Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation

Joshua S. Hodas

A DISSERTATION

IN

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

1994

Dale Miller
Supervisor of Dissertation

Mark Steedman
Graduate Group Chairperson

In memory of Simon Maurice Plotnick
and John Genereaux
Who, each in his own way,
taught me that numbers were fun.

Acknowledgements

First,the formalities: The author has been funded by ONR N00014-88-K-0633, NSF CCR-87-05596, NSF CCR-91-02753, and DARPA N00014-85-K-0018 through the University of Pennsylvania.

Now that that's out of the way, down to business.

First, I must thank Dale Miller, my advisor and collaborator for the last several years, without whom this endeavor would never have begun; for planting so many fruitful ideas, and helping me see them through, thank you.

The material in this dissertation spans several areas of computer science, logic, and computational linguistics —many of which I had only limited knowledge of at the start of this work. Therefore, there are many experts in each area to whom I am indebted for many helpful conversations: to Jean-Yves Girard, Andre Scedrov, Jean Gallier, Jim Lipton, Serenella Cerrito, Pat Lincoln, James Harland, Jawahar Chirimar, Roy Dyckhoff, Max Kanovich, and Gianluigi Bellin for discussions on linear logic, proof theory, complexity, and other topics; to Fernando Pereira, Stu Shieber, Mark Steedman, Mitch Marcus, Mark Johnson, Dick Oehrle, Bob Frank, Mark Hepple, and Elizabeth Hodas for discussions about parsing techniques for filler-gap dependencies; to Carl Gunter, Jon Riecke, Phil Wadler, and Paul Hudak for discussions about linear functional languages; and to Remo Pareschi for conversations about nearly every aspect of this work at one time or another.

Finally, to my family: Sheila, Morton and Ana, Steven and Catherine, Rachel and Jonathan; for years of support, thank you. To Elizabeth, my wife, for all you have done, for all you have been, thank you is not even close to enough.

Abstract

Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation

Author: Joshua S. Hodas
Supervisor: Dale Miller

It is an unfortunate reality that in "logic programming" as exemplified by Prolog few interesting problems can be solved using the purely logical fragment of the language. The programmer must instead resort to the use of extra-logical features about which sound reasoning is difficult or impossible. A great deal of work in the theory of logic programming languages in the last decade has centered on the question of whether, by basing languages on richer logical systems than Horn clauses, we might obtain a system in which the logical core of the system covers a greater percentage of the problems programmers face.

When logic programming is based on the proof theory of intuitionistic logic, it is natural to allow implications in goals and in the bodies of clauses. Attempting to prove a goal $D \supset G$ causes the system to add the assumption D to the current program and then attempt to prove G. This sort of addition to the language has been studied in depth by Miller, Nadathur, Gabbay, and others and has been shown to yield many useful features at the language level, including notions of hypothetical reasoning in databases, and a logic-based module system for logic programming.

There are many problems, though, which cannot be given an attractive treatment in systems based on classical or intuitionistic logic because, in those settings, no control is possible over whether, and how often, a formula is used during a proof. These are known as the relevant and affine constraints. This limitation can be addressed by using a fragment of Girard's linear logic to refine the intuitionistic notion of context.

In this thesis I summarize the theory and design of a logic programming language based on linear logic as first proposed by Hodas and Miller in 1991. Several example applications are shown to justify the language design. It is also shown how this system can be seen as specifying a new family of logic grammars for natural language processing which yield perspicuous and natural grammars for handling filler-gap dependencies.

Several aspects of the implementation of the language are discussed. In particular it is shown how issues that do not arise in implementing traditional systems can prove computationally crippling to a naive implementation of this logic. Therefore a formal operational semantics is proposed which circumvents these problems by delaying, as much as possible, otherwise non-deterministic choices in the proof search.

Finally, a further refinement of the system, in which the programmer can specify the relevant and affine constraints independently, thereby obtaining greater control, is discussed.

Contents

List of Figures

Chapter 1

Introduction

Niklas Wirth has attempted to sum up the motivation for structured imperative programming languages in a well known equation:

$$\textit{Programs} = \textit{Data Structures} + \textit{Algorithms}^{1}$$

In contrast, the motivation for logic programming, both as a field overall, and as an individual endeavor, is summed up in part by what has become known as "Kowalski's equation":

$$\textit{Programs} = \textit{Logic} + \textit{Control}$$

which is intended to express the fact that a logic program consists of just a set of logical formulas defining relationships between the classes of data objects which the program is to compute with. Computation is simply a side effect of the attempt to prove that certain relationships hold. The *Control* part of the equation corresponds to a small amount of extra-logical control of the proof search process which gives programs an operational as well as logical reading.

The advantage expressed in Kowalski's Equation is that while we have only limited tools for reasoning about data structures and algorithms, logic is a comparatively well understood field. If programs can be expressed mostly as logic and proofs, then more than ninety years of proof theory can be brought to bear to reason about programs. This idea has shown fruit in the extensive work on partial evaluation and abstract interpretation of logic programs and, in particular, in the ease with which these techniques can be proven to be sound.

Unfortunately, reality is not as pleasant as Kowalski's equation implies. Prolog, the dominant logic programming language, is based on the logic of Horn clauses, a simple logic with relatively limited expressive power.[2] While Pure Prolog (the part based just on Horn clause

[1] This is actually a symmetric, commutative reformulation. The original equation was *Algorithms* + *Data Structures* = *Programs* [Wirth, 1976]

[2] We must be careful here. It is easy to show that first-order Horn clauses, the logical language of Prolog,

logic) can be used for many simple applications, serious programming generally requires language features not supported by the logic. Therefore, all Prolog implementations include some "extra-logical" features to augment the expressiveness of the underlying logic. The end result is that reality is better matched by what I will call "Miller's equation":

$$\begin{aligned} \textit{Real Programs} = \textit{Horn Clause Logic} &+ \textit{Control} \\ &+ \textit{Modules} \\ &+ \textit{Higher Order Features} \\ &+ \textit{Dynamic Predicates} \\ &+ \textit{Communication} \\ &+ \ldots \end{aligned}$$

The extra-logical features quickly overwhelm the logical core of the language, and their use makes it difficult or impossible to use logic to reason about logic programs.

This dissertation is the latest step in a course of research begun by Dale Miller in the mid-1980's with the development of λProlog a language that extends Prolog both at the term level — the term structure is taken from Church's simply typed λ-calculus— and at the formula level — a much larger class of formulas is allowed than in Prolog. The goal is to show that it is possible to find logics richer than Horn clauses which have the expressive power to directly support the features needed for real world programming. The hope is that it we might eventually reach a point where Miller's equation can be restated as something of the form:

$$\textit{Real Programs} = \textit{Higher Order Resource Logic} + \textit{Control}$$

While it is unlikely that we will ever find a logic rich enough to describe all the things programs must reason about, the goal is to make the logic part of the equation powerful enough so that it is no longer overwhelmed by the extra-logical parts.

1.1 Historical Perspective

As stated above, pure first-order Horn clauses are too weak to support many language features required for modern programming tasks. Up until this point most researchers interested in maintaining the declarative foundations of logic programming while supporting more

are Turing-complete [Tärnlund, 1977]. Thus when we speak of expressiveness, we are not talking of what could conceivably be programmed, but rather the ability of the language to support useful programming constructs.

powerful programming constructs have proposed languages based on richer fragments of intuitionistic first-order and higher-order logics.[3] This work has paralleled research in automated deduction systems and specification languages. For example, first-order and higher-order versions of hereditary Harrop formulas (formulas with no positive occurrences of disjunctions or existential quantifiers) have been used both as the basis of logic programming languages [Gabbay and Reyle, 1984; Hallnäs and Schroeder-Heister, 1990; McCarty, 1988; Miller, 1990; Miller *et al.*, 1991] and as specification languages for proof systems [Felty, 1989; Felty and Miller, 1988; Paulson, 1990; Pfenning, 1988].

In the case of first-order systems, much of the expressiveness of these extensions derives from allowing implications to occur in goals as well as in program clauses (which is the only place they are allowed in Horn clauses). In these languages, proving a goal $D \supset G$ causes the system to add the clauses in D to the current program (or proof context) and then attempt a proof of G. From a programming perspective this rule augments the otherwise flat language of Horn clauses with a useful notion of program scope: D can be seen as a module which is loaded with scope only over the call G.

Aside from providing a logical basis for module systems in logic programming, implications in goals have been exploited for a variety of uses:

- When using an intuitionistic meta-logic to design theorem provers, this extension makes it possible to use the meta-logic's proof context to store object-level hypotheses and eigen-variables. This lends itself to natural presentations of theorem provers for many object logics [Felty and Miller, 1988; Paulson, 1990].

- In computational linguistics, a relative clause, such as whom Anne married, is viewed as a relative pronoun followed by a sentence that is missing a noun phrase (Anne married *gap*). A proposed technique for parsing relative clauses is to first assume the existence of a noun phrase with no phonological structure (the *gap*) by using an implication to load a rule for this sort of noun phrase temporarily into the grammar, and then to attempt to parse a subordinate sentence [Pareschi, 1989; Pareschi and Miller, 1990].

- Logic programming is a natural setting for database applications. In these enriched systems, certain kinds of hypothetical reasoning ("if Sue passes CS121, will she be able to graduate?") are easy to model logically [Bonner *et al.*, 1989; Gabbay and Reyle, 1984; Miller, 1989*b*].

- A notion of state encapsulation, as used in object-oriented programming, can be approximated using these systems. An object's state can be represented by assumptions in a context [Hodas and Miller, 1990], and other logical features, such as embedded quantification, can be used to implement information hiding.

[3] Most logic programmers tend to think of Prolog as being based on classical logic. The next chapter will explain why intuitionistic logic has dominated the work in logical extensions of Prolog.

Thus these systems, which are based on an operational reading of implications in goals, provide a natural, logical treatment of the computing needs in a large number of settings. However, they are often too limiting. One problem is that once an item is placed into the context it cannot be removed (during the proof of the subordinate goal). A formula in a context can be used any number of times, or not at all. This creates problems in each of the applications described above:

- When designing theorem provers, it is not possible to specify any variations of the contraction rule: arbitrary contraction on all hypotheses is imposed by the meta-logic.
- When parsing relative clauses, there is no natural way to enforce the constraints that the assumed gap must actually be used while parsing the clause that follows (*whom Anne married John* is incorrectly accepted as a valid relative clause) and that the gap cannot be used to fill certain positions ("island constraints" [Ross, 1967; Pereira and Shieber, 1987]).
- When implementing a database system, there is no straightforward way to handle the updating and retracting of facts, and when solving hypothetical queries there is no way to require that the new assumption be relevant to the eventual conclusion.
- In object-oriented logic programming, updating the state of an object means changing the assumptions representing its state. The only change allowed with these systems is that of augmenting the state's representative assumptions. Thus as execution progresses an object's state becomes progressively more non-deterministic. This is not likely to be the intended interpretation of mutable state.

So, while these extensions provide a significant improvement over the logic of Horn clauses, they are still far too weak for many uses.

1.2 Contributions

In this dissertation I will discuss the use of a (mostly first-order) fragment of Girard's linear logic as the basis of a logic programming system. I will show how many applications, including those above, which require the use of extra-logical features in Prolog or λProlog can be directly implemented in this new setting. The system also provides novel solutions to problems that have no good solution in Prolog or λProlog. At the same time the system can be seen as a proper extension of pure Prolog. Prolog programs that do not make use of the new features will behave as expected by the programmer, and the new features are easy for the trained Prolog programmer to understand.

The core of this dissertation, which presents the theory and design of the proposed language, is based on joint work with Dale Miller carried out in 1990 and 1991. This work has been described in our paper in the 1991 Logic in Computer Science Conference [Hodas and Miller, 1991] and an extended version now in publication at the Journal of Information and

Computation [Hodas and Miller, 1994]. In those papers we showed how each of the problems described above can be addressed by adopting a more refined notion of proof context from Girard's linear logic. We presented a fragment of linear logic that makes a suitable logic programming language and permits very natural solutions to all of the above problems. We also described a new style of proof system (which we now refer to as an operational semantics) for the logic, which demonstrates a solution to a problem which arises from designing an interpreter for the language by naively implementing the standard linear logic sequent calculus. This new system is designed to track the resource consumption pattern of proof construction and is related to the system of *focusing proofs* developed independently by Andreoli [Andreoli, 1992].

This dissertation expands on [Hodas and Miller, 1991] and [Hodas and Miller, 1994], providing full proofs of most of the relevant theorems, and a set of considerably expanded examples. The principal new contributions of this dissertation include showing how even the "new" proof system of those papers has significant computational drawbacks, and how they can be fixed. The application of this work to the development of a new class of grammars for natural language processing is also discussed in depth. I also discuss a further refinement of the logical system which allows direct representation of several substructural logics within the system. This enables the programmer to work with exactly the set of constraints appropriate to a given problem.

The remainder of the dissertation describes details of a working implementation of the language which has been made available for researchers interested in this work. The system is called Lolli for the linear logic implication operator '$\multimap$' which looks like a lollipop. A complete guide to the language is provided.

1.3 Outline of the Dissertation

In Chapter 2 I introduce the logic of first-order positive Horn clauses which underlies Prolog. An important aspect of this introduction is the argument that it is better to consider logic programming as the search for proofs in intuitionistic, rather than classical, logic. I then introduce Miller, et al.'s definition of *uniform proof* [Miller, 1989*b*; Miller *et al.*, 1991] which is intended to formalize the notion of goal-directed search.

Chapter 3 presents hereditary Harrop formulas, which form an extension of Horn clauses and have served as the foundation of the language λProlog. After presenting the formal properties of the system, several examples are given which demonstrate how the richer formula language leads to a richer programming language. Much of this material has been presented by Miller and colleagues in a series of papers, though the discussion and examples of continuation-passing-style programming are new.

Chapter 4 introduces Girard's Linear Logic briefly, and then discusses the modifications that must be made to it to use it as the foundation of a logic programming language. It is shown that such a language can be seen as a proper extension of λProlog. A couple of small examples of the benefits of using this logic are then shown.

Chapter 5 presents a series of larger examples intended to demonstrate the utility of this new language. Several programming paradigms are discussed and demonstrated.

Chapter 6 presents an in-depth discussion of a particularly successful application of the system. It is shown that natural language parsers built in this system provide perspicuous solutions to several thorny problems. It is further posited that this is not just an implementation, but can be seen as specifying a new type of grammar formalism.

Chapter 7 discusses the dificulties that arise if language implementor approaches the problem of developing a working system naively from the logic presented in Chapter 4. A progression of systems of formal operational semantics is developed that address most of these problems.

Chapter 8 presents an extension of the logic and implementation of the main system which allows the programmer to work with not just linear constraints on the use of formulas, but also the weaker affine and relevance constraints. Examples are given demonstrating the use of this enriched system.

Chapter 9 discusses the details of a working prototype interpreter for the system implemented in the functional language ML. The actual implementation techniques are discussed briefly. After this there is an extensive presentation of the details of the language from the programmer's point of view, including a complete description of all the built in predicates of the system.

Chapter 10 describes the relationship between this work and several related systems proposed by researchers in the last few years.

Chapter 11 discusses future directions for this work.

Appendix *A* provides a brief intorduction to intuitionistic logic and the existential property as they pertain to the theory of logic programming.

Appendices *B* – *F* provide full proofs of some of the theorems from the core of the dissertation.

Appendix *G* gives the code for an extended version of the theorem proving example from Chapter 5.

Chapter 2

An Alternate View of the Foundations of Logic Programming

Ask any logic programmer what the proof system (or execution model) underlying Prolog is and ninety-nine percent will reply "resolution-refutation." This is a perfectly reasonable and truthful answer. But while it is historically true, it is unsatisfying. While Prolog's execution model, which is called SLD-resolution, is based on resolution-refutation, it is such a limited form of resolution-refutation that the relationship is tenuous at best. More important from the standpoint of this dissertation, it is very hard to see how to extend the SLD-resolution method to larger logics without losing the operational flavor of the language.

In this chapter I will introduce Horn clauses, the logic of pure Prolog. The proof theory of the system will be presented first in the traditional way —in terms of resolution refutation proofs— and then in the manner that will be used in the rest of this dissertation —in terms of the root-up search for proofs in a Gentzen-style sequent calculus. The latter approach will be justified by the ease with which it can be extended to richer logical systems.

I will assume that the reader has some familiarity with the basics of first-order logic and proof theory. Gallier covers all that one might need to know (and much more) [Gallier, 1986]. That work is particularly appropriate as an introduction to proof theory using Gentzen Sequent Calculi, which are used heavily here. For more on resolution there is Robinson's original paper introducing the technique [Robinson, 1965], as well as Lloyd's book on the foundations of logic programming [Lloyd, 1984].

2.1 Positive Horn Clauses

Given standard definitions of first-order terms and atomic formulae, we define the following two classes of formulae:[1]

Definition 2.1 A *goal formula* is a formula of the form:

$$\exists x_1. \cdots \exists x_n.(A_1 \wedge \cdots \wedge A_m)$$

where $m, n \geq 0$, $A_1, \ldots, A_m$ are atomic formulae, and $\{x_1, \ldots, x_n\}$ is the set of all of the free variables of $A_1 \wedge \cdots \wedge A_m$.

Definition 2.2 A *definite formula* or *positive Horn clause* is a formula of the form:

$$\forall x_1. \cdots \forall x_n.[(A_1 \wedge \cdots \wedge A_m) \supset A]$$

where $m, n \geq 0$, A and $A_1, \ldots, A_m$ are atomic formulae, and $\{x_1, \ldots, x_n\}$ is the set of all of the free variables of $A_1 \wedge \cdots \wedge A_m \supset A$. We call A the *head* of the clause and $A_1 \wedge \cdots \wedge A_m$ the *body*.

A pure Prolog program is just a finite set of Horn clauses. Each clause asserts either the truth of an atomic fact —if $m = 0$ then the clause asserts that A is true for all values of its free variables— or of a rule —A is true if $A_1 \wedge \cdots \wedge A_m$ is true. Because clauses are best read in this way, as giving definitions for their heads, they will generally be written head first using a reverse-implication connective (and without unnecessary extra parentheses), as in:

$$\forall x_1. \cdots \forall x_n.A \subset A_1 \wedge \cdots \wedge A_m$$

In general, the definition of Horn clauses allows for the use of disjunctions in negative positions (i.e. in goals, and in the bodies of clauses). [2] However, it is a theorem that every Horn clause with disjunctions can be converted (by distributivity) to an equivalent set of clauses without disjunctions. For instance, the clause $p \subset q \wedge (r \vee s)$ is equivalent to the set of clauses $\{p \subset q \wedge r, p \subset q \wedge s\}$. Therefore, at this point we will follow the convention of considering only disjunction-free clauses.

Throughout the rest of this dissertation we will use the following calligraphic conventions:

$a, \ldots, e$ will be used for individual constants.

f, g, h will be used for function names.

p, q, r will be used for predicate names.

[1] For the purposes of this and future definitions, $\top$ will be considered a zero-ary logical connective, not an atomic formula. As we will not be dealing with the question of negation, $\bot$ will not be needed.

[2] An occurrence of an expression at the top level on the right of the sequent arrow is negative. Each time it crosses to the left of either the sequent arrow or an implication, it changes polarity.

s, t, u will be used for arbitrary (generally first-order) terms.

x, y, z will be used for variable names.[3]

A will be used for atomic formulae only.

$B, C, \ldots$ will be used for arbitrary first-order formulae. In most instances G will be reserved for goal formulae (or parts of goal formulae) and D for definite formulae.

$\Gamma, \Delta, \Upsilon, \Psi$ will be used for multisets of formulae.

Θ will be used to represent a substitution/unifier.

Ξ will represent a proof or subproof.

Swash letters, ($\mathcal{A}, \mathcal{B}, \ldots$), will be used for the designation of particular proof systems.

Hand lettering, (P, D, G), will, on occasion, be used to stand for particular programs and queries, or for the families of formulae that make up programs and queries.

Set brackets, "$\{\ \}$", will be used to denote sets and multisets; there should never be any confusion of which is intended.

Angle brackets, "$\langle\ \rangle$", will be used to denote sequences (that is, ordered multisets). These will be used in some cases where a multiset might be expected, but in which the order of elements is significant, as when considering actual Prolog programs. They will also be used for tuples.

Square brackets, "[ ]", will be used interchangeably with parentheses in formulae, as needed, to help clarify scope.

In sequent calculus proofs (see below) that represent program executions, the use of unification will be represented by showing variables with their original names throughout the proof tree, with the value instantiated by unification shown in a shadow-box at the point in the tree that the unification inducing the instantiation occurs. In addition, the multiset Γ will be assumed to be the initial form of whatever program is under consideration at the time. As the proof context grows during a proof, it will often be helpful (from a typesetting standpoint) to give a new name to the augmented context. This will be done by showing the new name above an overbrace covering the context that the name is intended to represent.

2.2 The SLD-resolution Computation Model

Given a program P we wish to know whether goal G follows from that program. The SLD-resolution algorithm produces a proof that G follows from P as follows:

[3] Note that the conventions for $a \ldots z$ will be relaxed in proofs representing the execution of actual programs. The names of constants, variables, predicates, and functions chosen in those cases will just be the corresponding names (or fragments thereof) from the program under consideration. No confusion should arise from this dropping of convention.

1. Let the *current substitution*, Θ, be the empty substitution and,
   if $G = \exists x_1. \cdots \exists x_n. A_1 \wedge \cdots \wedge A_m$, let the *current goal set* be the set $\{A_1, \cdots, A_m\}$.

2. If the current goal set is empty, then G follows from P. Halt with *computed answer substitution* Θ.

3. Select a member A' of the current goal set, and some clause from P with head A such that A' unifies with A with *most general unifier* Θ'. If there is no pair of goals and rules such that the unification can succeed, then this attempt to show that G follows from P has failed. Note that each time a rule is selected from P all the variables in the rule must be renamed to new variables so that they do not collide with existing variables.

4. Let the current substitution be Θ composed with Θ'. Delete A' from the current goal set and, assuming the body of the selected rule is $A'_1 \wedge \cdots \wedge A'_n$, replace A with $\{\Theta'(A'_1), \cdots, \Theta'(A'_1)\}$.

5. Return to step 2.

Definition 2.3 The process of replacing a goal atom with the body of a rule defining that atom, as occurs in steps 3 and 4, is called *backchaining*.

A complete SLD proof corresponds to only an aspect of a particular Prolog computation. In this model, Prolog execution corresponds to the *search* for an SLD proof. At each step in the search for a proof, Prolog selects the first subgoal and the first rule (in the order the programmer types them in the program) whose head matches that subgoal. If the body of that rule cannot be proved —that is, if the choice eventually leads to a dead-end in step 2— then Prolog *backtracks* to this point and selects the next matching rule instead. It is easiest to think of Prolog execution as the depth first traversal of an *and/or* tree in which each *or* node corresponds to the selection of a rule (and has a child for each possible rule choice) and each *and* node represents the current goal at that point.

It is easy to show that SLD-resolution is sound. That is, if the algorithm can terminate successfully, then G logically follows from P. It can also be shown to be complete; if G logically follows from P, then there is some series of choices (of goals and rules) such that the algorithm will terminate successfully. In fact, the way in which a subgoal to be matched is selected from the current goal, called the *computation rule*, is unimportant. Any method of selecting goal atoms can lead to a successful termination. This is known as the *independence* of the computation rule [Lloyd, 1984]. It is important to note that the method of choosing rules is not independent, and a bad selection scheme can lead the system to be incomplete. As mentioned above, Prolog always selects the first goal and then selects the first rule in the program that can match. This can cause Prolog to go into a loop and fail to prove true goals, as, for example, with the goal p in the presence of the program $\langle p \subset p, p \rangle$.

While this is a serious limitation, the rigid selection scheme for goals and rules gives Prolog a natural, operational, or procedural reading. When a goal is replaced by the body of the first rule whose head matches it, each successive subgoal in the body must then be completed,

in turn, before the system attempts to deal with the next subgoal in the original goal. Thus each subgoal can be thought of as issuing a call to a procedure. The procedure executes until completion and then returns control to the point at which it was called. A group of rules with the same predicate as head can be seen as giving alternate definitions of the procedure named by that predicate. Because the rules are always selected in the order in which they occur in a program, the programmer knows exactly how execution will proceed.

Thus while Prolog is incomplete, it is predictable. This goal directed execution model is the reason that we can view Prolog as a programming system rather than as a theorem prover. In the rest of this dissertation, as extensions to Prolog are introduced it will be important both that they be sound and complete relative to some formal system (when viewed non-deterministically) and that they maintain this natural procedural reading (when implemented deterministically).

2.3 What Kind of Logic is Horn Clauses?

It was stated above that SLD-resolution is sound and complete. But sound and complete relative to what? Most logic programming texts take it for granted that truth corresponds to classical truth. One reason for this assumption is that resolution-refutation, of which SLD-resolution is a simplification, is a proof system for classical logic. But, in fact, Horn clauses form such a small language that its theory can be viewed as either classical or intuitionistic, and SLD-resolution provability corresponds to both classical and intuitionistic provability [Miller *et al.*, 1991].

The importance of the computed answer substitution in logic programming is a strong indication, however, that logic programming is really a game of intuitionistic logic. When the user asks for the truth (or provability) of a non-ground query $\exists x.G(x)$ the system is expected to return not only `yes` or `no`, but also a value of x for which $G(x)$ is true. But this is exactly the *existential property* of intuitionistic logic. Consider the formula:

$$\exists x.[[(p(a) \wedge p(b)) \supset q] \supset (p(x) \supset q)]$$

It is provable in classical logic, even though there is no one value for x such that the formula is provable. That is, neither $[(p(a) \wedge p(b)) \supset q] \supset (p(a) \supset q)$ nor $[(p(a) \wedge p(b)) \supset q] \supset (p(b) \supset q)$ is classically provable. Miller justifies his use of intuitionistic logic in part on the classical truth of this formula [Miller, 1989*b*]. For more on the existential property as it relates to logic programming, see Appendix *A*.

Another reason to look towards intuitionistic logic when considering extensions to Prolog is the question of minimal sets of connectives. In classical logic any formula is equivalent to some other formula constructed of just disjunctions, conjunctions, and negations. For instance $(p \supset q)$ is classically equivalent to $(\neg p \vee q)$. This assumption is central to the resolution-refutation technique. If we are interested in augmenting the behavior of Prolog, the new behaviors can come only from expanding the logical language. But if the new formulae

$$\frac{}{A \longrightarrow A}\ \textit{identity} \qquad \frac{}{\longrightarrow T}\ \top_R$$

$$\frac{\Gamma \longrightarrow C \quad \Delta, C \longrightarrow B}{\Gamma, \Delta \longrightarrow B}\ \textit{cut} \qquad \frac{\Gamma, B, B \longrightarrow C}{\Gamma, B \longrightarrow C}\ C \qquad \frac{\Gamma \longrightarrow C}{\Gamma, B \longrightarrow C}\ W$$

$$\frac{\Gamma, B \longrightarrow E}{\Gamma, B \wedge C \longrightarrow E}\ \wedge_{L_1} \qquad \frac{\Gamma, C \longrightarrow E}{\Gamma, B \wedge C \longrightarrow E}\ \wedge_{L_2}$$

$$\frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge_R$$

$$\frac{\Gamma, B \longrightarrow E \quad \Gamma, C \longrightarrow E}{\Gamma, B \vee C \longrightarrow E}\ \vee_L$$

$$\frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow B \vee C}\ \vee_{R_1} \qquad \frac{\Gamma \longrightarrow C}{\Gamma \longrightarrow B \vee C}\ \vee_{R_2}$$

$$\frac{\Gamma \longrightarrow B \quad \Gamma, C \longrightarrow E}{\Gamma, B \supset C \longrightarrow E}\ \supset_L \qquad \frac{\Gamma, B \longrightarrow C}{\Gamma \longrightarrow B \supset C}\ \supset_R$$

$$\frac{\Gamma, B[x \mapsto t] \longrightarrow C}{\Gamma, \forall x.B \longrightarrow C}\ \forall_L \qquad \frac{\Gamma \longrightarrow B[x \mapsto t]}{\Gamma \longrightarrow \exists x.B}\ \exists_R$$

$$\frac{\Gamma, B[x \mapsto c] \longrightarrow C}{\Gamma, \exists x.B \longrightarrow C}\ \exists_L \qquad \frac{\Gamma \longrightarrow B[x \mapsto c]}{\Gamma \longrightarrow \forall x.B}\ \forall_R$$

provided, in each case, c does not appear free in the conclusion

Figure 2.1: Gentzen's proof system $\mathcal{LJ}$ for first-order intuitionistic logic.

are reduced to the same set of connectives as the old ones it is hard to see how any new behaviors can be obtained. In first-order intuitionistic logic none of the standard connectives are reducible to the others.

2.4 The Gentzen Sequent Calculus for Intuitionistic Logic

One of the earliest formalizations of the proof theory of intuitionistic logic was given by Gentzen in 1935 [Gentzen, 1969]. A slightly modified version of the Gentzen *sequent calculus* for intuitionistic logic, which he called the system $\mathcal{LJ}$, is shown in Figure 2.1. This system formalizes proofs as follows:

Definition 2.4 A *sequent* is an expression of the form:

$$\Gamma \longrightarrow G$$

where Γ is a syntactic variable for a multiset of formulae and G is a single formula.[4] [5] The intended reading of the sequent is that if all the formulae in Γ are true, then G is true, or that from assumptions Γ we can prove G. One or more formulae or multisets may appear explicitly on the left of the arrow separated by a comma (","), which will stand for multiset union. Depending on the situation, Γ will be referred to as the *antecedent* of the sequent, the *proof context*, or the *(current) program*. G will similarly be called the *succedent*, the *query*, or the *goal*.[6]

Definition 2.5 An *inference rule* is a structure of the form:

$$\frac{\Gamma_1 \longrightarrow G_1 \quad \cdots \quad \Gamma_n \longrightarrow G_n}{\Gamma_0 \longrightarrow G_0}$$

where $n \geq 0$. The sequent below the line is called the *conclusion* and the sequents above the line, if any, are the *premises* or *assumptions*. The intended reading of an inference rule is that if all of the assumptions are true then it is valid to infer the truth of the conclusion. These rules are to be taken as schemata for actual inferences in which all the schematic variables are replaced by objects of the appropriate type. Inference rules with no assumptions are called *axioms* and define the valid *initial sequents* of the system: those whose truth is immediate.

There are two kinds of axioms, *logical* and *non-logical*. A logical axiom asserts the truth of a sequent based only on the logical structure of its antecedent and succedent. For instance, the logical axiom:

$$\frac{}{A \longrightarrow A}\ \textit{identity}$$

states that a sequent in which the succedent is the same as the antecedent is immediately true. In contrast, a non-logical axiom is one which depends on the particular term or predicate structure of some of its constituent formulae. The following axioms of Peano arithmetic are non-logical axioms:

$$\frac{}{\Gamma \longrightarrow \mathit{int}(0)}\ \textit{zero} \qquad \frac{}{\Gamma, \mathit{int}(x) \longrightarrow \mathit{int}(s(x))}\ \textit{successor}$$

For the rest of this dissertation we will limit our view to systems without non-logical axioms. Some proofs representing program executions will include non-logical axioms for the built-in predicates of the language being demonstrated, but these will not be considered in the theoretical portions of the dissertation.

We have also restricted this (and future) systems such that *identity* applies only to sequents with atomic succedents. That is, sequents in which the succedent is just an atomic formula.

[4] Appendix *A* discusses the importance of the reason for restricting the succedent to a single formula. Without this restriction this would become a proof system for classical logic.

[5] To be precise, intuitionistic sequents allow the succedent to be empty. The restriction to exactly on formula makes this *minimal* logic. However, since we will not be considering negation, this is a sensible restriction.

[6] It is now common to see sequent systems written with the provability symbol "$\vdash$" used in place of the sequent arrow. We prefer the traditional notation as it avoids confusion.

This is not necessary, but will simplify some arguments later. It is easy to show that the restriction does not limit the reasoning power of the system: any proof in a variant with non-atomic identity succedents can be reduced to one with this restriction. For instance, the axiom instance:

$$\frac{}{(q \wedge r) \supset p \longrightarrow (q \wedge r) \supset p}$$

can be replaced with the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q \longrightarrow q} \quad \dfrac{}{r \longrightarrow r}}{q, r \longrightarrow q \wedge r}\,\wedge_R}{q \wedge r \longrightarrow q \wedge r}\,\wedge_L \quad \dfrac{}{p \longrightarrow p}}{(q \wedge r) \supset p, q \wedge r \longrightarrow p}\,\supset_L}{(q \wedge r) \supset p \longrightarrow (q \wedge r) \supset p}\,\supset_R$$

Definition 2.6 A *sequent calculus derivation* is a tree structure (drawn root down) in which the nodes are decorated with valid sequents and each sequent, when treated as the conclusion with all its children treated as premises, forms an instance of one of the inference rules of the system. The sequent at the root of a tree is the *endsequent* of the derivation. The endsequent is the conclusion of the *last* or *final* rule of the derivation. Leaves which are not axioms are called the *open assumptions* of the derivation.

Definition 2.7 A *sequent calculus proof* is a derivation in which all of the leaves are decorated with valid initial sequents; that is, one with no open assumptions.

When more than one proof system is under consideration, it will be important to distinguish which system is intended. Therefore we will speak of, for example, an $\mathcal{LJ}$-proof or an $\mathcal{LJ}$-derivation.

Definition 2.8 A sequent $\Gamma \longrightarrow G$ is said to be *provable* in the system S (written $\Gamma \vdash_S G$) if the sequent $\Gamma \longrightarrow G$ has an S-proof, that is, if there is an S-proof with it as endsequent. A formula, G, is said to be provable in S if $\vdash_S G$.

Most of the rules in Figure 2.1 are either axioms or *introduction rules*, which define the behavior of the logical connectives. For example, the $\wedge_R$ rule states that if when everything in Γ is true B is also true, and when Γ is true C is true, then when Γ is true the compound formula $B \wedge C$ is true.

Three of the rules stand out, however. The *cut* rule explains how to reason with lemmas, an important part of forming logical arguments.[7] The C (for *contraction*) and W (for *weakening*) rules are called structural rules, and are used to control the multiplicity of formulae in the antecedent. They are used, for instance, when copies of a formula are used in both parts of a

[7] Most of the rules come in left/right pairs. Girard argues that cut and identity form such a pair, with identity the left rule and cut the right rule [Girard *et al.*, 1989]. *Identity* can also be seen as specifying that provability is reflexive, and *cut* that it is transitive.

tree. For example, an $\mathcal{LJ}$-proof of $(p \wedge q) \supset (p \wedge q)$ could be built as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{p \longrightarrow p}}{p \wedge q \longrightarrow p}\ \wedge L_1 \quad \dfrac{\overline{q \longrightarrow q}}{p \wedge q \longrightarrow q}\ \wedge L_2}{p \wedge q, p \wedge q \longrightarrow p \wedge q}\ \wedge_R}{p \wedge q \longrightarrow p \wedge q}\ C}{\longrightarrow (p \wedge q) \supset (p \wedge q)}\ \supset_R$$

Gentzen's original formulation of $\mathcal{LJ}$ used a list, rather than multiset, in the antecedent. That version included a third structural rule, *interchange* (sometimes called *exchange* or *permutation*), used to rearrange the elements of the antecedent. Using a list structure without providing the exchange rule leads to a non-commutative logic.

The structural rules are often viewed as unimportant, and, as will be shown in the next section, can often be made implicit in the system. In fact, though, it can be argued that the structural rules are the most important ones in the system, as they determine the behavior of the logical connectives in crucial ways. The study of *substructural* logics, in which one or more of the structural rules has been eliminated, has been one of the most fruitful areas of logic research for the last several decades. In later sections we will show how restricting the use of the structural rules can enrich the power of a logic programming language based on such a system.

2.5 Logic Programming as Sequent Calculus Proof Search

As described above, inference figures are designed to be read top-down. That is, from the premises to the conclusion. In that way they describe valid inferences. It is, however, possible to read them from the conclusion to the premises, as instructions for how to search for proofs. Thus the $\wedge_R$ rule can be read as stating that if you wish to prove $B \wedge C$ in some context, try to prove B and C independently in that context. We will take the view that logic programming execution corresponds to the search for proofs using this technique. In order to see how this works, it will help to first simplify the system $\mathcal{LJ}$.

The central theorem of the paper in which Gentzen introduced the sequent calculus was that the cut rule can be eliminated from $\mathcal{LJ}$ without changing the set of sequents provable in the system. The essential idea is that rather than proving a lemma and then referring to it in other parts of the proof, the system should reprove the lemma each time it is needed. Proofs without *cut* can thus be substantially (in fact, hyper-exponentially) larger, but they are simpler. In the case of $\mathcal{LJ}$, Gentzen did not simply prove that any $\mathcal{LJ}$-provable sequent could be proved without *cut*. He gave an algorithm (guaranteed to terminate) which would convert an $\mathcal{LJ}$-proof with instances of *cut* into a cut-free proof of the same endsequent [Gentzen, 1969].

The eliminability of *cut* is not just an interesting philosophical result. It has enormous importance to theorem proving (both manual and automated) and logic programming. Consider that the cut rule is the only rule in which a formula (C, the *cut formula*) appears in the

premises of the rule but not in the conclusion. Without this result, the root-upward search for proofs might require guessing non-deterministically at what the needed cut formula might be. Instead the prover can always restrict its view to formulae of which it already has knowledge. Therefore only logics for which the cut-elimination theorem holds will be of interest, and we will always be looking for cut-free proofs.

If we are interested only in cut-free proofs, it is possible to modify $\mathcal{LJ}$ to do away with explicit weakening and contraction rules. Explicit weakening[8] can be discarded in favor of implicit weakening simply by replacing *identity* and $\top_R$ with, respectively:

$$\frac{}{\Gamma, A \longrightarrow A}\ \textit{identity} \qquad \frac{}{\Gamma \longrightarrow \top}\ \top_R$$

To remove weakening from an $\mathcal{LJ}$-proof, for each instance of weakening just remove the instance and add the weakened formula to the antecedents of all the sequents above it in the proof tree. The leaves of the modified tree will be valid instances of the new versions of *identity* and $\top_R$.

Removing contraction from the system involves a bit more work, including changing the way antecedents are treated in each of the connective introduction rules. Whenever a formula is used in a left-introduction rule, it must be explicitly reproduced in the antecedent of the premises. This replaces any contractions that might have been needed. If the formula was not actually needed in the premises, it will simply be carried along as extra baggage to be weakened away at the leaves. The right-hand rules are also changed to make sure they handle antecedents correctly.[9]

Figure 2.2 gives a variant of $\mathcal{LJ}$ called $\mathcal{LJ}'$with implicit weakening and contraction. It is straightforward to show that if a sequent has a cut-free $\mathcal{LJ}$-proof, then it has an $\mathcal{LJ}'$-proof, and vice-versa. This is shown by giving an algorithm to convert valid cut-free $\mathcal{LJ}$-proofs to $\mathcal{LJ}'$-proofs and back.

The system $\mathcal{H}$, given in Figure 2.3 is just a subset of the rules from $\mathcal{LJ}'$ sufficient for Horn clause provability. If P is a multiset of definite clauses and G is a goal formula, then $P \longrightarrow G$ has an $\mathcal{H}$-proof if and only if it has an $\mathcal{LJ}'$-proof. This is clear by the *sub-formula property* that arises from the cut-elimination theorem. It is possible to simplify $\mathcal{H}$ slightly by limiting the $\supset_L$ rule to clauses with atomic heads (since this is a restriction on Horn clauses) as in:

$$\frac{\Gamma, B \supset A \longrightarrow B \quad \Gamma, B \supset A, A \longrightarrow E}{\Gamma, B \supset A \longrightarrow E}\ \supset_L$$

but it does not change the system in any interesting way.

If we use unification to delay the choice of instantiation in the $\exists_R$ and $\forall_L$ rules, it is easy to see how Prolog execution can be mapped onto the search for $\mathcal{H}$ proofs. But the system

[8] When reading rules conclusion-up, weakening is often called *thinning*, and contraction is called *expansion*, for obvious reasons. We find it less confusing to use just one name for each rule, however.

[9] It is also possible to eliminate the need for explicit contractions by representing the antecedents of sequents as sets rather than multisets. This is a common technique and has been used by this author in other settings.

$$\frac{}{\Gamma, A \longrightarrow A}\ \textit{identity} \qquad \frac{}{\Gamma \longrightarrow \top}\ \top_R$$

$$\frac{\Gamma, B, C \longrightarrow E}{\Gamma, B \wedge C \longrightarrow E}\ \wedge_L \qquad \frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge_R$$

$$\frac{\Gamma, B \longrightarrow E \quad \Gamma, C \longrightarrow E}{\Gamma, B \vee C \longrightarrow E}\ \vee_L$$

$$\frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow B \vee C}\ \vee_{R_1} \qquad \frac{\Gamma \longrightarrow C}{\Gamma \longrightarrow B \vee C}\ \vee_{R_2}$$

$$\frac{\Gamma, B \supset C \longrightarrow B \quad \Gamma, B \supset C, C \longrightarrow E}{\Gamma, B \supset C \longrightarrow E}\ \supset_L \qquad \frac{\Gamma, B \longrightarrow C}{\Gamma \longrightarrow B \supset C}\ \supset_R$$

$$\frac{\Gamma, \forall x.B, B[x \mapsto t] \longrightarrow C}{\Gamma, \forall x.B \longrightarrow C}\ \forall_L \qquad \frac{\Gamma \longrightarrow B[x \mapsto t]}{\Gamma \longrightarrow \exists x.B}\ \exists_R$$

$$\frac{\Gamma, \exists x.B, B[x \mapsto c] \longrightarrow C}{\Gamma, \exists x.B \longrightarrow C}\ \exists_L \qquad \frac{\Gamma \longrightarrow B[x \mapsto c]}{\Gamma \longrightarrow \forall x.B}\ \forall_R$$

provided, in each case, c does not appear free in the conclusion

Figure 2.2: The proof system $\mathcal{LJ}'$ with implicit weakening and contraction.

$$\frac{}{\Gamma, A \longrightarrow A}\ \textit{identity} \qquad \frac{}{\Gamma \longrightarrow \top}\ \top_R$$

$$\frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge_R \qquad \frac{\Gamma \longrightarrow B[x \mapsto t]}{\Gamma \longrightarrow \exists x.B}\ \exists_R$$

$$\frac{\Gamma, \forall x.B, B[x \mapsto t] \longrightarrow C}{\Gamma, \forall x.B \longrightarrow C}\ \forall_L \qquad \frac{\Gamma, B \supset C \longrightarrow B \quad \Gamma, B \supset C, C \longrightarrow E}{\Gamma, B \supset C \longrightarrow E}\ \supset_L$$

Figure 2.3: The proof system $\mathcal{H}$ for first-order positive Horn clauses.

as described is still more powerful than necessary. At any point in the proof search several rules may be applicable —that is, an open assumption may match more than one rule schema (for instance, a left rule and a right rule might both apply)— and there is nothing in the logic to specify how to choose. This is in contrast to SLD-resolution where, while the program may allow several choices for backchaining, backchaining is the only action available at each step. As we begin to look at more complex logical systems, it will be important to have an equally goal-directed form of sequent calculus proof search.

2.5.1 Uniform proofs

How does one define goal-directedness in the sequent calculus? Miller, et al. answered this question in the 1980's (in the context of hereditary Harrop formulae, which will be discussed in the next chapter) by noting that during the execution of a Prolog query the goal can be decomposed uniformly, based only on its structure and without reference to the program, until atomic goals are reached. The program is only consulted when atomic goals are to be

proved. From this idea they extracted the following definition [Miller, 1989*b*; Miller *et al.*, 1991]:

Definition 2.9 A cut-free sequent proof is a *uniform proof* if for every occurrence of a sequent in the proof for which the right-hand side is not an atomic formula, that sequent is the conclusion of a right-introduction rule.

It is easy to show the following proposition:

Proposition 2.1 Let Γ be a multiset of Horn clauses and let G be a goal formula. Then the sequent $\Gamma \longrightarrow G$ is provable in the system $\mathcal{H}$ if and only it has a uniform proof in $\mathcal{H}$.

Proof. The proof in the reverse direction is immediate, since a uniform $\mathcal{H}$-proof is an $\mathcal{H}$-proof. In the forward direction the proof is in the form of an algorithm which converts an arbitrary $\mathcal{H}$-proof over the restricted formula language to a uniform proof of the same endsequent. To describe this algorithm we will need to define the notion of a *non-uniform rule occurrence*, which is any occurrence of a left-rule in which the right-hand side of the conclusion is not an atomic formula.
The algorithm is stated as follows:

1. If the proof is uniform, terminate.
2. Select a non-uniform rule occurrence with the property that the subproof(s) rooted at premise(s) of the rule are uniform. (That such a choice can be made is immediate given the assumption that identity inferences have only atomic right-hand sides.)
3. One or more of the premise(s) of the rule selected will, necessarily, be the conclusion(s) of (a) right-rule(s). There are only 4 such combinations of left-rules below right-rules possible in an $\mathcal{H}$-proof. For each possibility we show how to permute the rules to move the left-rule up through the right rule. In each case we assume that the Ξ_i are uniform proofs of their respective endsequents.
   - An instance of $\forall_L$ beneath $\exists_R$. A proof structure of the form:

$$\dfrac{\dfrac{\begin{matrix}\Xi_1\\ \Gamma, B[x_1 \mapsto t_1] \longrightarrow C[x_2 \mapsto t_2]\end{matrix}}{\Gamma, B[x_1 \mapsto t_1] \longrightarrow \exists x_2.C}\ \exists_R}{\Gamma, \forall x_1.B \longrightarrow \exists x_2.C}\ \forall_L$$

     is converted to a structure of the form:

$$\dfrac{\dfrac{\begin{matrix}\Xi_1\\ \Gamma, B[x_1 \mapsto t_1] \longrightarrow C[x_2 \mapsto t_2]\end{matrix}}{\Gamma, \forall x_1.B \longrightarrow C[x_2 \mapsto t_2]}\ \forall_L}{\Gamma, \forall x_1.B \longrightarrow \exists x_2.C}\ \exists_R$$

- An instance of $\forall_L$ beneath an $\wedge_R$. A proof structure of the form:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma, B[x \mapsto t] \longrightarrow C_1} \quad \overset{\Xi_2}{\Gamma, B[x \mapsto t] \longrightarrow C_2}}{\Gamma, B[x \mapsto t] \longrightarrow C_1 \wedge C_2}\ \wedge_R}{\Gamma, \forall x.B \longrightarrow C_1 \wedge C_2}\ \forall_L$$

  is converted to a structure of the form:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma, B[x \mapsto t] \longrightarrow C_1}}{\Gamma, \forall x.B \longrightarrow C_1}\ \forall_L \quad \dfrac{\overset{\Xi_2}{\Gamma, B[x \mapsto t] \longrightarrow C_2}}{\Gamma, \forall x.B \longrightarrow C_2}\ \forall_L}{\Gamma, \forall x.B \longrightarrow C_1 \wedge C_2}\ \wedge_R$$

- An instance of $\supset_L$ beneath $\exists_R$. A proof structure of the form:

$$\dfrac{\overset{\Xi_1}{\Gamma \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma, C \longrightarrow E[x \mapsto t]}}{\Gamma, C \longrightarrow \exists x.E}\ \exists_R}{\Gamma, B \supset C \longrightarrow \exists x.E}\ \supset_L$$

  is converted to a structure of the form:

$$\dfrac{\overset{\Xi_1}{\Gamma \longrightarrow B} \quad \overset{\Xi_2}{\Gamma, C \longrightarrow E[x \mapsto t]}}{\dfrac{\Gamma, B \supset C \longrightarrow E[x \mapsto t]}{\Gamma, B \supset C \longrightarrow \exists x.E}\ \exists_R}\ \supset_L$$

- The case of an instance of $\supset_L$ beneath an $\wedge_R$ is a bit more complicated. A proof structure of the form:

$$\dfrac{\overset{\Xi_1}{\Gamma \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma, C \longrightarrow E_1} \quad \overset{\Xi_3}{\Gamma, C \longrightarrow E_2}}{\Gamma, C \longrightarrow E_1 \wedge E_2}\ \wedge_R}{\Gamma, B \supset C \longrightarrow E_1 \wedge E_2}\ \supset_L$$

  is converted to a structure of the form:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma \longrightarrow B} \quad \overset{\Xi_2}{\Gamma, C \longrightarrow E_1}}{\Gamma, B \supset C \longrightarrow E_1}\ \supset_L \quad \dfrac{\overset{\Xi_1}{\Gamma \longrightarrow B} \quad \overset{\Xi_3}{\Gamma, C \longrightarrow E_2}}{\Gamma, B \supset C \longrightarrow E_2}\ \supset_L}{\Gamma, B \supset C \longrightarrow E_1 \wedge E_2}\ \wedge_R$$

4. In each of the cases of step 3 it is possible that the subproofs rooted at the premises of the new final inference are not uniform, since new non-uniform rule instances may have been created by permuting the left rule upwards. The algorithm is therefore called recursively on each of these subproofs. Since each recursive call is on a smaller proof

structure, termination is assured. With the completion of the recursion(s) the number of non-uniform rule occurrences in the overall proof is reduced by one.

5. Return to the step 1.

Since each pass through the outer loop of this algorithm produces a proof of the original endsequent with one fewer non-uniform rule occurrence, the algorithm will terminate. Thus, any sequent provable in $\mathcal{H}$ can be proved by a uniform $\mathcal{H}$-proof. ■

The last case of the permutation step is worth examining as it demonstrates the computational cost of restricting a proof system to a goal-directed backwards chaining form of proof search. Suppose each of the conjuncts, E_1 and E_2, relies on C. In the original proof, the body of the clause for C is proved only once. In the uniform proof, the entire subproof, Ξ_1, of the body of the clause must be repeated. An example of this sort of duplication is discussed in Section 2.6. Nevertheless, uniformity appears to describe that aspect of Prolog which gives it its procedural reading. Therefore it seems to be an appropriate restriction to place on any logic that is to serve as a logic programming language.[10] To that end Miller, et al. made the following definition [Miller *et al.*, 1991]:

Definition 2.10 A triple $\langle D, G, \vdash \rangle$, where D and G are sets of formulae in some logical language and $\vdash$ is a provability relation, is an *(abstract) logic programming language* if for every finite subset $\Gamma \subseteq D$ and for every $G \in G$, the sequent $\Gamma \longrightarrow G$ has a proof in the proof system $\vdash$ if and only if it has a uniform proof in $\vdash$. The set D represents those formulae that are taken to be program clauses and the set G are those formulae that are taken to be goals.

The previous proof then amounts to a proof of the following proposition:

Proposition 2.2 If $D_{\mathcal{H}}$ is the set of Horn clauses given in Definition 2.2, $G_{\mathcal{H}}$ is the set of goal formulae given in Definition 2.1, and $\vdash_{\mathcal{H}}$ refers to provability using the inference rules in Figure 2.3, then $\langle D_{\mathcal{H}}, G_{\mathcal{H}}, \vdash_{\mathcal{H}} \rangle$ is an abstract logic programming language.

2.5.2 Backchaining

It is possible to simplify $\mathcal{H}$ significantly by recognizing that in uniform proofs the two left rules and *identity* are used only in a very specific way: to perform backchaining. First we define the following set:

Definition 2.11 Let D be a Horn clause as in Definition 2.2. Define $|D|$ to be the smallest set of pairs of the form $\langle \Delta, D' \rangle$, where Δ is a multiset of goal formulae, such that:

[10] Harland has considered the notion of uniform proof in some depth and shown a connection to several issues in proof theory and the theory of logic programming [Harland, 1991; Harland and Pym, 1991; Harland, 1993] Andreoli has proposed the alternative notion of *focusing proof* as a key feature for logic programming in linear logic [Andreoli, 1992]. That work, however, is principally involved with a multi-conclusion, classical logic, and seems to relate to concurrent, more than sequential, systems.

$$\frac{}{\Gamma \longrightarrow \top}\ \top_R \qquad \frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge_R \qquad \frac{\Gamma \longrightarrow B[x \mapsto t]}{\Gamma \longrightarrow \exists x.B}\ \exists_R$$

$$\frac{\Gamma, D \longrightarrow G_1 \quad \cdots \quad \Gamma, D \longrightarrow G_n}{\Gamma, D \longrightarrow A}\ BC$$

provided $n \geq 0$, A is atomic, and $\langle \{G_1, \ldots, G_n\}, A \rangle \in |B|$.

Figure 2.4: The modified proof system $\mathcal{H}'$ for first-order positive Horn clauses.

1. $\langle \emptyset, D \rangle \in |D|$,
2. if $\langle \Delta, \forall x.D' \rangle \in |D|$ then for all closed terms t, $\langle \Delta, D'[t \mapsto x] \rangle \in |D|$, and
3. if $\langle \Delta, G \supset D' \rangle \in |D|$ then $\langle \Delta \uplus \{G\}, D' \rangle \in |D|$ (where '$\uplus$' is multiset union).

Informally, if $\langle \Delta, A \rangle \in |D|$ then A is an instance of the head formula of D and the clause D can be used to establish the formula A if each of the formulae in the set Δ can be established;[11] that is, A might be proved by backchaining over D. Figure 2.4 shows a modified proof system, $\mathcal{H}'$, for Horn clauses. It has only one left-rule, namely BC, and proofs in $\mathcal{H}'$ are necessarily uniform since BC applies only to sequents with atomic succedents.

Proposition 2.3 Let Γ be a multiset of Horn clauses, and let G be a goal formula. Then, the sequent $\Gamma \longrightarrow G$ has a proof in $\mathcal{H}$ if and only if it has a proof in $\mathcal{H}'$.

Since uniform proofs are complete for $\mathcal{H}$ it is sufficient to show that there is a proof in $\mathcal{H}'$ if and only if there is a uniform proof in $\mathcal{H}$. The proof is given by showing that the rules in a uniform $\mathcal{H}'$-proof can be permuted so that the left rules occur in bunches and that these bunches can be replaced by instances of the BC rule. The proof will not be shown here. A proof of this proposition has been given by Miller for a richer language, as will be discussed in the next chapter [Miller, 1989*b*]. A similar proof is given in Chapter 4 for the case of a linear-logic programming language.

2.5.3 Adding disjunctions to the system

In SLD-resolution it was easier to eliminate disjunctions from the program in advance than to consider how to deal with them in the proof system.[12] In contrast, extending $\mathcal{H}'$ to directly handle disjunctions in goals, and hence in the bodies of clauses, is straightforward; all that is

[11] In fact, because of the simple structure of Horn clauses, the set Δ will always be either empty or a singleton (a conjunction of atomic formulae). In subsequent chapters this definition will be enriched such that Δ is truly a set.

[12] While SLD-resolution does not take disjunctions into account directly, Prolog execution actually incorporates the consideration of alternate disjuncts directly into the proof search process, rather than preprocess the clauses to remove them.

needed is the pair of $\vee_R$ inference figures from the system $\mathcal{LJ}'$:

$$\frac{\Gamma \longrightarrow B_1}{\Gamma \longrightarrow B_1 \vee B_2}\ \vee_{R_1} \qquad \frac{\Gamma \longrightarrow B_2}{\Gamma \longrightarrow B_1 \vee B_2}\ \vee_{R_2}$$

Adding these rules to $\mathcal{H}'$ (and making the appropriate changes to the definitions of goal formulae and definite clauses) will not interfere with the completeness of uniform proofs for that system.

2.6 Optimization by Sequent Proof Transformation

The fact that Prolog's execution model is based on such a clear, simple logic is a potential boon to implementers. Many optimizations are possible based solely on manipulations of the logic. Such transformations are easy to understand, and, more importantly, their correctness is immediate. For instance, given a goal $p \wedge p$ in the presence of a context Γ which includes clauses with p as head formula, we know that the uniform proof constructed for the goal will be of the form:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma \longrightarrow G} \quad \overline{\Gamma, p \longrightarrow p}}{\Gamma, G \supset p \longrightarrow p}\ \supset_L \quad \dfrac{\overset{\Xi_2}{\Gamma \longrightarrow G} \quad \overline{\Gamma, p \longrightarrow p}}{\Gamma, G \supset p \longrightarrow p}\ \supset_L}{\Gamma, G \supset p \longrightarrow p \wedge p}\ \wedge_R$$

Confronted with this sort of duplicate goal, an implementation might relax the uniformity constraint, recognizing the advantage of first forward-chaining on the rule for p. The proof tree that results is then:

$$\dfrac{\overset{\Xi_i}{\Gamma \longrightarrow G} \quad \dfrac{\overline{\Gamma, p \longrightarrow p} \quad \overline{\Gamma, p \longrightarrow p}}{\Gamma, p \longrightarrow p \wedge p}\ \wedge_R}{\Gamma, G \supset p \longrightarrow p \wedge p}\ \supset_L$$

where Ξ_i is one of the two proofs of $\Gamma \longrightarrow G$ (which are not necessarily different, but could be) from the uniform proof. If the proof of $\Gamma \longrightarrow G$ is large then the savings may be considerable. In general the system would be free to apply this transformation to any atomic formula that will be proved more than once during a given query. The correctness of the transformation is immediate from the equivalence of uniform and non-uniform $\mathcal{H}$-proofs, given in Proposition 2.1. Obviously this is not a terribly realistic example, since simple repeated goals are unlikely to occur in real life and searching for them could be quite costly, but it is a simple example of the sorts of transformations that are possible in this setting.

2.6.1 Roadblocks to Proof Transformation

As with many optimization schemes, this one breaks down in the presence of side-effects and other non-logical operations. Unfortunately such operations are more common in real Prolog code than one might hope.

Horn clause logic is simply too weak to directly support a number of features that programmers demand. A particular problem is that pure Prolog programs are completely flat. All constants (including function and predicate names) and clauses are global. There is no notion of scope, modularity, or locality of reference. This is immediately apparent from the fact that the proof context is fixed in each inference figure of $\mathcal{H}'$.

To solve this problem every Prolog system provides extra-logical language features, such as module systems and the dynamic predicates `assert`/`retract`. Unfortunately, when these features are used it is no longer possible to reason soundly about programs. Suppose that during the proof of G in the last example some fact, say q, were `assert`'ed. Then the original proof tree would look like:

$$\dfrac{\dfrac{\overline{\Gamma, p \longrightarrow p} \quad \overset{\Xi}{\Gamma \longrightarrow G}}{\Gamma, G \supset p \longrightarrow p}\ \supset_L \qquad \dfrac{\overline{\boxed{q}, \Gamma, p \longrightarrow p} \quad \overset{\Xi'}{\boxed{q}, \Gamma \longrightarrow G}}{\boxed{q}, \Gamma, G \supset p \longrightarrow p}\ \supset_L}{\Gamma, G \supset p \longrightarrow p \wedge p}\ \wedge_R$$

and the transformation used before is no longer valid since the endsequents of Ξ and Ξ' are now different.

A common use of dynamic predicates is to temporarily assert information needed for a proof. For instance the proof of G might involve `assert`'ing the fact q temporarily, `retract`'ing it before the proof of G is completed. Such a local use of dynamic predicates would not invalidate the transformation of the proof of $p \wedge p$. Unfortunately, Prolog makes no distinction between local and global assertions, and detecting the difference would involve difficult (and logically precarious) analysis. In the next chapter we will see how expanding the logical language to include other operations can provide, among other features, just this sort of local assertion without weakening the ability to reason about programs.

Chapter 3

The Language λProlog and the Logic of Hereditary Harrop Formulas

Several researchers have, in recent years, proposed the idea of extending Prolog, not by adding extra-logical features, but rather by adopting a richer logical language that will directly support the features desired.

Of course, not all logics are appropriate for use as a logic programming language. A notion of goal directed proof, as formalized by something like the uniform proof constraint, seems to be a minimum requirement. Full first-order logic certainly fails on that count. That is, if D and G are taken to be all first-order formulae and $\vdash$ is taken to be either classical or intuitionistic provability, then the triple $\langle D, G, \vdash \rangle$ is not a logic programming language according to the definition given in the last chapter, since in each case there are formulae that are provable, but not uniformly provable.[1] For example, the sequents $p \vee q \longrightarrow p \vee q$ and $\exists x.p(x) \longrightarrow \exists x.p(x)$ are provable in intuitionistic logic, but neither has a uniform intuitionistic proof when we restrict initial sequents to ones with atomic succedents.

3.1 Hereditary Harrop formulae

While these examples show that we cannot easily extend program clauses to allow disjunctions or existentials, it is possible to extend goal formulae with implications and universal quantification, which occur only in very restricted form in Prolog definite clauses. We begin by replacing the definitions of program clause and goal formula with the following mutually recursive definitions:

Definition 3.1 The set of *goal formulae* is the smallest set satisfying the following conditions:

[1] Actually, the definition of uniform proofs was given only in terms of intuitionistic proofs. Miller has recently proposed an extension of the definition to multiple-conclusion (i.e. classical) systems [Miller, 1993].

- $\top$, and all atomic formulae A are goal formulae.
- If G_1 and G_2 are goal formulae, then $G_1 \wedge G_2$ and $G_1 \vee G_2$ are goal formulae.
- If G is a goal formula, then $\exists x.G$ and $\forall x.G$ are goal formulae.
- If G is a goal formula and D is a definite clause (as defined below), then $D \supset G$ is a goal formula.

Definition 3.2 The set of *definite formulae* or *program clauses* is the smallest set satisfying the following conditions:

- $\top$, and all atomic formulae A are definite formulae.
- If D_1 and D_2 are definite formulae, then $D_1 \wedge D_2$ is a definite formula.
- If D is a definite formula, then $\forall x.D$ is a definite formula.
- If G is a goal formula (as defined above) and D is a definite formula, then $G \supset D$ is a definite formula.

It is also possible to view the language as being freely generated by $\top$,$\wedge$,$\supset$,and $\forall$ with the addition of negative instances of $\exists$ and $\vee$. In any case, clauses and goals are now complex formulae, and, in particular, quantifiers and implications may now appear nested within formulae of either class.

The clauses of this language are called the *hereditary Harrop formulae* as they form a variant on a class of formulae originally studied by Harrop [Harrop, 1960]. They have been extensively studied by Miller and colleagues and form the logical basis of the language λProlog [Miller and Nadathur, 1986; Miller, 1989*b*; Miller, 1987; Miller and Nadathur, 1988; Miller, 1989*a*; Miller, 1990; Miller *et al.*, 1991]. This chapter will outline the principal features of the logic and the language.

3.1.1 Uniform Proofs

Figure 3.1 defines the proof system $\mathcal{HH}$ which consists of the set of rules from $\mathcal{LJ}'$ sufficient to prove sequents built from the new definitions of program and goal formulae.

Proposition 3.1 The triple $\langle D, G, \vdash_{\mathcal{HH}} \rangle$, where D and G are the set of all program clauses and goal formulae as defined above, is an abstract logic programming language.

This can be proved in a manner similar to the proof of Proposition 2.1, by permuting enough inference rules in an $\mathcal{HH}$-proof until it is uniform. Though there are many more cases in this proof, the fact that there are no left introduction rules for $\vee$ or $\exists$ makes all the necessary permutations possible. For a closely related proof see [Miller, 1989*b*].

As with Horn clauses, it is possible to simplify the proof system significantly and still not lose completeness. Let D be a definite formula, and redefine $|D|$ to be the smallest set of pairs of the form $\langle \Delta, D' \rangle$, where Δ is a multiset of goal formulae, such that:

$$\frac{}{\Gamma, A \longrightarrow A}\ \textit{identity} \qquad \frac{}{\Gamma \longrightarrow \top}\ \top_R$$

$$\frac{\Gamma, B, C \longrightarrow E}{\Gamma, B \wedge C \longrightarrow E}\ \wedge_L \qquad \frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge_R$$

$$\frac{\Gamma, B \supset C \longrightarrow B \quad \Gamma, B \supset C, C \longrightarrow E}{\Gamma, B \supset C \longrightarrow E}\ \supset_L \qquad \frac{\Gamma, B \longrightarrow C}{\Gamma \longrightarrow B \supset C}\ \supset_R$$

$$\frac{\Gamma, \forall x.B, B[x \mapsto t] \longrightarrow C}{\Gamma, \forall x.B \longrightarrow C}\ \forall_L \qquad \frac{\Gamma \longrightarrow B[x \mapsto c]}{\Gamma \longrightarrow \forall x.B}\ \forall_R$$

provided c does not appear free in the conclusion

$$\frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow B \vee C}\ \vee_{R_1} \qquad \frac{\Gamma \longrightarrow C}{\Gamma \longrightarrow B \vee C}\ \vee_{R_2} \qquad \frac{\Gamma \longrightarrow B[x \mapsto t]}{\Gamma \longrightarrow \exists x.B}\ \exists_R$$

Figure 3.1: $\mathcal{HH}$: A proof system for a fragment of intuitionistic logic.

1. $\langle \emptyset, D \rangle \in |D|$,
2. if $\langle \Delta, D_1 \wedge D_2 \rangle \in |D|$ then both $\langle \Delta, D_1 \rangle \in |D|$ and $\langle \Delta, D_2 \rangle \in |D|$,
3. if $\langle \Delta, \forall x.D' \rangle \in |D|$ then for all closed terms t, $\langle \Delta, D'[x \mapsto t] \rangle \in |D|$, and
4. if $\langle \Delta, G \supset D' \rangle \in |D|$ then $\langle \Delta \uplus \{G\}, D' \rangle \in |D|$.

Let $\mathcal{HH}'$ be the proof system, shown in Figure 3.2, which results from replacing the *identity*, $\supset_L$, $\wedge_L$, and $\forall_L$ rules in Figure 3.1 with the *BC* rule.

Proposition 3.2 Let Γ be a multiset of definite formulae and let G be a goal formula. Then, the sequent $\Gamma \longrightarrow G$ has a proof in $\mathcal{HH}$ if and only if it has a proof in $\mathcal{HH}'$.

The proof of this proposition has been given by Miller [Miller, 1989*b*], so it will not be given here. A similar proof will be given for the case of the linear-logic programming language developed in Chapter 4.

Since there is only one left-rule in $\mathcal{HH}'$, namely *BC*, and since that rule applies only to sequents with atomic succedents, proofs in $\mathcal{HH}'$ are necessarily uniform. Therefore, the following is immediate:

Proposition 3.3 The triple $\langle D, G, \vdash_{\mathcal{HH}'} \rangle$, where D and G are as defined above, is an abstract logic programming language.

3.2 Hereditary Harrop Formulae as an Extension of Prolog

From the standpoint of programming language design, what features are gained by considering this extension of goal and program formulae? We will examine each extension in turn and discuss what features it produces at the language level. First, though, we introduce the concrete syntax of λProlog:

$$\frac{}{\Gamma, A \longrightarrow A}\ \textit{identity} \qquad \frac{}{\Gamma \longrightarrow \top}\ \top_R$$

$$\frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge_R \qquad \frac{\Gamma, B \longrightarrow C}{\Gamma \longrightarrow B \supset C}\ \supset_R$$

$$\frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow B \vee C}\ \vee_{R_1} \qquad \frac{\Gamma \longrightarrow C}{\Gamma \longrightarrow B \vee C}\ \vee_{R_2} \qquad \frac{\Gamma \longrightarrow B[x \mapsto t]}{\Gamma \longrightarrow \exists x.B}\ \exists_R$$

$$\frac{\Gamma \longrightarrow B[x \mapsto c]}{\Gamma \longrightarrow \forall x.B}\ \forall_R$$

provided c does not appear free in the conclusion

$$\frac{\Gamma, B \longrightarrow G_1 \quad \ldots \quad \Gamma, B \longrightarrow G_n}{\Gamma, B \longrightarrow A}\ BC$$

provided $n \geq 0$, A *is atomic, and* $\langle \{G_1, \ldots, G_n\}, A \rangle \in |B|$.

Figure 3.2: The Proof System $\mathcal{HH}'$.

Constant and Variable Names – In Prolog, names beginning with a lower case character are assumed to be constants, and those beginning with an upper case character are assumed to be variables implicitly universally quantified at the outer limits of the clause. (In queries, the variables are assumed to be existentially quantified.) In λProlog, these same rules apply, but only to names that have not been explicitly quantified as described below.

Terms and Formulas – Terms and formulas in λProlog are written in *curried* form. So, for instance, the Prolog term `f(a,b,g(c,d))` is written as `(f a b (g c d))` in λProlog. This eliminates the need for commas to have two readings —as term constructors and as conjunctions— and merges well with λProlog's use of λ-terms as data structures (which will not be discussed at any length here). Lists are written in the style of ML, with '`::`' and '`nil`' used for building lists.[2]

Logical Connectives and Quantification – As in Prolog, '`,`' represents conjunction, '`;`' disjunction, and '`:-`' reverse implication. To this, '`=>`' is added for forward implication,[3] and '`pi x\B`' and '`sigma x\B`' for, respectively, explicit universal and existential quantification of the name '`x`' (which may be upper or lower case) in '`B`'.[4] Finally, '`true`' is used for the zero-ary connective '$\top$'.

[2] λProlog is a strongly typed language, with type inference. This is required due to the nature of the higher-order unification algorithm discussed below. Since we will not be making that strong a use of the higher-order features of λProlog it is possible to ignore the presence of types in our discussion.

[3] The symbols '`:-`' and '`=>`' are entirely interchangeable in programs and clauses; '`B => C`' can be used anywhere '`C :- B`' can be used.

[4] The names '`pi`' and '`sigma`' were chosen due to λProlog's connection to Church's Simple Theory of Types [Church, 1940]. Also, the notation `x \ B` is, in fact, the λProlog notation for the λ-term $\lambda x.B$; '`pi`' and '`sigma`' are defined predicates that act on λ-terms.

3.2.1 Hypothetical Queries in Database Systems

A natural use of implications in goals is to extend the types of queries supported by Prolog database programs to include hypothetical reasoning. Consider the following λProlog program for a simple registrar's database:

```
took sue cs120.   took sue cs121.  took sue cs240.
took bob cs120.   took bob cs370.

can_graduate S :- took S cs120, took S cs121,
                  took S cs240, took S cs370.
```

In Prolog the only queries possible here would be to ask if a given student had taken some particular course, and whether or not they were eligible to graduate. In contrast, in λProlog it is possible to ask whether, assuming that a given student takes a given course, would they then be able to graduate, as in:

```
?-  took sue cs370 => can_graduate sue.
yes.
?- took bob cs121 => can_graduate bob.
no.
```

This technique is much simpler and more easily controlled than using `assert` to accomplish the same result, since it is not necessary to keep track of the facts that were added and `retract` them at the end. So, for instance, after the last series of queries the query `(can_graduate sue)` would fail, since the database was not altered by the hypothetical queries.[5] In order to provide a more permanent alteration, λProlog includes the extra-logical goal `top`, which causes the system to begin a new read-prove-print loop. The goal `pop` is used to return to the previous top level. So, for example:

```
?- can_graduate sue.
no.
?- took sue cs370 => top.
?- can_graduate sue.
yes.
?- pop.
?- can_graduate sue.
no.
```

If the assumed fact in a hypothetical query includes free variables, the result is a query that cannot be readily mimicked in Prolog at all. Consider the query:

```
?- took sue C => can_graduate sue.
C == cs370.
```

[5] This sort of hypothetical database query built on embedded implication has been previously examined by other researchers [Bonner *et al.*, 1989; Gabbay and Reyle, 1984]. λProlog however, allows a more general class of queries than any of those proposals.

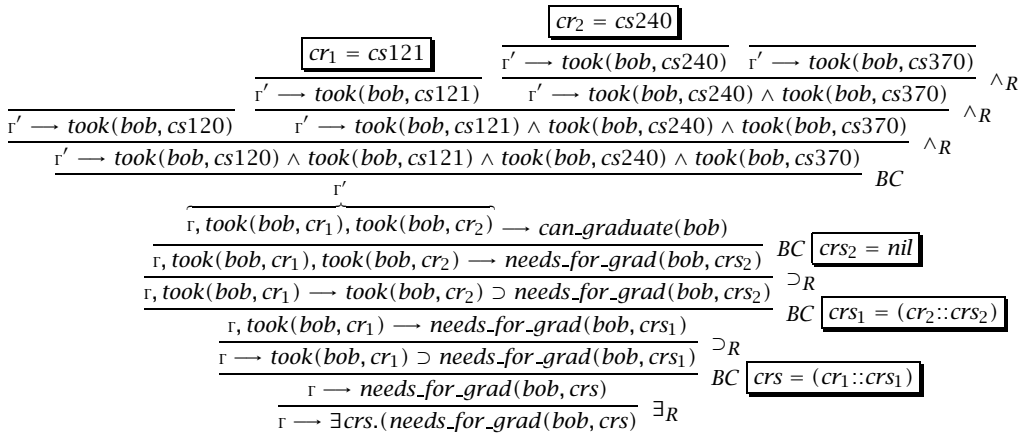


Figure 3.3: Search for the Solution of a Recursive Hypothetical Query

which corresponds to asking "Is there a course, C, such that if sue takes C she can graduate?". In order to respond to this query, λProlog adds `(took sue C)` to the database with `C` uninstantiated. Attempting to prove `(can_grad sue)` causes this new fact to be unified with the subgoal `(took sue cs370)` and thereby instantiates `C` to `cs370`. In Prolog, `assert`'ing a clause with a free variable in it causes the variable to be universally quantified in the assumed clause (just like any other clause variable). So `assert`'ing `(took sue C)` is tantamount to claiming she has taken every course under the sun.[6]

A variant of this sort of hypothetical query can be used (in combination with the Prolog search mechanism) to provide a degree of planning in database queries. If we want to know what courses a student needs to take in order to graduate, the following definition can be added to the last program:

```
needs_for_grad S nil :- can_graduate S.
needs_for_grad S (C::Cs) :- took S C ==> needs_for_grad S Cs.
```

Figure 3.3 shows a proof that bob needs to take cs121 and cs240 in order to graduate. In that proof, Γ stands for the whole registrar's database program including the new clauses above.

3.2.2 Providing Scope to Clauses

In a hypothetical database query the implication is used to limit the scope, or availability, of a database fact to a particular query. But it can be used to scope more complicated clauses, or even sets of clauses. Suppose the clauses $\{s_1, \ldots, s_n\}$ together define some predicate. Then, in

[6] This is the first example of how an interpreter for λProlog must differ considerably from one for Prolog: it must be prepared to maintain the identities of unistantiated logic variables inside the program database.

order to prove the query `(G1, (s1,...,sn) => G2, G3)` the system must construct a proof of the form:

$$\dfrac{\Gamma \longrightarrow G_1 \quad \dfrac{\dfrac{\vdots}{\Gamma, (s_1 \wedge \ldots \wedge s_n) \longrightarrow G_2}}{\Gamma \longrightarrow (s_1 \wedge \ldots \wedge s_n) \supset G_2} \supset_R \quad \dfrac{\vdots}{\Gamma \longrightarrow G_3}}{\Gamma \longrightarrow [(s_1 \wedge \ldots \wedge s_n) \supset G_2] \wedge G_3} \wedge_R}{\Gamma \longrightarrow G_1 \wedge [(s_1 \wedge \ldots \wedge s_n) \supset G_2] \wedge G_3} \wedge_R$$

Thus the clauses $\{s_1, \ldots, s_n\}$ are available only during the proof of the subgoal G_2. This technique can be used to provide local, helper predicates which are available only in the clause in which they are defined.

Consider an implementation of `reverse/2` which uses a helper predicate `rev/3` to implement an efficient (i.e. non-naive) list reversal:

```
rev nil K K.
rev (X::N) M K :- rev N (X::M) K.

reverse L K :- rev L nil K.
```

Miller used this example to show the usefulness of scoped classes: [Miller, 1989*b*; Miller, 1990]:

```
reverse L K :-
  (
   rev nil K,
   forall X \ forall M \ forall N \ rev (X::N) M :- rev N (X::M)
  )
    => rev L nil.
```

In this version the clauses for `rev` are available only within `reverse`. Also notice that the logic variable `K` which will be used by `reverse` to return its output, is free in the definition of `rev`. Thus, in this version, `rev` can be defined as a binary, rather than ternary, predicate.

3.2.3 Providing Scope to Names

Implications in goals do not, however, provide all the scoping mechanisms programmers would like. Because Prolog predicate definitions do not have to occur as contiguous blocks, clauses introduced by an implication may interact with existing definitions.[7] Fortunately, universal quantifiers in goals provide a way of restricting the scope of names in much the same way that implications restrict the scope of clauses. Since the constant used to instantiate the

[7] Other proposals have been made for logically based scoping mechanisms that do not have this property, in particular the system of *Contextual Logic Programming* [Monteiro and Porto, 1989]. However there are times when such a feature is useful. In object-oriented logic programming it is useful to let a subclass add clauses to the definition of a predicate inherited from some other class [Hodas and Miller, 1990]. Therefore we prefer a system that provides this feature, but which also provides a way to guarantee true locality when that is desirable.

universally quantified variable in the $\forall_R$ rule cannot occur free in the lower sequent, the scope of the new name is restricted to the proof of the quantified goal.[8]

If quantifiers are allowed to range over objects of higher type, such as function and predicate names, then this scoping of names can be used to insure that assumed clauses do not interfere with existing definitions. Programs can then be truly modular. For instance, it is possible to rewrite the last example as follows:

```
reverse L K :- forall rev \ (
  (
   rev nil K,
   forall X \ forall M \ forall N \ rev (X::N) M :- rev N (X::M)
  )
    => rev L nil).
```

In this version, the universal quantification of `rev` insures that the two assumed clauses define a new predicate, available for use only within the definition of `reverse`.

3.2.4 A Modules System Based Entirely on Logic

The module system of λProlog demonstrates that it is possible to produce a rich module system that is nevertheless founded entirely on logical principles. The system incorporates the ideas discussed above as follows, the module definition:

```
MODULE mod  x1 ... xn.
LOCAL y1 ... ym.

D1 x1 ... xn y1 ... ym z11 ... zl1.
       .         .          .
       .         .          .
       .         .          .
Dp x1 ... xn y1 ... ym z1p ... zlp.
```

associates to `mod` the parameters $x_1 \ldots x_n$, the local constants $y_1 \ldots y_m$, and the clauses $H_1 \ldots H_p$, which may contain free occurrences of the variables $x_1 \ldots x_n$ and constants $y_1 \ldots y_m$, as well as clause-level variables $z_1 \ldots z_l$ (which are different for each clause). When the module is loaded within a goal formula, using the syntax (`mod` $t_1 \ldots t_n$ `==>` G), that goal is considered only as short-hand for the goal

```
forall y1 \ ( ...  forall ym\ (
     (forall z11 \ ( ...  forall zl1\ (
         D1 t1 ... tn y1 ... ym z11 ... zl1) ... ) =>
```

[8] This is a second way in which the λProlog interpreter differs from Prolog's. Because the constant cannot occur in the lower sequent, the system must insure that free logic variables in the lower sequent never become instantiated to terms containing the new constant. This requires an extension of the occurs check during unification.

```
MODULE sorter order.

LOCAL build traverse insert append empty tree.

append nil K K.
append (X::L) K (X::M) :- append L K M.

build nil Tr Tr.
build (H::Tl) Tr1 Tr3 :- insert H Tr1 Tr2, build Tl Tr2 Tr3.

insert V empty (tree V empty empty).
insert V (tree R Tr1 Tr2) Tr3 :- order V R,
                                 insert V Tr1 Tr1a,
                                 Tr3 = (tree R Tr1a Tr2).
insert V (tree R Tr1 Tr2) Tr3 :- insert V Tr2 Tr2a,
                                Tr3 = (tree R Tr1 Tr2a).

traverse empty nil.
traverse (tree R Tr1 Tr2) L :- traverse Tr1 L1, traverse Tr2 L2,
                               append L1 (R::L2) L.

sort L K :- build L empty T, traverse T K.
```

Figure 3.4: A Module Implementing a Sorting Predicate

$$
\begin{array}{l}
\qquad\vdots \\
\texttt{(forall } z_{1p} \texttt{ \textbackslash (... forall } z_{lp} \texttt{\textbackslash (} \\
\quad D_p\ t_1\ \texttt{...}\ t_n\ y_1\ \texttt{...}\ y_m\ z_{1p}\ \texttt{...}\ z_{lp}\texttt{) ...) => G) ...)}
\end{array}
$$

It is assumed (and enforced by the interpreter via renaming) that the formula G and the terms $t_1, \ldots, t_n$ do not contain free occurrences of $y_1, \ldots, y_m$.

Thus we might define a module which implements a sorting predicate as in Figure 3.4. The actual sort is done by building a binary search tree from the elements to be sorted, and then traversing it. The details of the implementation, however, are totally hidden. The `sort/2` predicate is the only name exported by the module. The module is also parameterized by the order in which the list is to be sorted. The following interaction demonstrates the module's use, and shows how the `LOCAL` definition blocks the use of the hidden names:

```
?- sorter '<' ==> sort (2::6::5::9::4::6::nil) L.
L == 2 :: 4 :: 5 :: 6 :: 6 :: 9 :: nil.
?- sorter '<' ==> append (1::2::nil) (3::4::nil) L.
no.
```

Modules Cum Objects

Several papers have suggested that, in the setting of positive Horn clauses, the proper view of objects is not as a pairing of state information with a set of behaviors, but rather simply as a set of behaviors. McCabe, for example, identified objects with parameterized modules of Prolog clauses [McCabe, 1989]. A goal that is used to "send a message" to an object is proved within the module corresponding to that object. The module system of λProlog supports a similar notion of object-sans-state by identifying them with abstract data types as outlined by Miller [Miller, 1989*a*]. For example, a module for an object representing a locomotive (based on one of McCabe's examples [McCabe, 1989, pages 47-54]) could be given by:

```
MODULE locomotive.
LOCAL train.

make_train (train S Cl Co) S Cl Co.

color (train S Cl Co) Cl.
speed (train S Cl Co) S.
country (train S Cl Co) Co.

journey_time Train Distance Time :-
       speed Train S, Time is Distance/S.
```

A query about a particular train, such as how long one that travels 30 miles per hour will take to go 100 miles, would then be posed as:

```
?- locomotive ==> exists Tr \ (
      make_train Tr 30 blue usa, journey_time Tr 100 Time).
Time == 3.333333.
```

The use of the explicit existential scoping of the name `Tr` is necessary as otherwise the system would be forced to draw the abstract-type constructor chosen for the local constant `train` out of the scope of the module in its attempt to print an answer substitution for`Tr`. This would cause the query to fail.

A point that should be made about this style of programming is that an object can just as easily be built via its selectors. A French train that travels 125 miles per hour could be built, and a trip time computed, by the following query:

```
?- locomotive ==> exists Tr \ (
      country L france, speed L 125, journey_time L 250 Time).
Time == 2.
```

This example also demonstrates the possibility of having partially described objects, where certain descriptors are left uninstantiated. This possibility (which is generally unavailable in traditional object-oriented languages) is useful in many applications.

3.2.5 Higher-Order Quantification

The use of higher-order quantification is an extremely important aspect of λProlog whose applications have been studied extensively. It does complicate the theory and implementation of the system considerably, however. For example, there is the question of exactly what formulas quantifiers should be allowed to range over. If the range is too wide, the completeness of uniform proofs could be compromised [Miller, 1990; Miller *et al.*, 1991]. Therefore some restrictions must be made.

In addition, once higher-order quantification is allowed, it is natural to introduce higher-order unification, which requires the use of λ-terms as data structures, since, for instance, the unification problem $(F\ a = g\ a\ b)$ has $(F = \lambda x.(g\ x\ b))$ as an answer. Unfortunately, this complicates the system further, since higher-order unification is undecidable, and even when it terminates may not have a most-general unifier. For instance, the vacuous abstraction $(F = \lambda x.(g\ a\ b))$ is an equally general answer for the last problem. The unification algorithm also requires that the term language of the system be strictly typed.

A discussion of all the theoretical and implementational issues involved with adopting the higher-order model is beyond the scope of this dissertation, and has been well covered by Nadathur and Miller [Nadathur, 1987; Miller *et al.*, 1991]. Most of these problems will not be of issue here, however, as we will use the higher-order quantifiers in very limited ways: to support scoping as described above, and continuation-passing as described below. Neither of these requires full higher-order unification, λ-terms, or types, and the system Lolli, which is the subject of the remainder of this dissertation, includes none of these.

Continuation-Passing Programming Style

One particularly useful application of predicate quantification is to develop a continuation-passing style of logic programming. With continuation-passing the ordinary flow of program control is modified by passing around explicit goals to be proved at a later point in the execution. Consider, for example, the clause:

```
pred G :- goal1, goal2, G.
```

When `pred` is called it will execute `goal1` and `goal2` in turn, and then execute whatever goal is passed in for `G`. Only when this goal has completed will `pred` return. Why is this desirable when it could just as easily be accomplished with the goal `(pred, G)` (where the body of `pred` is redefined to be just `(goal1, goal2)`)? In this case there is no particular difference. But if the clause is instead of the form:

```
pred G :- assumption => (goal1, goal2, G).
```

the result is quite different. The goal passed in for `G` is now executed in an augmented context. This makes it possible to stretch the scope of an implication without tearing it (into a general `assert/retract`).

This style of programming has proven extremely useful in λProlog, and will be used heavily in subsequent chapters. While it requires higher-order quantification, it does not in general

require the full higher-order unification algorithm as long as the continuation variable is instantiated by the time the goal G is called.

Memoizing Predicates

An interesting, and easy to understand use of this technique is for implementing memoized predicates. Consider the following definition of the predicate for computing Fibonacci numbers:

```
fib 0 0.
fib 1 1.
fib N F :- N1 is (N - 1), N2 is (N - 2),
           fib N1 F1,
           fib N2 F2,
           F is (F1 + F2).
```

The proof size for a query (fib *n* F) is clearly exponential in the value of *n*. This cost can be reduced to linear, however, if each time a given Fibonacci number is computed, its value is stored for later reference.[9] The system would thus be able to avoid unnecessary recomputation. This can be accomplished using continuation passing as follows:

```
fib N F :- memo 0 0  => memo 1 1  => fiba N F true.
fiba N F G :- memo N F, G.
fiba N F G :- N1 is (N - 1), N2 is (N - 2),
              fiba N1 F1
                  (fiba N2 F2
                       (F is (F1 + F2), memo N F => G)).
```

Here the bulk of the computation is performed by fiba/3 which is initially called with true as a continuation. This continuation is used to terminate the recursion when the computation is complete. Each time fiba is called, it first checks to see if this Fibonacci number has already been computed. If it has, then the current continuation is called. If it has not, then fiba is called recursively to compute (or look up) the previous Fibonacci number, with a continuation that calls fiba to compute (or look up) the second previous Fibonacci number, with, in turn, a continuation that adds the two numbers together and memorizes the result. Finally, the original continuation is called, within the scope of the newly memorized value.

Figure 3.5 presents a computation, using this program, which computes the fifth Fibonacci number, which is five. The proof has been divided into three parts for presentation purposes. Each piece of the proof fits into the piece below it at the point marked Ξ.

[9] Actually, the new version is linear only if we are only counting the number of additions performed. If we count the expense of search then the new version is quadratic, due to the cost of searching for memoized values. If, however, some system of indexing is used (even though these are dynamic predicates) then the complexity returns to being linear.

$$
\Xi_2 = \left\{
\dfrac{
\dfrac{\boxed{f_{12} = 1}}{\Gamma''' \longrightarrow m(2, f_{12})} \quad \dfrac{\boxed{f_1 = 2 + 1 = 3}}{\Gamma'' \longrightarrow is(f_1, (f_{11} + f_{12}))} \quad \dfrac{\overbrace{\Gamma''', m(4, f_1)}^{\Gamma''''} \longrightarrow f(3, f_2, [is(f_0, (f_1 + f_2)) \wedge m(5, f_0) \supset \top])}{\Gamma''' \longrightarrow m(4, f_1) \supset G_1} \supset R
}{
\dfrac{\Gamma''' \longrightarrow m(2, f_{12}) \wedge is(f_1, (f_{11} + f_{12}) \wedge (m(4, f_1) \supset G_1)}{\overbrace{\Gamma''', m(3, f_{11})}^{\Gamma'''} \longrightarrow f(2, f_{12}, [is(f_1, (f_{11} + f_{12})) \wedge m(4, f_1) \supset G_1])} BC
}{
\Gamma'' \longrightarrow m(3, f_1 1) \supset f(2, f_{12}, [is(f_1, (f_{11} + f_{12})) \wedge m(4, f_1) \supset G_1])
} \supset R
\right.
$$

$$
\left.
\dfrac{
\dfrac{\boxed{f_2 = 2}}{\Gamma'''' \longrightarrow m(3, f_2)} \quad \dfrac{\boxed{f_0 = 3 + 2 = 5}}{\Gamma'''' \longrightarrow is(f_0, (f_1 + f_2))} \quad \dfrac{\overline{\Gamma'''', m(5, f_0) \longrightarrow \top}}{\Gamma'''' \longrightarrow m(5, f_0) \supset \top} \supset R
}{
\dfrac{\Gamma'''' \longrightarrow m(3, f_2) \wedge is(f_0, (f_1 + f_2)) \wedge (m(5, f_0) \supset \top)}{\Gamma''''} BC
} \wedge R
\right\}
$$

$$
\Xi_1 = \left\{
\dfrac{
\dfrac{\boxed{f_{32} = 0}}{\Gamma' \longrightarrow m(0, f_{32})} \quad \dfrac{\boxed{f_{21} = 1 + 0 = 1}}{\Gamma' \longrightarrow is(f_{21}, (f_{31} + f_{32}))} \quad \dfrac{\overbrace{\Gamma', m(2, f_{21})}^{\Gamma''} \longrightarrow f(1, f_{22}, [is(f_{11}, (f_{21} + f_{22})) \wedge m(3, f_{11}) \supset G_2])}{\Gamma' \longrightarrow m(2, f_{21}) \supset G_3} \supset R
}{
\Gamma' \longrightarrow m(0, f_{32}) \wedge is(f_{21}, (f_{31} + f_{32})) \wedge (m(2, f_{21}) \supset G_3)
} \wedge R
\right.
$$

$$
\left.
\dfrac{
\dfrac{\boxed{f_{22} = 1}}{\Gamma'' \longrightarrow m(1, f_{22})} \quad \dfrac{\boxed{f_{11} = 1 + 1 = 2}}{\Gamma'' \longrightarrow is(f_{11}, (f_{21} + f_{22}))} \quad \dfrac{\Xi_2}{\Gamma'' \longrightarrow m(3, f_{11}) \supset G_2} \wedge R
}{
\dfrac{\Gamma'' \longrightarrow m(1, f_{22}) \wedge is(f_{11}, (f_{21} + f_{22})) \wedge (m(3, f_{11}) \supset G_2)}{\Gamma''} BC
}
\right\}
$$

$$
\dfrac{
\dfrac{\boxed{f_{31} = 1}}{\Gamma' \longrightarrow m(1, f_{31})} \quad \dfrac{\Xi_1}{\Gamma' \longrightarrow f(0, f_{32}, [is(f_{21}, (f_{31} + f_{32})) \wedge m(2, f_{21}) \supset G_3])} \wedge R
}{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\Gamma' \longrightarrow m(1, f_{31}) \wedge [f(0, f_{32}, [is(f_{21}, (f_{31} + f_{32})) \wedge m(2, f_{21}) \supset G_3])]}{\Gamma' \longrightarrow f(1, f_{31}, [f(0, f_{32}, [is(f_{21}, (f_{31} + f_{32})) \wedge m(2, f_{21}) \supset G_3])])} BC
}{\Gamma' \longrightarrow f(2, f_{21}, \overbrace{[f(1, f_{22}, [is(f_{11}, (f_{21} + f_{22})) \wedge m(3, f_{11}) \supset G_2])]}^{G_3})} BC
}{\Gamma' \longrightarrow f(3, f_{11}, \overbrace{[f(2, f_{12}, [is(f_1, (f_{11} + f_{12})) \wedge m(4, f_1) \supset G_1])]}^{G_2})} BC
}{\Gamma' \longrightarrow f(4, f_1, \overbrace{[f(3, f_2, [is(f_0, (f_1 + f_2)) \wedge m(5, f_0) \supset \top])]}^{G_1})} BC
}{
\dfrac{\overbrace{\Gamma, m(0,0), m(1,1)}^{\Gamma'} \longrightarrow f(5, f_0, \top)}{\Gamma \longrightarrow m(0,0) \supset m(1,1) \supset f(5, f_0, \top)} \supset R
}{
\Gamma \longrightarrow fib(f_0, 5)
} BC
$$

Figure 3.5: A proof representing computation with the memoized version of Fibonacci.

Miller used this same example to demonstrate the use of implications in goals [Miller, 1989*b*]. However, in order to avoid introducing the continuation-passing style, he gave the following linear memoized version:

```
fib(N,M) :- memo(0,0) => memo(1,1) => fiba(N,M,2).
fiba(N,M,N) :- memo(N,M).
fiba(N,M,I) :- N1 is (I-1), N2 is (N-2), memo(N1,F1),
               memo(N2,F2), F is (F1+F2), I1 is (I+1),
               memo(I,F) => fiba(N,M,I1).
```

However this version is linear not because of the memoization, but rather because Miller has replaced the original algorithm with a tail-recursive one which builds the Fibonacci series from the bottom. This could have been accomplished without memoization with the ordinary Prolog code:

```
fib(0,0).
fib(1,1).
fib(N,M) :- fiba(N,M,1,0,1).
fiba(N,M,N,M,K).
fiba(N,M,I,FI,F2) :- I1 is I+1, F is F1 + F2, fiba(N,M,I1,F,F1).
```

The advantage of using continuation-passing to implement memoization is that it is a general transformation that does not require redesigning the original algorithm to yield the same benefits.

Chapter 4

Towards a Linear Logic Programming Language

4.1 The Limitations of Intuitionistic Logic

While the scoping facilities provided by implications and quantification in goals have many uses, they are still relatively limited. In particular, once a formula is added to the proof context, there is no way to control how many times it is used, or whether it is used at all. This lack of control over *multiplicity* and *relevance* can be a serious drawback when reasoning in many domains. Consider the clauses:

```
can_buy soda  :- have dollar.
can_buy candy :- have dollar.
```

These seem to be perfectly reasonable assertions. Unfortunately, they lead to the following interaction:

```
?- have dollar => (can_buy soda , can_buy candy).
yes.
```

which constructs the following proof:

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma, have(dollar) \longrightarrow have(dollar)}}{\Gamma, have(dollar) \longrightarrow can\_buy(soda)}\ BC \quad \dfrac{\overline{\Gamma, have(dollar) \longrightarrow have(dollar)}}{\Gamma, have(dollar) \longrightarrow can\_buy(candy)}\ BC}{\Gamma, have(dollar) \longrightarrow (can\_buy(soda) \wedge can\_buy(candy))}\ \wedge_R}{\Gamma \longrightarrow have(dollar) \supset (can\_buy(soda) \wedge can\_buy(candy))}\ \supset_R$$

the problem here is that the implicit contraction in the $\wedge_R$ rule means that one dollar can be used over and over to buy whatever the user desires.

Another way of looking at the multiplicity problem is that there is no way to remove formulae from the proof context within the subproof in which they are added. So, for instance,

in a database program there is no way to update values in the database; new facts can only be added. As another example, suppose we wish to go a step further towards object-oriented programming than in the last section by allowing objects to have mutable state represented by clauses stored in the database. This might be implemented using continuation-passing, as in the following module for a class of toggle switches:

```
MODULE switch name initial.
LOCAL sw.

sw initial.

setting name S :- sw S.

toggle name G :- sw on, sw off => G.
toggle name G :- sw off, sw on => G.
```

Unfortunately, the core of the proof of the goal `(switch s1 on) ==> (toggle s1 (setting s1 S))`, which asks what the value of the switch `s1` (which is initially set to `on`) is after the switch is toggled, proceeds as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, sw(s_1, on) \longrightarrow sw(s_1, on)} \quad \dfrac{\dfrac{\dfrac{\boxed{s = on} \text{ or } \boxed{s = off}}{\Gamma, sw(s_1, on), sw(s_1, off) \longrightarrow sw(s_1, s)}}{\Gamma, sw(s_1, on), sw(s_1, off) \longrightarrow setting(s_1, s)}\, BC}{\Gamma, sw(s_1, on) \longrightarrow sw(s_1, off) \supset setting(s_1, s)}\, \supset_R}{\Gamma, sw(s_1, on) \longrightarrow sw(s_1, on) \wedge (sw(s_1, off) \supset setting(s_1, s))}\, \wedge_R}{\Gamma, sw(s_1, on) \longrightarrow toggle(s_1, setting(s_1, s))}\, BC$$

Thus, because there is no way to retract the initial setting of the switch, toggling it causes it to have two values.

The problem of relevance is an issue in an interaction like:

```
?- state zone1 factory => state zone2 polluted.
yes.
```

The user is likely to assume that `zone2` becomes polluted because someone builds a toxic waste dump at nearby `zone1`. It is possible, however, that `zone2` was polluted already. Because extraneous facts can be discarded (by implicit weakening) at the leaves of a proof, there is no way of requiring that assumptions be relevant to the conclusion.

4.2 A Brief Introduction to Linear Logic

Linear logic, first proposed by Jean-Yves Girard in 1987 [Girard, 1987*a*], has been called "the computational logic behind logic" [Kanovich, 1993], because it is a logic in which the pattern of formula usage within a proof is of central importance. This focus is obtained by deleting (or,

more precisely, by restricting the use of) the structural rules of contraction and weakening. By controlling the use of these rules, the logic imposes constraints on when and how many times a formula can be appealed to in a proof.

An important effect of restricting the use of contraction and weakening is that there is a small explosion in the number of connectives in the logic. Consider two options for presenting the rule for conjunction in the succedent:

$$\frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge_R \qquad \frac{\Gamma \longrightarrow B \quad \Delta \longrightarrow C}{\Gamma, \Delta \longrightarrow B \wedge C}\ \wedge_R{}'$$

The first says that if when Γ is true both B and C are true, then when Γ is true $B \wedge C$ is true. The second says that if B is true when Γ is true and if C is true when Δ is true, then when everything in the multiset union $\Gamma \uplus \Delta$ is true, $B \wedge C$ is true. Each of these is a sound inference, and, in the presence of contraction and weakening, they are equivalent. This can be seen in the following transformations:

$$\frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge_R \quad \rightsquigarrow \quad \frac{\dfrac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma, \Gamma \longrightarrow B \wedge C}\ \wedge_R{}'}{\Gamma \longrightarrow B \wedge C}\ C^*$$

$$\frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma, \Delta \longrightarrow B \wedge C}\ \wedge_R{}' \quad \rightsquigarrow \quad \frac{\dfrac{\Gamma \longrightarrow B}{\Gamma, \Delta \longrightarrow B}\ W^* \quad \dfrac{\Delta \longrightarrow C}{\Gamma, \Delta \longrightarrow C}\ W^*}{\Gamma, \Delta \longrightarrow B \wedge C}\ \wedge_R$$

If, however, the structural rules are removed, then these rules represent different types of conjunction. In fact, most of the connectives split into two versions in linear logic; there are two conjunctions, '$\otimes$' (called "tensor") and '&' (called "with"), two disjunctions, '$⅋$' (called "par") and '$\oplus$' (called "o-plus"), two truths, '$\top$' (called "top") and '**1**' (called "one"), and two falsehoods, '$\bot$' (called "bottom") and '**0**' (called "zero"). The implication operator is written '$\multimap$' and is called "lollipop", and the negation of A is written '$A^{\bot}$' and is called "A-perp".

Totally removing the structural rules would lead to a system with limited expressive power since each formula could only be used exactly once in each branch of a proof.[1] Therefore, those rules are reintroduced, but in a modified form. They can only be applied to formulae that have been marked with a special modal connective written '!' and called "of course" or "bang".[2] In this way certain formulae can be treated as reusable or ignorable, while others can only be used exactly once.

As with traditional systems, linear logic has been studied both in classical (multiple-conclusion) and intuitionistic (single-conclusion) forms. As we will, in the spirit of the last two

[1] Lincoln, et. al., have shown that the strictly linear fragment of the logic (called multiplicative-additive linear logic, or MALL) is PSPACE-complete [Lincoln *et al.*, 1990].

[2] "Of course" is the original name of the modal (its dual, which will not be used in this paper, is written '?' and called "why not"), however, the name "bang" has spread from the University of Pennsylvania (where it was adopted from the same name used in programming languages) into fairly wide use. The name "lollipop" for linear implication also seems to have originated at the University of Pennsylvania.

$$\frac{}{B \longrightarrow B}\ \textit{identity} \qquad \frac{}{\Gamma, \mathbf{0} \longrightarrow B}\ \mathbf{0}_L \qquad \frac{}{\longrightarrow \mathbf{1}}\ \mathbf{1}_R \qquad \frac{}{\Delta \longrightarrow \top}\ \top_R$$

$$\frac{\Gamma \longrightarrow B}{\Gamma, \mathbf{1} \longrightarrow B}\ \mathbf{1}_L \qquad \frac{\Gamma \longrightarrow C \quad \Delta, C \longrightarrow B}{\Gamma, \Delta \longrightarrow B}\ \textit{cut}$$

$$\frac{\Gamma, B, C \longrightarrow E}{\Gamma, B \otimes C \longrightarrow E}\ \otimes_L \qquad \frac{\Gamma \longrightarrow B \quad \Delta \longrightarrow C}{\Gamma, \Delta \longrightarrow B \otimes C}\ \otimes_R$$

$$\frac{\Gamma, B \longrightarrow E}{\Gamma, B \,\&\, C \longrightarrow E}\ \&_{L_1} \qquad \frac{\Gamma, C \longrightarrow E}{\Gamma, B \,\&\, C \longrightarrow E}\ \&_{L_2} \qquad \frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \,\&\, C}\ \&_R$$

$$\frac{\Gamma, B \longrightarrow E \quad \Gamma, C \longrightarrow E}{\Gamma, B \oplus C \longrightarrow E}\ \oplus_L \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow B \oplus C}\ \oplus_{R_1} \qquad \frac{\Gamma \longrightarrow C}{\Gamma \longrightarrow B \oplus C}\ \oplus_{R_2}$$

$$\frac{\Gamma \longrightarrow B \quad \Delta, C \longrightarrow E}{\Gamma, \Delta, B \multimap C \longrightarrow E}\ \multimap_L \qquad \frac{\Gamma, B \longrightarrow C}{\Gamma \longrightarrow B \multimap C}\ \multimap_R$$

$$\frac{\Gamma, !B, !B \longrightarrow C}{\Gamma, !B \longrightarrow C}\ !_C \qquad \frac{\Gamma \longrightarrow C}{\Gamma, !B \longrightarrow C}\ !_W \qquad \frac{\Gamma, B \longrightarrow C}{\Gamma, !B \longrightarrow C}\ !_L \qquad \frac{!\Gamma \longrightarrow B}{!\Gamma \longrightarrow !B}\ !_R$$

$$\frac{\Gamma, B[x \mapsto t] \longrightarrow C}{\Gamma, \forall x.B \longrightarrow C}\ \forall_L \qquad \frac{\Gamma \longrightarrow B[x \mapsto t]}{\Gamma \longrightarrow \exists x.B}\ \exists_R$$

$$\frac{\Gamma, B[x \mapsto c] \longrightarrow C}{\Gamma, \exists x.B \longrightarrow C}\ \exists_L \qquad \frac{\Gamma \longrightarrow B[x \mapsto c]}{\Gamma \longrightarrow \forall x.B}\ \forall_R$$

provided, in each case, c does not appear free in the conclusion

Figure 4.1: The Proof System $\mathcal{ILL}$ for Intuitionistic Linear Logic

chapters, be interested only in cut-free uniform proofs, our focus will be on the intuitionistic variant, the rules for which are given in Figure 4.1. Note that this variation of the system (which is taken from Schellinx [Schellinx, 1991]) does not include the connectives ⅋, ?, or ⊥. In the $!_R$ rule, the notation '!Γ' refers to the multiset $\{!\,C \mid C \in \Gamma\}$. This rule can be read much like the right rule for the modal in S_4. It says that anything that is provable from a context in which all the formulae can be used arbitrarily many times can itself be used arbitrarily many times.

What is the meaning of all these new connectives? Linear implication can be seen as a refinement of intuitionistic implication. If we take the Curry-Howard isomorphism viewpoint, that implication corresponds to function construction, then the intuitionistic implication $A \supset B$ corresponds to the function that converts an A to a B. Linear implication simply refines this: $A \multimap B$ corresponds to a function that converts an A to a B while only referring to the A exactly once in the process. The "of course" connective grants permission to refer to a given input more than once, or to simply discard it. Thus (as Girard has shown) $A \supset B$ corresponds to $!A \multimap B$.

The two forms of conjunction can best be explained by example. Consider a restaurant dinner special where for five dollars you can have an appetizer, a sandwich, and a dessert. If there are two appetizers, three sandwiches, and two deserts to choose from, this offer can be

represented by the formula:

$$\$ \otimes \$ \otimes \$ \otimes \$ \otimes \$ \multimap [(a_1 \,\&\, a_2) \otimes (s_1 \,\&\, s_2 \,\&\, s_2) \otimes (d_1 \,\&\, d_2)]$$

Intuition may lead one to think that the choice for each course should be represented by a disjunction, since we would normally say "You can have the burger, or the rueben, or the club." But logical disjunction corresponds to a choice beyond our control. When $A \vee B$ is true, we only know (in the intuitionistic case) that at least one of A or B is true, but not which. In this case, each one of the choices for a given course is acceptable and must therefore be independently true. It is only that we cannot have all of them at once. Thus the additive conjunction, '&', has a certain disjunctive, or exclusionary, flavor to it.

Suppose the restaurant decides they need to get rid of some unpopular vegetables, so the offer is improved by adding a side-dish vegetable (one of three, but they make the choice) and unlimited coffee. The new version of the offer would then be represented by the formula:

$$\$ \otimes \$ \otimes \$ \otimes \$ \otimes \$ \multimap [(a_1 \,\&\, a_2) \otimes (s_1 \,\&\, s_2 \,\&\, s_3) \otimes (v_1 \oplus v_2 \oplus v_3) \otimes (d_1 \,\&\, d_2) \otimes (!c)]$$

4.3 Designing an Abstract Linear-Logic Programming Language

Girard has given a sound and complete mapping of intuitionistic logic into linear logic which is faithful not only to provability but to proofs. Therefore, since we have shown that full intuitionistic logic is not an abstract logic programming language, it is obvious that intuitionistic linear logic is not as well. Unfortunately, the degree to which uniform proofs fail to be complete is far more severe in $\mathcal{ILL}$ than in $\mathcal{LJ}'$. Whereas negative occurrences of disjunction and existential were the only roadblock to uniform proofs in $\mathcal{LJ}'$, additional connectives cause problems in $\mathcal{ILL}$. For example, the sequents $a \otimes b \longrightarrow a \otimes b$, $!a \longrightarrow !a \otimes !a$, $!a \,\&\, b \longrightarrow !a$, and $b \otimes (b \multimap !a) \longrightarrow !a$ are all provable in linear logic but do not have uniform proofs.

Most of the problems derive from the impermutability of the left-hand rules for the modal '!' with the right-hand rules for '!' and '⊗'. In $\mathcal{LJ}$ many of these same problems would occur, were it not possible to make the structural rules implicit, as they were in $\mathcal{LJ}'$. Can the left-hand ! rules be made implicit in the same way? To that end we consider the system $\mathcal{L}$ shown in Figure 4.2. In this system the antecedents of sequents have two parts, separated by a semicolon, ';'. Each compartment of the antecedent is a multiset of formulae, as before. But, formulae in the left compartment (which will be called the *unbounded* or *intuitionistic context*) are treated as though they carried the '!' modal, while those in the right compartment (called the *bounded* or *linear context*) are treated as usual. In particular, as will be shown below, the $\mathcal{L}$ sequent:

$$B_1, \ldots, B_n; C_1, \ldots, C_m \longrightarrow B$$

can be mapped to the linear logic sequent:

$$!B_1, \ldots, !B_n, C_1, \ldots, C_m \longrightarrow B.$$

$$\frac{}{\Gamma; B \longrightarrow B}\ \textit{identity} \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}\ \top_R \qquad \frac{}{\Gamma; \varnothing \longrightarrow \mathbf{1}}\ \mathbf{1}_R$$

$$\frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ \textit{absorb}$$

$$\frac{\Gamma; \Delta, B_1 \longrightarrow C}{\Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&_{L_1} \qquad \frac{\Gamma; \Delta, B_2 \longrightarrow C}{\Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&_{L_2} \qquad \frac{\Gamma; \Delta \longrightarrow B \quad \Gamma; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \,\&\, C}\ \&_R$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \quad \Gamma; \Delta_2, C \longrightarrow E}{\Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap_L \qquad \frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}\ \multimap_R$$

$$\frac{\Gamma; \varnothing \longrightarrow B \quad \Gamma; \Delta, C \longrightarrow E}{\Gamma; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow_L \qquad \frac{\Gamma, B; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow_R$$

$$\frac{\Gamma; \Delta, B[x \mapsto t] \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall_L \qquad \frac{\Gamma; \Delta \longrightarrow B[x \mapsto c]}{\Gamma; \Delta \longrightarrow \forall x.B}\ \forall_R$$

provided that c does not appear free in the lower sequent.

$$\frac{\Gamma; \varnothing \longrightarrow C}{\Gamma; \varnothing \longrightarrow !C}\ !R \qquad \frac{\Gamma; \Delta_1 \longrightarrow B_1 \quad \Gamma; \Delta_2 \longrightarrow B_2}{\Gamma; \Delta_1, \Delta_2 \longrightarrow B_1 \otimes B_2}\ \otimes R$$

$$\frac{\Gamma; \Delta \longrightarrow B[x \mapsto t]}{\Gamma; \Delta \longrightarrow \exists x.B}\ \exists_R \qquad \frac{\Gamma; \Delta \longrightarrow B_1}{\Gamma; \Delta \longrightarrow B_1 \oplus B_2}\ \oplus_{R_1} \qquad \frac{\Gamma; \Delta \longrightarrow B_2}{\Gamma; \Delta \longrightarrow B_1 \oplus B_2}\ \oplus_{R_2}$$

$$\frac{\Gamma_1; \Delta_1 \longrightarrow B \quad \Gamma_2; \Delta_2, B \longrightarrow C}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \longrightarrow C}\ \textit{cut} \qquad \frac{\Gamma_1; \varnothing \longrightarrow B \quad \Gamma_2, B; \Delta \longrightarrow C}{\Gamma_1, \Gamma_2; \Delta \longrightarrow C}\ \textit{cut}!$$

Figure 4.2: $\mathcal{L}$: A proof system for the connectives '$\top$', '&', '$\multimap$', '$\Rightarrow$', '$\forall$','**1**', '!', '$\otimes$', '$\oplus$', and '$\exists$'.

In linear logic, the left-hand operator introduction rules (other than $!_L$) act on un-!'ed formulae. In $\mathcal{L}$ this is mimicked by the fact that all of the left-hand introduction rules operate only on formulae in the bounded context. In order to simulate the dereliction rule, $!_L$, which is used to make a !'ed formula available for such use, the system $\mathcal{L}$ includes a rule called *absorb*, which copies a formula from the unbounded context to the bounded one. It also includes an implicit contraction of the formula, so that it will be available as needed for repeated use.[3]

Given this style of sequent, it is necessary to have two kinds of implications: the linear implication, for which the right-introduction rule adds its assumption to the bounded context, and the intuitionistic implication (written '$\Rightarrow$'), for which the right-introduction rule adds its assumption to the unbounded context. Of course, the intended meaning of $B \Rightarrow C$ is $(!B) \multimap C$. This equivalence is apparent in the formulation of the $\Rightarrow_L$ rule, in which the body of the selected clause must be proved in a context in which the bounded context is empty. This corresponds to the restriction on the form of the proof context in the $!_R$ rule of $\mathcal{ILL}$.

[3]This system has much in common with the system of "focusing proofs" which was developed by Andreoli. The goal of that work was to reduce the degree of enforced sequentiality in multiple-conclusion linear-logic proofs, to aid in the design of concurrent logic-programming languages based on linear logic [Andreoli, 1992]. Since sequentiality and permutability are inevitably interlinked in the same way that uniformity and permutability are, it is no surprise that the two results are similar. The two systems were, however, developed independently, though we have adopted Andreoli's use of the name *absorb* for that rule, which occurs in his system as well.

This system has been designed to support logic programming over the following two family of formulae:

Definition 4.1 The set of *goal formulae* is the smallest set satisfying the following conditions:

- $\top$, $\mathbf{1}$, and all atomic formulae A are goal formulae.
- If G_1 and G_2 are goal formulae, then $G_1 \otimes G_2$, $G_1 \,\&\, G2$, and $G_1 \oplus G_2$ are goal formulae.
- If G is a goal formula, then $!G$ is a goal formula.
- If G is a goal formula, then $\exists x.G$ and $\forall x.G$ are goal formulae.
- If G is a goal formula and D is a definite clause (as defined below), then $D \multimap G$ and $D \Rightarrow G$ are goal formulae.

Definition 4.2 The set of *definite formulae* or *program clauses* or *linear hereditary Harrop formulae* is the smallest set satisfying the following conditions:

- $\top$, and all atomic formulae A are definite formulae.
- If D_1 and D_2 are definite formulae, then $D_1 \,\&\, D_2$ is a definite formula.
- If D is a definite formula, then $\forall x.D$ is a definite formula.
- If G is a goal formula (as defined above) and D is a definite formula, then $G \multimap D$ and $G \Rightarrow D$ are definite formulae.

It is also possible to view the language as being freely generated by $\top$, &, $\multimap$, $\Rightarrow$, and $\forall$ with the addition of negative occurances of $\otimes$, !, $\exists$, and $\oplus$.

The formal relationship between $\mathcal{L}$ and $\mathcal{ILL}$ can now be stated:

Proposition 4.1 Let G be a goal formula, and Γ and Δ multisets of definite formulae as defined above. Let B° be the result of repeatedly replacing all occurrences of $C_1 \Rightarrow C_2$ in B with $(!C_1) \multimap C_2$, and let $\Gamma^\circ = \{C^\circ \mid C \in \Gamma\}$. Then the sequent $\Gamma; \Delta \longrightarrow G$ is $\mathcal{L}$-provable if and only if the sequent $!(\Gamma^\circ), \Delta^\circ \longrightarrow G^\circ$ is $\mathcal{ILL}$-provable.

Proof. The proof in each direction can be shown by presenting a simple transformation between proofs in these two proof systems. The complete proof is given in Appendix C. ∎

Note that the system $\mathcal{L}$ has two different cut rules, one for cutting a formula into each of the two compartments of the antecedent. It is easy to argue for the admissibility of these rules since we know by the last proposition that sequents in this system are provable if and only if related sequents are provable in $\mathcal{ILL}$, and these cut rules map to instances of the cut rule for $\mathcal{ILL}$, which Girard has shown to be admissible in that system. However, such an argument provides little of the computational insight that can often be gained from a direct proof of cut elimination. Therefore a direct proof of cut elimination for the system $\mathcal{L}$ is given

$$\frac{\Gamma;\varnothing \longrightarrow G_1 \quad \cdots \quad \Gamma;\varnothing \longrightarrow G_n \quad \Gamma;\Delta_1 \longrightarrow G'_1 \quad \cdots \quad \Gamma;\Delta_m \longrightarrow G'_m}{\Gamma;\Delta_1,\ldots,\Delta_m,D \longrightarrow A}\ BC$$

provided $n, m \geq 0$, A is atomic, and $\langle\{G_1,\ldots,G_n\},\{G'_1,\ldots,G'_m\},A\rangle \in \|D\|$.

Figure 4.3: Backchaining for the proof system $\mathcal{L}'$.

in Appendix D.[4] Once restricted to the domain of cut-free proofs, the next step, as before, is to show that goal-directed provability is complete for this new system.

Proposition 4.2 Let B be a goal formula, and let Γ and Δ be multisets of definite formulae. Then the sequent $\Gamma;\Delta \longrightarrow B$ has a proof in $\mathcal{L}$ if and only if it has a uniform proof in $\mathcal{L}$.

Proof. The complete proof is given in Appendix E. ∎

It follows immediately from Proposition 4.2 that if D and G are the sets of all goal and definite clauses respectively, then the triple $\langle D, G, \vdash_{\mathcal{L}}\rangle$ is an abstract logic programming language (where that definition is appropriately modified to take into account the new style of sequent).

As with system $\mathcal{HH}'$ it is possible to restrict uniform proofs even further in the sense that the left-hand rules can be restricted to a form of backchaining. Consider the following definition: let D be a definite formula, then $\|D\|$ is the smallest set of triples of the form $\langle\Gamma,\Delta,D'\rangle$ where Γ and Δ are multisets of goal formulae, such that:

1. $\langle\varnothing,\varnothing,D\rangle \in \|D\|$,
2. if $\langle\Gamma,\Delta,D_1 \,\&\, D_2\rangle \in \|D\|$ then both $\langle\Gamma,\Delta,D_1\rangle \in \|D\|$ and $\langle\Gamma,\Delta,D_2\rangle \in \|D\|$,
3. if $\langle\Gamma,\Delta,\forall x.D'\rangle \in \|D\|$ then for all closed terms t, $\langle\Gamma,\Delta,D'[x \mapsto t]\rangle \in \|D\|$,
4. if $\langle\Gamma,\Delta,G \Rightarrow D'\rangle \in \|D\|$ then $\langle\Gamma \uplus \{G\},\Delta,D'\rangle \in \|D\|$, and
5. if $\langle\Gamma,\Delta,G \multimap D'\rangle \in \|D\|$ then $\langle\Gamma,\Delta \uplus \{G\},D'\rangle \in \|D\|$.

Let $\mathcal{L}'$ be the proof system that results from replacing the *identity*, $\multimap_L$, $\Rightarrow_L$, $\&_L$, and $\forall_L$ rules in Figure 4.2 with the *backchaining* inference rule in Figure 4.3.

Proposition 4.3 Let G be a goal formula, and let Γ and Δ be multisets of definite formulae. The sequent $\Gamma;\Delta \longrightarrow G$ has a proof in $\mathcal{L}$ if and only if it has a proof in $\mathcal{L}'$.

Proof. (Due to Miller.) In the reverse direction it is easy to show that each occurrence of a *BC* rule in $\mathcal{L}'$ can be converted to (possibly) several occurrences of the $\&_L$, $\multimap_L$, $\Rightarrow_L$, $\forall_L$, and identity rules in $\mathcal{L}$. The proof in the forward direction is more involved, and is due to Miller.

[4] Of course, cut will only be eliminable from the core system in which all formulae are freely generated by the connectives $\top$, &, $\multimap$, $\Rightarrow$, and $\forall$. The additional connectives which occur only in negative positions are considered only once we restrict our view to cut-free proofs.

Let Ξ be an $\mathcal{L}$-proof of $\Gamma;\Delta \longrightarrow B$. Mark certain occurrences of formulae in the bounded part of some sequents in Ξ as follows. The atomic formula (there is exactly one) in the bounded context of every instance of *identity* is marked. By referring to Figure 4.2 we then mark additional formula occurrences using induction on the structure of proofs as follows:

- If the B_i formula occurrence in the $\&_{L_i}$ rule is marked, then mark the occurrence of $B_1 \& B_2$ in its conclusion.
- If the $B[x \mapsto t]$ formula occurrence in the $\forall_L$ rule is marked, then mark the occurrence of $\forall x.B$ in its conclusion.
- if the C formula occurrence in the right-hand premise of the $\multimap_L$ rule is marked, then mark the occurrence of $B \multimap C$ in its conclusion
- if the C formula occurrence in the right-hand premise of the $\Rightarrow_L$ rule is marked, then mark the occurrence of $B \Rightarrow C$ in its conclusion.

We now make the following definition, as in [Miller, 1989*b*]:

Definition 4.3 An occurrence of a left-introduction rule is *simple* if the occurrence of the formula containing the logical connective introduced is marked. A uniform proof in which all occurrences of left-introduction rules are simple is called a *simple proof*.

Now observe two facts about simple proofs. First, if Ξ is simple, then Ξ can be transformed directly into an $\mathcal{L}'$-proof: simply collapse all chains of left-introduction rules (following the marking process) into one *BC* inference rule. Second, by permuting inference rules, any uniform $\mathcal{L}$-proof can be transformed into a simple proof. The proof of this is similar to the proof of Proposition 4.2. Find a non-simple occurrence of a left-introduction rule for which the subproofs of its premise(s) are simple proofs. One of the premises of this non-simple occurrence of a left-introduction rule must also be a left-introduction rule. Permute these two left-introduction rules. Consider, for example, the following case where these two left-introduction rules are $\multimap_L$.

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B_1} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2, B_2 \longrightarrow C_1} \quad \overset{\Xi_3}{\Gamma;\Delta_3, C_2 \longrightarrow E}}{\Gamma;\Delta_2,\Delta_3, B_2, C_1 \multimap C_2 \longrightarrow E}\ \multimap_L}{\Gamma;\Delta_1,\Delta_2,\Delta_3, B_1 \multimap B_2, C_1 \multimap C_2 \longrightarrow E}\ \multimap_L$$

Here we assume Ξ_1, Ξ_2, and Ξ_3 are simple proofs. This proof structure is then converted to the following proof by permuting these two inference rule occurrences.

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B_1} \quad \overset{\Xi_2}{\Gamma;\Delta_2, B_2 \longrightarrow C_1}}{\Gamma;\Delta_1,\Delta_2, B_1 \multimap B_2 \longrightarrow C_1}\ \multimap_L \quad \overset{\Xi_3}{\Gamma;\Delta_3, C_2 \longrightarrow E}}{\Gamma;\Delta_1,\Delta_2,\Delta_3, B_1 \multimap B_2, C_1 \multimap C_2 \longrightarrow E}\ \multimap_L$$

It may be necessary to continue permuting inference rules in this fashion since, in this case, the subproof of the sequent $\Gamma;\Delta_1,\Delta_2, B_1 \multimap B_2 \longrightarrow C_1$ may not be simple. The result of continuing

this process is then a simple proof of the sequent $\Gamma; \Delta_1, \Delta_2, \Delta_3, B_1 \multimap B_2, C_1 \multimap C_2 \longrightarrow E$. In this way, all non-simple occurrences of left-introduction rules can be eliminated, giving rise to a simple proof, which can, in turn, be converted to an $\mathcal{L}'$ proof. ■

Note that, unlike the system $\mathcal{HH}'$, proofs in $\mathcal{L}'$ are not necessarily uniform due to the presence of the *absorb* rule, which may act on sequents with non-atomic right-hand sides. Nevertheless, it is easy to see that all uses of this rule can be pushed up the tree so that they occur only immediately below instances of *BC*. Such $\mathcal{L}'$ proofs are then uniform.

4.4 An Embedding of the System $\mathcal{HH}'$ into $\mathcal{L}'$

As stated earlier, Girard has presented a mapping of intuitionistic logic into linear logic that preserves not only provability but also proofs [Girard, 1987*a*]. On the fragment of intuitionistic logic containing $\top$, $\wedge$, $\supset$, $\vee$, $\exists$, and $\forall$, the translation is given by:

$$
\begin{aligned}
A^* &= A, \text{ where } A \text{ is atomic} \\
\top^* &= \mathbf{1} \\
(B_1 \wedge B_2)^* &= (B_1^*) \,\&\, (B_2^*) \\
(B_1 \vee B_2)^* &= (B_1^*) \oplus (B_2^*) \\
(B_1 \supset B_2)^* &= !(B_1^*) \multimap (B_2^*) \\
(\forall x.B)^* &= \forall x.(B^*) \\
(\exists x.B)^* &= \exists x.(B^*)
\end{aligned}
$$

However, if we are willing to focus attention on only cut-free proofs in $\mathcal{HH}'$ and $\mathcal{L}'$, it is possible to define a "tighter" translation. Consider the following two mutually recursive translation functions:

$$
\begin{aligned}
A^+ = A^- &= A, \text{ where } A \text{ is atomic} \\
\top^+ &= \mathbf{1} \\
\top^- &= \top \\
(B_1 \wedge B_2)^+ &= (B_1^+) \otimes (B_2^+) \\
(B_1 \wedge B_2)^- &= (B_1^-) \,\&\, (B_2^-) \\
(B_1 \supset B_2)^+ &= (B_1^-) \Rightarrow (B_2^+) \\
(B_1 \supset B_2)^- &= (B_1^+) \multimap (B_2^-) \\
(\forall x.B)^+ &= \forall x.(B^+) \\
(\forall x.B)^- &= \forall x.(B^-) \\
(B_1 \vee B_2)^+ &= (B_1^+) \oplus (B_2^+) \\
(\exists x.B)^+ &= \exists x.(B^+)
\end{aligned}
$$

Proposition 4.4 Let G be a goal formula and Γ be a multiset of definite formulae as defined in Definitions 3.1 and 3.2 —that is let them be as defined for the system $\mathcal{HH}'$. Then, if $\Gamma^- = \{D^- \mid D \in \Gamma\}$, then the sequent $\Gamma \longrightarrow G$ has a proof in $\mathcal{HH}'$ if and only if the sequent $\Gamma^-; \varnothing \longrightarrow G^+$ has a proof in $\mathcal{L}'$. Furthermore, there is an isomorphism of such proofs.

Proof. The proof is by providing a transformation of proofs from each system into the other. In the forwards direction, let Ξ be an $\mathcal{HH}'$-proof of $\Gamma \longrightarrow G$. This proof can be converted to a proof Ξ^* of $\Gamma^-; \varnothing \longrightarrow G^+$ by replacing the inference rules $\top_R$, $\wedge_R$, $\vee_R$, $\supset_R$, $\forall_R$, and $\exists_R$ with the $\mathcal{L}'$ inference rules $\mathbf{1}_R$, $\otimes_R$, $\oplus_R$, $\Rightarrow_R$, $\forall_R$, and $\exists_R$. Instances of the *BC* rule of $\mathcal{HH}'$ need to be converted to *BC* paired with absorb in $\mathcal{L}'$. That this is possible is apparent since if D is a hereditary Harrop formula $\langle \Delta, A \rangle \in |D|$ if and only if $\langle \varnothing, \Delta^+, A \rangle \in \|D^-\|$.

For the converse, let Ξ^* be an $\mathcal{L}'$-proof of $\Gamma^-; \varnothing \longrightarrow G^+$. As was mentioned in the last section, we can assume that the only occurrences of the absorb rule are such that their premise is the conclusion of an instance of the *BC* rule. Such a proof can be converted to a proof in $\mathcal{L}'$ by reversing the conversion given for the first case. ■

Notice that in the construction in $\mathcal{L}'$ of uniform proofs of sequents translated from intuitionistic logic, the bounded part of sequents is non-empty only when a clause is moved into it so that the backchaining rule can be applied. If we relate this observation to the construction of normal λ-terms, where backchaining corresponds to the application of a typed constant or variable, we draw the obvious conclusion that there is exactly one head symbol for normal λ-terms.

4.5 The Concrete Syntax of the Linear-Logic Programming Language Lolli

Given the λProlog syntax for hereditary Harrop formula programs, the mapping given in the last section suggests a concrete syntax for the programming language Lolli which is built on the connectives of $\mathcal{L}'$. The syntax, which is summarized in the table in Figure 4.4, is designed so that conventional Prolog and λProlog programs remain unchanged both syntactically and behaviorally when used in this new setting.[5] This section will summarize the core syntax of Lolli. Chapter 9 contains a more detailed discussion of the language as implemented, including a description of all the evaluable predicates and other extra-logical support features.

As with λProlog, terms and atoms are written in curried form and the standard quantifier assumptions are made. It is straightforward to confirm that existing Prolog and λProlog programs are written, and run, as expected. For instance, the λProlog query:

```
pi X \ pi Y \
   (memb X (X::Y))  =>
pi X \ pi Y \ pi Z \
   (memb X (Y::Z) :- neq X Y, memb X Z) =>
memb M (a::b::nil).
```

represents the formula:

[5] There is one difference in the syntax of hereditary Harrop formulae when written in Lolli. The use of `forall` and `exists` as syntax for the explicit quantifiers, as opposed to `pi` and `sigma`, represents a personal preference of this author.

Connective	Parity	Syntax
$\top$	+	`erase`
1	+	`true`
&	+	`&`
	-	`&`
$\otimes$	+	`,`
$\oplus$	+	`;`
$\multimap$	+	`-o`
	-	`:-`
$\Rightarrow$	+	`=>`
	-	`<=`
!	+	`{...}`
$\forall x.B$	+	`forall x\B`
	-	`forall x\B`
$\exists x.B$	+	`exists x\B`

Figure 4.4: The Mapping of $\mathcal{L}'$ Connectives onto Lolli Concrete Syntax

$$\exists M.[(\forall X.\forall Y.memb(X, X :: Y)) \supset (\forall X.\forall Y.\forall Z.(memb(X, Y :: Z) \subset (neq(X, Y) \wedge memb(X, Z)))) \supset memb(M, a :: b :: nil)]$$

which, when translated (using the '$^+$' translation) into the new system becomes:

$$\exists M.[(\forall X.\forall Y.memb(X, X :: Y)) \Rightarrow (\forall X.\forall Y.\forall Z.(memb(X, Y :: Z) \circ\!\!- (neq(X, Y) \otimes memb(X, Z)))) \Rightarrow memb(M, a :: b :: nil)]$$

which has the concrete syntax:

```
forall X \ forall Y \
   (memb X (X::Y))  =>
forall X \ forall Y \ forall Z \
   (memb X (Y::Z) :- neq X Y, memb X Z) =>
memb M (a::b::nil).
```

And, when run, this query will have the same execution profile as the original λProlog query. Only when a program makes explicit use of the new connectives of Lolli will new behaviors be seen.

The syntax and semantics of the Lolli module system are essentially the same as that described in the last chapter, except that the file name extension is '`.ll`' rather than '`.mod`', and the module loading operator is '`--o`' rather than '`==>`'. In addition, because Lolli is essentially first-order —it allows quantifiers to range over higher types, but the unification algorithm is essentially first order— types and kinds (which are an important aspect of the λProlog module system, though they were left out of the discussion in the last chapter since they are

not relevant here) are neither needed nor supported. A future implementation of Lolli may support L_λ-unification, but will likely still be type-free.[6] Note that since constants are untyped, predicate names may be reused at different arities, as in ordinary Prolog. This is not allowed in λProlog.

Since it is assumed that most program clauses are intended to be treated in the traditional way (as reusable and ignorable), clauses in a module are loaded into the unbounded context, unless they are preceded with the keyword `LINEAR`. Thus if the Lolli module `mod` is defined as:

```
MODULE mod  x1 ... xn.
LOCAL y1 ... ym.

D1 x1 ... x_n y1 ... ym z11 ... zl1.
         ...          ...
LINEAR Di x1 ... xn y1 ... ym z1i ... zli.
         ...          ...
Dp x1 ... xn y1 ... ym z1p ... zlp.
```

then the goal (`mod` $t_1 \ldots t_n$ `--o` G) is syntactic sugar for the goal:

```
forall y1 \ ( ...  forall ym\ (
     (forall z11 \ ( ...  forall zl1\ (
        D1 t1 ... tn y1 ... ym z11 ... zl1) ... ) =>
                  ⋮
     (forall z1i \ ( ...  forall zli\ (
        Di t1 ... tn y1 ... ym z1i ... zli) ... ) -o
                  ⋮
     (forall z1p \ ( ...  forall zlp\ (
        Dp t1 ... tn y1 ... ym z1p ... zlp) ... ) => G ) ... )
```

Each of the clauses is added to the appropriate context. But, it should be noted that, as in λProlog clauses are added (by implication or module loading) to the top of the current program. Further, the relative order of clauses added to the linear and intuitionistic contexts is maintained for the purposes of searching for matches. The separation of the two contexts is a fiction maintained for the non-deterministic theory only.

4.6 Linear Logic as a Programming Language

What has been gained by the addition of the new connectives of linear logic? This question will be answered in depth in Chapters 5 and 6. But a good deal can be understood by revisiting the examples from the beginning of this chapter. Reconsider the clauses:

[6] L_λ is a fragment of the language of higher-order λ-terms for which Miller has shown that unification is decidable and generates most-general unifiers [Miller, 1991*b*; Miller, 1991*a*].

```
can_buy soda  :- have dollar.
can_buy candy :- have dollar.
```

In Lolli they represent the formulas:

$$\Gamma = \begin{cases} can\_buy(soda) \circ\!\!- have(dollar) \\ can\_buy(candy) \circ\!\!- have(dollar) \end{cases}$$

If these clauses are loaded into the unbounded context, the problem query behaves correctly (if we load the dollar into the linear context).

```
?- have dollar -o (can_buy soda , can_buy candy).
no.
```

In attempting to prove this query, the system builds the following derivation, but is unable to complete the proof due to a shortage of cash in the upper-rightmost leaf:

$$\dfrac{\dfrac{\dfrac{}{\Gamma; have(dollar) \longrightarrow have(dollar)}}{\Gamma; have(dollar) \longrightarrow can\_buy(soda)}\,BC \quad \dfrac{\Gamma; \varnothing \longrightarrow have(dollar)}{\Gamma; \varnothing \longrightarrow can\_buy(candy)}\,BC}{\dfrac{\Gamma; have(dollar) \longrightarrow (can\_buy(soda) \otimes can\_buy(candy))}{\Gamma; \varnothing \longrightarrow have(dollar) \multimap (can\_buy(soda) \otimes can\_buy(candy))}\,\multimap_R}\,\otimes_R$$

The example of object-state may be recast in a similar way. If we define a class of switch objects as:

```
MODULE switch name initial.
LOCAL sw.

LINEAR sw initial.

setting name S :- sw S.

toggle name G :- sw on, sw off -o G.
toggle name G :- sw off, sw on -o G.
```

then the core of the proof of the goal `(switch s1 on) --o (toggle s1 (setting s1 S))`, proceeds as follows:

$$\dfrac{\dfrac{}{\Gamma; sw(s_1, on) \longrightarrow sw(s_1, on)} \quad \dfrac{\dfrac{\dfrac{\boxed{s = off}}{\Gamma; sw(s_1, off) \longrightarrow sw(s_1, s)}}{\Gamma; sw(s_1, off) \longrightarrow setting(s_1, s)}\,BC}{\Gamma; \varnothing \longrightarrow sw(s_1, off) \multimap setting(s_1, s)}\,\multimap_R}{\dfrac{\Gamma; sw(s_1, on) \longrightarrow sw(s_1, on) \otimes (sw(s_1, off) \multimap setting(s_1, s))}{\Gamma; sw(s_1, on) \longrightarrow toggle(s1, setting(s_1, s))}\,BC}\,\otimes_R$$

with the desired result that the switch is in the toggled position during the proof of `(setting s1 S)`.

Chapter 5

A Collection of Lolli Programming Examples

Thus far this dissertation has consisted almost entirely of theoretical arguments about the design of a logic for logic programming. While that discussion has been motivated by a few small example problems, they have been quite simple and limited. The purpose of this chapter is to present a large enough selection of examples to give the reader a sense of the utility of the language we have described.

In order to make the examples easy to understand, most are still relatively simple, and will stay close to the pure system, making only limited use of the extra-logical features of the implementation. In some cases a simple, pure version of the example will be given, followed by a longer, more feature-rich version. Chapter 9 should be used as a reference for the built-in extra-logical features such as the input/output predicates and guard expressions.

Almost all of these examples will rely on the continuation-passing style of programming discussed in Chapter 3. It is important to understand those examples before attempting to tackle these.

5.1 Using the Linear Context as a Mixing Bowl

One simple use of the linear context is to use it as a sort of mixing bowl into which data that is to be manipulated is tossed. The data is manipulated in whatever way is desired, and then collected out of the context. This section consists of a handful of simple variations on this theme.

5.1.1 Permuting a List

```
MODULE permuter.

distribute nil    G  :- G.
distribute (X::L) G  :- (item X -o distribute L G).

collect nil.
collect (X::L)  :- item X, collect L.

permute L K :- distribute L (collect K).
```

This is the paradigmatic example of using the linear context as a mixing bowl, in that no changes are made to the data elements themselves. The `distribute/2` predicate loads the elements of the list in its first argument into the bounded context, with each element of the list stored in a separate atomic clause for the `item/1` predicate. Once the entire list has been loaded into the bounded context, the continuation goal in the second argument is called.

The `collect/1` predicate simply repeatedly proves `item` in order to collect the values stored in the bounded context into a list. Notice that, assuming there are no other consumptive or weakening goals, the first clause for `collect` cannot be used successfully until all the `item`'s have been collected, since the linear constraint will not allow a query to succeed if there are formulas left in the linear context. Thus this definition is entirely deterministic even without the use of Prolog's cut mechanism.

Because the choice of the `item` that is `collect`'ed in each pass is non-deterministic (in theory, in reality it is top-down), the `permute/2` predicate can succeed for every possible arrangement of the first argument. Thus a typical interaction might be:

```
?- permuter --o permute (1::2::3::nil) K.
K_1 <- 3 :: 2 :: 1 :: nil;
K_1 <- 3 :: 1 :: 2 :: nil.
yes
```

The proof built to provide the first response (once the module is loaded) is shown in Figure 5.1.

The Modality of ! and the `demo` Predicate

In order to make use of `permute` to correctly permute the elements of a list, not allowing any pre-existing proof context to intrude, we must guarantee two things about the context: first, the predicates `permute`, `item`, `distribute`, and `collect` cannot be used as head symbols in any part of the context except as specified above and, second, the bounded part of a context must be empty at the start of the computation of a permutation.

It is possible to handle the first condition by making use of appropriate universal quantifiers (as embodied in `LOCAL` definitions) over the predicate names `permute`, `item`, `distribute`, and `collect`. The second condition —that the unbounded part of a context is empty— can be

$$\Xi = \left\{ \dfrac{\dfrac{\boxed{x_1 = 3}}{\Gamma; it(3) \longrightarrow it(x_1)} \quad \dfrac{\dfrac{\boxed{x_2 = 2}}{\Gamma; it(2) \longrightarrow it(x_2)} \quad \dfrac{\dfrac{\boxed{x_3 = 1}}{\Gamma; it(1) \longrightarrow it(x_3)} \quad \dfrac{\boxed{l_3 = \mathsf{nil}}}{\Gamma; \emptyset \longrightarrow coll(l_3)}\otimes_R}{\Gamma; it(1) \longrightarrow it(x_3) \otimes coll(l_3)}\, BC\, \boxed{l_2 = x_3 :: l_3}}{\Gamma; it(1) \longrightarrow coll(l_2)}\otimes_R}{\Gamma; it(1), it(2) \longrightarrow it(x_2) \otimes coll(l_2)}\, BC\, \boxed{l_1 = x_2 :: l_2}}{\Gamma; it(1), it(2) \longrightarrow coll(l_1)}\otimes_R}{\Gamma; it(1), it(2), it(3) \longrightarrow it(x_1) \otimes coll(l_1)}\, BC\, \boxed{k = x_1 :: l_1}}{\Gamma; it(1), it(2), it(3) \longrightarrow dist(\mathsf{nil}, coll(k))}\, BC \right.$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; it(1), it(2), it(3) \overset{\Xi}{\longrightarrow} dist(\mathsf{nil}, coll(k))}{\Gamma; it(1), it(2) \longrightarrow it(3) \multimap dist(\mathsf{nil}, coll(k))}\multimap_R}{\Gamma; it(1), it(2) \longrightarrow dist((3::\mathsf{nil}), coll(k))}\, BC}{\Gamma; it(1) \longrightarrow it(2) \multimap dist((3::\mathsf{nil}), coll(k))}\multimap_R}{\Gamma; it(1) \longrightarrow dist((2::3::\mathsf{nil}), coll(k))}\, BC}{\Gamma; \emptyset \longrightarrow it(1) \multimap dist((2::3::\mathsf{nil}), coll(k))}\multimap_R}{\Gamma; \emptyset \longrightarrow dist((1::2::3::\mathsf{nil}), coll(k))}\, BC}{\Gamma; \emptyset \longrightarrow permute((1::2::3::\mathsf{nil}), k)}\, BC$$

Figure 5.1: A Proof Corresponding to the Execution of `permute`

managed by making use of the modal nature of ! in goals, which has not been discussed so far.

One extension to logic programming languages that has been studied for several years is the `demo`-predicate [Bowen and Kowalski, 1982]. The intended meaning of attempting a query of the form `demo(`D`,`G`)` in context Γ is simply attempting the query G in the context containing only D; that is, the main context is forgotten during the scope of the `demo`-predicate. The use of a !'ed goal has a related meaning.

Consider proving the sequent $\Gamma; \Delta \longrightarrow (!\, G_1) \otimes G_2$. Given our analysis of proofs in Chapter 4, this is provable if and only if the two sequents $\Gamma; \emptyset \longrightarrow G_1$ and $\Gamma; \Delta \longrightarrow G_2$ are provable. In other words, the use of the "of course" operator forces G_1 to be proved with an empty bounded context. Thus, with respect to bounded resources, the goal $!(D \multimap G)$ behaves similarly to `demo(`D`,`G`)`. In a sense, since bounded resources can come and go within contexts, they can be viewed as "contingent" resources, whereas unbounded resources are "necessary". The "of course" operator attached to a goal ensures that the provability of the goal depends only on the necessary and not the contingent resources of the context.

It is now clear how to define the permutation of a list given the example program above; redefine the `permute` predicate with the clause:

```
permute L K  :-  {distribute L (collect K)}.
```

or, equivalently, the clause:

```
permute L K  <=  distribute L (collect K).
```

This forces the bounded part of the context to be empty at the start of the (sub)computation of a permutation.

5.1.2 Sorting a List

It is easy to see how to modify the last example to sort, rather than permute, a list. All that is necessary is to control the circumstances under which an item is read back from the linear context: it must be greater than or equal to (for example) the last element read in. If this constraint is enforced for all elements, then the constructed list will be sorted.

```
MODULE sorter ordering.
LOCAL collect distribute item.

distribute nil    G :- G.
distribute (X::L) G :- item X -o distribute L G.

collect nil.
collect (X::nil)  :- item X.
collect (X::Y::L) :- item X, collect (Y::L), ordering X Y.

sort L K <= distribute L (collect K).
```

This module is parameterized by the ordering to be used in sorting the list. Typical uses would include:

```
?- sort '>=' --o sort (1::3::5::2::4::6::0::nil) A.
A_1 <- 6 :: 5 :: 4 :: 3 :: 2 :: 1 :: 0 :: nil.
yes
?- sort '=<' --o sort (1::3::5::2::4::6::0::nil) A.
A_97424 <- 0 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: nil.
yes
?- sort '=<' --o sort (1::2::3::nil) (2::3::1::nil).
no
?- sort '=<' --o sort A (1::2::nil).
A_181288 <- 2 :: 1 :: nil;
A_181288 <- 1 :: 2 :: nil.
yes
```

Obviously, this is an enourmously inefficient —$O(n!)$ in the worst case— way to sort a list, but it is instructive nonetheless.

5.1.3 Multiset Rewriting

In the last two examples, the list items were simply loaded into the linear context and read back. This idea can be expanded upon to show how the bounded part of a context can be

employed to do multiset rewriting. Let H be the *multiset rewriting system* $\{\langle L_i, R_i\rangle \mid i \in I\}$ where for each $i \in I$ (a finite index set), L_i and R_i are finite multisets. Define the relation $M \rightsquigarrow_H N$ on finite multisets to hold if there is some $i \in I$ and some multiset C such that M is $C \uplus L_i$ and N is $C \uplus R_i$. Let $\rightsquigarrow_H^*$ be the reflexive and transitive closure of $\rightsquigarrow_H$.

Given a rewriting system H, we wish to specify the meaning of a binary predicate `rewrite` such that `(rewrite L K)` is provable if and only if the multisets encoded by `L` and `K` stand in the $\rightsquigarrow_H^*$ relation. Let Γ_0 be the following set of formulas (these are independent of H):

```
MODULE rewriter rulemodule.
LOCAL collect distribute item rew.

distribute nil    G :- G.
distribute (X::L) G :- (item X -o distribute L G).

collect nil.
collect (X::L) :- item X, collect L.

rew K :- collect K.

rewrite L K <= (rulemodule rew item --o
                          distribute L (rew K)).
```

Taken alone, these clauses give a slightly different version of the `permute` program of the first example. The only addition is the `rew/1` predicate, which will be used as a socket into which we can plug a particular rewrite system (the rules for which will be in the module `rulemodule`).

In order to encode a rewrite system H, each rewrite rule in H is given by a formula specifying an additional clause for the `rew` predicate as follows: If H contains the rewrite pair $\langle\{a_1,\ldots,a_n\},\{b_1,\ldots,b_m\}\rangle$ then this pair is encoded as the formula

`rew K :- item` a_1`, ..., item` a_n`, (item` b_1 `-o (... -o (item` b_m `-o rew K)...)).`

If either n or m is zero, the appropriate portion of the formula is deleted. Operationally, this clause reads the a_i's out of the bounded context, loads the b_i's, and then attempts another rewrite. Let Γ_H be the set resulting from encoding each pair in H.

As an example, if H is the set of pairs $\{\langle\{a,b\},\{b,c\}\rangle, \langle\{a,a\},\{a\}\rangle\}$ then Γ_H is the set of formulas:

```
MODULE rule1 rew item.

rew K :- item a, item b, (item b -o (item c -o rew K)).
rew K :- item a, item a,            (item a -o rew K).
```

The following claim is easy to prove about this specification: if M and N are multisets represented as the lists `L` and `K`, respectively, then $M \rightsquigarrow_H^* N$ if and only if the goal `(rewrite L K)` is provable from the context $\Gamma_0, \Gamma_H; \varnothing$.

Notice that the rule module is parameterized by the names used for the rewriting predicate and for the item storage. This is necessary since those names are scoped locally in the rewriting module.

One drawback of this example is that `rewrite` is a predicate on lists, though its arguments are intended to represent multi-sets, and are operated on as such. Therefore, for each M, N pair this program generates a factor of $n!$ more proofs than the corresponding rewriting proofs, where n is the cardinality of the set underlying the multiset N. This redundancy could be addressed either by implementing a data type for multisets or, perhaps, by investigating a non-commutative variant of linear logic.

5.1.4 An Expression Compiler for a Register Machine

The last example of this style of programming is a simple expression compiler for a register machine. Here the linear context is used to store resources representing the availability of registers. Each formula represents a single register. The code for the compiler is given in Figure 5.2. A query consists of a call to `compile/2` with the expression to be compiled and the number of registers available as arguments. The query fails if the program cannot find a way to compile the given expression for a machine with that many registers. Otherwise, it prints out the resultant assembly code. A sample interaction, asking the system whether $((1 + 2) * (3 + 4) * (5 * 6))$ can be compiled to a machine with 14 registers is shown below:

```
?- compile --o top.
?- compile ((1 + 2) * (3 + 4) * (5 * 6)) 14.

((load r14) 1)
((load r13) 2)
(((add r13) r14) r13)

((load r14) 3)
((load r12) 4)
(((add r12) r14) r12)

(((mul r12) r13) r12)

((load r13) 5)
((load r14) 6)
(((mul r14) r13) r14)

(((mul r14) r12) r14)
yes.
```

```
% A simple expression compiler.
%   compile Exp Numregs
% succeeds (and prints out the compiled code) if it can
% find a way to compile the expression in that many registers.

MODULE compile.

% define not(G) in terms of guard expression
not G :- G -> false | one.

% load a list of bounded resources into the database
% then execute continuation G.
distribute nil G :- G.
distribute (H::T) G :- reg H -o distribute T G.

% print out  the list of code triples
writec (A,B,C) :- writec A, nl, writec B, nl, writec C, nl.
writec C :- not (C = (A,B,D)), write C.

% Given a number N, generate a list of that many register names
reglist N Rs :- (N = 0) -> (Rs = nil)
                         | (Rs = (R::RL), implode (r::N::nil) R,
                                          N1 is (N - 1), reglist N1 RL).

% The expression compiler, with register reuse.
compile Expr Numregs:- reglist Numregs Regs,
                distribute Regs (comp Expr Code Reg (true)), writec Code.

comp (X + Y) (C1,C2,(add R3 R1 R2)) R3 G :-
        comp X C1 R1 (comp Y C2 R2 (distribute (R1::R2::nil) (reg R3,G))).
comp (X - Y) (C1,C2,(sub R3 R1 R2)) R3 G :-
        comp X C1 R1 (comp Y C2 R2 (distribute (R1::R2::nil) (reg R3,G))).
comp (X * Y) (C1,C2,(mul R3 R1 R2)) R3 G :-
        comp X C1 R1 (comp Y C2 R2 (distribute (R1::R2::nil) (reg R3,G))).
comp (X / Y) (C1,C2,(div R3 R1 R2)) R3 G :-
        comp X C1 R1 (comp Y C2 R2 (distribute (R1::R2::nil) (reg R3,G))).

comp X (load R X) R G :- not (X = A + B), not (X = A - B),
                         not (X = A * B), not (X = A / B), reg R, G.
```

Figure 5.2: A Simple Expression Compiler With Register Re-use

```
MODULE propositional.

pv (and A B) :- pv B & pv A.
pv (imp A B) :- hyp A -o pv B.
pv (or  A B) :- pv A.
pv (or  A B) :- pv B.
pv G :- hyp (and A B), (hyp A -o hyp B -o pv G).
pv G :- hyp (or  A B), ((hyp A -o pv G) & (hyp B -o pv G)).
pv G :- hyp (imp C B), ((hyp (imp C B) -o pv C) &
                        (hyp B -o pv G)).
pv G :- hyp false, erase.
pv G :- hyp G, erase.
```

Figure 5.3: A Specification of Propositional Intuitionistic Logic

5.2 Context Management in Theorem Provers

Intuitionistic logic is a useful meta-logic for the specification of provability of various object logics. For example, consider axiomatizing provability in propositional, intuitionistic logic (over the logical symbols `imp`, `and`, `or`, and `false` for object-level implication, conjunction, disjunction, and absurdity). A very natural specification of the natural deduction inference rule for implication introduction is:

```
pv (imp A B) :- hyp A => pv B.
```

where `pv` and `hyp` are meta-level predicates denoting provability and hypothesis. Operationally, this clause states that one way to prove `(imp A B)` is to add `(hyp A)` to the context and attempt a proof of `(pv B)`. In the same setting, conjunction elimination can be expressed by the formula

```
pv G :- hyp (and A B), (hyp A => hyp B => pv G).
```

This clause states that in order to prove some formula `G`, first check to see if there is a conjunction, say `(and A B)`, in the context and, if so, attempt a proof of `G` in which the context is extended with the two hypotheses `A` and `B`. Other introduction and elimination rules can be axiomatized similarly. Finally, the clause:

```
pv G :- hyp G.
```

is needed to actually complete a proof. With the complete specification, it is easy to prove that there is a proof of `(pv G)` in the meta-logic if and only if there is a proof of `G` in the object logic.

Unfortunately, using intuitionistic logic as the meta-theory does not extend naturally to describing provability in logics that have restricted contraction rules —such as linear logic

itself— because hypotheses are maintained in intuitionistic logic contexts and hence can be used zero or more times. Even in describing provability for propositional intuitionistic logic there are some drawbacks. For instance, it is not possible to logically express the fact that a conjunctive or disjunctive formula in the proof context only needs to be eliminated at most once. So, for example, in the specification of conjunction elimination, once the context is augmented with the two conjuncts, the conjunction itself is no longer needed in the context.

If, however, we replace the intuitionistic meta-logic with our refinement based on linear logic these observations about use and re-use in intuitionistic logic can be specified elegantly, as is done in Figure 5.3. In that specification, a hypothesis is both "read from" and "written into" the context only during the elimination of implications. All the other elimination rules simply "read from" the context; they do not "write back." The last two clauses in Figure 5.3 use a $\otimes$ with $\top$: these allow any unused hypotheses to be erased, since the object logic has no restrictions on weakening.

Of course even this refined specification cannot be used effectively with a depth-first interpreter because if the implication left rule can be used once, it can be used any number of times, thereby causing the interpreter to loop. (This problem is even more severe in the first, intuitionistic, implementation since this problem occurs with all of the elimination rules.)

Fortunately, a contraction-free presentation of propositional intuitionistic logic has been given by Dyckhoff [Dyckhoff, 1992]. Representing Dyckhoff's system in this setting involves replacing the one formula specifying implication elimination in Figure 5.3 with five special cases of implication elimination as in Figure 5.4. The fifth of the special cases applies only to implications with atomic bodies. Therefore, the predicate `isatom/1` is also added. It is defined in terms of negation as failure, which is in turn defined in terms of guard expressions, as discussed in Chapter 9. Executing this linear logic program in a depth-first interpreter yields a decision procedure for propositional intuitionistic logic.

Appendix *G* gives the implementation of an enriched version of this program which generates LaTeX code of the proofs it constructs. The code for both versions is included with the current release of the Lolli system.

5.3 Object Oriented Programming and Mutable State

Another use for the bounded context is to store clauses that are intended to represent mutable data, for instance the state of some object. This was in fact, one of the applications that motivated this work in linear-logic programming. In an earlier paper, we suggested that the scoping mechanisms described in Chapter 3 could be seen as providing most of the features needed to implement an object-oriented style of logic programming, including abstraction, message passing and inheritance [Hodas and Miller, 1990]. Unfortunately, as shown in that paper, and in Chapter 4, intuitionistic logic is too weak to provide a proper notion of mutable state.

```
MODULE propositional.

pv (and A B) :- pv B & pv A.
pv (imp A B) :- hyp A -o pv B.
pv (or A B) :- pv A.
pv (or A B) :- pv B.
pv G :- hyp (and A B), (hyp A -o hyp B -o pv G).
pv G :- hyp (or A B), ((hyp A -o pv G) & (hyp B -o pv G)).

pv G :- hyp (imp (imp C D) B),
          ((hyp (imp D B) -o pv (imp C D)) & (hyp B -o pv G)).
pv G :- hyp (imp (C and D) B), (hyp (imp C (imp D B)) -o pv G).
pv G :- hyp (imp (C or  D) B), (hyp (imp C B) -o hyp (imp D B) -o pv G).
pv G :- hyp (imp false B), pv G.
pv G :- hyp (imp A B), isatom A, hyp A, (hyp B -o hyp A -o pv G).

pv G :- hyp false, erase.
pv G :- hyp G, erase.

isatom A :- not (A = (and B C)), not (A = (or B C)),
            not (A = (imp B C)).

not G :- G -> false | true.  % If G then fail, otherwise succeed.
```

Figure 5.4: A Contraction-Free Specification of Propositional Intuitionistic Logic

5.3.1 A Simple Example: Toggling a Switch

In Chapter 4 the following module was given as an implementation of a switch with mutable state:

```
MODULE switch name initial.
LOCAL sw.

LINEAR sw initial.

setting name S :- sw S.

toggle name G :- sw on, sw off -o G.
toggle name G :- sw off, sw on -o G.
```

A sample execution proof was given that showed that the system is "correct", at least in the sense that it does successfully toggle the state of a switch. Unfortunately, the system still does not behave quite as one would expect. Consider the following interaction (recall that, as explained in Section 3.2.1, the evaluable predicate `top` causes a new read-prove-print loop to be invoked at that point):

```
?- switch s1 on --o top.
?- toggle s1 top.
?- setting s1 S.
S_51 <- off
yes
?- switch s2 off --o top.
?- setting s1 S.
no
```

The problem is that once the second switch is initialized there are two formulas in the linear context. Any future goal must use both of these formulas if its proof is to succeed. Thus, in order to check the state of one switch, the state of the other must be discarded:

```
?- setting s1 S, erase.
S_96 <- off
yes
```

The best solution is to modify the definition of `setting` to include the call to `erase`.

It is important to remember that checking the state of an object will read its value out of the context, making it unavailable for subsequent use. In this case, the use of `erase` consumes the values of all objects. Thus, when combining a call to `setting` with other goals, the conjunction should be formed using the '&' operator. This duplicates the whole context for each of the conjuncts.

5.4 A Data Base Query Language

A program that implements a simple data base query language is displayed in Figure 5.5. The data base is stored in both the bounded and unbounded parts of the context using the predicate `entry/1`. Entries in the bounded part can be retracted or updated; entries in the unbounded context are permanent and cannot be updated or retracted. A session using this program might proceed as follows:

```
?- database --o db.

Command: enter (student josh).
Command: enter (student ramesh).
Command: commit (student josh).
Command: enter (professor jean).
Command: enter (professor dale).
Command: check (professor jean).
professor jean is an entry.
Command: necessary (professor jean).
Try again.
Command: update (student ramesh) (professor ramesh).
student ramesh was updated to professor ramesh.
Command: update (student josh) (professor josh).
student josh is a necessary fact, and cannot be updated.
Command: quit.

aborted...
```

This example shows some of the limitations of linear contexts. For example, while it is possible to check whether an entry is committed, it is not possible to find out if an entry is contingent and not committed without stepping outside the logic by using negation-as-failure. Thus it is not possible (using just the logic) to check if a command is attempting to retract a necessary (committed) entry on which that operation is ineffective.

```
MODULE database.

db :- write_sans "Command: ", read Command, do Command.
db :- write_sans  "Try again.", nl, db.

do (enter E)   :- entry E -o db.
do (commit E)  :- entry E => db.
do (retract E) :- ({entry E} ->
                     (write E, write_sans " is a necessary fact, and ",
                      write_sans "cannot be retracted.", nl,  db)
                   | (entry E, write E, write_sans " was retracted.",
                      nl, db)
                  ).

do (update E N) :- ({entry E} ->
                      (write E, write_sans " is a necessary fact, and ",
                       write_sans "cannot be updated.", nl,        db)
                    | (entry E, write E, write_sans  " was updated to ",
                       write N, write_sans ".", nl, entry N -o db)
                   ).
do (check Q)    :- (entry Q, erase, write Q,
                    write_sans " is an entry.", nl) & db.
do (necessary Q):- ({entry Q}, erase, write Q,
                    write_sans " is a necessary entry", nl) & db.
do quit         :- abort.
```

Figure 5.5: A simple data base query program

Chapter 6

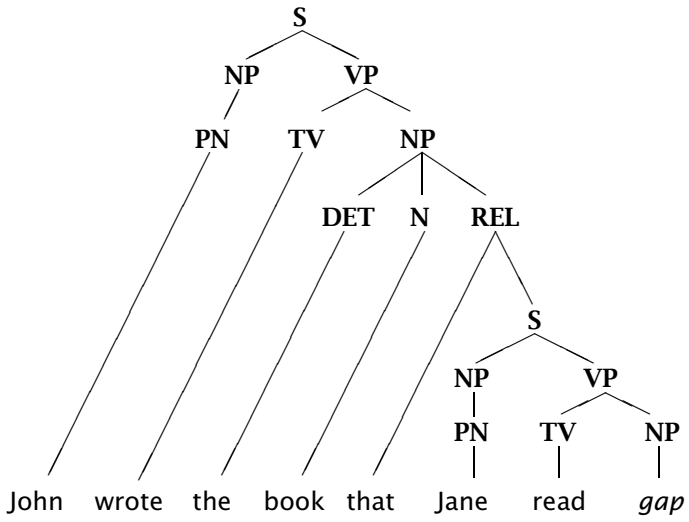
$\mathcal{L}$ as an Extension of Phrase Structure Grammar

6.1 Unbounded Dependencies in Natural Language Processing

It is now standard in the fields of computational linguistics and natural language processing to view a relative clause as being formed by a relative pronoun followed by a sentence that is missing a noun phrase. For example, the sentence:

John wrote the book that [Jane read *gap*].

can be thought of as having the following parse tree, where *gap* marks the spot where the missing noun phrase would be, if the clause were a sentence:



This sort of construction is called an *unbounded dependency* (in this case an *unbounded filler-gap dependency*), because two features (in this case the filler, that, and the *gap*) which stand in a dependency relationship can be separated in a sentence by arbitrary grammatical structures. And while in this example the extraction site is the rightmost constituent of the subordinate sentence, it can in fact occur almost anywhere. So, for instance, the following sentence is also acceptable:

John wrote the book that [Jane gave *gap* to Jill].

A common technique for implementing parsers that can handle such structures in logic programming is the technique of *gap threading* in which a difference list of gaps is passed around as a parameter in a *Definite Clause Grammar* (DCG).[1] The state of this list changes as gaps are introduced (by encountering relative pronouns, for example) and discharged (by completing a parse that uses the gap). In turn the state of the list controls whether the parse is allowed to use a gap in place of an **NP** at a given point in the parse [Pereira and Shieber, 1987]. Unfortunately, this technique requires tedious modification of the entire grammar, even the parts that are not involved in parsing structures that need the gap list.

Pareschi has proposed a method of handling unbounded filler-gap dependencies at the logic level, rather than in the term language. This technique made use of the enhanced goal language of λProlog. The basic idea is to temporarily augment the grammar with a new rule for gap noun phrases only during the parse of a relative clause. Unfortunately, because the management of proof context in intuitionistic logic is too coarse, the grammars that result are unsound (though in a different way than the difference-list grammars), accepting many non-grammatical sentences [Pareschi, 1989; Pareschi and Miller, 1990].

In this chapter I will begin by describing the basics of the Generalized Phrase Structure Grammar formalism. I will then present the traditional gap threading technique as well as the system proposed by Pareschi and Miller and explain their shortcomings. Finally, I will present a solution to the problem inspired by Pareschi and Miller's work but implemented in Lolli. This solution, addresses all of the failings of the previous solutions, while maintaining the naturality of Pareschi and Miller's system. I will assume that the reader is familiar with the basics of phrase structure grammars and definite clause grammars (their Prolog analogues).

The ideas in this chapter were first briefly presented in the joint papers with Miller which introduced our system of linear-logic programming [Hodas and Miller, 1991; Hodas and Miller, 1994].[2] The idea was developed in greater depth by this author in a paper for the 1992 Joint International Conference and Symposium on Logic Programming [Hodas, 1992]. This chapter is largely a revision of that paper.

[1] Note that, while it can be viewed as a method for implementing GPSG, gap threading was actually developed independently of GPSG.

[2] In his dissertation Pareschi proposed the idea of using linear logic to solve the problems in his system. He did not, however, predict all the additional benefits that would arise from this sort of solution.

6.2 Generalized Phrase Structure Grammar

The Generalized Phrase Structure Grammar (GPSG) formalism developed by Gerald Gazdar [Gazdar, 1981; Gazdar, 1982; Gazdar *et al.*, 1985] demonstrated that it is possible to parse grammatical structures involving unbounded dependencies, such as relative clauses and **wh** questions, using a phrase structure grammar. Previously it had been supposed that such constructs were too complex for phrase structure grammars, which are context-free, but rather required the strength of transformational grammar.

The basic ideas in GPSG are quite simple. It posits, for instance, that the body of a relative clause is a sentence that is missing a noun phrase somewhere. So, if sentences belong to the category **S**, and noun phrases to the category **NP**, then the rule for (one particular form of) relative clause would be:

REL⟶ REL-PRON S/NP

where **S/NP** is the derived category of sentences missing a noun phrase. This requires, in turn, that rules be given for generating/parsing the derived category. These new rules are generated from the original grammar in a relatively straightforward manner. So, if the original grammar were:

S ⟶ NP VP
NP ⟶ PN
NP ⟶ DET N
VP ⟶ TV NP
VP ⟶ DTV NP NP

then the new grammar, which allows relative clauses in noun phrases, would consist of that grammar augmented with:

NP ⟶ DET N REL
REL ⟶ REL-PRON S/NP

S/NP ⟶ NP VP/NP
VP/NP ⟶ TV NP/NP
VP/NP ⟶ DTV NP/NP NP
VP/NP ⟶ DTV NP NP/NP
NP/NP ⟶ ϵ

6.2.1 Island Constraints

In general, for each rule in the original grammar which defines a category that could dominate an **NP** (i.e., could occur above an **NP** in a parse tree) there will be a new version of that rule for each category on the right of the rule that could dominate an **NP**. Note, however, that we have not included the derived rule:

S/NP ⟶ NP/NP VP

This was done in order to block extraction from within the subject noun phrase. If this rule were included in the grammar then the grammar (were it also extended to handle stative verbs and prepositional phrases) would incorrectly admit such sentences as:

* John wrote the book [that [the story in *gap* is long]].

This sort of restriction, in which certain constructions are disallowed in certain positions is called an *island constraint* [Ross, 1967]. We will have more to say about this particular island constraint, which is sometimes called the *generalized left-branch condition*, below.

6.3 GPSG in Prolog

There are many ways to approach implementing GPSG style grammars in Prolog. Obviously, the grammar could be implemented directly as a DCG with rules defining the base categories as well as each of the derived categories. This is not an attractive option, however, since, depending on the number of derived categories, the resulting grammar can be substantially (potentially quadratically) larger than the core grammar on which it is based. Gazdar points out, however, that since the rules for the derived categories are formed so uniformly from the original grammar, it is possible to use the original grammar on its own, together with some switches controlling whether the parser selects rules as is or in derived form [Gazdar, 1981, page 161].

To understand how this idea can be implemented in DCG grammars, consider for example the two rules:

S ⟶ NP VP
S/NP ⟶ NP VP/NP

If we think in terms of a switch that can have two values, `gap` and `nogap`, then these two rules can be mapped to the single DCG rule:

```
s(Gap) --> np, vp(Gap).
```

When the variable `Gap` is instantiated to `nogap` this rule corresponds to the first GPSG rule. When it is instantiated to `gap` it corresponds to the second GPSG rule. So, for instance, the grammar above can be implemented with the following DCG:

```
s          --> s(nogap).
s(Gap)     --> np(nogap), vp(Gap).
np(gap)    --> [].
np(nogap)  --> pn.
np(nogap)  --> det, n.
np(nogap)  --> det, n, rel.
vp(Gap)    --> tv, np(Gap).
vp(Gap)    --> dtv, np(Gap), np(nogap).
vp(Gap)    --> dtv, np(nogap), np(Gap).
rel        --> rel-pron, s(gap).
```

Each rule where the head is parameterized by nogap corresponds to a rule in the core grammar only. Rules with heads parameterized by gap are slashed rules only. Finally, those parameterized by Gap act as core rules when the parameter is instantiated to nogap, but as slashed rules when it is instantiated to gap. It is easy to see that this DCG implements the grammar faithfully.

This implementation has problems though. First, as pointed out by Pereira and Shieber, because the two DCG rules for verb phrases with ditransitive verbs are identical when Gap is instantiated to nogap, simple sentences with ditransitive verbs will have two identical parses, depending on which rule is used in the parse [Pereira and Shieber, 1987]. In addition, the system of switches is too limited for grammars intended to handle multiple extractions from nested structures. Therefore gap-threading parsers (as these sorts of systems are called) are typically implemented with a difference list of gaps in place of the simple toggle. In such an implementation, the DCG above becomes:

```
s(S)           --> s([]-[]).
s(F0-F)        --> np([]-[]), vp(F0-F).
np([gap|F]-F)  --> [].
np(F-F)        --> pn.
np(F-F)        --> det, np([]-[]).
np(F-F)        --> det, np([]-[]), rel.
vp(F0-F)       --> tv, np(F0-F).
vp(F0-F)       --> dtv, np(F0-F1), np(F1-F).
rel            --> rel-pron, s([gap|F]-F).
```

Although the difference list in this grammar will never grow beyond a single element, in more complex grammars it can. Notice, though, that there is now just a single rule for ditransitive verb phrases. The difference list of gaps is divided between the two object noun phrases; if there is just one gap in the list it will go to one of the two noun phrases. This technique for implementing GPSG parsers has been developed extensively by Pereira and Shieber [Pereira and Shieber, 1987] and others.

Unfortunately, while using a difference list to manage gap distribution solves these problems, the solution is still unsatisfactory. In particular, a problem common to both implementations is that they are difficult to construct. Insuring that the gap distribution is correct can

be quite subtle. Precisely because the filler-gap dependency is unbounded, portions of the grammar that seem unconnected with the problem at hand require significant adjustment to insure that they do not interfere with the transmission of the gap information from the gap's introduction to its discharge, since the two may be separated by almost any structure.

6.4 GPSG in Intuitionistic Logic

Notice that the first DCG implementation of the grammar above includes the rule:

```
np(gap) --> [].
```

which, if we ignore the switch argument, corresponds to the grammar rule:

NP $\longrightarrow \epsilon$

In some sense the purpose of the switches, the difference lists, and even the slashes in the original grammar, is to control when this rule can be used. These machinations are necessary because DCG's and PSG's, like Horn clauses, are flat.

In his dissertation, and later in a paper with Miller, Pareschi proposed using the expanded formula language of λProlog to enhance the expressiveness of DCG's [Pareschi and Miller, 1990].[3] In particular, Pareschi proposed using the $\supset_r$ rule to restrict the scope of the **NP** $\longrightarrow \epsilon$ rule to the derivation of the **S** that forms the body of the relative clause.

If we imagine augmenting the standard DCG syntax with the '=>' operator from λProlog then using this idea we end up with the following implementation of the grammar (note that we also switch at this point to the λProlog representation of lists as built from `nil` and +):

```
s   --> np, vp.
np  --> pn.
np  --> det, n.
np  --> det, n, rel.
vp  --> tv, np.
vp  --> dtv, np, np.
rel --> rel-pron, (np --> nil) => s.
```

The key feature of this implementation is that the bulk of the grammar is unchanged from the core grammar. The only gap-related machinery is in the final rule. When this rule is invoked the rule `np --> nil` is added to the grammar and an attempt is made to parse for an **S**. This rule may be used to complete any rule looking for an **NP** during the parse of the subordinate **S**.

[3] Pareschi's dissertation, and the paper with Miller focused both on parsing and the construction of semantics. In this chapter we will be concerned only with the former, so I will summarize only that aspect of the work. However, if we consider a version of Lolli with λ-terms and unification in the style of λProlog then the treatment of semantics could be mapped directly to this presentation.

Actually, λProlog does not directly support DCG syntax, and Pareschi and Miller described their techniques in terms of the underlying Prolog (and λProlog) clauses. Such clauses contain a difference list of words as an argument to every predicate, which is used to distribute the words in the string being parsed in much the same way that the gap-list above distributes gaps. This representation was used to advantage by Pareschi and Miller, as will be described below. This presentation will mostly be in terms of the augmented DCG syntax. For more on the relationship between DCG syntax and the underlying clauses see Pereira and Shieber [Pereira and Shieber, 1987].

6.4.1 Island Constraints

In order to handle restrictions on where a gap is allowable (such as the restriction on extraction from a subject noun phrase already discussed), Pareschi proposed modifying the general scheme to load so-called "gap locator" rules rather than the gap noun phrase rules themselves. These gap locator rules are just partially interpreted versions of the rules that can use the gap. So for instance, rather than assuming a gap noun phrase rule, the rule for relative clauses might become:

```
rel --> rel-pron, (vp --> tv) => (vp --> dtv np) => s.
```

While this works, it is roughly equivalent to defining the grammar to include rules for derived categories up front; and can become quite cumbersome if there are many categories immediately dominated by **S** which can dominate an **NP**.

6.4.2 Problems Due to Contraction and Weakening

A serious problem with this simple representation, which Pareschi recognized, is that, as mentioned above, the `np --> nil` rule can be used to complete *any* rule looking for an **NP** during the parse of the subordinate **S**. In particular, it could be used to fill both the direct and indirect object positions in a ditransitive verb phrase. So the grammar above incorrectly admits:

* John wrote the book that [Jane gave *gap gap*].

Pareschi's solution to this problem was to take advantage of the fact that he was working with λProlog clauses rather than in the DCG syntax by directly manipulating quantifiers to block this situation. The solution relies on the fact that the intended translation of the augmented DCG syntax:

```
rel --> rel-pron, (np --> nil) => s.
```

is that the assumed rule is to have the same quantifier treatment as any other rule. So the translation into λProlog clauses would be:

```
rel S0 S1 :- rel-pron S0 S2, (pi T0 \ (np T0 T0)) => s S2 S1.
```

Pareschi noted that if the explicit quantification of T0 is dropped, then when the gap noun phrase rule is added to the program in an attempt to parse a relative clause, T0 will be an uninstantiated logic variable. The first time that rule is used to fill a noun phrase in the subordinate sentence structure the variable will become instantiated to that string position. After that the rule cannot be used to complete a different noun phrase, unless the parse fails back to some point before this.[4] Unfortunately, this technique does not work to block double use of a gap if the presumed occurrences are next to each other (as in the example above) since the two gaps then have identical locator values.[5]

A similar problem with this system is that there is no logical method to require that an introduced gap be used. That is, the grammar will admit:

> * John wrote the book that [Jane read a magazine].

since

> Jane read a magazine.

is a valid sentence, and that is what the parser is looking for once it has seen the relative pronoun and assumed the scoped noun phrase rule.

In this case Pareschi's solution relies on the fact that his system builds semantic terms in conjunction with parse trees, and that this erroneous acceptance would lead to a vacuous abstraction —i.e. a term of the form $\lambda x.t$ where x is not free in t— in the semantic term constructed for the relative clause. It is possible to check for the presence of such a term using the λProlog unification mechanism in conjunction with negation-as-failure. Pareschi, therefore, discards any parse that constructs a semantic term with a vacuous abstraction.

6.5 GPSG in Lolli

While Pareschi's solutions to the problems in the last section work, they are unsatisfying in several ways. First, as pointed out before, they are only partial solutions. Equally seriously from a philosophical standpoint, it seems unfortunate that dual problems of multiplicity and relevance in the treatment of the assumed noun phrase rule should have such disparate solutions. The solutions feel a bit too close to "hacking" the grammar.

In recognizing the limitations of his system, Pareschi suggested that a solution might be found in Girard's linear logic, which at the time had only recently been introduced [Girard, 1987*a*].

The solution should be readily apparent. The rules of the grammar are loaded into the unbounded portion of the proof context, since they are intended to be available for use as many times as needed, or not at all. In contrast, the temporary rules for gapped noun phrases

[4] If this is unclear, look at the discussion of quantification in hypothetical queries in the registrar's data base example in Chapter 3.

[5] This problem was pointed out by Pereira at the defense of this dissertation.

are loaded into the linear context. Since a gap is stored as a bounded resource, it must be used exactly once in the subordinate sentence.

Thus if we further augment DCG syntax with the Lolli operator -o, then the Lolli version of the grammar becomes:

```
s   --> np, vp.
np  --> pn.
np  --> det, n.
np  --> det, n, rel.
vp  --> tv, np.
vp  --> dtv, np, np.
rel --> rel-pron, (np --> nil) -o s.
```

In order to insure that the gap rules are managed correctly, we must further assume, by analogy to the embedding of λProlog into Lolli given in Chapter 4, that the comma and DCG arrow now stand for '⊗' and '∘–' respectively. Each time a '⊗' conjunction goal is encountered during the parse of the subordinate **S**, the gap is carried up into the proof of only one side of the conjunction.

6.5.1 Coordination Constraints

In addition to yielding solutions to the problems that Pareschi and Miller encountered, the linear logic setting affords simple treatments of other parsing issues. One particularly nice feature of this system is its ability to specify the management of gaps across coordinate structures, such as conjuncts. GPSG proposes that any category can be expanded by the conjunction of two or more structures of the same category. So, for instance:

S ⟶ **S** and **S**.

It is required, however, that any gaps discharged in one conjunct also be discharged in the other. This is known as a *coordination constraint*. If the language level conjunction is represented in the grammar by the second form of logical conjunction, '&', then coordination constraints are handled automatically. Since the '&' goal duplicates the linear context into both branches of the proof, both of the subordinate parses must consume the same gaps. Thus, if the clause:

```
s  --> (s, (and::nil)) & s.
```

is added to the grammar,[6] then the system will admit the sentences

[John wrote the book] and [Jane read the magazine].

[6] In reality, since performing a parse in this setting corresponds to top-down parsing, we cannot add this or any similar rule to the proof as it would cause the parser to loop. Thus we must actually add a rule of the form:

```
s  --> (np, vp, (and::nil)) & (np, vp).
```

but this is a technical issue only.

and

John wrote the book that [Jane read *gap*] and [Jill discarded *gap*].

but will reject:

* John wrote the book that [Jane read *gap*] and [Jill discarded the magazine].

6.5.2 Island Constraints

In a similar manner, it is possible to use the ! operator in goals to specify restrictions on extraction in the grammar. For example, in order to block extraction from subject noun phrases, the first rule of the grammar can be rewritten as:

```
s --> {np}, vp.
```
[7]

Recall from Figure 4.2 that the proof rule for ! in a goal is:

$$\frac{\Gamma; \varnothing \longrightarrow A}{\Gamma; \varnothing \longrightarrow !A}\ !_R$$

Thus, if Γ is the grammar above, with the first rule modified as stated, together with lexical definitions for a sufficient set of words, then attempting to parse the relative clause:

that [Jane saw *gap*]

leads to a proof of the form shown in Figure 6.1. In contrast, attempting to parse the clause:

* that [the story in *gap* is long]

will fail, because the gap *np* formula will be unavailable for use in the branch of the proof attempting to parse the **NP** "*the story in *gap*", since the !'ed *np* goal forces the gap *np* in the linear context to the other side of the tree.

Unfortunately, this technique is a bit too coarse-grained. For instance, it blocks the acceptance of

who [*gap* saw Jane]

which should be allowed —the restriction on extraction from the subject noun should not block extraction of the entire subject, only its substructures. This problem can be circumvented by having multiple rules for relative clauses: one, as we have already shown, which introduces a gap and attempts to parse for an **S**, and one which simply attempts to parse for a **VP**. The other choice would be to avoid the use of '!' entirely by using gap locator rules as in Pareschi's system; but this seems unattractive for reasons already cited.

[7] This use of '{}' conflicts with their existing use for embedding Prolog code in a DCG clause. Since the square brackets, '[]', which are used for lists in Prolog and DCG clauses are not used for that purpose in Lolli, they are used for embedding Lolli code in Lolli DCG clauses.

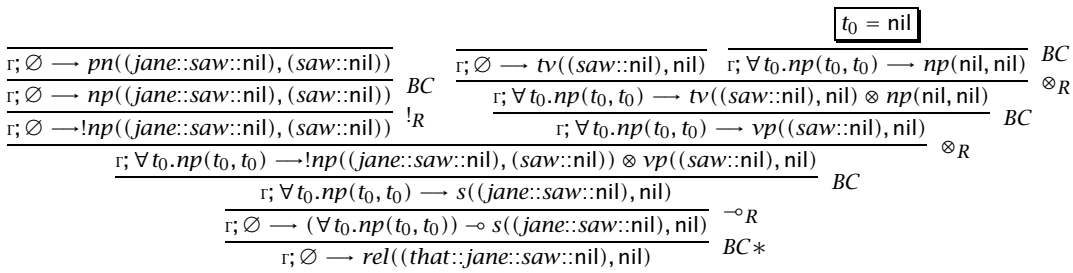


Figure 6.1: The Parse-Proof of "that jane saw"

6.5.3 Pied Piping

An interesting construction related to relative clauses is the so-called "pied-piping" construction, in which the relative pronoun is not the filler, but rather is properly embedded as the rightmost constituent within the filler. An example of this, taken from Pareschi, is the relative clause:

> the sister of whom [Paul married *gap*]

In this case, the filler can be seen as being formed by concatenating the relative pronoun onto an **NP/NP**, with the added restriction that the *gap* occur only as the rightmost element of the **NP/NP**. It is interesting that this sort of restricted extraction is exactly the sort that can be modeled well in Categorial Grammar, which cannot model arbitrary extraction well. In contrast, in our system, which handles arbitrary extractions easily, there is no straightforward way to restrict the location of the gap to one particular position.

If we are willing to leave the representation of grammar rules as propositional formulas, and move to the representation with explicit string lists and quantifiers (as Pareschi did) then we can also adopt Pareschi's solution to this problem. Consider the clauses:

```
rel  S0 S1 :- relf S0 S2, (forall T0 \ (np T0 T0)) -o s S2 S1.
relf S0 S1 :- (np S2 S2) -o np S0 S2, rel-pron S2 S1.
```

The first clause just replaces the clause for relative pronouns with one in which the filler is in the new category **RELF**. The second clause says that a **RELF** is either just a relative pronoun (if the assumed gap is used immediately to fill in for the noun phrase, in which case $s_2 = s_0$), or it is an **NP/NP** followed by a relative pronoun. Notice, though, that the rule assumed for the gap does not have its string position variables universally quantified. Rather they are pre-bound to the same position as the end of the **NP/NP**. Thus it can only be used in that position.

6.5.4 Parasitic Gaps

Historically, one of the most difficult filler-gap constructions to handle is the so-called "parasitic gaps" in which a single filler can be used to fill two gaps, but, unlike coordinate structures,

the second gap is optional. For instance, both of the following standard examples are acceptable noun phrases:

The book that [I discarded *gap* [without reading *gap*]]

The book that [I discarded *gap* [without reading the index]]

Pareschi proposes parsing such constructions by introducing two gaps but relaxing the relevance constraint on the second. Unfortunately in $\mathcal{L}$ there is no simple way to relax the relevance constraint, though an extended system in which this is possible will be discussed in Chapter 8. Another possible solution is to construct a special coordination constraint between the two verb phrases as in the following two clauses:

```
vp  --> (vp, (without::nil)) &  vp.
vp  --> (vp, (without::nil)) , {vp}.
```

The first rule is for the case in which the parasitic gap occurs, and the second for when it does not. In the second rule it is necessary to ensure that the gap is used in the first verb phrase and not the second, since otherwise the grammar would admit:

* The book that [I discarded the magazine [without reading *gap*]]

6.5.5 Other Types of Filler-Gap Dependencies

While the examples in this chapter have thus far dealt only with relative clauses, GPSG proposes solutions to many other sorts of unbounded dependencies. For instance, given a category **Q** of non-**wh** questions, the category can be expanded to cover some **wh** questions with GPSG rules of the form:

Q $\longrightarrow$ **wh-NP Q/NP**

So that from questions like:

Did Jane read the book?

one gets:

What [did Jane read *gap*]?

It should be apparent that such extensions are easy to add in this setting. Figure 6.2 shows a larger grammar than those presented up till now that parses several forms of sentences and questions. This grammar is designed to return the parse tree it constructs as an argument of the non-terminal parsed. Only the grammar itself is shown, the pre-terminals and lexicon are removed for the sake of brevity. Figure 6.3 shows a sample interaction with the parser, with several examples like those from the paper properly parsed or rejected. Note that a translator (written in Lolli) from the augmented DCG syntax to Lolli syntax is included with the current Lolli distribution.

```
GRAMMAR big_grammar.

parse Str Tree :- explode_words Str Lst,
                  (sent Tree Lst nil ; quest Tree Lst nil).

sent (sent NP VP)  --> {np NP}, vp VP.
sent (and (sent NP1 VP1) (sent NP2 VP2)) -->
        {np NP1}, vp VP1, (and::nil)   &   {np NP2}, vp VP2.

quest (quest VFA NP AP) --> vfa VFA, np NP, ap AP.
quest (quest VFA NP VP) --> vfa VFA, np NP, vp VP.
quest (quest NPWH Q)    --> npwh NPWH, np (np gap) -o quest Q.
quest (quest APW  Q)    --> apwh APW,  ap (ap gap) -o quest Q.
quest (quest PPW  Q)    --> ppwh PPW,  pp (pp gap) -o quest Q.
quest (quest NPWH VP)   --> npwh NPWH, vp VP.

npwh (npwh NWH) --> nwh NWH.
npwh (npwh which OptAP N) --> (which::nil), optap OptAP, n N.
apwh (apwh AWH) --> awh AWH.
ppwh (ppwh PWH) --> pwh PWH.

sbar (sbar that S) --> (that::nil), sent S.
qbar (qbar NPWH VP) --> npwh NPWH, vp VP.
qbar (qbar NPWH S)  --> npwh NPWH, np (np gap) -o sent S.

np (np PNposs) --> pnposs PNposs.
np (np Det Nposs OptPP OptRel) --> det Det, nposs Nposs, optpp OptPP, optrel OptRel.
pnposs (pnposs PN) --> pn PN.
pnposs (pnposs PN s Nposs) --> pn PN, (s::nil), nposs Nposs.
nposs (nposs OptAP N) --> optap OptAP, n N.
nposs (nposs OptAP N s Nposs) --> optap OptAP, n N, (s::nil), nposs Nposs.

vp (vp DV NP PP)   --> dv DV,   np NP, pp PP.
vp (vp TV NP)      --> tv TV,   np NP.
vp (vp IV OptPP)   --> iv IV,   optpp OptPP.
vp (vp Stv Sbar)   --> stv Stv, sbar Sbar.
vp (vp TV NP Sbar) --> tv TV,   np NP, sbar Sbar.
vp (vp Qv Qbar)    --> qv Qv,   qbar Qbar.
vp (vp Vfa VP)     --> vfa Vfa, vp VP.
vp (vp Vfa AP)     --> vfa Vfa, ap AP.

optpp (optpp epsilon) --> nil.
optpp (optpp PP)      --> pp PP.
pp (pp P NP) --> p P, np NP.

optap (optap epsilon) --> nil.
optap (optap AP)      --> ap AP.
ap (ap A) --> a A.

optrel (optrel epsilon) --> nil.
optrel (optrel Rel)     --> rel Rel.
rel (rel that VP)   --> (that::nil), {vp VP}.
rel (rel who VP)    --> (who::nil),  {vp VP}.
rel (rel that S)    --> (that::nil), {np (np gap) -o sent S}.
rel (rel whom S)    --> (whom::nil), {np (np gap) -o sent S}.
rel (rel P whom S)  --> p P, (whom::nil), {pp (pp gap) -o sent S}.
rel (rel P which S) --> p P, (which::nil), {pp (pp gap) -o sent S}.
```

Figure 6.2: A Large Lolli Definite Clause Grammar.

```
?- big_grammar --o top.

?- parse 'the program that john wrote halted' T.

?T <- s (np (det the) (nposs (n program)) (optpp epsilon)
            (optrel (rel that (s (np (pnposs (pn john)))
                                 (vp (tv wrote) (np gap))))))
        (vp (iv halted) (optpp epsilon)).


?- parse 'i told mary that john wondered who jane saw' T.

?T <- s (np (pnposs (pn i)))
        (vp (tv told) (np (pnposs (pn mary)))
            (sbar that (s (np (pnposs (pn john)))
                          (vp (qv wondered)
                              (qbar (npwh (nwh who))
                                    (s (np (pnposs (pn jane)))
                                       (vp (tv saw) (np gap)))))))) .


?- parse 'i told that john wondered who jane saw sally' T.

no


?- parse 'which computer did john write the program on' T.

?T <- q (npwh which (optap epsilon) (n computer))
        (q (vfa did) (np (pnposs (pn john)))
           (vp (dv write) (np (det the) (nposs (n program))
                              (optpp epsilon) (optrel epsilon))
               (pp (p on) (np gap)))) .
```

Figure 6.3: A Sample Interaction With The Large Lolli Definite Clause Grammar.

6.5.6 The Problem of Crossing Dependencies

Using $\mathcal{L}$ as the foundation of this sort of system induces what may be considered either a bug or a feature, depending on one's point of view. In constructing a grammar designed to admit constructions with multiple extractions from nested structures, it is not possible to block the generation of parse trees in which the filler-gap pairs are allowed to cross. Thus, when asked to parse the question:

> Which book [did you wonder who [john told *gap* that [you wrote *gap*]]]?

a version of the large grammar modified to mark gap introduction/discharge site pairs with unique names produces the following two parse trees:

```
quest (npwh which (optap epsilon) (n book) Gap_5492)
      (quest (vfa did)
             (np (pnposs (pn you)))
             (vp (qv wonder)
                 (qbar (npwh (nwh who) Gap_30922)
                       (sent (np (pnposs (pn john)))
                          (vp (tv told) (np Gap_30922)
                              (sbar that (sent (np (pnposs (pn you)))
                                               (vp (tv wrote) (np Gap_5492))
      )))))))

quest (npwh which (optap epsilon) (n book) Gap_5492)
      (quest (vfa did)
             (np (pnposs (pn you)))
             (vp (qv wonder)
                 (qbar (npwh (nwh who) Gap_30922)
                       (sent (np (pnposs (pn john)))
                             (vp (tv told) (np Gap_5492)
                                 (sbar that (sent (np (pnposs (pn you)))
                                                  (vp (tv wrote) (np Gap_30922))
      )))))).
```

In the first parse tree, which is correct, the introduction/discharge pairs are nested. In the second pair they overlap, which is not allowed in English. Crossing dependencies are, however, allowed in a limited form in some languages. Until now there have been few formalisms that could admit such constructions. Nevertheless, it is necessary to reject this ambiguity in English.

The problem is caused by the fact that $\mathcal{L}$ is a commutative logic. Thus, once the gap rules are loaded into the linear context they can be rearranged in any order. This is the same problem that occurred in the multiset rewriting example in Chapter 5. To solve this problem at the logical level will presumably require examining non-commutative variants of this logic. In the meantime, a simple extra-logical solution is available which relies on the ordering of the Lolli search procedure: modify the gap introductions to be of the form:

```
rel --> (npgap --> nil) -o s.
```

We then add explicit gap termination rules of the form:

```
np -->  once npgap.
```

This way when a gap is used, only the most recently introduced gap of the appropriate type can be used. The parse cannot re-succeed using an earlier gap. It is interesting to note that this reintroduces what GPSG calls "gap termination rules", which were, until now, unnecessary in this setting.

6.5.7 The Question of Complexity

When faced with a new grammar formalism, one of the first questions one wants answered is what the computational complexity of the accepted languages is. This is important both because it offers a lower bound on the efficiency of parsers that can be built, and because it provides an immediate feel for how likely an explanation of natural language the system might be. Interest is currently focused on systems that are slightly more than context free, such as Steedman's Combinatory Categorial Grammars [Steedman, 1988] and Joshi's Tree Adjoining Grammars [Joshi, 1986; Joshi, 1983].

It is a legitimate question to ask then, what is the computational complexity of the formalism underlying these parsers. At present we are prepared to provide only a partial answer. It is well known that fragments of propositional linear logic are far more complex than equivalent fragments of intuitionistic logic. Lincoln, et al. have shown that full propositional intuitionistic linear logic is undecidable [Lincoln *et al.*, 1990]. More recently, Kanovich has claimed that if G is a formula built out of a single propositional letter, '$\multimap$', and '&', and Γ and Δ are multisets of such formulas, then the system in which sequents are of the form:

$$!\Gamma, \Delta \longrightarrow G$$

is also undecidable [Kanovich, 1993]. This is significant, since it is clearly a sublanguage of $\mathcal{L}$. This is not necessarily bad news, though. While propositional $\mathcal{L}$ is apparently undecidable, the grammars we have constructed do not begin to utilize the full formula language. In particular, our grammar rules have had only a single level of nested implications. It seems likely that if we impose this as a formal restriction we may obtain a significantly less expressive language.

Chapter 7

An Operational Semantics of Resource Management in Proofs

At this point we have seen several examples that demonstrate the benefits of a logic programming language based on the logic $\mathcal{L}$. We have, however, said nothing about the practicality of implementing such a language.

Given the rather simple structure of the $\mathcal{L}'$ proof rules (they are all Horn clauses), it is an easy matter to build a prototype interpreter for this logic just by mapping each proof rule to a Prolog clause, with the proof context managed as a Prolog data structure. However, a serious computational issue must be addressed.

While the definition of uniform proofs guarantee that the proof process is driven by the shape of the goal formula, the sequent system still requires the proof to commit to certain decisions early. For instance, to apply the $\exists_R$ or *BC* proof rules, we must supply the term being substituted for the quantified variable. The standard technique for implementing logic programming languages is to use unification to delay this choice, and that is the technique used here. In Lolli, however, there is a related problem that does not occur in the implementation of the other languages we have discussed. Nothing in the analysis made in Chapter 4 provides any guidance when an interpreter is forced to divide up the resources in the bounded context in order to apply the $\otimes_R$ or *BC* rules in a bottom-up fashion. Trying all possible partitions of the bounded context is, of course, exponential in the number of formulae therein. Clearly, a better strategy is needed if this is to be a usable logic programming language.

Fortunately, given the restricted form of proof contexts, it is possible to view the process of proof building as one in which resource formulae are used and, if they are in the bounded part of the context, deleted. Thus, one way to attempt a proof of $G_1 \otimes G_2$ from a context is to first try to prove G_1, deleting bounded formulae as they are used in backchaining. If the search for a proof of G_1 is successful, the resulting context is then used to prove the second goal G_2. If the correct amount of resources were left to prove G_2, then the compound goal $G_1 \otimes G_2$ will have been proved without splitting the context artificially.

With this motivation, this chapter will develop an operational semantics of Lolli execution which models the consumption of resources in the root-upwards search for $\mathcal{L}$ proofs. This semantics will itself be written in terms of proof trees in a new system. It will be shown that this new system, in which the partitioning of resources is done lazily, is sound and complete relative to $\mathcal{L}$.

7.1 The *IO* System

We begin by making the following definitions:

Definition 7.1 An *IO-context* is a list of formulae (also called *resources*), each of which is either a definite formula (as defined in Chapter 4), a definite formula marked with a '!', or an instance of the new constant *del*. The list is built with the special constructors *::* and *nil*.

Definition 7.2 An *IO-sequent* is an expression of the form:

$$I\{G\}O$$

where I and O are *IO*-contexts, and G is a goal formula.

The intended reading of an *IO*-sequent is that given *IO*-context I, a proof for G can be found that returns the *IO*-context O. To make this informal notion precise, we need the following definitions regarding *IO*-contexts.

Definition 7.3 The ternary relation *pickR*(I, O, R) holds if R occurs in the *IO*-context I, and O results from replacing that occurrence of R in I with the new constant *del*; this achieves deletion. The relation also holds if $!R$ occurs in I, and I and O are equal; !'ed formulae are not deleted.

Definition 7.4 An *IO*-context O is a *subcontext* of I, denoted by the predicate *subcontext*(O, I) if O arises from replacing zero or more non-!'ed components of I with *del*.

Figure 7.1 provides a specification of the predicate $I\{G\}O$ for the propositional fragment of $\mathcal{L}$. This presentation is further simplified in that it assumes that the heads of definite formulae are atomic formulae, rather than themselves being definite formulae as allowed in Definition 4.2. While the proofs in this chapter will be in terms of this restricted system, Figure 7.2 gives the Prolog specification of the IO proof system for the full propositional system. In the Prolog specification, $I\{G\}O$ is written using the syntax `prove(I,O,G)`. Formulae are written essentially as described in Section 4.5, except that $\otimes$ is written as `x`, and $!G$ as `bang(G)`.

The Prolog specification also includes the axiomatization of the `subcontext` and `pickR` predicates, which are treated essentially as side conditions on axioms in the formal presentation. The Prolog code also implements only the propositional part of this logic since Prolog has no

$$\frac{}{I\{1\}I} \qquad \frac{\overline{\mathsf{subcontext}(O,I)}}{I\{\top\}O} \qquad \frac{I\{G\}I}{I\{!\,G\}I}$$

$$\frac{I\{G_1\}M \quad M\{G_2\}O}{I\{G_1 \otimes G_2\}O} \qquad \frac{I\{G_1\}O \quad I\{G_2\}O}{I\{G_1 \,\&\, G_2\}O}$$

$$\frac{R{::}I\{G\}\mathsf{del}{::}O}{I\{R \multimap G\}O} \qquad \frac{!\,R{::}I\{G\}\,!\,R{::}O}{I\{R \Rightarrow G\}O}$$

$$\frac{\overline{\mathsf{pickR}(I,O,A)}}{I\{A\}O} \qquad \frac{\overline{\mathsf{pickR}(I,M,G \multimap A)} \quad M\{G\}O}{I\{A\}O} \qquad \frac{\overline{\mathsf{pickR}(I,O,G \Rightarrow A)} \quad O\{G\}O}{I\{A\}O}$$

Figure 7.1: Operational Semantics for the Propositional Fragment of $\mathcal{L}$'

natural representation of object-level quantification. If λProlog were used for this specification, such quantifiers could be implemented directly using λ-abstractions. This would require just three more clauses: one for proving universally and existentially quantified goals, and one for backchaining over a universal quantifier.

In order to state the correctness of this specification, we will need the following definition:

Definition 7.5 Whenever it is the case that $\mathsf{subcontext}(O,I)$ holds, the difference, $I - O$, of two *IO*-contexts, is the pair $\langle \Gamma, \Delta \rangle$ where Γ is the multiset of all formulae R such that $!\,R$ is an element of the list I (and hence O), and Δ is the multiset of all formulae R which occur in I and where the corresponding position in O is the symbol *del*. Thus, Δ is the multiset of formulae deleted in moving from I to O.

We are now in a position to make the following claim:

Proposition 7.1 Let I and O be *IO*-contexts where O is a subcontext of I. Let $I - O$ be the pair $\langle \Gamma, \Delta \rangle$ and let G be a goal formula. Then *IO*-sequent $I\{G\}O$ is derivable if and only if $\Gamma; \Delta \longrightarrow G$ has an $\mathcal{L}'$-proof.

Proof. (Originally developed by Miller.) We first show by induction on the structure of *IO*-derivations that if $I\{G\}O$ is derivable then $\Gamma; \Delta \longrightarrow G$ has an $\mathcal{L}'$-proof.
Base Case: The two base cases of proving the goals $\mathbf{1}$ and $\top$ are immediate:

- In the first case, $I = O$, $I - I = \langle \Gamma, \varnothing \rangle$ and $\overline{I\{\mathbf{1}\}I}$. But then $\overline{\Gamma; \varnothing \longrightarrow \mathbf{1}}$ holds.
- In the second case, $\dfrac{\overline{\mathsf{subcontext}(O,I)}}{I\{\top\}O}$ and $I - O = \langle \Gamma, \Delta \rangle$. But then $\overline{\Gamma; \Delta \longrightarrow \top}$ holds.

Induction Hypothesis: In each case, if Ξ_i is an *IO*-derivation of its endsequent, $I_i\{G_i\}O_i$, and $I_i - O_i = \langle \Gamma_I, \Delta_i \rangle$, then $\Gamma_i; \Delta_i \longrightarrow G_i$ is $\mathcal{L}'$-provable.
Inductive Cases:

$\dfrac{\overset{\Xi}{I\{G\}I}}{I\{!\,G\}I}$ **is an** *IO*-**derivation:** As in the first base case, $I = I$ and $I - I = \langle \Gamma, \varnothing \rangle$. Then, by the induction hypothesis, $\Gamma; \varnothing \longrightarrow G$ has an $\mathcal{L}'$-proof. But then $\Gamma; \varnothing \longrightarrow !\,G$ is provable by the $!_R$ rule.

$$\frac{\overset{\Xi_1}{I\{G_1\}M} \quad \overset{\Xi_2}{M\{G_2\}O}}{I\{G_1 \otimes G_2\}O}$$ **is an *IO*-derivation:** Let $I - M$ be $\langle \Gamma, \Delta_1 \rangle$ and let $M - O$ be $\langle \Gamma, \Delta_2 \rangle$. By the induction hypothesis, $\Gamma; \Delta_1 \longrightarrow G_1$ and $\Gamma; \Delta_2 \longrightarrow G_2$ have $\mathcal{L}'$-proofs. Then, since $I - O = \langle \Gamma; \Delta_1 \uplus \Delta_2 \rangle$, by the $\otimes_R$ rule, so too does $\Gamma; \Delta_1, \Delta_2 \longrightarrow G_1 \otimes G_2$.

$$\frac{\overset{\Xi_1}{I\{G_1\}O} \quad \overset{\Xi_2}{I\{G_2\}O}}{I\{G_1 \,\&\, G_2\}O}$$ **is an *IO*-derivation:** Let $I - O$ be $\langle \Gamma, \Delta \rangle$. By the induction hypothesis, $\Gamma; \Delta \longrightarrow G_1$ and $\Gamma; \Delta \longrightarrow G_2$ have $\mathcal{L}'$-proofs. Then, by the $\&_R$ rule, so too does $\Gamma; \Delta \longrightarrow G_1 \,\&\, G_2$.

$$\frac{R :: I\{\overset{\Xi}{G}\}\mathsf{del} :: O}{I\{R \multimap G\}O}$$ **is an *IO*-derivation:** Let $I - O$ be $\langle \Gamma, \Delta \rangle$. Then $(R :: I) - (\mathsf{del} :: O) = \langle \Gamma, \Delta \uplus \{R\} \rangle$. Since, by the induction hypothesis, $\Gamma; \Delta, R \longrightarrow G$ has an $\mathcal{L}'$-proof, then, by $\multimap_R$, so too does $\Gamma; \Delta \longrightarrow R \multimap G$.

$$\frac{!R :: I\{\overset{\Xi}{G}\}!R :: O}{I\{R \Rightarrow G\}O}$$ **is an *IO*-derivation:** Let $I - O$ be $\langle \Gamma, \Delta \rangle$. Since, by the induction hypothesis, $\Gamma, R; \Delta \longrightarrow G$ has an $\mathcal{L}'$-proof, then, by $\Rightarrow_R$, so too does $\Gamma; \Delta \longrightarrow R \Rightarrow G$.

The three remaining cases involve the pickR predicate. Assume that $\mathsf{pickR}(I, O, R)$ holds and that $I - O$ is the pair $\langle \Gamma, \Delta \rangle$. Notice that either Δ is $\{R\}$ or Δ is empty and $R \in \Gamma$.

$$\frac{}{\mathsf{pickR}(I, O, A)}$$
$$I\{A\}O$$ **is an *IO*-derivation:** If Δ is $\{A\}$ then $\Gamma; \Delta \longrightarrow A$ is provable by the degenerate form of the *BC* rule with no premises. Otherwise, Δ is empty, $I = O$, and $!A \in I$ (and $!A \in \Gamma$). Then that same sequent is proved using the degenerate *BC* rule above an instance of *absorb* on $!A$.

$$\frac{\mathsf{pickR}(I, M, G \multimap A) \quad M\{\overset{\Xi}{G}\}O}{I\{A\}O}$$ **is an *IO*-derivation:** By the induction hypothesis, $\Gamma; \Delta' \longrightarrow G$ has an $\mathcal{L}'$-proof, Ξ', where $M - O$ is $\langle \Gamma, \Delta' \rangle$. If Δ is $\{G \multimap A\}$ then $\Gamma; \Delta, \Delta' \longrightarrow A$ has an $\mathcal{L}'$-proof built from Ξ' using the *BC* rule. Otherwise, Δ is empty and an instance of *absorb* must also be used.

$$\frac{\mathsf{pickR}(I, O, G \Rightarrow A) \quad O\{\overset{\Xi}{G}\}O}{I\{A\}O}$$ **is an *IO*-derivation:** By the induction hypothesis, $\Gamma; \varnothing \longrightarrow G$ has an $\mathcal{L}'$-proof, Ξ', where $O - O$ is $\langle \Gamma, \varnothing \rangle$. If Δ is $\{G \Rightarrow A\}$ then $\Gamma; \Delta \longrightarrow A$ has an $\mathcal{L}'$-proof built from Ξ' using the *BC* inference rule. Otherwise, Δ is empty and an instance of *absorb* must also be used.

The reverse direction of this proposition follows by simply inverting the construction described above. ■

As with the rules for $\mathcal{L}$', the rules for the *IO* system are in the form of Horn clauses, and can be seen to directly specify an interpreter for the logic of $\mathcal{L}$. But in this case the system

```
prove(I, I, 1)       :- !.
prove(I, O, erase)   :- !, subcontext(O,I).
prove(I, O, G1 & G2) :- !, prove(I,O,G1), prove(I,O,G2).
prove(I, O, R -o G)  :- !, prove(R::I,del::O,G).
prove(I, O, R => G)  :- !, prove(bang(R)::I,bang(R)::O,G).
prove(I, O, G1 x G2) :- !, prove(I,M,G1), prove(M,O,G2).
prove(I, I, bang(G)) :- !, prove(I,I,G).
prove(I, O, A)       :- pickR(I,M,R), bc(M,O,A,R).

bc(I, I, A, A).
bc(I, O, A, G -o R)  :- bc(I,M,A,R), prove(M,O,G).
bc(I, O, A, G => R)  :- bc(I,O,A,R), prove(O,O,G).
bc(I, O, A, R1 & R2) :- bc(I,O,A,R1); bc(I,O,A,R2).

pickR(bang(R)::I, bang(R)::I, R).
pickR(R::I,       del::I,     R).
pickR(S::I,       S::O,       R) :- pickR(I,O,R).

subcontext(del::O, R::I) :- subcontext(O,I).
subcontext(S::O,   S::I) :- subcontext(O,I).
subcontext(nil,    nil).
```

Figure 7.2: A Prolog Implementation of the *IO* System.

requires no non-deterministic splitting of the proof context. Consider the behavior of a Prolog interpreter attempting to prove $I\{G_1 \otimes G_2\}O$. First the interpreter tries to prove $I\{G_1\}M$, resulting in a (possibly partial) instantiation of the metavariable M. If this succeeds, then $M\{G_2\}O$ is attempted. If this second attempt fails, the interpreter retries $I\{G_1\}M$ looking for some different pattern of consumption to find, hopefully, a new instantiation for M, before re-attempting a proof of $M\{G_2\}O$.

7.2 Remaining Nondeterminism in the *IO* System

While the system presented above removes a significant unnecessary nondeterminism in the implementation of $\mathcal{L}$, it still has a significant problem. Consider an attempt to solve the query:

```
?-  a -o a -o a -o (erase , erase).
```

This results in the construction of an *IO*-derivation of the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\mathsf{subcontext}(M,(a::a::a::\mathsf{nil}))}}{(a::a::a::\mathsf{nil})\{\top\}M}\quad\dfrac{\overline{\mathsf{subcontext}((\mathsf{del}::\mathsf{del}::\mathsf{del}::\mathsf{nil}),M)}}{M\{\top\}(\mathsf{del}::\mathsf{del}::\mathsf{del}::\mathsf{nil})}}{(a::a::a::\mathsf{nil})\{\top\otimes\top\}(\mathsf{del}::\mathsf{del}::\mathsf{del}::\mathsf{nil})}}{(a::a::\mathsf{nil})\{a\multimap(\top\otimes\top)\}(\mathsf{del}::\mathsf{del}::\mathsf{nil})}}{(a::\mathsf{nil})\{a\multimap a\multimap(\top\otimes\top)\}(\mathsf{del}::\mathsf{nil})}}{\mathsf{nil}\{a\multimap a\multimap a\multimap(\top\otimes\top)\}\mathsf{nil}}$$

Unfortunately, there are eight different instantiations of the metavariable M —and hence eight different proofs— that will lead to a successful response to the query. This situation is particularly problematic in the case where this sort of goal occurs within a larger goal which fails for other reasons. In that case the system will be forced to attempt all the different solutions to this problem, even though it can't possibly affect the provability of the overall goal.

While this may seem an artificial example, in practice this situation actually occurs with some frequency. Consider the object-oriented programming example in Chapter 5. If several switches are defined, then, in the course of execution, the bounded context may be loaded with resources representing the state of a large number of objects. At the same time, using object selectors will cause a number of $\top$ goals to be executed. If an overall query fails, this may trigger an exponential number of re-evaluations of those goals.

7.3 A Lazy Operational Semantics

It should be readily apparent that the intent of using $\top$ in a goal is not to dispose of some particular set of resources, but rather to reintroduce arbitrary weakening into a particular branch of a proof. Thus if more than one instance of $\top$ occurs in a given branch, the intent is not that they divide the weakening task between them, but rather that they act idempotently.

To this end, we introduce a variation of the *IO* system called $IO^{\top}$. In this system, the rules for which are given in Figure 7.3, the output context is exactly those formulae that were not explicitly deleted by pickR. An instance of $\top$ as a goal does not effect the contexts directly, but rather sets a status flag indicating that it has been encountered, and thereby provides permission to weaken formulae not used directly in the proof.

Several new side conditions are added (and axiomatized) for this system. The simplest is given the following definition:

Definition 7.6 The *IO*-context I is the *contextintersection* of the *IO*-contexts O_1 and O_2 (which are assumed to both be, in turn, subcontexts of some fourth context), denoted by the predicate *contextintersection*(O_1, O_2, I), if I has the value *del* in those positions where either O_1 or O_2 does, and has the same value as both O_1 and O_2 where they are the same.

The or condition is used just to compute the boolean valued function of that name. To understand the meaning of the other two side conditions, it is easiest to think in terms of the execution of a Prolog meta-interpreter for the system. Some of the conditions are used simply

$$\frac{}{I[1](I,\bot)} \qquad \frac{}{I[\top](I,\top)} \qquad \frac{I[G](I,E)}{I[!\,G](I,\bot)}$$

$$\frac{I[G_1](M,E_{O_1}) \quad M[G_2](O,E_{O_2}) \quad \mathsf{or}(E_{O_1},E_{O_2},E_O)}{I[G_1 \otimes G_2](O,E_O)}$$

$$\frac{I[G_1](O_1,E_{O_1}) \quad I[G_2](O_2,E_{O_2}) \quad \mathsf{check}(E_{O_1},E_{O_2},E_O,O_1,O_2,O)}{I[G_1 \,\&\, G_2](O,E_O)}$$

$$\frac{R{::}I[G](O_R{::}O,E) \quad \mathsf{check}_2(E,R,O_R)}{I[R \multimap G](O,E)}$$

$$\frac{!\,R{::}I[G](!\,R{::}O,E)}{I[R \Rightarrow G](O,E)}$$

$$\frac{\dfrac{}{\mathsf{pickR}(I,O,A)}}{I[A](O,\bot)} \qquad \frac{\dfrac{}{\mathsf{pickR}(I,M,G \multimap A)} \quad M[G](O,E)}{I[A](O,E)}$$

$$\frac{\dfrac{}{\mathsf{pickR}(I,O,G \Rightarrow A)} \quad O[G](O,E)}{I[A](O,\bot)}$$

$$\frac{}{\mathsf{or}(\bot,\bot,\bot)} \qquad \frac{}{\mathsf{or}(\bot,\top,\top)} \qquad \frac{}{\mathsf{or}(\top,\bot,\top)} \qquad \frac{}{\mathsf{or}(\top,\top,\top)}$$

$$\frac{}{\mathsf{check}(\bot,\bot,\bot,O,O,O)} \qquad \frac{\dfrac{}{\mathsf{contextIntersection}(O_1,O_2,O)}}{\mathsf{check}(\top,\top,\top,O_1,O_2,O)}$$

$$\frac{\dfrac{}{\mathsf{subcontext}(O,O_1)}}{\mathsf{check}(\bot,\top,\bot,O,O_1,O)} \qquad \frac{\dfrac{}{\mathsf{subcontext}(O,O_1)}}{\mathsf{check}(\top,\bot,\bot,O_1,O,O)}$$

$$\frac{}{\mathsf{check}_2(\bot,R,\mathsf{del})} \qquad \frac{}{\mathsf{check}_2(\top,R,\mathsf{del})} \qquad \frac{}{\mathsf{check}_2(\top,R,R)}$$

Figure 7.3: An Operational Semantics for $\mathcal{L}$ with Lazy Splitting and Weakening

to set the value of the permission flag in the lower sequent. For instance, the side condition in the rule for tensor goals can be read as saying that it is sufficient for $\top$ to be encountered at the top level in either conjunct in order for weakening to be in force for the proof of the entire goal. Other conditions perform checks to ensure that the subproofs of the premises have had the proper pattern of resource consumption, and set the permission flag accordingly. For example, the side condition of the rule for & goals can be understood as follows:

- If $\top$ has not been encountered as a top level goal in the subproof of either conjunct, then each of the subproofs must have consumed the same set of resources. If so, the permission flag for the conjunction is false, and the output context is the same as that for the two conjuncts.
- If $\top$ is encountered in only one of the two subproofs, then it serves only to allow that subproof to explicitly consume fewer resources than the other subproof (because the remainder can be thought to have been weakened by the encounter with $\top$). Therefore, the permission flag for the overall conjunction is false, and the output context is the context produced by the subproof that did not encounter $\top$.
- If both subproofs encountered $\top$ as a goal, then in each subproof it serves as permission to weaken away those formulae used in the other subproof but not in that one. The set of formulae used directly for the overall proof is the union of those used directly in each subproof, so the output context is set to the intersection of the output contexts from the two subproofs. The permission flag for the conjunction is set to true, since additional formulae could be weakened by the two instances of $\top$.

Additional intuition into the meaning of the various side conditions can be gained by examining the details of the soundness and completeness proof for $IO^{\top}$which follows below. The net result of this design is that in $IO^{\top}$ the use of subcontext is restricted to cases where (in a Prolog implementation of the system) both of its arguments will be instantiated when it is called. This avoids the nondeterministic creation of subcontexts. So, for instance, the query:

```
?-  a -o a -o a -o (erase , erase).
```

has only a single $IO^{\top}$-derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{E_1 = \top}}{I[\top](I,E_1)} \quad \dfrac{\boxed{E_2 = \top}}{I[\top](I,E_2)} \quad \dfrac{\boxed{E = \top}}{\mathsf{or}(E_1,E_2,E)}}{\overbrace{(a{::}a{::}a{::}\mathsf{nil})}^{I}[\top \otimes \top](\overbrace{(a{::}a{::}a{::}\mathsf{nil})}^{I},E) \quad \overline{\mathsf{check}_2(E,a,a)}}}{(a{::}a{::}\mathsf{nil})[a \multimap (\top \otimes \top)]((a{::}a{::}\mathsf{nil}),E) \quad \overline{\mathsf{check}_2(E,a,a)}}}{(a{::}\mathsf{nil})[a \multimap a \multimap (\top \otimes \top)]((a{::}\mathsf{nil}),E) \quad \overline{\mathsf{check}_2(E,a,a)}}}{\mathsf{nil}[a \multimap a \multimap a \multimap (\top \otimes \top)](\mathsf{nil},E)}$$

7.4 Soundness and Correctness of the Lazy Semantics

Proposition 7.2 $I\{G\}O$ is provable if and only if either:

- $I[G](O,\bot)$ is provable, or
- there is an O_1 such that $I[G](O_1,\top)$ is provable, and subcontext(O,O_1).

Proof. In each direction, the proof is given in the form of a mapping —defined inductively on the structure of proofs— from one proof system to the other. Note that showing soundness in this case is somewhat harder than in the case of *IO* relative to $\mathcal{L}'$. In that case, there was a one-to-one correspondence between proofs in the two systems, only failed constructions (in which a bad partitioning is chosen) are conflated. In this case, in contrast, many valid *IO* proofs conflate to a single $IO^\top$ proof (which was, of course, the whole point of the design of the new system).

First, we show the completeness of $I[G](O,*)$ relative to $I\{G\}O$ by giving a transformation from *IO* proofs to $IO^\top$ proofs. In order to state this proof we will need the following alternate definition of the difference of two *IO*-contexts (this one is in fact closer to the normal sense of difference), and of the masked-sum of two contexts:

Definition 7.7 Whenever it is the case that *subcontext*(O,I), the *difference* $I-O$ is defined as an *IO*-context with *del* in those positions where I and O agree, and the same value as I in those positions where they disagree.

Definition 7.8 In the case where I and I' are *IO*-contexts of equal length, and I' is non-*del* only in positions in which I is *del*, then the *masked-sum* $I+I'$ is an *IO*-context with exactly the non-*del* elements of both I and I'. If I is longer than I' then I' is *del*-extended to the appropriate length.

We will also need the following observations, which could be proved by simple structural induction on *IO*-derivations (resp. $IO^\top$-derivations):

Proposition 7.3 Whenever it is the case that the *IO*-sequent $I\{G\}O$ has a valid *IO*-derivation (resp. $IO^\top$-derivation), then subcontext(O,I) holds.

Proposition 7.4 Whenever it is the case that the *IO*-sequent $I\{G\}O$ has a valid *IO*-derivation (resp. $IO^\top$-derivation), then any non-del position in O can be deleted (i.e. set to del) uniformly in every *IO*-context in the derivation without effecting the validity of the derivation.

Proposition 7.5 If Ξ is an *IO*-derivation (resp. $IO^\top$-derivation) of the sequent $I\{G\}O$ (resp. $I[G](O,*)$) then if I' is an *IO*-context that is non-*del* only where I (and hence O) is *del*, the tree, $\Xi^{+I'}$, formed by replacing every *IO*-context in Ξ with its masked-sum with I', is a valid *IO*-derivation (resp. $IO^\top$-derivation).

To paraphrase the last proposition, if a given position is del in every *IO*-context in a derivation, then filling this position uniformly with some given definite formula throughout the derivation will not affect the validity of the derivation.

We are now in a position to proceed with the main proof.

Base Cases:

if $\Xi = \dfrac{}{I\{\mathbf{1}\}I}$ **then:** $\Xi^* = \dfrac{}{I[\mathbf{1}](I,\bot)}$

if $\Xi = \dfrac{\mathsf{subcontext}(O,I)}{I\{\top\}O}$ **then:** $\Xi^* = \dfrac{}{I[\top](I,\top)}$ and $\mathsf{subcontext}(O,I)$

if $\Xi = \dfrac{\mathsf{pickR}(I,O,A)}{I\{A\}O}$ **then:** $\Xi^* = \dfrac{\mathsf{pickR}(I,O,A)}{I[A](O,\bot)}$

Induction Hypothesis: If $I\{\overset{\Xi}{G}\}O$ is an IO-derivation of its endsequent, then either:

- $I[\overset{\Xi^*}{G}](O,\bot)$ is an $IO^\top$-derivation, or
- there is an O_1 such that $I[\overset{\Xi^*}{G}](O_1,\top)$ is is an $IO^\top$-derivation, and $\mathsf{subcontext}(O,O_1)$.

Inductive Cases:

$\dfrac{I\{\overset{\Xi}{G}\}I}{I\{!\,G\}I}$ **is an** IO**-derivation:** Then, by the induction hypothesis, one of the following cases holds:

- $I[\overset{\Xi^*}{G}](I,\bot)$, and therefore:
$$\frac{I[\overset{\Xi^*}{G}](I,\bot)}{I[!\,G](I,\bot)}$$
- $I[\overset{\Xi^*}{G}](O,\top)$ and $\mathsf{subcontext}(I,O)$, but then, since, by analogy to Proposition 7.3, the provability of $A[G](B,*)$ implies $\mathsf{subcontext}(B,A)$, we may assume $\mathsf{subcontext}(O,I)$. Therefore it must be that $O = I$. So, we have:
$$\frac{I[\overset{\Xi^*}{G}](I,\top)}{I[!\,G](I,\bot)}$$

$\dfrac{I\{\overset{\Xi_1}{G_1}\}M \quad M\{\overset{\Xi_2}{G_2}\}O}{I\{G_1 \otimes G_2\}O}$ **is an** IO**-derivation:** Then, by the induction hypothesis, one of four cases holds:

- $I[\overset{\Xi_1{}^*}{G_1}](M,\bot)$ and $M[\overset{\Xi_2{}^*}{G_2}](O,\bot)$, and therefore:
$$\frac{I[\overset{\Xi_1{}^*}{G_1}](M,\bot) \quad M[\overset{\Xi_2{}^*}{G_2}](O,\bot) \quad \dfrac{}{\mathsf{or}(\bot,\bot,\bot)}}{I[G_1 \otimes G_2](O,\bot)}$$

- $I[G_1]^{*}(M, \bot)$ and $M[G_2]^{*}(O_1, \top)$ and $\mathsf{subcontext}(O, O_1)$, and therefore:

$$\frac{I[G_1]^{*}(M, \bot) \quad M[G_2]^{*}(O_1, \top) \quad \overline{\mathsf{or}(\bot, \top, \top)}}{I[G_1 \otimes G_2](O_1, \top)}, \text{ and } \mathsf{subcontext}(O, O_1)$$

- $I[G_1]^{*}(M_1, \top)$ and $\mathsf{subcontext}(M, M_1)$ and $M[G_2]^{*}(O, \bot)$. Therefore, if $M_- = M_1 - M$ then:

$$\frac{I[G_1]^{*}(M_1, \top) \quad M_1[G_2]^{*} + M_-(O + M_-, \bot) \quad \overline{\mathsf{or}(\top, \bot, \top)}}{I[G_1 \otimes G_2](O + M_-, \top)}, \text{ and } \mathsf{subcontext}(O, O + M_-)$$

- $I[G_1]^{*}(M_1, \top)$ and $\mathsf{subcontext}(M, M_1)$ and $M[G_2]^{*}(O_1, \top)$ and $\mathsf{subcontext}(O, O_1)$. So, if $M_- = M_1 - M$, then:

$$\frac{I[G_1]^{*}(M_1, \top) \quad M_1[G_2]^{*} + M_-(O_1 + M_-, \top) \quad \overline{\mathsf{or}(\top, \top, \top)}}{I[G_1 \otimes G_2](O_1 + M_-, \top)}, \text{ and } \mathsf{subcontext}(O, O_1 + M_-)$$

$\dfrac{\Xi_1 \quad \Xi_2}{I\{G_1\}O \quad I\{G_2\}O}{\over I\{G_1 \,\&\, G_2\}O}$ **is an** *IO*-**derivation:** Then, by the induction hypothesis, one of four cases holds:

- $I[G_1]^{*}(O, \bot)$ and $I[G_2]^{*}(O, \bot)$, and therefore:

$$\frac{I[G_1]^{*}(O, \bot) \quad I[G_2]^{*}(O, \bot) \quad \overline{\mathsf{check}(\bot, \bot, \bot, O, O, O)}}{I[G_1 \,\&\, G_2](O, \bot)}$$

- $I[G_1]^{*}(O, \bot)$ and $I[G_2]^{*}(O_1, \top)$ and $\mathsf{subcontext}(O, O_1)$, and therefore:

$$\frac{I[G_1]^{*}(O, \bot) \quad I[G_2]^{*}(O_1, \top) \quad \dfrac{\overline{\mathsf{subcontext}(O, O_1)}}{\mathsf{check}(\bot, \top, \bot, O, O_1, O)}}{I[G_1 \,\&\, G_2](O, \bot)}$$

- $I[G_1]^{*}(O_1, \top)$ and $\mathsf{subcontext}(O, O_1)$ and $I[G_2]^{*}(O, \bot)$, and therefore:

$$\frac{I[G_1]^{*}(O_1, \top) \quad I[G_2]^{*}(O, \bot) \quad \dfrac{\overline{\mathsf{subcontext}(O, O_1)}}{\mathsf{check}(\top, \bot, \bot, O_1, O, O)}}{I[G_1 \,\&\, G_2](O, \bot)}$$

- $I[G_1]\overset{\Xi_1}{(}O_1, \top)$ and $\mathsf{subcontext}(O, O_1)$ and $I[G_2]\overset{\Xi_2}{(}O_2, \top)$ and $\mathsf{subcontext}(O, O_2)$. So, if $O_\cap = O_1 \cap O_2$:

$$\frac{I[G_1]\overset{\Xi_1}{(}O_1, \top) \quad I[G_2]\overset{\Xi_2}{(}O_2, \top) \quad \dfrac{\overline{\mathsf{contextIntersection}(O_1, O_2, O_\cap)}}{\mathsf{check}(\top, \top, \top, O_1, O_2, O_\cap)}}{I[G_1 \,\&\, G_2](O_\cap, \top)}\ , \text{ and } \mathsf{subcontext}(O, O_\cap)$$

$\dfrac{R::I\{\overset{\Xi}{G}\}\mathsf{del}::O}{I\{R \multimap G\}O}$ **is an** *IO*-**derivation:** Then, by the induction hypothesis, one of three cases holds:

- $R::I[G]\overset{\Xi}{(}\mathsf{del}::O, \bot)$, and therefore:

$$\frac{R::I[G]\overset{\Xi}{(}\mathsf{del}::O, \bot) \quad \overline{\mathsf{check}_2(\bot, R, \mathsf{del})}}{I[R \multimap G](O, \bot)}$$

- $R::I[G]\overset{\Xi}{(}\mathsf{del}::O_1, \top)$ and $\mathsf{subcontext}(\mathsf{del}::O, \mathsf{del}::O_1)$, and therefore:

$$\frac{R::I[G]\overset{\Xi}{(}\mathsf{del}::O_1, \top) \quad \overline{\mathsf{check}_2(\top, R, \mathsf{del})}}{I[R \multimap G](O_1, \top)}\ , \text{ and } \mathsf{subcontext}(O, O_1)$$

- $R::I[G]\overset{\Xi}{(}R::O_1, \top)$ and $\mathsf{subcontext}(\mathsf{del}::O, R::O_1)$, and therefore:

$$\frac{R::I[G]\overset{\Xi}{(}R::O_1, \top) \quad \overline{\mathsf{check}_2(\top, R, R)}}{I[R \multimap G](O_1, \top)}\ , \text{ and } \mathsf{subcontext}(O, O_1)$$

$\dfrac{!R::I\{\overset{\Xi}{G}\}!R::O}{I\{R \Rightarrow G\}O}$ **is an** *IO*-**derivation:** Then, by the induction hypothesis, one of two cases holds:

- $!R::I[G]\overset{\Xi}{(}!R::O, \bot)$, and therefore:

$$\frac{!R::I[G]\overset{\Xi}{(}!R::O, \bot)}{I[R \Rightarrow G](O, \bot)}$$

- $!R::I[G]\overset{\Xi}{(}!R::O_1, \top)$ and $\mathsf{subcontext}(!R::O, !R::O_1)$, and therefore:

$$\frac{!R::I[G]\overset{\Xi}{(}!R::O_1, \top)}{I[R \Rightarrow G](O_1, \top)}\ , \text{ and } \mathsf{subcontext}(O, O_1)$$

$\dfrac{\overline{\mathsf{pickR}(I, M, G \multimap A)} \quad M\{G\}^{\Xi} O}{I\{A\}O}$ **is an** *IO*-**derivation:** Then, by the induction hypothesis, one of two cases holds:

- $M[G]^{\Xi *}(O, \bot)$, and therefore:

$$\dfrac{\overline{\mathsf{pickR}(I, M, G \multimap A)} \quad M[G]^{\Xi *}(O, \bot)}{I[A](O, \bot)}$$

- $M[G]^{\Xi *}(O_1, \top)$ and $\mathsf{subcontext}(O, O_1)$, and therefore:

$$\dfrac{\overline{\mathsf{pickR}(I, M, G \multimap A)} \quad M[G]^{\Xi *}(O_1, \top)}{I[A](O_1, \top)}, \text{ and } \mathsf{subcontext}(O, O_1)$$

$\dfrac{\overline{\mathsf{pickR}(I, O, G \Rightarrow A)} \quad O\{G\}^{\Xi} O}{I\{A\}O}$ **is an** *IO*-**derivation:** Then, by the induction hypothesis, one of two cases holds:

- $O[G]^{\Xi *}(O, \bot)$, and therefore:

$$\dfrac{\overline{\mathsf{pickR}(I, O, G \multimap A)} \quad O[G]^{\Xi *}(O, \bot)}{I[A](O, \bot)}$$

- $O[G]^{\Xi *}(O_1, \top)$ and $\mathsf{subcontext}(O, O_1)$. But then, as in the first inductive case, it must be that $\mathsf{subcontext}(O_1, O)$, and therefore $O_1 = O$. So we have:

$$\dfrac{\overline{\mathsf{pickR}(I, O, G \multimap A)} \quad O[G]^{\Xi *}(O, \top)}{I[A](O, \bot)}$$

Next, we show the soundness of $I[G](O, *)$ relative to $I\{G\}O$. That is, if $I[G](O, \bot)$ is $IO^{\top}$-derivable, then $I\{G\}O$ is *IO*-derivable, and if $I[G](O, \top)$ is $IO^{\top}$-derivable, then for all O_1 such that $\mathsf{subcontext}(O_1, O)$, $I\{G\}O_1$ is *IO*-derivable. We will show both cases (where $* = \top, \bot$) simultaneously, though for some of the rules (including all of the base cases) there is only a single case.

Base Cases:

If $\Xi = \overline{I[\mathbf{1}](I, \bot)}$ **then:** $\Xi^{+} = \overline{I\{\mathbf{1}\}I}$.

If $\Xi = \dfrac{}{I[\top](I,\top)}$ **then:** for all O such that $\mathsf{subcontext}(O,I)$, $\Xi^+ = \dfrac{\mathsf{subcontext}(O,I)}{I\{\top\}O}$ is an IO-derivation.

If $\Xi = \dfrac{\mathsf{pickR}(I,O,A)}{I[A](O,\bot)}$ **then:** $\Xi^+ = \dfrac{\mathsf{pickR}(I,O,A)}{I\{A\}O}$.

Induction hypothesis: If $I[G](O,E)$ (with derivation Ξ) is an $IO^\top$-derivation of its endsequent, then either:

- $E = \bot$ and $I\{G\}O$ (with derivation Ξ^+) is an IO-derivation, or
- $E = \top$ and for all O_1 such that $\mathsf{subcontext}(O_1,O)$, $I\{G\}O_1$ (with derivation Ξ^+) is an IO-derivation.

Inductive Cases:

$\dfrac{I[G](I,E)\ \Xi}{I[!\,G](I,\bot)}$ **is an** $IO^\top$**-derivation:** Then, by the induction hypothesis, one of two cases holds:

- $E = \bot$ and $I\{G\}I$ (with derivation Ξ^+), and therefore:
$$\frac{I\{G\}I\ \Xi^+}{I\{!\,G\}I}$$
- $E = \top$ and for all O_1 such that $\mathsf{subcontext}(O_1,I)$, $I\{G\}O_1$ (with derivation Ξ^+) is an IO-derivation. But then since $\mathsf{subcontext}(I,I)$, we certainly have
$$\frac{I\{G\}I\ \Xi^+}{I\{!\,G\}I}$$

$\dfrac{\mathsf{pickR}(I,M,G \multimap A) \quad M[G](O,E)\ \Xi}{I[A](O,E)}$ **is an** $IO^\top$**-derivation:** Then, by the induction hypothesis, one of two cases holds:

- $E = \bot$ and $M\{G\}O$ (with derivation Ξ^+), and therefore:
$$\frac{\mathsf{pickR}(I,M,G \multimap A) \quad M\{G\}O\ \Xi^+}{I\{A\}O}$$
- $E = \top$ and for all O_1 such that $\mathsf{subcontext}(O_1,O)$, $M\{G\}O_1$ (with derivation Ξ^+), and therefore:

  for all O_1 such that $\mathsf{subcontext}(O_1,O)$, $\dfrac{\mathsf{pickR}(I,M,G \multimap A) \quad M\{G\}O_1\ \Xi^*}{I\{A\}O_1}$

$\dfrac{\overline{\mathsf{pickR}(I, O, G \Rightarrow A)} \quad O[G]\overset{\Xi}{(}O, E)}{I[A](O, \bot)}$ **is an $IO^{\top}$-derivation:** Then, by the induction hypothesis, one of two cases holds:

- $E = \bot$ and $O\overset{\Xi^{+}}{\{}G\}O$, and therefore:

$$\dfrac{\overline{\mathsf{pickR}(I, O, G \multimap A)} \quad O\overset{\Xi^{+}}{\{}G\}O}{I\{A\}O}$$

- $E = \top$ and for all O_1 such that $\mathsf{subcontext}(O_1, O)$, $O\overset{\Xi^{+}}{\{}G\}O_1$, but then, as in the first inductive case, we know $\mathsf{subcontext}(O, O)$, and therefore:

$$\dfrac{\overline{\mathsf{pickR}(I, O, G \multimap A)} \quad O\overset{\Xi^{+}}{\{}G\}O}{I\{A\}O}$$

$\dfrac{I\overset{\Xi_1}{[}G_1](M, E_{O_1}) \quad M\overset{\Xi_2}{[}G_2](O, E_{O_2}) \quad \overline{\mathsf{or}(E_{O_1}, E_{O_2}, E_O)}}{I[G_1 \otimes G_2](O, E_O)}$ **is an $IO^{\top}$-derivation:** Then, by the induction hypothesis, one of four cases holds:

- $E_{O_1} = E_{O_2} = E_O = \bot$ and $I\overset{\Xi_1{}^{+}}{\{}G_1\}M$ and $M\overset{\Xi_2{}^{+}}{\{}G_2\}O$, and therefore:

$$\dfrac{I\overset{\Xi_1{}^{+}}{\{}G_1\}M \quad M\overset{\Xi_2{}^{+}}{\{}G_2\}O}{I\{G_1 \otimes G_2\}O}$$

- $E_{O_1} = \bot$ and $E_{O_2} = E_O = \top$ and $I\overset{\Xi_1{}^{+}}{\{}G_1\}M$ and for all O_1 such that $\mathsf{subcontext}(O_1, O)$, $M\overset{\Xi_2{}^{+}}{\{}G_2\}O_1$, and therefore:

$$\text{for all } O_1 \text{ such that } \mathsf{subcontext}(O_1, O), \quad \dfrac{I\overset{\Xi_1{}^{+}}{\{}G_1\}M \quad M\overset{\Xi_2{}^{+}}{\{}G_2\}O_1}{I\{G_1 \otimes G_2\}O_1}$$

- $E_{O_2} = \bot$ and $E_{O_1} = E_O = \top$ and for all M_1 such that $\mathsf{subcontext}(M_1, M)$, $I\overset{\Xi_1{}^{+}}{\{}G_2\}M_1$ and $M\overset{\Xi_2{}^{+}}{\{}G_2\}O$. Now, by Proposition 7.4, we may assume that for every IO-context such that $\mathsf{subcontext}(O_1, O)$, and for every corresponding M_1 constructed by deleting the

same resources from M, $M_1\{G_2\}O_1$ has a valid IO-derivation. Therefore:

for all O_1 such that $\mathsf{subcontext}(O_1, O)$, and corresponding M_1, $\dfrac{\overset{\Xi_1{}^+}{I\{G_1\}M_1} \quad \overset{\Xi_2{}^+}{M_1\{G_2\}O_1}}{I\{G_1 \otimes G_2\}O_1}$

- $E_{O_1} = E_{O_2} = E_O = \top$ and for all M_1 such that $\mathsf{subcontext}(M_1, M)$, $\overset{\Xi_1{}^+}{I\{G_2\}M_1}$ and for all O_1 such that $\mathsf{subcontext}(O_1, O)$, $\overset{\Xi_2{}^+}{M\{G_2\}O_1}$, and therefore, since $\mathsf{subcontext}(M, M)$:

  for all O_1 such that $\mathsf{subcontext}(O_1, O)$, $\dfrac{\overset{\Xi_1{}^+}{I\{G_1\}M} \quad \overset{\Xi_2{}^+}{M\{G_2\}O_1}}{I\{G_1 \otimes G_2\}O_1}$

$\dfrac{\overset{\Xi_1}{I[G_1](O_1, E_{O_1})} \quad \overset{\Xi_2}{I[G_2](O_2, E_{O_2})} \quad \overline{\mathsf{check}(E_{O_1}, E_{O_2}, E_O, O_1, O_2, O)}}{I[G_1 \,\&\, G_2](O, E_O)}$ **is an $IO^\top$-derivation:** Then, by the induction hypothesis, one of four cases holds:

- $E_{O_1} = E_{O_2} = E_O = \bot$ and $O = O_1 = O_2$ and $\overset{\Xi_1{}^+}{I\{G_1\}O}$ and $\overset{\Xi_2{}^+}{I\{G_2\}O}$, and therefore:

$$\frac{\overset{\Xi_1{}^+}{I\{G_1\}O} \quad \overset{\Xi_2{}^+}{I\{G_2\}O}}{I\{G_1 \,\&\, G_2\}O}$$

- $E_{O_1} = E_O = \bot$ and $E_{O_2} = \top$ and $O_1 = O$ and $\mathsf{subcontext}(O, O_2)$ and $\overset{\Xi_1{}^+}{I\{G_1\}O}$ and for all O_1 such that $\mathsf{subcontext}(O_1, O_2)$, $\overset{\Xi_2{}^+}{I\{G_2\}O_1}$, and therefore, since $\mathsf{subcontext}(O, O_2)$:

$$\frac{\overset{\Xi_1{}^+}{I\{G_1\}O} \quad \overset{\Xi_2{}^+}{I\{G_2\}O}}{I\{G_1 \,\&\, G_2\}O}$$

- $E_{O_2} = E_O = \bot$ and $E_{O_1} = \top$ and $O_2 = O$ and $\mathsf{subcontext}(O, O_1)$ and for all O_{11} such that $\mathsf{subcontext}(O_{11}, O_1)$, $\overset{\Xi_1{}^+}{I\{G_2\}O_{11}}$ and $\overset{\Xi_2{}^+}{I\{G_2\}O}$. Then, since $\mathsf{subcontext}(O, O_1)$:

$$\frac{\overset{\Xi_1{}^+}{I\{G_1\}O} \quad \overset{\Xi_2{}^+}{I\{G_2\}O}}{I\{G_1 \,\&\, G_2\}O}$$

- $E_{O_1} = E_{O_2} = E_O = \top$ and $\mathsf{contextIntersection}(O_1, O_2, O)$ and for all O_{11} such that $\mathsf{subcontext}(O_{11}, O_1)$, $\overset{\Xi_1{}^+}{I\{G_2\}O_{11}}$ and for all O_{21} such that $\mathsf{subcontext}(O_{21}, O_2)$, we

know that $I\{G_2\}O_{21}$ (with $\Xi_2{}^+$), and therefore, since, by the definition of contextIntersection, O is the largest C such that $\mathsf{subcontext}(C, O_1)$ and $\mathsf{subcontext}(C, O_2)$:

$$\text{for all } O_3 \text{ such that } \mathsf{subcontext}(O_3, O), \quad \frac{I\{G_1\}O_3 \quad M\{G_2\}O_3}{I\{G_1 \,\&\, G_2\}O_3}$$

$\dfrac{R::I[G](O_R::O, E) \quad \overline{\mathsf{check}_2(E, R, O_R)}}{I[R \multimap G](O, E)}$ **is an $IO^\top$-derivation:** Then, by the induction hypothesis, one of three cases holds:

- $E = \bot$ and $O_R = \mathsf{del}$ and $R::I\{G\}\mathsf{del}::O$, and therefore:

$$\frac{R::I\{G\}\mathsf{del}::O}{I\{R \multimap G\}O}$$

- $E = \top$, $O_R = \mathsf{del}$, and for all O_1 such that $\mathsf{subcontext}(\mathsf{del}::O_1, \mathsf{del}::O)$, $R::I\{G\}\mathsf{del}::O_1$, and therefore:

$$\text{for all } O_1 \text{ such that } \mathsf{subcontext}(O_1, O), \quad \frac{R::I\{G\}\mathsf{del}::O_1}{I\{R \multimap G\}O_1}$$

- $E = \top$ and $O_R = R$ and for all O_1 and O_{R1} such that $\mathsf{subcontext}(O_{R1}::O_1, R::O)$, $R::I\{G\}O_{R1}::O_1$, and therefore:

$$\text{for all } O_1 \text{ such that } \mathsf{subcontext}(O_1, O), \quad \frac{R::I\{G\}\mathsf{del}::O_1}{I\{R \multimap G\}O_1}$$

$\dfrac{!R::I[G](!R::O, E)}{I[R \Rightarrow G](O, E)}$ **is an $IO^\top$-derivation:** Then, by the induction hypothesis, there are two cases:

- $E = \bot$ and $!R::I\{G\}!R::O$, and therefore:

$$\frac{!R::I\{G\}!R::O}{I\{R \Rightarrow G\}O}$$

- $E = \top$ and for all O_1 such that $\mathsf{subcontext}(!R::O_1, !R::O)$, $!R::I\{\overset{\Xi^+}{G}\}!R::O_1$, and therefore:

$$\text{for all } O_1 \text{ such that } \mathsf{subcontext}(O_1, O),\ \frac{!R::I\{\overset{\Xi^+}{G}\}!R::O_1}{I\{R \Rightarrow G\}O_1}$$

Therefore every $\mathcal{L}$ proof has a corresponding $IO^\top$-derivation, and vice-versa. ■

The current public release of the Lolli system is a nearly direct implementation of the $IO^\top$ semantics.

Chapter 8

Logic Programming with Multiple Context Management Schemes

The system described thus far has succeeded in meeting its design goals in many ways. It has been found to be an attractive system for implementing a variety of programs in which the management of clausal resources during execution is of interest. However, in many cases the system has proven to be less than ideal, being too restrictive. In some cases the programmer wants to bar weakening but allow contraction, in others the opposite is desired. While both these situations can be simulated to some extent in the existing system, the programs that result are not as clear as one would hope, and their execution profiles may be less than ideal. In this chapter, therefore, we introduce a further refinement of the system $\mathcal{L}$ in which there are now four separate contexts:

Linear: Neither contraction nor weakening are available.

Affine: Only weakening is available; clauses may be discarded but not duplicated.

Relevant: Only contraction is available; clauses may be duplicated but not discarded.

Intuitionistic: Both contraction and weakening are available.

As with $\mathcal{L}$, there is an implication operator corresponding to each of the contexts used to load clauses into that context.

8.1 Motivations for a New System

To understand the limitations of Lolli as described so far, consider one of the simplest motivating examples for that system: the simulation of a toggle switch. The state of a named switch is represented by a clause for the binary relation $sw(name, value)$ which is stored in the linear context. The first version of the implementation of such a system was given in Chapter 4 as:

```
MODULE switch name initial.
LOCAL sw.

LINEAR sw initial.

setting name S :- sw S.

toggle name G :- sw on, sw off -o G.
toggle name G :- sw off, sw on -o G.
```

Unfortunately, as explained in Chapter 5, this version behaves poorly when there is more than one switch. The solution offered was to modify the definition of `setting` to include an instance of `erase`. But that solution is somewhat confusing. The problem is that the formula used to store the state of the switch cannot be weakened or contracted, while the programmer really wants to restrict only contraction. It seems that such a constraint should be indicated at the point that the clauses for the predicate are assumed, rather than at the point that the predicate is used.

If we were to augment the proof system for $\mathcal{L}'$ with a variant of the $\mathbf{1}_L$ rule from $\mathcal{ILL}$ to provide a form of weakening for just the special formula $\mathbf{1}$, as in:

$$\frac{\Gamma;\Delta \longrightarrow C}{\Gamma;\Delta,\mathbf{1} \longrightarrow C}\ \mathbf{1}_L$$

then affine reasoning (which allows weakening but not contraction) for a formula A could be simulated by replacing instances of A with instances of $(A \,\&\, \mathbf{1})$. The interpreter could weaken this formula by first using the $\&_{L_2}$ rule to select the $\mathbf{1}$ portion of the formula, and then using the new rule to discard the $\mathbf{1}$. The switch program would then be replaced by:

```
MODULE switch name initial.
LOCAL sw.

LINEAR (sw initial & true).

setting name S :- sw S.

toggle name G :- sw on, (sw off & true) -o G.
toggle name G :- sw off, (sw on & true) -o G.
```

which would have the desired behavior.

Unfortunately, this rule cannot be added to $\mathcal{L}'$ without compromising the completeness of uniform proofs and correspondingly complicating the proof procedure. Even then, the resulting programs would still be somewhat less readable than one would hope.

A similar problem occurs when the programmer wants to use relevant reasoning, as in artificial intelligence applications. In such a setting the answer to "Does A imply B?" should be "yes" only if A was actually used to demonstrate B, not if B is true regardless. On the other

hand, it is not generally of interest whether the assertion A was referenced more than once in the proof of B. Relevant behavior can be simulated in $\mathcal{L}'$ goals by adding the assumption to both contexts simultaneously. Adding it to the linear context guarantees that it must be used at least once; adding it to the intuitionistic context allows it to be used as many additional times as needed. Thus the relevant goal $A \xrightarrow{R} B$ can be replaced by the $\mathcal{L}'$ goal $A \Rightarrow (A \multimap B)$. Unfortunately the execution profile of programs encoded in this way is somewhat less than ideal, since the interpreter spends a good deal of time enforcing the linear constraint needlessly.

8.2 An Omnibus Logic

Bollen has developed a logic programming language with relevant implication in goals [Bollen, 1991]. That system, which is discussed in Chapter 10, is substantially weaker than $\mathcal{L}$, however, in that arbitrary nesting of quantifiers and implications is not allowed. In addition it says nothing about the affine and linear constraints which have been shown to have many useful applications (though they were not, of course, the topic Bollen was considering).

In this section we introduce a new system, $\mathcal{O}$, whose rules are given in Figure 8.1. The system is similar to $\mathcal{L}'$, but the left hand sides of sequents are composed of four separate contexts. Left to right these are the intuitionistic, relevant, affine, and linear contexts.

As with $\mathcal{L}'$, the structure of the axioms and the *absorb* rules determines much of the behavior of the system. In *identity* axioms, both the intuitionistic and affine contexts may have arbitrary contents, while the relevant context must be empty and the linear context must contain only the formula being matched. This enables implicit weakening in the intuitionistic and affine contexts but not the other two.

The abs_I rule makes a copy of a formula in the intuitionistic context and makes it available for use in the linear context. In contrast, the abs_A rule removes the formula being absorbed from the affine context when it is added to the linear context, so that the formula cannot be reused. Finally, the abs_R rule removes its formula from the relevant context but places copies in both the intuitionistic and linear contexts. Thus, once the formula has been used once, it can then be used zero or more additional times.

The next section takes the form of a series of propositions about the properties of $\mathcal{O}$ establishing its status relative to $\mathcal{L}$ and other systems.

8.3 The Logical Status of the System $\mathcal{O}$

The first two propositions describe the relationship between $\mathcal{L}$ and $\mathcal{O}$:

Proposition 8.1 The system $\mathcal{O}$ is complete for $\mathcal{L}$. That is, if the $\mathcal{L}$ sequent $\Gamma;\Delta \longrightarrow C$ is provable, then $\Gamma;\varnothing;\varnothing;\Delta \longrightarrow C$ is provable in $\mathcal{O}$.

$$\frac{}{\Gamma;\varnothing;\Psi;B \longrightarrow B}\ identity \qquad \frac{}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow \top}\ \top R$$

$$\frac{\Gamma,B;\Upsilon;\Psi;\Delta,B \longrightarrow C}{\Gamma,B;\Upsilon;\Psi;\Delta \longrightarrow C}\ abs_I \qquad \frac{\Gamma,B;\Upsilon;\Psi;\Delta,B \longrightarrow C}{\Gamma;\Upsilon,B;\Psi;\Delta \longrightarrow C}\ abs_R \qquad \frac{\Gamma;\Upsilon;\Psi;\Delta,B \longrightarrow C}{\Gamma;\Upsilon;\Psi,B;\Delta \longrightarrow C}\ abs_A$$

$$\frac{\Gamma;\Upsilon;\Psi;\Delta,B_1 \longrightarrow C}{\Gamma;\Upsilon;\Psi;\Delta,B_1 \,\&\, B_2 \longrightarrow C}\ \&L_1 \qquad \frac{\Gamma;\Upsilon;\Psi;\Delta,B_2 \longrightarrow C}{\Gamma;\Upsilon;\Psi;\Delta,B_1 \,\&\, B_2 \longrightarrow C}\ \&L_2 \qquad \frac{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B \quad \Gamma;\Upsilon;\Psi;\Delta \longrightarrow C}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B \,\&\, C}\ \&R$$

$$\frac{\Gamma;\Upsilon;\Psi_1;\Delta_1 \longrightarrow B \quad \Gamma;\Upsilon;\Psi_2;\Delta_2,C \longrightarrow E}{\Gamma;\Upsilon;\Psi_1,\Psi_2;\Delta_1,\Delta_2,B \multimap C \longrightarrow E}\ \multimap L \qquad \frac{\Gamma;\Upsilon;\Psi;\Delta,B \longrightarrow C}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B \multimap C}\ \multimap R$$

$$\frac{\Gamma;\varnothing;\Psi_1;\varnothing \longrightarrow B \quad \Gamma;\Upsilon;\Psi_2;\Delta,C \longrightarrow E}{\Gamma;\Upsilon;\Psi_1,\Psi_2;\Delta,B \overset{A}{\multimap} C \longrightarrow E}\ \overset{A}{\multimap} L \qquad \frac{\Gamma;\Upsilon;\Psi,B;\Delta \longrightarrow C}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B \overset{A}{\multimap} C}\ \overset{A}{\multimap} R$$

$$\frac{\Gamma;\Upsilon;\varnothing;\varnothing \longrightarrow B \quad \Gamma;\Upsilon;\Psi;\Delta,C \longrightarrow E}{\Gamma;\Upsilon;\Psi;\Delta,B \overset{R}{\multimap} C \longrightarrow E}\ \overset{R}{\multimap} L \qquad \frac{\Gamma;\Upsilon;\Psi;\Delta,B \longrightarrow C}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B \overset{R}{\multimap} C}\ \overset{R}{\multimap} R$$

$$\frac{\Gamma;\varnothing;\varnothing;\varnothing \longrightarrow B \quad \Gamma;\Upsilon;\Psi;\Delta,C \longrightarrow E}{\Gamma;\Upsilon;\Psi;\Delta,B \Rightarrow C \longrightarrow E}\ \Rightarrow L \qquad \frac{\Gamma,B;\Upsilon;\Psi;\Delta \longrightarrow C}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B \Rightarrow C}\ \Rightarrow R$$

$$\frac{\Gamma;\Upsilon;\Psi;\Delta,B[x \mapsto t] \longrightarrow C}{\Gamma;\Upsilon;\Psi;\Delta,\forall x.B \longrightarrow C}\ \forall L \qquad \frac{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B[x \mapsto c]}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow \forall x.B}\ \forall R$$

provided that c is not free in the lower sequent.

$$\frac{}{\Gamma;\varnothing;\Psi;\varnothing \longrightarrow \mathbf{1}}\ \mathbf{1}_R \qquad \frac{\Gamma;\Upsilon;\Psi_1;\Delta_1 \longrightarrow B_1 \quad \Gamma;\Upsilon;\Psi_2;\Delta_2 \longrightarrow B_2}{\Gamma;\Upsilon;\Psi_1,\Psi_2;\Delta_1,\Delta_2 \longrightarrow B_1 \otimes B_2}\ \otimes_R$$

$$\frac{\Gamma;\varnothing;\varnothing;\varnothing \longrightarrow C}{\Gamma;\varnothing;\varnothing;\varnothing \longrightarrow !C}\ !_R$$

$$\frac{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B[x \mapsto t]}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow \exists x.B}\ \exists_R \qquad \frac{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B_1}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B_1 \vee B_2}\ \vee_{R_1} \qquad \frac{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B_2}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B_1 \vee B_2}\ \vee_{R_2}$$

Figure 8.1: System $\mathcal{O}$ for Intuitionistic, Relevant, Affine, and Linear Implication

Proof. The proof is immediate, since each step in the $\mathcal{L}$ proof can be mapped directly to an $\mathcal{O}$ step by inserting the two empty contexts into the antecedents of the sequents. ∎

Proposition 8.2 The system $\mathcal{O}$ is sound for $\mathcal{L}$ augmented with the rule:

$$\frac{\Gamma;\Delta \longrightarrow C}{\Gamma;\Delta,\mathbf{1} \longrightarrow C}\ \mathbf{1}_L$$

in the sense that if C° is the result of recursively replacing all instances of $(D \stackrel{R}{\multimap} E)$ and $(D \stackrel{A}{\multimap} E)$ in C with $(D \Rightarrow (D \multimap E))$ and $((D \,\&\, \mathbf{1}) \multimap E)$, respectively, and if $\Gamma^\circ = \{C^\circ | C \in \Gamma\}$ and $\Gamma^{\&\mathbf{1}} = \{(C \,\&\, \mathbf{1}) | C \in \Gamma\}$, then, if the $\mathcal{O}$-sequent $\Gamma;\Upsilon;\Psi;\Delta \longrightarrow C$ is $\mathcal{O}$-provable, then $\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&\mathbf{1}}, \Delta^\circ \longrightarrow C^\circ$ is provable in the augmented $\mathcal{L}$.

Proof. The proof is in the form of an algorithm, inductive on the structure $\mathcal{O}$ proofs, which converts such proofs to $\mathcal{L}$ proofs (of appropriately converted versions of the same endsequent). The complete proof is given in Appendix F. ∎

It was shown in Chapter 4 that there is an encoding of hereditary Harrop formulae, and their corresponding sequents, into $\mathcal{L}$ such that a sequent is provable in intuitionistic logic if and only if the encoded sequent is provable in $\mathcal{L}$. By Propositions 8.1 and 8.2, then, the same holds true for $\mathcal{O}$. (The "if and only if" is maintained, because the new $\mathbf{1}_L$ rule in Proposition 2 will not occur in the proof of an encoded Harrop sequent.) The next step is to demonstrate that, in spite of its enrichment relative to $\mathcal{L}$, $\mathcal{O}$ still can still be considered a reasonable foundation for a logic-programming language.

In the remainder of this chapter, let the terms definite formula and goal formula refer to the definitions in Chapter 4, extended to allow arbitrary occurrences of the operators '$\stackrel{R}{\multimap}$' and '$\stackrel{A}{\multimap}$'. The rest of the propositions are stated without proof, as the proofs are all variants of those from Chapters 4 and 7.

Proposition 8.3 Let B be a goal formula, and let Γ, Υ, Ψ, and Δ be multisets of definite formulae. Then the sequent $\Gamma;\Upsilon;\Psi;\Delta \longrightarrow C$ has a proof in $\mathcal{O}$ if and only if it has a uniform proof in $\mathcal{O}$.

8.4 Backchaining in $\mathcal{O}$

As with $\mathcal{O}'$ it is possible to replace the left-hand operator rules with a single rule for backchaining. Consider the following definition: let the syntactic variable D range over the set of definite formulae, then redefine $\|D\|$ as the smallest set of quintuples of the form $\langle \Gamma, \Upsilon, \Psi, \Delta, D' \rangle$ where Γ, Υ, Ψ, and Δ are multisets of goal formulae, such that:

1. $\langle \varnothing, \varnothing, \varnothing, \varnothing, D \rangle \in \|D\|$,
2. if $\langle \Gamma, \Upsilon, \Psi\Delta, D_1 \,\&\, D_2 \rangle \in \|D\|$ then both $\langle \Gamma, \Upsilon, \Psi, \Delta, D_1 \rangle \in \|D\|$ and $\langle \Gamma, \Upsilon, \Psi, \Delta, D_2 \rangle \in \|D\|$,
3. if $\langle \Gamma, \Upsilon, \Psi, \Delta, \forall x.D' \rangle \in \|D\|$ then for all closed terms t, $\langle \Gamma, \Upsilon, \Psi, \Delta, D'[x \mapsto t] \rangle \in \|D\|$,

$$\frac{\begin{array}{rcl}\Gamma;\varnothing;\varnothing;\varnothing \longrightarrow G_{\Gamma 1} & \cdots & \Gamma;\varnothing;\varnothing;\varnothing \longrightarrow G_{\Gamma m} \\ \Gamma;\Upsilon;\varnothing;\varnothing \longrightarrow G_{\Upsilon 1} & \cdots & \Gamma;\Upsilon;\varnothing;\varnothing \longrightarrow G_{\Upsilon n} \\ \Gamma;\varnothing;\Psi_1;\varnothing \longrightarrow G_{\Psi 1} & \cdots & \Gamma;\varnothing;\Psi_o;\varnothing \longrightarrow G_{\Psi o} \\ \Gamma;\Upsilon;\Psi_{o+1};\Delta_1 \longrightarrow G_{\Delta 1} & \cdots & \Gamma;\Upsilon;\Psi_{o+p};\Delta_p \longrightarrow G_{\Delta p}\end{array}}{\Gamma;\Upsilon;\Psi_1,\ldots,\Psi_{o+p};\Delta_1,\ldots,\Delta_m, D \longrightarrow A}\ BC$$

provided $m, n, o, p \geq 0$, A is atomic, and
$\langle\{G_{\Gamma 1},\ldots,G_{\Gamma m}\},\{G_{\Upsilon 1},\ldots,G_{\Upsilon n}\},\{G_{\Psi 1},\ldots,G_{\Psi o}\},\{G_{\Delta 1},\ldots,G_{\Delta p}\},A\rangle \in \|D\|$.

Figure 8.2: Backchaining for the proof system $\mathcal{O}$.

4. if $\langle\Gamma,\Upsilon,\Psi,\Delta,G \Rightarrow D'\rangle \in \|D\|$ then $\langle\Gamma \uplus \{G\},\Upsilon,\Psi,\Delta,D'\rangle \in \|D\|$, and
5. if $\langle\Gamma,\Upsilon,\Psi,\Delta,G \stackrel{R}{-\!\!\circ} D'\rangle \in \|D\|$ then $\langle\Gamma,\Upsilon \uplus \{G\},\Psi,\Delta,D'\rangle \in \|D\|$.
6. if $\langle\Gamma,\Upsilon,\Psi,\Delta,G \stackrel{A}{-\!\!\circ} D'\rangle \in \|D\|$ then $\langle\Gamma,\Upsilon,\Psi \uplus \{G\},\Delta,D'\rangle \in \|D\|$.
7. if $\langle\Gamma,\Upsilon,\Psi,\Delta,G \multimap D'\rangle \in \|D\|$ then $\langle\Gamma,\Upsilon,\Psi,\Delta \uplus \{G\},D'\rangle \in \|D\|$.

Let $\mathcal{O}'$ be the proof system that results from replacing the *identity*, $\multimap_L$, $\stackrel{R}{-\!\!\circ}_L$, $\stackrel{A}{-\!\!\circ}_L$,$\Rightarrow_L$, $\&_L$, and $\forall_L$ rules in Figure 8.1 with the *backchaining* inference rule in Figure 8.2.

Proposition 8.4 Let G be a goal formula, and let Γ, Υ, Ψ and Δ be multisets of definite formulae. The sequent $\Gamma;\Upsilon;\Psi;\Delta \longrightarrow G$ has a proof in $\mathcal{O}$ if and only if it has a proof in $\mathcal{O}'$.

It is important to note that the fact that the proofs of properties of $\mathcal{O}$ (and $\mathcal{L}$) are relatively simple is the result of the careful restriction of these systems to a few well behaved operators. Any attempt to integrate these different forms of reasoning over a broad set of operators is likely to prove quite difficult. Witness the complexity of Girard's system $\mathcal{LU}$, which attempts to unify classical and intuitionistic reasoning [Girard, 1991]. The fact that a useful language results from this work demonstrates that sequential logic programming is really based mostly on the logic of implication.

8.5 An Extension of the *IO* System to $\mathcal{O}$

As with $\mathcal{L}$, the completeness of uniform proofs is not enough to yield an efficient interpreter for the resulting programming language, due to the need to partition the affine and linear contexts in applying many of the systems rules. Fortunately, the operational semantics developed in Chapter 7 can easily be extended to handle the logic of $\mathcal{O}$.

Figure 8.3 gives a variant of the *IO* system modified for the enriched language of $\mathcal{O}$. This system makes use of a new marker, '$\stackrel{!}{R}$' used when a formula is added to the *IO*-context by the relevant implication '$\stackrel{R}{-\!\!\circ}$'. The axiomatizations of subcontext, Asubcontext, Rsubcontext, and pickR are also included for clarity. The $IO^{\top}$ system can be similarly extended.

$$\frac{}{I\{1\}I} \qquad \frac{\overline{\mathsf{subcontext}(O,I)}}{I\{\top\}O} \qquad \frac{I\{G\}I}{I\{!\,G\}I}$$

$$\frac{I\{G_1\}M \qquad M\{G_2\}O}{I\{G_1 \otimes G_2\}O} \qquad \frac{I\{G_1\}O \qquad I\{G_2\}O}{I\{G_1 \,\&\, G_2\}O}$$

$$\frac{R{::}I\{G\}\mathsf{del}{::}O}{I\{R \multimap G\}O} \qquad \frac{!\,R{::}I\{G\}\,!\,R{::}O}{I\{R \Rightarrow G\}O}$$

$$\frac{\overset{!}{R}\,R{::}I\{G\}\,!\,R{::}O}{I\{R \multimap G\}O} \qquad \frac{\overset{!}{A}\,R{::}I\{G\}\,\overset{!}{A}\,R{::}O}{I\{R \overset{A}{\multimap} G\}O} \qquad \frac{\overset{!}{A}\,R{::}I\{G\}\mathsf{del}{::}O}{I\{R \overset{A}{\multimap} G\}O}$$

$$\frac{\overline{\mathsf{pickR}(I,O,A)}}{I\{A\}O} \qquad \frac{\overline{\mathsf{pickR}(I,M,G \multimap A)} \quad M\{G\}O}{I\{A\}O} \qquad \frac{\overline{\mathsf{pickR}(I,O,G \Rightarrow A)} \quad O\{G\}O}{I\{A\}O}$$

$$\frac{\overline{\mathsf{pickR}(I,M,G \overset{R}{\multimap} A)} \quad M\{G\}O \quad \overline{\mathsf{Rsubcontext}(O,M)}}{I\{A\}O}$$

$$\frac{\overline{\mathsf{pickR}(I,M,G \overset{A}{\multimap} A)} \quad M\{G\}O \quad \overline{\mathsf{Asubcontext}(O,M)}}{I\{A\}O}$$

$$\frac{\mathsf{Asubcontext}(I,I)}{\mathsf{Asubcontext}(\mathsf{del}{::}I, \overset{!}{A}\,R{::}I)} \qquad \frac{\mathsf{Asubcontext}(I,I)}{\mathsf{Asubcontext}(R{::}I, R{::}I)}$$

$$\frac{\mathsf{Asubcontext}(I,I)}{\mathsf{Rsubcontext}(!R{::}I, \overset{!}{R}\,R{::}I)} \qquad \frac{\mathsf{Asubcontext}(I,I)}{\mathsf{Rsubcontext}(R{::}I, R{::}I)}$$

$$\overline{\mathsf{pickR}(!R{::}I, !R{::}I, R)} \qquad \overline{\mathsf{pickR}(\overset{!}{R}\,R{::}I, !R{::}I, R)} \qquad \overline{\mathsf{pickR}(\overset{!}{A}\,R{::}I, \mathsf{del}{::}I, R)}$$

$$\frac{R \neq !R \quad R \neq \overset{!}{R}\,R \quad R \neq \overset{!}{A}\,R}{\mathsf{pickR}(R{::}I, \mathsf{del}{::}I, R)} \qquad \frac{\mathsf{pickR}(I,I,R)}{\mathsf{pickR}(R'{::}I, R'{::}I, R)}$$

Figure 8.3: Operational Semantics for the Propositional Fragment of $\mathcal{O}$

8.6 Using the New Features of $\mathcal{O}$

Returning to the original motivation for developing this extension of $\mathcal{L}$, programs can now be written which use the new forms of reasoning directly. For instance, if affine implication is given the concrete syntax '--@',[1] and the new declarations AFFINE is added to the module system, then the switch example can be written as:

```
MODULE switch name initial.
LOCAL sw.

AFFINE (sw initial & true).

setting name S :- sw S.

toggle name G :- sw on, sw off --@ G.
toggle name G :- sw off, sw on --@ G.
```

which will behave properly, even when multiple switches have been defined.

If relevant implication is given the concrete syntax '->>' (and the new declarations RELEVANT is added to the module system) then it is possible to implement relevant reasoning systems like the following one, which is taken from Bollen's article [Bollen, 1991].

```
MODULE pollution.

state zone1 (downwind-of zone2).
state zone4 toxic-dump.
state zone4 populated.

state Z polluted :- state Z1 factory, state Z (downwind-of Z1).
state Z danger-to-pop :- state Z toxic-dump, state Z populated.
```

In this setting we can have the following interaction:

```
?- pollution --o top.
?- state zone2 factory ->> state polluted zone2.
yes.
?- state zone2 factory ->> state danger-to-pop zone4.
no.
?- state danger-to-pop zone4.
yes.
```

While this particular interaction does not actually make use of the allowed contraction, it is not hard to conceive of queries that would.

[1] Finding a reasonable concrete syntax for the new arrows of this system has been difficult. The hope is that the at-sign in '--@' will at least be mnemonic for 'affine'. Suggestions for better choices are welcome.

Chapter 9

A Functional Interpreter

In figure 7.2 Prolog code was given that implements an interpreter for the propositional fragment of the language. As was mentioned in the accompanying text, it is simple to extend this implementation to the propositional language if it is done in λProlog.

However, while λProlog is an excellent tool for experimenting with prototype interpreters, it lacks certain tools (such as an op declaration) that are useful for building practical systems. Programming large examples entirely in abstract syntax can be cumbersome. Therefore a practical prototype interpreter for the full language has been built using ML as the implementation language.

The decision to use a functional language (which is closer to imperitave programming than is logic programming) for the implementation was made partially in hope that it might be helpful in later understanding how to compile the language. In fact, comparing the Prolog and ML implementations has provided some food for thought about what the proper execution model of the language should be.

9.1 Logic Programming Interpreters via Continuations

The design of the interpreter is based on a recent paper of Elliot and Pfenning in which they describe ML interpreters for a series of successively richer logic programming languages, starting with a propositional Horn clause interpreter and ending with an interpreter for a significant fragment of λProlog [Elliott and Pfenning, 1991]. The core implementation technique in all these interpreters is based on the notion of "success continuation".[1] The essential idea is that incidents corresponding to failure in the proof search process initiate a function return, while successe causes some designated continuation function to be called. Thus the ML function return stack is used to manage the proof search backtrack stack.

[1] Pfenning credits the idea of success continuations to Carlsson [Carlsson, 1984]. But the technique appears to be older than that. For instance it appears to have been used as the implementation model for Mellish's implementation of PopLog several years earlier.

Suppose that the clauses of a Horn clause program are stored as a list of ML data structures, either `Datom`'s (for atomic assertions) or `Dimpl`'s for implications. Then if `A` is an atomic goal, `subst` is the current substitution, and `succ_cont` is the function we want to call after successfully proving `A`, then the code to prove `A` might be:

```
fun solve_atom A nil _ _ = ()
  | solve_atom A (Datom(A1) :: prog) subst succ_cont =
                    unify A A1 subst succ_cont
                    ; solve_atom A prog subst succ_cont
  | solve_atom A (Dimpl(G,A1) :: prog) subst succ_cont =
                    unify A A1 subst
                          (fn new_subst => solve G new_subst succ_cont)
                    ; solve_atom A prog subst succ_cont
```

The three cases of this code can be understood as follows:

- If there are no clauses left to choose from (ie. the program is `nil`) then we must fail. So the function simply terminates (with `unit` as its return value).
- If the next clause in the program is an atomic fact, `unify` is called to see if it matches the goal atom. If they can be unified, `unify` will call the success continuation, passing it the new augmented substitution that resulted from the unification. If, however, unification fails, `unify` simply returns, in which case the second part of the sequence is called to search the program for further matches. This will also happen if unification at first succeeds, but the success continuation later fails and returns. In other words, the semicolon in the ML code corresponds to a choice point in the proof search.
- If the next clause in the program is an implication, `unify` is called to see if the head of the clause matches the goal atom. In this case, if unification succeeds we need to solve the body of the clause before we can go ahead and call the outer success continuation. Thus `unify` is given an augmented success continuation that first solves `G`, the body of the clause, and then calls the original continuation. As with the last case, a choice point is laid down to search for alternate matches.

With minor variations this basic idea is powerful enough to implement a variety of logic programming languages. To implement hereditary Harrop formulae, two changes must be made. First, a list of active eigen-variables (the variable `uvars` in the code fragment below) must be passed around along with the other parameters so that the eigen-variable constraint can be checked during unifications. Further, since the program is no longer stable during proof search —it grows with each goal implication— success continuations must accept the program as a parameter along with the growing substitution. The interpreter for linear logic programs is more complicated since programs both grow and contract. In particular, the code to handle clause selection must also manage the deletion of bounded resources as they are selected, and their un-deletion upon backtracking. But the basic idea is the same; the modified

program is passed as a parameter to the success continuation. So, for instance, the code to handle a tensor goal is:

```
fun solve ...
  | solve (Gtensor(g1,g2)) prog uvars subst sc =
              solve g1 prog uvars subst
                    (fn newsubst => (fn newprog =>
                       solve g2 newprog uvars newsubst sc))
  | solve ...
```

9.2 In Search of a Concrete Syntax

In building the prototype interpreter, the selection of a reasonable concrete syntax was an important step that is, to an extent, still ongoing. Initially, `x` and `&` were used as syntax for ⊗ and & respectively, `-o`, `o-`, `=>`, and `<=` were used for the various forms of linear and intuitionistic implication, and the special form `filename --o goal` was used to load a module. This syntax was chosen in order to make it clear to the programmer what logical construct was being used.

Later, the embedding of hereditary Harrop formulae into linear logic, as described in Section 4.4, suggested another syntax. That syntax, in which ⊗ is represented as '`,`' and `o-` is replaced by `:-`, is intended as a conservative extension of existing Prolog and λProlog syntax. Programs that do not make use of linear features are written just as they would be Prolog.

The remainder of this chapter is in the form of a language guide for Lolli as it now stands. It attempts to cover, in one place, everything you need to know about writing Lolli programs, assuming that you have done some Prolog (or λProlog) programming, and understand the basic ideas underlying Lolli as described in Chapter 4.

9.3 Core Syntax

9.3.1 Terms and Atoms

In its present form Lolli is a higher-order language with essentially first-order unification. That is, variables may stand in any position —or more precisely, quantifiers may range over all object types— but the unification algorithm is essentially that of Prolog, except that it includes the occurs check. λ-terms (as in λProlog), and their unification, are not supported. It is hoped that these will be added in a future version.

As with λProlog, terms and atoms in the language are written in curried form. So, for instance, the Prolog term `f(a,g(b,c,h(d)),e)` is written in Lolli as `(f a (g b c (h d)) e)`. Lists are also constructed as in λProlog, with the ML-style constructors '`::`' and '`nil`'. Thus the Prolog list `[a,[b,c],d|E]` is the Lolli list `(a::(b::c::nil)::d::E)`. Constants in the language are essentially untyped, but can be divided into three classes:

Atomic constants, and variable names can consist of any combination of characters, including spaces, new-lines, or other control codes. If a name contains only alphanumeric characters (plus the character '_') then it can be written directly. If it contains other characters, it can be written either by enclosing the entire name in single quotes, (`''`), or by preceding non-alphanumeric characters with a caret, (`^`). Lolli will always use the latter method when printing names, except for the output of `write_sans` as described below.

Unless explicitly quantified, names beginning with an upper-case letter or an underscore are assumed to be logic variables (that is, existentially quantified at the outermost level of the containing query, or universally quantified at the outermost level of the containing clause). All other names are assumed to be constants. It is important to note that while '_' is a valid variable name in Lolli, it is not anonymous. Lolli does not support anonymous variables. Thus the unification "`(f _ _) = (f b c)`" fails in Lolli.

Certain names (`,`, `;`, `&`, `-o`, `--o`, `:-`, `=>`, `<=`, `-->`, `=`, `=:=`, `=\=`, `=<`, `>=`, `<`, `>`, `is`, `+`, `-`, `*`, `/`, `::`) are used for built-in predicates or logical operators and are expected by the parser to occur in infix position. To use these names in other ways, just put single quotes around them, i.e. `(verb_aux 'is')`.

Strings are any combination of characters surrounded by double quotes. Strings are expected by some of the built-in predicates (particularly when a file name is needed), and are also useful as prompts, since the `write_sans` built-in will print them without the quotes (see below).

Integers are any combination of digits only. They are expected when arithmetic evaluation is done.

9.3.2 Clauses and Goals

The formula language of Lolli is quite rich compared to that of Prolog, and is described in depth in Chapter 4. The following is intended to summarize the concrete syntax used for the various logical operators:

`true` – Corresponds to the zero-ary operator **1**: This goal always succeeds, provided any resources in the bounded context will be used elsewhere in the goal surrounding this one.

`erase` – Corresponds to the zero-ary operator $\top$: This goal always succeeds, and in so doing consumes any resources in the bounded context that are not otherwise (thus far, or in the future) consumed by the surrounding compound goal. In effect it exempts the surrounding goal from the requirement that it use all of the resources in the bounded context.

`{A}` – Corresponds to the formula $(!A)$. Used in goal position, this can succeed only if `A` can be proved without using any clauses from the current bounded (linear) context. In clause

position, this indicates that the clause should be placed in the unbounded context. (This use is discouraged and is a throwback the earlier formulation of the logic underlying Lolli used in [Hodas and Miller, 1991]. Use the intuitionistic implication operator instead.)

A , B – Corresponds to the multiplicative conjunction ($A \otimes B$): Used as a goal, this says to attempt a proof of A and, if it succeeds, to attempt a proof of B using whatever elements of the bounded context were not consumed during the proof of A. That is, it divides the bounded context between the two conjuncts.

This operator can also be used on the left of a linear implication, (or as the outer connective in a clause marked LINEAR in a module), in which case it simply refers to the conjunction of the two subclauses. That is, `((a , b) -o c)` is equivalent to `(a -o (b -o c))`. For proof theoretic reasons —it would compromise the completeness of uniform proofs (see Chapter 4)— it cannot occur on the left of an intuitionistic implication.

A & B – Corresponds to the additive conjunction ($A \& B$): As a goal this is similar to `(A , B)`, but it succeeds only if B can be proved using exactly the same set of resources from the bounded context as the proof of A. That is, it duplicates the bounded context for each conjunct. If the bounded context is empty then the two types of conjunction behave the same.

When this operator occurs as the outer level connective in a clause (or on the left of an implication goal) it acts in a somewhat disjunctive manner, saying that one or the other of the clauses may be selected, but not both. (If the clause pair is in the unbounded context this behavior is mitigated, since the overall pair may be used twice, and each subclause selected in turn). In the head of a clause it says that any of the conjuncts may match as the head. These behaviors are seen in the following program and interaction:

```
MODULE with_clause.

LINEAR (a & b).
(c & d) :- a.
(e & f) :- b.
```

```
?- with_clause --o top.
?- a.
yes
?- c.
yes
?- a,b.
no
?- c,d.
no
?- erase , ((a & b) => top).  % Remove (A & B), replace with !(A & B)
?- a,b,c.
yes
```

```
?- a,d,c,a,a,f.
yes
```

This feature is most useful, I think, in providing multiple heads for a single clause. For that purpose I would recommend formatting of the form:

```
( (head_1 ... ) &
  (head_2 ... ) &
   ...
  (head_n ... ) ) :- body_of_clause.
```

A ; B – Corresponds to the multiplicative disjunction ($A \oplus B$): This goal attempts a proof of A, and if it succeeds the overall goal succeeds. If it fails then B is attempted, and the overall goal succeeds if it does. This is the same behavior as in Prolog.

A -o B – Corresponds to linear implication ($A \multimap B$): In goal position this means add the clause A to the bounded context (unless it is of the form '{A1}' in which case it is added to the unbounded context as described above) and attempt to prove the goal B. While clause ordering is not an issue in the pure logic, search in Lolli (as in Prolog) occurs from the beginning of a program and continues downwards. Implications add their assumptions to the beginning of a program.

A :- B – Corresponds to linear implication ($B \multimap A$): Used in clause position, A is taken to be the head of the clause, and the goal B is its body. Unlike Prolog, the head A need not be an atom, though it will most often be. Rather, it can be any valid clause. To understand this, look at the definition of backchaining in the simple Prolog implementation of the logic given in Chapter 7.

In actuality, the operators ':-' and '-o' are fully interchangeable, with (A -o B) equivalent to (B :- A).

A => B – Corresponds to the intuitionistic implication ($A \Rightarrow B$): The same as (A -o B), but adds the clause A to the unbounded context. As mentioned above, the same effect can be achieved by ({A} -o B), though that style is discouraged.

A <= B – Corresponds to the intuitionistic implication ($B \Rightarrow A$): Similar to (A :- B), but, when used as a clause (with head A and body B) in backchaining, the body must be provable without using any resources in the current bounded context. The same effect can be achieved by (A :- {B}), though that style is discouraged.

As above, the operators '<=' and '=>' are fully interchangeable, with (A => B) equivalent to (B <= A).

forall x \ A – Corresponds to the universal quantification $\forall x.A$: As a goal, it directs the system to substitute a new constant for x in A and then attempt to prove the modified version of A. The bound name may begin with either upper or lower case, it makes no difference. During the proof of A, the system must insure that any logic variables that

are currently free in A, or the current program, are not bound to terms containing the new constant. This check insures that the new constant cannot be drawn outside of the context in which it is created.

exists x \ A – Corresponds to the existential quantification $\exists x.A$: This can be thought of as turning a goal into a yes/no question. That is the goal will succeed if there is a substitution for the variable 'x', but it won't return the substitution. If this is a part of a compound goal, the effect is to bind 'x' locally, distinguishing it from other 'x's which may occur in the overall goal. As with forall, bound names may begin with upper or lower case. It should be noted that while free variables in goals are treated, from a logical standpoint, as though they were existentially quantified, the system distinguishes between those that are implicitly quantified (for which substitutions are printed) and those that are explicitly quantified (for which they are not). For example:

```
?- f x -o f A.
A_1 <- x.
?- exists A \ (f x -o f A).
yes.
```

Test -> Succeeds | Fails – The guard expression is the one extra-logical operator included in the current release of Lolli. It first attempts a proof of Test. If that succeeds, it then attempts to prove Succeeds, and the overall goal succeeds only if Succeeds does. If Test fails, then Fails is attempted, and the overall goal succeeds only if Fails does. In no case is Test reattempted.

This extra-logical can be used to implement many of the other such operators common to logic programming. For instance, negation-as-failure can be implemented with the clause:

```
not G :-  G -> fail | true.
```

The operator once, which succeeds if its operand does, but can succeed at most once, is defined as:

```
once G :-  G -> true | fail.
```

Note that the resource consumption of a guard expression (assuming the overall goal succeeds) is equivalent to either (Test , Succeeds), or to just Fails, depending on which case holds. This seems the most sensible choice, since Test will usually be some simple predicate which consumes no resources.

The material above does not fully explain all of the ways in which the logical operators may be combined. The possible formulae of Lolli can be summarized by the formulation of three classes of formulae, R, D, and G, (where G comprises the class of all valid goal formulae) as follows:

$$R \;:=\; \texttt{true} \mid A \mid R_1 \;\&\; R2 \mid R \;\texttt{:-}\; G \mid G \;\texttt{-o}\; R \mid R \;\texttt{<=}\; G \mid G \;\texttt{=>}\; R \mid \texttt{forall}\; x \setminus R$$

D := `erase` | R | $\{R\}$ | $D1$ `,` $D2$

G := `true` | `erase` | A | $\{G\}$ | G_1 `&` G_2 | G_1 `,` G_2 | D `-o` G | G `:-` D
| G `<=` R | R `=>` G | `forall` x `\` G | `exists` x `\` G | G_1 `;` G_2 | (G_1 `->` G_2 `|` G_3)

Note that this does not correspond precisely to the formulation of clauses and goals given in Chapter 4. It is a bit more general, and contains some formulae inherited from an earlier formulation of the logic as described in [Hodas and Miller, 1991].

The associativity and precedence of the operators, in ascending order of precedence is:

right – **`forall`**, **`exists`**

left – **`<=`**, **`o-`**, **`-->`** (for use in DCGs.)

right – **`;`**

right – **`&`**

right – **`,`**

right – **`=>`**, **`-o`**

right – **`--o`**

right – **`->`**

left – **`|`**

left – **`=`**, **`=:=`**, **`=\=`**, **`=<`**, **`>=`**, **`<`**, **`>`**, **`is`**

left – **`+`**, **`-`**, **`*`**, **`/`**

right – **`::`**

9.4 Modules

While it is possible to use Lolli exclusively interactively, using implications to load clauses into the program contexts, this quickly grows tedious. Some way of reading programs from text files is desired. To this end, Lolli has adopted an extended (and modified) variant of λProlog's module system.

The basic form of a module is a name declaration followed by an arbitrary number of clauses, with '`.`' used to mark the end of each clause. I.e.:

```
MODULE modulename.

clause_1.
clause_2.
...
```

This module must be stored in a file named `modulename.ll`.

As with Prolog, and λProlog, (and subject to the constraints mentioned above), variables (names beginning with an upper case character or underscore) are assumed to be universally quantified at the boundary of the clause in which they occur.

Because it is assumed that most of the clauses in a program are intended to be usable as many or as few times as needed, clauses not otherwise marked are loaded into the unbounded context (as though they had been asserted with the '=>' operator). If a clause is intended to be put in the bounded context it is marked with the keyword LINEAR, i.e.:

```
MODULE modulename.

clause 1.
LINEAR clause 2.
...
```

Two mechanisms are provided for controlling the binding (and availability) of names that occur in a module. First, in order to support notions of abstraction and hiding, the LOCAL declaration, which takes a space delimited list of names (which can be constant, function, or predicate names), declares that those names are local to the module, and cannot be seen outside. Thus in the following module, the predicate pr1 cannot be called from outside this module:

```
MODULE modulename.

LOCAL pr1.

clause 1.
...
pr1 :- body1.
pr1 :- body2.
...
```

A module may also be parameterized by a series of names, which are placed after the modulename in the module declaration. This provides a way of specifying behavior of clauses in the module without adding extra parameters to the individual predicates. Thus a module defining a sorting predicate could be given by:

```
MODULE sort ordering.

LOCAL collect unpack hyp.

collect nil.
collect (X::nil) :- hyp X.
collect (X::Y::L) :- collect (Y::L), hyp X, ordering X Y.

unpack nil G :- G.
unpack (X::L) G :- hyp X -o unpack L G.

sort L K :- unpack L (collect K).
```

Modules are loaded using the '--o' operator, as described below. So, if we wished to use this predicate to sort some list in descending order, the query would be:

```
?- (sort '>=') --o (sort (1::3::5::2::4::6::0::nil) A).
A_1 <- 6 :: 5 :: 4 :: 3 :: 2 :: 1 :: 0 :: nil.
```

Note that the overloading of the `sort` identifier as module name and predicate name in this example is not a problem. As explained in Chapter 4, the entire module syntax is actually syntactic sugar for a combination of quantifiers and implications.

9.5 Built-in (Evaluable) Predicates

An important goal of this implementation was to provide enough built-in predicates to make it possible to write interesting Lolli programs. The built-in predicates can be classified in four groups: input/output, arithmetic, control, and miscellaneous.

Note that, due to the nature of the implementation, it is illegal to specify clauses for predicates with the same name as a built-in predicate, even if the predicate you are attempting to define is of a different arity.

9.5.1 Input/Output

`write Term` – Writes the current instantiation of `Term` to current output. Strings are written with surrounding double quotes, atomic constants are written directly, with non-alphanumeric characters preceded by a caret. Complex terms are written with as few parentheses as possible, using the same associativity and precedence as the input parser.

`write_sans Term` – If `Term` is instantiated to a string, the text of the string is written to the current output, without surrounding quotes. If it is instantiated to a constant, the constant is written out without quotes or carets for non-alphanumeric characters. Finally, if `Term` is any other structure, (including an uninstantiated variable, or a complex term with string sub-terms) this predicate behaves the same as **`write`**/1.

write_clause Term – This predicate behaves the same as **write**/1 except that the term is treated as though it occurs as a program clause, i.e.. it has parity -1. So, for instance,

```
?- write ((a -o b) -o c), nl.
(b :- a) -o c
yes.
?- write_clause ((a -o b) -o c), nl.
c :- (a -o b)
solved
```

write_raw Term – This predicate is intended mostly for debugging Lolli's parser. It prints out the given term in prefix form with all its parentheses. This enables you to check that a term is parsing the way you think it should.

nl – Writes a newline character to the current output.

read Term – Reads a term from the current input and unifies it with Term. The input is terminated either by a period or end-of-file. If end-of-file is (or has been) reached before any input is read, then the atomic constant end_of_file is returned by the input parser.

telling Filename Goal – Sets the current output to the named file for the duration of the proof of the goal given by Goal. Filename must be instantiated to a string constant. Success or failure of the subgoal will cause the output to be reset to standard input. Failing back into the goal will not cause the output to be re-opened. Further, the system only tracks one output stream. If a nested subgoal causes output to redirect to a second file, its completion will cause output to return to standard output, not to the first file. This behavior may change in the future to allow a stack of output streams.

seeing Filename Goal – This predicate behaves the same as **telling**/2 but controls the input stream.

9.5.2 Arithmetic

The arithmetic predicates are essentially the same as for Prolog:

Term1 is Term2 – Term2 —which must be instantiated to an arithmetic expression built of integers and the operators +, -, *, and / (integer division)— is evaluated and the result is unified with Term1. Note that, in the current release, no operator precedence is assumed for the arithmetic operators. Use parentheses as necessary.

Term1 =:= Term2 – Term1 and Term2 are evaluated, and checked for equality,

Term1 =\= Term2 – inequality,

Term1 >= Term2 – greater-than-or-equals,

Term1 =< Term2 – less-than-or-equals,

Term1 > Term2 – greater-than, or

Term1 < Term2 – less-than, respectively.

Note: Be careful when entering '>=' and '=<', since '=>' and '<=' are defined as intuitionistic implication.

9.5.3 Control

fail – This goal always fails.

top – Start a new read-prove-print loop, with the current proof context as a base. Bounded formulae in the new base context must be used in any query that is to succeed. However, each successive query will be started with those formulae restored to the context. I.e.,

```
?- a -o top.
?- true.
no.
?- a.
yes.
?- a -o (a, a).
yes.
```

pop – This causes the current goal and the current read-prove-print loop to be exited, and control returned to the loop that was executing at the time of the most recent call to **top**/1. Any goals that are pending in that loop are also aborted. For instance, if we continue the session in the last example,

```
?- a -o top, a.
?- a.
no.
?- a, a.
yes.
?- pop, a.
Returning to previous top level...
?- a.
yes.
```

If end-of-file (^D on most systems) is issued at the Lolli prompt, it is taken as a short-cut for **pop**/1. Attempts to **pop**/1 out of the outermost (top-level) read-prove-print loop will cause the following message to be printed:

```
?- pop.
You are now at the top level. Use 'bye' to leave Lolli.
```

popall – Similar to **pop**/1, but returns directly to the outermost read-prove-print loop.

abort – Immediately abort the current goal and return to the current read-prove-print loop.

```
?- abort, a.
aborted...
```

bye – Immediately abort the current goal and exit Lolli. Alternate forms, `exit` and `quit` are also accepted.

Term --o Goal – Take the head of `Term` to be a module name, and its arguments to be actual parameters. Load the named module, appropriately parameterized, into the current context, and attempt to prove `Goal`.

load Term – This is just syntactic sugar for `(Term --o top)`.

9.5.4 Miscellaneous

Term1 = Term2 – Attempts to unify `Term1` with `Term2`.

timing Goal Term – - Attempts to prove the goal `Goal`. If this succeeds, unify `Term` with the number of microseconds used in the proof. This time includes time spent garbage collecting by the underlying ML runtime system. If subsequent failure causes `Goal` to be reattempted and resucceed, then the timer will also include all time spent on all of the intervening computations.

cd Dirname – Causes the current directory to be changed to `Dirname`, which must be instantiated to a string constant.

system Command Result – Passes the string `Command` to the operating system as a command. The result code returned by the operating system is then unified with `Result`.

explode String Term – Unifies `Term` with a list built by turning each character of `String` into an atomic constant.

explode_words String Term – Unifies `Term` with a list built by turning each word (i.e., white-space delimited chunk of characters) in `String` into an atomic constant.

var Term – This goal succeeds if and only if `Term` is an uninstantiated logic variable.

nonvar Term – This goal succeeds if and only if `Term` is anything but an uninstantiated logic variable.

generalize Term1 Term2 – This predicate scans `Term1` to find any unbound logic variables, and then unifies `Term2` with a term built from `Term1`, but with all those unbound variables explicitly universally quantified. For example,

```
?- generalize (f A b (c D)) E.
E_1 <- forall A_3 \ (forall D_2 \ f A_3 b (c D_2)).
```

Note that the implementation of generalize is potentially unsound, as it does not check for variable capture. The circumstances that would force such a capture are, however, unlikely to occur in practice —in fact it's not clear that they can occur at all.

Chapter 10

Related Work

There are many ways in which linear logic can be fruitfully exploited to address aspects of logic programming. Girard used a non-commutative linear logic to model the difference between the classical, "external" logic of Horn clauses and the "internal" logic of Prolog that arises from the use of depth-first search [Girard, 1987*b*]. Cerrito applied classical linear logic to the problem of formalizing finite failure for certain kinds of Horn clause programs where negations are permitted in the bodies of clauses [Cerrito, 1990]. Gabbay has used the ideas that arise from linear logic to isolate different computational aspects of proof search in a variety of systems in an attempt to identify complete yet efficient search techniques [Gabbay, 1991]. In particular, he was interested in those languages in which search may be restricted to using formulas linearly. This work is conceptually related to, though somewhat broader than, the recent work in linearizing intuitionistic implication by Dyckhoff [Dyckhoff, 1992] and Lincoln, Scedrov, and Shankar [Lincoln *et al.*, 1991].

In this chapter I would like to focus, however, on three proposals more directly related to this one.

10.1 Andreoli and Pareschi's Linear Objects

Andreoli and Pareschi have, in a long series of papers, presented one of the most successful proposals for linear logic programming to date [Andreoli and Pareschi, 1989; Andreoli and Pareschi, 1990*a*; Andreoli and Pareschi, 1990*b*; Andreoli and Pareschi, 1990*c*; Andreoli, 1990; Andreoli and Pareschi, 1991*a*; Andreoli and Pareschi, 1991*b*; Andreoli, 1992].

To be precise, these papers actually deal with two different systems. The first, simpler language is called Linear Objects, or *LO*. The second language is a considerable enrichment —in fact they show that it is complete for all of classical linear logic— and is called LinLog. This discussion will focus primarily on *LO*, with some discussion of LinLog at the end.

Their proposal shares a motivating example with Lolli: the implementation of mutable objects with inheritance and message passing in a logical setting. However, that is where the

similarity between our systems ends. In almost every design area, the systems are virtually orthogonal in the the approach taken.

The most important distinction, which permeates the proposal, is that *LO* and LinLog are based on the multiple conclusion version of linear logic. This choice was made because their goal was to refine the techniques used for representing objects in concurrent logic programming languages. Andreoli and Pareschi observed that object-oriented programming can be seen as encompassing two quite disparate types of dynamism:

Short-Term Dynamism This is the dynamism of mutable state. In responding to messages, the internal state of an object may change over time.

Long-Term Dynamism This is the dynamism of program design and specification. Over time, the description of a class may change. It may take on new instance variables and behaviors. Existing behaviors should extend automatically, however. They should not need to be redefined if the part of the object that they manipulate is not redesigned.

They noted that concurrent logic programming provides a good solution for dealing with short-term dynamism. In concurrent logic programming many goals may be treated simultaneously, and the reduction of any one goal may be an ongoing process, not intended to terminate, but rather to evolve. Clauses (which can also be thought of as rewrite rules) are generally of the simple form:

```
old_state :- new_state.
```

Suppose, for instance, that we wish to define a class of graphical cursors, which have internal state variables X and Y (for the current position) and *Mode* (*on* or *off*, depending on whether the cursor should leave a trail behind when it moves).[1] Then the state of a cursor would be represented by the predicate $curs(In\_stream, X, Y, Mode, Out\_stream)$, where *In_stream* and *Out_stream* are communications channels to other objects (or the user). The program to mange these objects could then be given by:

```
delay curs(In,_,_,_,_) if var(In).

curs([moveto(X1,Y1)|In],_,_,off,Out) :- curs(In,X1,Y1,off,Out).
curs([moveto(X1,Y1)|In],X,Y,on,[draw(X,Y,X1,Y1)|Out]) :-
       curs(In,X1,Y1,off,Out).
curs([turn_on|In],X,Y,_,Out) :- curs(In,X,Y,on,Out).
curs([turn_off|In],X,Y,_,Out) :- curs(In,X,Y,off,Out).

curs([],_,_,_,[]).
```

The first clause specifies that whenever the input message stream of a `curs` goal is unbound, that goal should simply freeze until something binds it to a message list. The last

[1] This example is taken directly from Andreoli's thesis [Andreoli, 1990].

clause says that if the input message stream is the empty stream, that should be taken as a signal for this particular cursor object to terminate. The remainder of the rules specify the methods for the class, telling it how to respond to various messages.

Andreoli and Pareschi point out that while this does a good job with short-term dynamism of cursors, it does a bad job with long-term dynamism. If a new instance variable is added to cursors, say an ink color, then all the methods must be redefined for an arity-six *curs* predicate, or unifications will fail. They show that in a similar way, the Ψ-terms of Aït-Kaci's Login language[Aït-Kaci, 1986], which extends Prolog's unification algorithm to order-sorted algebras and terms with record-like structure, can handle long-term dynamism well, but fail on short-term dynamism.

The solution they propose is a fragment of multiple conclusion linear logic over formulae defined as follows (they specify only the propositional fragment):

Definition 10.1 The set of *view formulae* is the smallest set satisfying the following conditions:

- All atomic formulae A are view formulae.
- If V_1 and V_2 are view formulae, then $V_1 ⅋ V_2$ is a view formula.

Definition 10.2 The set of *goal formulae* is the smallest set satisfying the condition:

- $\top$ and all atomic formulae A are goal formulae.
- If G_1 and G_2 are goal formulae, then $G_1 \,\&\, G2$, and $G_1 ⅋ G_2$ are goal formulae.

Definition 10.3 The set of *methods*, or *definite formulae*, is the smallest set satisfying the following condition:

- If G is a goal formula (as defined above) and V is a view formula, then $G \multimap V$ and $G \Rightarrow V$ are methods.

The idea is that each instance variable of an object is stored in a separate predicate, and the '⅋' of all these predicates together forms the state of the object. The behavior of *LO* is then quite simple. A method can be applied if the formulas ⅋'ed together in its head form a submultiset of one of the current goals. If they do, replace that submultiset with the goal on the right. Thus, methods need not be redefined when additional instance variables are added to a class. A view of the original object is still a view of the enriched object.

The authors give several examples to show that this rich notion of context on the right (as opposed to the left as it occurs in $\mathcal{L}$) together with the rules for '⅋' and '&' provide a very natural solution to representing both forms of dynamism. They go on to give examples of how this type of context can be used to implement many of the kinds of applications that implications in goals are used for in λProlog. For instance, the following version of efficient reverse (which was demonstrated in λProlog in Chapter 3) is instructive (here `@` is used as concrete syntax for '⅋', and `#t` stands for '$\top$'):

```
reverse(L,K) <= rev(L,[]) @ result(K).

rev([X|L],M) <- rev(L,[X|M]).

rev([],M) @ result(M) <= #t.
```

The system LinLog is based on a much richer fragment of multiple conclusion linear logic. In fact, they show that the fragment is complete, in that there is a reduction from the full system to it. Nevertheless, through a lengthy development, in many ways analogous to our development of $\mathcal{L}$, they are, in the end, able to produce a simple goal directed proof system (which produces what they refer to as *focusing proofs*) which is complete for the fragment. This is an admirable technical accomplishment. Unfortunately, while it allows them to encode many other systems into this one, it is never made totally clear what this enriched fragment offers over *LO* when it comes to their main intended application of object-oriented programming.

10.2 Harland and Pym's Uniform Proof-Theoretic Linear Logic Programming

At roughly the same time that the first paper on Lolli was released, Harland and Pym released a pair of papers (one an extended version of the other) in which they proposed a fragment of linear logic that could be used as the basis of a logic programming language [Harland and Pym, 1991; Harland and Pym, 1990]. From the standpoint of formal underpinnings, this is the work with the closest relationship to Lolli.

Harland and Pym approached the problem of designing a programming language purely from the theoretical side. The goal was simply to find a significant fragment of linear logic for which uniform proofs are complete. There is no discussion of applications, and only limited discussion of implementation.

The principal difference between the system proposed by Harland and Pym, and that arrived at for Lolli, is in the fragment of the logic considered. The first, and most important, distinction is that they consider a multiple conclusion system, i.e. one in which the succedent of a sequent is a multiset of goal formulae. Thus, in one system, they subsume features of both Lolli, and Linear Objects. Neither is fully contained within their system, however. [2]

Sequents in their system are constructed out of multisets of goal and definite formulae defined as follows (with one initial definition needed for the two main definitions):

[2] While the core of the papers assumes a multiple conclusion setting, the discussion of implementation issues reverts to a single conclusion setting. The authors do briefly discuss the complications that arise from the need to select a goal in the multiple conclusion setting, and suggest that "This question is further complicated due to the fact that clauses may be deleted from the program, and that programs and goals may be split into smaller programs and goals during the computation proccess. Given these complications it might be best to reduce all formulae in the goal to atoms before proceeding with any resolution step." This is a first step towards Andreoli's focusing proofs, but it is not worked out any further.

Definition 10.4 The set of *literal formulae* is the smallest set satisfying the following conditions:

- $\top$, $\bot$, $\mathbf{1}$, $\mathbf{0}$, and all atomic formulae A are literal formulae.

Definition 10.5 The set of *goal formulae* is the smallest set satisfying the condition:

- if L is a literal formula, then L and $L^{\bot}$ are goal formulae.
- If G_1 and G_2 are goal formulae, then $G_1 \otimes G_2$, $G_1 \,\&\, G2$, $G_1 \parr G_2$, and $G_1 \oplus G_2$ are goal formulae.
- If G is a goal formula, then $\exists x.G$, and $\forall x.G$ are goal formulae.
- If G is a goal formula and D is a definite clause (as defined below), then $D \multimap G$ is a goal formula.

Definition 10.6 The set of *definite formulae* or *program clauses* is the smallest set satisfying the following conditions:

- All literal formulae are definite formulae.
- If D_1 and D_2 are definite formulae, then $D_1 \,\&\, D_2$ and $D_1 \otimes D_2$ are definite formulae.
- If D is a definite formula, then $\forall x.D$ and $!D$ are definite formulae.
- If G is a goal formula (as defined above) and L is a literal formula, then $G \multimap L$ is a definite formula.

These sets contrast with those for $\mathcal{L}$ is several ways, the most important of which are:

- $D_1 \otimes D_2$ is allowed as a definite formula, where it not allowed in $\mathcal{L}$.
- The goal $G_1 \parr G_2$ is allowed, though only in the multiple conclusion setting, which has no analogue in $\mathcal{L}$.
- Logical negation (as distinct from negation-by-failure) is allowed in goals.
- The goal $!G$ is not allowed.
- Rules must be of the form $G \multimap L$, which, when '$\otimes$' is allowed in definite formulae, is strictly less expressive than $G \multimap D$. Note however, that this does not make them less expressive than $\mathcal{L}$ rules (where '$\otimes$' is not allowed). That is because restricting $\mathcal{L}$ to atomic-headed rules does not change the expressivity of $\mathcal{L}$ programs, and such a restriction makes $\mathcal{L}$ definite formulae a strict subset of these definite formulae (except that, since $!G$ is not a goal formula in this system, $G \Rightarrow L$ is also not a definite formula).

Harland and Pym first show that proofs in this system conform to a notion of uniform proof completeness that has been augmented to handle the multiple conclusion setting, and relaxed to allow certain restricted non-uniform occurences of the $\otimes_L$ and $!_L$ rules. They then develop a "resolution proof system" for the system that is closely related to $\mathcal{L}'$. The left rules are replaced by two resolution rules (one for backchaining on linear rules, one for backchaining on rules with an outermost occurence of '!') and the right rules are modified to implicitly contract any !'ed formulas in the antecedent. Finally, in the section on implementation, they discuss problems similar to those that lead us to introduce the *IO* system. Unfortunately, some of these problems are more severe in their setting, and the solution is a bit more involved. Also, no proof of its correctness is given.

While the definite formulae in this system form a larger class than the definite formulae of $\mathcal{L}$, it is not clear whether this leads to any useful features at the language level. This is why the absence of motivating examples is such a serious gap: determining whether these differences, which are motivated by proof-theoretic considerations, yield any real benefits is left entirely to the reader.

At the top level, allowing a '$\otimes$' in definite formulas is of no particular benefit, since at that level it is logically equivalent to the comma. In the next chapter we see that '$\otimes$' may be helpful within definite formulae as it occurs in the unfolded versions of programs written in $\mathcal{L}$. It is not clear, however, whether this small benefit is worth the other sacrifices that must be made to allow this usage.

The benefits of negated goals is also unclear, since negation is not allowed in definite formulae. Therefore, its only effect is to move a formula to the left across the sequent arrow. It seems mostly to provide an alternative to linear implication in goals.

The exclusion of goals of the form $!G$ appears to be a real loss. In Chapters 5 and 6 several examples were given that showed the utility of such goals: to check whether a fact in a database was committed; to ensure that one phase of execution was immune from interference by another (i.e. the `demo/2` predicate); and to handle island constraints during parsing. If we must give them up, then what we gain in their stead should be of obvious use.

10.3 Bollen's Relevant Logic Programming

As discussed in Chapter 8, Bollen has proposed a language —which he calls CLOGPROG, for Conditional LOGic PROGramming— which extends Prolog by adding relevant implication in goals [Bollen, 1991]. Bollen's paper shares a great deal philosophically, and technically, with this dissertation. And yet (or perhaps, therefore), it is the proposal about which we will have the least to say.

Bollen's starting off point is Gabbay's N-Prolog, which allows intuitionistic implications in goals; but he makes a convincing argument that for most applications relevant implication is more desirable. He then presents the intuitive meaning of his extension, by way of a simple example application (partly reproduced in Chapter 8): an expert sysem for reasoning about environmental damage. Finally, he presents the formal specification of the system, first in

terms of an extension of the SLD algorithm, then in terms of a Gentzen sequent calculus.

In his discussion of the Gentzen formulation of his system he presents a proviso that is essentially the definition of uniform-proof completeness. He then presents arguments for various restrictions of the formula language necessary to make this proviso go through. Unfortunately, his logical argument is somewhat flawed, insisting on full invertibility, rather than permutability. He also requires that, while the system is restricted to clauses with atomic heads, it should have the same expressive power as though it allowed fully nested implications. Therefore, his system is somewhat simpler even than it needs to be. In particular, while some nesting of quantifiers is allowed, it is not as rich as the nesting allowed in $\mathcal{L}$. Explicit quantifiers can occur around clauses to be assumed, but they cannot occur in goal position. Similarly, while Bollen concludes, as we did in Chapter 4, that the ordinary conjunction should map to '&' in antecedents and to'$\otimes$' in goals, these are the only uses that he allows.[3]

The most important point to be made, though, is that Bollen's system forms a direct sublanguage of Lolli. Bollen is interested in implications in goals only from the standpoint of hypothetical reasoning, and makes no mention of their other applications (i.e. modules). In this context his restiction to just one sort of implication is sensible, and leads to a logic that is somewhat simpler than $\mathcal{L}$ (or $\mathcal{O}$). Even without the enriched logic of $\mathcal{O}$, the relevant goal implication $A \stackrel{R}{-\!\circ} B$ can be simulated by the compound implication $A \multimap (A \Rightarrow B)$. His specification of SLC proof search (his variant of SLD resolution for CLOGPROG) can also be seen to specify a fragment of the *IO* semantics.

Perhaps the most interesting suggestion made by Bollen is in his discussion of the actual implementation of CLOGPROG. In that system, the ordinary top-down search of the database used in Prolog has been modified so that rules that have not yet been used will be selected before those that have already been used at least once. Since every successful search must use all the conditional assumptions at least once, this is an appealing way to reduce the number of failed searches. On the other hand, it is not clear whether this change will adversely affect the programmer's intuitions about program execution.

[3] In actuality, in Bollen's restricted system this may not be much of a limitation, since in the presence of contraction on the left (as is allowed in relevant logic) instances of $\&_R$ can be replaced with $\otimes_R$. However, as other context management schemes are considered, the distinct operators take on additional meaning.

Chapter 11

Future Work

In this disseration I have described a fragment of linear logic, and asserted that it can be viewed as a logic programming language. This view was supported by showing several things, among them:

- Uniform proofs are complete for the fragment, so the fragment can be seen to support goal-directed proof search.
- Practical and useful programs can be written in the language which cannot be written in Horn clauses or in hereditary Harrop formulas.
- While a direct, naive implementation of the logic will produce a system with an unreasonable amount of non-deterministic search, this non-determinism can be mostly eliminated by a careful analysis that shows how to delay any decisions of how to split up the set of bounded resources.
- This lazy operational semantics can be implemented straightforwardly as an interpreter for the language in both Prolog and ML.

There is much yet to be explored about these logics and the languages built on them, however.

11.1 Investigating Programming Styles and Paradigms

To date I have written two to three dozen small (up to 150 lines) Lolli programs of various types. This has been sufficient to establish certain obvious strengths in the language, and to influence the current choice of concrete syntax.

The underlying theory of the system has also been significantly influenced by the writing of examples. For instance, it was in experimenting with object-oriented programming examples that the seriousness of the non-determinism left in the *IO* semantics was established; this then motivated the development of the $IO^{\top}$ refinement. The design of the system *O* was similarly

example driven. It is important that more and larger programs be written to further clarify the desirable features for such a system.

Certainly, it seems from Chapter 5 that continuation passing is an important paradigm in this setting. Therefore, one area that I plan to focus on is to see if it is possible to find an attractive sugared syntax for continuation passing. This would likely take the form of something analogous to DCG syntax, but with an implicit continuation parameter. It may in fact be appropriate for there to be more than one syntax, depending on the application. For example, the implementation of mutable data structures seems to be a common use of the continuation passing style. It is possible to imagine a syntax like:

```
pred ... :- ... store := New; G.
```

to be used in place of:

```
pred ... G :- ... store Old, store New -o G.
```

Hudak has recently proposed implementing mutable data types in the purely functional language Haskell using continuations in a style very similar to that used in Chapter 5. He has proposed a syntax similar to this one.

11.2 Clarifying the Logical Status of $\mathcal{O}$ and Implementing It

The refinementof $\mathcal{L}$ presented in Chapter 8 is a relatively recent development, and there are several questions about its logical status that remain open.

11.2.1 Cut Rules and Their Admissibility

The system $\mathcal{O}$ was presented without cut rules, and shown to be complete relative to the cut-free fragment of $\mathcal{L}$. However, it would would be desirable, for stylistic completeness, to present cut rules for the system and show that they are admissible. By analogy to the development of $\mathcal{L}$, it seems that there should be four cut rules; one for cutting into each context. The following rules seem reasonable:

$$\frac{\Gamma_1;\Upsilon_1;\Psi_1;\Delta_1 \longrightarrow B \quad \Gamma_2;\Upsilon_2;\Psi_2;\Delta_2, B \longrightarrow C}{\Gamma_1,\Gamma_2;\Upsilon_1,\Upsilon_2;\Psi_1,\Psi_2;\Delta_1,\Delta_2 \longrightarrow C}\ cut_L \qquad \frac{\Gamma_1;\varnothing;\varnothing;\varnothing \longrightarrow B \quad \Gamma_2, B;\Upsilon;\Psi;\Delta \longrightarrow C}{\Gamma_1,\Gamma_2;\Upsilon;\Psi;\Delta \longrightarrow C}\ cut_I$$

$$\frac{\Gamma_1;\varnothing;\Psi_1;\varnothing \longrightarrow B \quad \Gamma_2;\Upsilon;\Psi_2, B;\Delta \longrightarrow C}{\Gamma_1,\Gamma_2;\Upsilon;\Psi_1,\Psi_2;\Delta \longrightarrow C}\ cut_A \qquad \frac{\Gamma_1;\Upsilon_1;\varnothing;\varnothing \longrightarrow B \quad \Gamma_2;\Upsilon_2, B;\Psi;\Delta \longrightarrow C}{\Gamma_1,\Gamma_2;\Upsilon_1,\Upsilon_2;\Psi;\Delta \longrightarrow C}\ cut_R$$

While indirect admissibility arguments are technically sufficient, as stated in Chapter 4 a direct proof is more desirable; and thus far such a proof has eluded me. Nevertheless, I expect to be able to find one soon.[1]

[1] This is particularly likely given a recent reformulation of the cut elimination algorithm for $\mathcal{L}$ to the form given

11.2.2 The Relationship of $\mathcal{O}$ to Other Logics

We know from Chapter 8 that $\mathcal{O}$ is complete relative to $\mathcal{L}$ and sound relative to an augmented variant. Since the intent of the design of $\mathcal{O}$ was to augment Lolli with certain new features, this is, in some sense, sufficient. Nevertheless, since these features are intended to capture the behavior of other logical systems (relevant and affine logic), it would be good to have a formal statement (and proof) of the relationship between $\mathcal{O}$ and those sysems.

11.2.3 New Modals

Again, by analogy to $\mathcal{L}$ (and to the cut rules proposed above) it would seem that a balanced presentation of $\mathcal{O}$ would include rules for two new modals, '$^!_A$' and '$^!_R$', specified as:

$$\frac{\Gamma;\Upsilon;\varnothing;\varnothing \longrightarrow C}{\Gamma;\Upsilon;\varnothing;\varnothing \longrightarrow {}^!_R\, C}\ {}^!_R \qquad \frac{\Gamma;\varnothing;\Psi;\varnothing \longrightarrow C}{\Gamma;\varnothing;\Psi;\varnothing \longrightarrow {}^!_A\, C}\ {}^!_A$$

While it is not clear what application these modals would have at the language level, they seem reasonable to include. It is important to note, however, that they have no apparent conversion to $\mathcal{L}$-formulas, so they would block the soundness proof from going through.

11.2.4 The Implementation and Concrete Syntax of $\mathcal{O}$

Because it is only a recent proposal, the logic of $\mathcal{O}$ has not yet been added to the Lolli implementation. I would like to see all these logical issues resolved before implementing $\mathcal{O}$, to make sure we are implementing the correct system. Also, it will be necessary to adjust the $IO^\top$ semantics to account for the new behaviors. Finally, there is the choice of concrete syntax for the new implication operators. In Chapter 8, '->>' and '--@' were used for relevant and affine implication, respectively. Unfortunately, these are not very mnemonic. It would be good if a more suggestive notation could be found.

11.3 L_λ Unification

Most of this dissertation has concentrated on the first-order variant of the language. While many useful programs can be written in such a system, Nadathur, Miller and others have demonstrated the advantages of working in a higher-order setting [Nadathur and Miller, 1990; Felty and Miller, 1988; Hannan and Miller, 1988].

It is simple enough to modify a system, as has been done with the actual implementation of Lolli, to support higher-order logic in the sense that terms may include predicate as well as

in Appendix *D*. Previously, the first stage of that algorithm involved a global manipulation of the proof tree to remove instances of *cut*!. That manipulation did not extend to $\mathcal{O}$-proofs. Now that a permutation argument has been found for $\mathcal{L}$ I expect it to extend straightforwardly to $\mathcal{O}$.

function constants, and quantifiers may range over all types. However, even more functionality is gained by enriching the term structure of the language to include λ-terms, as in λProlog. In fact, Miller has argued that once a logic programming language supports explicit nested quantification, as in hereditary Harrop formulas, such an extension to the term language is needed to take full advantage of the extension to the logical language. For instance, if Γ includes clauses for the append program, then the sequent $\Gamma \longrightarrow \exists y.\forall x.append([1 :: 2 :: nil], x, y)$ has no solution in firt-order terms. But if the term structure has been extended to include λ-terms, then the related sequent $\Gamma \longrightarrow \exists F.\forall x.append([1 :: 2 :: nil], x, F(x))$ can be proved by assigning $\lambda y.([1 :: 2 :: y])$ to F.

Unfortunately, as discussed earlier, supporting this enriched term structure is problematic from the standpoint of building practical implementations. Higher-order unification over λ-terms is undecidable, and even when two terms unify, they do not necessarily have a most general unifier. Another complication is that such a system requires a strictly typed term language.

Recently, however, Miller has identified the system he calls L_λ which has neither of these problems [Miller, 1991*a*; Miller, 1991*b*]. Unification in this system, which places strict limitations on the forms that λ-terms may take, is decidable and has most-general unifiers. I plan to extend the ML implementation of the interpreter to support L_λ unification.

11.4 Compilation and Deriving an Abstract Machine

One of the most important factors contributing to Prolog's acceptance as a usable langauge was David H. D. Warren's development, in the late 1970's and early 1980's, of techniques for compiling what had previously been a purely interpreted language [Warren, 1977; Warren, 1983]. This work was particularly noteworthy in that it took the form of the description of an *abstract machine*, an architecture specially designed to support the features of Prolog. The architecture, which has since become known as the Warren Abstract Machine, was designed so that it is clear both how to compile Prolog to it, and also how to implement the machine itself as either an interpreter on, or a compiler to, existing general purpose architectures.

Since that time it has become accepted that the description of a logic programming language should include a description of some extended WAM to which the language can be compiled. For example, Lamma, Mello, and Natali compared various proposals for "logical" module systems by describing an extension to the WAM to which they could all be compiled [Lamma *et al.*,]. Similarly, David S. Warren has proposed the XWAM as an architecture for a Prolog-like language which supports forward chaining using extension tables [Warren, 1990]. It is my hope to describe a WAM-like abstract machine for Lolli.

In general such architectures (including the WAM itself) have been arrived at through ad-hoc techniques, with no attempt to show that the architectures were correct, or even complete for the language being compiled. Recently, though, a few attempts have been made to derive the machines through mathematical techniques. In 1987 Kursawe showed how to derive the

unification related features of the WAM through iterative partial evaluation of a Prolog meta-interpreter [Kursawe, 1987]. Nilsson has extended that work to the control structures of the WAM [Nilsson, 1990]. Unfortunately, this work has been difficult, and slow to produce results.

In contrast, Börger has had much success with a somewhat different technique, which grew out of his work for the Prolog standardization committee [Börger, 1990]. Börger had already given a precise operational semantics for real (ie. non-pure) Prolog in terms of Gurevich's evolving algebras. He was then able to refine these descriptions until they gave a specification of the underlying architecture [Börger and Rosenzweig, 1991].

I plan to pursue both these techniques on a parallel track. While Börger's methods appear to yield results more quickly, the partial evaluation methods yield results that are more arguably "correct". The structures involved in this technique are well understood, and many properties are guaranteed by the logic.

It should be noted that, due to the embedding given in Chapter 4, an abstract machine for this language would also be suitable for first-order hereditary Harrop formulas. I would expect, therefore, that the result would be related to the WAM extensions formulated by Nadathur and Jayaraman to handle the implicational goals of λProlog. [Jayaraman and Nadathur, 1991]. The principal goal is that, whatever system results, it should be as conservative as possible in that it should cause little or no additional overhead relative to the WAM wen running in the (embedded) Horn clause fragment, and little or no overhead relative to the Nadathur/Jayaraman proposal when operating on the (embedded) hereditary Harrop fragment.

11.5 Partial Evaluation of Linear Logic Programs

In the last section the technique of partial evaluation was mentioned. In that context I was using the term to refer to unrolling an interpreter to analyze its behavior. In general, though, partial evaluation is a powerful optimization technique that can be used by compilers to increase the efficiency of generated code.

In the case of logic programming, partial evaluation takes the form of the *fold/unfold* operations which amount to compile time forward and backward chaining. For example, a Prolog compiler may recognize that the program:

```
a :- b, c.
b :- d, e.
d.
   .
   .
   .
```

can be safely converted to the program:

```
a :- e, c.
b :- e.
d.
   .
   .
   .
```

if the only rules for `b` and `d` are the ones shown. This is accomplished by iteratively unrolling the definition of `b` in the position where it is called in the rule for `a`.

One of the greatest problems with `assert/retract`, from the point of view of implementors, is that they make it almost impossible to apply these techniques. Because of the logical basis of resource handling in linear logic it should be possible to define fold/unfold operations for this language that are sound and complete even over dynamic programs.

The definitions of these operations are not immediately obvious, however. In particular, as elements of the bounded portion of a proof context are unfolded into elements of the unbounded part, the interactions are subtle. Consider the proof context $\langle a \multimap b; a \rangle$. Unfolding the rule in the unbounded portion results in the new context $\langle \varnothing; b \,\&\, a \rangle$. This makes sense since the original context allows a proof of either a or b, but not both.

Applying unfold to the context $\langle b \multimap (a \multimap c); a, b \rangle$ presents some problems, however. Logically, this is equivalent to the context $\langle (b \otimes a) \multimap c; a, b \rangle$. Then, using the argument above, unfolding yields $\langle \varnothing; c \,\&\, (a \otimes b) \rangle$. Again, this makes sense, since it says that in the original context we could prove either c (consuming a and b along the way) or prove both of a and b. Unfortunately, the resulting context is not valid in the language as described thus far, as it contains a negative occurence of $\otimes$. This implies that defining fold/unfold may necessitate extending the logic somewhat.

Appendix A

Intuitionistic Logic and the Existential Property

Intuitionistic logic, proposed by Brouwer, grew out of the constructivist philosophical movement of the late 1800's. The constructivists felt that mathematical reasoning, as exemplified by the use of techniques like proof by contradiction, was deeply flawed. Consider the following classical proof:

> Is there a pair of irrational numbers, a and b such that a^b is rational?
>
> **Proof.** Suppose $a = b = \sqrt{2}$, which is irrational. Now if $\sqrt{2}^{\sqrt{2}}$ is rational, a^b is rational, and we are done. If not, let $a = \sqrt{2}^{\sqrt{2}}$ and let $b = \sqrt{2}$, both of which are (by assumption) irrational. Then $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = \sqrt{2}^2 = 2$, and we are done. ■

From the constructivist point of view this is not much of a proof, since we still don't have an actual example of a pair of irrational numbers with the desired property. We only know (or believe) that such a pair exists. The constructivists argue that if you want to convince them that something exists, your proof should construct an example of such a thing. In formal terms, the constructivists argue that any reasonable logic should have the following two properties:

Definition A.1 A logic S has the *existential property* if, whenever $\vdash_S \exists x.G$ for some formula G, there is some value (called the *witness*) t for the variable x such that $\vdash_S G[x \mapsto t]$.

Definition A.2 A logic S has the *disjunctive property* if, whenever $\vdash_S G_1 \vee G_2$ for some formulae G_1 and G_2, then either $\vdash_S G_1$ or $\vdash_S G_2$.

Brouwer identified the *law of the excluded middle* ($B \vee \neg B$), which is an axiom of classical logic, as the principal cause of non-constructivity in logical systems. In the proof above, it was assumed that either $\sqrt{2}^{\sqrt{2}}$ was rational, or it was not; then something was argued for each

case. The problem is that we never know which case actually holds. Brouwer argued that a logic based on intuition should never make such an appeal.[1]

In the Gentzen calculus there are two ways to introduce the law of the excluded middle. On way is to introduce it explicitly with a new axiom:

$$\frac{}{\longrightarrow B \vee \neg B}\ \mathit{middle}$$

The other way is to expand the definition of sequents to allow multisets of formulae on both sides of the arrow, and change the interpretation of a sequent to: "if all of the formulae on the left are true, then at least one of the formulae on the right is true." In this *multiple-conclusion* setting it is possible to derive the law of the excluded middle as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{B \longrightarrow B}}{\longrightarrow B, \neg B}\ \neg_R}{\longrightarrow B, B \vee \neg B}\ \vee_{R_2}}{\longrightarrow B \vee \neg B, B \vee \neg B}\ \vee_{R_1}}{\longrightarrow B \vee \neg B}\ C_R$$

Restricting sequents to a single formula in the succedent bars this derivation and guarantees the existential and disjunctive properties.

As stated in Chapter 2, the importance of the computed answer substitution in logic programming is a good indicator that logic programming is a game of intuitionistic logic. However, the existential and disjunctive properties are not enough to guarantee that a valid logic programming query has a computed answer substitution. These properties are defined only in terms of a top-level existential (or disjunction) in a formula provable from an empty context: that is, a tautology.

So, for instance, in intuitionistic logic it is possible to prove $p \vee q$ in the context $p \vee q$, while neither p nor q alone is provable from that context. This can be seen from the structure of the following proof:

$$\dfrac{\dfrac{\overline{p \longrightarrow p}}{p \longrightarrow p \vee q}\ \vee_{R_1} \quad \dfrac{\overline{q \longrightarrow q}}{q \longrightarrow p \vee q}\ \vee_{R_2}}{p \vee q \longrightarrow p \vee q}\ \vee_L$$

If, however, we restrict the sorts of formulae that are allowed to occur on the left hand side of a sequent arrow, then it is possible to extend the existential and disjunctive properties considerably. Harrop showed that if we define the following set of formulae (where B is an arbitrary first-order formula):

$$D := A \mid B \supset D \mid \forall x.D \mid D_1 \wedge D_2$$

[1] There are several rules, such as $\neg\neg B \equiv B$, that either derive from, or are equivalent to, the law of the excluded middle. Constructivists bar the use of any of these.

and then let Γ be a set of D-formulae and B an arbitrary formula, then sequents of the form $\Gamma \longrightarrow B$ have the existential property [Harrop, 1960]. This result follows because such sequents are provable if and only if they have a proof in which the last rule is the introduction rule for the principal operator of the succedent. That is, they have proofs that are uniform at the root.

Miller has shown an even richer result. If Γ is a program built from hereditary Harrop formulae as defined in Definition 3.2, and G is in the related class of goals, then, if $\Gamma \longrightarrow G$ is provable, then, for every sequent $\Gamma' \longrightarrow \exists x.G'$ that occurs in the proof, there is some value t for x such that $\Gamma' \longrightarrow G'[x \mapsto t]$ is provable (and similarly for disjunctions).

These results guarantee the existence of computed answer substitutions in logic programming languages based on hereditary Harrop formulae. Since Horn clauses are just a proper sublanguage of hereditary Harrop formulae, the same result applies for Horn clause programming. Harland has recently shown that, in fact, hereditary Harrop formulae form the largest sublanguage of intuitionistic logic for which this property holds [Harland, 1993].

Appendix B

Some Small Properties of $\mathcal{L}$ Proofs

In this appendix we prove a few technical properties about $\mathcal{L}$ sequent proofs. These properties will be needed in the proofs given in subsequent appendices.

B.1 Rectifying Proofs

The first property is actually stated without proof, since the proof is a standard one.

Proposition B.1 Any proof in intuitionistic logic, linear logic, or $\mathcal{L}$ can be *rectified* so that all eigenconstants (that is, the constants introduced by instances of the $\forall_R$ and $\exists_L$ rules) are distinct, and distinct from all other constants in the proof.

Proof. See [Gentzen, 1969]. ∎
From this point on we will restrict our view to rectified proofs with atomic intial sequents.

B.2 Adding Resources to the Unbounded Contexts of Proofs

Definition B.1 If Ξ is a valid $\mathcal{L}$proof, and Γ is a multiset of linear hereditary Harrop formulae, then $\Xi^{+\Gamma}$ is the structure that results from adding the formulae in Γ to the intuitionistic context of every sequent in Ξ.

Proposition B.2 Given a cut-free $\mathcal{L}$-proof Ξ of the sequent $\Gamma'; \Delta \longrightarrow B$, the structure $\Xi^{+\Gamma}$ is an $\mathcal{L}$-proof of the sequent $\Gamma', \Gamma; \Delta \longrightarrow B$.

Proof. We will assume that the proof has been further rectified so that the eigenconstants are distinct from any constants in Γ. The proof is by induction on the structure of Ξ:

Base Cases: There are two base cases, corresponding to the two axiom schemata of $\mathcal{L}$.

$$id: \quad \Xi = \quad \overline{\Gamma'; B \longrightarrow B} \qquad \Xi^{+\Gamma} = \quad \overline{\Gamma', \Gamma; B \longrightarrow B}$$

$$\top R: \quad \Xi = \quad \overline{\Gamma'; \Delta \longrightarrow \top} \qquad \Xi^{+\Gamma} = \quad \overline{\Gamma', \Gamma; \Delta \longrightarrow \top}$$

In both cases, $\Xi^{+\Gamma}$ is a valid $\mathcal{L}$ axiom instance.

Inductive Hypothesis: In each of the following cases we assume that the Ξ_i are $\mathcal{L}$-proofs and the $\Xi_i^{+\Gamma}$ are the valid $\mathcal{L}$-proofs which result from adding Γ to the intuitionistic contexts of every sequent in the Ξ_i.

Inductive Step:

$$absorb: \quad \Xi = \dfrac{\overset{\Xi}{\Gamma', B; \Delta, B \longrightarrow C}}{\Gamma', B; \Delta \longrightarrow C}\ absorb \qquad \Xi^{+\Gamma} = \dfrac{\overset{\Xi^{+\Gamma}}{\Gamma', \Gamma, B; \Delta, B \longrightarrow C}}{\Gamma', \Gamma, B; \Delta \longrightarrow C}\ absorb$$

$$\&_R: \quad \Xi = \dfrac{\overset{\Xi_1}{\Gamma'; \Delta \longrightarrow B} \quad \overset{\Xi_2}{\Gamma'; \Delta \longrightarrow C}}{\Gamma'; \Delta \longrightarrow B \,\&\, C}\ \&_R \qquad \Xi^{+\Gamma} = \dfrac{\overset{\Xi_1^{+\Gamma}}{\Gamma', \Gamma; \Delta \longrightarrow B} \quad \overset{\Xi_2^{+\Gamma}}{\Gamma', \Gamma; \Delta \longrightarrow C}}{\Gamma', \Gamma; \Delta \longrightarrow B \,\&\, C}\ \&_R$$

$$\&_{Li}: \quad \Xi = \dfrac{\overset{\Xi}{\Gamma'; \Delta, B_i \longrightarrow C}}{\Gamma'; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&_{Li} \qquad \Xi^{+\Gamma} = \dfrac{\overset{\Xi^{+\Gamma}}{\Gamma', \Gamma; \Delta, B_i \longrightarrow C}}{\Gamma', \Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&_{Li}$$

$$\multimap_R: \quad \Xi = \dfrac{\overset{\Xi}{\Gamma'; \Delta, B \longrightarrow C}}{\Gamma'; \Delta \longrightarrow B \multimap C}\ \multimap_R \qquad \Xi^{+\Gamma} = \dfrac{\overset{\Xi^{+\Gamma}}{\Gamma', \Gamma; \Delta, B \longrightarrow C}}{\Gamma', \Gamma; \Delta \longrightarrow B \multimap C}\ \multimap_R$$

$$\multimap_L: \quad \Xi = \dfrac{\overset{\Xi_1}{\Gamma'; \Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma'; \Delta_2, C \longrightarrow E}}{\Gamma'; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap_L \qquad \Xi^{+\Gamma} = \dfrac{\overset{\Xi_1^{+\Gamma}}{\Gamma', \Gamma; \Delta_1 \longrightarrow B} \quad \overset{\Xi_2^{+\Gamma}}{\Gamma', \Gamma; \Delta_2, C \longrightarrow E}}{\Gamma', \Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap_L$$

$$\Rightarrow_R: \quad \Xi = \dfrac{\overset{\Xi}{\Gamma', B; \Delta \longrightarrow C}}{\Gamma'; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow_R \qquad \Xi^{+\Gamma} = \dfrac{\overset{\Xi^{+\Gamma}}{\Gamma', \Gamma, B; \Delta \longrightarrow C}}{\Gamma', \Gamma; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow_R$$

$$\Rightarrow_L: \quad \Xi = \dfrac{\overset{\Xi_1}{\Gamma'; \varnothing \longrightarrow B} \quad \overset{\Xi_2}{\Gamma'; \Delta, C \longrightarrow E}}{\Gamma'; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow_L \qquad \Xi^{+\Gamma} = \dfrac{\overset{\Xi_1^{+\Gamma}}{\Gamma', \Gamma; \varnothing \longrightarrow B} \quad \overset{\Xi_2^{+\Gamma}}{\Gamma', \Gamma; \Delta, C \longrightarrow E}}{\Gamma', \Gamma; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow_L$$

$$\forall_R{}^{1}: \quad \Xi = \dfrac{\overset{\Xi}{\Gamma'; \Delta \longrightarrow B[x \mapsto c]}}{\Gamma'; \Delta \longrightarrow \forall x.B}\ \forall_R \qquad \Xi^{+\Gamma} = \dfrac{\overset{\Xi^{+\Gamma}}{\Gamma', \Gamma; \Delta \longrightarrow B[x \mapsto c]}}{\Gamma', \Gamma; \Delta \longrightarrow \forall x.B}\ \forall_R$$

$$\forall_L: \quad \Xi = \dfrac{\overset{\Xi}{\Gamma'; \Delta, B[x \mapsto t] \longrightarrow C}}{\Gamma'; \Delta, \forall x.B \longrightarrow C}\ \forall_L \qquad \Xi^{+\Gamma} = \dfrac{\overset{\Xi^{+\Gamma}}{\Gamma', \Gamma; \Delta, B[x \mapsto t] \longrightarrow C}}{\Gamma', \Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall_L$$

The correctness of the resulting structures in each case is apparent. Therefore the proposition is proved. ∎

[1] If the proof were not rectified first, this case would require manipulating the eigenconstant if it were the same as a constant in Γ.

B.3 Restricting the Form of the Cut Rules

As a corollary to the last proposition, we may now assume, without loss of generality, that the cut rules of $\mathcal{L}$ have the restricted form:

$$\frac{\Gamma;\Delta_1 \longrightarrow B \quad \Gamma;\Delta_2, B \longrightarrow C}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\ \mathit{cut} \quad \text{and} \quad \frac{\Gamma;\varnothing \longrightarrow B \quad \Gamma, B;\Delta \longrightarrow C}{\Gamma;\Delta \longrightarrow C}\ \mathit{cut}!$$

since, for instance, a proof of the form:

$$\frac{\overset{\Xi_1}{\Gamma_1;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma_2;\Delta_2, B \longrightarrow C}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \longrightarrow C}\ \mathit{cut}$$

can be replaced by:

$$\frac{\overset{\Xi_1{}^{+\Gamma_2}}{\Gamma_1,\Gamma_2;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2{}^{+\Gamma_1}}{\Gamma_1,\Gamma_2;\Delta_2, B \longrightarrow C}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \longrightarrow C}\ \mathit{cut}$$

If we make this restriction, then the structure of the $\mathcal{L}$ rules is such that we can, and will, assume that in every upward path from the root of a $\mathcal{L}$proof the intuitionistic context is monotonically increasing.

Appendix C

A Proof of the Equivalence of $\mathcal{L}$ and a Fragment of $\mathcal{ILL}$

Proposition 4.1 Let G be a goal formula, and Γ and Δ multisets of definite formulae as defined in Chapter 4. Let B° be the result of recursively replacing all occurrences of $C_1 \Rightarrow C_2$ in B with $(!C_1^\circ) \multimap C_2^\circ$, and let $\Gamma^\circ = \{C^\circ \mid C \in \Gamma\}$. Then the sequent $\Gamma; \Delta \longrightarrow B$ is $\mathcal{L}$-provable if and only if the sequent $!(\Gamma^\circ), \Delta^\circ \longrightarrow B^\circ$ is $\mathcal{ILL}$-provable.

Proof. The proof is in two parts. Each part consists of giving a terminating algorithm to convert proofs in one of the systems to proofs in the other.

$\mathcal{L}$-provability implies $\mathcal{ILL}$-provability: We give a conversion of an $\mathcal{L}$ proof of the sequent $\Gamma; \Delta \longrightarrow B$ to an $\mathcal{ILL}$ proof of the sequent $!(\Gamma^\circ), \Delta^\circ \longrightarrow B^\circ$. In the resulting proof, inferences marked $!_W^*$ and $!_C^*$ represent multiple instances of the individual weakening and contraction rules, as needed.

Base Cases: In each case, the converted proof is a valid axiom of $\mathcal{ILL}$.

$$\textit{identity}: \quad \frac{}{\Gamma; B \longrightarrow B} \quad \rightsquigarrow \quad \frac{\dfrac{}{B^\circ \longrightarrow B^\circ}}{!\Gamma^\circ, B^\circ \longrightarrow B^\circ}\, !W^*$$

$$\top_R: \quad \frac{}{\Gamma; \Delta \longrightarrow \top}\, \top_R \quad \rightsquigarrow \quad \frac{}{!\Gamma^\circ, \Delta^\circ \longrightarrow \top}\, \top_R$$

Induction Hypothesis: If Ξ_i is an $\mathcal{L}$ proof of the sequent $\Gamma_i; \Delta_i \longrightarrow B_i$ then Ξ_i° is a the result of converting that proof to a valid $\mathcal{ILL}$ proof of the sequent $!\Gamma_i^\circ, \Delta_i^\circ \longrightarrow B_i^\circ$.

Inductive Step: Select any rule instance that is the root of a sub-proof that has been entirely converted and convert it to an $\mathcal{ILL}$ derivation according to the scheme below:

$$absorb: \quad \frac{\Gamma, B; \Delta, \overset{\Xi}{B} \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ absorb \quad \rightsquigarrow \quad \frac{\dfrac{!\Gamma^\circ, !B^\circ, \Delta^\circ, \overset{\Xi^\circ}{B^\circ} \longrightarrow C^\circ}{!\Gamma^\circ, !B^\circ, !B^\circ, \Delta^\circ \longrightarrow C^\circ}\ !_L}{!\Gamma^\circ, !B^\circ, \Delta^\circ \longrightarrow C^\circ}\ !_C$$

$$\&_R: \quad \frac{\Gamma; \Delta \longrightarrow \overset{\Xi_1}{B} \quad \Gamma; \Delta \longrightarrow \overset{\Xi_2}{C}}{\Gamma; \Delta \longrightarrow B \,\&\, C}\ \&_R \quad \rightsquigarrow \quad \frac{!\Gamma^\circ, \Delta^\circ \longrightarrow \overset{\Xi_1^\circ}{B^\circ} \quad !\Gamma^\circ, \Delta^\circ \longrightarrow \overset{\Xi_2^\circ}{C^\circ}}{!\Gamma^\circ, \Delta^\circ \longrightarrow B^\circ \,\&\, C^\circ}\ \&_R$$

$$\&_L: \quad \frac{\Gamma; \Delta, \overset{\Xi}{B_i} \longrightarrow C}{\Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&_{L_i} \quad \rightsquigarrow \quad \frac{!\Gamma^\circ, \Delta^\circ, \overset{\Xi^\circ}{B_i^\circ} \longrightarrow C^\circ}{!\Gamma^\circ, \Delta^\circ, B_1^\circ \,\&\, B_2^\circ \longrightarrow C^\circ}\ \&_{L_i}$$

$$\multimap_R: \quad \frac{\Gamma; \Delta, \overset{\Xi}{B} \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}\ \multimap_R \quad \rightsquigarrow \quad \frac{!\Gamma^\circ, \Delta^\circ, \overset{\Xi^\circ}{B^\circ} \longrightarrow C^\circ}{!\Gamma^\circ, \Delta^\circ \longrightarrow B^\circ \multimap C^\circ}\ \multimap_R$$

$$\multimap_L: \quad \frac{\Gamma; \Delta_1 \longrightarrow \overset{\Xi_1}{B} \quad \Gamma; \Delta_2, \overset{\Xi_2}{C} \longrightarrow E}{\Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap_L \quad \rightsquigarrow \quad \frac{\dfrac{!\Gamma^\circ, \Delta_1^\circ \longrightarrow \overset{\Xi_1^\circ}{B^\circ} \quad !\Gamma^\circ, \Delta_2^\circ, \overset{\Xi_2^\circ}{C^\circ} \longrightarrow E^\circ}{!\Gamma^\circ, !\Gamma^\circ, \Delta_1^\circ, \Delta_2^\circ, B^\circ \multimap C^\circ \longrightarrow E^\circ}\ \multimap_L}{!\Gamma^\circ, \Delta_1^\circ, \Delta_2^\circ, B^\circ \multimap C^\circ \longrightarrow E^\circ}\ !_C^*$$

$$\Rightarrow_R: \quad \frac{\Gamma, B; \Delta \overset{\Xi}{\longrightarrow} C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow_R \quad \rightsquigarrow \quad \frac{!\Gamma^\circ, !B^\circ, \Delta^\circ \overset{\Xi^\circ}{\longrightarrow} C^\circ}{!\Gamma^\circ, \Delta^\circ \longrightarrow !B^\circ \multimap C^\circ}\ \multimap_R$$

$$\Rightarrow_L: \quad \frac{\Gamma; \varnothing \longrightarrow \overset{\Xi_1}{B} \quad \Gamma; \Delta, \overset{\Xi_2}{C} \longrightarrow E}{\Gamma; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow_L \quad \rightsquigarrow \quad \frac{\dfrac{!\Gamma^\circ, \varnothing \longrightarrow \overset{\Xi_1^\circ}{B^\circ}}{!\Gamma^\circ, \varnothing \longrightarrow !B^\circ}\ !_R \quad !\Gamma^\circ, \Delta^\circ, \overset{\Xi_2^\circ}{C^\circ} \longrightarrow E^\circ}{\dfrac{!\Gamma^\circ, !\Gamma^\circ, \Delta^\circ, !B^\circ \multimap C^\circ \longrightarrow E^\circ}{!\Gamma^\circ, \Delta^\circ, !B^\circ \multimap C^\circ \longrightarrow E^\circ}\ !_C^*}\ \multimap_L$$

$$\forall_R: \quad \frac{\Gamma; \Delta \longrightarrow \overset{\Xi}{B[x \mapsto c]}}{\Gamma; \Delta \longrightarrow \forall x.B}\ \forall_R \quad \rightsquigarrow \quad \frac{!\Gamma^\circ, \Delta^\circ \longrightarrow \overset{\Xi^\circ}{B^\circ[x \mapsto c]}}{!\Gamma^\circ, \Delta^\circ \longrightarrow \forall x.B^\circ}\ \forall_R$$

$$\forall_L: \quad \frac{\Gamma; \Delta, \overset{\Xi}{B[x \mapsto t]} \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall_L \quad \rightsquigarrow \quad \frac{!\Gamma^\circ, \Delta^\circ, \overset{\Xi^\circ}{(B[x \mapsto t])^\circ} \longrightarrow C^\circ}{!\Gamma^\circ, \Delta^\circ, \forall x.B^\circ \longrightarrow C^\circ}\ \forall_L$$

This last case depends on the reduction $B^\circ[x \mapsto t] = (B[x \mapsto t])^\circ$ which is immediate in the first-order setting.

Since each choice of the inductive case selects an unconverted rule and converts it without creating any new unconverted rules, the termination of this procedure is obvious. In addition, in each case, the validity of the converted structure is apparent.

$\mathcal{ILL}$-provability implies $\mathcal{L}$-provability: We give a conversion of an $\mathcal{ILL}$ proof of the sequent

$!\Gamma^\circ, \Delta^\circ \longrightarrow B^\circ$ to an $\mathcal{L}$ proof of the sequent $\Gamma; \Delta \longrightarrow B$.

Note that in an $\mathcal{ILL}$ proof of a sequent which arises from the translation of an $\mathcal{L}$-sequent any sub-proof whose endsequent has $!B$ as its consequent, must, of necessity, have an instance of $!_R$ as its last rule, or end in a sequence of left $!$ rules preceeded by $!_R$. It is simple to show that such a sequence of proof steps may be permuted so that the $!_R$ is the last rule. In order to simplify the following proof, we will, without loss of generality, assume that the linear logic proofs have been pre-proccessed in this manner.

Because we are concerned only with proofs of sequents whose formulae, B°, are the conversion of formulae freely generated by &, $\multimap$, $\Rightarrow$, and $\forall$, and since the logic has cut-elimination and the sub-formula property, we need not consider the $\otimes$ rules. Further, because we are restricting our view to proofs with atomic initial sequents, these restrictions, together with the pre-processing described above, insure that instances of $!R$ will occur only immediately before instances of $\multimap_L$.

Base Cases: In each case, the converted proof is a valid axiom of $\mathcal{L}$.

$$\textit{identity}: \quad \frac{}{B \longrightarrow B} \quad \rightsquigarrow \quad \frac{}{\varnothing; B \longrightarrow B}$$

$$\top_R: \quad \frac{}{!\Gamma^\circ, \Delta^\circ \longrightarrow \top}\,\top_R \quad \rightsquigarrow \quad \frac{}{\Gamma; \Delta \longrightarrow \top}\,\top_R$$

Inductive Hypothesis: If Ξ_i° is an $\mathcal{ILL}$ proof of the sequent $!\Gamma_i^\circ, \Delta_i^\circ \longrightarrow B_i^\circ$, then Ξ_i is a the result of converting that proof to a valid $\mathcal{L}$ proof of the sequent $\Gamma_i; \Delta_i \longrightarrow B_i$.

Inductive Step: Select any rule instance that is the root of a sub-proof that has been entirely

converted and convert it to an $\mathcal{L}$ derivation according to the scheme below.

$$\&_R: \quad \frac{\overset{\Xi_1^\circ}{!\Gamma^\circ, \Delta^\circ \longrightarrow B^\circ} \quad \overset{\Xi_2^\circ}{!\Gamma^\circ, \Delta^\circ \longrightarrow C^\circ}}{!\Gamma^\circ, \Delta^\circ \longrightarrow B^\circ \,\&\, C^\circ}\ \&_R \quad \rightsquigarrow \quad \frac{\overset{\Xi_1}{\Gamma; \Delta \longrightarrow B} \quad \overset{\Xi_2}{\Gamma; \Delta \longrightarrow C}}{\Gamma; \Delta \longrightarrow B \,\&\, C}\ \&_R$$

$$\&_L: \quad \frac{\overset{\Xi^\circ}{!\Gamma^\circ, \Delta^\circ, B_i^\circ \longrightarrow C^\circ}}{!\Gamma^\circ, \Delta^\circ, B_1^\circ \,\&\, B_2^\circ \longrightarrow C^\circ}\ \&_{L_i} \quad \rightsquigarrow \quad \frac{\overset{\Xi}{\Gamma; \Delta, B_i \longrightarrow C}}{\Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&_{L_i}$$

$$\multimap_R: \quad \frac{\overset{\Xi^\circ}{!\Gamma^\circ, !B^\circ, \Delta^\circ \longrightarrow C^\circ}}{!\Gamma^\circ, \Delta^\circ \longrightarrow\, !B^\circ \multimap C^\circ}\ \multimap_R \quad \rightsquigarrow \quad \frac{\overset{\Xi}{\Gamma, B; \Delta \longrightarrow C}}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow_R$$

$$\frac{\overset{\Xi^\circ}{!\Gamma^\circ, \Delta^\circ, B^\circ \longrightarrow C^\circ}}{!\Gamma^\circ, \Delta^\circ \longrightarrow B^\circ \multimap C^\circ}\ \multimap_R \quad \rightsquigarrow \quad \frac{\overset{\Xi}{\Gamma; \Delta, B \longrightarrow C}}{\Gamma; \Delta \longrightarrow B \multimap C}\ \multimap_R$$

$$\multimap_L: \quad \frac{\overset{\Xi_1^\circ}{!\Gamma_1^\circ \longrightarrow B^\circ}}{!\Gamma_1^\circ \longrightarrow\, !B^\circ} \quad \frac{\overset{\Xi_2^\circ}{!\Gamma_2^\circ, \Delta^\circ, C^\circ \longrightarrow E^\circ}}{!\Gamma_1^\circ, !\Gamma_2^\circ, \Delta^\circ, !B^\circ \multimap C^\circ \longrightarrow E^\circ}\ \multimap_L \quad \rightsquigarrow \quad \frac{\overset{\Xi_1{}^{+\Gamma_2}}{\Gamma_1, \Gamma_2; \varnothing \longrightarrow B} \quad \overset{\Xi_2{}^{+\Gamma_1}}{\Gamma_1, \Gamma_2; \Delta, C \longrightarrow E}}{\Gamma_1, \Gamma_2; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow_L$$

$$\frac{\overset{\Xi_1^\circ}{!\Gamma_1^\circ, \Delta_1^\circ \longrightarrow B^\circ} \quad \overset{\Xi_2^\circ}{!\Gamma_2^\circ, \Delta_2^\circ, C^\circ \longrightarrow E^\circ}}{!\Gamma_1^\circ, !\Gamma_2^\circ, \Delta_1^\circ, \Delta_2^\circ, B^\circ \multimap C^\circ \longrightarrow E^\circ}\ \multimap_L \quad \rightsquigarrow \quad \frac{\overset{\Xi_1{}^{+\Gamma_2}}{\Gamma_1, \Gamma_2; \Delta_1 \longrightarrow B} \quad \overset{\Xi_2{}^{+\Gamma_1}}{\Gamma_1, \Gamma_2; \Delta_2, C \longrightarrow E}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap_L$$

$$!_W: \quad \frac{\overset{\Xi^\circ}{!\Gamma^\circ, \Delta^\circ \longrightarrow C^\circ}}{!\Gamma^\circ, !B^\circ, \Delta^\circ \longrightarrow C^\circ}\ !_W \quad \rightsquigarrow \quad \overset{\Xi^{+B}}{\Gamma, B; \Delta \longrightarrow C}$$

$$!_C: \quad \frac{\overset{\Xi^\circ}{!\Gamma^\circ, !B^\circ, !B^\circ, \Delta^\circ \longrightarrow C^\circ}}{!\Gamma^\circ, !B, \Delta^\circ \longrightarrow C^\circ}\ !_C \quad \rightsquigarrow \quad \Gamma, B; \overset{\Xi}{\Delta \longrightarrow C}$$

$$!_L: \quad \frac{\overset{\Xi^\circ}{!\Gamma^\circ, \Delta^\circ, B^\circ \longrightarrow C^\circ}}{!\Gamma^\circ, !B^\circ, \Delta^\circ \longrightarrow C^\circ}\ !_L \quad \rightsquigarrow \quad \frac{\overset{\Xi^{+B}}{\Gamma, B; \Delta, B \longrightarrow C}}{\Gamma, B; \Delta \longrightarrow C}$$

$$\forall_L: \quad \frac{\overset{\Xi^\circ}{!\Gamma^\circ, \Delta^\circ, B^\circ[x \mapsto t] \longrightarrow C^\circ}}{!\Gamma^\circ, \Delta^\circ, \forall x.B^\circ \longrightarrow C^\circ}\ \forall_L \quad \rightsquigarrow \quad \frac{\overset{\Xi}{\Gamma; \Delta, B[x \mapsto t] \longrightarrow C}}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall_L$$

$$\forall_R: \quad \frac{\overset{\Xi^\circ}{!\Gamma^\circ, \Delta^\circ \longrightarrow B^\circ[x \mapsto c]}}{!\Gamma^\circ, \Delta^\circ \longrightarrow \forall x.B^\circ}\ \forall_R \quad \rightsquigarrow \quad \frac{\overset{\Xi}{\Gamma; \Delta \longrightarrow B[x \mapsto c]}}{\Gamma; \Delta \longrightarrow \forall x.B}\ \forall_R$$

Since each iteration of the inductive case selects an unconverted rule and converts it without creating any new unconverted rules, the termination of this procedure is obvious. In addition, in each case, the validity of the converted structure is apparent.

Thus we have provided correct terminating algorithms to convert proofs from each system

into the other. Provability in the two systems is therefore equivalent. ■

Appendix D

A Proof of Cut Elimination for the System $\mathcal{L}$

Proposition D.1 The two cut rules of system $\mathcal{L}$ are admissible. That is, any $\mathcal{L}$ proof can be converted to an equivalent proof of the same endsequent in which neither cut rule occurs.

Proof. In the traditional manner, the cut-elimination property for $\mathcal{L}$ is shown by giving an algorithm to remove all occurrences of either cut rule, together with a proof that the algorithm terminates on any given proof. The algorithm itself has two main phases. The first phase removes all instances of the *cut*! rule, replacing each with (possibly) multiple instances of *cut*. The second phase, removes instances of *cut* from a proof with no instances of *cut*!.

Phase I Given a proof Ξ:

1. If Ξ has no occurrences of *cut*! then this phase is complete.
2. Select a topmost occurrence of *cut*!; that is, an occurrence for which the subproofs of its premises contain no instances of *cut*!. That occurrence may be assumed to be the root of a subproof of the form:

$$\Xi_0 = \dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \overset{\Xi_2}{\Gamma, B;\Delta \longrightarrow C}}{\Gamma;\Delta \longrightarrow C}\ \textit{cut}!$$

3. We now distinguish cases based on the last rule of the subproof Ξ_2. In the first two cases —where Ξ_2 is just an instance of *identity* or $\top_R$— the instance of *cut*! is removed entirely. In the remaining case the rules are permuted such that the *cut*! has a smaller proof over its right premise. In some cases the *cut*! rule will be duplicated, but again each new instance will have a smaller proof of its right premise. Note that in the third case an instance of *cut* is created as well. This will happen each time an instance of *cut*! passes through an *absorb* on the cut formula.

$$\textit{identity}: \quad \dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \dfrac{}{\Gamma, B; C \longrightarrow C}}{\Gamma; C \longrightarrow C}\ \textit{cut}!$$

$$\rightsquigarrow \quad \dfrac{}{\Gamma; C \longrightarrow C}$$

$$\top_R: \quad \dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \dfrac{}{\Gamma, B; \Delta \longrightarrow \top}\ \top_R}{\Gamma; \Delta \longrightarrow \top}\ \textit{cut}!$$

$$\rightsquigarrow \quad \dfrac{}{\Gamma; \Delta \longrightarrow \top}\ \top_R$$

$$\textit{absorb}_1: \quad \dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma, B; \Delta, B \longrightarrow C}}{\Gamma, B; \Delta \longrightarrow C}\ \textit{absorb}}{\Gamma; \Delta \longrightarrow C}\ \textit{cut}!$$

$$\rightsquigarrow \quad \dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \overset{\Xi_2}{\Gamma, B; \Delta, B \longrightarrow C}}{\Gamma; \Delta, B \longrightarrow C}\ \textit{cut}!}{\Gamma; \Delta \longrightarrow C}\ \textit{cut}$$

$$\textit{absorb}_2: \quad \dfrac{\overset{\Xi_1}{\Gamma, D;\varnothing \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma, D, B; \Delta, D \longrightarrow C}}{\Gamma, D, B; \Delta \longrightarrow C}\ \textit{absorb}}{\Gamma, D; \Delta \longrightarrow C}\ \textit{cut}!$$

$$\rightsquigarrow \quad \dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma, D, B; \Delta, D \longrightarrow C}}{\Gamma, D; \Delta, D \longrightarrow C}\ \textit{absorb}}{\Gamma, D; \Delta \longrightarrow C}\ \textit{cut}!$$

$$\textit{cut}: \quad \dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma, B; \Delta_1 \longrightarrow B'} \quad \overset{\Xi_3}{\Gamma, B; \Delta_2, B' \longrightarrow C}}{\Gamma, B; \Delta_1, \Delta_2 \longrightarrow C}\ \textit{cut}}{\Gamma; \Delta_1, \Delta_2 \longrightarrow C}\ \textit{cut}!$$

$$\rightsquigarrow \quad \dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \overset{\Xi_2}{\Gamma, B; \Delta_1 \longrightarrow B'}}{\Gamma; \Delta_1 \longrightarrow B'}\ \textit{cut}! \quad \dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow B} \quad \overset{\Xi_3}{\Gamma, B; \Delta_2, B' \longrightarrow C}}{\Gamma; \Delta_2, B' \longrightarrow C}\ \textit{cut}!}{\Gamma; \Delta_1, \Delta_2 \longrightarrow C}\ \textit{cut}$$

$$\&_R: \quad \dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \qquad \dfrac{\Gamma, B;\Delta \longrightarrow C_1 \;(\Xi_2) \quad \Gamma, B;\Delta \longrightarrow C_2 \;(\Xi_3)}{\Gamma, B;\Delta \longrightarrow C_1 \,\&\, C_2}\,\&_R}{\Gamma;\Delta \longrightarrow C_1 \,\&\, C_2}\,cut!$$

$$\rightsquigarrow \quad \dfrac{\dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \quad \Gamma, B;\Delta \longrightarrow C_1 \;(\Xi_2)}{\Gamma;\Delta \longrightarrow C_1}\,cut! \qquad \dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \quad \Gamma, B;\Delta \longrightarrow C_2 \;(\Xi_3)}{\Gamma;\Delta \longrightarrow C_2}\,cut!}{\Gamma;\Delta \longrightarrow C_1 \,\&\, C_2}\,\&_R$$

$$\&_L: \quad \dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \qquad \dfrac{\Gamma, B;\Delta, C_i \longrightarrow D \;(\Xi_2)}{\Gamma, B;\Delta, C_1 \,\&\, C_2 \longrightarrow D}\,\&_{L_i}}{\Gamma;\Delta, C_1 \,\&\, C_2 \longrightarrow D}\,cut!$$

$$\rightsquigarrow \quad \dfrac{\dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \quad \Gamma, B;\Delta, C_i \longrightarrow D \;(\Xi_2)}{\Gamma;\Delta, C_i \longrightarrow D}\,cut!}{\Gamma;\Delta, C_1 \,\&\, C_2 \longrightarrow D}\,\&_L$$

$$\multimap_R: \quad \dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \qquad \dfrac{\Gamma, B;\Delta, D \longrightarrow C \;(\Xi_2)}{\Gamma, B;\Delta \longrightarrow D \multimap C}\,\multimap_R}{\Gamma;\Delta \longrightarrow D \multimap C}\,cut!$$

$$\rightsquigarrow \quad \dfrac{\dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \quad \Gamma, B;\Delta, D \longrightarrow C \;(\Xi_2)}{\Gamma;\Delta, D \longrightarrow C}\,cut!}{\Gamma;\Delta \longrightarrow D \multimap C}\,\multimap_R$$

$$\multimap_L: \quad \dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \qquad \dfrac{\Gamma, B;\Delta_1 \longrightarrow D \;(\Xi_2) \quad \Gamma, B;\Delta_2, C \longrightarrow E \;(\Xi_3)}{\Gamma, B;\Delta_1, \Delta_2, D \multimap C \longrightarrow E}\,\multimap_L}{\Gamma;\Delta_1, \Delta_2, D \multimap C \longrightarrow E}\,cut!$$

$$\rightsquigarrow \quad \dfrac{\dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \quad \Gamma, B;\Delta_1 \longrightarrow D \;(\Xi_2)}{\Gamma;\Delta_1 \longrightarrow D}\,cut! \qquad \dfrac{\Gamma;\varnothing \longrightarrow B \;(\Xi_1) \quad \Gamma, B;\Delta_2, C \longrightarrow E \;(\Xi_3)}{\Gamma;\Delta_2, C \longrightarrow E}\,cut!}{\Gamma;\Delta_1, \Delta_2, D \multimap C \longrightarrow E}\,\multimap_L$$

$\Rightarrow_R$:
$$\dfrac{\Gamma;\varnothing \longrightarrow B \quad \overset{\Xi_1}{} \qquad \dfrac{\Gamma, B, D;\Delta \longrightarrow C}{\Gamma, B;\Delta \longrightarrow D \Rightarrow C}\ \Rightarrow_R \overset{\Xi_2}{}}{\Gamma;\Delta \longrightarrow D \Rightarrow C}\ cut!$$

$\rightsquigarrow$
$$\dfrac{\Gamma;\varnothing \longrightarrow B \quad \overset{\Xi_1}{} \qquad \dfrac{\Gamma, B, D;\Delta \longrightarrow C}{\Gamma, D;\Delta \longrightarrow C}\ \overset{\Xi_2}{}}{\Gamma;\Delta \longrightarrow D \Rightarrow C}\ \Rightarrow_R \quad cut!$$

$\Rightarrow_L$:
$$\dfrac{\Gamma;\varnothing \longrightarrow B \quad \overset{\Xi_1}{} \qquad \dfrac{\Gamma, B;\varnothing \longrightarrow D \quad \overset{\Xi_2}{} \qquad \Gamma, B;\Delta, C \longrightarrow E}{\Gamma, B;\Delta, D \Rightarrow C \longrightarrow E}\ \Rightarrow_L \overset{\Xi_3}{}}{\Gamma;\Delta, D \Rightarrow C \longrightarrow E}\ cut!$$

$\rightsquigarrow$
$$\dfrac{\dfrac{\Gamma;\varnothing \longrightarrow B \quad \overset{\Xi_1}{} \qquad \Gamma, B;\varnothing \longrightarrow D \quad \overset{\Xi_2}{}}{\Gamma;\varnothing \longrightarrow D}\ cut! \qquad \dfrac{\Gamma;\varnothing \longrightarrow B \quad \overset{\Xi_1}{} \qquad \Gamma, B;\Delta, C \longrightarrow E \quad \overset{\Xi_3}{}}{\Gamma;\Delta, C \longrightarrow E}\ cut!}{\Gamma;\Delta, D \Rightarrow C \longrightarrow E}\ \Rightarrow_L$$

$\forall_R$:
$$\dfrac{\Gamma;\varnothing \longrightarrow B \quad \overset{\Xi_1}{} \qquad \dfrac{\Gamma, B;\Delta \longrightarrow C[x \mapsto c]}{\Gamma, B;\Delta \longrightarrow \forall x.C}\ \forall_R \overset{\Xi_2}{}}{\Gamma;\Delta \longrightarrow \forall x.C}\ cut!$$

$\rightsquigarrow$
$$\dfrac{\Gamma;\varnothing \longrightarrow B \quad \overset{\Xi_1}{} \qquad \dfrac{\Gamma, B;\Delta \longrightarrow C[x \mapsto c]}{\Gamma;\Delta \longrightarrow C[x \mapsto c]}\ \overset{\Xi_2}{}}{\Gamma;\Delta \longrightarrow \forall x.C}\ \Rightarrow_R \quad cut!$$

$\forall_L$:
$$\dfrac{\Gamma;\varnothing \longrightarrow B \quad \overset{\Xi_1}{} \qquad \dfrac{\Gamma, B;\Delta, C[x \mapsto t] \longrightarrow D}{\Gamma, B;\Delta, \forall x.C \longrightarrow D}\ \forall_L \overset{\Xi_2}{}}{\Gamma;\Delta, \forall x.C \longrightarrow D}\ cut!$$

$\rightsquigarrow$
$$\dfrac{\Gamma;\varnothing \longrightarrow B \quad \overset{\Xi_1}{} \qquad \dfrac{\Gamma, B;\Delta, C[x \mapsto t] \longrightarrow D}{\Gamma;\Delta, C[x \mapsto t] \longrightarrow D}\ \overset{\Xi_2}{}}{\Gamma;\Delta, \forall x.C \longrightarrow D}\ \Rightarrow_R \quad cut!$$

4. Return to step 1.

Phase II In this phase we restrict our view to proofs without *cut*!. Instances of *cut* in a proof are removed or replaced by cuts on smaller formulas. In order to specify this phase we will need to define two measures used to describe an instance of the *cut* rule.

- δ, the **degree** of a cut rule, which is the degree of the cut formula, defined in the usual way.
- ρ, the **rank** of a cut rule, defined as in Gentzen. That is, as the sum of the left-rank and right-rank where:

  Left-rank is the largest number of consecutive sequents, beginning from the left premise of the *cut* instance, in which the cut formula occurs in the the succedent of the sequent.

  Right-rank is the largest number of consecutive sequents, beginning from the right premise of the *cut* instance, in which the cut formula occurs in the linear context of the sequent.

We further define the **measure** of an instance of the *cut* rule to be the pair (δ, ρ), and the **cut-measure** of a proof tree to be the triple (δ, ρ, ν), where (δ, ρ) is the largest measure (under lexicographic ordering of pairs) of any instance of *cut* in the tree, and ν is the number of instances of *cut* in the tree with that measure.

Given a proof Ξ, the algorithm now proceeds as follows:

1. If Ξ has no occurrences of *cut* then this phase is complete.
2. Select the occurrence of *cut* of highest degree in Ξ. If there is more than one occurrence of this degree, pick the one with the highest rank. If there is more than one of these to choose from, pick any one which qualifies. Now, this occurrence may be assumed to be the root of a subproof of the form:

$$\Xi_0 = \dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma;\Delta_2, B \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\ \mathit{cut}$$

where B is the cut formula.

   We will distinguish between two main cases; when the cut rank, ρ, of this instance of *cut* is 2, and when it is greater.

   (a) If $\rho = 2$ then there are five ways in which this can occur. In the first four the cut rule will be entirely removed at this stage. In the last case it will be replaced by one or two cuts of indeterminate rank, but lower degree. The cases are as follows:

      i. The right premise of the cut is proved by an instance of the $\top_R$ axiom, and the left premise is either proved by an instance of the *identity* axiom on, or is the conclusion of the right rule for the principal operator of, the cut formula. In either case, the entire subproof Ξ_0 is extraneous, and is replaced by an instance of $\top_R$. In its most schematic form, this case is:

$$\dfrac{\overset{\Xi}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{}{\Gamma;\Delta_2, B \longrightarrow \top}\,\top_R}{\Gamma;\Delta_1,\Delta_2 \longrightarrow \top}\,cut \qquad \text{becomes:} \quad \dfrac{}{\Gamma;\Delta_1,\Delta_2 \longrightarrow \top}\,\top_R$$

ii. The left premise of the cut is proved by an instance of the $\top_R$ axiom. But then the cut formula is $\top$. And since the right-rank is also 1, the right premise must also be proved by an instance of $\top_R$ as this is the only rule that could act on a $\top$ in the linear context. Therefore, this is really the same as the last case, and:

$$\dfrac{\dfrac{}{\Gamma;\Delta_1 \longrightarrow \top}\,\top_R \quad \dfrac{}{\Gamma;\Delta_2, \top \longrightarrow \top}\,\top_R}{\Gamma;\Delta_1,\Delta_2 \longrightarrow \top}\,cut \qquad \text{becomes:} \quad \dfrac{}{\Gamma;\Delta_1,\Delta_2 \longrightarrow \top}\,\top_R$$

iii. The right premise is proved by an instance of the *identity* axiom, and the left is as in case 1. Then:

$$\dfrac{\overset{\Xi}{\Gamma;\Delta \longrightarrow C} \quad \overline{\Gamma; C \longrightarrow C}}{\Gamma;\Delta \longrightarrow C}\,cut \qquad \text{becomes:} \quad \Gamma;\Delta \overset{\Xi}{\longrightarrow} C$$

iv. The left premise is proved by an instance of the *identity* axiom, and the right one is either proved by an instance of *identity* on, or is the conclusion of the left-rule for the principal operator of, the cut formula. Again, the entire subproof is extraneous, and:

$$\dfrac{\overline{\Gamma; B \longrightarrow B} \quad \Gamma;\Delta \overset{\Xi}{\longrightarrow} C}{\Gamma;\Delta, B \longrightarrow C}\,cut \qquad \text{becomes:} \quad \Gamma;\Delta \overset{\Xi}{\longrightarrow} C$$

v. The last, multi-part, case is where the left and right premises are the conclusions of the right and left rules, respectively, for the principal operator of the cut formula. In each case the cut rule will be replaced by one or more cuts of lower degree.

A. The cut formula is $B_1 \,\&\, B_2$. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B_1} \quad \overset{\Xi_2}{\Gamma;\Delta_1 \longrightarrow B_2}}{\Gamma;\Delta_1 \longrightarrow B_1 \,\&\, B_2}\,\&_R \quad \dfrac{\overset{\Xi_3}{\Gamma;\Delta_2, B_i \longrightarrow C}}{\Gamma;\Delta_2, B_1 \,\&\, B_2 \longrightarrow C}\,\&_L}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\,cut$$

becomes:

$$\dfrac{\overset{\Xi_i}{\Gamma;\Delta_1 \longrightarrow B_i} \quad \overset{\Xi_3}{\Gamma;\Delta_2, B_i \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\,cut$$

B. The cut formula is $B_1 \multimap B_2$. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1, B_1 \longrightarrow B_2}}{\Gamma;\Delta_1 \longrightarrow B_1 \multimap B_2}\,\multimap_R \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2 \longrightarrow B_1} \quad \overset{\Xi_3}{\Gamma;\Delta_3, B_2 \longrightarrow C}}{\Gamma;\Delta_2,\Delta_3, B_1 \multimap B_2 \longrightarrow C}\,\multimap_L}{\Gamma;\Delta_1,\Delta_2,\Delta_3 \longrightarrow C}\,cut$$

becomes:

$$\dfrac{\dfrac{\overset{\Xi_2}{\Gamma;\Delta_2 \longrightarrow B_1} \quad \dfrac{\overset{\Xi_1}{\Gamma;\Delta_1, B_1 \longrightarrow B_2} \quad \overset{\Xi_3}{\Gamma;\Delta_3, B_2 \longrightarrow C}}{\Gamma;\Delta_1,\Delta_3, B_1 \longrightarrow C}\ cut}{\Gamma;\Delta_1,\Delta_2,\Delta_3 \longrightarrow C}\ cut$$

C. The cut formula is $B_1 \Rightarrow B_2$. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma, B_1;\Delta_1 \longrightarrow B_2}}{\Gamma;\Delta_1 \longrightarrow B_1 \Rightarrow B_2}\Rightarrow_R \quad \dfrac{\overset{\Xi_2}{\Gamma;\varnothing \longrightarrow B_1} \quad \dfrac{\overset{\Xi_3}{\Gamma;\Delta_2, B_2 \longrightarrow C}}{\Gamma;\Delta_2, B_1 \Rightarrow B_2 \longrightarrow C}\Rightarrow_L}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\ cut$$

becomes:

$$\dfrac{\dfrac{\overset{\Xi_2}{\Gamma;\varnothing \longrightarrow B_1} \quad \dfrac{\overset{\Xi_1}{\Gamma, B_1;\Delta_1 \longrightarrow B_2} \quad \overset{\Xi_3}{\Gamma;\Delta_2, B_2 \longrightarrow C}}{\Gamma, B_1;\Delta_1,\Delta_2 \longrightarrow C}\ cut}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\ cut!$$

Now, here we have introduced an instance of *cut*!, which is otherwise barred in this phase. Therefore, we immediately apply the algorithm in phase I to this subproof. This will replace the new instance of *cut*! with possibly many new instances of *cut*. Each of these will be of lower degree than the original instance, however.

D. The cut formula is $\forall x.B$. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B[x \mapsto c]}}{\Gamma;\Delta_1 \longrightarrow \forall x.B}\forall_R \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2, B[x \mapsto t] \longrightarrow C}}{\Gamma;\Delta_2, \forall x.B \longrightarrow C}\forall_L}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\ cut$$

becomes:

$$\dfrac{\overset{\Xi_1^{[c\mapsto t]}}{\Gamma;\Delta_1 \longrightarrow B[x \mapsto t]} \quad \overset{\Xi_2}{\Gamma;\Delta_2, B[x \mapsto t] \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\ cut$$

(b) If $\rho > 2$ then either the left premise is not the conclusion of the right rule for the principal operator of the cut formula, or the right premise is not the conclusion of the matching left rule. In each case we will permute the problem rule application down below the *cut* rule.

i. If the left premise is not the conclusion of a right rule acting on the cut formula, then since the only right rule that could occur in that spot is the one for the principal operator of the cut formula, we need only consider left rules. Whatever left rule does occur in that spot is moved below the cut rule.

A. The left premise is the conclusion of an instance of $\&_L$. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1, E_i \longrightarrow B}}{\Gamma;\Delta_1, E_1 \,\&\, E_2 \longrightarrow B}\,\&_{L_i} \quad \Gamma;\Delta_2, \overset{\Xi_2}{B \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2, E_1 \,\&\, E_2 \longrightarrow C}\; cut$$

becomes:

$$\dfrac{\Gamma;\Delta_1, \overset{\Xi_1}{E_i \longrightarrow B} \quad \Gamma;\Delta_2, \overset{\Xi_2}{B \longrightarrow C}}{\dfrac{\Gamma;\Delta_1,\Delta_2, E_i \longrightarrow C}{\Gamma;\Delta_1,\Delta_2, E_1 \,\&\, E_2 \longrightarrow C}\,\&_{L_i}}\; cut$$

B. The left premise is the conclusion of an instance of $\forall_L$. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1, E[x \mapsto t] \longrightarrow B}}{\Gamma;\Delta_1, \forall x.E \longrightarrow B}\,\forall_L \quad \Gamma;\Delta_2, \overset{\Xi_2}{B \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2, \forall x.E \longrightarrow C}\; cut$$

becomes:

$$\dfrac{\Gamma;\Delta_1, \overset{\Xi_1}{E[x \mapsto t] \longrightarrow B} \quad \Gamma;\Delta_2, \overset{\Xi_2}{B \longrightarrow C}}{\dfrac{\Gamma;\Delta_1,\Delta_2, E[x \mapsto t] \longrightarrow C}{\Gamma;\Delta_1,\Delta_2, \forall x.E \longrightarrow C}\,\forall_L}\; cut$$

C. The left premise is the conclusion of an instance of $\multimap_L$. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow E_1} \quad \Gamma;\Delta_2, \overset{\Xi_2}{E_2 \longrightarrow B}}{\Gamma;\Delta_1,\Delta_2, E_1 \multimap E_2 \longrightarrow B}\,\multimap_L \quad \Gamma;\Delta_3, \overset{\Xi_3}{B \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2,\Delta_3, E_1 \multimap E_2 \longrightarrow C}\; cut$$

becomes:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow E_1} \quad \dfrac{\Gamma;\Delta_2, \overset{\Xi_2}{E_2 \longrightarrow B} \quad \Gamma;\Delta_3, \overset{\Xi_3}{B \longrightarrow C}}{\Gamma;\Delta_2,\Delta_3, E_2 \longrightarrow C}\; cut}{\Gamma;\Delta_1,\Delta_2,\Delta_3, E_1 \multimap E_2 \longrightarrow C}\,\multimap_L$$

D. The left premise is the conclusion of an instance of $\Rightarrow_L$. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow E_1} \quad \Gamma;\Delta_1, \overset{\Xi_2}{E_2 \longrightarrow B}}{\Gamma;\Delta_1, E_1 \multimap E_2 \longrightarrow B}\,\Rightarrow_L \quad \Gamma;\Delta_2, \overset{\Xi_3}{B \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2, E_1 \Rightarrow E_2 \longrightarrow C}\; cut$$

becomes:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\varnothing \longrightarrow E_1} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_1, E_2 \longrightarrow B} \quad \overset{\Xi_3}{\Gamma;\Delta_2, B \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2, E_2 \longrightarrow C}\ cut}{\Gamma;\Delta_1,\Delta_2, E_1 \Rightarrow E_2 \longrightarrow C}\ \Rightarrow_L$$

E. The last case is where the left premise is the conclusion of an instance of *absorb*. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma, E;\Delta_1, E \longrightarrow B}}{\Gamma, E;\Delta_1 \longrightarrow B}\ absorb \quad \overset{\Xi_2}{\Gamma, E;\Delta_2, B \longrightarrow C}}{\Gamma, E;\Delta_1,\Delta_2 \longrightarrow C}\ cut$$

becomes:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma, E;\Delta_1, E \longrightarrow B} \quad \overset{\Xi_2}{\Gamma, E;\Delta_2, B \longrightarrow C}}{\Gamma, E;\Delta_1,\Delta_2, E \longrightarrow C}\ cut}{\Gamma, E;\Delta_1,\Delta_2 \longrightarrow C}\ absorb$$

ii. If the right premise is not the conclusion of an instance of the left rule for the principal operator of the cut formula, then in this case, both left and right rules could occur in this position. Whatever rule does occur in that spot is moved below the cut rule.

A. The right premise is the conclusion of an instance of $\&_L$. Then:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2, E_i, B \longrightarrow C}}{\Gamma;\Delta_2, E_1 \,\&\, E_2, B \longrightarrow C}\ \&_{L_i}}{\Gamma;\Delta_1,\Delta_2, E_1 \,\&\, E_2 \longrightarrow C}\ cut$$

becomes:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma;\Delta_2, E_i, B \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2, E_i \longrightarrow C}\ cut}{\Gamma;\Delta_1,\Delta_2, E_1 \,\&\, E_2 \longrightarrow C}\ \&_{L_i}$$

B. The right premise is the conclusion of an instance of $\&_R$. Then:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2, B \longrightarrow E_1} \quad \overset{\Xi_3}{\Gamma;\Delta_2, B \longrightarrow E_2}}{\Gamma;\Delta_2, B \longrightarrow E_1 \,\&\, E_2}\ \&_R}{\Gamma;\Delta_1,\Delta_2 \longrightarrow E_1 \,\&\, E_2}\ cut$$

becomes:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma;\Delta_2, B \longrightarrow E_1}}{\Gamma;\Delta_1,\Delta_2 \longrightarrow E_1}\ cut \quad \dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_3}{\Gamma;\Delta_2, B \longrightarrow E_2}}{\Gamma;\Delta_1,\Delta_2 \longrightarrow E_2}\ cut}{\Gamma;\Delta_1,\Delta_2 \longrightarrow E_1 \,\&\, E_2}\ \&_R$$

Here one cut rule is replaced with two, but each is of lower rank than the original.

C. The right premise is the conclusion of an instance of $\forall_L$. Then:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2, E[x \mapsto t], B \longrightarrow C}}{\Gamma;\Delta_2, \forall x.E, B \longrightarrow C}\ \forall_L}{\Gamma;\Delta_1,\Delta_2, \forall x.E \longrightarrow C}\ cut$$

becomes:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma;\Delta_2, E[x \mapsto t], B \longrightarrow C}}{\dfrac{\Gamma;\Delta_1,\Delta_2, E[x \mapsto t] \longrightarrow C}{\Gamma;\Delta_1,\Delta_2, \forall x.E \longrightarrow C}\ \forall_L}\ cut$$

D. The right premise is the conclusion of an instance of $\forall_R$. Then:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2, B \longrightarrow E[x \mapsto c]}}{\Gamma;\Delta_2, B \longrightarrow \forall x.E}\ \forall_R}{\Gamma;\Delta_1,\Delta_2 \longrightarrow \forall x.E}\ cut$$

becomes:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma;\Delta_2, B \longrightarrow E[x \mapsto c]}}{\dfrac{\Gamma;\Delta_1,\Delta_2 \longrightarrow E[x \mapsto c]}{\Gamma;\Delta_1,\Delta_2 \longrightarrow \forall x.E}\ \forall_R}\ cut$$

E. The right premise is the conclusion of an instance of $\multimap_L$. This can occur in two ways, depending on which premise of the $\multimap_L$ rule contains the cut formula. Either:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2, B \longrightarrow E_1} \quad \overset{\Xi_3}{\Gamma;\Delta_3, E_2 \longrightarrow C}}{\Gamma;\Delta_2,\Delta_3, E_1 \multimap E_2, B \longrightarrow C}\ \multimap_L}{\Gamma;\Delta_1,\Delta_2,\Delta_3, E_1 \multimap E_2 \longrightarrow C}\ cut$$

becomes:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma;\Delta_2, B \longrightarrow E_1}}{\Gamma;\Delta_1,\Delta_2 \longrightarrow E_1}\ cut \quad \overset{\Xi_3}{\Gamma;\Delta_3, E_2 \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2,\Delta_3, E_1 \multimap E_2 \longrightarrow C}\ \multimap_L$$

or,

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2 \longrightarrow E_1} \quad \overset{\Xi_3}{\Gamma;\Delta_3, E_2, B \longrightarrow C}}{\Gamma;\Delta_2,\Delta_3, E_1 \multimap E_2, B \longrightarrow C}\ \multimap_L}{\Gamma;\Delta_1,\Delta_2,\Delta_3, E_1 \multimap E_2 \longrightarrow C}\ cut$$

becomes:

$$\dfrac{\overset{\Xi_2}{\Gamma;\Delta_2 \longrightarrow E_1} \quad \dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_3}{\Gamma;\Delta_3, E_2, B \longrightarrow C}}{\Gamma';\Delta_1,\Delta_3, E_2 \longrightarrow C}\ cut}{\Gamma;\Delta_1,\Delta_2,\Delta_3, E_1 \multimap E_2 \longrightarrow C}\ \multimap_L$$

F. The right premise is the conclusion of an instance of $\multimap_R$. Then:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta_2, B, E_1 \longrightarrow E_2}}{\Gamma;\Delta_2, B \longrightarrow E_1 \multimap E_2}\ \multimap_R}{\Gamma;\Delta_1,\Delta_2 \longrightarrow E_1 \multimap E_2}\ cut$$

becomes:

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma;\Delta_2, B, E_1 \longrightarrow E_2}}{\dfrac{\Gamma;\Delta_1,\Delta_2, E_1 \longrightarrow E_2}{\Gamma;\Delta_1,\Delta_2 \longrightarrow E_1 \multimap E_2}\ \multimap_R}\ cut$$

G. The right premise is the conclusion of an instance of $\Rightarrow_L$. This can only occur in one way due to the restricted form of the left premise of the $\Rightarrow_L$ rule.

$$\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma;\varnothing \longrightarrow E_1} \quad \overset{\Xi_3}{\Gamma;\Delta_2, E_2, B \longrightarrow C}}{\Gamma;\Delta_2, E_1 \Rightarrow E_2, B \longrightarrow C}\ \Rightarrow_L}{\Gamma;\Delta_1,\Delta_2, E_1 \Rightarrow E_2 \longrightarrow C}\ cut$$

becomes:

$$\dfrac{\overset{\Xi_2}{\Gamma;\varnothing \longrightarrow E_1} \quad \dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \overset{\Xi_3}{\Gamma;\Delta_2, E_2, B \longrightarrow C}}{\Gamma;\Delta_1,\Delta_2, E_2 \longrightarrow C}\ cut}{\Gamma;\Delta_1,\Delta_2,\Delta_3, E_1 \Rightarrow E_2 \longrightarrow C}\ \Rightarrow_L$$

H. The right premise is the conclusion of an instance of $\Rightarrow_R$. Then:

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma, E_1;\Delta_2, B \longrightarrow E_2}}{\Gamma;\Delta_2, B \longrightarrow E_1 \Rightarrow E_2}\Rightarrow_R}{\Gamma;\Delta_1,\Delta_2 \longrightarrow E_1 \Rightarrow E_2}\,cut$$

becomes:

$$\dfrac{\dfrac{\overset{\Xi_1{}^{+\{E_1\}}}{\Gamma, E_1;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma, E_1;\Delta_2, B \longrightarrow E_2}}{\Gamma, E_1;\Delta_1,\Delta_2 \longrightarrow E_2}\,cut}{\Gamma;\Delta_1,\Delta_2 \longrightarrow E_1 \Rightarrow E_2}\Rightarrow_R$$

which is a proof, by Proposition $B.2$.

I. The last case is where the right premise is the conclusion of an instance of *absorb*. Then:

$$\dfrac{\overset{\Xi_1}{\Gamma, E;\Delta_1 \longrightarrow B} \quad \dfrac{\overset{\Xi_2}{\Gamma, E;\Delta_2, B, E \longrightarrow C}}{\Gamma, E;\Delta_2, B \longrightarrow C}\,absorb}{\Gamma, E;\Delta_1,\Delta_2 \longrightarrow C}\,cut$$

becomes:

$$\dfrac{\overset{\Xi_1}{\Gamma, E;\Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma, E;\Delta_2, B, E \longrightarrow C}}{\dfrac{\Gamma, E;\Delta_1,\Delta_2, E \longrightarrow C}{\Gamma, E;\Delta_1,\Delta_2 \longrightarrow C}\,absorb}\,cut$$

3. Return to step 1.

The proof of the admissibility of the two cut rules is now reduced to a proof of the termination of this algorithm. The termination of phase I is apparent, since each iteration either removes an instance of *cut*!, replaces an instance with one with a smaller proof of its right premise, or replaces an instance with two instances each with a smaller proof of its right premise. Thus all instances of *cut*! can be removed from a proof, and that rule is therefore admissable. The termination of phase II is proved by induction on the cut-measure of the subject proof tree.

Base Case: The base case is a proof tree with cut-measure $(\delta = 1, \rho = 2, \nu = n)$ for some n. In this case step $2a$ is invoked, but only the first four cases (in restricted form) can hold since the degree is 1. Each of these cases results in the elimination of the instance of *cut* under consideration. Two cases now hold:

- If this was the only instance of *cut* in the proof —that is, if the cut-measure of the tree was $(1, 2, 1)$— then the proof is now cut-free and the algorithm terminates.
- If there were other instances of *cut* in the proof then they must be of the same measure (since otherwise the cut-measure of the proof would have been larger to

begin with), and the cut-measure of the proof has been reduced to $(1, 2, n-1)$. The algorithm will loop, and the same situation will hold. This process continues until there are no more instances of *cut* and the algorithm terminates.

Thus termination for proofs with cut-measure $(\delta = 1, \rho = 2, \nu = n)$ has been shown.

Induction Hypothesis: All instances of *cut* can be removed from proof trees with cut-measure less than $(\delta = d, \rho = r, \nu = n)$.

Inductive Step: Assume Ξ is a proof of cut-measure $(\delta = d, \rho = r, \nu = n)$.

- If $\rho = 2$ then step $2a$ is invoked. In the first four cases the instance of *cut* under consideration is removed (as in the base case). If there was more than one instance with that measure then the cut-measure of the tree drops to $(\delta = d, \rho = 2, \nu = n-1)$. If this was the only such instance (ie. $\nu = 1$) then the cut-measure of the proof is reduced to $(\delta < d, \rho = r', \nu = n')$, which is lower in the lexicographic ordering regardless of the values of r' and n'.

  If case $2(a)\nu$ is invoked then in the first and fourth subcases the instance of *cut* under consideration is replaced with one of lower degree and therefore, as above, either ν will decrease (with the other elements of the measure remaining fixed) or δ will decrease. In the second and third subcases the *cut* instance is replaced by two, or possibly many, instances. But again all the new instances are of lower degree, so the cut-measure of the proof goes down as above.

  Therefore, if $\rho = 2$ the algorithm will produce, in one step, a proof tree with lower cut-measure. By the induction hypothesis the algorithm will terminate on that new tree.

- If $\rho > 2$ then step $2b$ is invoked. In all but one of the subcases of this step the instance of *cut* being acted upon is replaced by one of the same degree but lower rank. Some of the cases reduce the rank by 1, others by more, but this is not important. Therefore, as before, the cut-measure of is reduced to either $(\delta = d, \rho < r, \nu = n')$ or $(\delta = d, \rho = r, \nu = n-1)$. The only case to be careful of is step $2(b)iiB$ in which the *cut* instance is replaced by two new instances. Each of the new instances is of lower rank than the original, however, so this is no problem.

  Therefore, if $\rho > 2$ the algorithm will produce, in one step, a proof tree with lower cut-measure. By the induction hypothesis the algorithm will terminate on that new tree.

Therefore the termination of Phase II on any input proof tree with no instances of *cut*! has been shown, and the cut rules of the system are admissible. ■

Appendix E

A Proof of Uniform-Proof-Completeness for the System $\mathcal{L}$

Proposition 4.2 Let B be a goal formula, and let Γ and Δ be multisets of definite formulae. Then the sequent $\Gamma;\Delta \longrightarrow B$ has a proof in $\mathcal{L}$ if and only if it has a uniform proof in $\mathcal{L}$.

Proof. The forward direction —that if the sequent has a uniform cut-free proof, it has a proof— is immediate, since a uniform proof in $\mathcal{L}$ is certainly an $\mathcal{L}$ proof. To show the reverse direction —that any $\mathcal{L}$ provable sequent has a uniform proof— is shown by giving a procedure, inductive over the structure of the non-uniform proof, which converts it to a uniform proof.
Base Cases: Since we assume that the original proof has only atomic axioms, it is immediate that the initial sequents of the proof are already uniform.
Inductive Hypothesis: Assume that the proofs Ξ_i are uniform cut-free proofs with atomic axioms.
Inductive Step: In each step we will select the highest (furthest from the root of the proof) non-uniform rule pair — one in which a left rule whose conclusion has a non-atomic consequent is preceeded by a right rule — and permute the rules according to the schema below. In each case we will see that the inductive measure decreases.

Permuting $\&_L$ through right rules:

$$\frac{\&_R}{\&_L}: \quad \dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta,B_i\longrightarrow C_1}\quad\overset{\Xi_2}{\Gamma;\Delta,B_i\longrightarrow C_2}}{\Gamma;\Delta,B_i\longrightarrow C_1\,\&\,C_2}\,\&_R}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow C_1\,\&\,C_2}\,\&_L \quad\rightsquigarrow\quad \dfrac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta,B_i\longrightarrow C_1}}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow C_1}\,\&_L\quad\dfrac{\overset{\Xi_2}{\Gamma;\Delta,B_i\longrightarrow C_2}}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow C_2}\,\&_L}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow C_1\,\&\,C_2}\,\&_R$$

$$\frac{\multimap_R}{\&_L}: \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma;\Delta,B_i,C_1\longrightarrow C_2}}{\Gamma;\Delta,B_i\longrightarrow C_1\multimap C_2}\,\multimap_R}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow C_1\multimap C_2}\,\&_L \quad\rightsquigarrow\quad \dfrac{\dfrac{\overset{\Xi}{\Gamma;\Delta,B_i,C_1\longrightarrow C_2}}{\Gamma;\Delta,B_1\,\&\,B_2,C_1\longrightarrow C_2}\,\&_L}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow C_1\multimap C_2}\,\multimap_R$$

$$\frac{\Rightarrow_R}{\&_L}: \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma,C_1;\Delta,B_i\longrightarrow C_2}}{\Gamma;\Delta,B_i\longrightarrow C_1\Rightarrow C_2}\,\Rightarrow_R}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow C_1\Rightarrow C_2}\,\&_L \quad\rightsquigarrow\quad \dfrac{\dfrac{\overset{\Xi}{\Gamma,C_1;\Delta,B_i\longrightarrow C_2}}{\Gamma,C_1;\Delta,B_1\,\&\,B_2\longrightarrow C_2}\,\&_L}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow C_1\Rightarrow C_2}\,\Rightarrow_R$$

$$\frac{\forall_R}{\&_L}: \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma;\Delta,B_i\longrightarrow C[x\mapsto c]}}{\Gamma;\Delta,B_i\longrightarrow \forall x.C}\,\forall_R}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow \forall x.C}\,\&_L \quad\rightsquigarrow\quad \dfrac{\dfrac{\overset{\Xi}{\Gamma,C_1;\Delta,B_i\longrightarrow C[x\mapsto c]}}{\Gamma,C_1;\Delta,B_1\,\&\,B_2\longrightarrow C[x\mapsto c]}\,\&_L}{\Gamma;\Delta,B_1\,\&\,B_2\longrightarrow \forall x.C}\,\forall_R$$

Permuting $\multimap_L$ through right rules:

$$\Xi_a \quad = \quad \overset{\Xi_1}{\Gamma; \Delta_1, \longrightarrow B_1}$$

$$\Xi_b \quad = \quad \frac{\overset{\Xi_2}{\Gamma; \Delta_2, B_2 \longrightarrow C1} \quad \overset{\Xi_3}{\Gamma; \Delta_2, B_2 \longrightarrow C2}}{\Gamma; \Delta_2, B_2 \longrightarrow C_1 \,\&\, C_2}\ \&_R$$

$$\Xi_c \quad = \quad \frac{\overset{\Xi_1}{\Gamma; \Delta_1, \longrightarrow B_1} \quad \overset{\Xi_2}{\Gamma; \Delta_2, B_2 \longrightarrow C_1}}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_1}\ \multimap_L$$

$$\Xi_d \quad = \quad \frac{\overset{\Xi_1}{\Gamma; \Delta_1, \longrightarrow B_1} \quad \overset{\Xi_3}{\Gamma; \Delta_2, B_2 \longrightarrow C_2}}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_2}\ \multimap_L$$

$\dfrac{\&_R}{\multimap_L}$:
$$\frac{\dfrac{\Xi_a \qquad \Xi_b}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_1 \,\&\, C_2}\ \multimap_L}{}$$
$\rightsquigarrow$
$$\frac{\Xi_c \qquad \Xi_d}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_1 \,\&\, C_2}\ \&_R$$

$\dfrac{\multimap_R}{\multimap_L}$:
$$\frac{\dfrac{\Xi_1 \qquad \dfrac{\overset{\Xi_2}{\Gamma; \Delta_2, B_2, C_1 \longrightarrow C_2}}{\Gamma; \Delta_2, B_2 \longrightarrow C_1 \multimap C_2}\ \multimap_R}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_1 \multimap C_2}\ \multimap_L}{}$$
$\rightsquigarrow$
$$\frac{\dfrac{\Xi_1 \qquad \overset{\Xi_2}{\Gamma; \Delta_2, B_2, C_1 \longrightarrow C_2}}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2, C_1 \longrightarrow C_2}\ \multimap_L}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_1 \multimap C_2}\ \multimap_R$$

$\dfrac{\Rightarrow_R}{\multimap_L}$:
$$\frac{\dfrac{\Xi_1 \qquad \dfrac{\overset{\Xi_2}{\Gamma, C_1; \Delta_2, B_2 \longrightarrow C_2}}{\Gamma; \Delta_2, B_2 \longrightarrow C_1 \Rightarrow C_2}\ \Rightarrow_R}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_1 \Rightarrow C_2}\ \multimap_L}{}$$
$\rightsquigarrow$
$$\frac{\dfrac{\overset{\Xi_1^{+C_1}}{\Gamma, C_1; \Delta_1 \longrightarrow B_1} \qquad \overset{\Xi_2}{\Gamma, C_1; \Delta_2, B_2 \longrightarrow C_2}}{\Gamma, C_1; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_2}\ \multimap_L}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_1 \Rightarrow C_2}\ \Rightarrow_R$$

$\dfrac{\forall_R}{\multimap_L}$:
$$\frac{\dfrac{\Xi_1 \qquad \dfrac{\overset{\Xi_2}{\Gamma; \Delta_2, B_2 \longrightarrow C_1[x \mapsto c]}}{\Gamma; \Delta_2, B_2 \longrightarrow \forall x.C_1}\ \forall_R}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow \forall x.C_1}\ \multimap_L}{}$$
$\rightsquigarrow$
$$\frac{\dfrac{\Xi_1 \qquad \overset{\Xi_2}{\Gamma; \Delta_2, B_2 \longrightarrow C_1[x \mapsto c]}}{\Gamma, C_1; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow C_1[x \mapsto c]}\ \multimap_L}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \longrightarrow \forall x.C_1}\ \forall_R$$

Permuting $\Rightarrow_L$ through right rules:

$$\Xi_a \quad = \quad \overset{\Xi_1}{\Gamma; \varnothing, \longrightarrow B_1}$$

$$\Xi_b \quad = \quad \frac{\overset{\Xi_2}{\Gamma; \Delta, B_2 \longrightarrow C_1} \quad \overset{\Xi_3}{\Gamma; \Delta, B_2 \longrightarrow C_2}}{\Gamma; \Delta, B_2 \longrightarrow C1 \,\&\, C_2}\ \&_R$$

$$\Xi_c \quad = \quad \frac{\overset{\Xi_1}{\Gamma; \varnothing \longrightarrow B_1} \quad \overset{\Xi_2}{\Gamma; \Delta, B_2 \longrightarrow C_1}}{\Gamma; \Delta, B_1 \Rightarrow B_2 \longrightarrow C_1}\ \Rightarrow_L$$

$$\Xi_d \quad = \quad \frac{\overset{\Xi_1}{\Gamma; \varnothing, \longrightarrow B_1} \quad \overset{\Xi_3}{\Gamma; \Delta, B_2 \longrightarrow C_2}}{\Gamma; \Delta, B_1 \Rightarrow B_2 \longrightarrow C_2}\ \Rightarrow_L$$

$\dfrac{\&_R}{\Rightarrow_L}$:
$$\frac{\overset{\Xi_a}{\cdots} \quad \overset{\Xi_b}{\cdots}}{\Gamma; \Delta, B_1 \Rightarrow B_2 \longrightarrow C_1 \,\&\, C_2}\ \Rightarrow_L \quad \rightsquigarrow \quad \frac{\overset{\Xi_c}{\cdots} \quad \overset{\Xi_d}{\cdots}}{\Gamma; \Delta, B_1 \multimap B_2 \longrightarrow C_1 \,\&\, C_2}\ \&_R$$

$\dfrac{\multimap_R}{\Rightarrow_L}$:
$$\frac{\overset{\Xi_1}{\Gamma; \varnothing \longrightarrow B_1} \quad \dfrac{\overset{\Xi_2}{\Gamma; \Delta, B_2, C_1 \longrightarrow C_2}}{\Gamma; \Delta, B_2 \longrightarrow C_1 \multimap C_2}\ \multimap_R}{\Gamma; \Delta, B_1 \Rightarrow B_2 \longrightarrow C_1 \multimap C_2}\ \Rightarrow_L \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\Xi_1}{\Gamma; \varnothing \longrightarrow B_1} \quad \overset{\Xi_2}{\Gamma; \Delta, B_2, C_1 \longrightarrow C_2}}{\Gamma; \Delta, B_1 \Rightarrow B_2, C_1 \longrightarrow C_2}\ \Rightarrow_L}{\Gamma; \Delta, B_1 \Rightarrow B_2 \longrightarrow C_1 \multimap C_2}\ \multimap_R$$

$\dfrac{\Rightarrow_R}{\Rightarrow_L}$:
$$\frac{\overset{\Xi_1}{\Gamma; \varnothing \longrightarrow B_1} \quad \dfrac{\overset{\Xi_2}{\Gamma, C_1; \Delta, B_2 \longrightarrow C_2}}{\Gamma; \Delta, B_2 \longrightarrow C_1 \Rightarrow C_2}\ \Rightarrow_R}{\Gamma; \Delta, B_1 \Rightarrow B_2 \longrightarrow C_1 \Rightarrow C_2}\ \Rightarrow_L \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\Xi_1^{+C_1}}{\Gamma, C_1; \varnothing \longrightarrow B_1} \quad \overset{\Xi_2}{\Gamma, C_1; \Delta, B_2 \longrightarrow C_2}}{\Gamma, C_1; \Delta, B_1 \Rightarrow B_2 \longrightarrow C_2}\ \Rightarrow_L}{\Gamma; \Delta, B_1 \Rightarrow B_2 \longrightarrow C_1 \Rightarrow C_2}\ \Rightarrow_R$$

$\dfrac{\forall_R}{\Rightarrow_L}$:
$$\frac{\overset{\Xi_1}{\Gamma; \varnothing \longrightarrow B_1} \quad \dfrac{\overset{\Xi_2}{\Gamma; \Delta, B_2 \longrightarrow C_1[x \mapsto c]}}{\Gamma; \Delta, B_2 \longrightarrow \forall x.C_1}\ \forall_R}{\Gamma; \Delta, B_1 \multimap B_2 \longrightarrow \forall x.C_1}\ \Rightarrow_L \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\Xi_1}{\Gamma; \varnothing \longrightarrow B_1} \quad \overset{\Xi_2}{\Gamma; \Delta, B_2 \longrightarrow C_1[x \mapsto c]}}{\Gamma, C_1; \Delta, B_1 \multimap B_2 \longrightarrow C_1[x \mapsto c]}\ \Rightarrow_L}{\Gamma; \Delta, B_1 \multimap B_2 \longrightarrow \forall x.C_1}\ \forall_R$$

Permuting $\forall_L$ through right rules:

$$\frac{\&_R}{\forall_L}: \quad \frac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta,B[x\mapsto t]\longrightarrow C_1} \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta,B[x\mapsto t]\longrightarrow C_2}}{\Gamma;\Delta,\forall x.B\longrightarrow C_2}\,\forall_L}{\Gamma;\Delta,\forall x.B\longrightarrow C_1\,\&\,C_2}\,\&_R \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\Xi_1}{\Gamma;\Delta,B[x\mapsto t]\longrightarrow C_1}}{\Gamma;\Delta,\forall x.B\longrightarrow C_1}\,\forall_L \quad \dfrac{\overset{\Xi_2}{\Gamma;\Delta,B[x\mapsto t]\longrightarrow C_2}}{\Gamma;\Delta,\forall x.B\longrightarrow C_2}\,\forall_L}{\Gamma;\Delta,\forall x.B\longrightarrow C_1\,\&\,C_2}\,\&_R$$

$$\frac{\multimap_R}{\forall_L}: \quad \frac{\dfrac{\overset{\Xi}{\Gamma;\Delta,B[x\mapsto t],C_1\longrightarrow C_2}}{\Gamma;\Delta,B[x\mapsto t]\longrightarrow C_1\multimap C_2}\,\multimap_R}{\Gamma;\Delta,\forall x.B\longrightarrow C_1\multimap C_2}\,\forall_L \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\Xi}{\Gamma;\Delta,B[x\mapsto t],C_1\longrightarrow C_2}}{\Gamma;\Delta,\forall x.B,C_1\longrightarrow C_2}\,\forall_L}{\Gamma;\Delta,\forall x.B\longrightarrow C_1\multimap C_2}\,\multimap_R$$

$$\frac{\Rightarrow_R}{\forall_L}: \quad \frac{\dfrac{\overset{\Xi}{\Gamma,C_1;\Delta,B[x\mapsto t]\longrightarrow C_2}}{\Gamma;\Delta,B[x\mapsto t]\longrightarrow C_1\Rightarrow C_2}\,\Rightarrow_R}{\Gamma;\Delta,\forall x.B\longrightarrow C_1\Rightarrow C_2}\,\forall_L \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\Xi}{\Gamma,C_1;\Delta,B[x\mapsto t]\longrightarrow C_2}}{\Gamma,C_1;\Delta,\forall x.B\longrightarrow C_2}\,\forall_L}{\Gamma;\Delta,\forall x.B\longrightarrow C_1\Rightarrow C_2}\,\Rightarrow_R$$

$$\frac{\forall R^1}{\forall_L}: \quad \frac{\dfrac{\overset{\Xi}{\Gamma;\Delta,B[x\mapsto t]\longrightarrow C[x\mapsto c]}}{\Gamma;\Delta,B[x\mapsto t]\longrightarrow \forall x.C}\,\forall_R}{\Gamma;\Delta,\forall x.B\longrightarrow \forall x.C}\,\forall_L \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\Xi}{\Gamma;\Delta,B[x\mapsto t]\longrightarrow C[x\mapsto c]}}{\Gamma;\Delta,\forall x.B\longrightarrow C[x\mapsto c]}\,\forall_L}{\Gamma;\Delta,\forall x.B\longrightarrow \forall x.C}\,\forall_R$$

Permuting *absorb* through right rules:

$$\frac{\&_R}{absorb}: \quad \frac{\dfrac{\overset{\Xi_1}{\Gamma,B;\Delta,B\longrightarrow C_1} \quad \dfrac{\overset{\Xi_2}{\Gamma,B;\Delta,B\longrightarrow C_2}}{\Gamma,B;\Delta,B\longrightarrow C_1\,\&\,C_2}\,absorb}{\Gamma,B;\Delta\longrightarrow C_1\,\&\,C_2}\,\&_R \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\Xi_1}{\Gamma,B;\Delta,B\longrightarrow C_1}}{\Gamma,B;\Delta\longrightarrow C_1}\,absorb \quad \dfrac{\overset{\Xi_2}{\Gamma,B;\Delta,B\longrightarrow C_2}}{\Gamma,B;\Delta\longrightarrow C_2}\,absorb}{\Gamma,B;\Delta\longrightarrow C_1\,\&\,C_2}\,\&_R$$

[1] Any possible complication due to eigenconstants here is avoided by the rectification of the proofs.

$$\frac{\multimap_R}{absorb}: \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma, B; \Delta, B, C_1 \longrightarrow C_2}}{\Gamma, B; \Delta, B \longrightarrow C_1 \multimap C_2}\ \multimap_R}{\Gamma, B; \Delta \longrightarrow C_1 \multimap C_2}\ absorb \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma, B; \Delta, B, C_1 \longrightarrow C_2}}{\Gamma, B; \Delta, C_1 \longrightarrow C_2}\ absorb}{\Gamma, B; \Delta \longrightarrow C_1 \multimap C_2}\ \multimap_R$$

$$\frac{\Rightarrow_R}{absorb}: \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma, B, C_1; \Delta, B \longrightarrow C_2}}{\Gamma, B; \Delta, B \longrightarrow C_1 \Rightarrow C_2}\ \Rightarrow_R}{\Gamma, B; \Delta \longrightarrow C_1 \Rightarrow C_2}\ absorb \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma, B, C_1; \Delta, B \longrightarrow C_2}}{\Gamma, B, C_1; \Delta \longrightarrow C_2}\ absorb}{\Gamma, B; \Delta \longrightarrow C_1 \Rightarrow C_2}\ \Rightarrow_R$$

$$\frac{\forall_R}{absorb}: \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma, B; \Delta, B \longrightarrow C[x \mapsto c]}}{\Gamma, B; \Delta, B \longrightarrow \forall x.C}\ \forall_R}{\Gamma, B; \Delta \longrightarrow \forall x.C}\ absorb \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma, B; \Delta, B \longrightarrow C[x \mapsto c]}}{\Gamma, B; \Delta \longrightarrow C[x \mapsto c]}\ absorb}{\Gamma, B; \Delta \longrightarrow \forall x.C}\ \forall_R$$

■

Appendix F

A Proof of the Soundness of $\mathcal{O}$ Relative to $\mathcal{L}$

Proposition 8.2 The system $\mathcal{O}$ is sound for $\mathcal{L}$ augmented with the rule:

$$\frac{\Gamma;\Delta \longrightarrow C}{\Gamma;\Delta,\mathbf{1} \longrightarrow C}\ \mathbf{1}_L$$

in the sense that if C° is the result of recursively replacing all instances of $(D \stackrel{R}{\multimap} E)$ and $(D \stackrel{A}{\multimap} E)$ in C with $(D \Rightarrow (D \multimap E))$ and $((D \,\&\, \mathbf{1}) \multimap E)$, respectively, and if $\Gamma^\circ = \{C^\circ | C \in \Gamma\}$ and $\Gamma^{\&\mathbf{1}} = \{(C \,\&\, \mathbf{1}) | C \in \Gamma\}$, then, if the $\mathcal{O}$-sequent $\Gamma;\Upsilon;\Psi;\Delta \longrightarrow C$ is $\mathcal{O}$-provable, then $\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow C^\circ$ is provable in the augmented $\mathcal{L}$.

Proof. The proof is in the form of an algorithm, inductive on the structure $\mathcal{O}$ proofs, which converts such proofs to $\mathcal{L}$ proofs (of appropriately converted versions of the same endsequents). In the converted proofs, a rule marked $\&\mathbf{1}_L^*$ refers to applying the $\&_{L_2}$ rule below the $\mathbf{1}_L$ rule sufficiently many times to remove some group of $(B \,\&\, \mathbf{1})$ pairs from the bounded context.

Base Cases:

$$\textit{identity}:\ \frac{}{\Gamma;\varnothing;\Psi;B \longrightarrow B} \quad\rightsquigarrow\quad \frac{\dfrac{}{\Gamma^\circ;B^\circ \longrightarrow B^\circ}}{\Gamma^\circ;(\Psi^\circ)^{\&\mathbf{1}},B^\circ \longrightarrow B^\circ}\ \&\mathbf{1}_L^*$$

$$\top_R:\ \frac{}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow \top}\ \top_R \quad\rightsquigarrow\quad \frac{}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow \top}$$

$$\mathbf{1}_R:\ \frac{}{\Gamma;\varnothing;\Psi;\varnothing \longrightarrow \mathbf{1}_R} \quad\rightsquigarrow\quad \frac{\dfrac{}{\Gamma^\circ;\varnothing \longrightarrow \mathbf{1}}}{\Gamma^\circ;(\Psi^\circ)^{\&\mathbf{1}},\longrightarrow \mathbf{1}}\ \&\mathbf{1}_L^*$$

Inductive Cases: In each case assume that if Ξ is an $\mathcal{O}$ proof, that Ξ° is an $\mathcal{L}$ proof of the appropriately converted version of the same endsequent.

$$abs_I\colon \frac{\Gamma, B; \Upsilon; \Psi; \overset{\Xi}{\Delta}, B \longrightarrow C}{\Gamma, B; \Upsilon; \Psi; \Delta \longrightarrow C}\ abs_I \quad \rightsquigarrow \quad \frac{\overset{\Xi^\circ}{\Gamma^\circ, B^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ, B^\circ \longrightarrow C^\circ}}{\Gamma^\circ, B^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ \longrightarrow C^\circ}\ abs$$

$$abs_R\colon \frac{\Gamma, B; \Upsilon; \Psi; \overset{\Xi}{\Delta}, B \longrightarrow C}{\Gamma; \Upsilon, B; \Psi; \Delta \longrightarrow C}\ abs_R \quad \rightsquigarrow \quad \overset{\Xi^\circ}{\Gamma^\circ, \Upsilon^\circ, B^\circ; \Upsilon^\circ, B^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ \longrightarrow C^\circ}$$

$$abs_A\colon \frac{\Gamma; \Upsilon; \Psi; \overset{\Xi}{\Delta}, B \longrightarrow C}{\Gamma; \Upsilon; \Psi, B; \Delta \longrightarrow C}\ abs_A \quad \rightsquigarrow \quad \frac{\overset{\Xi^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, B^\circ, \Delta^\circ \longrightarrow C^\circ}}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, (B^\circ)^{\&}\mathbf{1}\Delta^\circ \longrightarrow C^\circ}\ \&_{L_1}$$

$$\multimap_L\colon \frac{\overset{\Xi_1}{\Gamma; \Upsilon; \Psi_1; \Delta_1 \longrightarrow B} \quad \overset{\Xi_2}{\Gamma; \Upsilon; \Psi_2; \Delta_2, C \longrightarrow E}}{\Gamma; \Upsilon; \Psi_1, \Psi_2; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap_L \quad \rightsquigarrow$$

$$\frac{\dfrac{\overset{\Xi_1^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi_1^\circ)^{\&}\mathbf{1}, \Delta_1^\circ \longrightarrow B^\circ} \quad \overset{\Xi_2^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi_2^\circ)^{\&}\mathbf{1}, \Delta_2^\circ, C^\circ \longrightarrow E^\circ}}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, \Upsilon^\circ, (\Psi_1^\circ)^{\&}\mathbf{1}, (\Psi_2^\circ)^{\&}\mathbf{1}, \Delta_1^\circ, \Delta_2^\circ, B^\circ \multimap C^\circ \longrightarrow E^\circ}\ \multimap_L}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi_1^\circ)^{\&}\mathbf{1}, (\Psi_2^\circ)^{\&}\mathbf{1}, \Delta_1^\circ, \Delta_2^\circ, B^\circ \multimap C^\circ \longrightarrow E^\circ}\ abs^*$$

$$\multimap_R\colon \frac{\Gamma; \Upsilon; \Psi; \overset{\Xi}{\Delta}, B \longrightarrow C}{\Gamma; \Upsilon; \Psi; \Delta \longrightarrow B \multimap C}\ \multimap_R \quad \rightsquigarrow \quad \frac{\overset{\Xi^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ, B^\circ \longrightarrow C^\circ}}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ \longrightarrow B^\circ \multimap C^\circ}\ \multimap_R$$

$$\Rightarrow_L\colon \frac{\overset{\Xi_1}{\Gamma; \Upsilon; \varnothing; \varnothing \longrightarrow B} \quad \overset{\Xi_2}{\Gamma; \Upsilon; \Psi; \Delta, C \longrightarrow E}}{\Gamma; \Upsilon; \Psi; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow_L \quad \rightsquigarrow$$

$$\frac{\overset{\Xi_1^{\circ +\Upsilon^\circ}}{\Gamma^\circ, \Upsilon^\circ; \varnothing \longrightarrow B^\circ} \quad \overset{\Xi_2^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ, C^\circ \longrightarrow E^\circ}}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ, B^\circ \Rightarrow C^\circ \longrightarrow E^\circ}\ \Rightarrow_L$$

$$\Rightarrow_R\colon \frac{\Gamma, B; \Upsilon; \Psi; \overset{\Xi}{\Delta} \longrightarrow C}{\Gamma; \Upsilon; \Psi; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow_R \quad \rightsquigarrow \quad \frac{\overset{\Xi^\circ}{\Gamma^\circ, B^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ \longrightarrow C^\circ}}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ \longrightarrow B^\circ \Rightarrow C^\circ}\ \Rightarrow_R$$

$$\overset{R}{\multimap}_L\colon \frac{\overset{\Xi_1}{\Gamma; \Upsilon; \varnothing; \varnothing \longrightarrow B} \quad \overset{\Xi_2}{\Gamma; \Upsilon; \Psi; \Delta, C \longrightarrow E}}{\Gamma; \Upsilon; \Psi; \Delta, B \overset{R}{\multimap} C \longrightarrow E}\ \overset{R}{\multimap}_L \quad \rightsquigarrow$$

$$\frac{\dfrac{\overset{\Xi_1^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ \longrightarrow B^\circ}}{\Gamma^\circ, \Upsilon^\circ; \varnothing \longrightarrow B^\circ}\ abs^* \quad \dfrac{\dfrac{\overset{\Xi_1^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ \longrightarrow B^\circ}}{\Gamma^\circ, \Upsilon^\circ; \varnothing \longrightarrow B^\circ}\ abs^* \quad \overset{\Xi_2^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ, C^\circ \longrightarrow E^\circ}}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ, B^\circ \multimap C^\circ \longrightarrow E^\circ}\ \multimap_L}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ, B^\circ \Rightarrow (B^\circ \multimap C^\circ) \longrightarrow E^\circ}\ \Rightarrow_L$$

$$\overset{R}{\multimap}_R\colon \frac{\Gamma; \Upsilon, B; \Psi; \overset{\Xi}{\Delta} \longrightarrow C}{\Gamma; \Upsilon; \Psi; \Delta \longrightarrow B \overset{R}{\multimap} C}\ \multimap_R \quad \rightsquigarrow \quad \frac{\dfrac{\overset{\Xi^\circ}{\Gamma^\circ, \Upsilon^\circ, B^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ, B^\circ \longrightarrow C^\circ}\ \multimap_R}{\Gamma^\circ, \Upsilon^\circ, B^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ \longrightarrow (B^\circ \multimap C^\circ)}\ \Rightarrow_R}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi^\circ)^{\&}\mathbf{1}, \Delta^\circ \longrightarrow B^\circ \Rightarrow (B^\circ \multimap C^\circ)}$$

$$\stackrel{A}{-\!\!\circ}_L\colon \frac{\overset{\Xi_1}{\Gamma;\varnothing;\Psi_1;\varnothing \longrightarrow B} \quad \overset{\Xi_2}{\Gamma;\Upsilon;\Psi_2;\Delta, C \longrightarrow E}}{\Gamma;\Upsilon;\Psi_1,\Psi_2;\Delta, B \stackrel{A}{-\!\!\circ} C \longrightarrow E}\ \stackrel{A}{-\!\!\circ}_L \quad \rightsquigarrow$$

$$\frac{\dfrac{\overset{\Xi_1^{\circ +\Upsilon}}{\Gamma^\circ,\Upsilon^\circ;(\Psi_1^\circ)^{\&\mathbf{1}} \longrightarrow B^\circ} \quad \dfrac{\overline{\Gamma^\circ,\Upsilon^\circ;\varnothing \longrightarrow \mathbf{1}}\ \mathbf{1}_R}{\Gamma^\circ,\Upsilon^\circ;(\Psi_1^\circ)^{\&\mathbf{1}} \longrightarrow \mathbf{1}}\ \&\mathbf{1}_L^*}{\Gamma^\circ,\Upsilon^\circ;(\Psi_1^\circ)^{\&\mathbf{1}} \longrightarrow (B^\circ \,\&\, \mathbf{1})}\ \&_R \qquad \overset{\Xi_2^\circ}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi_2^\circ)^{\&\mathbf{1}},\Delta^\circ, C^\circ \longrightarrow E^\circ}}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi_1^\circ)^{\&\mathbf{1}},(\Psi_2^\circ)^{\&\mathbf{1}},\Delta^\circ,(B^\circ \,\&\, \mathbf{1}) \multimap C^\circ \longrightarrow E^\circ}\ \multimap_L$$

$$\stackrel{A}{-\!\!\circ}_R\colon \frac{\overset{\Xi}{\Gamma;\Upsilon;\Psi,B;\Delta \longrightarrow C}}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B \stackrel{A}{-\!\!\circ} C}\ \multimap_R \quad \rightsquigarrow \quad \frac{\overset{\Xi^\circ}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},(B^\circ \,\&\, \mathbf{1}),\Delta^\circ \longrightarrow C^\circ}}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow (B^\circ \,\&\, \mathbf{1}) \multimap C^\circ}\ \Rightarrow_R$$

$$\&_{L_i}\colon \frac{\overset{\Xi_i}{\Gamma;\Upsilon;\Psi;\Delta,B_i \longrightarrow C}}{\Gamma;\Upsilon;\Psi;\Delta,B_1 \,\&\, B_2 \longrightarrow C}\ \&_{L_i} \quad \rightsquigarrow \quad \frac{\overset{\Xi_i^\circ}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ,B_i^\circ \longrightarrow C^\circ}}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ,(B_1^\circ \,\&\, B_2^\circ) \longrightarrow C^\circ}\ \&_{L_i}$$

$$\&_R\colon \frac{\overset{\Xi_1}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B_1} \quad \overset{\Xi_2}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B_2}}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B_1 \,\&\, B_2}\ \&_R \quad \rightsquigarrow$$

$$\frac{\overset{\Xi_1^\circ}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow B_1^\circ} \quad \overset{\Xi_2^\circ}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow B_2^\circ}}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow B_1^\circ \,\&\, B_2^\circ}\ \&_R$$

$$\forall_L\colon \frac{\overset{\Xi}{\Gamma;\Upsilon;\Psi;\Delta,B[x \mapsto t] \longrightarrow C}}{\Gamma;\Upsilon;\Psi;\Delta,\forall x.B \longrightarrow C}\ \forall_L \quad \rightsquigarrow \quad \frac{\overset{\Xi^\circ}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ,(B[x \mapsto t])^\circ \longrightarrow C^\circ}}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ,\forall x.(B^\circ) \longrightarrow C^\circ}\ \forall_L$$

$$\forall_R\colon \frac{\overset{\Xi}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B[x \mapsto c]}}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow \forall x.B}\ \forall_R \quad \rightsquigarrow \quad \frac{\overset{\Xi^\circ}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow (B[x \mapsto c])^\circ}}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow \forall x.(B^\circ)}\ \forall_R$$

$$\exists_R\colon \frac{\overset{\Xi}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B[x \mapsto t]}}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow \exists x.B}\ \exists_R \quad \rightsquigarrow \quad \frac{\overset{\Xi^\circ}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow (B[x \mapsto t])^\circ}}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow \exists x.(B^\circ)}\ \exists_R$$

$$\oplus_{R_i}\colon \frac{\overset{\Xi_i}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B_i}}{\Gamma;\Upsilon;\Psi;\Delta \longrightarrow B_1 \oplus B_2}\ \oplus_{R_i} \quad \rightsquigarrow \quad \frac{\overset{\Xi_i^\circ}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow B_i^\circ}}{\Gamma^\circ,\Upsilon^\circ;\Upsilon^\circ,(\Psi^\circ)^{\&\mathbf{1}},\Delta^\circ \longrightarrow B_1^\circ \oplus B_2^\circ}\ \oplus_{R_i}$$

$$!_R\colon \frac{\overset{\Xi}{\Gamma;\varnothing;\varnothing;\varnothing \longrightarrow B}}{\Gamma;\varnothing;\varnothing;\varnothing \longrightarrow !B}\ !_R \quad \rightsquigarrow \quad \frac{\overset{\Xi^\circ}{\Gamma^\circ;\varnothing \longrightarrow B^\circ}}{\Gamma^\circ;\varnothing \longrightarrow !(B^\circ)}\ !_R$$

$$\otimes_R\colon \frac{\overset{\Xi_1}{\Gamma;\Upsilon;\Psi_1;\Delta_1 \longrightarrow B_1} \quad \overset{\Xi_2}{\Gamma;\Upsilon;\Psi_2;\Delta_2 \longrightarrow B_2}}{\Gamma;\Upsilon;\Psi_1,\Psi_2;\Delta_1,\Delta_2 \longrightarrow B_1 \otimes B_2}\ \otimes_R \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\overset{\Xi_1^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi_1^\circ)^{\&}\mathbf{1}, \Delta_1^\circ \longrightarrow B_1^\circ} \quad \overset{\Xi_2^\circ}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi_2^\circ)^{\&}\mathbf{1}, \Delta_2^\circ \longrightarrow B_2^\circ}}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, \Upsilon^\circ, (\Psi_1^\circ)^{\&}\mathbf{1}, (\Psi_2^\circ)^{\&}\mathbf{1}, \Delta_1^\circ, \Delta_2^\circ \longrightarrow B_1^\circ \otimes B_2^\circ}\ \otimes_R}{\Gamma^\circ, \Upsilon^\circ; \Upsilon^\circ, (\Psi_1^\circ)^{\&}\mathbf{1}, (\Psi_2^\circ)^{\&}\mathbf{1}, \Delta_1^\circ, \Delta_2^\circ \longrightarrow B_1^\circ \otimes B_2^\circ}\ abs^*$$

■

Appendix G

An Enriched Version of the Dyckhoff Theorem Prover Example

```
% An enriched version of the Dyckhoff theorem prover for propositional
% intuitionistic logic that generates LaTeX code for the proofs it builds.
% The generated LaTeX code is dependent on the proof.sty style file.

MODULE propositional.

% Being an atom is defined in terms of negation-as-failure.

atom A :- not (A = (and B C)), not (A = (or B C)), not (A = (imp B C)),
          not (A = false), not (A = true).

not G :- G -> fail | true.

% collect the current list of hypotheses from the linear context.

gamma Flst :- hyp Fhd -> (gamma Ftl, Flst = (Fhd::Ftl)) | Flst = nil.

% The core of the prover. Basically the same as the basic version.
% But each rule collects the current list of hypotheses and uses it
% to build the proof term that is returned in the second argument.

pv true (top_R Gamma):- gamma Gamma.
pv C (false_L Gamma C) :- (hyp false, erase) & gamma Gamma.
pv C (identity Gamma C) :- (hyp C, erase) & gamma Gamma.

pv (and A B) (and_R Gamma A B Aproof Bproof) :-
        (pv A Aproof & pv B Bproof) & gamma Gamma.
pv (imp A B) (imp_R Gamma A B Bproof) :-
        (hyp A -o pv B Bproof) & gamma Gamma.
pv (or A B) ((or_R 1) Gamma A B Aproof) :-
        (pv A Aproof) & gamma Gamma.
pv (or A B) ((or_R 2) Gamma A B Bproof) :-
        (pv B Bproof) & gamma Gamma.
pv C (and_L Gamma C Cproof) :-
        (hyp (and A B), hyp A -o hyp B -o pv C Cproof) & gamma Gamma.
pv C (or_L Gamma C CproofA CproofB) :-
        (hyp (or A B), (hyp A -o pv C CproofA & hyp B -o pv C CproofB))
                & gamma Gamma.
pv C (true_L Gamma C Cproof) :-
        (hyp true, pv C Cproof) & gamma Gamma.
pv C ((imp_L 1) Gamma C A1impA2proof Cproof) :-
        (hyp (imp (imp A1 A2) B),
                (hyp (imp A2 B) -o pv (imp A1 A2) A1impA2proof
                & hyp B -o pv C Cproof)) & gamma Gamma.
pv C ((imp_L 2) Gamma C Cproof) :-
        (hyp (imp (and A1 A2) B),
                hyp (imp A1 (imp A2 B)) -o pv C Cproof) & gamma Gamma.
pv C ((imp_L 3) Gamma C Cproof) :-
        (hyp (imp (or A1 A2) B),
                hyp (imp A1 B) -o hyp (imp A2 B) -o pv C Cproof) & gamma Gamma.
pv C ((imp_L 4) Gamma C Cproof) :-
        (hyp (imp false B), pv C Cproof) & gamma Gamma.
pv C ((imp_L 5) Gamma C Cproof) :-
        (hyp (imp true B), hyp B -o pv C Cproof) & gamma Gamma.
pv C ((imp_L 6) Gamma C Cproof) :-
        (hyp (imp A B), atom A, hyp A,
                hyp B -o hyp A -o pv C Cproof) & gamma Gamma.
```

Figure G.1: The Core of the Enriched Dyckhoff Theorem Prover

```
% This module implements the routines to output LaTeX code for proofs.

MODULE proofprinter.

% pretty-print a proof (just adds $$ on each side of the proof).

pp Proof :- ws ^$^$, nl, wp Proof, nl, ws ^$^$, nl.

ws A :- write_sans A.

% write out a list of formulas (write-gamma)

wg nil.
wg (F::nil) :- wf F.
wg (F1::F2::Fs) :- wf F1, ws ^,, wg (F2::Fs).

% write out a formula

wf true :- ws ^\top, ws ^ .
wf false :- ws ^\perp, ws ^ .
wf A :- atom A, ws A, ws ^ .
wf (and A B) :- ws ^(, wf A, ws ^\wedge,  ws ^ , wf B, ws ^).
wf (or A B)  :- ws ^(, wf A, ws ^\vee,    ws ^ , wf B, ws ^).
wf (imp A B) :- ws ^(, wf A, ws ^\supset, ws ^ , wf B, ws ^).

% write out a proof term

wp (top_R Gamma) :-
        ws ^\infer^[^\top_R^]^{, wg Gamma, ws ^\lra^\top,ws ^}^{^}.
wp (false_L Gamma C) :-
        ws ^\infer^[^\perp_L^]^{, wg Gamma, ws ^\lra, ws ^ , wf C, ws ^}^{^}.
wp (identity Gamma C) :-
        ws ^\infer^{, wg Gamma, ws ^\lra, ws ^ , wf C, ws ^}^{^}.
wp (and_R Gamma A B Aproof Bproof) :-
        ws ^\infer^[^\wedge_R^]^{, wg Gamma, ws ^\lra, ws ^ , wf (and A B),
                ws ^}^{, wp Aproof, ws ^ ^&^ , wp Bproof, ws ^}.
wp (imp_R Gamma A B Bproof) :-
        ws ^\infer^[^\supset_R^]^{, wg Gamma, ws ^\lra, ws ^ , wf (imp A B),
                ws ^}^{, wp Bproof, ws ^}.
wp ((or_R N) Gamma A B Nproof) :-
        ws ^\infer^[^\vee_^{R_, ws N, ws ^}^]^{, wg Gamma, ws ^\lra, ws ^ ,
                wf (or A B), ws ^}^{, wp Nproof, ws ^}.
wp  (and_L Gamma C Cproof) :-
        ws ^\infer^[^\wedge_L^]^{, wg Gamma, ws ^\lra, ws ^ , wf C,
                ws ^}^{, wp Cproof, ws ^}.
wp  (or_L Gamma C CproofA CproofB) :-
        ws ^\infer^[^\vee_L^]^{, wg Gamma, ws ^\lra, ws ^ , wf C,
                ws ^}^{, wp CproofA, ws ^ ^&^ , wp CproofB, ws ^}.
wp (true_L Gamma C Cproof) :-
        ws ^\infer^[^\top_L^]^{, wg Gamma, ws ^\lra, ws ^ , wf C,
                ws ^}^{, wp Cproof, ws ^}.
wp ((imp_L 1) Gamma C A1impA2proof Cproof) :-
        ws ^\infer^[^\supset_^{L_1^}^]^{, wg Gamma, ws ^\lra, ws ^ , wf C,
                ws ^}^{, wp A1impA2proof, ws ^ ^&^ , wp Cproof, ws ^}.
wp ((imp_L A) Gamma C Cproof) :- not (A = 1),
        ws ^\infer^[^\supset_^{L_1^}^]^{, wg Gamma, ws ^\lra, ws ^ , wf C,
                ws ^}^{, wp Cproof, ws ^}.
```

Figure G.2: The Proof Pretty-Printing Module

```
% This is a sample file that shows how to interface to the enriched prover.
% Thanks to Bob Constable for some of the more interesting formulas.

% Sample use:
%
% ?- test --o prop --o test.
% Output File Name: "test_output".
% solved
%

MODULE test.

prop 1  (imp a a).
prop 2  (imp (or a b) (or b a)).
prop 3  (imp (and a b) (and b a)).
prop 4  (imp a (imp b a)).
prop 5  (imp (imp a (imp b c)) (imp (imp a b) (imp a c))).
prop 6  (imp (imp (imp a b) a) (imp (imp a b) b)).
prop 7  (imp (imp (imp a b) a) a).
prop 8  (imp (and (imp a b) (imp c a)) (imp c b)).
prop 9  (imp (or a b) (imp (imp a c) (imp (imp b c) c))).
prop 10 (imp (or a b) (imp (imp a c) (imp (imp (and b d) c) (imp d c)))).
prop 11 (imp (or (imp a b) (imp a c)) (imp a (or b c))).
prop 12 (imp (or a b) (imp (imp a c) (imp (imp b (imp d c)) (imp d c)))).
prop 13 (imp (and (and a a) (imp a b)) b).
prop 14 (imp a (imp a (imp (imp a b) b))).
prop 15 (imp (or c (imp c false)) (imp (and (imp a b) (imp (imp a c) b)) b)).
prop 16 (imp (or b (imp b false)) (imp (and (imp a b) (imp (imp a c) b)) b)).
prop 17 (imp (or a (imp a false)) (imp (and (imp a b) (imp (imp a c) b)) b)).
prop 18 (imp (imp b (imp b false)) (imp (imp (imp b false) false) b)).


test :- write_sans "Output File Name: ", read FN, (telling FN dotest ; true).

% This does the work. Note that we could use the system/2 predicate to
% actuall call LaTex to compile the code, and Xdvi to display it.

dotest :- ws ^\documentstyle^[proof^]^{article^}, nl,nl,
          ws ^\begin^{document^},nl,nl,
          ((prop A B, nl, ws ^\noindent^ Example^ , ws A, ws ^:^ ,
                  (pv B P -> succeeded P | failed B), fail);
           (ws ^\end^{document^},nl,nl)).

failed B :- ws ^ the^ sequent^ ^$^\lra^ , wf B,
            ws ^$^ is^ not^ provable^., nl, nl.

succeeded P :- nl, pp P.
```

Figure G.3: A Sample Interface to the Prover

```
\documentstyle[proof]{article}

\begin{document}

\noindent Example 1:
$$
\infer[\supset_R]{\lra (a \supset a )}{\infer{a \lra a }{}}
$$

\noindent Example 2:
$$
\infer[\supset_R]
  {\lra ((a \vee b )\supset (b \vee a ))}
  {\infer[\vee_L]
     {(a \vee b )\lra (b \vee a )}
     {\infer[\vee_{R_2}]
        {a \lra (b \vee a )}
        {\infer{a \lra a }{}} &
      \infer[\vee_{R_1}]
        {b \lra (b \vee a )}
        {\infer{b \lra b }{}}}}
$$

          .
          .
          .

\noindent Example 7: the sequent $\lra (((a \supset b )\supset a )\supset a )$
                     is not provable.

\noindent Example 8:
$$
\infer[\supset_R]
  {\lra (((a \supset b )\wedge (c \supset a ))\supset (c \supset b ))}
  {\infer[\supset_R]
     {((a \supset b )\wedge (c \supset a ))\lra (c \supset b )}
     {\infer[\wedge_L]
        {c ,((a \supset b )\wedge (c \supset a ))\lra b }
        {\infer[\supset_{L_1}]
           {(c \supset a ),(a \supset b ),c \lra b }
           {\infer[\supset_{L_1}]
              {c ,a ,(a \supset b )\lra b }
              {\infer{a ,b ,c \lra b }{}}}}}}
$$
\end{document}
```

Figure G.4: Some of the LaTeX Code Generated by the System

Example 1:

$$\dfrac{\dfrac{}{a \longrightarrow a}}{\longrightarrow (a \supset a)}\ \supset_R$$

Example 2:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \longrightarrow a}}{a \longrightarrow (b \vee a)}\ \vee_{R_2} \quad \dfrac{\dfrac{}{b \longrightarrow b}}{b \longrightarrow (b \vee a)}\ \vee_{R_1}}{(a \vee b) \longrightarrow (b \vee a)}\ \vee_L}{\longrightarrow ((a \vee b) \supset (b \vee a))}\ \supset_R$$

$$\vdots$$

Example 7: the sequent $\longrightarrow (((a \supset b) \supset a) \supset a)$ is not provable.
Example 8:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a, b, c \longrightarrow b}}{c, a, (a \supset b) \longrightarrow b}\ \supset_{L_1}}{(c \supset a), (a \supset b), c \longrightarrow b}\ \supset_{L_1}}{c, ((a \supset b) \wedge (c \supset a)) \longrightarrow b}\ \wedge_L}{((a \supset b) \wedge (c \supset a)) \longrightarrow (c \supset b)}\ \supset_R}{\longrightarrow (((a \supset b) \wedge (c \supset a)) \supset (c \supset b))}\ \supset_R$$

Figure G.5: Some of the Output Generated by the System

Bibliography

[Aït-Kaci, 1986] Hassan Aït-Kaci. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):185–215, 1986.

[Andreoli and Pareschi, 1989] J.-M. Andreoli and R. Pareschi. From logic to objects. Technical report, ECRC, 1989.

[Andreoli and Pareschi, 1990*a*] J.-M. Andreoli and R. Pareschi. Formulae as active representation of data. In *Actes du 9eme Séminaire sur la Programmation en Logique*, 1990.

[Andreoli and Pareschi, 1990*b*] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In *Proceeding of the Seventh International Conference on Logic Programming, Jerusalem*, May 1990.

[Andreoli and Pareschi, 1990*c*] J.-M. Andreoli and R. Pareschi. LO and behold! concurrent structured processes. In *Proceedings of OOPSLA 90*, 1990.

[Andreoli and Pareschi, 1991*a*] J.-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proceedings of OOPSLA 91*, pages 212–229, 1991.

[Andreoli and Pareschi, 1991*b*] Jean-Marc Andreoli and Remo Pareschi. Logic programming with sequent systems: a linear logic approach. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, volume 475 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1991.

[Andreoli, 1990] J.-M. Andreoli. *Proposal for a Synthesis of Logic and Object-Oriented Programming Paradigms*. PhD thesis, University of Paris VI, 1990. English Summary.

[Andreoli, 1992] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 1992.

[Bollen, 1991] A. W. Bollen. Relevant logic programming. *Journal of Automated Reasoning*, 7(4):563–586, December 1991.

[Bonner *et al.*, 1989] Anthony J. Bonner, L. Thorne McCarty, and Kumar Vadaparty. Expressing database queries with intuitionistic logic. In *Logic Programming: Proceedings of the North American Conference*, pages 831–850, 1989.

[Börger and Rosenzweig, 1991] Egon Börger and Dean Rosenzweig. From Prolog algebras towards WAM – a mathematical study of implementation. In Egon Börger, editor, *CSL'90. Fourth Workshop on Computer Science Logic*, number 533 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[Börger, 1990] Egon Börger. A logical operational semantics of full Prolog. Technical Report IWBS Report 111, IBM, March 1990.

[Bowen and Kowalski, 1982] Kenneth A. Bowen and Robert A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.-A. Tarnlund, editors, *Logic programming*, volume 16 of *APIC studies in data processing*, pages 153 – 172. Academic Press, 1982.

[Carlsson, 1984] Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2(4):347 – 359, 1984.

[Cerrito, 1990] Serenella Cerrito. A linear semantics for allowed logic programs. In John Mitchell, editor, *Proceedings of the Fifth Annual Symposium on Logic in Computer Science, Philadelphia, PA*, pages 219 – 227, June 1990.

[Church, 1940] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Dyckhoff, 1992] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3), September 1992.

[Elliott and Pfenning, 1991] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289 – 325. MIT Press, 1991.

[Felty and Miller, 1988] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pages 61 – 80, Argonne, IL, May 1988. Springer-Verlag.

[Felty, 1989] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, August 1989.

[Gabbay and Reyle, 1984] D. M. Gabbay and U. Reyle. N-Prolog: An extension of Prolog with hypothetical implications. I. *Journal of Logic Programming*, 1:319 – 355, 1984.

[Gabbay, 1991] Dov M. Gabbay. Algorithmic proof with diminishing resources, part 1. In Egon Börger, editor, *CSL'90. Fourth Workshop on Computer Science Logic*, number 533 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[Gallier, 1986] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper & Row, 1986.

[Gazdar *et al.*, 1985] Gerald J. M. Gazdar, Ewan Klein, Geoffrey K. Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, Mass., 1985.

[Gazdar, 1981] Gerald J. M. Gazdar. Unbounded dependencies and coordinate structure. *Linguistic Inquiry*, 12(2):154–184, 1981.

[Gazdar, 1982] Gerald J. M. Gazdar. Phrase structure grammar. In P. Jacobson and G. K. Pullum, editors, *The Nature of Syntactic Representation*, pages 131–186. Reidel, Dordrecht, 1982.

[Gentzen, 1969] Gerhard 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.

[Girard *et al.*, 1989] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[Girard, 1987*a*] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Girard, 1987*b*] Jean-Yves Girard. Towards a geometry of interaction. In *Categories in Computer Science*, volume 92 of *Contemporary Mathematics*, pages 69 – 108. AMS, June 1987.

[Girard, 1991] Jean-Yves Girard. On the unity of logic. Technical Report 26, Université Paris VII, June 1991.

[Hallnäs and Schroeder-Heister, 1990] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. 1. Clauses as rules. *Journal of Logic and Computation*, pages 261–283, December 1990.

[Hannan and Miller, 1988] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In *Fifth International Logic Programming Conference*, pages 942–959, Seattle, Washington, August 1988. MIT Press.

[Harland and Pym, 1990] James Harland and David Pym. The uniform proof-theoretic foundation of linear logic programming. Technical Report ECS-LFCS-90-124, Laboratory for the Foundations of Computer Science, University of Edinburgh, November 1990.

[Harland and Pym, 1991] James Harland and David Pym. The uniform proof-theoretic foundation of linear logic programming (extended abstract). In *Proceedings of the 1991 International Logic Programming Symposium, San Diego*, November 1991.

[Harland, 1991] James Harland. *The Uniform Proof Theoretic Foundations of Logic Programming*. PhD thesis, University of Edinburgh, 1991.

[Harland, 1993] James Harland. A proof-theoretic analysis of goal-directed provability. *Journal of Logic and Computation*, 1993. To appear.

[Harrop, 1960] R. Harrop. Concerning formulas of the types $A \rightarrow B \vee C$, $A \rightarrow (Ex)B(x)$ in intuitionistic formal systems. *Journal of Symbolic Logic*, pages 27–32, 1960.

[Hodas and Miller, 1990] Joshua Hodas and Dale Miller. Representing objects in a logic programming language with scoping constructs. In David D. H. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 511–526. MIT Press, June 1990.

[Hodas and Miller, 1991] Joshus Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 32 – 42, Amsterdam, July 1991.

[Hodas and Miller, 1994] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1994. To appear.

[Hodas, 1992] Joshua Hodas. Specifying filler-gap dependency parsers in a linear-logic programming language. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 622 – 636, 1992.

[Jayaraman and Nadathur, 1991] Bharat Jayaraman and Gopalan Nadathur. Implementation techniques for scoping constructs in logic programming. In *Eighth International Logic Programming Conference*, Paris, France, June 1991. MIT Press.

[Joshi, 1983] Aravind K. Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural description? In David Dowty, Lauri Karttunen, and Arnold Zwicky, editors, *Natural language processing: psycholinguistic, computational and theoretical perspectives*, pages 206–250. Cambridge University Press, New York, 1983.

[Joshi, 1986] A.K. Joshi. An introduction to tree adjoining grammars. In A. Manaster-Ramer, editor, *The Mathematics of Language*. Benjamins, 1986.

[Kanovich, 1993] Max Kanovich. The expressive power of initial fragments of linear logic. A talk given at the Linear Logic Workshop, Cornell University, June 1993.

[Kursawe, 1987] Peter Kursawe. How to invent a Prolog machine. *New Generation Computing*, 5(1), 1987.

[Lamma *et al.*,] Lamma, Mello, and Natali. An extended warren abstract machine for the execution of structured logic programs. *Journal of Information and Computation*. In review.

[Lincoln *et al.*, 1990] P. Lincoln, J. Mitchell, A. Scedrov, and N. Shankar. Decision problems for propositional linear logic. Technical Report SRI-CSL-90-08, SRI International, 1990.

[Lincoln *et al.*, 1991] Patrick Lincoln, Andre Scedrov, and Natrajan Shankar. Linearizing intuitionistic implication. In Giles Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 51–62, July 1991.

[Lloyd, 1984] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1984.

[McCabe, 1989] F. G. McCabe. *Logic and Objects: Language, application, and implementation*. PhD thesis, Imperial College of Science and Technology, 1989.

[McCarty, 1988] L. T. McCarty. Clausal intuitionistic logic I. Fixed point semantics. *Journal of Logic Programming*, 5:1 – 31, 1988.

[Miller and Nadathur, 1986] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[Miller and Nadathur, 1988] Dale Miller and Gopalan Nadathur. λProlog Version 2.7. Distribution in C-Prolog and Quintus sources, July 1988.

[Miller *et al.*, 1991] 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.

[Miller, 1987] Dale Miller. Hereditary harrop formulas and logic programming. In *Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science*, pages 153–156, Moscow, August 1987.

[Miller, 1989*a*] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[Miller, 1989*b*] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 – 108, 1989.

[Miller, 1990] Dale Miller. Abstractions in logic programming. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329 – 359. Academic Press, 1990.

[Miller, 1991*a*] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497 – 536, 1991.

[Miller, 1991*b*] Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, Paris, France, June 1991. MIT Press.

[Miller, 1993] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in Lecture Notes in Computer Science, pages 242–265. Springer-Verlag, 1993.

[Monteiro and Porto, 1989] L. Monteiro and A. Porto. Contextual logic programming. In *Proceedings of the 6th International Conference on Logic Programming, Lisbon*. The MIT Press, 1989.

[Nadathur and Miller, 1990] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777 – 814, October 1990.

[Nadathur, 1987] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming*. PhD thesis, University of Pennsylvania, May 1987.

[Nilsson, 1990] Ulf Nilsson. Towards a methodology for the design of abstract machines for logic programming languages. Technical Report LiTH-IDA-R-90-12, Linköping University, 1990.

[Pareschi and Miller, 1990] Remo Pareschi and Dale Miller. Extending definite clause grammars with scoping constructs. In David D. H. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 373–389. MIT Press, June 1990.

[Pareschi, 1989] Remo Pareschi. *Type-driven Natural Language Analysis*. PhD thesis, University of Edinburgh, 1989.

[Paulson, 1990] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361 – 386. Academic Press, 1990.

[Pereira and Shieber, 1987] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*, volume 10. CLSI, Stanford, CA, 1987.

[Pfenning, 1988] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Lisp and Functional Programming Conference*, 1988.

[Robinson, 1965] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[Ross, 1967] John Robert Ross. *Constraints on Variables in Syntax*. PhD thesis, Massachusetts Institute of Technology, 1967.

[Schellinx, 1991] H. Schellinx. Some syntactical observations on linear logic. *Journal of Logic and Computation*, 1(4):537–559, 1991.

[Steedman, 1988] Mark J. Steedman. Combinators and grammars. In Bach Oehrle and Wheeler, editors, *Categorial Grammars and Natural Language Structures*, 1988.

[Tärnlund, 1977] Sten-Åke Tärnlund. Horn clause computability. *BIT*, 17:215 – 226, 1977.

[Warren, 1977] David H. D. Warren. Implementing Prolog – compiling predicate logic programs. Technical Report Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, May 1977. Also published as second half of Warren's thesis.

[Warren, 1983] David H. D. Warren. An abstract Prolog instruction set. Technical Report Technical Report 309, SRI International, October 1983.

[Warren, 1990] David S. Warren. The XWAM: A machine that integrates prolog and deductive database query evaluation. In Jonathan W. Mills, editor, *Second NACLP Workshop on Logic Programming Architectures and Implementations, Austin, Texas, November 1990*, 1990.

[Wirth, 1976] Niklas Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, NJ, 1976.