathcal{G} \mid \mathcal{R}$$
$$\mathcal{R} ::= \exists\overline{x}(A_m^{\perp} \otimes \ldots \otimes A_m^{\perp} \otimes \mathcal{G}) \mid \mathcal{R} \oplus \mathcal{R}$$

$\mathcal{LC}$ **[140]:**

$$\mathcal{D} ::= !\forall\overline{x}(\mathcal{G} \multimap A_1 \parr \ldots \parr A_n) \qquad \mathcal{G} ::= A \mid \mathbf{1} \oplus \bot \mid \top \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \exists x\mathcal{G}$$

Lolli [70]:

$$\mathcal{D} ::= \top \mid A \mid \mathcal{D} \,\&\, \mathcal{D} \mid \mathcal{G} \multimap \mathcal{D} \mid \mathcal{G} \Rightarrow \mathcal{D} \mid \forall x.\mathcal{D}$$
$$\mathcal{G} ::= \top \mid A \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D} \Rightarrow \mathcal{G} \mid \forall x.\mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathbf{1} \mid \mathcal{G} \otimes \mathcal{G} \mid !\mathcal{G} \mid \exists x\mathcal{G}$$

Forum [106]:

$$\mathcal{D} ::= \mathcal{G} \qquad \mathcal{G} ::= A \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \multimap \mathcal{G} \mid \mathcal{G} \Rightarrow \mathcal{G} \mid \top \mid \bot \mid \forall x\mathcal{G}$$

Lygon$_2$ (Section 3.11)

$$\mathcal{D} ::= \ (C_1 \,\&\, \ldots \,\&\, C_n) \mid !C$$
$$C ::= \forall\overline{x}(\mathcal{G} \multimap (A_1 \parr \ldots \parr A_n))$$
$$\mathcal{G} ::= A \mid \mathbf{1} \mid \bot \mid \top \mid \mathcal{G} \otimes \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathcal{G} \parr \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D}^{\perp} \mid \forall x\mathcal{G} \mid \exists x\mathcal{G} \mid !G \mid ?G$$

The class of formulae allowed in LinLog is a subset of the Lygon_2 class of formulae.

Since there exists an algorithm [6] which can translate an arbitrary proof in linear logic to one in the LinLog fragment, it follows that any proof in linear logic can be carried out within Lygon_2 via a syntactic translation. This result is not actually of much practical use – although a proof will exist in the Lygon_2 fragment of linear logic we can not necessarily expect the Lygon system to be able to find it efficiently.

ACL

ACL [85, 88] (A Concurrent Language) uses linear logic as the basis for a concurrent programming languages. The actual language[1] [128] is an ML-like functional programming language with concurrency primitives inspired by linear logic's connectives.

Some early work with ACL looked at process equivalence relations [86]. More recent work centers around the development of language tools such partial evaluators [73] and abstract interpreters [84].

ACL has also been used as an implementation language for a concurrent object-oriented programming language and in this context there has been work on providing a type system for the language which is capable of detecting "message not understood" errors at compile time [87].

Like LO, ACL sacrifices completeness by implementing "don't care" nondeterminism. The class of formulae used in ACL is a subset of the Lygon_2 class of formulae.

$\mathcal{LC}$

$\mathcal{LC}$ (Linear Chemistry) [140] is a minimal concurrent language fairly similar in capabilities to ACL. It is interesting for a number of reasons. Firstly, it was systematically derived using a generalisation of uniformity. Secondly, it makes use of the formula $\mathbf{1} \oplus \bot$ as a primitive which terminates a single process – if there are other processes then the process quietly vanishes (using $\bot$) otherwise the proof is successful (using $\mathbf{1}$). Finally, an interesting property of the language is that all proofs in the $\mathcal{LC}$ fragment of linear logic are actually sticks rather than trees – each inference rule has at most one premise.

[1] Available from `camille.is.s.u-tokyo.ac.jp:pub/hacl`.

$\mathcal{LC}$ has, to the best of my knowledge, not been implemented. The class of formulae allowed in $\mathcal{LC}$ is a subset of the Lygon$_2$ class. Since $\mathcal{LC}$ (like ACL and LO) sacrifices completeness by implementing "don't care" nondeterminism, it is incapable of expressing programs involving backtracking such as the graph search programs of section 5.3.

Lolli

Lolli [66, 69, 70] (named for the lollipop connective of linear logic ($\multimap$)) was the first linear logic based logic programming language to be designed using uniformity. Apart from being based on intuitionistic linear logic and thus not allowing $\otimes$ (which is needed for the concurrent applications of linear logic) Lolli is a rich language, particularly in its class of program formulae. The Lolli implementation [68] is available from its World Wide Web page [67]. Note that $F \Rightarrow G$ is defined as $(!F) \multimap G$, this is also the case for Forum.

The language has been used for a number of example programs [66] which demonstrate basic techniques of linear logic programming (for example, permutations, toggling etc.). It has also been used to specify filler-gap dependency parsers [65]. Lolli has been used to implement event calculus programs by a number of researchers [5, 31].

Lolli and Lygon are fairly similar in methodology. Roughly speaking, Lolli can be seen as Lygon minus concurrency. Indeed, the class of formulae usable in Lygon$_2$ is a superset of the Lolli class.

Forum

In some aspects Forum [106, 109] is fairly similar to Lygon. The main difference is that Forum satisfies a variant of asynchronous uniformity which does not require that atomic goals guide the proof search process. Forum includes $\bot$ in programs and in clause heads and as a result it is more of a specification language than a logic programming language. Both of Forum's designers have commented [71, 111] that Forum does not appear to be a logic programming language. Two of the reasons why Forum is too expressive to be considered a logic programming language involve higher order quantification. The third is the presence of program clauses of the form $\mathcal{G} \multimap \bot$. Such clauses can be resolved

against at any time regardless of the goal.

Forum has a naïve implementation in λ-Prolog and an SML implementation (similar in style to Lolli's implementation) which is available from
`ftp://ftp.cs.hmc.edu/pub/hodas/Forum/forum.tar.Z`.

Work on Forum has mostly concentrated on its use as a specification language. It has been applied to the specification of an ML-like language, a simple RISC processor [34] logical inference systems in both natural deduction and sequent calculus styles [106] and transition systems [104].

There is also a group at Università di Genova working on object oriented programming in a linear logic framework which use Forum [38].

The class of program formulae permitted by Lygon$_2$ is precisely a normalised form [106] of the Forum class of program formulae excluding $\perp$ headed clauses. Forum's class of goal formulae is limited to asynchronous formulae. Although an attempt is made in [106, 109] to add synchronous connectives (such as $\exists$) using logical equivalences the synchronous connectives are not first class citizens – for example there exists a program P and formula F such that $P \vdash \exists x F$ is provable but where there isn't a term t such that $P \vdash F[t/x]$ is provable. Lygon$_2$, on the other hand, fully supports synchronous connectives in goals as first class citizens. Note that the optimisations presented in section 4.4 allow Lygon$_2$ programs which only use the Forum subset of goal formulae to execute as efficiently as Forum. Thus, with the exception of (i) higher orderness (which is orthogonal), and (ii) $\perp$ headed clauses (which are undesirable in a logic programming language), Lygon$_2$ is a strict superset of Forum.

6.2 Concurrent Programming

One of the more exciting aspects of linear logic based logic programming languages is their natural applicability to concurrency. It is interesting to compare the linear logic approach to concurrent logic programming with other approaches.

There are three main "generations" of concurrent logic programming languages:

1. Committed Choice Guarded Horn Clause Languages

2. Concurrent Constraint Programming Languages

3. Linear Logic Based Languages

The first generation of concurrent logic programming languages emerged in the early 80's. These languages dropped backtracking from Prolog and modified unification so it would suspend under certain conditions. For example, in GHC, unifying a goal A with the head of clause cannot bind variables in A. If the unification could succeed by binding a variable in A then it *suspends*. This suspension is the synchronisation primitive.

Languages in this first generation included Parlog [50], Concurrent Prolog [125, chapters 2 & 5], GHC [125, chapter 4] and, later, Strand [43]. Although these languages have become less prominent in the research community they are still alive – see for example [76, 77]. A survey of "first generation" concurrent logic programming can be found in [126].

The second generation includes such languages as AKL [80] and Oz [39]. These languages move away from the proof search interpretation of computation and view processes as operating over a shared constraint store. As a result it becomes difficult to view these languages as logic programming languages. Since a general comparison between linear logic programming languages' approach to concurrency and more general approaches (e.g., rendezvous, remote procedure call (RPC) etc.) is beyond the scope of this section, we shall not discuss these languages further. For a survey of approaches to concurrency we refer the reader to [17].

Looking at the linear logic way of doing concurrency in a logic programming framework, one of the main differences from first generation languages is that unification is left unchanged – linear logic provides new operations which allow communication and synchronisation to be expressed. The primary advantage of this is that linear logic based concurrent logic programming languages have semantics which are both sound *and complete* with respect to the logic. Although first generation concurrent logic programming languages are in general sound, they rely heavily on the programmer guiding the proof search and are far from complete. Note that second generation languages also suffer from a lack of completeness with respect to the logical semantics – there is a distinction between *asking* a constraint and *telling* a constraint. If a constraint is used wrongly, say, it is

made an ask rather than a tell, then the implementation may be unable to find a solution. In the case where an ask is wrongly made into a tell the system could fail.

Although most of the concurrent logic programming languages based on linear logic have also sacrificed completeness by opting for "don't care" nondeterminism (i.e., abandoning backtracking) it is possible [131] to use languages with backtracking for concurrent programming and guarantee at compile time that the appropriate parts of the program will not backtrack.

A problem with sacrificing completeness is that it opens a gap between the simple declarative semantics and the visualisation semantics described in section 3.12. Concurrent logic programming languages based on linear logic have, or in some cases have the potential for, simple logical semantics.

On a more pragmatic level, it appears that linear logic based concurrent logic programming languages are more expressive than older proposals. Additionally, they do not suffer from the problem that multiple inputs to a process need to be merged by the programmer – leading to both spaghetti code and a loss of efficiency (which can be fixed by adding an additional construct to the language – see [125, chapters 15 & 16]). This problem also affects "second generation" concurrent logic programming languages such as AKL [80, chapter 7].

Chapter 7

Conclusions and Further Work

This thesis has covered the *design, implementation* and *applications* of the linear logic programming language *Lygon*.

In order to design Lygon we defined and compared a number of characterisers of logic programming. The notion of *left-focused* proof steps was applied to capture the intuition that a resolution step is guided by the atomic goal. The characterisers were generalised to the multiple conclusion setting and Lygon was derived using one of the two generalisations of our extension of uniformity.

In chapter 4 we tackled the implementation aspects and systematically derived an efficient set of rules for the management of resources in a linear logic proof. Soundness and completeness were shown. The problem of active formula selection was also tackled and known properties of linear logic were applied to yield a (partial) solution to this problem.

Finally, in chapter 5 we investigated methodologies and idioms for Lygon programming. A range of program idioms were identified including the use of $\top$ to simulate an affine mode, the use of tokens to control and terminate a concurrent computation, the use of $\&$ to enable non-destructive testing of a linear context and a range of techniques for manipulating the linear context. Particular applications which were developed included concurrent programs, Lygon meta-interpreters and programs operating on graph data structures.

This thesis has demonstrated that a logic programming language based on multiple

conclusion linear logic can be systematically derived and effectively implemented. Such a language has its own set of programming idioms and offers significant additional expressiveness over classical logic programming languages such as Prolog.

The extra features added by the use of linear logic are useful in solving a variety of problems and add a number of language features which previously were handled in logic programming languages using ad hoc and non-logical means. Some features which are handled in a pure logical fashion in Lygon include concurrency, a form of fact assertion and retraction and of global state and modelling states and actions.

The work of this thesis forms a solid foundation for further work with Lygon. In addition to various issues mentioned in sections 3.12, 4.5 and 5.8 there are a number of areas for further work.

7.1 Implementation

The current Lygon implementation is an interpreter. Whilst it has been satisfactory for our work it is desirable in the longer term to investigate issues involved in the compilation of Lygon. In addition to the usual issues associated with compiling Prolog, the efficient compilation of Lygon requires some knowledge of how the linear context is used by the program. For example, if it can be determined that a certain predicate does not make any use of the linear context then we can avoid having to tag the context and pass it to the predicate. Likewise, if a predicate only makes use of a certain linear fact then we only need to tag a part of the linear context.

Another application of analyses of context usage concerns active formula selection. In a goal of the form p , Δ we can not commit to resolving p against a program clause since it is possible that Δ may introduce **neg** p at some point. If it can be determined that this is not the case then it may be possible to select p for resolution immediately. This is also influenced by the connectives which the program uses; given the program $p \leftarrow p$ and the goal $p \parr G$ we can not commit to resolving p if G contains $\top$.

Lygon can be seen as "Prolog + linear logic". Since these two aspects are orthogonal it is feasible to consider languages which combine linear logic features with, say, CLP($\Re$) [79] or Mercury [133]. One strategy for developing a Lygon compiler would be

to consider a "Mercury + Linear Logic" language and compile it into Mercury.

A second area with significant potential for implementation related further work is the debugging and visualisation of Lygon programs. The operational semantics of Lygon programs is considerably more complex than that of Prolog programs due to the presence of the linear context and the potential for having multiple goals evolving concurrently. The current Lygon debugger handles the linear context adequately but is not useful for debugging concurrent programs. In the long run it would seem desirable to make use of *algorithmic debugging* [127]. By debugging at the logical level the complex operational semantics can be avoided. A group at Melbourne University is working on advanced debugging environments for NU-Prolog, Mercury and Lygon.

7.2 Negation as Failure and Aggregates

Presently Lygon offers simple Negation as Failure a la Prolog. The issue of failure is significantly more interesting for linear logic than it is for classical logic. A proof can fail for a larger number of reasons – there could be an excess of resources for example. Limited forms of negation can actually be done in a pure fashion in Lygon. It is possible to use a goal of the form *! F* to check that the linear context is empty. Likewise, by using $\&$ and a predicate we can check that certain predicates are absent. If the linear predicates that may be present are *p*, *q* and *r* then the clause *consume* $\leftarrow$ *(((q* $\oplus$ *r)* $\otimes$ *consume)* $\oplus$ **1***)* and goal *consume* $\&$ *G* can be used to check that no linear predicates *p* are present. It is unclear to what extent Negation as Failure can be accomplished in pure Lygon.

An *aggregate* construct collects all solutions for a goal. Intuitively *solutions(*$\exists x$ *Goal,Solns)* is true if *Solns* contains a list of variable instantiations for all solutions of *Goal*:

$$\mathit{Goal}[x_1 \leftarrow t_1] \wedge \mathit{Goal}[x_2 \leftarrow t_2] \wedge \ldots \wedge \mathit{Goal}[x_n \leftarrow t_n] \wedge \mathit{Solns} = [t_1, t_2, \ldots, t_n]$$

In Lygon the question of resources arises. Should all solutions to *Goal* split the resources between them (i.e. use $\otimes$ instead of $\wedge$)? Should all solutions to *Goal* be required to consume the same resources (i.e. use $\&$ instead of $\wedge$)? Does it make sense to allow each solution to consume any resources desired and have the *solutions* predicate consume the entire context (i.e., ensure that *solutions* is always called as *solutions* $\&$ *G*)?

7.3 Other

Languages such as Lygon and Forum cleanly integrate concurrent behavior, backtracking, updateable state and symbolic computation. Current "hot" areas in which these languages could potentially be applied include agents (which can be viewed as concurrent planning (see sections 5.4 and 5.5)), applications on the Web (which generally involve concurrent symbolic processing) and co-ordination applications.

Other applications for linear logic programming languages include modelling database transactions and further work in artificial intelligence, such as belief revision.

The derivation of logic programming languages from non-classical logics has so far focused exclusively on top-down derivations. In the context of deductive databases [139] the complementary approach is of importance. Under the *bottom-up* execution model rules are applied to derive new facts from old facts. The implementation is typically "set at a time" rather than "tuple at a time" and the notion of a goal plays a much lesser role. For example, in classical logic, given the rule $p(X) \rightarrow q(X)$ and the facts $p(1)$ and $p(2)$ a bottom-up implementation will apply the rule to derive the new facts $q(1)$ and $q(2)$.

The derivation of bottom-up logic programming languages for non-classical logics has only recently begun to be explored [61]. Applications for the resulting languages include active databases.

Bibliography

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] Gul Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] Vladimir Alexiev. Applications of linear logic to computation: An overview. Technical Report TR93-18, University of Alberta, December 1993.

[4] Vladimir Alexiev. Applications of linear logic to computation: An overview. *Bulletin of the IGPL*, 2(1):77–107, March 1994.

[5] Vladimir Alexiev. The event calculus as a linear logic program. Technical Report 95-24, University of Alberta, 1995.

[6] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

[7] Jean-Marc Andreoli, Paolo Ciancarini, and Remo Pareschi. Interaction abstract machines. Technical Report ECRC-92-23, European Computer-Industry Research Centre (ECRC), ECRC GMBH Arabellastr. 17, D-81925 München, Germany, 1992.

[8] Jean-Marc Andreoli and Remo Pareschi. LO and behold! concurrent structured processes. *SIGPLAN Notices*, 25(10):44–56, 1990.

[9] Jean-Marc Andreoli and Remo Pareschi. Communication as fair distribution of knowledge. In Andreas Parpcke, editor, *Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 212–229, 1991.

[10] Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[11] Jean-Marc Andreoli and Remo Pareschi. Associative communication and its optimization via abstract interpretation. Technical Report ECRC-92-24, European Computer-Industry Research Centre (ECRC), ECRC GMBH Arabellastr. 17, D-81925 München, Germany, 1992.

[12] Jean-Marc Andreoli and Remo Pareschi. Abstract interprtation of linear logic programming. In D. Miller, editor, *International Logic Programming Symposium*, pages 295–314. MIT Press, 1993.

[13] Henry G. Baker. Lively linear Lisp – 'look ma, no garbage!'. *SIGPLAN Notices*, 27(8):89–98, August 1992.

[14] Henry G. Baker. Sparse polynomials and linear logic. *SIGSAM Bulletin*, 27(4):10–14, December 1993.

[15] Henry G. Baker. Linear logic and permutation stacks – the forth shall be first. *Computer Architecture News*, 22(1):34–43, March 1994.

[16] Henry G. Baker. A "linear logic" quicksort. *SIGPLAN Notices*, 29(2):13–18, February 1994.

[17] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[18] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[19] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, Computing Science Institute, University of Nijmegen, December 1993.

[20] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In R. Shyamasundar, editor, *Proceedings of the 13th Conference on the Foundations of Software Technology & Theoretical Computer Science (FST&TCS13)*, pages 41–51. Springer-Verlag, LNCS 761, 1993.

[21] Erik Barendsen and Sjaak Smetsers. Uniqueness type inference. In M. Hermenegildo and D. Swierstra, editors, *Proceedings of Programming Languages: Implementation, Logics and Programs (PLILP'95)*, pages 189–207. Springer-Verlag, LNCS 982, 1995.

[22] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Conference Record of the Seventeenth Annual Symposium on Principles of Programming Languages*, pages 81–94, San Francisco, California, January 1990.

[23] Edoardo S. Biagioni. A Structured TCP in Standard ML. Technical Report CMU-CS-94-171, Department of Computer Science, Carnegie-Mellon University, July 1994.

[24] A. W. Bollen. Relevant logic programming. *Journal of Automated Reasoning*, 7:563–585, 1991.

[25] A. Bonner and L. McCarty. Adding negation-as-failure to intuitionistic logic programming. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 681–703, Austin, October 1990. ALP, MIT Press.

[26] John Boreczky and Lawrence A. Rowe. Building Common Lisp Applications with Reasonable Performance. Technical Report UCB//CSD-93-763, University of California at Berkeley, Department of Computer Science, June 1993.

[27] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[28] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[29] Stefania Castellani and Paolo Ciancarini. Exploring the coordination space with LO. Technical Report UBLCS-94-6, University of Bologna, April 1994.

[30] Iliano Cervesato. Lollipops taste of vanilla too. In A. Momigliano and M. Ornaghi, editors, *Proof-Theoretical Extensions of Logic Programming*, pages 60–66, Santa Margherita Ligure, Italy, June 1994.

[31] Iliano Cervesato, Luca Chittaro, and Angelo Montanari. Modal event calculus in Lolli. Technical Report CMU-CS-94-198, School of Computer Science, Carnegie Mellon University, October 1994.

[32] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the Fifth International Workshop on Extensions of Logic Programming — ELP'96*, LNAI 1050, pages 67–81. Springer, 1996.

[33] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *Lisp and Functional Programming*, pages 139–150. ACM, 1992.

[34] Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, 1995.

[35] Sitt Sen Chok and Kim Marriott. Parsing visual languages. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 90–98, 1995.

[36] Andrew Davison. A concurrent logic programming encoding of Petri nets. In G. Gupta, G. Mohag, and R. Topor, editors, *Proceedings of the 16th Australian Computer Science Conference*, pages 379–386, 1993.

[37] W. De Hoon. Designing a spreadsheet in a pure functional graph rewriting language. Master's thesis, Catholic University Nijmegen, 1993.

[38] Giorgio Delzanno and Maurizio Martelli. Objects in Forum. In John Lloyd, editor, *International Logic Programming Symposium*, pages 115–129, Portland, Oregon, December 1995. MIT Press.

[39] Denys Duchier. The Oz programming system. `http://ps-www.dfki.uni-sb.de/oz/`, 1996.

[40] P. Dung. Hypothetical logic programming. In *Proc. 3rd International Workshop on Extensions of Logic Programming*, pages 61–73. Springer-Verlag, 1992.

[41] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, September 1992.

[42] Norbert Eisinger. A multi-agent diary manager coordinated with LO. Technical Report ECRC-93-11, European Computer-Industry Research Centre (ECRC), ECRC GMBH Arabellastr. 17, D-81925 München, Germany, July 1993.

[43] Ian Foster and Stephen Taylor. Strand: A practical parallel programming tool. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming, Proc. of the North American Conference*, pages 497–512, Cleveland, 1989. The MIT Press.

[44] Didier Galmiche and Guy Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135(1):67–110, December 1994.

[45] G. Gentzen. Investigations into logical deductions, 1935. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[46] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[47] Jean-Yves Girard. Linear logic : Its syntax and semantics. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, chapter 0. Cambridge University Press, 1995.

[48] J.Y. Girard. Logic and exceptions: A few remarks. Technical Report 18, Équipe de logique Mathématique, December 1990.

[49] J.Y. Girard and Y. Lafont. Linear logic and lazy computation. In *TAPSOFT*. Springer-Verlag LNCS 250, 1987.

[50] S. Gregory. *Parallel logic programming in PARLOG*. Addison Wesley, Reading, Mass., 1987.

[51] Alessio Guglielmi. Concurrency and plan generation in a logic programming language with a sequential operator. In Pascal Van Hentenryck, editor, *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, pages 240–254, Massachusetts Institute of Technology, 1994. The MIT Press.

[52] S. Hanks and D. MacDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[53] J. Harland. On normal forms and equivalence for logic programs. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 146–160, Washington, USA, November 1992. The MIT Press.

[54] James Harland. *On Hereditary Harrop Formulae as a Basis for Logic Programming*. PhD thesis, University of Edinburgh, 1991.

[55] James Harland. On goal-directed provability in classical logic. In *Proceedings of the ICLP '94 Post-conference Workshop on Proof-Theoretical Extensions of Logic Programming*, pages 10–18, Santa Margherita Ligure, June 1994.

[56] James Harland. A proof-theoretic analysis of goal-directed provability. *Journal of Logic and Computation*, 4(1):69–88, 1994.

[57] James Harland and David Pym. A note on the implementation and applications of linear logic programming languages. In Gopal Gupta, editor, *Seventeenth Annual Australasian Computer Science Conference*, pages 647–658, 1994.

[58] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: A brief overview. In John Lloyd, editor, *International Logic Programming Symposium*, page 636, Portland, Oregon, December 1995. MIT Press.

[59] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: A system demonstration. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, LNCS 1101, page 599. Springer, July 1996.

[60] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: An overview. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, LNCS 1101, pages 391–405. Springer, July 1996.

[61] James Harland, David Pym, and Michael Winikoff. Bottom-up execution of linear logic programming languages. Technical Report 97/1, Melbourne University, Department of Computer Science, Parkville 3052, AUSTRALIA, 1997.

[62] P. H. Hartel. Benchmarking implementations of lazy functional languages II – two years later. Technical Report CS-94-21, Dept. of Comp. Sys, Univ. of Amsterdam, December 1994.

[63] Pieter Hartel and Koen Langendoen. Benchmarking implementations of lazy functional languages. In *Functional Programming & Computer Architecture*, pages 341–349, Copenhagen, June 93.

[64] Fergus Henderson. Home page of the Mercury project. `http://www.cs.mu.oz.au/mercury`, 1996.

[65] Joshua Hodas. Specifying filler-gap dependency parsers in a linear-logic programming language. In K. Apt, editor, *Joint International Conference and Symposium on Logic Programming*, pages 622–636. MIT Press, November 1992.

[66] Joshua Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation*. PhD thesis, University of Pennsylvania, 1994.

[67] Joshua Hodas. Lolli home page.
`http://www.cs.hmc.edu/~hodas/research/lolli/`, 1995.

[68] Joshua S. Hodas. Lolli: An extension of λprolog with linear context management. In D. Miller, editor, *Workshop on the λProlog Programming Language*, pages 159–168, Philadelphia, Pennsylvania, August 1992.

[69] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic (extended abstract). In *Proc. IEEE Symposium on Logic in Computer Science*, pages 32–42. IEEE, 1991.

[70] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 10(2):327–365, 1994.

[71] Joshua S. Hodas and Jeffrey Polakow. Forum as a logic programming language: Preliminary results and observations.
`http://www.cs.hmc.edu/~hodas/papers/`, 1996.

[72] Jonathan Hogg and Philip Wadler. Real world applications of functional programming. Available from `http://www.dcs.gla.ac.uk/fp/realworld/`, November 1996.

[73] H. Hosoya, N. Kobayashi, and A. Yonezawa. Partial evaluation for concurrent languages and its correctness. Technical Report To Appear., Department of Information Science, University of Tokyo, 1996.

[74] Paul Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[75] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, chapter 2, pages 17–42. Addison Wesley, 1990.

[76] Matthew Huntbach. An introduction to RGDC as a concurrent object-oriented language. *Journal of Object-Oriented Programming*, 8(5):29–37, September 1995.

[77] Matthew M. Huntbach and Graem A. Ringwood. Programming in concurrent logic languages. *IEEE Software*, pages 71–82, November 1995.

[78] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[79] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 111–119, January 1987.

[80] Sverker Janson. *AKL: A Multiparadigm Programming Language*. PhD thesis, Department of Computer Science, Uppsala University, 1994.

[81] S. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[82] S. Kleene. *Mathematical Logic*. Wiley and Sons, 1968.

[83] S.C. Kleene. Permutability of inferences in Gentzen's calculi LK and LJ. *Memoirs of the American Mathematical Society*, 10:1–26, 1952.

[84] N. Kobayashi, M. Nakade, and A. Yonezawa. Static analysis on communication for asynchronous concurrent programming languages. Technical Report 95-04, Department of Information Science, University of Tokyo, April 1995.

[85] Naoki Kobayashi and Akinori Yonezawa. ACL – a concurrent linear logic programming paradigm. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 279–294, Vancouver, Canada, 1993. The MIT Press.

[86] Naoki Kobayashi and Akinori Yonezawa. Logical, testing, and observation equivalence for processes in a linear logic programming. Technical Report 93-4, Department of Information Science, University of Tokyo, 1993.

[87] Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In Carrie Wilpolt, editor, *Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 31–45. ACM, October 1994.

[88] Naoki Kobayashi and Akinori Yonezawa. Typed higher-order concurrent linear logic programming. Technical Report 12, University of Tokyo, 1994.

[89] Alexei P. Kopylov. Decidability of linear affine logic. In D. Kozen, editor, *Tenth IEEE Symposium on Logic in Computer Science*, pages 496–504, June 1995.

[90] Robert Kowalski and Merek Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[91] Yves Lafont. Introduction to linear logic. Lecture Notes for the Summer School in Constructive Logics and Category Theory, August 1988.

[92] Yves Lafont. Functional programming and linear logic. Lecture Notes for the Summer School on Functional Programming and Constructive Logic. Glasgow., September 1989.

[93] Patrick Lincoln, Andre Scedrov, and Natarajan Shankar. Linearizing intuitionistic implication. *Annals of Pure and Applied Logic*, 60:151–177, 1993.

[94] Patrick D. Lincoln. *Computational Aspects of Linear Logic*. PhD thesis, Stanford University, August 1992.

[95] P.D. Lincoln and N. Shankar. Proof search in first-order linear logic and other cut-free sequent calculi. In *Ninth IEEE Symposium on Logic in Computer Science*, pages 282–291, 1994.

[96] J. W. Lloyd. Combining functional and logic programming languages. In M. Bruynooghe, editor, *Proceedings of the 1994 International Logic Programming Symposium*, pages 43–57. MIT Press, 1994.

[97] J. W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 1994.

[98] Ian Mackie. Lilac: A functional programming language based on linear logic. Master's thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1991.

[99] Ian Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, October 1994.

[100] Dave Mason. Applications of functional programming. Available from `http://www.scs.ryerson.ca/dmason/common/functional.html`, June 1997.

[101] M. Masseron. Generating plans in linear logic II: A geometry of conjunctive actions. *Theoretical Computer Science*, 113:371–375, 1993.

[102] M. Masseron, C. Tollu, and J. Vauzeilles. Generating plans in linear logic. In K.V. Nori and C.E. Veni Madhavan, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 63–75. Springer-Verlag, LNCS 472, December 1990.

[103] M. Masseron, C. Tollu, and J. Vauzeilles. Generating plans in linear logic I: Actions as proofs. *Theoretical Computer Science*, 113:349–371, 1993.

[104] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus: Preliminary report. *Electronic Notes in Theoretical Computer Science*, 3, 1996.

[105] D. Miler, G. Nadathur, and A. Scedrov. Hedreditary Harrop formulas and uniform proof systems. In *Proceedings of the Second Annual Conference on Logics in Computer Science*, pages 98–105, June 1987.

[106] D. Miller. A multiple-conclusion meta-logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[107] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, pages 79–108, 1989.

[108] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the Workshop on Extensions of Logic Programming*, pages 242–265. Springer-Verlag LNCS 660, 1992.

[109] Dale Miller. A multiple-conclusion meta-logic. In *Logic in Computer Science*, pages 272–281, 1994.

[110] Dale Miller. A survey of linear logic programming. *Computational Logic: The Newsletter of the European Network in Computational Logic*, 2(2):63–67, December 1995.

[111] Dale Miller. The forum specification language. `http://www.cis.upenn.edu/~dale/forum/`, 1996.

[112] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[113] Alberto Momigliano. Theorem proving via uniform proofs. Manuscript, 1993.

[114] Gopalan Nadathur. Uniform provability in classical logic. Technical Report TR-96-09, University of Chicago, 1996.

[115] Gopalan Nadathur and Donald W. Loveland. Uniform proofs and disjunctive logic programming (extended abstract). In *Tenth Annual Symposium on Logic in Computer Science*, pages 148–155, July 1995.

[116] Lee Naish. Automating control of logic programs. *Journal of Logic Programming*, 2(3):167–183, October 1985.

[117] Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[118] Mehmet A. Orgun and Wanli Ma. An overview of temporal and modal logic programming. In D. M. Gabbay and H. J. Ohlbach, editors, *First International Conference on Temporal Logic*, pages 445–479. Springer-Verlag LNAI 827, July 1994.

[119] M. Persson, K. Oedling, and D. Eriksson. Switching software architecture prototype using real time declarative language. In *International Switching Symposium*, October 1992.

[120] J.L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, September 1977.

[121] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[122] David Pym and James Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, April 1994.

[123] A. Scedrov. A brief guide to linear logic. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 377–394. World Scientific Publishing Co., 1993.

[124] A. Scedrov. Linear logic and computation: A survey. In H. Schwichtenberg, editor, *Proof and Computation, Proceedings Marktoberdorf Summer School 1993*, pages 379–395. NATO Advanced Science Institutes, Series F, Springer-Verlag, Berlin, 1995.

[125] Ehud Shapiro, editor. *Concurrent Prolog*. MIT Press, 1987.

[126] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[127] Ehud Y. Shapiro. *Algorithmic program debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[128] Toshihiro Shimizu and Naoki Kobayashi. HACL Version 0.1 user's manual. FTP with HACL release `camille.is.s.u-tokyo.ac.jp:pub/hacl`, June 1994.

[129] Duncan C. Sinclair. Solid modelling in Haskell. In *Glasgow functional programming workshop*, pages 246–263. Springer-Verlag, 1990.

[130] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In Schneider and Ehrig, editors, *Proceedings of Graph Transformations in Computer Science*, pages 358–379. Springer-Verlag, LNCS 776, 1994.

[131] Zoltan Somogyi. Stability of logic programs: how to connect don't-know nondeterministic logic programs to the outside world. Technical Report 87/11, Department of Computer Science, Melbourne University, September 1987.

[132] Zoltan Somogyi, Fergus Henderson, Thomas Conway, Andrew Bromage, Tyson Dowd, David Jeffery, Peter Ross, Peter Schachte, and Simon Taylor. Status of the Mercury system. In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 207–218, 1996.

[133] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and Richard O'Keefe. Logic programming for the real world. In Donald A. Smith, editor, *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming*, pages 83–94, Portland, Oregon, 1995.

[134] Leon Sterling and Ehud Shapiro. *The Art of Prolog (second edition)*. MIT Press, 1994.

[135] Tanel Tammet. Proof search strategies in linear logic. Programming Methodology Group 70, University of Göteborg and Chalmers University of Technology, March 1993.

[136] Tanel Tammet. Completeness of resolution for definite answers. Programming Methodology Group 79, University of Göteborg and Chalmers University of Technology, April 1994.

[137] Naoyuki Tamura and Yukio Kaneda. Resource management method for a compiler system of a linear logic programming language. In Michael Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, page 545. MIT Press, 1996.

[138] Naoyuki Tamura and Yuko Kaneda. Extension of WAM for a linear logic programming language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Second Fuji International Workshop on Functional and Logic Programming*, pages 33–50. World Scientific, November 1996.

[139] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask, and J. Harland. The Aditi deductive database system. *VLDB Journal*, 3(2):245–288, April 1994.

[140] Paolo Volpe. Concurrent logic programming as uniform linear proofs. In Giorgio Levi and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming*, pages 133–149. Springer-Verlag LNCS 850, September 1994.

[141] Philip Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990.

[142] Philip Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[143] David Wakeling and Colin Runciman. Linearity and laziness. In *Functional Programming and Computer Architecture*, pages 215–240. Springer-Verlag LNCS 523, 1991.

[144] David S. Warren. Programming the PTQ grammar in XSB. In Raghu Ramakrishna, editor, *Applications of Logic Databases*, pages 217–234. Kluwer Academic, 1994.

[145] G. White. The design of a situation-based Lygon metainterpreter: I. Simple changes and persistence. Technical Report 729, Department of Computer Science, Queen Mary and Westfield College, University of London, October 1996.

[146] Michael Winikoff. Hitch hiker's guide to Lygon 0.7. Technical Report 96/36, Melbourne University, October 1996.

[147] Michael Winikoff. Lygon home page. `http://www.cs.mu.oz.au/~winikoff/lygon`, 1996.

[148] Michael Winikoff and James Harland. Deterministic resource management for the linear logic programming language Lygon. Technical Report 94/23, Melbourne University, 1994.

[149] Michael Winikoff and James Harland. Implementation and development issues for the linear logic programming language Lygon. In *Australasian Computer Science Conference*, pages 563–572, February 1995.

[150] Michael Winikoff and James Harland. Implementing the linear logic programming language Lygon. In John Lloyd, editor, *International Logic Programming Symposium*, pages 66–80, Portland, Oregon, December 1995. MIT Press.

[151] Michael Winikoff and James Harland. Deriving logic programming languages. Technical Report 95/26, Melbourne University, 1996.

[152] Michael Winikoff and James Harland. Some applications of the linear logic programing language Lygon. In Kotagiri Ramamohanarao, editor, *Australasian Computer Science Conference*, pages 262–271, February 1996.

[153] Yi Xiao Xu. Debugging environment design for the logic programming language Lygon. Master's thesis, Department of Computer Science, Royal Melbourne Institute of Technology, February 1995.

[154] Helinna Yiu. Programming constructs for Lygon. Master's thesis, Department of Computer Science, Royal Melbourne Institute of Technology, 1997.

Appendix A

Proofs for Chapter 3

A.1 Proofs for Section 3.3

PROPOSITION 2

$B \rightsquigarrow A$.

Proof: *For B to be stronger than A means that for any program and for any goal, the existence of a proof satisfying criterion B implies that there must be a proof satisfying criterion A. In this case, the proof satisfying B has no sub-proof of the form $\Gamma \vdash$ which implies that it has no occurrence of the weakening right rule since in a single conclusioned setting, weakening right introduces a sequent of the above form. Contraction on the right is not relevant to a single conclusioned setting and hence the given proof also satisfies criterion A.* ■

PROPOSITION 3

$A \not\rightsquigarrow B$.

Proof: *There exist a goal and a program which have a proof satisfying criterion A but which do not have a proof satisfying criterion B. Consider the proof:*

$$\dfrac{\dfrac{}{p \vdash p}}{p, \neg p \vdash}$$

This proof does not use the weakening right rule yet it has an unavoidable proof of a sequent of the form $\Gamma \vdash$ ■

PROPOSITION 4

$C \not\rightarrowtail A$.

Proof: *There exist a program and goal which can be proven in a way which satisfies criterion C but which cannot be proven in a way that satisfies criterion A. Consider the proof:*

$$\dfrac{\dfrac{\dfrac{\overline{q \vdash q}}{q, \neg q \vdash}}{q, \neg q \vdash p(t)}\, W-R}{q, \neg q \vdash \exists x p(x)}$$

■

PROPOSITION 5

$C \not\rightarrowtail B$.

Proof: $B \rightsquigarrow A$ *and* $C \not\rightarrowtail A$, *by transitivity we have that* $C \not\rightarrowtail B$. ■

PROPOSITION 6

$A \not\rightarrowtail C$, $B \not\rightarrowtail C$.

Proof: *The (only) proof of* $\exists x p(x) \vdash \exists x p(x)$ *satisfies both criteria* A *and* B *yet violates criterion* C. ■

PROPOSITION 7

$D_{strong} \rightsquigarrow D_{weak}$.

Proof: *Obvious from definition.* ■

PROPOSITION 8

$D_{weak} \rightsquigarrow C$.

Proof: *Consider a proof satisfying D_{weak} which violates C. We show that this proof can always be transformed into a proof satisfying C. A violation of C is an inference whose conclusion is of the form $\Gamma \vdash \exists x F$. Since the proof satisfies D_{weak} and has a non-atomic goal the sequent must be the conclusion of some right rule. This right rule can be either $\exists$ — in which case we are done — or a structural rule, specifically weakening (since contraction right is not applicable in a single conclusion setting). By applying the*

following transformation we obtain a proof which satisfies criterion C*.*

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash \\ \vdots\end{array}}{\Gamma \vdash \exists x F}\, W-R \qquad \Longrightarrow \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash \\ \vdots\end{array}}{\Gamma \vdash F[t/x]}\, W-R}{\Gamma \vdash \exists x F}\, \exists - R$$

■

PROPOSITION 9

$D_{strong} \rightsquigarrow C$.

Proof: *In a proof that satisfies* D_{strong}*, any inference of the form* $\Gamma \vdash \exists x F$ *must be the conclusion of an* $\exists$ *rule and hence criterion* C *is satisfied.* ■

PROPOSITION 10

$C \not\rightsquigarrow D$.

Proof: *The only possible proof of the sequent* $?p \vdash ?p$ *satisfies criterion* C *yet violates both criteria* D_{weak} *and* D_{strong}

$$\dfrac{\dfrac{\dfrac{}{p \vdash p}\, Ax}{p \vdash ?p}\, ?-R}{?p \vdash ?p}\, ?-L$$

■

PROPOSITION 11

$D_{weak} \not\rightsquigarrow D_{strong}$.

Proof: *The sequent* $p, p^{\perp} \vdash (p \otimes q)$ *is provable in affine logic (which allows weakening but not contraction), however the only proof possible, begins by applying the* W*-R rule. This proof satisfies criterion* D_{weak} *yet necessarily violates criterion* D_{strong}*. Hence,* $D_{weak} \not\rightsquigarrow D_{strong}$*.*

$$\dfrac{\dfrac{\dfrac{}{p \vdash p}}{p, p^{\perp} \vdash}\, W}{p, p^{\perp} \vdash p \otimes q} \qquad \dfrac{\dfrac{}{p \vdash p} \quad p^{\perp} \vdash q}{p, p^{\perp} \vdash p \otimes q}$$

■

PROPOSITION 12

$A \not\rightarrow D$, $B \not\rightarrow D$.

Proof: *The proof of* $\exists x F \vdash \exists x F$ *necessarily violates criterion* D *yet satisfies both criteria* A *and* B. ■

PROPOSITION 13

$D \not\rightarrow A$.

Proof: *The sequent* $q, \neg q \vdash \exists x p(x)$ *can be proven in a way which satisfies criterion* D *but it cannot be proven without violating* A. *The proof involves moving* $\neg q$ *across to the succedent. In order to be able to apply the* $\neg - L$ *rule we need an empty succedent. Since the goal is not of the form* $\neg F$ *we can only obtain an empty succedent through an application of* $W - R$ *which violates criterion* A. ■

PROPOSITION 14

$D \not\rightarrow B$.

Proof: *The following proof is uniform according to* D *but it necessarily involves a successful proof of a sequent of the form* $\Gamma \vdash$

$$\frac{\overline{\bot \vdash}}{\bot \vdash \bot}$$

■

This hinges on the use of the linear logic connective $\bot$ as an explicit request to check whether the program is a tautology.

PROPOSITION 15

$F \rightsquigarrow D_{strong} \rightsquigarrow D_{weak}$.

Proof: *obvious from definition.* ■

PROPOSITION 16

$D \not\rightarrow E$.

Proof: *Consider the example given for criterion* E. *It satisfies criterion* D *but not criterion* E. ■

PROPOSITION 17

$F \rightsquigarrow C$.

Proof: *By transitivity from* $F \rightsquigarrow D$ *and* $D \rightsquigarrow C$. ■

PROPOSITION 18

$C \not\rightarrow F$.

Proof: *By transitivity from* $F \rightsquigarrow D$ *and* $C \not\rightarrow D$. ■

PROPOSITION 19

$A \not\rightarrow F$.

Proof: *By transitivity from* $F \rightsquigarrow D$ *and* $A \not\rightarrow D$. ■

PROPOSITION 20

$F \rightsquigarrow A$.

Proof: *Consider a sequent* $\Gamma \vdash G$ *occurring in a proof which satisfies criterion* F. *There are two cases:*

1. G *is non-atomic: the rule used is the rule which introduces its topmost connective. Hence the rule used is not weakening-right.*

2. G *is atomic: the rule used is a left or axiom rule and hence is not weakening-right.*

Hence the proof satisfies criterion A. ■

PROPOSITION 21

$F \not\rightarrow B$.

Proof: *The only proof of the sequent* $\bot \parr p \vdash p$ *is*

$$\dfrac{\dfrac{}{\bot \vdash}\,\bot - L \quad \dfrac{}{p \vdash p}\,Ax}{\bot \parr p \vdash p}\,\parr - L$$

Observe that the proof satisfies criterion F*; however it involves a proof of a sequent of the form* $\Gamma \vdash$ *and hence violates criterion* B. ■

PROPOSITION 22

$B \not\rightarrow F$.

Proof: *Follows from* $B \not\rightarrow D$ *and* $F \rightsquigarrow D$. ■

PROPOSITION 23

$E \not\rightarrow C$.

Proof: *The following proof satisfies criterion E yet necessarily violates criterion C.*

$$\dfrac{\dfrac{\vdots}{F[y/x] \vdash F[y/x]}}{\dfrac{F[y/x] \vdash \exists x F}{\exists x F \vdash \exists x F}}$$

■

COROLLARY:$E \not\rightarrowtail F$, $E \not\rightarrowtail D$ *(since $F \rightsquigarrow C$ and $D \rightsquigarrow C$)*

PROPOSITION 24

$E \not\rightarrowtail B$.

Proof: *Follows by transitivity from $F \not\rightarrowtail B$ and $F \rightsquigarrow E$.* ■

PROPOSITION 25

$E \not\rightarrowtail A$.

Proof: *The following proof satisfies criterion E yet necessarily violates criterion A.*

$$\dfrac{\dfrac{\overline{p \vdash p}}{p, \neg p \vdash}}{p, \neg p \vdash \exists x q}$$

■

PROPOSITION 26

$B \not\rightarrowtail E$.

Proof: *The proof of the sequent*

$$r \multimap (q \multimap p), q \otimes r \vdash p$$

satisfies criterion B but necessarily violates criterion E since we need to decompose one program formula before using another program formula and hence we cannot "focus" on a single program formula D.

$$\dfrac{\dfrac{\dfrac{\overline{r \vdash r} \quad \dfrac{\overline{q \vdash q} \quad \overline{p \vdash p}}{q \multimap p, q \vdash p} \multimap -L}{r \multimap (q \multimap p), q, r \vdash p} \multimap -L}{r \multimap (q \multimap p), q \otimes r \vdash p} \otimes -L$$

■

COROLLARY:$A \not\rightarrowtail E$ *(since $B \rightsquigarrow A$)*

LEMMA 27

Let $\Gamma \vdash F$ be a provable sequent where Γ and F are in a logic subset which satisfies criterion D_{strong}. Then all sequents occurring in the proof are in the logic subset satisfying criterion D_{strong}.

Proof: *There are two cases: F can be either an atom or a compound formula. If F is a compound formula then we know that there is a proof of the form*

$$\dfrac{\begin{array}{c}\vdots\\ \Pi \\ \vdots\end{array}}{\Gamma' \vdash F'} * - R$$

Consider now the sequent $\Gamma' \vdash F'$. It has a proof which satisfies criterion D_{strong} — namely Π. Hence $\Gamma' \vdash F'$ satisfies D_{strong}. If F is atomic then precisely the same situation entails except that $* - R$ is replaced by $* - L$. We can apply induction upwards and conclude that all sequents occurring in the proof satisfy criterion D_{strong}. ■

LEMMA 28

Let $\Gamma \vdash F$ be a provable sequent where Γ and F are in a logic subset which satisfies criterion E. Then all sequents occurring in the proof are in the logic subset satisfying criterion E.

Proof: *The proof is analogous to that of the previous lemma. The main difference is that a compound goal could be the conclusion of either a left or a right rule.* ■

PROPOSITION 29

Let $\Gamma \vdash \Delta$ be a sequent in a logic subset which satisfies criteria E and D_{strong} and such that $\Gamma \vdash \Delta$ is provable. Then there exists a proof of $\Gamma \vdash \Delta$ which satisfies criterion F.

Proof: *We apply induction from the root of the proof to the leaves. The base case is an inference with no premises. If the goal is atomic then since criterion E is satisfied the inference must be an Ax rule. If the goal is compound then since criterion D_{strong} is satisfied the rule must be a right logical rule. In either case the inference satisfies criterion F. For the induction step consider a sequent at the root of a (sub)-proof. By the induction hypothesis the sequent satisfies criteria E and D_{strong}. There are two cases:*

1. *The goal is compound: Since criterion D_{strong} is satisfied there is a proof which begins with a right rule where the premise(s) of the rule satisfy criterion D_{strong}.*

It is not immediately obvious that the premise(s) must also satisfy criterion E — lemma 28 guarantees that there exists a proof of the sequent, but the proof may not necessarily begin with a right rule. We can however transform a proof satisfying criterion E which begins with a left rule into one beginning with a right rule by permuting the appropriate right rule down. The right rule is known to permute with all left rules since criterion D_{strong} is satisfied.

2. *The goal is atomic: Since criterion E is satisfied there is a left-focused proof of the sequent. Furthermore the premise(s) of the first inference satisfies both criterion E (by lemma 28) and criterion D_{strong} (by lemma 27).*

In either case the appropriate condition for criterion F is met. ■

A.2 Proofs for Section 3.5

PROPOSITION 43

$B \not\rightarrowtail A$.

Proof: *The following classical logic proof of the sequent $p(a) \vee p(b) \vdash \exists x p(x)$ satisfies criterion B but involves an application of contraction-right.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p(a) \vdash p(a)}\,Ax}{p(a) \vdash \exists x p(x)}\,\exists - R \quad \dfrac{\dfrac{}{p(b) \vdash p(b)}\,Ax}{p(b) \vdash \exists x p(x)}\,\exists - R}{p(a) \vee p(b) \vdash \exists x p(x), \exists x p(x)}\,\vee - R}{p(a) \vee p(b) \vdash \exists x p(x)}\,C - R$$

■

Note that the proof used the multiplicative presentation of conjunction

$$\dfrac{\Gamma, F_1 \vdash \Delta \quad \Gamma', F_2 \vdash \Delta'}{\Gamma, \Gamma', F_1 \vee F_2 \vdash \Delta, \Delta'}\,\vee - L$$

since the standard additive rule encodes applications of contraction.

PROPOSITION 44

$A \not\rightarrowtail B$.

Proof: *There exists a sequent which has a proof satisfying A but which does not have a*

proof satisfying B. Consider the proof

$$\frac{\overline{p \vdash p}}{p, \neg p \vdash}$$

This proof does not use weakening or contraction but it has an unavoidable proof of a sequent of the form $\Gamma \vdash$. ■

PROPOSITION 45

$C_{strong} \rightsquigarrow C$.

Proof: *Obvious from definition.* ■

PROPOSITION 46

$C \not\rightarrow A$, $C_{strong} \not\rightarrow A$.

Proof: *There exist a program and goal which can be proven in a way which satisfies criterion C (and C_{strong}) but which cannot be proven without violating criterion A. Consider the sequent $q, \neg q \vdash \exists x p(x)$. In order to apply the axiom rule we need to eliminate the formula $\exists x p(x)$. This can be only be done using a weakening right rule which violates criterion A. It is possible to prove this sequent in a way which satisfies criterion C:*

$$\frac{\dfrac{\dfrac{\overline{q \vdash q}}{q \vdash q, p(t)}\, W-R}{q \vdash q, \exists x p(x)}\, \exists -R}{q, \neg q \vdash \exists x p(x)}\, \neg -L$$

■

PROPOSITION 47

$C \not\rightarrow B$, $C_{strong} \not\rightarrow B$.

Proof: *Consider the sequent $\bot \vdash \exists x(?p(x))$. It is provable and there is only a single possible proof — we need to eliminate the goal formula so we can apply the rule $\bot - L$. The elimination is done in a way which satisfies criteria C and C_{strong} but the proof involves a subproof of $\bot \vdash$ which violates criterion B.*

$$\frac{\dfrac{\dfrac{}{\bot \vdash}\, \bot -L}{\bot \vdash ?p(t)}\, W?-R}{\bot \vdash \exists x(?p(x))}\, \exists -R$$

■

PROPOSITION 48

$A \not\rightarrowtail C$, $B \not\rightarrowtail C$, $A \not\rightarrowtail C_{strong}$, $B \not\rightarrowtail C_{strong}$.

Proof: *The sequent $\exists x p(x) \vdash \exists x p(x)$ can be proven in precisely one way. This proof satisfies both criteria A and B and violates criteria C and C_{strong}.*

$$\dfrac{\dfrac{\overline{p(Y) \vdash p(Y)}}{p(Y) \vdash \exists x p(x)} \exists - R}{\exists x p(x) \vdash \exists x p(x)} \exists - L$$

■

PROPOSITION 49

$B \not\rightarrowtail D$, $A \not\rightarrowtail D$.

Proof: *Consider the proof of $\exists x F \vdash \exists x F$.* ■

PROPOSITION 50

$D_A \rightsquigarrow A$.

Proof: *There are two cases. If the goal contains a compound formula then the rule used to infer the sequent must have been a right rule which introduces a topmost connective and hence cannot have been a right structural rule. If there are no compound goals then the sequent must be the conclusion of either an Ax or left rule and hence cannot be a right structural rule.* ■

PROPOSITION 51

$D_S \rightsquigarrow A$.

Proof: *There are two cases. If there are no atoms in the goal then the rule used to infer the sequent must have been a right rule which introduces a topmost connective and hence can not have been a right structural rule. Otherwise the sequent must be either the conclusion of a right rule as above or the conclusion of Ax or a left rule and in neither case can the rule be a right structural rule.* ■

PROPOSITION 52

$D \rightsquigarrow C$.

Proof: *Consider a sequent of the form $\Gamma \vdash \exists x F$. Since it has only one compound conclusion and no atomic goals, it must be the result of an $\exists - R$ rule and hence it satisfies C.* ■

PROPOSITION 53

$D_A \rightsquigarrow C_{strong}$.

Proof: *Consider a sequent of the form* $\Gamma \vdash \exists x F, \Delta$. *According to* D_A *there exists a proof where this sequent is the conclusion of an* $\exists$*-R rule. This proof satisfies* C_{strong}. ■

PROPOSITION 54

$D_S \not\rightsquigarrow C_{strong}$.

Proof: *Consider the proof of the sequent* $\vdash \exists x p(x)^{\perp}, p(1) \,\&\, p(2)$. *This proof satisfies* D_S *but not* C_{strong}. ■

PROPOSITION 55

$D \not\rightsquigarrow B$.

Proof: *The following proof is uniform according to criterion* D *but it necessarily involves a successful proof of a sequent of the form* $\Gamma \vdash$.

$$\dfrac{\dfrac{}{\perp \vdash}}{\perp \vdash \perp}$$

■

PROPOSITION 56

$C \not\rightsquigarrow D$, $C_{strong} \not\rightsquigarrow D$.

Proof: *Consider the proof of the sequent* $a \oplus b \vdash a \oplus b$. ■

PROPOSITION 57

$D_S \not\rightsquigarrow D_A$.

Proof: *Follows by transitivity from the facts that* $D_S \not\rightsquigarrow C_{strong}$ *and* $D_A \rightsquigarrow C_{strong}$. ■

PROPOSITION 58

$D_A \rightsquigarrow D_S$.

Proof: *There are four cases corresponding to the four types of sequents that can occur in a proof:*

> $\Gamma \vdash \mathcal{C}$*:* D_A *guarantees that the sequent is the conclusion of a right rule which introduces the topmost connective of a formula* $F \in \mathcal{C}$. *This satisfies* D_S.
>
> $\Gamma \vdash \mathcal{A}$*:* D_A *guarantees that the proof of the sequent is left-focused which satisfies* D_S.

$\Gamma \vdash \mathcal{A}, \mathcal{C}$: *$D_S$ requires that the proof be either left-focused or the conclusion of a right rule. D_A guarantees the latter.*

$\Gamma \vdash$: *Neither D_A nor D_S have any restrictions on this case.*

■

PROPOSITION 59

Let D_{S+} be a criterion that modifies D_S by requiring that sequents of the form $\Gamma \vdash \mathcal{C}$ have a proof which introduces the topmost connective of F for all *$F \in \mathcal{C}$. Then D_{S+} is equivalent to D_A.*

Proof: *Observe that for sequents which are of the form $\Gamma \vdash$ or $\Gamma \vdash \mathcal{A}$ the two definitions (and indeed D_S as well) agree on the constraints to be imposed. For sequents of the form $\Gamma \vdash \mathcal{C}$ the requirement that right connectives permute gives agreement between D_A and D_{S+}.*

Consider now sequents which are of the form $\Gamma \vdash \mathcal{A}, \mathcal{C}$. This sequent satisfies D_A if we can apply a right rule without a loss of completeness. The only way in which D_A and D_{S+} can disagree is if there is a sequent of this form where we need to apply a left-focused proof step before we can apply a right rule. In order for this to occur we need to have an impermutability. Since there cannot be an impermutability between right connectives (by the definition of D_{S+}) the impermutability must be between a right connective and a left connective. This situation however violates D_S since we can use the given impermutability to construct a sequent of the form $\Gamma \vdash \mathcal{C}$ where the impermutability prevents a proof which begins with a right rule. ■