Proof theory, proof search, and logic programming

Draft: 25-08-2023

This draft includes 10 of a total of 15 planned chapters. The missing chapters are following:

- Chapters 8 and 9 deal with higher-order quantification (and these need significant work) and
- Chapters 11 (finite state machines), 14 (proof checking), and 15 (discussions). Of these, Chapter 14 will be a rewrite of an existing paper.

Comments and corrections are welcome.

© **Dale Miller**
Inria Saclay & Laboratoire d'Informatique (LIX)
1 rue Honoré d'Estienne d'Orves
Campus de l'École Polytechnique
91120 Palaiseau France
dale.miller at inria.fr

Contents

Preface

This book develops some of the proof theory of classical, intuitionistic, and linear logics and applies that theory to the design and applications of logic programming. In particular, we view computation based on logic programs as searching for specific kinds of proof. During the search for a proof, the *current logic program* $\mathcal{P}$ and the *current goal* G are recorded using the simple pairing construction, $\mathcal{P} \vdash G$, formally called a *sequent*. We shall use Gentzen's sequent calculus framework to formalize the notion of proof in our setting. Of all the many ways one might attempt a proof of the sequent $\mathcal{P} \vdash G$, we shall limit ourselves to proofs that are *goal-directed*. The notion of goal-directed proof search is formalized using the technical concept of *uniform proof* in which sequent calculus proofs are built using alternating phases, one performs *goal-reduction* and the other *backchaining*. The completeness of uniform proofs is a formal criterion for judging if a particular choice of goal formulas and logic programs yields a logic programming language.

This proof theory foundation allows us to define several logic programming languages based on first-order and higher-order classical, intuitionistic, and linear logics. In this way, we provide a proof-theoretic foundation for Prolog (using first-order Horn clauses in classical logic), λProlog (using higher-order hereditary Harrop formulas in intuitionistic logic), and two linear logic programming languages, named Lolli and Forum. As we shall illustrate, these increasingly expressive logic programming languages add abilities to the logic programming paradigm to express modular programming, higher-order programming, abstract datatypes, state encapsulation, and concurrency.

When attempting to develop the proof theory of uniform proofs, one notices that they allow for asymmetry between the goals and logic programs and that asymmetry makes the development of a proof theory for uniform proofs difficult. By making certain simple restrictions on the occurrences of logical connectives, uniform proofs become examples of *focused proofs*, and these provide an excellent setting to develop several proof-theoretic concepts, including cut elimination.

The reader of this book will learn the following, grouped into the three themes of this book.

1. Proof theory
   - Sequent calculus proof systems for classical, intuitionistic, and linear logics.
   - Major proof-theoretic properties of these sequent calculus proof systems, including cut-elimination for first-order and higher-order versions of intuitionistic and linear logics.
2. Proof search
   - The operational reading of logic formulas as programs via their impact on the construction of proofs.
   - The formalization of goal-directed search using the notion of uniform proofs.
   - Focused proof systems as a formal framework to provide structures to the sequent calculus to support the search for proofs.
3. Logic programming
   - Examples of deploying the meta-theory of sequent calculus proofs to support formal reasoning about logic programs.
   - The application of logic programming in intuitionistic and linear logic to several computer-science-motivated applications, including the specification of finite state machines, security protocols, operational semantics for other programming languages, and flexible proof-checking mechanisms.

This book does not discuss the Curry-Howard correspondence (the "proofs-as-programs" paradigm). Relating computation to proof search offers an entirely different dimension to the role of logic and proof in computational logic than the one offered by the Curry-Howard correspondence.

The reader of this book should be familiar with the basic syntactic properties of first-order logic and the (simply typed) λ-calculus. No background in the formal representation of proofs is needed, although such a background is helpful. We shall occasionally present examples of logic programs to help illustrate proof-theoretic concepts: such examples will be presented using the syntactic conventions of λProlog. While some familiarity with Prolog or λProlog is helpful for understanding the examples, the reader unfamilar with these languages should be able to comfortably read these examples since the syntactic conventions of λProlog closely match the syntactic conventions used in presenting formulas in (higher-order) logic.

The search for proofs has many dimensions that are not addressed here. In particular, this book does not cover topics related to the implementation of proof search: for example, unification and backtracking search are not explicitly discussed. Thus, the approach here differs from standard approaches to describing the foundations of Prolog using, for example, SLD-resolution: that approach explicitly employs the notion of term unification in such foundations. Here, our foundation formally leaves unification out of the picture, although it is implicitly used in developing our example applications.

The first part of this book, which ends with Chapter 9, describes how the sequent calculus can be used to design and reason about various logic programming languages based on classical, intuitionistic, and linear logics. In the second part of this book, which starts with Chapter 10, we describe applications of some of these logic programming languages.

Many chapters contain exercises designed to illustrate and explore ideas related to the main text. Exercises marked by (‡) have partial or complete solutions towards the end of this book.

Acknowledgments. Versions of this book have been used in graduate-level (M2) courses in Paris, Copenhagen, Venice, Bertinoro, and Pisa. I thank the many students from these courses and Gopalan Nadathur for their comments on earlier drafts of this book.

Chapter 1

Introduction

There are many ways to specify and reason about computation. The early work of Church, Turing, Gödel, Curry, and others revealed that several different specification devices—such as the λ-calculus, Turing machines, and recursive equations—all specified the same set of *computable functions*. Many programming languages—such as LISP, C, Pascal, and Ada—have been designed that can be used to implement (in principle) this same set of computable functions. Apparently, no programming language can be viewed as canonical: the choice of which programming language one uses comes down to issues such as which language has compilers for a particular piece of computer hardware, which language is being used by one's collaborators, etc.

Given that logic can be seen as arising from foundational concerns within mathematics and computer science, it is interesting to consider using logical expressions themselves as programs. The logic programming paradigm arises from directly addressing questions such as: How might logic be used directly as a programming language? How expressive can such logic programming languages be? What benefits arise from basing the syntax and operational meaning of programs on techniques and ideas formulated by logicians in the first half of the 20^{th} century?

This book addresses this latter set of questions. But first, we address the fact that there are many logics and kinds of proof by organizing them into a conceptually clean framework before attempting to deliver a foundation for logic-based programming.

1.1 A spectrum of logics

The syntax for terms and formulas will be given in Chapter 2 using the framework provided by Church in his Simple Theory of Types [1940]: in particular, both terms *and* formulas are simply typed λ-terms, and the equality of terms and formulas is identified with the equality of such λ-terms (i.e., by the equations of α, β, and η conversion). Terms that have a particular primitive formula type—the Greek letter omicron o (following [Church, 1940])—are classified, in fact, as formulas. The symbol $\wedge$, $\vee$, and $\supset$ are written in infix to denote conjunction, disjunction, and implications, respectively. Negation will be written as the prefix operator $\neg$.

In this book, logics are classified along two major axes. The first axis involves the universal $\forall$ and existential $\exists$ quantifiers. A logic with no quantifiers is a *propositional logic*. A logic with quantifiers is a *quantificational logic*. Quantifiers in this book will bind *typed variables* (again following Church [1940]). A logic in which the type of a quantified variable is limited to primitive and non-propositional types is *first-order*. A *higher-order* logic allows quantification at all types, including propositional and functional types.

The second axis consists of the following three logics.

- *Classical logic* is a logic of truth values. For example, propositional formulas are either true or false depending on the truth value of the propositional variables it contains. Such a truth value can be computed using truth tables. For example, the formulas $p \vee \neg p$ and $((p \supset q) \supset p) \supset p$ are true no matter what truth value is given to p and q.

- *Intuitionistic logic* can be seen as a logic based on a constructive approach to proof. For example, a proof that the formula $\exists x.B(x)$ is a theorem must contain a specific term, say t, and a proof that $B(t)$ is a theorem. Similarly, a proof that $B_1 \vee B_2$ is a theorem contains a specific value of $i \in \{1, 2\}$ and a proof of B_i. For this reason, the formula $p \vee \neg p$ may not be a theorem since, without more information about p, we might not be able to provide a proof of either p or $\neg p$. If p is a statement such as $3 = 4$ then we can prove $p \vee \neg p$ since we can presumably prove $\neg(3 = 4)$. However, if we know nothing about p, then we cannot prove either of these disjuncts.

- *Linear logic*, introduced by Girard [1987], can be seen as a logic of resources. For example, having one occurrence of p can be different from having two occurrences, as in $p \wedge p$. As such, it is possible to model vending machines (e.g., two 50 cent coins yields one coffee), Petri nets, and process calculi.

Gentzen [1935] introduced the *sequent calculus* as a technical device to represent proofs in both classical and intuitionistic logics. The sequent calculus also provides an ideal setting for describing proofs for linear logic. As a result, we adopt the sequent calculus here and stress the modular and straightforward way in which it can be used to describe provability in these three logics. Our approach here does not attempt to merge classical, intuitionistic, and linear logics into one logic: instead, we view these logics as having different but closely related proof systems. In fact, the proof systems will be so closely related, that results in one of these logics can often be lifted with slight modification to provide results in another logic.

1.2 Logic and the specification of computations

Logic can be applied to the specification of computing in a few ways. We give an overview of these roles for logic in order to identify the particular niche that is our focus in this book.

In the specification of computation, logic is generally used in one of two approaches. In the *computation-as-model* approach, computations are encoded as mathematical structures, containing such items as nodes, transitions, and states. Logic is used in an external sense to make statements *about* those structures. That is, computations are used as models for logical formulas. Intensional operators, such as the triples of Hoare logic or the modals of temporal and dynamic logics, are often employed to express propositions about state changes. This use of logic to represent and reason about computation is probably the oldest and most broadly successful use of logic specifications with computation.

The *computation-as-deduction* approach uses pieces of logic's syntax (such as formulas, terms, types, and proofs) as elements of the specified computation. In this more rarefied setting, there are two different approaches to how computation is modeled.

The *proof normalization* approach views the state of a computation as a proof term and the process of computing as normalization (know variously as β-reduction or cut-elimination). Functional programming can be explained using proof-normalization as its theoretical basis [Martin-Löf, 1982] and has been used to justify the design of new functional programming languages [Abramsky, 1993].

The *proof search* approach views the state of a computation as a sequent (a structured collection of formulas) and the process of computing as the process of searching for a proof of a sequent: the changes that take place in sequents capture the dynamics of computation. This perspective on computation is the subject of this book.

Both of these programming paradigms include nondeterminism in their computational mechanisms. When functional programming languages are designed based on proof normalization, explicit control of the order in which redexes are rewritten are usually carefully described: such controls are often associated with either *call-by-value* or the *call-by-name*. In general, evaluation in functional programming languages is so tightly controlled that evaluation becomes deterministic. Computation based on searching for proofs is also nondeterministic. Removing some elements of nondeterminism is often a design goal of most logic programming languages and their interpreters. In general, however, some nondeterminism is retained in logic programming languages: it presence and exploitation provides some of the expressiveness of the logic programming paradigm.

The separation of proof normalization from proof search given above is informal and suggestive: such a division helps point out different sets of concerns represented by these two broad approaches. For example, proof normalization focuses on describing rewritings and their confluence, while proof search focuses on the nondeterminism and the reverse reading of inference rules. Of course, new advances in computational logic and proof theory might allow us to merge or reorganize this classification.

1.3 Proof search and logic programming

The earliest theoretical framework for logic programming was not an analysis of proofs but rather of resolution *refutations* [Robinson, 1965] and, in particular, SLD-resolution. This choice of foundations for logic programming was unfortunate for at least the following reasons.

1. Resolution is used to *refute*: that is, it attempts to derive a contradiction. This choice is counterintuitive since logic programming certainly seems to be about *proving* a goal formula from a collection of other formulas (the logic program).

2. Most refutation systems work with formulas that are in conjunctive normal form and Skolem normal form. Unfortunately, the only logic we wish to study for which restricting to such normal forms is possible is classical logic. Furthermore, these normal forms are not preserved when higher-order, predicate variables are substituted with expressions containing quantifiers and connectives.

3. A key inference step in resolution is the computation of *most general unifiers*. In many ways, unification seems to be part of the *implementation* behind the interplay of quantification and equality. It seems more natural first to try to understand that interplay before forcing one to implement it.

It is thus appealing to find a different approach to describing logic programming that is cast in terms of proving and in which normal forms and unification are not required. The sequent calculus provides just such a setting. Furthermore, removing unification from the abstract notion of proof search has a couple of benefits. First, it makes it possible for the interplay between universal and existential quantifiers to be explored without forcing the use of *Skolem functions*. Second, the use of *most general unifiers* within resolution means that it cannot handle those situations where most general unifiers do not exist (which can happen when attempting to unify simply typed λ-terms [Huet, 1975]).

1.4 Designing logic programming languages

A concern in the early history in the development of Prolog focused on how best to control search within a Prolog interpreter. For example, Kowalski [1979] proposed the equation

$$\text{Algorithm} = \text{Logic} + \text{Control},$$

which makes the important point that there is a gap between logic (here, first-order Horn clause specifications) and algorithms. For example, the naive Horn clause specification of the Fibonacci series could yield both the exponential-time algorithm and the linear time algorithm depending on whether a top-down (goal-directed) or a bottom-up (program-directed) proof search is employed. Clearly, the programmer must be able to have some control over which of these algorithms ultimately arises from this single logic specification. Various non-logical features have also been added to Prolog—such as the cut `!` and negation-as-failure—in order to allow for some explicit control of search.

Given that the logical foundation of Prolog is rather weak (see the discussion in Section 5.13), the design of new logic programming languages have made several additional extensions to logic, yielding a equation more like the following.

$$\begin{aligned}\text{Programming} = \text{Logic} \quad & + \quad \text{Control} \quad + \quad \text{Input/Output} \\ & + \quad \text{Higher-order programming} \\ & + \quad \text{Data abstractions} \\ & + \quad \text{Modules} \\ & + \quad \text{Concurrency} + \ldots.\end{aligned}$$

Such extensions are generally made in an *ad hoc* fashion and logic, which was the motivation and the intriguing starting point for a language like Prolog, was moved from center stage. With such an approach to building a programming language, the features added to address, say, higher-order programming can interact in complex ways with features that were added to address, say, modules. Describing such interaction of features can greatly complicate the design, implementation, and semantics of a programming language.

A interesting project is to see how one might satisfy the equation

$$\text{Programming} = \text{Logic}.$$

If this equation is at all possible, then one will certainly need to rethink what is meant by "Programming" and by "Logic." This book explores reinterpreting "Logic" by moving from first-order classical logic of Horn clauses to intuitionistic and linear logics possibly based on higher-order quantification. Chapters 12 through 14 provide several extended examples in which the task of programming and the use of rich logics coincide.

1.5 Why use logic to write programs?

Several benefits arise from writing programs as logic formulas and viewing computation as the construction of proofs. We list several here.

1. Logical formulas come with various operations on them that generally satisfy useful properties. For example, applying substitutions into formulas or replacing a subformula with a logically equivalence subformula is meaningful. Thus, applying substitutions into programs and then applying, say, modus ponens to two program clauses could well be expected to return a new, meaning-preserving program element.

2. There are generally multiple ways to describe central concepts in logic. For example, the set of theorems can usually be described as both the set of all provable formulas and the set of all true formulas (based on some suitable model theory). Also, provability might be characterized in strikingly different ways: via, for example, sequent calculus proofs, natural deduction, resolution refutations, tableaux, etc. Thus, different models of logic program execution might be structured in different ways while preserving the original declarative meaning of the program.

3. Proof theory generally comes with various kinds of abstractions, and a suitably designed logic programming language can harness these. For example, higher-order intuitionistic logic can provide logic programs with abstract data types, modular programming, and higher-order programming. Furthermore, all new features do not have undefined or complex interactions.

4. The meaning of logics we consider here have universally accepted descriptions. Thus, logic programs can, in principle, be meaningful many years in the future even if no particular compiler or interpreter used to execute them today is available in that future time.

Such benefits from using logic as a programming language are rather striking and worthy of additional exploration.

1.6 Bibliographic notes

The *Stanford Encyclopedia of Philosophy* has good, overview articles on proof theory [Rathjen and Sieg, 2020], the development of proof theory [Plato, 2018], intuitionistic logic [Moschovakis, 2021], linear logic [Di Cosmo and Miller, 2019], and Church's Simple Theory of Types [Benzmüller and Andrews, 2019].

For more about the use of resolution and SLD-resolution to describe logic programming based on Horn clauses in first-order classical logic, see the early papers [Apt and Emden, 1982] and [Emden and Kowalski, 1976], as well as textbooks such as [Gallier, 1986] and [Lloyd, 1987]. The author has written about the influences that logic programming and proof theory have had on each other [2021] as well as a survey [2022] describing several decades of research into using proof theory as a foundation for logic programming.

Chapter 2

Terms, formulas, and sequents

This book covers topics in both first-order and higher-order logic. Only first-order quantification is used in Chapters 3 through 7 while higher-order quantification will be used in the remaining chapters. This chapter provides the basic syntactic definitions and operations for higher-order quantification and higher-order substitutions: the first-order variants of quantification and substitution can be seen as a natural restriction on the general setting.

In his 1940 paper, Church presented the *simple theory of types (STT)* as a higher-order version of classical logic in which the simply typed λ-calculus is used to organize its syntax. Since Church's goal for STT was to formulate a logical foundation for mathematics, he also added to STT various mathematically motivated axioms, such as those for choice, extensionality, and infinity. By ignoring these mathematical axioms, one has a logical system, called *elementary theory of types (ETT)* [Andrews, 1974], that is useful for exploring the nature of higher-order quantification within logic.

2.1 Untyped λ-terms

While we will employ simply typed λ-terms throughout this book, we briefly consider the untyped λ-calculus, which shares an equality theory with the simply typed terms.

We shall start our syntax presentation by assuming that there is a fixed and denumerably infinite set of *tokens* (or identifiers). In this section, we will use the term *token* and *variable* interchangeable. Later in this chapter, when we introduce different ways to declare the type and scope of bindings for tokens, we shall distinguish between token-as-variable and token-as-constant. Such tokens are considered as variables in the λ-calculus. There are two other ways to build λ-term. Given two terms, say M and N, their *application* is (MN) (applications is the infix juxtaposition operation and it associates to the left). Given a term M and a token x, the *abstraction* of x over M is $(\lambda x.M)$. Here, the token x is a bound variable with scope M. We shall often drop the outermost parentheses and the period to improve readability.

The usual notions of free and bound occurrences of variables are assumed. If two terms differ up to an alphabetic change of their bound variables, we say that these terms are α-convert. We identify two terms up to such α-conversion. A subexpression of the form $(\lambda x.M)N$ is a *β-redex* and a subexpression of the form $(\lambda x.(Mx))$, where x has no free occurrence in M, is an *η-redex*. Replacing an occurrence of the β-redex $((\lambda x.M)N)$ with the capture-avoiding substitution of N for x in M, also written as $M[x/N]$, is called *β-reduction*. The converse relation is called β-expansion. A term is β-convertible to a term s if there is a sequence (including the empty sequence) of β-reductions and β-expansions steps that rewrites t to s. Replacing an occurrence of an η-redex $(\lambda x.(Mx))$ with M is called *η-reduction*. The converse relation is called η-expansion. A term is η-convertible to a term s if there is a sequence (including the empty sequence) of η-reductions and η-expansions steps that rewrites t to s. A term M is $\beta\eta$-convertible to N if there is a sequence of β-conversion and η-conversion steps that carries M to N. When we use the terms *β-conversion* and *$\beta\eta$-conversion*, we always assume the α-conversion rule is implicit.

A term is *β-normal* if it does not contain a β-redex. Stated in a positive form, a term is β-normal if it has the form $\lambda x_1 \dots \lambda x_n.(ht_1 \dots t_m)$ where $n, m \geq 0$ and where $h, x_1, \dots, x_n$ are tokens, and the terms $t_1, \dots, t_m$ are all in β-normal form. In this case, we call the list $x_1, \dots, x_n$ the *binder*, the token h the *head*, and the list $t_1, \dots, t_m$ the *arguments* of the term.

Exercise 2.1. Not all λ-terms are β-convertible to a term that is β-normal. Of the following terms, determine which is not β-convertible to a β-normal term and which are. In the latter case, compute that normal form.

1. $((\lambda x.y)(\lambda x.x))$
2. $((\lambda x.x)(\lambda x.x))$
3. $((\lambda x.(xx))(\lambda x.x))$
4. $((\lambda x.(xx))(\lambda x.(xx)))$
5. $((\lambda x.y)((\lambda x.(xx))(\lambda x.(xx))))$

Exercise 2.2. Church numerals are the following sequence of closed λ-terms:

$$(\lambda f \lambda x.x) \quad (\lambda f \lambda x.(fx)) \quad (\lambda f \lambda x.(f(fx))) \quad (\lambda f \lambda x.(f(f(fx)))) \quad \dots$$

These terms can be used to encode the natural numbers $0, 1, 2, 3, \dots$. The two λ-terms

$$S = \lambda N \lambda M \lambda f \lambda x.((Nf)(Mfx)) \qquad P = \lambda N \lambda M \lambda f \lambda x.((N(Mf))x)$$

can be used to compute the sum (using S) and product (using P) of two Church numerals. Check this claim by computing the β-normal forms of the following two λ-terms, which encode $2+3$ and 2×3.

$$((S\ (\lambda f.\lambda x.(f(fx))))\ (\lambda f.\lambda x.(f(f(fx)))))$$

$$((P\ (\lambda f.\lambda x.(f(fx))))\ (\lambda f.\lambda x.(f(f(fx)))))$$

Exercise 2.3. (‡) Computing β-normal forms can cause the size of terms to grow quickly. For example, consider the following sequence of λ-terms.

$$\begin{array}{lclll}
E_0 & = & \big(((\lambda g \lambda e.e) & (\lambda e \lambda f(e(ef))))\big) & (\lambda f \lambda x(f(fx)))\big) \\
E_1 & = & \big(((\lambda g \lambda e.(ge)) & (\lambda e \lambda f(e(ef))))\big) & (\lambda f \lambda x(f(fx)))\big) \\
E_2 & = & \big(((\lambda g \lambda e.(g(ge))) & (\lambda e \lambda f(e(ef))))\big) & (\lambda f \lambda x(f(fx)))\big) \\
E_3 & = & \big(((\lambda g \lambda e.(g(g(ge)))) & (\lambda e \lambda f(e(ef))))\big) & (\lambda f \lambda x(f(fx)))\big)
\end{array}$$

The term E_n is the Church numeral encoding n applied twice to the encoding of 2. The β-normal form of E_0 encodes 2 while E_1 reduces to the encoding of 4. What number is encoded by the β-normal form of E_n?

As the previous two exercises show, it is possible to use λ-terms to compute. That observation is often used as a starting point for describing functional programming based on λ-terms. While the dynamics of β-reduction will be important for us here, we shall employ those dynamics in a straightforward fashion: β-reduction will usually be used to instantiate quantified expressions.

Exercise 2.4. (‡) Is there an expression N such that $(\lambda x.w)[N/w]$ is equal to $\lambda y.y$ (modulo α-conversion, of course)? Phrased slightly differently, is there an expression N such that $((\lambda w \lambda x.w)N)$ has $(\lambda y.y)$ as a β-normal form? The expression N may or may not have free occurrences of variables.

2.2 Types

Let $\mathcal{S}$ be a fixed, non-empty set of tokens. The tokens in $\mathcal{S}$ will be used as *primitive types* (also called *sorts*). The set of *types* is the smallest set of expressions that contains the primitive types and is closed under the construction of *arrow types*, denoted by the binary, infix symbol $\rightarrow$. The Greek letters τ and σ are used as syntactic variables ranging over types. The type constructor $\rightarrow$ associates to the right: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

These types are called *simple types.* Such type expressions do not contain binders nor are they polymorphic. Instead, these types are used as *syntactic types* in order to separate expressions of different *syntactic categories.* For example, in Section 13.2, the syntax of the π-calculus is encoded using two primitive types n (for names) and p (for process). The type $n \rightarrow p$ is a syntactic type denoting a name abstraction over a process. This type is not intended to denote all functions from names to processes. Of course, every abstraction of type $n \to p$ does indeed represent a function from names to processes: for example, if $M : n \to p$ and N is a name, then the β-normal form of (MN) is a process (the result of substituting N for the abstracted variable of M). However, there are functions from names to processes that do not correspond to an actual syntactic expression of type $n \to p$: for example, the function that maps a particular name, say a, to the process expression P_1 and all other names to a different process P_2 is not encoded in the syntax as an expression of type $n \to p$.

Let τ be the type $\tau_1 \to \cdots \to \tau_n \to \tau_0$ where $\tau_0 \in \mathcal{S}$ and $n \geq 0$. The types $\tau_1, \ldots, \tau_n$ are the *argument types of* τ while the type τ_0 is the *target type of* τ. If $n = 0$ then τ is τ_0 and the list of argument types is empty. The *order* of a type τ is defined as follows: If τ is primitive then τ has order 0; otherwise, the order of τ is one greater than the maximum order of the argument types of τ. As a recursive definition, the order of a type, written $\mathrm{ord}(\tau)$, can be defined by the following two clauses.

$$\begin{array}{rcl} \mathrm{ord}(\tau) & = & 0 \quad \text{provided } \tau \in \mathcal{S} \\ \mathrm{ord}(\tau_1 \to \tau_2) & = & \max(\mathrm{ord}(\tau_1) + 1, \mathrm{ord}(\tau_2)) \end{array}$$

Note that τ has order 0 or 1 if and only if all the argument types of τ are primitive types.

2.3 Signatures and typed terms

Signatures are used to formally *declare* that certain tokens are assigned a certain type. In particular, a *signature (over S)* is a set Σ (possibly empty) of pairs, written as $x\!:\!\tau$, where τ is a type and x is a token. We require signatures to be *determinate* in the sense that for every token x, if $x : \tau$ and $x : \sigma$ are members of Σ then τ and σ are the same type expression.

A signature Σ is said to have order n if every type associated to a token in Σ has order less than or equal to n. Thus, Σ is a *first-order signature* if whenever $h : \tau$ is a member of Σ, $\mathrm{ord}(\tau) \leq 1$.

A *typing judgment*, $\Sigma \Vdash t : \tau$, relates a signature Σ, a λ-term t, and a type τ. We consider the variables in Σ as being bound over such a judgment. Common inference rules for determining such typing rules are the following.

$$\frac{}{\Sigma, x : \tau \Vdash x : \tau} \qquad \frac{\Sigma \Vdash t : \sigma \to \tau \quad \Sigma \Vdash s : \sigma}{\Sigma \Vdash (t\ s) : \tau} \qquad \frac{\Sigma, x : \tau \Vdash M : \sigma}{\Sigma \Vdash (\lambda x.M) : \tau \to \sigma}$$

In the last inference rule, it is assumed that the bound variable x does not occur in Σ. These three typing rules can be used with terms not in β-normal

$$\frac{\Sigma, x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash t : \tau_0}{\Sigma \Vdash \lambda x_1 \ldots \lambda x_n . t : \tau_1 \to \cdots \to \tau_n \to \tau_0}$$

$$\frac{\Sigma \Vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \Vdash t_n : \sigma_n \qquad h : \sigma_1 \to \cdots \to \sigma_n \to \tau_0 \in \Sigma}{\Sigma \Vdash (h\ t_1\ \cdots\ t_n) : \tau_0}$$

Figure 2.1: Typing judgment for Σ-terms of type τ. Here, both rules are restricted so that $\tau_o \in \mathcal{S}$ and $n \geq 0$. Also, the variables $x_1, \ldots, x_n$ are assumed to not occur in Σ.

form. However, in this book, we shall restrict the typing judgment so that only β-normal formulas are given types. Thus, we shall adopt the inference rules in Figure 2.1 as the official rules for this judgment.

When the judgment $\Sigma \Vdash t{:}\tau$ is provable, we say that t is a Σ-*term of type* τ. Note that if a term is given a type, then that term is β-normal. Furthermore, any term that is given a type is also said to be in *$\beta\eta$-long normal form.* This normal form can be arrived at by first computing the β-normal form, and then applying some η-expansion steps. For example, if $i \in \mathcal{S}$, then the judgment $\Sigma \Vdash \lambda x.x : (i \to i) \to i \to i$ is not provable, but the judgment

$$\Sigma \Vdash \lambda x \lambda y.xy : (i \to i) \to i \to i,$$

based on the η-expanded version of the term, is provable.

Exercise 2.5. (‡) Fix a set of sorts S and a signature Σ over S. Prove that if there are *primitive* types τ and τ' such that $\Sigma \Vdash t{:}\tau$ and $\Sigma \Vdash t{:}\tau'$, then $\tau = \tau'$. Show that this statement is not true if we allow τ and τ' to be non-primitive.

2.4 Formulas

Most descriptions of predicate logic first present *terms* and then present *formulas* as a separate structure that incorporates terms. Following Church [1940], we shall instead define *formulas* as terms of the particular type o (the Greek letter omicron).

When defining the formulas of a given logic (e.g., first-order classical logic), we shall first fix the declaration of the *logical constants.* That signature, which we denote as Σ_{-1} (the signature of the basement), attributes to various tokens types which have target type o.

These logical constants are divided into two groups: propositional constants and quantifiers. The *propositional constants* are given types that only use the primitive type o and that have order 0 or 1. For example, in Chapter 4, the propositional connectives in the formulas for classical and intuitionistic first-order logic are declared by the following signature.

$$\{\boldsymbol{t} : o,\ \boldsymbol{f} : o,\ \wedge : o \to o \to o,\ \vee : o \to o \to o,\ \supset : o \to o \to o\}$$

The binary symbols $\wedge$, $\vee$, and $\supset$ are written as infix operators. For example, the λ-term $((\wedge\ P)\ Q)$ is written in the more common form $(P \wedge Q)$. Also, $\wedge$ and $\vee$ associating to the left and $\supset$ associating to the right and $\wedge$ has higher priority than $\vee$, which has higher priority than $\supset$.

There are two classes of *quantifiers* we consider in this book, namely, $\forall_\tau$, for universal quantification for type τ, and $\exists_\tau$, for existential quantification for type τ. Both $\forall_\tau$ and $\exists_\tau$ are assigned the type $(\tau \to o) \to o$. In principle, there are denumerably infinite many such quantifiers, one for each type τ. The expressions $\forall_\tau(\lambda x.B)$ and $\exists_\tau(\lambda x.B)$ are abbreviated as $\forall_\tau x.B$ and $\exists_\tau x.B$, respectively, or as simply $\forall x.B$ and $\exists x.B$ if the value of the type subscript is not important or can easily be inferred from context. Note that the binding operation of quantification is identified as the binding operation of the underlying λ-calculus.

After fixing the set of logical constants, we generally fix the non-logical symbols by picking another signature Σ_0. Let $c : \tau_1 \to \cdots \to \tau_n \to \tau_0 \in \Sigma_0$, where τ_0 is a primitive type and $n \geq 0$. If τ_0 is o, then c is a *predicate symbol of arity n*. If $\tau_0 \in \mathcal{S}\backslash\{o\}$ (i.e., τ_0 is not o), then c is a *function symbol of arity n*. A $\Sigma_{-1} \cup \Sigma_0$-term of type o is also called a $\Sigma_{-1} \cup \Sigma_0$-*formula*, or more usually either a Σ_0-*formula* (since Σ_{-1} is usually fixed) or just a *formula* (if Σ_0 is understood).

A logic is *propositional* if the only logical connectives it contains are propositional connectives (i.e., no quantifiers). A logic is *first-order* if the only quantifiers allowed in its formulas are contained in the set

$$\{\forall_\tau : (\tau \to o) \to o \mid \tau \in \mathcal{S}\backslash\{o\}\} \cup \{\exists_\tau : (\tau \to o) \to o \mid \tau \in \mathcal{S}\backslash\{o\}\}.$$

The types in this signature are of order 2. The restriction on the type of quantifiers, namely $\tau \in \mathcal{S}\backslash\{o\}$, implies that in a first-order formula, the only quantification is over primitive (and non-formula) types. A logic that provides no restriction on the types used in quantification is a *higher-order logic*.

Assume that Σ_{-1} declares logical connectives for a first-order logic and that Σ_0 is a first-order signature. Let τ be a primitive type different from o. A first-order term t of type τ is either a token of type τ or it is of the form $(f\ t_1 \ldots t_n)$ where f is a function symbol of type $\tau_1 \to \cdots \to \tau_n \to \tau$ and, for $i = 1, \ldots, n$, t_i is a term of type τ_i. In the latter case, f is the head and $t_1, \ldots, t_n$ are the arguments of this term. Similarly, a first-order formula either has a logical symbol as its head, in which case, it is said to be *non-atomic*, or a non-logical symbol at its head, in which case it is *atomic*.

As mentioned above, formulas in both classical and intuitionistic first-order logic make use of the same set of logical connectives, namely, $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\boldsymbol{t}$ (truth), $\boldsymbol{f}$ (absurdity), $\forall_\tau$ (universal quantification over type τ), and $\exists_\tau$ (existential quantification over type τ). The negation of B, sometimes written as $\neg B$, is an abbreviation for the formula $B \supset \boldsymbol{f}$.

The nesting of implications within formulas will prove to be a useful feature of formulas to quantify. We define *clausal order* of formulas using the following recursion on formulas in classical and intuitionistic logic.

$$\begin{array}{rcl}
\text{order}(A) & = & 0 \quad \text{provided } A \text{ is atomic, } \boldsymbol{t}\text{, or } \boldsymbol{f} \\
\text{order}(B_1 \wedge B_2) & = & \max(\text{order}(B_1), \text{order}(B_2)) \\
\text{order}(B_1 \vee B_2) & = & \max(\text{order}(B_1), \text{order}(B_2)) \\
\text{order}(B_1 \supset B_2) & = & \max(\text{order}(B_1) + 1, \text{order}(B_2)) \\
\text{order}(\forall x.B) & = & \text{order}(B) \\
\text{order}(\exists x.B) & = & \text{order}(B)
\end{array}$$

This measure counts the number of times implications are nested to the left of implications. In particular, $\text{order}(\neg B) = \text{order}(B) + 1$. The clausal order of a finite set or multiset of formulas is the maximum clausal order of any formula in that set or multiset. Note the similarity to the way the order of types is given in Section 2.2.

The *polarity of a subformula occurrence* within a formula is defined as follows. If a subformula C of B occurs to the left of an even number of occurrences of implications in B, then C is a *positive* subformula occurrence of B. On the other hand, if a subformula C occurs to the left of an odd number of occurrences of implication in a formula B, then C is a *negative* subformula occurrence of B. More formally:

- B is a positive subformula occurrence of B.
- If C is a positive subformula occurrence of B then C is a positive subformula occurrence in $B \wedge B'$, $B' \wedge B$, $B \vee B'$, $B' \vee B$, $B' \supset B$, $\forall_\tau x.B$, and $\exists_\tau x.B$; C is also a negative subformula occurrence in $B \supset B'$.
- If C is a negative subformula occurrence of B then C is a negative subformula occurrence in $B \wedge B'$, $B' \wedge B$, $B \vee B'$, $B' \vee B$, $B' \supset B$, $\forall_\tau x.B$, and $\exists_\tau x.B$; C is also a positive subformula occurrence in $B \supset B'$.

2.5 Sequents

Proof and provability generally need to be given for a collection of formulas instead of a single, isolated formula. For example, a typical way to describe the provability of the implication $B \supset C$ is to pose the hypothetical judgment involving two formulas: if B then C. The *sequents* introduced by Gentzen [1935] are one way to organize the multiple formulas that are involved in stating a provable statement. In their simplest form, sequents are a pair, written $\Gamma \vdash \Delta$, of the two collections of formula Γ and Δ. Gentzen used $\longrightarrow$ instead of $\vdash$ for building a sequent but we will follow the more traditional approach and use $\vdash$ largely since the arrow notion is used in many other computational-oriented situations (see, for example, Chapter 13). Consider a mathematician's attempt at a proof: at the top of her page, she lists the formulas in Γ as assumptions, and at the bottom of the page, she displays the formula B that is her goal to prove. The sequent $\Gamma \vdash B$, in which there is exactly one formula to the right of the $\vdash$, can be used to encode that state of her proof attempt. More intuition about sequents and logical reasoning will be given in Section 3.1.

Within this book, sequents will vary somewhat in structure: we outline here these variations.

Collections of formulas in sequents will be either lists or multisets or sets. Sequents can also be *one-sided* or *two-sided*. One-sided sequents are usually written as $\vdash \Delta$ and two-sided sequents are usually written as $\Gamma \vdash \Delta$: here, Γ and Δ are one of the three kinds of collections of formulas mentioned above. Sometimes we shall see multiple collections of formulas, separated by a semicolon, on both the left and right sides of sequents; for example, $\Gamma; \Gamma \vdash \Delta; \Delta'$ and $\vdash \Delta; \Delta'$. In the two-sided sequent $\Gamma \vdash \Delta$, we shall say that Γ is this sequent's *antecedent* or *left-hand side* and that Δ is its *succedent* or *right-hand side*. Finally, we will add $\Downarrow$ to certain sequents when we discussed *focused proof systems*: in particular, $\Sigma : \Gamma \Downarrow D \vdash A$ in Section 5.4 and $\Sigma : \Psi; \Delta \Downarrow B \vdash \Gamma; \Upsilon$ in Section 6.7.

The formulas in a sequent are typed, and the signatures that declare the type of the token in those formulas must be clearly specified. As in the previous section, we shall generally assume that once we pick a particular logic (classical, intuitionistic, or linear), we have fixed the signature Σ_{-1}. Furthermore, a set of non-logical constants Σ_0 will often be fixed as well. Finally, the rules that Gentzen gives for the treatment of quantifiers involves the introduction of *eigenvariables*: these variables may appear free in the formulas of some sequents. To properly declare those variables and their types, we shall often prefix a sequent with a signature: for example, $\Sigma : \vdash \Delta$ and $\Sigma : \Gamma \vdash \Delta$. In all these cases, a formula that appears in Δ or Γ must be given type o using the union of the three signatures Σ_{-1}, Σ_0, and Σ.

We note some issues concerning matching expressions with schematic variables. For example, let B denote a formula and let Γ and Γ' denote collections of formulas. Considering what it means to match the expressions B, Γ' and Γ', Γ'' to a given collection, which we assume contains $n \geq 0$ formulas.

- If the given collection is a list, then B, Γ' matches if the list is non-empty and B is the first formula and Γ' is the remaining list. The expression Γ', Γ'' matches if Γ' is some prefix and Γ'' is the remaining suffix of that list: there are $n + 1$ possible matches.

- If the given collection is a multiset then B, Γ' matches if the multiset is non-empty and B is a formula in the multiset and Γ' is the multiset resulting from deleting one occurrence of B. The expression Γ', Γ'' matches if the multiset union of Γ' and Γ'' is Γ: there can be as many as 2^n possible matches since each member of Γ can be placed in either Γ' or Γ''.

- If the given collection is a set then B, Γ' matches if the set is non-empty and B is a formula in the set and Γ' is either the given set or the set resulting from removing B from the set. The expression Γ', Γ'' matches if the set union of Γ' and Γ'' is Γ: there can be as many as 3^n possible matches, since each member of Γ can be placed in either Γ' or Γ'' or in both.

2.6 Bibliographic notes

The approach to specifying terms and formulas in elementary type theory (ETT) is a popular choice in the construction of modern theorem prover systems: for example, ETT is used in the HOL family of provers [Gordon, 2000] as well as in Isabelle [Paulson, 1994], Abella [Baelde et al., 2014], and the logic programming language λProlog [Miller and Nadathur, 2012]. The textbooks Andrews [1986] and Farmer [2023] treat this logic in detail.

For a comprehensive treatments of the untyped λ-calculus, see [Barendregt, 1984], and of the typed λ-calculus, see [Krivine, 1990; Barendregt et al., 2013]. The use of untyped λ-terms here is similar to the so-called “Curry-style” of typed λ-terms: bound variables are not assumed globally to have types but are provided a type when they are initially bound. This approach to typing contrasts that used by Church, where variables have types independently of whether or not they are bound. For more about these different approaches to types in the λ-calculus, see [Pfenning, 2008].

The perspective that (natural deduction) proofs correspond to (dependently) typed λ-terms and that β-reductions correspond to (functional) computation is part of the well known *Curry-Howard correspondence* approach to modeling computation (see [Sørensen and Urzyczyn, 2006]). This approach to computation is not used in this book: instead, we model computation as the search for (cut-free) proofs, an approach that is often referred to as the *proof search approach to computation.*

Richer types than the simple ones introduced in this chapter are indeed useful within logical formulas and logic programming more specifically. For example, the programming language λProlog has a form of polymorphic typing [Nadathur and Pfenning, 1992; Appel and Felty, 2004; Miller and Nadathur, 2012] and the Elf logic programming language (based on the LF logical framework) uses dependently type λ-terms [Pfenning, 1989; Pfenning and Schürmann, 1999].

Chapter 3

Sequents calculus proofs rules

A familiar form of formal proof, often attributed to Frege and Hilbert, accepts certain formulas as *axioms* (e.g., $(p \supset (q \supset p))$ and $(((p \supset q \supset r) \supset (p \supset q) \supset (p \supset r))))$ and certain *inference rules* (e.g., from p and $(p \supset q)$ conclude q). A formal *Frege proof* is a list of formulas such that every formula occurrence in that list is either an axiom or the result of applying an inference rule to previous formulas in the list. Such proof structures are easy to trust: any provable formula (i.e., by appearing in such a list of formulas) must be as trustworthy as the trust one puts into the axioms and inference rules. However, such proof objects have so little structure that it is hard to imagine effective proof search mechanisms for them. In contrast, the notion of sequent calculus proofs provides a much more valuable way of structuring proofs. As we shall see, such proof structures are natural for modeling abstract execution models in the logic programming paradigm.

3.1 Sequent calculus and proof search

The sequent calculus makes at least two significant departures from Frege proofs. First, while inference rules are applied to formulas in Frege proofs, they are applied to sequents—a more complex structure—in the sequent calculus. Second, there are no axioms used within the sequent calculus proof systems we study here: the burden of proof falls entirely on inference rules over sequents.

In Section 2.5, we presented sequents as formal, syntactic structures that contain one or more collections of formulas with an outer layer of variable bindings (denoted by the associated eigenvariable signature). Before formally presenting inference rules in Section 3.2 involving such sequents, we provide an intuitive reading of sequents by providing an informal reading of two-sided sequents in which the right-hand side is a collection containing exactly one occurrence of a formula. Consider, for example, attempting to prove that for every natural number n, the product $n(n+1)$ is even. An informal proof of this fact can be organized as follows. To prove that this is true for all natural numbers, pick some arbitrary number, say, m. Now, m is either even or odd. If m is even, then the product $m(m+1)$ is even. If m is odd, then $m+1$ is even and, again, the produce $m(m+1)$ is even. Hence, in either case, this product is even.

A first step in formalizing this proof would be to identify (and name) three lemmas about natural numbers that this argument accepts as previously proved.

L_1 $\forall n.(\text{even } n) \lor (\text{odd } n)$
L_2 $\forall n.(\text{odd } n) \supset (\text{even } (s\ n))$
L_3 $\forall n, m, p.((\text{even } n) \lor (\text{even } m)) \supset (\text{times } n\ m\ p) \supset (\text{even } p)$

For these lemmas to be proper formulas as defined in the previous chapter, we must assume that the set of sorts contains a primitive type nat $\in \mathcal{S}$ and that the signature of non-logical constants Σ_0 must contain the following declarations:

$$z : \text{nat},\ \ s : \text{nat} \to \text{nat},$$
$$\mathit{even} : \text{nat} \to o,\ \ \mathit{odd} : \text{nat} \to o,\ \ \mathit{times} : \text{nat} \to \text{nat} \to \text{nat} \to o$$

We assume that natural numbers are encoded as z, $(s\ z)$, $(s\ (s\ z))$, etc and that the predicate (times $n\ m\ p$) hold precisely when p is the product $n \times m$. Imagine that we now take a blank sheet of a paper and write at the top the three lemmas that we accept as assumptions and write at the bottom of that sheet the formula $\forall n, p.(\text{times } n\ (s\ n)\ p) \supset (\text{even } p)$. Our task is to fill in the gap between the assumptions at the top and the conclusion at the bottom. A sequent is essentially a representation of the status of that sheet of paper: in this case, that sequent (named T_1) would be

$$T_1 \qquad \cdot; L_1, L_2, L_3 \vdash \forall n, p.(\text{times } n\ (s\ n)\ p) \supset (\text{even } p).$$

The prefix, which is just the dot $\cdot$, is meant to show that there are no variables bound over this particular sequent. One way to make progress on finishing a proof of this sequent is to take a new sheet of paper on which we write the assumptions L_1, L_2, L_3 and (times n $(s\ n)$ p) at the top and write the conclusion (even p) at the bottom of that sheet. Thus, we now have an additional assumption that p is the product $n(n+1)$ and the different conclusion (even p). This new state in the construction of a formal proof is represented by the sequent

$$T_2 \qquad n, p; L_1, L_2, L_3, (\text{times } n\ (s\ n)\ p) \vdash (\text{even } p).$$

Note here that the variables n and p are bound over this sequent. The next step in building proof uses lemma L_1 to add the assumption $(\text{even } n) \lor (\text{odd } n)$.

That is, our sheet of paper now have five formulas at the top: it is encoded as the sequent

$$T_3 \qquad n, p; L_1, L_2, L_3, (\text{times } n\ (s\ n)\ p), (\text{even } n) \vee (\text{odd } n) \vdash (\text{even } p).$$

The case analysis induced by the disjunctive assumption leads the proof to have two subproofs. That is, the current sheet of paper can be replaced by two sheets that are identical except that one of those sheets replaces that disjunction with (even n) and the other sheet replaces it with (odd n). These two sheets are encoded with the two sequents

$$\begin{array}{ll} T_4 & n, p; L_1, L_2, L_3, (\text{times } n\ (s\ n)\ p), (\text{even } n) \vdash (\text{even } p) \\ T_5 & n, p; L_1, L_2, L_3, (\text{times } n\ (s\ n)\ p), (\text{odd } n) \vdash (\text{even } p) \end{array}$$

One way to represent the status of a proof's development is to organize these sequents into the tree

$$\dfrac{\dfrac{\dfrac{T_4 \qquad T_5}{T_3}}{T_2}}{T_1}$$

To complete the formal description of this proof, we need to label each horizontal line by the name of an inference rule. For example, the uppermost horizontal line is justified by the "rule of cases" (also called the $\vee$L rule in Chapter 4). As this tree shows, the process of proving sequent T_1 has reduced it to attempting to prove the two sequent T_4 and T_5.

This proof can be completed by appealing to lemma L_3 to justify sequent T_4 and appealing to lemmas L_2 and L_3 to justify sequent T_5.

Our subsequent study of sequent calculus proofs will not, however, focus on capturing natural or human-readable proofs. Instead, we focus on low-level aspects of proof that will ultimately make it possible to automate proof search for, at least, some fragments of logic. The analysis of sequent calculus proofs by Gentzen and others has led to richer sequents than those motivated above. In particular, a sequent of the form $x, y : B_1, B_2, B_3 \vdash C$ can naturally be linked to the single formula $\forall x \forall y.[(B_1 \wedge B_2 \wedge B_3) \supset C]$. The usual treatment of the sequent calculus also allows for the more general (albeit less intuitive) multiple-conclusion sequent. In particular, the comma on the left can be viewed as a conjunction, while the comma on the right can be viewed as a disjunction. For example, the sequent $x, y : B_1, B_2, B_3 \vdash C_1, C_2$ is linked to the formula $\forall x \forall y.[(B_1 \wedge B_2 \wedge B_3) \supset (C_1 \vee C_2)]$.

3.2 Inference rules

An inference rule in a sequent calculus proof system has a single sequent as its conclusion and zero or more sequents as its premises. Of the numerous

$$\frac{\Sigma : \Gamma, B, C, \Gamma' \vdash \Delta}{\Sigma : \Gamma, C, B, \Gamma' \vdash \Delta}\, xL \qquad \frac{\Sigma : \Gamma \vdash \Delta, B, C, \Delta'}{\Sigma : \Gamma \vdash \Delta, C, B, \Delta'}\, xR$$

$$\frac{\Sigma : \Gamma, B, B \vdash \Delta}{\Sigma : \Gamma, B \vdash \Delta}\, cL \qquad \frac{\Sigma : \Gamma \vdash \Delta, B, B}{\Sigma : \Gamma \vdash \Delta, B}\, cR$$

$$\frac{\Sigma : \Gamma \vdash \Delta}{\Sigma : \Gamma, B \vdash \Delta}\, wL \qquad \frac{\Sigma : \Gamma \vdash \Delta}{\Sigma : \Gamma \vdash \Delta, B}\, wR$$

Figure 3.1: Structural rules.

inference rules used in the various sequent calculi presentations we meet in this book, all inference rules belong to exactly one of the following three broad classes of rules: the *structural rules*, the *identity rules*, and the *introduction rules*. We examine each of these classes separately below by showing examples of each of these classes of rules.

3.2.1 Structural rules

Since sequents describe relationships among formulas, the nature of a formula's context is an important feature of proofs. To analyze the interplay between a formula and its context, it is sometimes desirable to explore the structural differences provided by lists, multisets, and sets. For example, one might want an inference rule to permute items explicitly in a context or to replace two occurrences of the same formula with one occurrence. There are three standard structural rules, called *exchange*, *contraction*, and *weakening*, and they are presented in Figure 3.1 in both left and right side versions. All these structural rules can be used with contexts that are list structures. The exchange rules, *xL* and *xR*, allows exchanging two consecutive elements. This structural rule does not make sense when contexts are multisets or sets. The contraction rules, *cL* and *cR*, can be used on lists and multisets to replace two occurrences of the same formula with one occurrence: this structural rule is not invoked on set contexts. The weakening rules, *wL* and *wR*, can insert a formula into a context. If used with a list, these rules insert the new formula occurrence only at the end of the context. If contexts are sets, the only structural rules that make sense to specify are the weakening rules.

In this book, we shall never use the exchange rules, and contexts will almost always be either multisets or sets.

Exercise 3.1. Let Δ' be a permutation of the list Δ. Show that a sequence of *xR* rules can derive the sequent $\Sigma : \Gamma \vdash \Delta$ from the sequent $\Sigma : \Gamma \vdash \Delta'$.

$$\frac{}{\Sigma : B \vdash B}\ \mathit{init} \qquad \frac{\Sigma : \Gamma \vdash \Delta, B \qquad \Sigma : B, \Gamma' \vdash \Delta'}{\Sigma : \Gamma, \Gamma' \vdash \Delta, \Delta'}\ \mathit{cut}$$

Figure 3.2: The two identity rules: initial and cut.

3.2.2 Identity rules

The identity rules consist of the *initial* rule and the *cut* rule, examples of which are displayed in Figure 3.2. Both of these rules contain repeated occurrences of schema variables: in the initial rule, the variable B is repeated in the conclusion, and in the cut rule, the variable B is repeated in the premises. Checking if an application of one of these rules is correct requires comparing the identity of two occurrences of formulas. While the structural rules address the structure of the contexts used in forming sequents, the identity rules address the meaning of the sequent symbol $\vdash$. In particular, these two rules can be seen as stating that $\vdash$ is reflexive and transitive. In Section 4.2, we illustrate that, in a certain sense, these two rules describe dual aspects of $\vdash$.

Sometimes, an inference rule with zero premises is called an *axiom*. We shall reserve that term for a *formula* that is accepted as the starting point of some forms of proofs (e.g., the Frege proofs describe at the start of this chapter). Since sequents are not formulas, we use other names (e.g., initial sequents) for leaves in sequent calculus proof trees.

3.2.3 Introduction rules

The final group of inference rules contains the *introduction* rules, so called because they introduce one occurrence of a logical connective into the conclusion of the inference rule. In two-side sequent systems, a logical connective is introduced on the left and right by two different, small sets of inference rules. Here, the term "a small collection" means a collection of 0, 1, or 2 rules. (In the informal reading of sequents provided in Section 3.1, a left-introduction rule describes how to reason *from* a logical connective while the right-introduction rule describes how to reason *to* a logical connective.) If the sequent is one-sided, then the left-introduction rules are usually replaced by a right-introduction for the connective that is its De Morgan dual. Thus, one-sided systems are usually limited to those logics where all connectives have De Morgan duals. The only one-sided sequent proof system in this book appears in Chapter 6 when we present linear logic.

Figure 3.3 presents a few examples of introduction rules for some logical connectives. That figure provides two left introduction rules and one right in-

$$\frac{\Sigma : B, \Gamma \vdash \Delta}{\Sigma : B \wedge C, \Gamma \vdash \Delta}\ \wedge\mathrm{L} \qquad \frac{\Sigma : C, \Gamma \vdash \Delta}{\Sigma : B \wedge C, \Gamma \vdash \Delta}\ \wedge\mathrm{L}$$

$$\frac{\Sigma : \Gamma \vdash \Delta, B \qquad \Sigma : \Gamma \vdash \Delta, C}{\Sigma : \Gamma \vdash \Delta, B \wedge C}\ \wedge\mathrm{R} \qquad \frac{}{\Sigma : \Gamma \vdash \Delta, \boldsymbol{t}}\ \boldsymbol{t}\mathrm{R}$$

$$\frac{\Sigma : \Gamma_1 \vdash \Delta_1, B \qquad \Sigma : C, \Gamma_2 \vdash \Delta_2}{\Sigma : B \supset C, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \supset\mathrm{L} \qquad \frac{\Sigma : B, \Gamma \vdash \Delta, C}{\Sigma : \Gamma \vdash \Delta, B \supset C}\ \multimap\mathrm{R}$$

$$\frac{\Sigma \Vdash t : \tau \qquad \Sigma : \Gamma, B[t/x] \vdash \Delta}{\Sigma : \Gamma, \forall_\tau x\ B \vdash \Delta}\ \forall\mathrm{L} \qquad \frac{\Sigma, y : \tau\ :\ \Gamma \vdash \Delta, B[y/x]}{\Sigma : \Gamma \vdash \Delta, \forall_\tau x\ B}\ \forall\mathrm{R}$$

Figure 3.3: Examples of left and right introduction rules.

troduction rule for conjunction, whereas both implication and universal quantification are given one left and one right introduction rule each. There is one right introduction rule and zero left introduction rule for $\boldsymbol{t}$.

Also illustrated in Figure 3.3 is the role that the signature Σ plays in the specification of the quantifier introduction rules. In particular, the introduction of the universal quantifier $\forall$ on the left uses the signature and the judgment $\Sigma \Vdash t : \tau$ to determine the range of suitable substitution terms t. On the other hand, the right introduction rule for $\forall$ changes the signature from $\Sigma \cup \{y : \tau\}$ above the line to Σ below the line. Note that if we were to think of signatures as lists of distinct typed variables, we must maintain that the variable y is not free in any formula in the rule's conclusion. By viewing quantifiers as bindings in formulas and signatures as binders for sequents, the inference rule $\forall R$ essentially allows for the *mobility* of a binder: reading this inference rule from premise to conclusion, the binder for y *moves* from a sequent-level binding to the formula level binding for x. At no point is the binder replaced with a "free variable." Of course, this movement of the binder is only allowed if no occurrences of the bound variable above the line are unbound below the line. Thus, all occurrences of y in the upper sequent must appear in the displayed occurrence of $B[y/x]$. Such a sequent-level bound variable is called an *eigenvariable*. Note that since we identify all binding structures that differ by only an alphabetic change of variables, the $\forall$R rule could also be written as

$$\frac{\Sigma, x : \tau\ :\ \Gamma \vdash \Delta, B}{\Sigma : \Gamma \vdash \Delta, \forall_\tau x\ B}\ \forall\mathrm{R}.$$

In this form, the mobility of the binder for x is more apparent.

The premise $\Sigma \Vdash t : \tau$ for the $\forall$L rule should actually be written as $\Sigma_{-1} \cup \Sigma_0 \cup \Sigma \Vdash t : \tau$ where Σ_{-1} and Σ_0 are the signatures for the logical and non-logical constants, respectively. Since both these signatures are global for any particular proof, we write this condition with only the smaller signature for convenience. Also, one has the choice to either include this typing judgment as a part of the proof (hence, the proof of the typing judgment is a subproof of a proof of the conclusion to this rule) or as a side condition, namely, the requirement that that premise is provable (in this case, the proof of that side condition is not incorporated into the sequent proof).

3.3 Additive and multiplication inference rules

When an inference rule has two premises, there are two natural ways to relate the contexts in the two premises with the context in the conclusion. An inference rule is *multiplicative* if contexts in the premises are merged to form the context in the conclusion. The cut rule in Figure 3.2 and the $\supset$L rule in Figure 3.3 are examples of *multiplicative* rules. A rule is *additive* if the contexts in the premises are the same as the context in the conclusion. The $\wedge$R rule in Figure 3.3 is additive. An additive version of the cut inference rule can be written as

$$\frac{\Sigma : \Gamma \vdash \Delta, B \qquad \Sigma : B, \Gamma \vdash \Delta}{\Sigma : \Gamma \vdash \Delta}\ .$$

The use of the terms multiplicative and additive will be explained when the *exponentials* of linear logic are presented in Section 6.3.2.

Another way to describe the difference between additive and multiplicative rules is the following. We call a formula occurring in the conclusion of an inference rule that is not introduced in that rule a *context formula*. In an additive rule, every occurrence of a context formula in the concluding sequent has an occurrence in *both* premise sequents. In a multiplicative rule, every occurrence of a context formula in the concluding sequent appears in *exactly one* premise sequent. In both of these cases, these occurrences in the conclusion and the premises are always on the same side of their respective sequents.

Another presentation of the introduction rules for conjunction involves the multiplicative form of the $\wedge$R rule.

$$\frac{\Sigma : B, C, \Gamma \vdash \Delta}{\Sigma : B \wedge C, \Gamma \vdash \Delta}\ \wedge\mathrm{L}^m \qquad \frac{\Sigma : \Gamma_1 \vdash \Delta_1, B \qquad \Sigma : \Gamma_2 \vdash \Delta_2, C}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, B \wedge C}\ \wedge\mathrm{R}^m$$

Exercise 3.2. (‡) Show that if the structural rules of weakening and contraction are available, then the rules $\wedge$R and $\wedge$L (from Figure 3.3) can be derived from $\wedge\mathrm{R}^m$ and $\wedge\mathrm{L}^m$, and conversely.

This exercise illustrates that the structural rules allows an additive rule to account for a multiplicative rule, and vice versa. Another connection between these concepts is suggested by the following collection of inference rules.

$$\dfrac{\dfrac{\dfrac{}{\Sigma : B \vdash B}\,\mathit{init} \quad \dfrac{}{\Sigma : B \vdash B}\,\mathit{init}}{\Sigma : B \vdash B \wedge B}\,\wedge\mathrm{R} \quad \dfrac{\Sigma : \Gamma, B, B \vdash \Delta}{\Sigma : \Gamma, B \wedge B \vdash \Delta}\,\wedge\mathrm{L}^m}{\Sigma : \Gamma, B \vdash \Delta}\,\mathit{cut}$$

Thus, if we adopt $\wedge\mathrm{L}^m$ and $\wedge\mathrm{R}$ as the left and right introduction rules for conjunction, we can infer the *cL* rule. Since we do not wish for the contraction rule to enter our proof systems without an *explicit* reference to the contraction rule, then we must chose carefully how we pair left and right introduction rules. Gentzen's original sequent calculus (and the ones we adopt for classical and intuitionistic logics in Chapter 4) paires $\wedge\mathrm{L}$ with $\wedge\mathrm{R}$. When we turn to linear logic in Chapter 6, we will allow for two conjunctions, written as & and $\otimes$, where the right introduction rule for & is the additive rule and the right introduction rule for $\otimes$ is the multiplicative. The left introduction rule for & will be similar to $\wedge\mathrm{L}$ and for $\otimes$ will be similar to $\wedge\mathrm{L}^m$.

It is interesting to comment on the relative costs of naive implementations of additive versus multiplicative binary inference rules. There are two directions for implementing such applications. The *proof building direction* works by being given two premises and building the conclusion. The *proof search direction* works by being given the conclusion and nondeterministically building premises. Applying the proof building direction to a given *additive* inference rule can be expensive since one must check that the context formulas are the same (as multisets or sets) in the two premises: this check on equality of multisets can involve thousands of formulas (at least in the logic programming setting we are targeting). At the same time, the proof search direction is inexpensive for additive rules: given the conclusion, we need to build premises that contain pointers to the same object that forms the contexts in the conclusion. On the other hand, applying the proof building direction to a given *multiplicative* inference rule can be inexpensive since one can build the conclusion by pairing together the pointers to contexts in the premises: there is no need to check equality of context formulas. At the same time, the proof search direction can be expensive since there are exponentially many possible splittings of contexts that may need to be considered.

3.4 Sequent calculus proofs

Given the definitions of formulas and sequents in Chapter 2 and the presentation of inference rules in the previous section, we are can now define proofs, in particular, *sequent calculus proofs*. Unlike terms and sequents, such proof structures do not introduce new notions of bindings. This observation contrasts the usual Curry-Howard correspondence approach, where proofs are identified with natural deduction proofs, which, themselves, are encoded by various kinds of λ-terms.

Assume that a signature of logical constants Σ_{-1} is given and that a collection of inference rules are specified. Derivations and proofs will be represented by finite trees with labeled nodes and edges containing at least one edge. Nodes are labeled by occurrences of inference rules or by two *improper rules*, *open* and *root*. All trees contain exactly one node labeled *root*, called the *root node*. Let N be another node in the tree. The edge leading from N to the root node (via some path of edges) is called its *out-arc* while the other $n \geq 0$ arcs terminating at N are called its *in-arcs*: in this case, n is the *in-degree* of the node N. If N is labeled with *open*, then N must have zero in-arcs. If N is labeled by an occurrence of a proper inference rule, the out-arc must be labeled with the conclusion of the inference rule occurrence, and the in-arcs must be labeled with the premise sequents. Of course, sequent labels are determined to be equal using the rules of λ-expression.

Let **S** be a sequent. A *derivation for* **S** is such a labeled tree in which the in-arc to the root is labeled with **S**. The smallest open derivation for **S** is a tree with two nodes, one labeled with *root* and one labeled with *open* and with the edge between them labeled with **S**. A derivation for **S** without any nodes labeled *open* is a *proof of* **S**. In these cases, the sequent **S** is also called the *endsequent* of the derivation or the proof. Given these definitions, a derivation can also be considered a "partial proof."

When we write derivation trees, leaves with no line over them are taken as ending in an open node. If there is a line, then we assume that that line denotes an inference rule with no premises: in other words, the tree ends with a proper inference rule that has an in-degree of zero.

Assume that we have picked a particular style of sequent (e.g., one-sided, two-sided, etc). By a *proof system* for such sequents, we mean a collection of inference rules for those sequents, such as those described in Section 3.2. Let $\mathcal{X}$ be such a set of rules for two-sided sequent. We write $\Sigma : \Gamma \vdash_{\mathcal{X}} \Delta$ to denote the fact that the sequent $\Sigma : \Gamma \vdash \Delta$ has a proof in $\mathcal{X}$. If Σ is empty, we write just $\Gamma \vdash_{\mathcal{X}} \Delta$. If Γ is also empty, we write $\vdash_{\mathcal{X}} \Delta$. If the proof system is assumed, the subscript $\mathcal{X}$ is not written. Thus, $\vdash \Delta$ will mean that the sequent $\cdot : \cdot \vdash \Delta$ is provable. If a one-sided proof system is used instead, the same conventions apply except that we do not write the left-hand context (keeping just the signature). Note that the $\vdash$ symbol is used in two different ways: it is used to mark a syntactic expression as being a sequent, and it is used to be the proposition that a certain sequent is provable. The reader should be able to always disambiguate between these two senses of the $\vdash$ symbol.

Exercise 3.3. Consider a (trivial) sequent calculus proof system containing just the cut and initial inference rules. Describe what can be proved using just those two rules. Show that every provable sequent can be proved without the cut rule.

3.5 Permutations of inference rules

Sequent calculus inference rules can often be permuted over each other. For example, assume that the following three introduction rules are part of a proof system.

$$\frac{\Sigma : \Gamma_1 \vdash \Delta_1, B \qquad \Sigma : C, \Gamma_2 \vdash \Delta_2}{\Sigma : B \supset C, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2} \supset\!\mathrm{L} \qquad \frac{\Sigma : B, \Gamma \vdash \Delta, C}{\Sigma : \Gamma \vdash \Delta, B \supset C} \multimap\!\mathrm{R}$$

$$\frac{\Sigma : B, \Gamma \vdash \Delta \qquad \Sigma : C, \Gamma \vdash \Delta}{\Sigma : B \vee C, \Gamma \vdash \Delta} \vee\mathrm{L}$$

Here, the left and right-hand contexts are assumed to be multisets. In the first derivation in Figure 3.4, the right introduction rule for implication is below the left introduction of a disjunction. The second derivation in that figure has the same root and leaf sequents but introduction rules are switched. (Note that the latter derivation uses two occurrences of the right introduction of implication while the former proof uses only one occurrence of that rule.)

Sometimes inference rules can be permuted if additional structural rules are employed. For example, consider the first derivation in Figure 3.5. It is possible to switch the order of the two introduction rules it contains, but this requires introducing some weakenings and a contraction, as is witnessed by the second derivation in that figure. If these additional structural rules are not permitted in a given proof system (as we shall see is the case in intuitionistic logic), then the original two inference rules cannot be permuted.

Understanding when inference rules can be permuted over each other can make it possible to improve the effectiveness of searching for proofs. Consider again, for example, the derivations in Figure 3.4. Imagine attempting to find a proof of the sequent $\Sigma : \Gamma, p \vee q \vdash r \supset s, \Delta$ following the development of the first derivation in that figure: namely, we first do an $\multimap$R rule followed by the $\vee$L rule. Additionally, assume that there is, in fact, no proof of the left premise $\Sigma : \Gamma, p, r \vdash s, \Delta$: that is, an exhaustive search fails to find a proof of this sequent. If we employ a naive proof search strategy, we might make another attempt to find a proof of the endsequent by switching the application of the $\vee$L and the $\multimap$R rules. As it is clear from the second derivation, this other order of rule applications will lead to an attempt to prove the same left premise for which we already know no proof exists. Clearly, this particular second attempt at proving this endsequent does not need to be made.

An inference rule asserts that whenever its premises are provable, its conclusion is provable. The converse—that is, if the conclusion is provable then

$$\dfrac{\dfrac{\Sigma:\Gamma,p,r\vdash s,\Delta \qquad \Sigma:\Gamma,q,r\vdash s,\Delta}{\Sigma:\Gamma,p\vee q,r\vdash s,\Delta}\;\vee\mathrm{L}}{\Sigma:\Gamma,p\vee q\vdash r\supset s,\Delta}\;\multimap\mathrm{R}$$

$$\dfrac{\dfrac{\Sigma:\Gamma,p,r\vdash s,\Delta}{\Sigma:\Gamma,p\vdash r\supset s,\Delta}\;\multimap\mathrm{R} \qquad \dfrac{\Sigma:\Gamma,q,r\vdash s,\Delta}{\Sigma:\Gamma,q\vdash r\supset s,\Delta}\;\multimap\mathrm{R}}{\Sigma:\Gamma,p\vee q\vdash r\supset s,\Delta}\;\vee\mathrm{L}$$

Figure 3.4: Two derivations that differ in the order of two inference rules.

$$\dfrac{\dfrac{\Sigma:\Gamma_1,r\vdash\Delta_1,p \qquad \Sigma:\Gamma_2,q\vdash\Delta_2,s}{\Sigma:\Gamma_1,\Gamma_2,p\supset q,r\vdash\Delta_1,\Delta_2,s}\;\supset\mathrm{L}}{\Sigma:\Gamma_1,\Gamma_2,p\supset q\vdash\Delta_1,\Delta_2,r\supset s}\;\multimap\mathrm{R}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma:\Gamma_1,r\vdash\Delta_1,p}{\Sigma:\Gamma_1,r\vdash\Delta_1,p,s}\;wR}{\Sigma:\Gamma_1\vdash\Delta_1,p,r\supset s}\;\multimap\mathrm{R} \qquad \dfrac{\dfrac{\Sigma:\Gamma_2,q\vdash\Delta_2,s}{\Sigma:\Gamma_2,q,r\vdash\Delta_2,s}\;wL}{\Sigma:\Gamma_2,q\vdash\Delta_2,r\supset s}\;\multimap\mathrm{R}}{\Sigma:\Gamma_1,\Gamma_2,p\supset q\vdash\Delta_1,\Delta_2,r\supset s,r\supset s}\;\supset\mathrm{L}}{\Sigma:\Gamma_1,\Gamma_2,p\supset q\vdash\Delta_1,\Delta_2,r\supset s}\;cR$$

Figure 3.5: Two derivations that illustrate the permutation of inference rules supported by structural rules.

all the premises are provable—does not always hold. In the event that this converse does hold for an inference rule, we say that that rule is *invertible.* From the point of view of searching for a proof, whenever invertible introduction rules are available to prove a given sequent, they can be applied in any order and without considering any other order of applying them. One way to show that an inference rule is invertible is to show that for every pair of inference rules for which the rule in question appears above another inference rule, the order of that pair of rules can be switched.

As we shall see, sequent calculus proofs are composed of tiny rules. Also, given a sequent calculus proof of an endsequent, many trivial variants of that proof also exist: permuting inference rules can generate some of them. Also, nothing prevents irrelevant steps to be inserted at almost any point. The unstructured nature of sequent calculus proofs is useful for proving results such as the cut-elimination theorem. But when one wants to apply sequent calculus proof systems to various computer science projects (one of our goals here), we must first attempt to find more structure within such proofs. Ultimately, we shall describe such additional structure by introducing *uniform proofs*: these are greatly constrained cut-free proofs where proof construction is divided into two alternating phases that capture *goal reduction* and *backward chaining*, two operations familiar to those working with logic programs. The notion of uniform proofs will naturally lead to the closely related notion of *focused proofs*: both of these style proofs are introduced in Chapter 5.

3.6 Cut-elimination and its consequences

In the construction of proofs in mathematics, discovering useful lemmas is a key activity. For example, consider again the example from Section 3.1 where the focus was on proving that the product $n(n+1)$ is even for all natural numbers n. The part of the proof of this theorem that we illustrated was particularly simple since we employed the three lemmas L_1, L_2, and L_3. Of course, these three lemmas needed to be discovered and proved. The inference rule called cut in Figure 3.2 is used to formally allow lemmas to be proved and used in a proof. For example, assume that L_1, L_2, and L_3 have sequent calculus proofs Ξ_1, Ξ_2, and Ξ_3, respectively. The following derivation injects those lemmas into the proof of our original theorem, $(\forall n, p.(\text{times } n \ (s\ n)\ p) \supset (\text{even } p))$, which we abbreviate as the formula G.

$$\dfrac{\begin{matrix}\Xi_1\\ \cdot;\cdot\vdash L_1\end{matrix} \qquad \dfrac{\begin{matrix}\Xi_2\\ \cdot;\cdot\vdash L_2\end{matrix} \qquad \dfrac{\begin{matrix}\Xi_3\\ \cdot;\cdot\vdash L_3\end{matrix} \qquad \begin{matrix}\vdots\\ \cdot;L_1,L_2,L_3\vdash G\end{matrix}}{\cdot;L_1,L_2\vdash G}\ \mathit{cut}}{\cdot;L_1\vdash G}\ \mathit{cut}}{\cdot;\cdot\vdash G}\ \mathit{cut}$$

Thus, these instances of the cut-rule allow us to move from searching for a proof of G to searching for a proof of G using L_1, L_2, and L_3.

For all of the sequent calculus proof systems we consider in this book, the *cut-elimination theorem* holds: that is, a sequent has a proof if and only if it has a *cut-free proof* (a proof with no occurrences of the cut rule). We shall prove two cut-elimination theorems in subsequent chapters: Section 5.5 provides one for intuitionistic logic, and Section 6.8 presents one for linear logic. This central theorem of sequent calculus proof systems has several consequences, some of which we describe below.

The consistency of a logic is usually a simple consequence of cut-elimination. For example, if a formula B and its negation $B \supset \boldsymbol{f}$ are provable, then the sequents $\cdot \vdash B$ and $\cdot \vdash B \supset \boldsymbol{f}$ are provable. Since the rule for introducing implication on the right is invertible (as we shall see in Section 4.4), it must be the case that the sequent $B \vdash \boldsymbol{f}$ is provable. By applying the cut inference rule to proofs of the two sequents $\cdot \vdash B$ and $B \vdash \boldsymbol{f}$ yields a proof of $\cdot \vdash \boldsymbol{f}$. By the cut-elimination theorem, however, the sequent $\cdot \vdash \boldsymbol{f}$ has a proof without cuts. Thus, the last inference rule of this proof must be either an introduction rule or a structural rule. Generally, there is no introduction rule for $\boldsymbol{f}$ on the right. Also, the structural rules will not yield a provable sequent either. Thus, there can be no cut-free proof of $\cdot \vdash \boldsymbol{f}$, and hence a formula and its negation cannot both be provable.

The success of proving the cut-elimination theorem also signals that certain aspects of the logic's proof system were well designed. For example, in two-sided sequents, logical connectives generally have left-introduction and right-introduction rules. If we think of a sequent as describing a sheet of paper with the assumptions listed at the top of the page and the conclusion at the bottom of the page, then the left and right introduction rules yield two *senses* to how connectives are used within a proof. In particular, the left-introduction rules describe how we argue *from* formulas while the right-introduction rules describe how we argue *to* formulas. For example, the $\multimap$R rule in Figure 3.3 describes how one uses hypothetical reasoning to prove the formula $B \supset C$ while the $\supset$L rule shows that we use $B \supset C$ as an assumption by attempting a proof of B and by attempting the original sequent again, but this time with the additional assumption C added to the set of hypotheses. Of course, if we consider the model-theoretic semantics of the connectives, they usually have only one *sense*: for example, $B \wedge C$ is true if and only if B and C are true. The cut-elimination theorem implies that the two senses attributed to a logical connective work together to define one logical connective. We return to this aspect of cut-elimination in Sections 4.2 and 5.6.

When formulas involve only first-order quantification, a formula occurring in a sequent in a cut-free proof is always a subformula of some formula of the endsequent. This invariant is the so-called *subformula property* of cut-free proofs. When searching for a proof, one needs only to choose and rearrange subformulas of the endsequent: of course, instantiations of quantified expressions must also be considered as subformulas. In the first-order setting, all proper subformulas of a given formula have fewer occurrences of logical connectives and quantifiers. Thus, having proofs restricted to arrangements of subformulas is an interesting and powerful restriction. However, in the higher-order setting, instantiating a predicate variable can result in larger formulas with many more occurrences of logical connectives and quantifiers. In that setting, the subformula property fails, even for cut-free proofs.

When one attempts to use the sequent calculus to formalize proofs of mathematically interesting theorems, one discovers that the cut rule is used a great deal. Eliminating cut in such a proof would necessarily yield a huge and low-level proof where all lemmas are "in-lined" and reproved at every instance of their use. Cut-free proofs can thus be very big objects. For example, if one uses the number of nodes in a proof as a measure of its size, there are cases where cut-free proofs are hyperexponentially bigger than proofs allowing cut (see Exercise 2.3 for a similar explosive growth). Thus, sequents with proofs of rather small size can have cut-free proofs that require more inference rules than the number of sub-atomic particles in the universe. If a cut-free proof is actually computed and stored in some computer memory, the theorem that that proof proves is likely to be *mathematically uninteresting*. This observation does not disturb us here since we are interested in cut-free proofs as tools for describing *computation* only. For us, cut-free proofs are not illuminating and readable proofs but structures more akin to the classic notion of Turing machines configurations: they provide a low-level and detailed trace of a computation.

Recording a computation as a cut-free proof can be superior to recording Turing machine configurations, since there are several deep ways to reason with actual proof structures. For example, assume that we have a cut-free proof of the two-sided sequent $\mathcal{P} \vdash G$ for some logic, say, $\mathcal{X}$. As we shall see, in many approaches to proof search, it is natural to identify the left-hand context $\mathcal{P}$ to the specification of a (logic) program and G as the goal or query to be established. A cut-free proof of such a sequent is then a trace that this goal can be established from this program. Now assume that we can prove $\mathcal{P}' \vdash^{+} \mathcal{P}$ where $\mathcal{P}'$ is some other logic program and $\vdash^{+}$ is provability in $\mathcal{X}^{+}$ which is some strengthening of $\mathcal{X}$ in which, say, induction principles are added (as well as cut). If the stronger logic satisfies cut elimination, then we know that $\mathcal{P}' \vdash G$ has a cut-free proof in the stronger logic $\mathcal{X}^{+}$. If things have been organized well, it can then become a simple matter to see that cut-free proofs of such sequents do not make use of the stronger proof principles, and, hence, $\mathcal{P}' \vdash G$ has a cut-free proof in $\mathcal{X}$. Thus, using cut-elimination, we have been able to move from a *formal* proof about programs $\mathcal{P}$ and $\mathcal{P}'$ immediately to the conclusion that whatever goals can be established for $\mathcal{P}$ can be established for $\mathcal{P}'$. Clearly, the ability to do such direct, logically principled reasoning about programs and computations should be an interesting aspect of the proof search paradigm to explore.

3.7 Bibliographic notes

In this chapter, we have presented a broad overview of sequent calculus proof systems. In subsequent chapters, starting with the next one, we will present specific sequent calculus proof systems. These proof systems will have a cut-elimination theorem: we shall prove the cut-elimination theorem for a couple of them using techniques that are not standard. There are several good references for the more standard approaches that cover such results for logics other than classical, intuitionistic, and linear logics. Gentzen's original proof [1935] is a good place to read about proving this result for classical and intuitionistic logics. For more modern presentations, see [Gallier, 1986; Girard et al., 1989; Negri and von Plato, 2001; Bimbó, 2015]. Mechanized approaches to proving cut-elimination can be found in [Pfenning, 2000; Miller and Pimentel, 2013].

Kleene [1952] presents a detailed analysis of permutability of inference rules for classical and intuitionistic sequent systems.

Statman [1978] showed that there exist a sequence $H_0,\ H_1,\ H_2, \ldots$ of theorems of first-order classical logic such that the size of H_n and of a sequent calculus proof (with cut) of H_n is linear in n, while the size of the shortest cut-free proof of H_n is *hyperexponential* in n. Here, the *hyperexponential function* can be defined as $h(0) = 1$ and $h(n + 1) = 2^{h(n)}$.

Chapter 4

Classical and intuitionistic logics

Classical and intuitionistic logics provide foundations to many formal systems used in computational logic, including interactive and automatic theorem proving, logic programming, model checking, programming language type systems, and formalized versions of mathematics. We shall assume that the reader has some elementary familiarity with these two logics.

There are several formal ways to describe the difference between these two logics. Two well-known ways to characterize their differences are the following.

1. Intuitionistic logic results from admitting only those proofs that can be seen as providing *constructive evidence* of what is proved. Classical logic admits these proofs as well as many others that do not need to be constructive. For example, the axiom of the *excluded middle* is an accepted proof principle in classical logic.

2. The semantics of intuitionistic logic is based on *possible world semantics* or *Kripke models* [Kripke, 1965; Troelstra and van Dalen, 1988], in which classical logic models are arranged to a tree structure and where the truth value of implication at a given world relies on truth values in all reachable worlds.

Gentzen provided a different characterization entirely, and it involves the role of structural inference rules within the sequent calculus. This characterization plays an essential role in this book: in fact, a careful reading of Gentzen's characterization will help us motivate the introduction of linear logic in Chapter 6.

This chapter presents sequent calculus proofs for classical and intuitionistic logics that are small variations on Gentzen's *LK* and *LJ* proof systems

[Gentzen, 1935]. After presenting some basic properties of those proof systems, we highlight some issues that arise when systematically searching for proofs in those proof systems.

Exercise 4.1. (‡) Prove that there are irrational numbers, a and b such that a^b is rational. An easy, non-constructive proof starts with the observation that $\sqrt{2}^{\sqrt{2}}$ is either rational or irrational (an instance of the excluded middle). Complete that proof. Can you provide a constructive proof of this statement?

4.1 Classical and intuitionistic inference rules

Both intuitionistic and classical logics will use the same connectives: in particular, the signature of logical connectives, Σ_{-1}, for both of these logics is

$$\{\boldsymbol{f} : o, \boldsymbol{t} : o, \wedge : o \to o \to o, \vee : o \to o \to o, \supset : o \to o \to o\} \cup$$
$$\{\forall_\tau : (\tau \to o) \to o, \exists_\tau : (\tau \to o) \to o\}_{\tau \in \mathcal{S} \setminus \{o\}}$$

Here, the set of primitive types S is assumed to be fixed and to contain the type o. Note that if we use $\{o\}$ for S, then this signature does not contain any quantifiers and is, therefore, the signature for a propositional logic.

To provide a proof system for provability in classical and intuitionistic logics, we use sequents of the form $\Sigma : \Gamma \vdash \Delta$, where both Γ and Δ are multisets of Σ-formulas. The introduction, identity, and the structural rules for this proof system are given in Figure 4.1, 4.2, and 4.3, respectively. Of the four inference rules with two premises, $\supset$L and *cut* are multiplicative rules while $\wedge$R and $\vee$L are additive.

The left and right introduction rules for $\boldsymbol{t}$ and $\boldsymbol{f}$ can be derived from the binary connective for which they are the unit. In particular, the $\wedge$R has two premises for the binary connective. The n-ary generalization of the $\wedge$R will have n premises. Since $\boldsymbol{t}$ is the unit for $\wedge$, we can interpret it as the 0-ary conjunction. Thus, the $\boldsymbol{t}$R rule has 0 premises. Furthermore, the n-ary version of the $\wedge$L rule has n instances, one for each of its n conjuncts. Thus, there is no left-introduction rule for $\boldsymbol{t}$ since it is the 0-ary version of $\wedge$. A similar but dual argument illustrates how to derive the introduction rules for $\boldsymbol{f}$ from the rules for $\vee$.

Provability in *classical logic* is given using the notion of a **C**-proof, which is any proof using inference rules in Figure 4.1, Figure 4.2, and Figure 4.3. Provability in *intuitionistic logic* is given using the notion of an **I**-proof, which is any **C**-proof in which the right-hand side of all sequents contain exactly one formula. A proof system that can only use such restricted sequents is called a *single-conclusion proof system*. When no such restriction is imposed on sequents (as in **C**-proofs), such a proof system is called a *multiple-conclusion proof system*.

$$\frac{\Sigma : B, \Gamma \vdash \Delta}{\Sigma : B \wedge C, \Gamma \vdash \Delta}\ \wedge\mathrm{L} \qquad \frac{\Sigma : C, \Gamma \vdash \Delta}{\Sigma : B \wedge C, \Gamma \vdash \Delta}\ \wedge\mathrm{L} \qquad \frac{}{\Sigma : \Gamma \vdash \Delta, \boldsymbol{t}}\ \boldsymbol{t}\mathrm{R}$$

$$\frac{\Sigma : B, \Gamma \vdash \Delta \qquad \Sigma : C, \Gamma \vdash \Delta}{\Sigma : B \vee C, \Gamma \vdash \Delta}\ \vee\mathrm{L} \qquad \frac{\Sigma : \Gamma \vdash \Delta, B \qquad \Sigma : \Gamma \vdash \Delta, C}{\Sigma : \Gamma \vdash \Delta, B \wedge C}\ \wedge\mathrm{R}$$

$$\frac{}{\Sigma : \Gamma, \boldsymbol{f} \vdash \Delta}\ \boldsymbol{f}\mathrm{L} \qquad \frac{\Sigma : \Gamma \vdash \Delta, B}{\Sigma : \Gamma \vdash \Delta, B \vee C}\ \vee\mathrm{R} \qquad \frac{\Sigma : \Gamma \vdash \Delta, C}{\Sigma : \Gamma \vdash \Delta, B \vee C}\ \vee\mathrm{R}$$

$$\frac{\Sigma \Vdash t : \tau \qquad \Sigma : \Gamma, B[t/x] \vdash \Delta}{\Sigma : \Gamma, \forall_\tau x\ B \vdash \Delta}\ \forall\mathrm{L} \qquad \frac{\Sigma, c : \tau : \Gamma \vdash \Delta, B[c/x]}{\Sigma : \Gamma \vdash \Delta, \forall_\tau x\ B}\ \forall\mathrm{R}$$

$$\frac{\Sigma, c : \tau : \Gamma, B[c/x] \vdash \Delta}{\Sigma : \Gamma, \exists_\tau x\ B \vdash \Delta}\ \exists\mathrm{L} \qquad \frac{\Sigma \Vdash t : \tau \qquad \Sigma : \Gamma \vdash \Delta, B[t/x]}{\Sigma : \Gamma \vdash \Delta, \exists_\tau x\ B}\ \exists\mathrm{R}$$

$$\frac{\Sigma : \Gamma_1 \vdash \Delta_1, B \qquad \Sigma : C, \Gamma_2 \vdash \Delta_2}{\Sigma : B \supset C, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \supset\mathrm{L} \qquad \frac{\Sigma : B, \Gamma \vdash \Delta, C}{\Sigma : \Gamma \vdash \Delta, B \supset C}\ \multimap\mathrm{R}$$

Figure 4.1: Introduction rules.

$$\frac{}{\Sigma : B \vdash B}\ \mathit{init} \qquad \frac{\Sigma : \Gamma_1 \vdash \Delta_1, B \qquad \Sigma : B, \Gamma_2 \vdash \Delta_2}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

Figure 4.2: Identity rules.

$$\frac{\Sigma : \Gamma \vdash \Delta}{\Sigma : \Gamma, B \vdash \Delta}\ wL \qquad \frac{\Sigma : \Gamma \vdash \Delta}{\Sigma : \Gamma \vdash \Delta, B}\ wR$$

$$\frac{\Sigma : \Gamma, B, B \vdash \Delta}{\Sigma : \Gamma, B \vdash \Delta}\ cL \qquad \frac{\Sigma : \Gamma \vdash \Delta, B, B}{\Sigma : \Gamma \vdash \Delta, B}\ cR$$

Figure 4.3: Structural rules: contraction and weakening.

Let Σ be a given first-order signature over the primitive types in S, let Δ and Γ be a finite multisets of Σ-formulas, and let B be a Σ-formula. We write $\Sigma; \Delta \vdash_C \Gamma$ if the sequent $\Sigma : \Delta \vdash \Gamma$ has a **C**-proof. We write $\Sigma; \Delta \vdash_I B$ if the sequent $\Sigma : \Delta \vdash B$ has an **I**-proof.

The restriction on **I**-proofs (that all sequents in the proof have singleton right-hand sides) implies that **I**-proofs do not contain occurrences of structural rules on the right (i.e., no occurrences of *cR* and *wR*) and that every occurrence of the $\supset$L rule and the *cut* rule are instances of the following two inference rules.

$$\frac{\Sigma : \Gamma_1 \vdash B \qquad \Sigma : C, \Gamma_2 \vdash E}{\Sigma : B \supset C, \Gamma_1, \Gamma_2 \vdash E} \supset\mathrm{L} \qquad \frac{\Sigma : \Gamma_1 \vdash B \qquad \Sigma : B, \Gamma_2 \vdash E}{\Sigma : \Gamma_1, \Gamma_2 \vdash E} \; cut$$

(That is, the formula on the right-hand side of the conclusion must move to the right premise and not to the left premise.) These observations can give an alternative characterization of **I**-proofs.

The following proposition is easily proved by induction on the structure of sequent calculus proofs.

Proposition 4.2. *Let Ξ be a **C**-proof of $\Sigma : \Gamma \vdash B$. Then Ξ is an **I**-proof if and only if Ξ contains no occurrences of either cR or wR and, in every occurrence in Ξ of an $\supset$L and a cut rule, the right-hand side of the conclusion is the same as the right-hand side of the right premise.*

Proof. The forward direction is immediate. Thus, assume that the **C**-proof Ξ of $\Sigma : \Gamma \vdash B$ satisfies the two conditions of the converse. We proceed by induction on the structure of proofs. Consider the last inference rule of Ξ. If that rule is an instance of either the *init*, $\boldsymbol{t}$R, or $\boldsymbol{f}$L rule, the conclusion is immediate. Otherwise, if that last inference rule is an instance of either $\supset$L or *cut* then, given the inductive restrictions, the premises have proofs satisfying the same restrictions, namely that the two premises are single-conclusion sequents. Thus, by the inductive assumption, the proofs of the premises must be **I**-proofs. If the last rule of Ξ is any other inference rule (the *wR* and *cR* rules are not possible), the inductive argument holds trivially. □

This alternative characterization of **I**-proofs as restricted **C**-proofs prefigures two important features of linear logic (Chapter 6). The first condition (on the absence of *wR* and *cR*) means that the contexts used to describe intuitionistic logic are *hybrid*: the left-hand-side of sequents allow the structural rules while right-hand-side of sequents do not allow structural rules. This kind of hybrid use of contexts will be exploited in richer ways in linear logic. The second condition means that there is something special hidden in the intuitionistic implication and, as we shall see in Section 6.5, that special feature is captured by the ! *exponential* of linear logic.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{B \vdash B}\ init}{B \vdash B, \boldsymbol{f}}\ wR}{\vdash B, B \supset \boldsymbol{f}}\ \multimap\mathrm{R}}{\vdash B, B \vee (B \supset \boldsymbol{f})}\ \vee\mathrm{R}}{\vdash B \vee (B \supset \boldsymbol{f}), B \vee (B \supset \boldsymbol{f})}\ \vee\mathrm{R}}{\vdash B \vee (B \supset \boldsymbol{f})}\ cR$$

Figure 4.4: A **C**-proof of the excluded middle.

One difference we have with Gentzen's formulation of sequent calculus is that he had *negation* as a logical connective. Here, when we write the negation of a formula, $\neg B$, we shall mean $B \supset \boldsymbol{f}$. In Section 4.5, we return to these two different treatments of negation.

A formula of the form $B \vee \neg B$ is an example of an *excluded middle*: in terms of truth values, B is either true or false, and there is no third possibility. Figure 4.4 contains a **C**-proof for this formula. A slight variation of this proof yields a **C**-proof of $B \vee (B \supset C)$ for any formulas B and C.

Exercise 4.3. (‡) Provide proofs for each of the following sequents. If an **I**-proof exists, present that proof. Assume that the signature for non-logical constants is $\Sigma_0 = \{p : o,\ q : o,\ r : i \to o,\ s : i \to i \to o,\ a : i,\ b : i\}$.

1. $(p \wedge (p \supset q) \wedge (p \wedge q \supset r)) \supset r$
2. $(p \supset q) \supset (\neg q \supset \neg p)$
3. $(\neg q \supset \neg p) \supset (p \supset q)$
4. $p \vee (p \supset q)$
5. $((p \supset q) \supset p) \supset p$
6. $(r\ a \wedge r\ b \supset q) \supset \exists x(r\ x \supset q)$
7. $\exists y \forall x(r\ x \supset r\ y)$
8. $\forall x \forall y(s\ x\ y) \supset \forall z(s\ z\ z)$

Exercise 4.4. Take the formulas in Exercise 4.3 which have **C**-proofs but no **I**-proof and reorganize them into **I**-proofs in which appropriate instances of the excluded middle are added to the left-hand context. For example, give an **I**-proof of the sequent

$$\Sigma\ :\ r\ a \vee \neg r\ a\ \vdash (r\ a \wedge r\ b \supset q) \supset \exists x(r\ x \supset q).$$

Exercise 4.5. (‡) Let A be an atomic formula. Describe all pairs of formulas $\langle B, C\rangle$ where B and C are different members of the set

$$\{A, \neg A, \neg\neg A, \neg\neg\neg A\}$$

such that $B \vdash C$ has a **C**-proof. Make the same list such that $B \vdash C$ has an **I**-proof.

Exercise 4.6. The multiplicative version of $\wedge$R is the inference rule

$$\frac{\Sigma \ : \ \Gamma_1 \ \vdash B, \Delta_1 \qquad \Sigma \ : \ \Gamma_2 \ \vdash C, \Delta_2}{\Sigma \ : \ \Gamma_1, \Gamma_2 \ \vdash B \wedge C, \Delta_1, \Delta_2} \ .$$

Show that a sequent has a **C**-proof (resp. **I**-proof) if and only if it has one in the proof system that results from replacing $\wedge$R with the multiplicative version. Similarly, consider the multiplicative version of the $\vee$L rule, namely,

$$\frac{\Sigma \ : \ B, \Gamma_1 \ \vdash \Delta_1 \qquad \Sigma \ : \ C, \Gamma_2 \ \vdash \Delta_2}{\Sigma \ : \ B \vee C, \Gamma_1, \Gamma_2 \ \vdash \Delta_1, \Delta_2} \ .$$

Show that a sequent has a **C**-proof if and only if it has a **C**-proof where the additive $\vee$L is replaced with this multiplicative rule.

The notion of provability based on sequents given in this section is not equivalent to the more usual presentations of classical and intuitionistic logic [Fitting, 1969; Gentzen, 1935; Prawitz, 1965; Troelstra, 1973] in which signatures are not made explicit, and substitution terms (the terms used in $\forall$L and $\exists$R) are not constrained to be built from such signatures. The following example illustrates the main reason they are not equivalent. Let S be the set $\{i, o\}$ of two sorts and let Σ_0, the signature of non-logical constants, be just $\{p : i \to o\}$. Now consider the sequent

$$\cdot : \forall_i x\,(p\ x) \vdash \exists_i x\,(p\ x).$$

This sequent has no proof even though $\exists_i x\,(p\ x)$ follows from $\forall_i x\,(p\ x)$ in the traditional presentations of classical and intuitionistic logics. The reason for this difference is that there are no $\{p : i \to o\}$-terms of type i: that is, the type i is *empty* in this signature. Thus we need the following additional definition. The signature Σ *inhabits* the set of primitive types S if for every $\tau \in \mathcal{S}$ different than o, there is a Σ-term of type τ. When Σ inhabits S, the notion of provability defined above coincides with the more traditional presentations.

Exercise 4.7. (‡) Assume that the set of sorts S contains the two tokens i and j and that the only non-logical constant is $f : i \to j$. In particular, assume that there are no constants of type i declared in the non-logical signature. Is there an **I**-proof of

$$(\exists_j x\, \boldsymbol{t}) \vee (\forall_i y \exists_j x\, \boldsymbol{t}).$$

Under the same assumption, does the formula

$$(\exists_j x\, \boldsymbol{t}) \vee (\forall_i x\, \boldsymbol{f})$$

have a **C**-proof? An **I**-proof? What comparison can you draw between proving this formula and the formula in Exercise 4.3(4)?

The structural rule of weakening allows for adding a formula into the left or right side of sequents (reading the inference rule from premise to conclusion). A strengthening rule is an inference rule that allows for deleting a formula from either the left or right side of a sequent. In general, strengthening is not an admissible rule. The following exercise provides a simple instance of when strengthening is possible.

Exercise 4.8. Show that if there is a **C**-proof (resp., an **I**-proof) of $\Sigma : \Gamma, \boldsymbol{t} \vdash \Delta$ then there is a **C**-proof (an **I**-proof) of $\Sigma : \Gamma \vdash \Delta$.

As we noted at the beginning of this chapter, there are many ways to describe the difference between classical and intuitionistic logic. The following exercise contains yet another way to present this difference.

Exercise 4.9. (‡) Consider adding the following rule (taken from [Gabbay, 1985])

$$\frac{\Sigma : \Gamma \vdash B}{\Sigma : \Gamma \vdash C}\ \textit{restart}$$

to **I**-proofs. This rules has the proviso that on the path from the occurrence of this rule to the root of the proof, there is a sequent with B as its succedent. The spirit of this rules is that during the search for a proof of single-conclusion sequents, one can ignore the right-hand side of a sequent (here, C) and restart an attempt to prove a previous right-hand side (here, B). Such a restart would be useful during proof search if the previous occurrence of B was in a sequent whose left-hand side was different from Γ. Prove that a formula has a **C**-proof if and only if it has an **I**-proof with the restart rule added.

4.2 The identity rules and their elimination

As it turns out, almost all forms of the identity rules can be eliminated from proofs without losing completeness in both classical and intuitionistic logics. In particular, all cuts can be eliminated and all initial rule involving non-atomic formulas can be eliminated.

An occurrence of the initial rule of the form $\Sigma : B \vdash B$ is an *atomic initial rule* if B is an atomic formula. A proof is *atomically closed* if every occurrence of the initial rule in it is an atomic initial rule. In classical and intuitionistic logic, we can restrict the initial rule to be atomic initial rules only.

Proposition 4.10. *If a sequent has a **C**-proof (resp, an **I**-proof) then it has a **C**-proof (resp, an **I**-proof) in which all occurrence of the init rule are atomic initial rules.*

Proof. The theorem follows if we prove that every sequent of the form $B \vdash B$ has a proof containing only atomic initial rules. We proceed by induction on the structure of B. Consider the cases where B is of the form $B_1 \supset B_2$ and of the form $\forall x_\tau . Bx$ and consider the following two derivations.

$$\dfrac{\dfrac{B_1 \vdash B_1 \qquad B_2 \vdash B_2}{B_1, B_1 \supset B_2 \vdash B_2}\,\supset\mathrm{L}}{B_1 \supset B_2 \vdash B_1 \supset B_2}\,\multimap\mathrm{R} \qquad \dfrac{\dfrac{\Sigma, y:\tau : By \vdash By}{\Sigma, y:\tau : \forall x_\tau . Bx \vdash By}\,\forall\mathrm{L}}{\Sigma : \forall x_\tau . Bx \vdash \forall x_\tau . Bx}\,\forall\mathrm{R}$$

Clearly, in these two cases, one instance of an initial rule can be replaced by other instances of the initial rule involving smaller formulas. By applying the inductive hypothesis on the premises of these derivations completes the proof for these cases. We leave the remaining cases to the reader to complete. □

The fact that the initial rules involving non-atomic formulas can be replaced by introduction rules and initial rules on subformulas is an important and desirable property of a proof system. In general, however, atomic initial rules cannot be removed from proofs. Atoms are built from non-logical constants, such as predicates and function systems, and their meaning comes from outside logic. In particular, it is via non-logical symbols and atomic formulas that we shall eventually specify *logic programs* to sort lists, represent transition systems, etc. Atoms are the plugs for programmers to impact the development of proofs (we turn our attention to logic programs in the next chapter).

The following inference rule resembles the cut rule but at the level of terms.

$$\dfrac{\Sigma \Vdash t : \tau \qquad \Sigma, x : \tau : \Delta \vdash \Gamma}{\Sigma : \Delta[t/x] \vdash \Gamma[t/x]}\ \mathit{subst}$$

The following exercise states that this rule is admissible.

Exercise 4.11. Let Ξ be a **C**-proof (resp., **I**-proof) of $\Sigma, x : \tau : \Gamma \vdash \Delta$ and let t be a Σ-term. The result of substituting t for the bound variable x in this sequent and the corresponding bound variables to x is all other sequents in Ξ yields a **C**-proof (resp., **I**-proof) Ξ' of the sequent $\Sigma : \Gamma[t/x] \vdash \Delta[t/x]$. The arrangement of inference rules in Ξ and in Ξ' are the same.

The cut rule can also be restricted to atomic formulas, although it is more complex to prove that restriction. For example, consider the follow occurrence of the cut rule.

$$\dfrac{\overset{\Xi_1}{\Sigma : \Gamma_1 \vdash B, \Delta_1} \qquad \overset{\Xi_2}{\Sigma : \Gamma_2, B \vdash \Delta_2}}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

To argue that this cut can be eliminated, we need to consider the many cases that might arise when examining the last inference rule in both the Ξ_1 and Ξ_2 subproofs. Ultimately, we hope to rewrite the proof displayed above into another proof of the same endsequent in which the last inference rule is no longer the cut rule. We highlight here only those cases where the last inference rule in Ξ_1 is the right-introduction rule for B and Ξ_2 is the left-introduction rule for B.

Consider a proof that contains the following cut with a conjunctive formula in which the two occurrences of that conjunction are immediately introduced in the two subproofs to cut.

$$\dfrac{\dfrac{\overset{\Xi_1}{\Sigma : \Gamma_1 \vdash A_1, \Delta_1} \qquad \overset{\Xi_2}{\Sigma : \Gamma_1 \vdash A_2, \Delta_1}}{\Sigma : \Gamma_1 \vdash A_1 \wedge A_2, \Delta_1}\ \wedge\mathrm{R} \qquad \dfrac{\overset{\Xi_3}{\Sigma : \Gamma_2, A_i \vdash \Delta_2}}{\Sigma : \Gamma_2, A_1 \wedge A_2 \vdash \Delta_2}\ \wedge\mathrm{L}}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

Here, i is either 1 or 2. This derivation can be rewritten to

$$\dfrac{\overset{\Xi_i}{\Sigma : \Gamma_1 \vdash A_i, \Delta_1} \qquad \overset{\Xi_3}{\Sigma : \Gamma_2, A_i \vdash \Delta_2}}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}.$$

In the process of reorganizing the proof in this manner, either Ξ_1 or Ξ_2 is discarded, and the new occurrence of cut is on a subformula of $A_1 \wedge A_2$.

Consider a proof which contains the following cut on an implicational formula and where the two occurrences of that implication are immediately introduced in the two premises of the cut.

$$\dfrac{\dfrac{\overset{\Xi_1}{\Sigma : \Gamma_1, A_1 \vdash A_2, \Delta_1}}{\Sigma : \Gamma_1 \vdash A_1 \supset A_2, \Delta_1}\ \multimap\mathrm{R} \qquad \dfrac{\overset{\Xi_2}{\Sigma : \Gamma_2 \vdash A_1, \Delta_2} \qquad \overset{\Xi_3}{\Sigma : \Gamma_3, A_2 \vdash \Delta_3}}{\Sigma : \Gamma_2, \Gamma_3, A_1 \supset A_2 \vdash \Delta_2, \Delta_3}\ \supset\mathrm{L}}{\Sigma : \Gamma_1, \Gamma_2, \Gamma_3 \vdash \Delta_1, \Delta_2, \Delta_3}\ \mathit{cut}$$

This derivation can be rewritten to

$$\dfrac{\dfrac{\overset{\Xi_2}{\Sigma : \Gamma_2 \vdash A_1, \Delta_2} \qquad \overset{\Xi_1}{\Sigma : \Gamma_1, A_1 \vdash A_2, \Delta_1}}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, A_2}\ \mathit{cut} \qquad \overset{\Xi_3}{\Sigma : \Gamma_3, A_2 \vdash \Delta_3}}{\Sigma : \Gamma_1, \Gamma_2, \Gamma_3 \vdash \Delta_1, \Delta_2, \Delta_3}\ \mathit{cut}$$

In the process of reorganizing the proof in this manner, the cut on $A_1 \supset A_2$ is replaced by two instances of cut, one on A_1 and the other one A_2.

Consider a proof that contains the following cut with $\boldsymbol{t}$ in which the premise where $\boldsymbol{t}$ is on the right-hand side is proved with the $\boldsymbol{t}$R.

$$\dfrac{\dfrac{}{\Sigma : \Gamma_1 \vdash \boldsymbol{t}, \Delta_1}\ \boldsymbol{t}\mathrm{R} \qquad \begin{matrix}\Xi \\ \Sigma : \Gamma_2, \boldsymbol{t} \vdash \Delta_2\end{matrix}}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

This proof can be changed to remove this occurrence of cut entirely as follows. First, the proof Ξ of $\Sigma{:}\Gamma_2, \boldsymbol{t} \vdash \Delta_2$ can be rewritten to the proof Ξ' of $\Sigma{:}\Gamma_2 \vdash \Delta_2$ by removing the occurrence of $\boldsymbol{t}$ in the endsequent and, hence, all the other occurrences of $\boldsymbol{t}$ that can be traced to that occurrence. (See Exercise 4.8.) Furthermore, Ξ' can be transformed to a proof Ξ'' of $\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2$ by simply adding weakening rules to it. The proof Ξ'' contains one fewer instances of the cut-rule than the original displayed proof above.

Consider a proof that contains the following cut with $\forall$ in which the two occurrences of that quantifier are immediately introduced in the two subproofs to cut.

$$\dfrac{\dfrac{\begin{matrix}\Xi_1 \\ \Sigma, x : \Gamma_1 \vdash Bx, \Delta_1\end{matrix}}{\Sigma : \Gamma_1 \vdash \forall x.Bx, \Delta_1}\ \forall\mathrm{R} \qquad \dfrac{\begin{matrix}\Xi_2 \\ \Sigma : \Gamma_2, Bt \vdash \Delta_2\end{matrix}}{\Sigma : \Gamma_2, \forall x.Bx \vdash \Delta_2}\ \forall\mathrm{L}}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

Here, t is a Σ-term. By Exercise 4.11, the proof Ξ_1 of $\Sigma, x : \Gamma_1 \vdash Bx, \Delta_1$ can be transformed into a proof Ξ_1' of $\Sigma : \Gamma_1 \vdash Bt, \Delta_1$ (notice that x is not free in any formula of Γ_1 and Δ_1 nor in the abstraction B). The above instance of cut can now be rewritten as

$$\dfrac{\begin{matrix}\Xi_1' \\ \Sigma : \Gamma_1 \vdash Bt, \Delta_1\end{matrix} \qquad \begin{matrix}\Xi_2 \\ \Sigma : \Gamma_2, Bt \vdash \Delta_2\end{matrix}}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

Exercise 4.12. Repeat the above rewriting of cut inference rules when the cut formula is $\boldsymbol{f}$, a disjunction, or an existential quantifier.

The above rewriting of cut rules suggests that each of the logical connectives, in isolation, have been given the appropriate left and right introduction rules. As mentioned in Section 3.6, each logical connective is given two senses: introduction on the right provides the means to prove a logical connective; introduction on the left provides the means to argue from a logical connective as an assumption. The cut-elimination procedure (partially described above) and the non-atomic-initial-sequent elimination procedure provide some of the justification that these two senses are describing the same connective.

Exercise 4.13. Define a new binary logical connective, written $\diamond$, giving it the left introduction rules for $\wedge$ but the right introduction rules for $\vee$. Can cut be eliminated from proofs involving $\diamond$? Can *init* be restricted to only atomic formulas? This connective is the "tonk" connective of Prior [1960].

Theorem 4.14 (Cut-elimination)**.** *If a sequent has a $\boldsymbol{C}$-proof (respectively, $\boldsymbol{I}$-proof) then it has a cut-free $\boldsymbol{C}$-proof (respectively, $\boldsymbol{I}$-proof).*

While we will not prove this theorem here, we will prove cut-elimination theorems for *focused* versions of sequent calculi: see the proofs of cut-elimination for a fragment of intuitionistic logic (Theorem 5.26) and for all of linear logic (Theorem 6.42). For now, we point out some issues related to proving such cut-elimination results as Theorem 4.14.

Sometimes cuts can be permuted locally although they cannot be eliminated globally. Consider adding to sequent calculus a *definition* mechanism for propositional formulas (the restriction to propositional formulas is only to simplify the presentation). Specifically, let $\mathcal{D}$ be a finite set of definitions which are pairs $A := B$ of a propositional letter A and a propositional formula B. Also add to the proof system in Section 4.1 the following two introduction rules for defined atoms (assuming that the definition $A := B$ is a member of $\mathcal{D}$).

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \vdash \Delta}\ \mathit{defL} \qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A}\ \mathit{defR}$$

Note that locally, the cut rule interacts well with these two introduction rules. For example, if the cut formulas in the premise of a cut rule are immediately introduced by these definition rules, we can have the following derivation.

$$\frac{\dfrac{\Gamma_1 \vdash \Delta_1, B}{\Gamma_1 \vdash \Delta_1, A}\ \mathit{defR} \qquad \dfrac{\Gamma_2, B \vdash \Delta_2}{\Gamma_2, A \vdash \Delta_2}\ \mathit{defL}}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

The cut rule can be applied to the premises of *defR* and *defL* as follows.

$$\frac{\Gamma_1 \vdash \Delta_1, B \qquad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

In this case, one instance of cut on the atomic formula A is replaced by another instance of cut on the possibly larger formula B. Without further restrictions on the class of formulas allowed in definitions, cuts cannot be eliminated. A logic extended with definitions can be inconsistent, as the following exercise illustrates.

Exercise 4.15. (‡) Let p be a non-logical constant of type o (a propositional constant). Let $\mathcal{D}$ contain just the definition $p := (p \supset \boldsymbol{f})$. Show how it is possible to write a cut-free proof for both $p \vdash \boldsymbol{f}$ and $\vdash p$. [Hint: the *cR* rule is needed.] As a consequence, there is a proof with cut of $\vdash \boldsymbol{f}$. Describe what happens when one attempts to eliminate the cut in this proof of $\boldsymbol{f}$.

We mentioned in Section 3.2.2 that the initial and cut rules can be seen as expressing dual aspects of $\vdash$. To illustrate that, let Σ be some signature and let $\mathcal{T}$ be the set of formula $\{B \supset B \mid B \text{ is a } \Sigma\text{-term}\}$. The *init* rule can be used to prove all members of $\mathcal{T}$. On the other hand, the *cut* rule can be seen as using members of this set as an assumption. In particular, a cut-inference rule can be replaced with an $\supset$L rule as follows.

$$\frac{\Sigma : \Gamma \vdash \Delta, B \qquad \Sigma : B, \Gamma' \vdash \Delta'}{\Sigma : \Gamma, \Gamma' \vdash \Delta, \Delta'}\ \mathit{cut} \qquad \frac{\Sigma : \Gamma \vdash \Delta, B \qquad \Sigma : B, \Gamma' \vdash \Delta'}{\Sigma : B \supset B, \Gamma, \Gamma' \vdash \Delta, \Delta'}\ \supset\text{L}$$

As a result of this observation, it is easy to see that a proof of $\Sigma : \Gamma \vdash \Delta$ can easily be converted to a cut-free proof of $\Sigma : \mathcal{T}', \Gamma \vdash \Delta$, where $\mathcal{T}'$ is a finite subset of $\mathcal{T}$.

The following example provides a simple illustration that shows that a proof with cuts can be small while a cut-free proof of the same endsequent must be much larger. Fix the non-logical signature to be $\{a : i, f : i \rightarrow i, p : i \rightarrow o\}$. The notation $(f^n\ t)$ denotes the term that result from n applications of f to the term t: i.e., $(f\ (f\ \ldots\ (f\ t)\ldots))$, where there are n occurrences of f applied to t. Clearly, the sequent $p\ a, \forall x(p\ x \supset p\ (f\ x)) \vdash p(f^n a)$ is provable for all $n \geq 0$. Let $\mathcal{P}$ be the multiset $\{p\ a, \forall x(p\ x \supset p\ (f\ x))\}$. For example, the following cut-free proof proves that $p(f(f(f\ a)))$ is a consequence of $\mathcal{P}$.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\mathcal{P} \vdash pa} \qquad \cfrac{}{\mathcal{P}, p(fa) \vdash p(fa)}}{\mathcal{P}, pa \supset p(fa) \vdash p(fa)}}{\mathcal{P} \vdash p(fa)}\ \dagger \qquad \cfrac{}{\mathcal{P}, p(f^2a) \vdash p(f^2a)}}{\mathcal{P}, p(fa) \supset p(f^2a) \vdash p(f^2a)}}{\mathcal{P} \vdash p(f^2a)}\ \dagger \qquad \cfrac{}{\mathcal{P}, p(f^3a) \vdash p(f^3a)}}{\mathcal{P}, p(f^2a) \supset p(f^3a) \vdash p(f^3a)}}{\mathcal{P} \vdash p(f^3a)}\ \dagger$$

The key inference steps in this proof, marked with † involve *cL* and $\forall$L. This style of proof could be generalized so that proving $p(f^n a)$ involves n instances of this combination of rules.

Exercise 4.16. Show that the shortest cut-free **I**-proof of $\mathcal{P} \vdash p(f^n a)$ has height that is linear in n.

Exercise 4.17. (‡) Show that it is possible to have proofs with *cut* of $p(f^{2^n} a)$ from $\mathcal{P}$ whose height is linear in n instead of in 2^n (as in the style of proof above). Do this by proving a series of lemmas in the construction of that proof.

A consequence of these two exercises is the fact that cut can yield (at least) exponentially shorter proofs.

4.3 Logical equivalence

Two Σ-formulas B and C are *equivalent*, written as $B \equiv C$, in classical (resp., intuitionistic) logic if the two sequents $\Sigma : B \vdash C$ and $\Sigma : C \vdash B$ are provable in classical (resp., intuitionistic) logic. Clearly, if two formulas are equivalent in intuitionistic logic, they are equivalent in classical logic. The converse is, however, not true. For example, $p \vee (p \supset q)$ is classically equivalence to $(p \supset p) \vee q$ but these are not equivalence in intuitionistic logic. The same holds for the pair of formulas $\forall x.(rx \supset p)$ and $(\exists x.rx) \supset p$.

Equivalences can be used to rewrite one logical formula to another logical formula so that equivalence is maintained. Thus, algebraic-style reasoning can be done on formulas. Sequences of rewritings provide a flexible way to prove equivalences without the explicit need to use the sequent calculus.

A common way to define the replacement of a subformula occurrence within a formula is to introduce a syntax such as $\mathcal{C}[A]$ and to think of $\mathcal{C}[\Box]$ as a formula with possibly several occurrences of the hole $\Box$. In that setting, if the formulas C and D can be written as $\mathcal{C}[A]$ and $\mathcal{C}[B]$, respectively, then we say that D results from replacing zero or more occurrences of the subformula A in C with D. A simple and more formal definition, however, is offered by the inductive definition given by the proof system in Figure 4.5. Let C and D be Σ-formulas. We say that D arises from replacing zero or more subformula occurrences of A in C with the formula B if $\Sigma : C \bowtie D$ is provable. Note that we use Σ as a binding mechanism for variables in the same style as we used Σ to bind eigenvariables in sequents.

Proposition 4.18. *Let A and B be Σ-formulas such that $A \equiv B$ in classical (resp., intuitionistic) logic. If $\Sigma : C \bowtie D$ is provable using the rules in Figure 4.5, then $C \equiv D$ in classical (resp., intuitionistic) logic.*

Proof. Let A and B be Σ-formulas and assume that assume that $A \equiv B$ in, say, intuitionistic logic. Hence both $\Sigma : A \vdash B$ and $\Sigma : B \vdash A$ have **I**-proofs. Also assume that $\Sigma : C \bowtie D$ is provable using the inference rules in Figure 4.5. The proof of this proposition follows from a straightforward induction on the structure of such proofs. We illustrate with one case. Assume that the last rule involved implications: thus, C is $C' \supset C''$ and D is $D' \supset D''$ and we know that $\Sigma : C' \bowtie D'$ and $\Sigma : C'' \bowtie D''$. The proof that $\Sigma : C' \supset C'' \vdash D' \supset D''$ is built with the following derivation

$$\dfrac{\dfrac{\Sigma : D' \vdash C' \quad \Sigma : C'' \vdash D''}{\Sigma : C' \supset C'', D' \vdash D''}\supset\mathrm{L}}{\Sigma : C' \supset C'' \vdash D' \supset D''}\supset\mathrm{R}$$

$$\frac{}{\Sigma : C \bowtie C} \qquad \frac{\Sigma : C \bowtie E \quad \Sigma : D \bowtie F}{\Sigma : C \wedge D \bowtie E \wedge F} \qquad \frac{\Sigma : C \bowtie E \quad \Sigma : D \bowtie F}{\Sigma : C \vee D \bowtie E \vee F}$$

$$\frac{\Sigma : C \bowtie E \quad \Sigma : D \bowtie F}{\Sigma : C \supset D \bowtie E \supset F} \qquad \frac{x : \tau, \Sigma : C \bowtie D}{\Sigma : \forall_\tau x.C \bowtie \forall_\tau x.D} \qquad \frac{x : \tau, \Sigma : C \bowtie D}{\Sigma : \exists_\tau x.C \bowtie \exists_\tau x.D}$$

$$\frac{}{\Sigma : A \bowtie B}\ \dagger$$

Figure 4.5: The inductive definition of how to replace some occurrences of A with B within a formula. The proviso † requires that A and B are Σ-formulas. Note that C, D, E, and F are schematic variables quantified per inference rule while A and B are given and fixed formulas.

and with the proofs that are guaranteed by the proofs of $\Sigma : C' \bowtie D'$ and $\Sigma{:}C'' \bowtie D''$. This case also holds for the other connectives and if we substitute classical for intuitionistic provability. □

We shall occasionally use such reasoning by logical equivalence, but we shall not incorporate equivalences into inference rules within our sequent calculus proof systems.

4.4 Invertible introduction rules

As defined in Section 3.5, an inference rule is *invertible* if whenever its conclusion is provable, all of its premises are provable.

Proposition 4.19. *The inference rules* $\boldsymbol{t}$*R,* $\vee$*L,* $\wedge$*R,* $\boldsymbol{f}$*L,* $\forall$*R,* $\exists$*L, and* $\multimap$*R from Figure 4.1 are invertible.*

Proof. The invertibility of $\boldsymbol{t}$R and $\boldsymbol{f}$L is immediate. We indicate how to prove the invertibility of $\multimap$R here: the invertibility of the other inference rules is proved similarly.

To show that the $\multimap$R rule is invertible, assume that $\Sigma : \Gamma \vdash \Delta, B \supset C$ has a **C**-proof Ξ. Given Proposition 4.10, we can assume that Ξ is atomically closed. We now proceed by induction on the structure of Ξ by considering all cases for its the last inference rule (which cannot be initial). In the case that this last inference rule is $\vee$L, then Ξ is of the form

$$\frac{\overset{\Xi_1}{\Sigma : \Gamma, P \vdash B \supset C, \Delta} \quad \overset{\Xi_2}{\Sigma : \Gamma, Q \vdash B \supset C, \Delta}}{\Sigma : \Gamma, P \vee Q \vdash B \supset C, \Delta}\ \vee\text{L}.$$

By the inductive hypothesis applied to Ξ_1 and Ξ_2, the sequents $\Sigma : \Gamma, P \vdash B \supset C, \Delta$ and $\Sigma : \Gamma, Q \vdash B \supset C, \Delta$ must have proofs that introduce the corresponding occurrences of $B \supset C$: let Ξ_1' and Ξ_2', respectively, be the proofs of the corresponding premises of these occurrences of $\multimap$R. The proof

$$\dfrac{\dfrac{\overset{\Xi_1'}{\Sigma : \Gamma, P, B \vdash C, \Delta} \qquad \overset{\Xi_2'}{\Sigma : \Gamma, Q, B \vdash C, \Delta}}{\Sigma : \Gamma, P \vee Q, B \vdash C, \Delta}\ \vee\mathrm{L}}{\Sigma : \Gamma, P \vee Q \vdash B \supset C, \Delta}\ \multimap\mathrm{R}$$

is a proof of the same sequent above but with $\multimap$R as its last inference: in other words, we have managed to permute an occurrence of $\multimap$R over $\vee$L into an an occurrence of $\vee$L over $\multimap$R. In order to treat the case where the last inference rule of Ξ is the cut rule, consider the following collection of inference rules (in which the signature for sequents is dropped).

$$\dfrac{\overset{\Xi_1}{\Gamma_1 \vdash E, \Delta_1} \quad \dfrac{\overset{\Xi_2}{E, \Gamma_2, B \vdash \Delta_2, C}}{E, \Gamma_2 \vdash \Delta_2, B \supset C}\ \multimap\mathrm{R}}{\Gamma_1, \Gamma_2 \vdash B \supset C, \Delta_1, \Delta_2}\ \mathit{cut} \longrightarrow \dfrac{\dfrac{\overset{\Xi_1}{\Gamma_1 \vdash E, \Delta_1} \quad \overset{\Xi_2}{E, \Gamma_2, B \vdash \Delta_2, C}}{\Gamma_1, \Gamma_2, B \vdash C, \Delta_1, \Delta_2}\ \mathit{cut}}{\Gamma_1, \Gamma_2 \vdash B \supset C, \Delta_1, \Delta_2}\ \multimap\mathrm{R}$$

$$\dfrac{\dfrac{\overset{\Xi_1}{\Gamma_1, B \vdash E, \Delta_1, C}}{\Gamma_1 \vdash E, \Delta_1, B \supset C}\ \multimap\mathrm{R} \quad \overset{\Xi_2}{E, \Gamma_2 \vdash \Delta_2}}{\Gamma_1, \Gamma_2 \vdash B \supset C, \Delta_1, \Delta_2}\ \mathit{cut} \longrightarrow \dfrac{\dfrac{\overset{\Xi_1}{E, \Gamma_1, B \vdash \Delta_1, C} \quad \overset{\Xi_2}{\Gamma_1 \vdash E, \Delta_2}}{\Gamma_1, \Gamma_2, B \vdash C, \Delta_1, \Delta_2}\ \mathit{cut}}{\Gamma_1, \Gamma_2 \vdash B \supset C, \Delta_1, \Delta_2}\ \multimap\mathrm{R}$$

Thus, we can permute an instance of $\multimap$R above *cut* to an instance of *cut* above $\multimap$R, no matter which premise the formula $B \supset C$ was placed. All other cases for the last inference rule in Ξ can be treated similarly. □

By invoking uses of structural rules, it is sometimes possible to permute additional instances of inference rules. For example, Figure 3.5 illustrates that an occurrence of $\supset$R above $\supset$L can be permuted so that these two rules are swapped. In order to perform that permutation, the structural rules of *wL*, *wR*, and *cR* are used: thus, such a permutation might not generally be possible within **I**-proofs.

It is possible to use cut-elimination to prove the invertibility of some introduction rules within cut-free proofs. For example, let Ξ be a **C**-proof of $\Gamma \vdash B \supset C, \Delta$ and consider the following proof involving both Ξ and the *cut* inference rule.

$$\dfrac{\dfrac{\overset{\Xi}{\Gamma \vdash B \supset C, \Delta} \quad \dfrac{\dfrac{}{B \vdash B}\ \mathit{init} \quad \dfrac{}{C \vdash C}\ \mathit{init}}{B, B \supset C \vdash C}\ \supset\mathrm{L}}{\Gamma, B \vdash C, \Delta}\ \mathit{cut}}{\Gamma \vdash B \supset C, \Delta}\ \supset\mathrm{R}.$$

If we apply the cut-elimination procedure to this proof, only inference rules above the cut are affected: in particular, the result of eliminating the cut will yield a proof that ends with the introduction of $B \supset C$. In this way, we have used cut elimination to transform Ξ into a proof that immediately introduces an occurrence of $\supset$, thereby proving the invertibility of $\supset$R

Exercise 4.20. (‡) Repeat the argument above to prove the invertibility of $\vee$L, $\wedge$R, $\forall$R, and $\exists$L.

4.5 Negation, false, and minimal logic

Our formalization of classical provability using **C**-proofs is essentially the same as Gentzen's use of the *LK* proof system. One important difference between our presentation of classical logic here and that used by Gentzen is that Gentzen chose not to use the units $\boldsymbol{t}$ and $\boldsymbol{f}$ within his sequent calculi. In particular, Gentzen's sequent system treats negation as a logical connective, meaning, of course, that he provided left and right introduction rules for negation, namely,

$$\frac{\Gamma \vdash B, \Delta}{\neg B, \Gamma \vdash \Delta}\ \neg L \quad \text{and} \quad \frac{\Gamma, B \vdash \Delta}{\Gamma \vdash \neg B, \Delta}\ \neg R.$$

These inference rules cannot be added directly to **I**-proofs since the $\neg L$ rule is inconsistent with the requirement that there is *exactly one* formula on the right-hand side. Gentzen's intuitionistic proof system *LJ* is defined as a restriction on *LK* in which all sequents have *at most one* formula on the right. With that restriction, $\neg L$ can be used whenever the concluding sequent has an empty right-hand side. Instances of *wR* can also appear in Gentzen's version of *LJ* proofs.

Exercise 4.21. *Minimal logic* is sometimes defined as intuitionistic logic without the *ex falso quodlibet* rule: from false, anything follows. Formally, we define an **M**-proof as an **I**-proof in which the $\boldsymbol{f}$L rule does not appear. Since $\boldsymbol{f}$L is the only inference rule for $\boldsymbol{f}$ in Figure 4.1, $\boldsymbol{f}$ is not treated as a logical connective within **M**-proofs. In particular, let B be a formula, and let q be a non-logical symbol of type o that does not occur in B. Let B' be the result of replacing all occurrences of $\boldsymbol{f}$ in B with q. Show that B has an **M**-proof if and only if B' has an **I**-proof.

The following lemma shows that the *ex falso quodlibet* inference rule is admissible in **I**-proofs.

Lemma 4.22. *If Ξ is an **I**-proof of $\Sigma : \Gamma \vdash \boldsymbol{f}$ then for any Σ-formula B, there is an **I**-proof Ξ' that has the same structure as Ξ but which proves $\Sigma : \Gamma \vdash B$.*

Proof. The proof is by induction on the structure of Ξ. Essentially, a few occurrences of $\boldsymbol{f}$ on the right of sequents are changed to B. Ultimately, an occurrence of a leaf sequent of the form $\Gamma', \boldsymbol{f} \vdash \boldsymbol{f}$ is converted to $\Gamma', \boldsymbol{f} \vdash B$. Another way to view this transformation of Ξ to Ξ' is to consider permuting the following cut up into the left premise.

$$\dfrac{\dfrac{\Xi}{\Gamma \vdash \boldsymbol{f}} \qquad \dfrac{}{\boldsymbol{f} \vdash B}\,\boldsymbol{f}\mathrm{L}}{\Gamma \vdash B}\,\mathit{cut}$$

□

We can now show that Gentzen's original *LJ* proof system, in which negation is a logical connective and where *wR* can appear, can be emulated directly by **I**-proofs. Formally, define a **G**-proof as a **C**-proof in which the rules for negation above are allowed and where the right-hand side of sequents are restricted to have at most one formula. We now show that every **G**-proof can be directly translated to an **I**-proof in which negation is replaced by "implies false". To this end, define the mapping $(B)^\circ$ that replaces every occurrence of $\neg C$ in B with $C \supset \boldsymbol{f}$. Similarly, we extend this function to multisets of formulas: $(\Gamma)^\circ = \{(B)^\circ \mid B \in \Gamma\}$. Finally, we further extend this mapping to work on sequents, as follows:

$$(\Gamma \vdash \Delta)^\circ = \begin{cases} (\Gamma)^\circ \vdash (\Delta)^\circ & \text{if } \Delta \text{ is not empty} \\ (\Gamma)^\circ \vdash \boldsymbol{f} & \text{if } \Delta \text{ is empty} \end{cases}$$

Clearly, the image of a sequent in a **G**-proof is a sequent with exactly one formula in the right-hand side.

Proposition 4.23. *Every* **G***-proof of the sequent* $\Sigma : \Gamma \vdash \Delta$ *can be converted to an* **I***-proof of the sequent* $\Sigma : (\Gamma)^\circ \vdash (\Delta)^\circ$.

Proof. All identity and introduction rules other than those for negation translate immediately from **G**-proofs to **I**-proofs. The case for negation rules is simple as well:

$$\dfrac{\Gamma \vdash B}{\neg B, \Gamma \vdash \cdot}\,\neg L \quad \longrightarrow \quad \dfrac{(\Gamma)^\circ \vdash (B)^\circ \qquad \dfrac{}{\boldsymbol{f} \vdash \boldsymbol{f}}\,\boldsymbol{f}\mathrm{L}}{(B)^\circ \supset \boldsymbol{f}, (\Gamma)^\circ \vdash \boldsymbol{f}}\,\supset\!\mathrm{L}$$

$$\dfrac{\Gamma, B \vdash \cdot}{\Gamma \vdash \neg B}\,\neg R \quad \longrightarrow \quad \dfrac{(\Gamma)^\circ, (B)^\circ \vdash \boldsymbol{f}}{(\Gamma)^\circ \vdash (B)^\circ \supset \boldsymbol{f}}\,\neg R$$

The only non-trivial change in proofs results when the **G**-proof ends with *wR*. In that case, the **G**-proof inference rule

$$\dfrac{\Gamma \vdash \cdot}{\Gamma \vdash B}\,wR$$

would allow us to conclude that the translation of the upper sequent, i.e., $(\Gamma)^\circ \vdash \boldsymbol{f}$ has an **I**-proof. By Lemma 4.22, we can conclude that $(\Gamma)^\circ \vdash (B)^\circ$ has an **I**-proof. $\square$

Thus, we can translate away Gentzen's use of negation in such a way that the role of wR in his LJ system can be absorbed into the $\boldsymbol{f}$L rule. As a result, we have a proof system—namely, **I**-proofs—for intuitionistic logic that has neither weakening nor contraction on the right. This observation is helpful for motivating the design of linear logic in Chapter 6. Thus, **I**-proofs (and the proof system for linear logic) will have the *ex falso quodlibet* rule while not having wR: the **G**-proof system, on the contrary, has both the *ex falso quodlibet* rule and the wR rule.

4.6 Choices to consider during the search for proofs

While Gentzen's original calculus is a good setting to prove the elimination of the cut rule (and, hence, also prove consistency), the direct application of that calculus to computational tasks is problematic for several reasons. Since we will be considering the search for proofs as a computation model, we now examine the many choices that are present when searching for a proof. We shall look for possible means to reduce some choices even if such reductions make proofs less amendable for mathematical (i.e., not automated) proof. The many choices in how one searches for sequent calculus proofs can be characterized as follows.

- It is always possible to apply the cut rule to any sequent. In that case, we need to produce a cut-formula (lemma) to prove on one branch and to use as an assumption on the other.
- The structural rules of contractions and weakening can always be applied to make additional copies of a formula or to remove formulas.
- There may be many non-atomic formulas in a sequent, and we can generally apply an introduction rule for every one of these formulas.
- One can also make the choice to check if a given sequent is initial.

Some of these choices produce sub-choices. For example, choosing the cut rule requires finding a cut-formula; choosing $\vee$R requires selecting a disjunct; choosing $\wedge$L requires selecting a conjunct; choosing $\forall$L or $\exists$R requires choosing a term t to instantiate a quantifier, and using the $\supset$L or *cut* rules require splitting the surrounding multiset contexts into pairs (for which there can be exponentially many splits).

All this freedom in searching for proofs is not, however, needed, and greatly reducing the sets of choices can still result in complete proof procedures. Most of the choices above can be addressed as follows.

- Given the cut-elimination theorem, we do not need to consider the cut rule and the problem of selecting a cut-formula. Such a choice forces us to move into a domain where proofs are more like computation traces than witnesses of mathematical arguments (see the discussion in Section 3.6). But since our goal here is the specification of computation, we shall generally live with this choice.

- Often, structural rules can be built into inference rules. For example, weakening can be delayed until the leaves of a proof and it can be built into the *init* rule. Also, instead of attempting to split the contexts when applying the $\supset$L rule, we can use the contraction rule to duplicate all the formulas and then place one copy on the left branch and one copy on the right branch.

- The problem of determining appropriate substitution terms in the $\forall$L and $\exists$R rules is a serious problem whose solution falls outside our investigations here. When systems based on proof search are implemented, they generally make use of various techniques, such as employing so-called *logic variables* and *unification* to determine instantiation terms lazily. Although such techniques are completely standard, we shall not discuss them here.

- While there is be significant nondeterminism involved in choosing among many possible introduction rules, that nondeterminism can generally be classified as either *don't-know nondeterminism*—where choices might need to be undone in order to find a complete proof and *don't-care nondeterminism*—where choices do not need to be undone.

Examples of don't-care nondeterminism are *invertible rules* (as defined in Section 3.5). Applying such invertible introduction rules does not lose completeness. While non-invertible introduction rules represent genuine choices (i.e., don't-know nondeterminism) in the search for proofs, we will provide in the next chapter some structure to those choices as well.

4.7 Bibliographic notes

In his 1935 paper, Gentzen introduced natural deduction. His plan in that paper was to use natural deduction to show that proofs in intuitionistic and classical logics can be *analytic*, i.e., that they can be limited to being free of lemmas. Although it seems clear that Gentzen knew how to use natural deduction to prove this result for intuitionistic logic [Plato and Gentzen, 2008], he did not see how to use natural deduction to prove this same result for classical logic. As a result, Gentzen invented the sequent calculus, and, in that setting, he was able to provide a single cut-elimination procedure that worked for both logics. From what we have illustrated in this chapter, it is not surprising that natural deduction has not served as a unifying framework for these two logics since (1) an important difference between sequent calculus proofs for classical and intuitionistic logics is the presence or absence of contraction and weakening on the right, and (2) natural deduction does not support those structural rules since the conclusion of a natural deduction proof is always a single formula (even when applied to classical logic).

There are many well-known proofs for cut-elimination for proof systems such as the one given by Figures 4.1, 4.2, and 4.3. For the detailed proofs of such cut-elimination theorems, see Gentzen's original paper [1935] as well as more modern treatments available in [Gallier, 1986, Chapter 6], [Girard et al., 1989, Chapter 13], [Negri and von Plato, 2001], and [Bimbó, 2015].

In [Girard et al., 1989, Chapter 5], Girard points out that the initial rule (recall Figure 4.2) implies that the left occurrence of B is stronger than the right occurrence of B, whereas the meaning of the cut rule is the opposite: a right occurrence of B is stronger than the left occurrence of B. This duality is also apparent in other presentations of these inference rules, such as in the Calculus of Structures [Guglielmi, 2007] and in uses of linear logic as a meta-logic for the sequent calculus (see Section 7.7 and Miller and Pimentel [2004, 2013]).

As was mentioned in Section 4.2, logic programs will be viewed in this book as theories that attribute meaning to programmer-supplied non-logical symbols. For example, suppose we wish to specify how to sort a list of numbers. In that case, we introduce a binary predicate, say, *sort*, to denote the relationship between lists of numbers and sorted lists of numbers. The logic program that describes how to compute this *sort* predicate is, in fact, a theory (collection of assumptions). (See Figure 5.6 for an explicit presentation of a logic program for specifying sorting.) Different proof-theoretic approaches to logic programming are available that do not use non-logical symbols in this way. For example, Hallnäs and Schroeder-Heister [1991] encode logic programs as *definitions* (which are given left and right introduction rules, as in Section 4.2). Horn clause logic programs also have rather direct and elegant encodings using fixed point expressions [McDowell and Miller, 2002; Tiu and Miller, 2005].

Chapter 5

Two abstract logic programming languages

We now apply the **C** and **I** proof systems to the description of logic programming languages in a high-level and implementation-independent fashion.

5.1 Goal-directed search

One approach to modeling logic programming is to view *logic programs* as assumptions, *goals* as queries to ask of a logic program, and *computation* as the process of attempting to prove a goal from a program. The state of an idealized interpreter can be represented as the two-sided sequent $\Sigma : \mathcal{P} \vdash G$, where Σ is the signature that declares a set of eigenvariables, $\mathcal{P}$ is a set of Σ-formulas denoting a program, and G is a Σ-formula denoting the goal we wish to prove from $\mathcal{P}$.

Central to viewing computation in logic programming seems to require the following restriction on the search for proofs. If G is not atomic, then its top-level logical connective should determine which inference rules should be used in an attempt to prove $\Sigma : \mathcal{P} \vdash G$: in particular, a right-introduction rule must be attempted. Thus, the *search semantics* for a logical connective at the head of a goal is fixed by the logic and is independent of the program. It is only when the goal is atomic, i.e., when its top-level symbol is *non-logical*, that the program $\mathcal{P}$ is consulted: the program is available to provide meaning for the non-logical, predicate constant at the head of atoms.

If we instantiate the above view of computation using the introduction rules given in Figure 4.1, we derive the following natural set of strategies.

- Reduce an attempt to prove $\Sigma : \mathcal{P} \vdash B_1 \wedge B_2$ to the attempts to prove the two sequents $\Sigma : \mathcal{P} \vdash B_1$ and $\Sigma : \mathcal{P} \vdash B_2$.

- Reduce an attempt to prove $\Sigma : \mathcal{P} \vdash B_1 \vee B_2$ to an attempt to prove either $\Sigma : \mathcal{P} \vdash B_1$ or $\Sigma : \mathcal{P} \vdash B_2$.
- Reduce an attempt to prove $\Sigma : \mathcal{P} \vdash \exists_\tau x.B$ to an attempt to prove $\Sigma : \mathcal{P} \vdash B[t/x]$, for some Σ-term t of type τ.
- Reduce an attempt to prove $\Sigma : \mathcal{P} \vdash B_1 \supset B_2$ to an attempt to prove $\Sigma : \mathcal{P}, B_1 \vdash B_2$.
- Reduce an attempt to prove $\Sigma : \mathcal{P} \vdash \forall_\tau x.B$ to an attempt to prove $\Sigma, c : \tau : \mathcal{P} \vdash B[c/x]$, where c is a token not in Σ.
- Attempting to prove $\Sigma : \mathcal{P} \vdash \boldsymbol{t}$ yields an immediate success.

These strategies suggest the following technical definition to formalize the notion of *goal-directed proof search*: a cut-free **I**-proof Ξ is a *uniform proof* if every occurrence of a sequent in Ξ that has a non-atomic right-hand side is the conclusion of a right-introduction rule. Searching for uniform proofs is now greatly restricted since building a uniform proof means applying right rules when the succedent has a logical connective. No left-introduction rules, no identity rules, and no structural rules can be considered when the right-hand side is a non-atomic formula. The definition of uniform proof provides no guidance for proof search when the right-hand side of a sequent is atomic. Such guidance will, however, soon appear.

Exercise 5.1. Show that uniform proofs are always atomically closed.

There are provable sequents for which no uniform proof exists. For example, let the non-logical constants be $\Sigma_0 = \{p : o, q : o, r : i \to o, a : i, b : i\}$ and let Σ be an signature. The sequents

$$\Sigma : (r\ a \wedge r\ b) \supset q \vdash \exists_i x(r\ x \supset q) \quad \text{and} \quad \Sigma : \cdot \vdash p \vee (p \supset q)$$

have **C**-proofs but no **I**-proofs (see Exercise 4.3), so clearly, they have no uniform proofs. The two sequents

$$\Sigma : p \vee q \vdash q \vee p \quad \text{and} \quad \Sigma : \exists_i x.\ r\ x \vdash \exists_i x.\ r\ x$$

have **I**-proofs but no uniform proofs.

One high-level way to define logic programming is to consider those collections of programs and goals for which uniform proofs are, in fact, complete. An *abstract logic programming language* is a triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ such that for all first-order signatures Σ_0, for all finite sets $\mathcal{P}$ of Σ_0-formulas from $\mathcal{D}$, and all Σ_0-formulas G of $\mathcal{G}$, we have $\Sigma_0 : \mathcal{P} \vdash G$ if and only if $\Sigma_0 : \mathcal{P} \vdash G$ has a uniform proof. Here, $\vdash$ is the provability relation associated to some particular logic, say, first-order classical or intuitionistic logic.

Both the definitions of uniform proof and abstract logic programming language are restricted to **I**-proofs. We shall refer to this as the *single-conclusion* version of these notions. After we introduce linear logic, we will present, in Section 6.7, a generalization of uniform proofs to multiple conclusion proof systems.

A theory Δ is said to satisfy the *disjunction property* if the provability of $\Sigma\colon\Delta \vdash B \vee C$ implies the provability of either $\Sigma\colon\Delta \vdash B$ or $\Sigma\colon\Delta \vdash C$. A theory Δ is said to satisfy the *existence property* if the provability of $\Sigma : \Delta \vdash \exists_\tau x.\ B$ implies the existence of a Σ-term t of type τ such that $\Sigma{:}\Delta \vdash B[t/x]$ is provable. Clearly, if uniform proofs are complete for a given theory and a notion of provability, that theory has both the disjunction and existence properties. In a sense, when uniform proofs are complete, these properties are satisfied at all points in building a cut-free proof.

5.2 Horn clauses

The first attempts to describe the provability of logic programs took place in the setting of performing *resolution refutations*: the choice of refuting over proving lead to a peculiar presentation of first-order Horn clauses. In that setting, Horn clauses were generally defined as the universal closure of disjunctions of *literals* (atomic formulas or their negation) that contain at most one positive literal (an atomic formula). That is, a clause is a closed formula for the form

$$\forall x_1 \ldots \forall x_n[\neg A_1 \vee \cdots \vee \neg A_m \vee B_1 \vee \cdots \vee B_p],$$

where $A_1, \ldots, A_m, B_1, \ldots, B_p$ are atomic formulas, $n, m, p \geq 0$, and $p \leq 1$. If $n = 0$ then the quantifier prefix is not written and if $m = p = 0$ then the body of the clause is considered to be $\boldsymbol{f}$. If the clause contained exactly one positive literal ($p = 1$), it is a *positive* Horn clause. If it contained no positive literal ($p = 0$), it is a *negative* Horn clause.

When we shift from the search for refutations to the search for sequent calculus proofs, it is natural to shift the presentation of Horn clauses to one of the following. Let τ be a syntactic variable that ranges over $S\backslash\{o\}$ (i.e., primitive types other than the type of formulas) and let A be a syntactic variable ranging over atomic formulas. Consider the following three, recursive definitions of the two syntactic categories of *program clauses* (*definite clause*), given by the syntactic variable D, and *goals*, given by the syntactic variable G.

$$\begin{aligned} G &::= \quad A \mid G \wedge G \\ D &::= \quad A \mid G \supset A \mid \forall_\tau x\ D. \end{aligned} \tag{5.1}$$

Program clauses using this presentation are of the form

$$\forall x_1 \ldots \forall x_n(A_1 \wedge \cdots \wedge A_m \supset A_0),$$

where we adopt the convention that if $m = 0$ then the implication is not written. A second, richer definition of these syntactic classes is the following.

$$\begin{array}{rc} G ::= & \boldsymbol{t} \mid A \mid G \wedge G \mid G \vee G \mid \exists_\tau x\, G \\ D ::= & \boldsymbol{t} \mid A \mid G \supset D \mid D \wedge D \mid \forall_\tau x\, D. \end{array} \tag{5.2}$$

Finally, a compact presentation of program clauses and goals is possible using only implication and universal quantification.

$$\begin{array}{rc} G ::= & A \\ D ::= & A \mid A \supset D \mid \forall_\tau x\, D. \end{array} \tag{5.3}$$

This last definition describes a program clause as a formula built from implications and universals such that there are no occurrences of logical connectives to the left of an implication. Program clauses using this presentation are of the form

$$\forall \bar{x}_1 (A_1 \supset \forall \bar{x}_2 (A_2 \supset \cdots \supset \forall \bar{x}_m (A_m \supset \forall \bar{x}_0 A_0) \ldots)),$$

where $\bar{x}_0, \ldots, \bar{x}_m$ are (possibly empty) lists of variables.

We use the symbol *fohc* to informally refer to the logic programming languages based on one of these three descriptions of *first-order Horn clauses*. Definition (5.1) above corresponds closely to the definition of Horn clauses given using disjunction of literals. In this case, positive clauses correspond to the D-formulas and the negation of G-formulas would all be negative clauses. Let $\mathcal{D}_1$ be the set of D-formulas and $\mathcal{G}_1$ be the set of G-formulas satisfying the recursion (5.2).

Exercise 5.2. For each of the three presentations of Horn clauses and goals above, show that the clausal order (see Section 2.4) of a formulas in $\mathcal{G}_1$ is 0 and of formulas in $\mathcal{D}_1$ is 0 or 1.

The following intuitionistic logic equivalences are sometimes called the *curry/uncurry equivalences*.

1. $\boldsymbol{t} \supset E \equiv E$
2. $(B \wedge C) \supset E \equiv (B \supset C \supset E)$
3. $(B \vee C) \supset E \equiv (B \supset E) \wedge (C \supset E)$
4. $(\exists x.B) \supset E \equiv \forall x.(B \supset E)$

They can be used (in part) to prove the following exercise.

Exercise 5.3. Let D be a Horn clause using (5.2). Show that there is a set Δ of Horn clauses using description (5.1) or (5.3) (your pick) such that D

is equivalent to the conjunction of formulas in Δ. Show that this rewriting might make the resulting conjunction exponentially larger than the original clause. (Take as the measure of a formula the number of occurrences of logical connectives it contains.)

Exercise 5.4. Let Σ be a signature, let $\mathcal{P}$ be a set of Σ-formulas in $\mathcal{D}_1$, and let G be a Σ-formula in $\mathcal{G}_1$. Let Ξ be a cut-free **C**-proof of $\Sigma : \mathcal{P} \vdash G$. Show that every sequent in Ξ is of the form $\Sigma : \mathcal{P}' \vdash \Delta$ such that $\mathcal{P}'$ is a subset of $\mathcal{D}_1$ and Δ is a subset of $\mathcal{G}_1$. Show also that the only introduction rules that can appear in Ξ are $\forall$L, $\wedge$L, $\supset$L, $\wedge$R, $\vee$R, $\exists$R, and $\boldsymbol{t}$R.

Exercise 5.5. Prove that Horn clause programs are always consistent by proving that for any signature Σ and any finite set of Horn clauses $\mathcal{P}$, the sequent $\Sigma : \mathcal{P} \vdash \boldsymbol{f}$ is not provable. Show that an **I**-proof of $\Sigma : \mathcal{P} \vdash G$ for a Horn goal G is also an **M**-proof.

We first show that in the Horn clause setting, classical provability is conservative over intuitionistic logic.

Proposition 5.6. *Let Σ be a signature, let $\mathcal{P}$ be a set of Σ-formulas in $\mathcal{D}_1$, and let G be a Σ-formula in $\mathcal{G}_1$. If $\Sigma : \mathcal{P} \vdash G$ has a **C**-proof then it has an **I**-proof.*

Proof. We show the following stronger result: if Δ is a multiset of G-formulas and $\Sigma : \mathcal{P} \vdash \Delta$ has a cut-free **C**-proof then there is a $G \in \Delta$ such that $\Sigma : \mathcal{P} \vdash G$ has an **I**-proof. We prove this by induction on the structure of a cut-free **C**-proof Ξ for $\Sigma : \mathcal{P} \vdash \Delta$.

There are three base cases for Ξ: $\boldsymbol{f}$L is not possible since $\boldsymbol{f}$ is not a member of $\mathcal{P}$ and the two other cases of $\boldsymbol{t}$R and *init* are immediate.

If the last inference rule in Ξ is a structural rule, the proof is straightforward again. For example, suppose the last inference in Ξ is a *cR*. In that case, this proof is of the form

$$\frac{\Sigma : \mathcal{P} \vdash G, G, \Delta}{\Sigma : \mathcal{P} \vdash G, \Delta}\ cR\ .$$

By the inductive hypothesis, there is an H in the multiset G, G, Δ such that $\Sigma : \mathcal{P} \vdash H$ has an **I**-proof: clearly, H is also a member of the multiset G, Δ.

Now consider all possible introduction rules that might be the last inference rule of Ξ (see Exercise 5.4). If that last rule is $\supset$L, then the proof has the form

$$\frac{\Sigma : \mathcal{P}_1 \vdash \Delta_1, G \qquad \Sigma : D, \mathcal{P}_2 \vdash \Delta_2}{\Sigma : G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash \Delta_1, \Delta_2}\ \supset\text{L}\ .$$

By the induction assumption, there is a formula $H_1 \in \Delta_1 \cup \{G\}$ for which $\Sigma : \mathcal{P}_1 \vdash H_1$ has an **I**-proof and a formula $H_2 \in \Delta_2$ for which $\Sigma : D, \mathcal{P}_2 \vdash H_2$ has an **I**-proof. In the case that $H_1 \in \Delta_1$, the **I**-proof of the sequent $\Sigma{:}\mathcal{P}_1 \vdash H_1$ can be extended with a series of wL rules to yield a proof of $\Sigma{:}G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash H_1$. On the other hand, if $H_1 = G$, then we build an **I**-proof using the following instance of an inference rule

$$\frac{\Sigma : \mathcal{P}_1 \vdash G \qquad \Sigma : D, \mathcal{P}_2 \vdash H_2}{\Sigma : G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash H_2} \supset\mathrm{L}\ ,$$

and the two promised **I**-proofs of the premises.

All the remaining cases of introduction rules can be treated similarly. □

Exercise 5.7. (‡) Assume that the Σ-formulas $D_0, \ldots, D_n$ $(n \geq 0)$ are Horn clauses using description (5.3). Prove that if the sequent $\Sigma : D_1, \ldots, D_n \vdash D_0$ has a **C**-proof then it has an **I**-proof.

It is the case that $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash \rangle$ is an abstract logic programming language if $\vdash$ is taken to be $\vdash_C$, $\vdash_I$, or $\vdash_M$.

Note that uniform proofs in *fohc* are very constrained. In particular, if we use the (5.2) presentation of Horn clauses, then it is only atoms or conjunctions of atoms that are both goals and program clauses. All the other connectives are either dismissed (such as $\boldsymbol{f}$) or are restricted to just half their "meaning:" when a disjunction and existential quantifier is encountered in proof search, only its right introduction rule is needed, and when an implication and a universal quantification is encountered, only its left-introduction rule is needed.

Exercise 5.8. (‡) Let $\mathcal{I}$ be the set of formulas using only implications and atomic formulas that are classical theorems but do not have uniform proofs. For example, Peirce's formula $((p \supset q) \supset p) \supset p$ is a member of $\mathcal{I}$. Prove that the smallest formula in $\mathcal{I}$ has three occurrences of implications.

Readers unfamiliar with specifying computations using Horn clauses might want to read Section 5.10 now to see examples of such specifications.

5.3 Hereditary Harrop formulas

A natural extension to Horn clauses, called the *first-order hereditary Harrop formulas*, allows implications and universal quantifiers in goals (and, thus, in the body of program clauses). Whereas cut-free proofs involving Horn clauses contain left-introduction rules for implications and universal quantifiers, proofs involving this extended set of formulas can contain also right-introduction rules for implications and universal quantifiers. Parallel to the three presentations of *fohc* in Section 5.2, the following three presentations of goals and program clauses describe first-order hereditary Harrop formulas.

$$\begin{aligned} G &::= \quad A \mid G \wedge G \mid D \supset G \mid \forall x.G \\ D &::= \qquad A \mid G \supset A \mid \forall x.D \end{aligned} \tag{5.4}$$

The definitions of G- and D-formulas are mutually recursive. Note that a negative (resp, positive) subformula of a G-formula is a D-formula (G-formula), and that a negative (positive) subformula of a D-formula is a G-formula (D-formula). A richer formulation is given by

$$\begin{array}{rl} G ::= & \boldsymbol{t} \mid A \mid G \wedge G \mid G \vee G \mid \exists x.G \mid D \supset G \mid \forall x.G \\ D ::= & A \mid G \supset D \mid D \wedge D \mid \forall x.D \end{array} \tag{5.5}$$

When referring to first-order hereditary Harrop formulas and goals, we shall assume this definition of formulas. We use $\mathcal{D}_2$ to denote the set of all such D-formulas and $\mathcal{G}_2$ for the set of all G-formulas.

A completely symmetric presentation can be given as

$$\begin{array}{rl} G ::= & \boldsymbol{t} \mid A \mid D \supset G \mid G \wedge G \mid \forall x.G \\ D ::= & \boldsymbol{t} \mid A \mid G \supset D \mid D \wedge D \mid \forall x.D \end{array} \tag{5.6}$$

In this presentation, D and G formulas are the same set of formulas, and there is no need for a definition that allows for mutual recursion. In Section 5.5, these formulas—which are generated from the set of connectives $\{\boldsymbol{t}, \wedge, \supset, \forall\}$—will be called $\mathcal{L}_0$-formulas.

We use the name *fohh* to denote *first-order hereditary Harrop formulas*: this name will refer to one of the presentations above. If the text is not explicit about which presentation is implied, we will assume the second presentation. He shall also use *fohh* to denote, in particular, the corresponding D-formulas: this is justified by the fact that the associated G-formulas are uniquely determined by the negative subformulas of D-formulas. The same comment also applies to our use of the term *fohc*.

Exercise 5.9. Let $D \in \mathcal{D}_2$. Then D is a Horn clause (using definition (5.2)) if and only if $\text{order}(D) < 2$.

We shall use the term *clause* not just for Horn clauses but for any formula, especially any formula that can be used as part of a logic program. Thus, for example, we often refer to hereditary Harrop formulas also by this term.

The following proposition shows that identifying the right-hand side with goals and the left-hand side with programs is maintained within cut-free **I**-proofs.

Proposition 5.10. *Let $\mathcal{P}$ be an fohh logic program and G an fohh goal and let Ξ be a cut-free **I**-proof of $\Sigma : \mathcal{P} \vdash G$. If $\Sigma' : \Gamma' \vdash B$ is a sequent in Ξ then Γ' is a fohh logic program and B is an fohh goal formula.*

This proposition is proved by a simple induction of the structure of cut-free **I**-proofs.

The triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_C \rangle$ is not an abstract logic programming language. For example, the formulas numbered 4, 5, 6, and 7 in Exercise 4.3 are hereditary Harrop goals that have classical proofs but no uniform proof.

We shall informally refer to the logic programming languages based on intuitionistic logic and one of these three descriptions of *first-order hereditary Harrop formulas* by simply *fohh* or as $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$.

Lemma 5.11. *Let $G \in \mathcal{G}_2$ be a non-atomic Σ-formula and let $\mathcal{P}$ be a finite multiset, all of whose members are Σ-formulas in $\mathcal{D}_2$. Assume that $\Sigma : \mathcal{P} \vdash G$ has an **I**-proof in which the last inference rule is not a right-introduction rule, and all premise sequents are proved by a uniform proof. There is a uniform proof of $\Sigma : \mathcal{P} \vdash G$.*

Proof. Let Ξ be a proof of $\mathcal{P} \vdash G$ satisfying the assumptions of this lemma. (For readability, we suppress explicitly writing the signature of a sequent.) The last inference rule of this proof is either one of two structural rules (cL or wL) or one of three left-introduction rules ($\wedge$L, $\forall$L, $\supset$L). In every case, the proof of the premises must be uniform proofs and, as a result, at least one premise must be proved by one of five right-introduction rules ($\wedge$R, $\vee$R, $\forall$R, $\exists$R, $\multimap$R). We proceed by induction on the height of the uniform proof of the right-most premise of this inference rule. All possible cases of left-rules occurring below a right-introduction rule must be considered.

Consider the case when an implication-left rule is applied when the right-hand side is a conjunction.

$$\dfrac{\begin{matrix}\Xi_0\\ \mathcal{P}_1 \vdash G\end{matrix} \qquad \dfrac{\begin{matrix}\Xi_1\\ D, \mathcal{P}_2 \vdash G_1\end{matrix} \qquad \begin{matrix}\Xi_2\\ D, \mathcal{P}_2 \vdash G_2\end{matrix}}{D, \mathcal{P}_2 \vdash G_1 \wedge G_2}\,\wedge\text{R}}{G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash G_1 \wedge G_2}\,\supset\text{L}$$

These rules can be permuted to form the following proof.

$$\dfrac{\dfrac{\begin{matrix}\Xi_0\\ \mathcal{P}_1 \vdash G\end{matrix} \qquad \begin{matrix}\Xi_1\\ \mathcal{P}_2, D \vdash G_1\end{matrix}}{G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash G_1}\,\supset\text{L} \qquad \dfrac{\begin{matrix}\Xi_0\\ \mathcal{P}_1 \vdash G\end{matrix} \qquad \begin{matrix}\Xi_2\\ \mathcal{P}_2, D \vdash G_2\end{matrix}}{G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash G_2}\,\supset\text{L}}{G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash G_1 \wedge G_2}\,\wedge\text{R}$$

If this proof is not uniform, apply the inductive assumption to the two subproofs with $\supset$L as their last rule. That induction returns a uniform proof for both $G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash G_1$ and $G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash G_2$ and a uniform proof for the end-sequent comes from applying $\wedge$R to those uniform proofs.

For another case, assume that $\supset$L is applied to a sequent with an implication on the right-hand side.

$$\dfrac{\begin{matrix}\Xi_1\\ \mathcal{P}_1 \vdash G\end{matrix} \qquad \dfrac{\begin{matrix}\Xi_2\\ D', D, \mathcal{P}_2 \vdash G'\end{matrix}}{D, \mathcal{P}_2 \vdash D' \supset G'}\,{\multimap}\mathrm{R}}{G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash D' \supset G'}\,{\supset}\mathrm{L}$$

These rules can be permuted to form the following proof.

$$\dfrac{\dfrac{\begin{matrix}\Xi_1\\ \mathcal{P}_1 \vdash G\end{matrix} \qquad \begin{matrix}\Xi_2\\ D, D', \mathcal{P}_2 \vdash G'\end{matrix}}{G \supset D, D', \mathcal{P}_1, \mathcal{P}_2 \vdash G'}\,{\supset}\mathrm{L}}{G \supset D, \mathcal{P}_1, \mathcal{P}_2 \vdash D' \supset G'}\,{\multimap}\mathrm{R}$$

If this proof is not uniform, then apply the inductive hypothesis to the right premise of the $\multimap$R rule.

All other cases can be proved similarly: permute a left-rule up over a right-introduction rule and invoke the inductive hypothesis. □

Proposition 5.12. *Let Σ be a signature, let $\mathcal{P}$ be a finite multiset of Σ-formulas in $\mathcal{D}_2$, and let G be a Σ-formula in $\mathcal{G}_2$. If $\Sigma : \mathcal{P} \vdash G$ has a cut-free **I**-proof then $\Sigma : \mathcal{P} \vdash G$ has a uniform proof.*

Proof. Assume that $\Sigma : \mathcal{P} \vdash G$ has a cut-free **I**-proof Ξ. By Proposition 4.10, we can also assume that Ξ is an atomically closed **I**-proof. If Ξ is not uniform, then there must be occurrences of left-rules (either left-introduction rules or left-structural rule) in Ξ whose conclusion is a sequent with a non-atomic right-hand side. Pick one of these occurrences so that the subproofs of its premises do not have other such occurrences. Thus, the premises of this inference rule occurrence are uniform. By Lemma 5.11, we can replace the subproof determined by this left rule with a uniform proof. In this way, we can continue to replace non-uniform subproofs with uniform proofs until such rewriting yields a uniform proof. □

This proposition formally asserts that the intuitionistic version of *fohh* is an abstract logic programming language.

Consider the following class of first-order formulas given by

$$H := A \mid B \supset H \mid \forall x\, H \mid H_1 \wedge H_2.$$

Here A ranges over atomic formulas and B over arbitrary first-order formulas. These H-formulas are known as *Harrop formulas*. Clearly, hereditary Harrop formulas are Harrop formulas.

Exercise 5.13. Consider the sequent $\Sigma : \mathcal{P} \vdash B$ where $\mathcal{P}$ is a set of Harrop formulas and B is an arbitrary formula. Show that Harrop formulas are "uniform at the root;" that is, if B is non-atomic, then this sequent is intuitionistically provable if and only if it has a **I**-proof that ends in a right-introduction rule. Are uniform proofs complete for such sequents?

Finally, note that since hereditary Harrop formulas do not have occurrences of $\boldsymbol{f}$ in them, the triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_M \rangle$ describes essentially the same abstract logic programming language as *fohh*.

The reader, who wishes to see examples of logic programs in *fohh* before considering more about their proof theory, can find some in Section 5.12.

5.4 Backchaining as focused rule application

The restriction to uniform proofs provides some information on how to structure proofs: in the bottom-up search for proofs, right-introduction rules are attempted whenever the antecedent is non-atomic, and left-rules are attempted only when the succedent is atomic. We now present a restriction on the application of left side rules, and we will eventually show that that restriction on proofs does not result in the loss of completeness.

To better structure the rules on the left, we first make two simple changes to the proof system for **I**-proofs. While *wL* can be applied at any point in the search for a uniform proof, it is also possible to delay applications of that rule until just before applying the *init* rule. This delay suggests that we can fold weakening into the *init* rule, yielding the derived inference rule

$$\frac{}{\Sigma : \Gamma, B \vdash B} \ .$$

Another use of a structural rule on the left can improve the complexity of the $\supset$L rule when searching for a proof. As we mentioned in Section 3.3, performing proof search with a multiplicative inference rule can be expensive since there can be an exponential number of ways to split the side contexts of the conclusion for use among the premises. The only multiplicative left-introduction rule is $\supset$L. Since contraction and weakening are available on the left (but not the right), the following variant of that inference rule is easily proved to be admissible (see Section 3.3).

$$\frac{\Sigma : \Gamma \vdash \Delta_1, B \qquad \Sigma : C, \Gamma \vdash \Delta_2}{\Sigma : B \supset C, \Gamma \vdash \Delta_1, \Delta_2}$$

Here, the *cL* rule is used to double the Γ context before splitting the left context. In this rule, the left context is treated additively, and the right context is treated multiplicatively. Given that we are speaking of **I**-proofs here, this rule can be simplified even further since the single formula on the right of the concluding sequent must move to the right of the right premise. Thus, we can rewrite this rule as

$$\frac{\Sigma : \Gamma \vdash B \qquad \Sigma : C, \Gamma \vdash E}{\Sigma : B \supset C, \Gamma \vdash E}$$

Now consider refining this last version of the left introduction of implication in the setting of uniform proofs. That is, consider the derivation

$$\dfrac{\dfrac{\Sigma : \mathcal{P} \vdash G \qquad \Sigma : D, \mathcal{P} \vdash A}{\Sigma : G \supset D, \mathcal{P} \vdash A} \supset\mathrm{L}}{\Sigma : \mathcal{P} \vdash A}\ cL$$

where A is atomic and where $G \supset D$ is a member of the multiset $\mathcal{P}$. Thus, to employ $G \supset D$ in backchaining, we first use *cL* to make a copy of it and then apply $\supset$L. Thus, we have reduced an attempt to prove the atomic formula A from program $\mathcal{P}$ to attempting to prove two things, one of which is still an attempt to prove A but this time from the larger multiset $\mathcal{P} \cup \{D\}$. It would seem natural to expect these inference rules are used only because this new instance of D is directly helpful in proving A. For example, D could itself be A, or some sequence of additional left-rules applied to D might reduce it to an occurrence of A.

We can formalize a proof system where left-introduction rules are used in such a direct or *focused* fashion by introducing a new style of sequent, namely, $\Sigma : \mathcal{P} \Downarrow D \vdash A$. While provability of this sequent will imply provable of the sequent $\Sigma : \mathcal{P}, D \vdash A$, the formula between the $\Downarrow$ and the $\vdash$, called the *focus* of this sequent, is the only formula on which left-introduction rules can be applied. The sequents $\Sigma : \mathcal{P} \vdash G$ and $\Sigma : \mathcal{P} \Downarrow D \vdash A$ have $\Downarrow$*fohh*-proofs if they have proofs using the $\Downarrow$*fohh*-proof system in Figure 5.1. This new proof system is an example of a *focused* proof system: we shall see two more such focused proof systems when we introduce linear logic in Chapter 6.

All $\Downarrow$*fohh*-proofs are composed of two phases. A *right-introduction phase* is a derivation composed of only right-introduction rules and where all open premises are sequents with atomic formulas on their right-hand sides. Such phases can be identified with the goal-reduction phase of proof search. A right-introduction phase for $\Sigma : \mathcal{P} \vdash G$ is empty (i.e., contain no inference rules) if and only if G is an atomic formula. A *left-introduction phase* is a derivation composed of left-introduction rules as well as the *init* and *decide* rules (see Figure 5.1) and where all open premises are sequents without the $\Downarrow$. A left-introduction phase for $\Sigma : \Gamma \Downarrow B \vdash A$ can never be empty: that is, such a phase must contain an inference rule (in particular, the *decide* rule). This phase can be identified with the backchaining phase of proof search that we have described earlier.

$$\frac{}{\Sigma : \mathcal{P} \vdash \boldsymbol{t}}\ \boldsymbol{t}\mathrm{R} \qquad \frac{\Sigma : \mathcal{P} \vdash G_1 \qquad \Sigma : \mathcal{P} \vdash G_2}{\Sigma : \mathcal{P} \vdash G_1 \wedge G_2}\ \wedge\mathrm{R}$$

$$\frac{y : \tau, \Sigma : \mathcal{P} \vdash G[y/x]}{\Sigma : \mathcal{P} \vdash \forall_\tau x\, G}\ \forall\mathrm{R} \qquad \frac{\Sigma : D, \mathcal{P} \vdash G}{\Sigma : \mathcal{P} \vdash D \supset G}\ \multimap\mathrm{R}$$

$$\frac{\Sigma : \mathcal{P} \vdash G_1}{\Sigma : \mathcal{P} \vdash G_1 \vee G_2}\ \vee\mathrm{R} \qquad \frac{\Sigma : \mathcal{P} \vdash G_2}{\Sigma : \mathcal{P} \vdash G_1 \vee G_2}\ \vee\mathrm{R}$$

$$\frac{\Sigma \Vdash t : \tau \qquad \Sigma : \mathcal{P} \vdash G[t/x]}{\Sigma : \mathcal{P} \vdash \exists_\tau x\, G}\ \exists\mathrm{R}$$

$$\frac{\Sigma : \mathcal{P} \Downarrow D \vdash A}{\Sigma : \mathcal{P} \vdash A}\ \mathit{decide} \qquad \frac{}{\Sigma : \mathcal{P} \Downarrow A \vdash A}\ \mathit{init}$$

$$\frac{\Sigma : \mathcal{P} \Downarrow D_1 \vdash A}{\Sigma : \mathcal{P} \Downarrow D_1 \wedge D_2 \vdash A}\ \wedge\mathrm{L} \qquad \frac{\Sigma : \mathcal{P} \Downarrow D_2 \vdash A}{\Sigma : \mathcal{P} \Downarrow D_1 \wedge D_2 \vdash A}\ \wedge\mathrm{L}$$

$$\frac{\Sigma : \mathcal{P} \vdash G \qquad \Sigma : \mathcal{P} \Downarrow D \vdash A}{\Sigma : \mathcal{P} \Downarrow G \supset D \vdash A}\ \supset\mathrm{L} \qquad \frac{\Sigma \Vdash t : \tau \qquad \Sigma : \mathcal{P} \Downarrow D[t/x] \vdash A}{\Sigma : \mathcal{P} \Downarrow \forall_\tau x. D \vdash A}\ \forall\mathrm{L}$$

Figure 5.1: The $\Downarrow$*fohh* proof system. In the *decide* rule, D is a member of $\mathcal{P}$. In all these rules, A is atomic.

It is important to note the following relationship between determinism and right-introduction phases and between nondeterminism and left-introduction phases. Let Σ be a signature and let $\mathcal{P}$ and G be a logic program and a goal formula, respectively, in *fohh* (all Σ-formulas). There always exists a right-introduction phase that ends in $\Sigma : \mathcal{P} \vdash G$, and that phase is unique up to the change of names of the eigenvariables. Thus, a right-introduction phase can be seen as a *function* that takes the endsequent $\Sigma : \mathcal{P} \vdash G$ as input and returns the unique multiset of sequents of the form $\Sigma' : \mathcal{P}' \vdash A$ (where A is an atomic formulas) that are the premises of that right-introduction phase. On the other hand, the left-introduction phase determines a nondeterministic *relation* between its endsequent, say, $\Sigma : \mathcal{P} \Downarrow D \vdash A$, and the multiset of sequents of the form $\Sigma : \mathcal{P} \vdash G$ that are the premises of a left-introduction phase.

Exercise 5.14. Given a sequence $a_0, a_1, \ldots, a_n$ of atomic (propositional) formulas ($n \geq 0$), define the sequence of propositional Horn clauses

$$D_n = a_0 \supset \cdots \supset a_{n-1} \supset a_n \quad (n \geq 0).$$

For example, D_0 is a_0, D_1 is $a_0 \supset a_1$, and D_2 is $a_0 \supset a_1 \supset a_2$. For a given $n \geq 0$, there are a great many uniform proofs of the sequent $D_0, \ldots, D_n \vdash$

a_n. Among these, consider those in which the left premise of the $\supset$L rule is trivial (proved by the initial rule). Those proofs use the formulas D_i in *forwardchaining* manner. How do such proofs differ in size to proofs based only on backchaining, i.e., $\Downarrow$*fohh*-proofs?

5.5 Formal properties of focused proofs

The proof system in Figure 5.1 is different from the original proof systems of Gentzen in that there is a lot of control over the application of introduction rules. In particular, the only way to prove a sequent that does not contain $\Downarrow$ is to perform a right-introduction rule or the *decide* rule. If a sequent contains the $\Downarrow$ then that sequent must be the conclusion of a left-introduction rule or the *init* rule. Furthermore, contraction and weakening are not separate rules but are built into other rules.

The preceding sections in this chapter present various theorems about the unfocused proof systems **I** and **C** and their relationship with Horn clauses and hereditary Harrop formulas. In general, the focused proof system is much more useful than those unfocused proof systems for our purposes here. Once we have proved the main theorems about the focused proof system $\Downarrow$ *fohh*, most of the results in the previous sections can be reproved immediately using those theorems.

The following proposition states that whatever is provable using $\Downarrow$*fohh*-proofs is also provable in intuitionistic proofs.

Proposition 5.15 (Soundness of $\Downarrow$*fohh*-proofs)**.** *Let Σ be a signature and let Γ be a multiset of definite Σ-formulas and let G be a goal Σ-formula. If the sequent $\Sigma : \Gamma \vdash G$ has a $\Downarrow$fohh-proof then it has an **I**-proof.*

Proof. This is proved by a simple induction of the structure of $\Downarrow$*fohh*-proofs. In that induction, sequents of the form $\Gamma \Downarrow D \vdash A$ are mapped to standard sequents of the form $\Gamma, D \vdash A$. $\square$

We will eventually prove that, for hereditary Harrop formulas, $\Downarrow$ *fohh*-proofs are complete for intuitionistic logic (Proposition 5.37). Before proving that theorem, we first develop some results about the inference rules in Figure 5.1. In particular, we note that the $\Downarrow$ *fohh*-proof system does not have a *cut* rule, and its *init* rule is restricted to atomic formulas. It is natural to ask if the *cut* rule and the general form of the *init* rule are admissible for $\Downarrow$*fohh*-proofs. However, just to ask that question requires us to restrict our attention to those formulas that are both goal formulas and definite clauses. Within *fohh*, these are the only formulas that can appear on the left and the right of the sequent arrow. Let $\mathcal{L}_0$ be the set of connectives $\{\boldsymbol{t}, \wedge, \supset, \forall\}$ and let an $\mathcal{L}_0$-*formula* be any first-order formula all of whose logical connectives come from $\mathcal{L}_0$. In particular, such formulas do not contain occurrences of disjunctions and existential quantifiers. Until we return to the issue of dealing with disjunctions and existential quantifiers in Section 5.9, we restrict our attention to $\mathcal{L}_0$ formulas, which are also the same as *fohh* using definition (5.6).

Since $\mathcal{L}_0$ formulas have no occurrences of $\boldsymbol{f}$, provability in intuitionistic and minimal logics coincide (see Section 4.5). Thus, for most of the rest of this chapter, we could replace references to intuitionistic logic with minimal logic when discussing the properties of $\Downarrow$ *fohh*-proofs. In addition, we emphasize the role of $\mathcal{L}_0$ formulas in this section by using the name $\Downarrow\mathcal{L}_0$*-proof system* for the proof system that results from removing the right-introduction rules for $\exists$ and $\vee$ from the $\Downarrow$*fohh*-proof system.

Let B be an $\mathcal{L}_0$ formula. The *paths in B* are those formulas P for which the following two-place relation $B \uparrow P$ is provable (here, A denotes an atomic formula).

$$\frac{}{A \uparrow A} \qquad \frac{B \uparrow P}{B \wedge C \uparrow P} \qquad \frac{C \uparrow P}{B \wedge C \uparrow P} \qquad \frac{C \uparrow P}{B \supset C \uparrow B \supset P} \qquad \frac{B \uparrow P}{\forall_\tau x.B \uparrow \forall_\tau x.P}$$

A formula which is a path has the form

$$\forall \bar{x}_1.(G_1 \supset \forall \bar{x}_2.(G_2 \supset \ldots \supset \forall \bar{x}_n.(G_n \supset A)\ldots)),$$

where $n \geq 0$, A is an atomic formula, $G_1, \ldots, G_n$ is a list of $\mathcal{L}_0$ of formulas, and where for each i such that $0 < i \leq n$, $\bar{x}_i$ is a list of variables. The formula A is the *target* of this path, the formulas $G_1, \ldots, G_n$ are the *arguments* of this path, and the list that results from concatenating the lists of variables $\bar{x}_1, \ldots, \bar{x}_n$ is the list of *bound variables* of this path. (We assume that all these bound variables are distinct.) We shall also present such a path using an *associated sequent*, namely, $\bar{x}_1, \ldots, \bar{x}_n : G_1, \ldots, G_n \vdash A$.

For example, the paths in $(p \wedge q) \supset (r \wedge s)$ are $(p \wedge q) \supset r$ and $(p \wedge q) \supset s$. Similarly, the formula

$$\forall x.p(x) \supset ((\forall y.q(x,y) \supset (r(x,y) \wedge r(y,x))) \wedge p(x))$$

(where p, q, and r are predicates) has three paths, namely,

$$\forall x.p(x) \supset \forall y.q(x,y) \supset r(x,y) \qquad x,y : p(x), q(x,y) \vdash r(x,y)$$

$$\forall x.p(x) \supset \forall y.q(x,y) \supset r(y,x) \qquad x,y : p(x), q(x,y) \vdash r(y,x)$$

$$\forall x.p(x) \supset p(x) \qquad x : p(x) \vdash p(x).$$

Here, we also display the associated sequent representation of the path. Note that the formula $\boldsymbol{t}$ has no paths, and if the formula B contains no occurrences of $\boldsymbol{t}$ and $\wedge$ then the only path in B is B itself.

Exercise 5.16. Let D be a hereditary Harrop formula defined using (5.4). Prove that D has exactly one path and that path is D.

Given the intuitionistically valid equivalences

$$B_1 \supset (B_2 \wedge B_3) \equiv (B_1 \supset B_2) \wedge (B_1 \supset B_3)$$
$$\forall x.\ (B_1 \wedge B_2) \equiv (\forall x.\ B_1) \wedge (\forall x.\ B_2),$$

it is easy to show the intuitionistic equivalence

$$B \equiv \bigwedge_{B \uparrow P} P.$$

We can even state the following two much stronger relationships between B and the conjunction of all paths in B.

1. The right-introduction phase that has endsequent $\Sigma{:}\Gamma \vdash B$ and the right-introduction phase that has endsequent $\Sigma : \Gamma \vdash \bigwedge_{B\uparrow P} P$ have exactly the same premises (modulo the order in which the premises are listed and modulo alphabetic changes in the names of eigenvariables).

2. The set of left-introduction phases with endsequent $\Sigma : \Gamma \Downarrow B \vdash A$ can be put in one-to-one correspondence with left-introduction phases with endsequent $\Sigma : \Gamma \Downarrow \bigwedge_{B\uparrow P} P \vdash A$ in such a way that corresponding premises are equal (modulo the order in which the premises are listed and modulo alphabetic changes in the names of eigenvariables).

These observations are stated more formally in the next two propositions.

Proposition 5.17. *Let B be an $\mathcal{L}_0$ formula and let the sequent $\Sigma : \Gamma \vdash B$ be the endsequent of a right-introduction phase. The premises of that phase are in one-to-one correspondence with paths in B such that the path P corresponds to the premise $\Sigma, \mathcal{X} : \Gamma, \mathcal{B} \vdash A$, where the sequent associated to P is $\mathcal{X} : \mathcal{B} \vdash A$. (The variables in $\mathcal{X}$ are chosen to be disjoint from Σ.)*

Proof. We prove this proposition by induction on the structure of the $\mathcal{L}_0$ formula B. In the case that B is $\boldsymbol{t}$, the set of paths in B is empty, and the set of premises of the right-introduction phase is also empty. If B is atomic, the end-sequent of the right-introduction phase is the same as its unique premise, which corresponds to adding no bound variables and no argument formulas (this phase is empty). If B is $B_1 \wedge B_2$ then the right-introduction phase ends with

$$\frac{\Sigma : \Gamma \vdash B_1 \quad \Sigma : \Gamma \vdash B_2}{\Sigma : \Gamma \vdash B_1 \wedge B_2} \; .$$

The premises of this phase are divided into those which are premises of the right-introduction phase with endsequent $\Sigma : \Gamma \vdash B_1$ and the premises of the right-introduction phase with endsequent $\Sigma : \Gamma \vdash B_2$. Since the paths in P are either paths in B_1 or in B_2, the inductive hypothesis immediately yields the required correspondence. If B is $B_1 \supset B_2$ then the right-introduction phase ends with

$$\frac{\Sigma : \Gamma, B_1 \vdash B_2}{\Sigma : \Gamma \vdash B_1 \supset B_2} \; .$$

The premises of this phase are also premises of the right-introduction phase with endsequent $\Sigma : \Gamma, B_1 \vdash B_2$. By the inductive hypothesis, a path P' in B_2 correspond to the premise $\Sigma, \mathcal{X} : \Gamma, B_1, \mathcal{B} \vdash A$, where $\mathcal{X} : \mathcal{B} \vdash A$ is the sequent associated to P'. By the definition of paths, the only difference between the path P and P' is that the former has B_1 as an additional argument. Thus, the correspondence is satisfied. The case where B is $\forall x.B'$ is similar to the previous case. □

The proposition above states that an attempt to prove $\Sigma : \Gamma \vdash B$ leads to an attempt to prove a series of sequents, one for each path in B. The structure of the left-introduction phases is described in the following proposition.

Proposition 5.18. *Let B be an $\mathcal{L}_0$ formula and A an atomic formula. The sequent $\Sigma : \Gamma \Downarrow B \vdash A$ is the endsequent of a left-introduction phase with premises*

$$\Sigma : \Gamma \vdash G_1 \; , \ldots , \; \Sigma : \Gamma \vdash G_n \quad (n \geq 0)$$

if and only if there is a path P in B with target A', arguments $B_1, \ldots, B_n$, and bound variables $\mathcal{X}$, and a substitution θ that maps the variables in $\mathcal{X}$ to Σ-terms such that $A'\theta = A$ and such that $G_1 = B_1\theta, \ldots, G_n = B_n\theta$.

Proof. We prove this proposition by induction on the structure of the $\mathcal{L}_0$ formula B. The case that B is $\boldsymbol{t}$ is impossible since there is no left-introduction rule for $\boldsymbol{t}$. If B is atomic, then B and A are equal since we assume that $\Sigma : \Gamma \Downarrow B \vdash A$ is the endsequent of a left-introduction phase (and the set of arguments of B is the empty set).

If B is $B_1 \wedge B_2$, we first assume that there is a left-introduction phase ending in $\Sigma : \Gamma \Downarrow B_1 \wedge B_2 \vdash A$. Thus, there is a left-introduction phase ending in $\Sigma : \Gamma \Downarrow B_i \vdash A$, where $i = 1$ or $i = 2$. By the inductive assumption, there is a path in B_i with target A', arguments $\mathcal{B}$, and bound variables $\mathcal{X}$, and a substitution θ that maps the variables in $\mathcal{X}$ to Σ-terms such that $A'\theta$ is equal to A and such that every premise of that left-introduction phase can be written as $\Sigma : \Gamma \vdash G\theta$ for each $G \in \mathcal{B}$. That same path is also a path in B, which completes this case. The converse is proved similarly.

If B is $B_1 \supset B_2$, we first assume that there is a left-introduction phase that ends with $\Sigma : \Gamma \Downarrow B_1 \supset B_2 \vdash A$ and the inference rule

$$\frac{\Sigma : \Gamma \vdash B_1 \quad \Sigma : \Gamma \Downarrow B_2 \vdash A}{\Sigma : \Gamma \Downarrow B_1 \supset B_2 \vdash A} .$$

By the inductive hypothesis, there is a path in B_2 with target A', arguments $\mathcal{B}$, bound variables $\mathcal{X}$, and a substitution θ that maps the variables in $\mathcal{X}$ to Σ-terms such that $A'\theta$ is equal to A and such that every premise of that left-introduction phase can be written as $\Sigma : \Gamma \vdash G\theta$ for each $G \in \mathcal{B}$. If we add to that path the argument B_1 then that path satisfies the required condition for a path in B. The converse is proved similarly.

Finally, assume that B is $\forall_\tau x.B'$. First assume that there is a left-introduction phase ending in $\forall_\tau x.B'$. Thus, there is a left-introduction phase ending in $\Sigma : \Gamma \Downarrow B'[t/x] \vdash A$ and inference rule

$$\frac{\Sigma : \Gamma \Downarrow B'[t/x] \vdash A}{\Sigma : \Gamma \Downarrow \forall x.B' \vdash A} .$$

for some Σ-term t. By the inductive assumption, there is a path in $B'[t/x]$ with target A', arguments $\mathcal{B}$, and bound variables $\mathcal{X}$, and a substitution θ that maps the variables in $\mathcal{X}$ to Σ-terms such that $A'\theta$ is equal to A and such that every premise of that left-introduction phase can be written as $\Sigma : \Gamma \vdash G\theta$ for each $G \in \mathcal{B}$. The required path through $\forall x.B'$ is then the same as for $B'[t/x]$ except that the required substitution is θ extended with the mapping of x to t. The converse can be proved similarly. $\square$

Note the dual use of paths: *all* paths of B are used to describe the right-introduction phase with endsequent $\Sigma : \Gamma \vdash B$, while *some* path of B is used to describe the left-introduction phase with endsequent $\Sigma : \Gamma \Downarrow B \vdash A$.

Exercise 5.19. Prove that if the sequent $\Sigma : \Gamma, B \vdash G$ has a proof Ξ in which no occurrences of *decide* pick the formula B as its focus, then there is a proof Ξ' of $\Sigma : \Gamma \vdash G$ that has the same tree structure of inference rules: the only difference is the sequents labeling those inference rules. This operation of removing an assumption in a sequent is called *strengthening*.

We are now able to prove the three main theorems related to $\Downarrow\mathcal{L}_0$-proofs: the admissibility of the (non-atomic) *init* rule, the admissibility of *cut*, and the completeness of $\Downarrow\mathcal{L}_0$-proofs with respect to intuitionistic provability.

Theorem 5.20 (Admissibility of initial). *Let Γ be a multiset of $\mathcal{L}_0$ Σ-formulas. If $B \in \Gamma$ then $\Sigma : \Gamma \vdash B$ has an $\Downarrow\mathcal{L}_0$-proof.*

$$\frac{\Sigma : \Gamma \vdash B \qquad \Sigma : \Gamma, B \vdash C}{\Sigma : \Gamma \vdash C}\ \mathit{cut}$$

Figure 5.2: The cut inference rule used in $\Downarrow^{+}\mathcal{L}_0$-proofs. The cut-formula B is restricted to be an $\mathcal{L}_0$-formula.

Proof. We describe how to build an $\Downarrow\mathcal{L}_0$-proof of $\Sigma : \Gamma \vdash B$ by induction on the structure of the $\mathcal{L}_0$ formula B. We first consider the right-introduction phase with the endsequent $\Sigma : \Gamma \vdash B$. By Proposition 5.17, for every path P in B, there is a premise sequent of that right-introduction phase of the form $\Sigma, \mathcal{X} : \Gamma, \mathcal{B} \vdash A$, where A, $\mathcal{B}$, and $\mathcal{X}$ are, respectively, the target, arguments, and bound variables of P. Now consider the premise that corresponds to P and use the *decide* rule to select $B \in \Gamma$ in order to initiate a left-introduction phase. By Proposition 5.18, there is a left-introduction phase that corresponds to P. By setting θ to the identity substitution on the variables in $\mathcal{X}$, we have $A = A\theta$ and where the left-introduction phase has the premises (where, $\mathcal{B} = \{B_1, \ldots, B_n\}$)

$$\Sigma, \mathcal{X} : \Gamma, \mathcal{B} \vdash B_1 \quad , \ldots , \quad \Sigma, \mathcal{X} : \Gamma, \mathcal{B} \vdash B_n \quad (n \geq 0).$$

We can conclude now by using the inductive hypotheses on each of these premises. $\square$

We next turn our attention to proving the cut-elimination theorem for $\Downarrow\mathcal{L}_0$-proofs. Figure 5.2 introduces the cut rule for the focused proof system for $\mathcal{L}_0$. The *cut* rule involves three sequents, none of which contains the $\Downarrow$. The proof system that combines the inference rules in the $\Downarrow\mathcal{L}_0$-proof system and in Figure 5.2 is called the $\Downarrow^{+}\mathcal{L}_0$ proof system, and proofs in that system will be called $\Downarrow^{+}\mathcal{L}_0$-proofs.

We introduce the following two measures. The size of a formula B, written as $|B|$, is the number of occurrences of logical connectives in B. The *size* of a formula is 0 if and only if that formula is an atom. The *height* of an $\Downarrow^{+}\mathcal{L}_0$-proof Ξ, also written as $|\Xi|$, is the maximum number of inference rules on a path in Ξ that does not go through a left premise of a cut rule: that is, the height of a proof that ends in a cut rule is one more than the height of its right premise. This height is always greater than or equal to 1.

The following two propositions can be proved by simple inductions on the structure of $\Downarrow\mathcal{L}_0$-proofs.

Proposition 5.21 (Weakening $\Downarrow^{+}\mathcal{L}_0$-proofs)**.** *Let Σ and Σ' be signatures such that $\Sigma \subseteq \Sigma'$ and let Γ and Γ' be two multisets of $\mathcal{L}_0$ formulas such that $\Gamma \subseteq \Gamma'$.*

If $\Sigma : \Gamma \vdash B$ has an $\Downarrow^{+}\mathcal{L}_0$-proof of height h then $\Sigma' : \Gamma' \vdash B$ has an $\Downarrow^{+}\mathcal{L}_0$-proof of height h.

Proposition 5.22 (Substitution into $\Downarrow\mathcal{L}_0$-proofs). *Let Σ be a signature, x be a variable not declared in Σ, and τ a primitive type. If $\Sigma, x : \tau : \Gamma \vdash B$ has an $\Downarrow^{+}\mathcal{L}_0$-proof of height h and t is a Σ-term of type τ then $\Sigma : \Gamma[t/x] \vdash B[t/x]$ has an $\Downarrow^{+}\mathcal{L}_0$-proof of height h.*

To prove the cut-elimination theorem for $\Downarrow^{+}\mathcal{L}_0$ proofs, we introduce a second cut rule, called the *key cut* rule (here, A is an atomic formula and B is an $\mathcal{L}_0$ formula).

$$\frac{\Sigma : \Gamma \vdash B \qquad \Sigma : \Gamma \Downarrow B \vdash A}{\Sigma : \Gamma \vdash A}\ cut_k$$

This cut rule is only used as a technical device to help prove cut-elimination. A *cut-free proof* is a proof that does not contain occurrences of either the *cut* or cut_k rule. Clearly, a cut-free $\Downarrow^{+}\mathcal{L}_0$-proof is an $\Downarrow\mathcal{L}_0$-proof. The height of a proof containing cut_k is defined as above but this time *cut* and cut_k are treated the same: in particular, the height of a proof that ends in the cut_k rule is one more than the height of its right premise.

Lemma 5.23. *Consider an occurrence of the cut rule of the form*

$$\frac{\begin{array}{c}\Xi_l\\ \Sigma : \Gamma \vdash B\end{array} \qquad \begin{array}{c}\Xi_r\\ \Sigma : \Gamma, B \vdash C\end{array}}{\Sigma : \Gamma \vdash C}\ cut,$$

where Ξ_l and Ξ_r are (cut-free) $\Downarrow\mathcal{L}_0$-proofs. We can transform this proof into a proof of $\Sigma : \Gamma \vdash C$ of smaller height in which there are no occurrences of the cut rule, but there might be several occurrences of the cut_k rule, all of which have cut-formula B.

Proof. Let Ξ_l be a $\Downarrow\mathcal{L}_0$-proof of $\Sigma : \Gamma \vdash B$ and let Ξ_r be a $\Downarrow\mathcal{L}_0$-proof of $\Sigma : \Gamma, B \vdash C$. We first convert Ξ_r to a new proof Ξ_r' also of $\Sigma : \Gamma, B \vdash C$ by replacing every occurrence of the *decide* rule applied to the cut formula B within Ξ_r, such as

$$\frac{\begin{array}{c}\Xi_0\\ \Sigma' : \Gamma', B \Downarrow B \vdash A\end{array}}{\Sigma' : \Gamma', B \vdash A}\ decide$$

(where $\Sigma \subseteq \Sigma'$ and $\Gamma \subseteq \Gamma'$), with the following occurrence of a cut_k rule

$$\frac{\begin{array}{c}\hat{\Xi}_l\\ \Sigma' : \Gamma' \vdash B\end{array} \qquad \begin{array}{c}\Xi_0\\ \Sigma' : \Gamma', B \Downarrow B \vdash A\end{array}}{\Sigma' : \Gamma', B \vdash A}\ cut_k.$$

Here $\hat{\Xi}_l$ is the result of weakening Ξ_l (Proposition 5.21). The resulting proof Ξ'_r has no occurrences of *decide* on B but many have several occurrences of cut_k with cut-formula B in Ξ'_r. Note that the height of Ξ_r and Ξ'_r is the same and that Ξ'_r is a proof of $\Sigma : \Gamma, B \vdash C$. Furthermore, since there are no occurrences of *decide* on B in Ξ'_r, we can strengthen Ξ'_r to get a proof Ξ_s of $\Sigma : \Gamma \vdash C$ with the same height as Ξ_r (proved by a simple induction on the structure of proofs, see Exercise 5.19). As a result, we can replace the original proof of $\Sigma : \Gamma \vdash C$ with the new proof Ξ_s with smaller height than Ξ_r. □

Lemma 5.24. *Consider an occurrence of the cut_k rule of the form*

$$\dfrac{\begin{matrix}\Xi_l\\ \Sigma : \Gamma \vdash B\end{matrix} \qquad \begin{matrix}\Xi_r\\ \Sigma : \Gamma \Downarrow B \vdash C\end{matrix}}{\Sigma : \Gamma \vdash C}\ cut_k,$$

where Ξ_l and Ξ_r are $\Downarrow^+\mathcal{L}_0$ proofs. We can transform this proof into a proof of $\Sigma : \Gamma \vdash C$ where this occurrence of cut_k is replaced with occurrences of the cut rule in which the cut-formulas are strictly smaller than B.

Proof. Consider an occurrence of the cut_k rule

$$\dfrac{\begin{matrix}\Xi_l\\ \Sigma : \Gamma \vdash B\end{matrix} \qquad \begin{matrix}\Xi_r\\ \Gamma \Downarrow B \vdash A\end{matrix}}{\Sigma : \Gamma \vdash A}\ cut_k,$$

where Ξ_l and Ξ_r are $\Downarrow^+\mathcal{L}_0$ proofs. If B is atomic, then B and A are equal and the result of eliminating this cut_k is Ξ_l. Thus, assume that B is not atomic. In that case, Ξ_l ends in a non-empty right-introduction phase and Ξ_r ends in a left-introduction phase. By Proposition 5.18, there is a path P in B with associated sequent $\mathcal{X} : B_1, \ldots, B_n \vdash A'$ such that the premises and subproofs of that left-introduction phase are

$$\begin{matrix}\Xi_1\\ \Sigma : \Gamma \vdash B_1\theta,\end{matrix}\ \ \ldots\ \ ,\begin{matrix}\Xi_n\\ \Sigma : \Gamma \vdash B_n\theta\end{matrix} \quad (n \geq 0)$$

and where $A'\theta$ is A, for some substitution θ. By Proposition 5.17, there is a premise in the right-introduction phase that corresponds to path P and is the sequent $\Sigma, \mathcal{X}{:}\Gamma, B_1, \ldots, B_n \vdash A'$ with its subproof Ξ_0. By repeated application of Proposition 5.22, we know that the sequent $\Sigma : \Gamma, B_1\theta, \ldots, B_n\theta \vdash A'\theta$ has a $\Downarrow\mathcal{L}_0$-proof, say, $\Xi_0\theta$. If we take these various $\Downarrow^+\mathcal{L}_0$-proofs and arrange them as follows, we have a proof in which the *cut* rule has n occurrences (remembering that A is equal to $A'\theta$).

$$\dfrac{\overset{\Xi_1}{\Sigma:\Gamma\vdash B_1\theta} \qquad \overset{\Xi_0\theta}{\Sigma:\Gamma,B_1\theta,\ldots,B_n\theta\vdash A}}{\Sigma:\Gamma,B_2\theta,\ldots,B_n\theta\vdash A}\ cut$$

$$\vdots$$

$$\dfrac{\overset{\Xi_n}{\Sigma:\Gamma\vdash B_n\theta} \qquad \Sigma:\Gamma,B_n\theta\vdash A}{\Sigma:\Gamma\vdash A}\ cut$$

Note that the size of each of the cut formulas $B_1\theta,\ldots,B_n\theta$ is strictly less than the size of the original cut formula B. □

Thus, Lemma 5.23 describes how one occurrence of *cut* on B can be replaced with several occurrences of cut_k on B, and Lemma 5.24 describes how an occurrence of cut_k on B can be replaced by several occurrences of *cut* on strictly smaller formulas than B.

Lemma 5.25. *An $\Downarrow^+\mathcal{L}_0$ proof that ends with a cut rule in which both premises have cut-free proofs can be replaced with a cut-free proof of the same endsequent.*

Proof. Consider the following occurrence of the *cut* inference rule

$$\dfrac{\overset{\Xi_l}{\Sigma:\Gamma\vdash B} \qquad \overset{\Xi_r}{\Sigma:\Gamma,B\vdash C}}{\Sigma:\Gamma\vdash C}\ cut$$

in which Ξ_l and Ξ_r are (cut-free) $\Downarrow\mathcal{L}_0$-proofs. We will show that the sequent $\Sigma:\Gamma\vdash C$ has a cut-free $\Downarrow\mathcal{L}_0$-proof by induction of the size of the cut formula B. First, apply Lemma 5.23 to conclude that there is a proof Ξ' of $\Sigma:\Gamma\vdash C$ that contains no occurrences of *cut* but which might have several instances of the cut_k rule with cut formula B. We can now do a second induction on the number of occurrences of cut_k in Ξ'. If that number is 0, then the proof Ξ' is the desired cut-free proof. Otherwise, assume that there is at least one occurrence of cut_k on B in Ξ'. If we pick an upper-most occurrence of cut_k and apply Lemma 5.24, we can convert that occurrence of cut_k to several occurrences of *cut* on strictly smaller formulas than B. By applying the inductive assumption, all of these occurrences of *cut* can be eliminated. We have now reduced the number of cut_k inference rules, and, hence, we have completed our proof. □

We can bring these lemmas together to prove the main cut-elimination theorem for $\Downarrow^+\mathcal{L}_0$ proofs.

Theorem 5.26 (Elimination of cuts)**.** *Let $\Gamma\cup\{G\}$ be a multiset of $\mathcal{L}_0$ Σ-formulas. If the sequent $\Sigma:\Gamma\vdash G$ has an $\Downarrow^+\mathcal{L}_0$-proof then it has an $\Downarrow\mathcal{L}_0$-proof.*

Proof. The proof is now a simple induction on the number of occurrences of the *cut* inference rules in a proof. In particular, pick an occurrence of the *cut* rule, which is the endsequent of a subproof in which both premises have cut-free proofs. By applying Lemma 5.25 to that occurrence of *cut*, we can replace it for a cut-free proof of the same sequent. The proof now follows from the inductive assumption. ☐

A consequence of the cut-elimination theorem for $\Downarrow^{+}\mathcal{L}_0$ proofs is the completeness of $\Downarrow\mathcal{L}_0$-proofs with respect of **I**-proofs (when all formulas are restricted to $\mathcal{L}_0$).

Theorem 5.27 (Completeness of $\Downarrow\mathcal{L}_0$-proofs for $\mathcal{L}_0$ formulas)**.** *Let $\Gamma \cup \{G\}$ be a multiset of $\mathcal{L}_0$ formulas. If the sequent $\Sigma : \Gamma \vdash G$ has a cut-free **I**-proof then it has an $\Downarrow\mathcal{L}_0$-proof.*

For convenience, we use the notation $\Sigma : \mathcal{P} \vdash_{\Downarrow} G$ to denote the proposition that the sequent $\Sigma : \mathcal{P} \vdash G$ has a $\Downarrow\mathcal{L}_0$-proof.

Proof. We prove this by showing that the inference rules of the intuitionistic proof system **I** are admissible in the $\Downarrow\mathcal{L}_0$-proof system. Since the right-introduction rules of **I** are the same as those in $\Downarrow\mathcal{L}_0$, these rules are trivially admissible. The admissibility of the *init* rule for **I** follows immediately from Proposition 5.20. The admissibility of the *wL* rule follows from Proposition 5.21. The admissibility of the *cL* rule is easily argued as follows. In an $\Downarrow\mathcal{L}_0$-proof of $\Sigma : \Gamma, B, B \vdash \Delta$, the *decide* rule may have been used on the two different occurrences of B. By changing all those *decide* rules to use the same occurrence of B and then deleting the other occurrence of B, we obtain an $\Downarrow\mathcal{L}_0$-proof of $\Sigma{:}\Gamma, B \vdash \Delta$. All that remains to show is that the left-introduction rules for the $\mathcal{L}_0$ connectives $\wedge$, $\supset$, and $\forall$ are admissible.

Admissibility of $\wedge L$. Assume that $B_1 \wedge B_2$ is an $\mathcal{L}_0$ Σ-formula. By Proposition 5.20, we have $\Sigma : B_1 \wedge B_2 \vdash_{\Downarrow} B_1 \wedge B_2$. An $\Downarrow\mathcal{L}_0$-proof of that sequent has immediate subproofs that yield both $\Sigma : B_1 \wedge B_2 \vdash_{\Downarrow} B_1$ and $\Sigma : B_1 \wedge B_2 \vdash_{\Downarrow} B_2$. In order to prove that $\wedge$L is admissible, assume that $\Sigma : B_1, \Gamma \vdash_{\Downarrow} E$. Using cut-admissibility (Theorem 5.26) with this sequent and the sequent $\Sigma : B_1 \wedge B_2 \vdash_{\Downarrow} B_1$, we conclude that $\Sigma : B_1 \wedge B_2, \Gamma \vdash_{\Downarrow} E$. A similar argument also concludes that if $\Sigma : B_2, \Gamma \vdash_{\Downarrow} E$, then $\Sigma : B_1 \wedge B_2, \Gamma \vdash_{\Downarrow} E$. Hence, both $\wedge$L rules in **I** are admissible.

Admissibility of $\supset L$. Assume that $B_1 \supset B_2$ is an $\mathcal{L}_0$ Σ-formula. By Proposition 5.20, we have $\Sigma : B_1 \supset B_2 \vdash_{\Downarrow} B_1 \supset B_2$. An $\Downarrow\mathcal{L}_0$-proof of that sequent has an immediate subproof that proves $\Sigma : B_1, B_1 \supset B_2 \vdash_{\Downarrow} B_2$. In order to prove that $\supset$L is admissible, assume that both $\Sigma : \Gamma_1 \vdash_{\Downarrow} B_1$ and $\Sigma : B_2, \Gamma_2 \vdash_{\Downarrow} E$. Using the Proposition 5.21, we have $\Sigma : \Gamma_1, \Gamma_2 \vdash_{\Downarrow} B_1$ and $\Sigma : B_2, \Gamma_1, \Gamma_2 \vdash_{\Downarrow} E$. Using cut-admissibility (Theorem 5.26), we conclude that

$\Sigma : \Gamma_1, \Gamma_2, B_1 \supset B_2 \vdash_{\Downarrow} B_2$ and $\Sigma : B_1 \supset B_2, \Gamma_1, \Gamma_2 \vdash_{\Downarrow} E$. Hence, the $\supset$L rule in **I** is admissible.

Admissibility of $\forall L$. Assume that $\forall_\tau x.B$ is an $\mathcal{L}_0$ Σ-formula and that τ is a primitive type. By Proposition 5.20, we have $\Sigma : \forall_\tau x.B \vdash_{\Downarrow} \forall_\tau x.B$. An $\Downarrow\mathcal{L}_0$-proof of that sequent has an immediate subproof that proves $\Sigma, y : \tau : \forall x.B \vdash_{\Downarrow} B[y/x]$, for a variable y not present in Σ. By Proposition 5.22, we have $\Sigma : \forall x.B \vdash_{\Downarrow} B[t/x]$, for any Σ-term t. In order to prove that $\forall$L is admissible, assume that $\Sigma : B[t/x], \Gamma \vdash E$ has an $\Downarrow\mathcal{L}_0$-proof. Then using cut elimination (Theorem 5.26), we can conclude that $\Sigma : \forall x.B, \Gamma \vdash E$ has an $\Downarrow\mathcal{L}_0$-proof. Hence, the $\forall$L rule in **I** is admissible. □

Another simple consequence of proving the cut-elimination for $\Downarrow^{+}\mathcal{L}_0$-proofs is the admissibility of cut for **I**-proofs when restricted to $\mathcal{L}_0$ formulas.

Theorem 5.28 (Admissibility of cut for **I**-proofs restricted to $\mathcal{L}_0$ formulas)**.** *The cut rule for **I**-proofs (Figure 4.2) is admissible for cut-free **I**-proofs when restricted to $\mathcal{L}_0$ formulas.*

Proof. We wish to prove that the single-conclusion version of the cut rule from Figure 4.2, namely,

$$\frac{\Sigma : \Gamma_1 \vdash B \qquad \Sigma : B, \Gamma_2 \vdash E}{\Sigma : \Gamma_1, \Gamma_2 \vdash E}\ \textit{cut}$$

is admissible in the cut-free **I**-proof system. Thus, assume that $\Sigma : \Gamma_1 \vdash B$ and $\Sigma : B, \Gamma_2 \vdash E$ have (cut-free) **I**-proofs. By Theorem 5.27, $\Sigma : \Gamma_1 \vdash B$ and $\Sigma : B, \Gamma_2 \vdash E$ have $\Downarrow\mathcal{L}_0$-proofs. Using Proposition 5.21, both $\Sigma : \Gamma_1, \Gamma_2 \vdash B$ and $\Sigma : B, \Gamma_1, \Gamma_2 \vdash E$ have $\Downarrow\mathcal{L}_0$-proofs. Using the admissibility of the *cut* rule (Proposition 5.26), we know that $\Sigma : \Gamma_1, \Gamma_2 \vdash E$ has an $\Downarrow\mathcal{L}_0$-proof. Using the soundness of $\Downarrow\mathcal{L}_0$-proofs (Proposition 5.15), we conclude that $\Sigma : \Gamma_1, \Gamma_2 \vdash E$ has an **I**-proof. □

The inference rule (where all formulas are $\mathcal{L}_0$ formulas)

$$\frac{\Sigma \Vdash t : \tau \qquad \Sigma, x : \tau\ : \Gamma \vdash B}{\Sigma : \Gamma[t/x] \vdash B[t/x]}\ \textit{instan}$$

is similar to the cut rule: the *instan* rule instantiates an eigenvariable while the *cut* rule instantiates a hypothesis. The following theorem shows that this inference rule is admissible for **I**-proofs. The proof of this theorem is similar and more straightforward than the one for cut-elimination. This theorem is a direct consequence of Proposition 5.22.

Theorem 5.29 (Substitution into **I**-proofs of $\mathcal{L}_0$ formulas)**.** *Let Σ be a signature, y be a variable not in Σ, τ be a primitive type, and $\Gamma \cup \{B\}$ are $\mathcal{L}_0$ formulas. If $\Sigma, y : \tau : \Gamma \vdash B$ has an **I**-proof and if Σ-term t of type τ, then $\Sigma : \Gamma[t/x] \vdash B[t/x]$ has an **I**-proof.*

5.6 Kripke model semantics

In this book, we do not generally deal with model theory. There is, however, a nice connection between a specific *Kripke model* and the proof theory of intuitionistic logic. In this section, we recast the cut-elimination result for $\Downarrow^{+}\mathcal{L}_0$ proofs in terms of truth in a *canonical Kripke model* for $\mathcal{L}_0$. This model is canonical in the sense that whenever this model makes a given $\mathcal{L}_0$-formula true, that formula is true in *all* Kripke models for $\mathcal{L}_0$.

A *dependent pair* is a pair $\langle\Sigma,\mathcal{P}\rangle$ where Σ is a (finite) signature and $\mathcal{P}$ is a (finite) set of $\mathcal{L}_0$ Σ-formulas. A dependent pair is also called a *world*. The order relation on worlds $\langle\Sigma,\mathcal{P}\rangle \preceq \langle\Sigma',\mathcal{P}'\rangle$ is defined to hold whenever $\Sigma\subseteq\Sigma'$ and $\mathcal{P}\subseteq\mathcal{P}'$. A *Kripke model* is a pair, $\langle\mathcal{W},I\rangle$, where $\mathcal{W}$ is a (possibly infinite) set of worlds and I is a function, called an *interpretation*, that maps the worlds in $\mathcal{W}$ to sets of atomic formulas in such a way that $I(\langle\Sigma,\mathcal{P}\rangle)$ is a set of atomic Σ-formulas. The mapping I must also be order preserving: that is, for all $w,w'\in\mathcal{W}$, if $w\preceq w'$ then $I(w)\subseteq I(w')$.

Let the pair $\langle\mathcal{W},I\rangle$ be a Kripke model, let $\langle\Sigma,\mathcal{P}\rangle\in\mathcal{W}$, and let B be a $\mathcal{L}_0$ Σ-formula. The three place *satisfaction* relation $I,\langle\Sigma,\mathcal{P}\rangle\Vdash B$ is defined by induction on the structure of B as follows.

- $I,\langle\Sigma,\mathcal{P}\rangle\Vdash B$ if B is atomic and $B\in I(\langle\Sigma,\mathcal{P}\rangle)$.
- $I,w\Vdash B\wedge B'$ if $I,w\Vdash B$ and $I,w\Vdash B'$.
- $I,w\Vdash B\supset B'$ if for every $w'\in\mathcal{W}$ such that $w\preceq w'$ and $I,w'\Vdash B$ then $I,w'\Vdash B'$.
- $I,\langle\Sigma,\mathcal{P}\rangle\Vdash\forall_\tau x.B$ if for every $\langle\Sigma',\mathcal{P}'\rangle\in\mathcal{W}$ such that $\langle\Sigma,\mathcal{P}\rangle\preceq\langle\Sigma',\mathcal{P}'\rangle$ and for every Σ'-term t of type τ, the relation $I,\langle\Sigma',\mathcal{P}'\rangle\Vdash B[t/x]$ holds.

Let $\langle\Sigma,\mathcal{P}\rangle$ be a dependent pair. The *canonical model* for $\langle\Sigma,\mathcal{P}\rangle$ is defined as the Kripke model with the set of worlds $\{\langle\Sigma',\mathcal{P}'\rangle \mid \langle\Sigma,\mathcal{P}\rangle\preceq\langle\Sigma',\mathcal{P}'\rangle\}$ and the interpretation I defined so that $I(\langle\Sigma',\mathcal{P}'\rangle)$ is the set of all atomic Σ'-formulas A such that $\Sigma':\mathcal{P}'\vdash A$ has a *cut-free* **I**-proof.

Note the rather different way provability and satisfaction treat an implicational formula. In order to prove the formula $B_1\supset B_2$ in the world $\langle\Sigma,\mathcal{P}\rangle$ (i.e., that the sequent $\Sigma:\mathcal{P}\vdash B_1\supset B_2$ is provable), we need to move to a single new world $\langle\Sigma,\mathcal{P}\cup\{B_1\}\rangle$ and try to prove B_2. In contrast, in order to show that $B_1\supset B_2$ is true in the world $\langle\Sigma,\mathcal{P}\rangle$ we need to examine *all* extensions to that world and check that B_2 is true in that world if B_1 is true in that world.

As we mentioned in Section 3.6, sequent calculus inference rules provide logical connectives with *two senses* within a proof: namely, there are different inference rules for introducing a given logical connective on the left and the right of a sequent. On the other hand, in the model-theoretic setting, logical connectives are given meaning in only one sense: there is only one clause defining the satisfiability of a given logical connective. The following lemma shows how the cut-admissibility result allows us to relate these different approaches to providing meaning to logical connectives.

Lemma 5.30. *The cut rule (Figure 5.2) and the instan rule (defined at the end of Section 5.5) are admissible for cut-free* ***I****-proofs if and only if the following holds: For every dependent pair $\langle\Sigma,\mathcal{P}\rangle$ and every Σ-formula B, it is the case that $\Sigma:\mathcal{P}\vdash B$ has a cut-free* ***I****-proof if and only if $I,\langle\Sigma,\mathcal{P}\rangle\Vdash B$, where I is the canonical model for $\langle\Sigma,\mathcal{P}\rangle$.*

In other words, the admissibility of *cut* and *instan* is equivalent to the fact that provability coincides with truth in the canonical model.

Proof. To prove the forward direction, assume that both the *cut* and *instan* rules are admissible for **I**-proofs. We now prove by induction on the structure of B that $\Sigma:\mathcal{P}\vdash_I B$ if and only if $I,\langle\Sigma,\mathcal{P}\rangle\Vdash B$.

Case: B is atomic. The equivalence is immediate.

Case: B is $B_1\wedge B_2$. This case is simple and immediate.

Case: B is $B_1\supset B_2$. Assume first that $\Sigma:\mathcal{P}\vdash_I B_1\supset B_2$. Hence, $\Sigma:\mathcal{P},B_1\vdash_I B_2$ (using the soundness and completeness of $\Downarrow\mathcal{L}_0$-proofs). To show $I,\langle\Sigma,\mathcal{P}\rangle\Vdash B_1\supset B_2$, assume that $\langle\Sigma',\mathcal{P}'\rangle\in\mathcal{W}$ is such that $\langle\Sigma,\mathcal{P}\rangle\preceq\langle\Sigma',\mathcal{P}'\rangle$ and $I,\langle\Sigma',\mathcal{P}'\rangle\Vdash B_1$. By the inductive hypothesis, $\Sigma':\mathcal{P}'\vdash_I B_1$ and by cut admissibility, $\Sigma':\mathcal{P}'\vdash_I B_2$. By induction again, we have $I,\langle\Sigma',\mathcal{P}'\rangle\Vdash B_2$. Thus, $I,\langle\Sigma,\mathcal{P}\rangle\Vdash B_1\supset B_2$. For the converse, assume $I,\langle\Sigma,\mathcal{P}\rangle\Vdash B_1\supset B_2$. Since $\Sigma:\mathcal{P},B_1\vdash_I B_1$, the inductive hypothesis yields $I,\langle\Sigma,\mathcal{P}\cup\{B_1\}\rangle\Vdash B_1$. By the definition of satisfaction of implication we must have $I,\langle\Sigma,\mathcal{P}\cup\{B_1\}\rangle\Vdash B_2$. Using the inductive hypothesis again, $\Sigma:\mathcal{P},B_1\vdash_I B_2$, and $\Sigma:\mathcal{P}\vdash_I B_1\supset B_2$.

Case: B is $\forall_\tau x.B_1$. Assume first that $\Sigma:\mathcal{P}\vdash_I\forall_\tau x.B_1$ and, hence, $\Sigma,d:\tau:\mathcal{P}\vdash_I B_1[d/x]$ for any variable d not in Σ. To show that $I,\langle\Sigma,\mathcal{P}\rangle\Vdash\forall_\tau x.B_1$, let $\langle\Sigma',\mathcal{P}'\rangle\in\mathcal{W}$ be such that $\langle\Sigma,\mathcal{P}\rangle\preceq\langle\Sigma',\mathcal{P}'\rangle$ and t be a Σ'-term of type τ. By the admissibility of the *instan* rule, we have $\Sigma';\mathcal{P}'\vdash_I B_1[t/x]$. By induction we have $I,\langle\Sigma',\mathcal{P}'\rangle\Vdash B_1[t/x]$. Thus, $I,\langle\Sigma,\mathcal{P}\rangle\Vdash\forall_\tau xB_1$. For the converse, assume $I,\langle\Sigma,\mathcal{P}\rangle\Vdash\forall_\tau xB_1$. Let d be a variable not a member of Σ. Since d is a $\Sigma\cup\{d\}$-term, $I,\langle\Sigma\cup\{d\},\mathcal{P}\rangle\Vdash B_1[d/x]$ by the definition of satisfaction of universal quantification. But by the inductive hypothesis again, $\Sigma,d:\tau;\mathcal{P}\vdash_I B_1[d/x]$ and $\Sigma:\mathcal{P}\vdash_I\forall_\tau xB_1$.

We now show the converse by assuming the equivalence: for every dependent pair $\langle\Sigma,\mathcal{P}\rangle$ and every Σ-formula B,

$$\Sigma:\mathcal{P}\vdash_I B \text{ if and only if } I,\langle\Sigma,\mathcal{P}\rangle\Vdash B,$$

where I is the canonical model for $\langle\Sigma,\mathcal{P}\rangle$. We now show that any sequent that can be proved using occurrences of the *cut* and *instan* rules can be proved without such rules. In particular, we claim that if $\langle\Sigma,\mathcal{P}\rangle \preceq \langle\Sigma',\mathcal{P}'\rangle$ then each of the following holds.

1. If $\Sigma';\mathcal{P}' \vdash_I B$ and $\Sigma:\mathcal{P},B \vdash_I C$ then $\Sigma';\mathcal{P}' \vdash_I C$.
2. If t is a Σ'-term of type τ and $\Sigma,x:\tau:\mathcal{P} \vdash_I B$ then $\Sigma':\mathcal{P}' \vdash_I B[t/x]$ (of course, x does not occur in Σ).

To prove the first claim, assume that $\Sigma':\mathcal{P}' \vdash_I B$ and $\Sigma:\mathcal{P},B \vdash_I C$. Thus, $\Sigma:\mathcal{P} \vdash_I B \supset C$. By the assumed equivalence, $I,\langle\Sigma',\mathcal{P}'\rangle \Vdash B$ and $I,\langle\Sigma,\mathcal{P}\rangle \Vdash B \supset C$. By the definition of satisfaction for implication, $I,\langle\Sigma',\mathcal{P}'\rangle \Vdash C$. By the assumed equivalence again, this yields $\Sigma':\mathcal{P}' \vdash_I C$.

To prove the second claim above, assume that t is a Σ'-term of type τ and that $\Sigma,x:\tau:\mathcal{P} \vdash_I C$. Thus, $\Sigma:\mathcal{P} \vdash_I \forall_\tau x.B$. By the assumed equivalence, $I,\langle\Sigma,\mathcal{P}\rangle \Vdash \forall_\tau x.B$. By the definition of satisfaction for universal quantification, we have $I,\langle\Sigma',\mathcal{P}'\rangle \Vdash B[t/x]$. By the assumed equivalence again, this yields $\Sigma':\mathcal{P}' \vdash_I B[t/x]$. $\square$

Given Theorems 5.26 and 5.29, this lemma provides an immediate proof of the following theorem.

Theorem 5.31. *Let $\langle\Sigma,\mathcal{P}\rangle$ be a dependent pair and let I be the canonical model for $\langle\Sigma,\mathcal{P}\rangle$. For all Σ-formulas B, $\Sigma:\mathcal{P} \vdash_I B$ if and only if $I \Vdash B$. In particular, for every $B \in \mathcal{P}$, $I \Vdash B$.*

The following simple argument supports our use of the term "canonical model". While we have not given a general definition of Kripke models (i.e., a notion of model that is not built from formulas and terms), whatever definition is used, they need to be sound: that is, if $\vdash_I B$ then B is true in every generalized Kripke model. Thus, if the $\mathcal{L}_0$ Σ-formula B is true in the canonical model for $\langle\Sigma,\emptyset\rangle$ then $\Sigma{:}\cdot \vdash_I B$ and, hence, B is true in every generalized Kripke model.

5.7 Backchaining as a single left rule

We can use the $\Downarrow\mathcal{L}_0$-proof system to define backchaining as a single inference rule instead of as a sequence of inference rules. In particular, let Σ be a signature and let Δ be a finite set of Σ-formulas. Define $|\Delta|_\Sigma$ to be the smallest set of pairs $\langle\Gamma,D\rangle$, where Γ is a multiset of formulas and D is a formula, such that

- if $D \in \Delta$ then $\langle\emptyset,D\rangle \in |\Delta|_\Sigma$,

- if $\langle \Gamma, D_1 \wedge D_2 \rangle \in |\Delta|_\Sigma$ then $\langle \Gamma, D_1 \rangle \in |\Delta|_\Sigma$ and $\langle \Gamma, D_2 \rangle \in |\Delta|_\Sigma$,
- if $\langle \Gamma, G \supset D \rangle \in |\Delta|_\Sigma$ then $\langle \Gamma \cup \{G\}, D \rangle \in |\Delta|_\Sigma$, and
- if $\langle \Gamma, \forall_\tau x\, D \rangle \in |\Delta|_\Sigma$ and t is a Σ-term of type τ then $\langle \Gamma, D[t/x] \rangle \in |\Delta|_\Sigma$.

Backchaining is now defined as the single inference rule

$$\frac{\{\Sigma : \Delta \vdash G \ \mid \ G \in \Gamma\}}{\Sigma : \Delta \vdash A}\ \mathrm{BC}, \quad \text{provided } A \text{ is atomic and } \langle \Gamma, A \rangle \in |\Delta|_\Sigma.$$

If Γ is empty, then this rule has no premises. Let the $\Downarrow\mathcal{L}_0'$-proof system contain the right-introduction rules in Figure 4.1 and the BC rule.

Straightforward inductive arguments prove the following two lemmas and proposition.

Lemma 5.32. *If P is a path in D (i.e., $D \uparrow P$ holds), and θ is a substitution, then $P\theta$ is a path in $D\theta$.*

Lemma 5.33. *Let Σ be an eigenvariable signature, let Γ be a multiset of Σ-formulas, and let $D \in \Gamma$. Then $\langle \Gamma, A \rangle \in |\{D\}|_\Sigma$ if and only if there is a path in D with bound variables $\bar{x}$, arguments $G_1, \ldots, G_n$ ($n \geq 0$), and target A' and there is a substitution θ mapping the variables $\bar{x}$ to Σ-terms such that Γ and $\{G_1\theta, \ldots, G_n\theta\}$ are equal and A and $A'\theta$ are equal.*

Proposition 5.34. *Let Σ be a signature, let $\mathcal{P}$ be a multiset of $\mathcal{L}_0$ Σ-formulas program and G be a Σ-formula. The sequent $\Sigma : \mathcal{P} \vdash G$ has an $\Downarrow\mathcal{L}_0'$-proof if and only if it has an **I**-proof.*

Proposition 5.35. *Let B be a propositional $\mathcal{L}_0$ formula: i.e., that is B contains only the logical connectives $\{\top, \wedge, \supset\}$. Show that it is decidable whether or not $\vdash_I B$ holds.*

Proof. Given the completeness of $\Downarrow \mathcal{L}_0'$-proofs (Proposition 5.34), we only need to find a decision procedure for $\Downarrow\mathcal{L}_0'$-proofs restricted to the connectives $\{\top, \wedge, \supset\}$. A systematic search for a such proofs can be described as follows. First, the provability of a non-border sequent can be reduced uniquely to the provability of border sequents. Second, the only inference rule in $\Downarrow\mathcal{L}_0'$ that has a border sequent as a conclusion is an instance of the backchaining rule BC and there are at most a finite number of such instances of BC that might be applicable. Finally, the only thing left to show is that the search space for this naive search procedure is finite. To show this, note that all border sequents $\Sigma : \Gamma \vdash A$ in an $\Downarrow\mathcal{L}_0'$-proof of $\vdash B$ are such that A is an atomic subformula of B and Γ is a finite multiset of subformulas of B. While there are an infinite number of finite *multisets* of subformulas of B there are only a finite number of finite *sets* of such subformulas. Also note that if Γ and Γ' are two multisets of formulas that are equal as sets (i.e., they differ only in the multiplicity of their members) then the border sequent $\Sigma : \Gamma \vdash A$ has an $\Downarrow\mathcal{L}_0'$-proof if and only if $\Sigma : \Gamma' \vdash A$ has an $\Downarrow\mathcal{L}_0'$-proof. As a result, the search space for determining whether or not $\vdash B$ has an $\Downarrow\mathcal{L}_0'$-proof can be described as the finite set of pairs $\langle \Delta, A \rangle$ where Δ is a set of subformulas of B and A is an atomic subformula of B. Thus, the naive search procedure can use this observation to ensure that it never loops and, in fact, always terminates. $\square$

5.8 Synthetic inference rules

One use of the two-phase $\Downarrow\mathcal{L}_0$-proof system is to justify replacing program clauses with inference rules. For example, consider a logic program $\mathcal{P}$ that consists of the two first-order Horn clauses

$$\forall x \forall y\ [\mathit{adj}\ x\ y \supset \mathit{path}\ x\ y] \quad \text{and} \quad \forall x \forall y \forall z\ [\mathit{adj}\ x\ y \wedge \mathit{path}\ y\ z \supset \mathit{path}\ x\ z].$$

Here, we are assuming that the two predicates *adj* and *path* have type $i \to i \to o$. Using the *decide* rule on the second of these formulas leads to an attempt to prove the sequent $\Sigma : \Gamma, \mathcal{P} \vdash \mathit{path}\ s\ t$ with the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \mathcal{P} \vdash \mathit{adj}\ s\ u \quad \Gamma, \mathcal{P} \vdash \mathit{path}\ u\ t}{\Gamma, \mathcal{P} \vdash \mathit{adj}\ s\ u \wedge \mathit{path}\ u\ t}\ \wedge L \qquad \dfrac{}{\Gamma, \mathcal{P} \Downarrow \mathit{path}\ s\ t \vdash \mathit{path}\ s\ t}\ \mathit{init}}{\Gamma, \mathcal{P} \Downarrow (\mathit{adj}\ s\ u \wedge \mathit{path}\ u\ t \supset \mathit{path}\ s\ t) \vdash \mathit{path}\ s\ t}\ \supset L}{\Gamma, \mathcal{P} \Downarrow \forall x \forall y \forall z\ (\mathit{adj}\ x\ y \wedge \mathit{path}\ y\ z \supset \mathit{path}\ x\ z) \vdash \mathit{path}\ s\ t}\ \forall L \times 3}{\Gamma, \mathcal{P} \vdash \mathit{path}\ s\ t}\ \mathit{decide}$$

(We suppressed the signatures associated with sequents for readability). If we ignore the seven inference rules within this derivation, we have the inference rule

$$\dfrac{\Sigma : \Gamma, \mathcal{P} \vdash \mathit{adj}\ s\ u \qquad \Sigma : \Gamma, \mathcal{P} \vdash \mathit{path}\ u\ t}{\Sigma : \Gamma, \mathcal{P} \vdash \mathit{path}\ s\ t}\ .$$

Similarly, deciding to use the first of these two formulas results in the inference rule

$$\dfrac{\Sigma : \Gamma, \mathcal{P} \vdash \mathit{adj}\ s\ t}{\Sigma : \Gamma, \mathcal{P} \vdash \mathit{path}\ s\ t}\ .$$

These latter inference rules are rather appealing since they do not mention any logical constants. Instead, they describe how an attempt to prove one atomic formula can lead to the attempt to prove one or two additional atomic formulas. Given this observation, we can *remove* these two Horn clauses from the logic program (assumptions on the left-hand side) and *insert* in the **I**-proof system the *synthetic inference rules*

$$\dfrac{\Sigma : \Gamma \vdash \mathit{adj}\ s\ t}{\Sigma : \Gamma \vdash \mathit{path}\ s\ t} \quad \text{and} \quad \dfrac{\Sigma : \Gamma \vdash \mathit{adj}\ s\ u \qquad \Sigma : \Gamma \vdash \mathit{path}\ u\ t}{\Sigma : \Gamma \vdash \mathit{path}\ s\ t}\ .$$

If we are using only Horn clauses, then it is possible to replace all program clauses in the left-hand context with synthetic inference rules that mention only atomic formulas.

More formally, we say that a sequent of the form $\Sigma : \Gamma \vdash A$, where A is an atomic formula, is a *border sequent* since such sequents appear at the border between a right-introduction phase (on the bottom) and a left-introduction phase (at the top). A synthetic inference rule is the inference rule that results from moving from a border sequent upwards through a *decide* rule and the left-introduction phase to the right-introduction phases: any open sequents will be border sequents.

While focusing on Horn clauses yields synthetic inference rules that only mention atoms, focusing on formulas of higher clause order leads to synthetic rules that contain logical connectives. For example, focusing on the propositional formula $((p \supset q) \supset r) \supset s$, which we assume is a member of Γ, would yield the synthetic inference rule

$$\frac{\Gamma, p \supset q \vdash r}{\Gamma \vdash s} \; .$$

We say that a synthetic inference rule in $\mathcal{L}_0$ is a *bipole* if that rule contains only atomic formulas in its conclusion and premises.

Exercise 5.36. Show that the synthetic inference rules that result from deciding on an $\mathcal{L}_0$ formula of clausal order at most 2 are bipoles.

It can be shown that the proof system that results from adding on top of the **I**-proof systems all the synthetic inference rules arising from a multiset of formulas of order two or less satisfies the cut admissibility property.

5.9 Disjunctive and existential goals

Now that we have addressed the soundness and completeness of $\Downarrow\mathcal{L}_0$-proofs for $\mathcal{L}_0$ formulas, we return to considering allowing disjunctions and existential quantifiers into formulas in the restricted setting of definition (5.5) of *fohh*. With this definition, **I**-proofs can have disjunctions and existential introduction rules on the right but not the left of its sequents. It turns out that we can capture the right-hand side proof-search behavior of these logical constants using *non-logical constants* as followings. Let $\hat{\vee}$ be a non-logical constant of type $o \to o \to o$ and $\hat{\exists}_\tau$ be a non-logical constant of type $(\tau \to o) \to o$ for every type τ. Consider the (infinite) set $\mathcal{C}$ of formulas that contains the two clauses

$$\forall_o P \, \forall_o Q \; [P \supset (P \mathbin{\hat{\vee}} Q)] \qquad \forall_o P \, \forall_o Q \; [Q \supset (P \mathbin{\hat{\vee}} Q)]$$

and, for every type τ, the clause

$$\forall_{\tau\to o} B\ \forall_\tau t\ [(B\ t) \supset (\hat{\exists}_\tau\ B)].$$

The members of $\mathcal{C}$ are Horn clauses, but they are not first-order Horn clauses since they contain quantifiers that are not of first-order type (since that type contains the type o). Such clauses are studied in more detail in Chapter 9 where we present *higher-order Horn clauses*. In that chapter, we will see that these higher-order clauses yield the following synthetic inference rules

$$\frac{\Sigma : \mathcal{P}, \mathcal{C} \vdash P}{\Sigma : \mathcal{P}, \mathcal{C} \vdash P \mathbin{\hat{\vee}} Q}\ ,\quad \frac{\Sigma : \mathcal{P}, \mathcal{C} \vdash Q}{\Sigma : \mathcal{P}, \mathcal{C} \vdash P \mathbin{\hat{\vee}} Q}\ ,\ \text{and}\quad \frac{\Sigma : \mathcal{P}, \mathcal{C} \vdash B\ t}{\Sigma : \mathcal{P}, \mathcal{C} \vdash \hat{\exists}_\tau B}\ .$$

Note that these rules exactly correspond to the $\vee$R and $\exists$R rules. Given this observation, we can now prove the following completeness theorem.

Proposition 5.37 (Completeness of $\Downarrow$*fohh*-proofs for *fohh*)**.** *Let Γ be an fohh logic program and G an fohh goal. If the sequent $\Sigma : \Gamma \vdash G$ has an **I**-proof then it has an $\Downarrow$fohh-proof.*

Proof. Assume that $\Sigma : \Gamma \vdash G$ has an **I**-proof Ξ. Let $\mathcal{C}(\Xi)$ be the smallest set of clauses such that the following holds. (When we write $\forall\Sigma'$ we mean a string of universal quantifiers, one for each variable in Γ'.)

1. If Ξ contains the inference rule

$$\frac{\Sigma, \Sigma' : \Gamma' \vdash B_i}{\Sigma, \Sigma' : \Gamma' \vdash B_1 \vee B_2}\ \vee\text{R}$$

   then $\mathcal{C}(\Xi)$ contains the clause $\forall\Sigma'[B_i \supset (B_1 \mathbin{\hat{\vee}} B_2)]$.

2. If Ξ contains the inference rule

$$\frac{\Sigma, \Sigma' \Vdash t : \tau \qquad \Sigma, \Sigma' : \Gamma' \vdash B[t/x]}{\Sigma, \Sigma' : \Gamma' \vdash \exists_\tau x.B}\ \exists\text{R}$$

   then $\mathcal{C}(\Xi)$ contains the clause $\forall\Sigma'[B[t/x] \supset (\hat{\exists}_\tau x.B)]$.

The set $\mathcal{C}(\Xi)$ is a set of essentially first-order Horn clauses: the only reason that they are not exactly members of *fohc* is that they can contain atomic formulas that might contain logical connectives (such atomic formulas have top-level symbols $\hat{\vee}$ and $\hat{\exists}$). Otherwise, only first-order quantification is used within these clauses. We shall assume here that this mild extension to *fohc* does not effect the proof theory results that we have already established for them. Chapter 9 will formally justify this assumption.

Let $\hat{\Gamma}$ and $\hat{G}$ be the result of replacing all occurrences of $\vee$ with $\hat{\vee}$ and of $\exists_\tau$ with $\hat{\exists}\tau$. It is now straightforward to convert the **I**-proof Ξ of $\Sigma : \Gamma \vdash G$ into an **I**-proof of $\Sigma : \mathcal{C}(\Xi), \hat{\Gamma} \vdash \hat{G}$. This conversion takes the rule

$$\frac{\Sigma, \Sigma' : \Gamma' \vdash B_i}{\Sigma, \Sigma' : \Gamma' \vdash B_1 \vee B_2}\ \vee R$$

and rewrites it into

$$\dfrac{\dfrac{\dfrac{\Sigma : \mathcal{C}(\Xi), \hat{\Gamma} \vdash \hat{B}_i \qquad \dfrac{}{\Sigma : \hat{B}_1 \mathbin{\hat{\vee}} \hat{B}_2 \vdash \hat{B}_1 \mathbin{\hat{\vee}} \hat{B}_2}\ \mathit{init}}{\Sigma : \mathcal{C}(\Xi), \hat{\Gamma}, \hat{B}_i \supset \hat{B}_1 \mathbin{\hat{\vee}} \hat{B}_2 \vdash \hat{B}_1 \mathbin{\hat{\vee}} \hat{B}_2}\ \supset L}{\Sigma, \Sigma' : \mathcal{C}(\Xi), \forall \Sigma'[B_i \supset (B_1 \mathbin{\hat{\vee}} B_2)], \hat{\Gamma}' \vdash \hat{B}_1 \mathbin{\hat{\vee}} \hat{B}_2}\ \forall L}{\Sigma, \Sigma' : \mathcal{C}(\Xi), \hat{\Gamma}' \vdash \hat{B}_1 \mathbin{\hat{\vee}} \hat{B}_2}\ cL$$

A similar conversion must also be done with the $\exists R$ inference rule. Thus, the original proof can be converted into an **I**-proof involving only $\mathcal{L}_0$ formulas. By Theorem 5.27, we know that the sequent $\Sigma : \mathcal{C}(\Xi), \hat{\Gamma} \vdash \hat{G}$ also has an $\Downarrow\mathcal{L}_0$-proof. Given that $\vee$ and $\exists$ cannot be top-level connectives of *fohh* program clauses, the left-hand context $\hat{\Gamma}$ will never get additional assumptions with target atoms containing $\hat{\vee}$ or $\hat{\exists}$ as their predicate symbol. This $\Downarrow\mathcal{L}_0$-proof can then be converted directly into an $\Downarrow\mathcal{L}_0$-proof of $\Sigma : \Gamma \vdash B_1 \vee B_2$ by noting that the only times a *decide* rule is used with a formula from $\mathcal{C}(\Xi)$ occurs when we are emulating either a $\vee R$ or $\exists R$ rule. The conversion of the proof is complete by replacing such *decide* rules and the phase above them with the right rule they are emulating. □

5.10 Examples of *fohc* logic programs

Figure 5.3 presents some examples of Horn clauses, along with two kinds of declarations. The syntax here is quite natural and follows the λProlog conventions. The `kind` declaration is used to declare members of the set of sorts S. In particular, the expression declares that `tok` is a token that is to be used as a primitive type. The expressions

```
type tok    <type expression>.
```

declares that the non-logical signature should contain the declaration of `tok` at the associated type expression. Logic program clauses are the remaining entries. In those entries, the infix symbol `:-` denotes the converse of $\supset$, a semicolon denotes a disjunction, a comma (which binds tighter than `:-` and the semicolon) denotes a conjunction of G-formulas while `&` denotes a conjunction of D-formulas. (In our current setting, both symbols denote the same logical connective $\wedge$. When we move to linear logic, these two conjunctions will be mapped to different linear logic connectives: see Section 6.5.) Tokens with

```
kind nat                 type.
type z                   nat.
type s                   nat -> nat.
type sum                 nat -> nat -> nat -> o.
type leq, greater        nat -> nat -> o.
```

```
sum z N N.
sum (s N) M (s P) :- sum N M P.
leq z N.
leq (s N) (s M)   :- leq N M.
greater N M       :- leq (s M) N.
```

Figure 5.3: *fohc* programs specifying relations over natural numbers.

initial capital letters are universally quantified with scope around an individual clause (which is terminated by a period).

In Figure 5.3, the symbol `nat` is declared to be a primitive type and `z` and `s` are used to construct natural numbers via zero and successor. The symbol `sum` is declared to be a relation of three natural numbers while the two symbols symbols `leq` and `greater` are declared to be binary relations on natural numbers. The following lines describe the meaning for these three predicates. For example, if the `sum` predicate holds for the triple M, N, and P then $N + M = P$: this relation is described recursively using the facts that $0+N = N$ and if $N+M = P$ then $(N+1)+M = (P+1)$. Similarly, relations describing $N \leq M$ and $N > M$ are also specified.

Similarly, Figure 5.4 introduces a primitive type for lists (of natural numbers) and two constructors for lists, namely, the empty list constructor `nil` and the non-empty list constructor, the infix symbol `::`. The binary predicate `sumup` relates a list of natural numbers with the sum of those numbers. The binary predicate `max` relates a list of numbers with the largest number in that list. The predicate `maxx` is an auxiliary predicate used to help compute the `max` relation.

Exercise 5.38. Informally describe the predicates specified by the clauses in Figures 5.5 and 5.6.

Exercise 5.39. Take a standard definition of Turing machine and show how to define an interpreter for a Turing machine in *fohc*. The specification should encode the fact that a given machine accepts a given word if and only if some atomic formula is provable.

```
kind nlist                 type.
type nil                   nlist.
type ::                    nat -> nlist -> nlist.
infixr ::                  5.
type sumup, max            nlist -> nat -> o.
type maxx                  nlist -> nat -> nat -> o.

sumup nil z.
sumup (N::L) S   :- sumup L T, sum N T S.

max L M          :- maxx L z M.
maxx nil A A.
maxx (X::L) A M :- leq X A,      maxx L A M.
maxx (X::L) A M :- greater X A, maxx L X M.
```

Figure 5.4: Some relations between natural numbers and lists

```
kind node                  type.
type a, b, c, d, e, f     node.
type adj, path             node -> node -> o.

adj a b & adj b c & adj c d & adj a c & adj e f.
path X X.
path X Z :- adj X Y, path Y Z.
```

Figure 5.5: Encoding a directed graph

```
type memb              nat  -> nlist -> o.
type append            nlist -> nlist -> nlist -> o.
type sort              nlist -> nlist -> o.
type split             nat -> nlist -> nlist -> nlist -> o.

memb X (X::L).
memb X (Y::L) :- memb X L.

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

split X nil nil nil.
split X (A::L) (A::S) B :- leq A X,     split X L S B.
split X (A::L) S (A::B) :- greater A X, split X L S B.
sort nil nil.
sort (X::L) S :- split X L Sm Big, sort Sm SmS,
                 sort Big BigS, append SmS (X::BigS) S.
```

Figure 5.6: More examples of Horn clause programs

5.11 Dynamics of proof search for *fohc*

Let Γ be a *fohc* program and G is an *fohc* goal, and let Ξ be a $\Downarrow\mathcal{L}_0$-proof of $\Sigma : \Gamma \vdash G$. Since there are no occurrences of $\multimap$R or $\forall$R in Ξ, every sequent occurring in Ξ has Σ as its signature and Γ as its left-hand side. Thus, if a program clause is ever needed (via the decide rule) during the search for a proof, it must be present at the beginning of that computation, along with all other clauses that might be needed during the computation. Thus, the logic of *fohc* does not directly support hierarchical programming in which certain program clauses are meant to be local within a particular scope. Similarly, all data structures built using first-order terms are built from a non-logical, fixed signature. Since signatures do not change during the search for proofs using first-order Horn clauses, all the constructors for data structures that need to be built during proof search must be available globally. In other words, *fohc* does not directly support hiding the internal details of data structures, an abstraction mechanism available in many programming languages via abstract data types.

If we only look at border sequents in $\Downarrow\mathcal{L}_0$-proofs in *fohc*, the only thing that changes when moving from border to border is the atomic right-hand sides. Given that we allow first-order terms (which can encode structures such as natural numbers, lists, trees, Turing machine tapes, etc.), it is easy to see that proof search in *fohc* has sufficient dynamics to encode general computation. Unfortunately, *all* of that dynamics takes place within *non-logical* contents, namely, within atomic formulas. As a result, logical techniques for analyzing computation via proof search have limited impact on what can be said directly about non-logical contexts. Thus, reasoning about properties of Horn clause programs will benefit little from logical and proof-theoretic analysis: most reasoning about Horn clause programs will almost always be based on viewing such programs as defining inductive structures. Chapter 10 provides an exception in which a static analysis of Horn clauses is given entirely relying on structural proof-theory instead of reducing Horn clause provability to inductive reasoning.

5.12 Examples of *fohh* logic programs

McCarthy [1989] described the problem of specifying the notion that a jar is *sterile* if every bacterium in it is dead. Consider proving that if a given jar `j` is heated, then that jar is sterile (given the fact that heating a jar kills all germs in that jar). Consider the *fohh* specification of this problem given in Figure 5.7. The expression `pi x\` denotes universal quantification of the variable `x` with a scope that extends as far to the right as consistent with parentheses or the end of the expression. The first of the clauses above can be

```
kind jar, bacterium      type.
type j                   jar.
type sterile, heated     jar -> o.
type dead                bacterium -> o.
type in                  bacterium -> jar -> o.

sterile X :- pi y\ in y X => dead y.
dead X    :- heated Y, in X Y.
heated j.
```

Figure 5.7: Heating a jar makes it sterile.

written as

$$\forall x(\forall y(\mathit{in}\ y\ x \supset \mathit{dead}\ y) \supset \mathit{sterile}\ x).$$

Note that no constructors for type `germ` are provided in Figure 5.7 and no explicit assumptions about the binary predicate `in` is given. The synthetic inference rule associate with this clause is

$$\frac{y : \mathit{bacterium}, \Sigma : \mathcal{P}, \mathit{in}\ y\ x \vdash \mathit{dead}\ y}{\Sigma : \mathcal{P} \vdash \mathit{sterile}\ x}\ .$$

Exercise 5.40. Construct the $\Downarrow\mathcal{L}_0$-proof of the goal formula *sterile* j from the logic program in Figure 5.7.

Another way to prove that a jar is sterile would be to use a microscope and search out every bacterium in the jar and confirm that they are dead. Unfortunately, this style of proof is not available in *fohh*. However, such proof strategies are possible in the stronger setting of model checking.

A specification for the binary predicate that relates a list with the reverse of that list can be given in *fohc* using the following program clauses.

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

Here, `reverse` is a binary relation on lists and the auxiliary predicate `rev` is a ternary relation on lists. By moving to *fohh*, it is possible to write the following specification instead.

```
reverse L K :- rv nil K => rv L nil.
rv (X::M) N :- rv M (X::N).
```

Here, the auxiliary predicate `rv` is also a binary predicate on lists. With this second specification, the use of non-logical context is slightly reduced in the sense that the atomic formula `(rev M K L)` in the first specification is encoded using the logical formula `(rv [] L => rv M K)` in the second specification. Note that the definition of reverse above has clausal order 2. It is possible to specify reverse with a clause of order 3 as follows.

```
reverse L K :-
  (pi X\ pi M\ pi N\ rv (X::M) N :- rv M (X::N)) =>
   rv nil K => rv L nil.
```

Here, not only the base case for `rv` is assumed in the body of `reverse` but also the recursive case. Given this encoding of reverse, no other program clauses can access either of these two clauses for `rv`.

Exercise 5.41. Reversing a pile of papers can informally be describing as: start by allocating an additional empty pile and then systematically move the top member of the original pile to the top of the newly allocated pile. When the original pile is empty, the other list is the reverse. Using the last specification of `reverse` above, show where, in the construction of a proof of the reverse relation, this informal computation takes place.

Note that *fohh* allows for a simple notion of modular logic programming. For example, let *classify*, *scanner*, and *misc* name (possibly large) collections of program clauses that have some specific role within a larger programming task: for example, *scanner* might contain code to convert a list of characters into a list of tokens prior to parsing, etc. Consider the following goal formula.

$$misc \supset ((classify \supset G_1) \wedge (scanner \supset G_2) \wedge G_3)$$

Attempting a proof of this goal will cause attempts of the three goals G_1, G_2, and G_3 with respect to different programs: *misc* and *classify* are used to prove G_1; *misc* and *scanner* are used to prove G_2; and *misc* is used to prove G_3. Thus, implicational goals can be used to structure the run-time environment of a program. For example, the code present in *classify* is not available during the proof attempt of G_2.

It is worth noting what it means to accumulating clauses from two different sources. For example, assume that the predicate `aux` is described by two different sets of clauses in *misc* and *scanner*, respectively. The description of `aux` in the accumulation of *misc* and *scanner* is given by mixing the clauses in these two separate sources. The resulting description of `aux` might not have a simple relationship to its descriptions in *misc* and *scanner* separately.

Classical logic does not support this discipline for the scoping of clauses. For example, the three goal formulas

$$D \supset (G_1 \vee G_2), \quad (D \supset G_1) \vee G_2, \quad \text{and} \quad G_1 \vee (D \supset G_2)$$

all provide different scopes for the clause D. However, in classical logic, the scoping of D is the same for all of these goals: given the classical equivalence

$B \supset C \equiv \neg B \vee C$, all three of these formulas are equivalent to $\neg D \vee G_1 \vee G_2$. In other words, classical logic allows for *scope extrusion*: while the scope of D in $(D \supset G_1) \vee G_2$ appears to be limited to G_1, that scope actually extrudes over the disjunction $G_1 \vee G_2$. Thus classical logic does not support the notion of scope that one usually wants from a module system.

5.13 Dynamics of proof search for *fohh*

Proof search using *fohh* programs and goals is a bit more dynamic than for *fohc*. In particular, both logic programs and signatures can grow. In this setting, every sequent in an $\Downarrow\mathcal{L}_0$-proof of the sequent $\Sigma : \Gamma \vdash G$ is either of the form

$$\Sigma, \Sigma' : \Gamma, \Gamma' \vdash G' \qquad \text{or} \qquad \Sigma, \Sigma' : \Gamma, \Gamma' \Downarrow D \vdash A.$$

Thus, the signature can grow by the addition of Σ' and the logic program can grown by the addition of Γ' (a *fohh* program over $\Sigma \cup \Sigma'$). More generally, it is the case that if the clausal order of Γ is $n \geq 1$ and the clausal order of G is at most $n - 1$, then the clausal order of Γ' is at most $n - 2$.

Since the terms used to instantiate quantifiers in the concluding sequent of the $\exists$R and $\forall$L inference rules range over the signature of that sequent, more terms are available for instantiation as proof search progresses. These additional terms include the eigenvariables of the proof that are introduced by $\forall$R inference rules. Note that once an eigenvariable is introduced, it is not instantiated by the proof search process. As a result, eigenvariables do not actually vary and, hence, act as locally scoped constants.

5.14 Limitations to *fohc* and *fohh* logic programs

Both *fohc* and *fohh* have certain limitations in how they can be used to represent computations. These limitations can be compared to the pumping lemmas for finite state machines and regular languages, which help to circumscribe the expressive power of those machines and languages. An immediate consequence of Proposition 5.21 is the following *monotoncity property* of intuitionistic provability: if $\Sigma : \Gamma \vdash_I G$ and if Γ' is a set of Σ-formulas containing Γ, then $\Sigma : \Gamma' \vdash_I G$. This proposition can be applied to solve the following two exercises.

Exercise 5.42.(‡) Consider the collection of declarations that accumulates the primitive types and non-logicals constants in Figure 5.3 along with declarations for a and $maxa$ which make them into predicates of one argument with sort *nat*. Show that there is no *fohh* logic program Γ that satisfies the following specification: For every nonempty set of natural numbers $N = \{n_1, \ldots, n_k\}$, let $\mathcal{A}$ be the set of atomic formulas $\{a\ n_1, \ldots, a\ n_k\}$. Then we require that Γ is such that $\mathcal{A}, \Gamma \vdash maxa\ m$ has an **I**-proof if and only if m is the maximum of the set N.

As was illustrated in Figure 5.4, the maximum of a set of numbers can be computed in *fohc* if that set of numbers is stored as a list within the non-logical context of an atomic formula and not in the logical context as require by the exercise above.

Exercise 5.43. (‡) Given the encoding of directed graphs as is illustrated in Figure 5.5, show that it is not possible to specify in *fohh* a predicate that is true of two nodes if and only if there is no path between them. Similarly, show that there is no specification in *fohh* of a predicate that holds of a node if and only if that node is not adjacent to another node.

As this exercise illustrates, it is possible to capture *reachability* within a graph but not, in general, *non-reachability*, at least when the adjacency graph is encoded as a set of atomic formulas as is the case in Figure 5.5.

There is a second class of weaknesses of *fohh* specifications that the following example illustrates. Consider the problem of specifying the removal of an element from a list. In particular, assume that we have the following signature Σ, written concretely as follows.

```
kind i                     type.
type a, b, c               i.
kind list                  type.
type nil                   list.
type ::                    i -> list -> list.
type remove                i -> list -> list -> o.
```

Here, `list` is the type of lists of elements of type `i` and that type `i` contains three elements. It is easy to show that it is impossible to find a specification, say $\mathcal{P}$ in *fohh* for the predicate `remove` such that

1. `(remove X L K)` is provable from Σ and $\mathcal{P}$ if and only if the list `K` is the result of removing *all* occurrences of `X` from `L`, and

2. the specification $\mathcal{P}$ does not contain occurrences of `a`, `b`, or `c`.

The last of these restrictions essentially says that `remove` should work no matter what terms of the type `i` exist. The proof of impossibility is immediate. If such a specification $\mathcal{P}$ existed, then $\mathcal{P}$ would must necessarily prove `(remove a [a,b,a] [b])`. Since `a` and `b` are not free in $\mathcal{P}$, then the universal quantification of such a goal is also provable: that is, $\mathcal{P}$ must also prove

```
  pi a\ pi b\ remove a (a::b::a::nil) (b::nil)).
```

But since that goal is provable, any instance of these quantifiers is also provable. Thus, `(remove a [a,a,a] [a])` is provable, which should not be the case.

This weakness results from the inability to specify the inequality of terms within the logic without explicitly referring to the constructor of terms. Suppose we allow the specification of `remove` to use the specific information about the structure of type `i`. In that case, it is possible to write the following specification of `remove`, which first specifies inequality on the three terms of type `i`.

```
type notequal     i -> i -> o.

notequal a b & notequal b a.
notequal a c & notequal a c.
notequal b c & notequal c b.

remove X nil nil.
remove X (X::L) K       :- remove X L K.
remove X (Y::L) (Y::K) :- notequal X Y, remove L K.
```

The following proposition is an immediate consequence of Exercise 4.11.

Proposition 5.44. *Let τ be a primitive type and let t be a Σ-term of type τ. If $x:\tau,\Sigma:\Gamma\vdash_I G$ then $\Sigma:\Gamma[t/x]\vdash_I G[t/x]$.*

Note that this proposition can be applied to non-logical constants of primitive types in the following sense. Consider a non-logical signature, Σ_0, that contains the declaration that $c:\tau$. Let Σ_0' be the result of removing $c:\tau$ from Σ. Then the sequent $\Sigma:\Gamma\vdash G$ is provable when the non-logical signature is Σ_0 if and only if the sequent $c:\tau,\Sigma:\Gamma\vdash G$ is provable when the non-logical signature is Σ_0', which (by the above proposition) implies that $\Sigma:\Gamma[t/c]\vdash G[t/c]$ holds for t a $\Sigma\cup\Sigma_0'$-term of type τ.

To illustrate applying Proposition 5.44, consider the type declarations in Figure 5.8: here i and j are primitive types. Note that terms of type i exist only in contexts where constants or variables of type j are declared. Figure 5.8 contains a specification of predicate *subSome* such that the goal $(subSome\ x\ s\ t\ r)$ is provable if and only if r is the result of substituting *some* occurrences of x (actually, of $(c\ x)$) in t with s.

Exercise 5.45. (‡) Prove that it is not possible in *fohh* to write a specification of *subAll* such that $(subAll\ x\ s\ t\ r)$ is provable if and only if r is the result of substituting *all* occurrences of x in t with s. Note that this specification would need to work in any extension of the non-logical signature (in particular, for extensions that contain constants of type j that do not occur in the specification of *subAll*).

```
type c                  j -> i.
type f                  i -> i.
type g                  i -> i -> i.
type subSome            j -> i -> i -> i -> o.

subSome X T (c X)   T.
subSome X T (c Y)   (c Y).
subSome X T (f U)   (f W)   :- subSome X T U W.
subSome X T (g U V) (g W Y) :- subSome X T U W,
                               subSome X T V Y.
```

Figure 5.8: Substitution of some occurrences.

Exercise 5.46. Write a *fohh* specification of *subOne* such that the goal

$$(subOne\ x\ s\ t\ r)$$

is provable if and only if r is the result of substituting *exactly one* occurrence of x in t with s. One might think that *subAll* can be specified using repeated calls to *subOne*. Given the previous exercise, this is not possible. Explain why.

5.15 Bibliographic notes

The early literature on logic programming did not use sequent calculus to encode proofs using Horn clauses: in fact, that literature used *refutations* instead of proof. For example, the papers by Emden and Kowalski [1976] and by Apt and Emden [1982] described logic programming using a restricted form of *resolution refutation* called *SLD-resolution*. The textbooks by Gallier [1986] and Lloyd [1987] provide more details about this approach to logic programming in classical logic.

A central design choice in our description of logic programming is the use of *goal-directed proof search* and the identification of the right-hand side of sequents with the goal and left-hand side of sequents with logic programs. This design choice goes back to 1986 [Miller and Nadathur, 1986; Miller, 1986]. A more general treatment of goal-directed proof search is given in the book by Gabbay and Olivetti [2000]. The book by Miller and Nadathur [2012] focuses on λProlog and presents several examples of logic programs written using first-order (and higher-order) hereditary Harrop formulas.

The focused proof system $\Downarrow\mathcal{L}_0$ takes the use of the $\Downarrow$ and the term "focus" from [Andreoli, 1992]. The first proofs of cut-elimination for focused proof system were done with linear logic: see Section 6.9 for such references. The proof theory of $\Downarrow\mathcal{L}_0$-proofs given in Section 5.5 uses techniques take from those references.

Harrop formulas were defined and shown to have the disjunction and existence properties in [Harrop, 1960].

Kripke models for intuitionistic logic were first introduced by Kripke in 1965, some years after he proposed such models for various modal logics in [Kripke, 1959]. The canonical Kripke model described in Section 5.6 is a simplified version of a model construction given in [Miller, 1992]. The Kripke lambda models built by Mitchell and Moggi [1991] are similar but more abstract and much more general than the model presented here.

Gentzen [1935] used the cut-elimination theorem for intuitionistic proof to help prove the decidability of propositional intuitionistic logic. His proof also required showing that contractions can be constrained in a certain way. The proof of decidability of intuitionistic provability over the connectives $\{\top, \wedge, \supset\}$ (Proposition 5.35) follows a similar outline since using focused proof systems greatly constrained the use of contraction.

One of the applications of hereditary Harrop formulas for logic programming is to help design modular programming abstractions for logic programming. Miller [1989b] proposed an early approach to modular programming in logic programming which later developed into the module system for λProlog [Kwon et al., 1993; Miller, 1994]. Numerous logic-based module designs for logic programming are surveyed in [Bugliesi et al., 1994].

The notion that synthetic inference rules (Section 5.8) can systematically be derived from formulas was an early project of Negri (see [Negri and von Plato, 2001]). A more general form of that early work is given in [Marin et al., 2022], where focused proof systems for both intuitionistic and classical logics are used to build various kinds of synthetic inference rules for those two logics.

As pointed out in Section 5.14, many important queries about graphs cannot be encoded using logic programs in *fohh*. The addition of *fixed points* to the logic and proof theory of this section has been proposed by Girard [1992] and Schroeder-Heister [1993]. That extension to logic permits capturing important forms of *negation-as-failure* as well as properties such as non-reachability and simulation [McDowell et al., 2003] as well as various other model checking problems [Heath and Miller, 2019].

As a result of Exercise 5.45, the implementation of substitution, typically needed when specifying theorem provers or operations that transform programs, must be *signature dependent*. That is, the constructors of certain types must be explicit in the specification. The notion of `copy`-clauses were proposed in [Miller, 1991; Miller and Nadathur, 2012] as a flexible and general avenue for making items in a signature available to a logic specification.

Chapter 6

Linear logic

The analysis of goal-directed proof search for classical and intuitionistic logics provided in Chapter 5 has at least the following three problems.

First, that analysis does not extend to all of classical logic nor intuitionistic logic. As we have seen, uniform provability, along with backchaining, provides an analysis of proof search for the $\mathcal{L}_0 = \{\boldsymbol{t}, \wedge, \supset, \forall\}$ fragment of intuitionistic logic, which is not a complete set of connectives for intuitionistic logic when quantification is restricted to be first-order.

Second, that analysis did not extend to multiple-conclusion sequents which is unfortunate since that setting allowed for a unified view of classical and intuitionistic proofs. Limiting proof search to single-conclusion sequents will limit our ability to use negation and De Morgan dualities to reason about logic programs.

Third, the proof search dynamics for our richest logic programming language so far, *fohh*, is rather weak: the left-hand side can only increase during proof search and, while the right-hand side can change, those changes occur essentially within atomic formulas (i.e., non-logical contexts). If sequents were able to change in more complex ways during proof search, logic programming could be more expressive and allow more direct uses of logic to reason about the computations specified.

As we shall see in this chapter, linear logic allows us to expand our analysis of proof search in such a way that we can address all three of these limitations.

6.1 Reflections on the structural inference rules

Before we present linear logic, we present several issues related to the role of contraction and weakening in **C**-proofs and **I**-proofs.

Controlling contractions improves proof search If the contraction rules are deleted from the classical and intuitionistic (unfocused) proof systems in Section 4.1, then the number of inference rules in a path in a proof can be bounded by the number of occurrences of logical connectives in the endsequent. Thus the search for cut-free proofs with such a modified proof system can be shown to be decidable. Using a more clever set of observations, Gentzen [1935] derived a decision procedure for propositional intuitionistic logic by seeing a way to limit the applications of contraction in that setting. The focused proof system $\Downarrow\mathcal{L}_0$ is a significant improvement over unfocused **I**-proofs in part because the structural rules are tightly regulated within $\Downarrow\mathcal{L}_0$ proofs: in particular, *wL* is built into the *init* rule and *cL* is built into the *decide* rule as well as the $\supset$L rule (in order to turn the usual multiplicative treatment of the left context into an additive treatment).

Invertible rules and contraction There is an interplay between structural rules and invertible introduction rules. Consider, for example, the following two introduction rules taken from the **C**-proof system (Section 4.1).

$$\frac{\Sigma : B, \Delta \vdash \Gamma \qquad \Sigma : C, \Delta \vdash \Gamma}{\Sigma : B \vee C, \Delta \vdash \Gamma}\ \vee\mathrm{L} \qquad\qquad \frac{\Sigma : B_i, \Delta \vdash \Gamma}{\Sigma : B_1 \wedge B_2, \Delta \vdash \Gamma}\ \wedge\mathrm{L}$$

The $\vee$L rule is *invertible*, meaning that if the conclusion is provable its two premises are provable. In this case, *cL* never needs to be applied to the formula $B \vee C$. On the other hand, the $\wedge$L rule is clearly not invertible and one might need to apply *cL* on this conjunction in order to access both conjunctions. For example, the proof of the formula $(p \wedge q) \supset (p \supset q \supset r) \supset r$ requires applying *cL* to $p \wedge q$. Since controlling contraction can help one design proof-search procedures, it is valuable to know that the applicability of contraction can be limited to those formula occurrences with non-invertible introduction rules.

Selecting between multiplicative and additive connectives If one of the introduction rules for a connective is multiplicative, we say that that connective is *multiplicative*. If one of the introduction rules for a connective is additive, we say that that connective is *additive*. In typical proof systems, such as our **I** and **C** proof systems (as well as Gentzen's *LJ* and *LK*), one must select an additive or a multiplicative version of each connectives: in the case of our proof system here, $\wedge$ and $\vee$ are additive while $\supset$ is multiplicative. In a fuller picture of proof theory, it seems unfortunate that we need to pick just one of these variants. While it is the case that the presence of weakening and contraction allows one to move interchangeably between the additive and multiplicative versions, we are considering proof systems where there are various restrictions on weakening and contraction. Thus, these different variants might be expected to behave differently within such proofs.

The collision of cut and the structural rules The interaction between cut and the structural rules can lead to undesirable dynamics in the usual way to perform cut-elimination. For example, consider the following instance of the cut rule.

$$\dfrac{\Delta \vdash C \qquad \Delta', C \vdash B}{\Delta, \Delta' \vdash B}\ \mathit{cut} \tag{$*$}$$

If the right premise is proved by a left-contraction rule from the sequent $\Delta', C, C \vdash B$, then cut-elimination proceeds by permuting the *cut* rule to the right premises, yielding the derivation

$$\dfrac{\dfrac{\Delta \vdash C \qquad \dfrac{\Delta \vdash C \qquad \Delta', C, C \vdash B}{\Delta, \Delta', C \vdash B}\ \mathit{cut}}{\Delta, \Delta, \Delta' \vdash B}\ \mathit{cut}}{\Delta, \Delta' \vdash B}\ cL.$$

In the intuitionistic variant of the sequent calculus, it is not possible for the occurrence of C in the left premise of $(*)$ to be contracted. If the cut inference in $(*)$ takes place in the classical proof system *LK*, it is possible that the left premise is the conclusion of a contraction applied to $\Delta \vdash C, C$. In that case, cut-elimination can also proceed by permuting the cut rule to the left premise.

$$\dfrac{\dfrac{\dfrac{\Delta \vdash C, C \quad \Delta', C \vdash B}{\Delta, \Delta' \vdash C, B}\ \mathit{cut} \qquad \Delta', C \vdash B}{\Delta, \Delta', \Delta' \vdash B, B}\ \mathit{cut}}{\Delta, \Delta' \vdash B}\ cL,\ cR$$

Thus, in *LK*, it is possible for both occurrences of C in $(*)$ to be contracted and, hence, the elimination of cut is nondeterministic since the cut rule can move to both the left and right premises. Such nondeterminism in cut-elimination is even more pronounced when we consider the collision of the cut rule with weakening in the following derivation.

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\Xi_1\\ \vdash B\end{matrix}}{\vdash C, B}\ wR \qquad \dfrac{\begin{matrix}\Xi_2\\ \vdash B\end{matrix}}{C \vdash B}\ wL}{\vdash B, B}\ \mathit{cut}}{\vdash B}\ cR$$

Cut-elimination here can yield either Ξ_1 or Ξ_2: thus, nondeterminism arising from weakening can lead to completely different proofs of B. This kind of example does not occur in the intuitionistic (single-sided) version of the sequent calculus.

Linear logic will make it possible to address these various issues, especially once we present focused proof systems for all of linear logic in Sections 6.7.

6.2 *LK* vs *LJ*: An origin story for linear logic

Gentzen restricted his *LJ* proof system for intuitionistic logic to be *LK* proofs in which there is at most one formula on the right. As we argued in Section 4.5, this restriction translates to the restriction that **I**-proofs are **C**-proofs in which the right-hand side of all sequents have exactly one formula. As we proved in Proposition 4.2, the following two restrictions guarantee that all sequents in a **C**-proof of the endsequent $\vdash B$ have exactly one formula in the right-hand context.

1. No structural rules are permitted on the right: i.e., proofs do not contain occurrences of *wR* and *cR*.

2. The two multiplicative rules, $\supset$L and *cut*, are restricted so that the formula on the right-hand side of the conclusion must also be the formula on the right-hand side of the rightmost premise.

To illustrate again this second restriction, recall the form of the $\supset$L rule.

$$\frac{\Sigma : \Delta_1 \vdash \Gamma_1, B \qquad \Sigma : C, \Delta_2 \vdash \Gamma_2}{\Sigma : B \supset C, \Delta_1, \Delta_2 \vdash \Gamma_1, \Gamma_2} \supset L$$

If the right-hand side of the conclusion contains one formula, that formula can move to the right-hand side of either the left or right premise. This extra condition, however, forces that formula to move only to the right premise and not to the left. Thus, the $\supset$L rule is doing two things: it introduces a connective *and* moves a side formula to a particular place. In this sense, implication within intuitionistic logic is different from all other logic connectives: the introduction rules of these other connectives are only involved in introducing a connective (in either an additive or multiplicative fashion). In Section 4.2, we noted that the *cut* rule can be emulated using the $\supset$L rule and a trivial implication: using this observation, the restriction on $\supset$L can explain the similar restriction on *cut*. In summary, the restriction on **I**-proofs can be used to say that (1) structural rules are only allowed on the left of the sequent and (2) implication seems to have more internal structure than is immediately apparent.

These two restrictions can be used to motivate a central and novel feature of linear logic. In particular, the fact that in intuitionistic proofs, some occurrences of formulas in a proof can be contracted while some cannot be contracted, will be captured in linear logic by the use of the two operators ! and ?. In particular, a formula of the form $!\,B$ on the left-hand side and a formula of the form $?\,B$ on the right-hand side can have weakening and contraction applied to them. In linear logic, these structural rules will not be applicable to any other occurrences of formulas. Thus, sequents in **C**-proofs can be encoded in linear logic using sequents of the form $!\,B_1, \ldots, !\,B_n \vdash ?\,C_1, \ldots, ?\,C_m$

$(n, m \geq 0)$ and sequents in **I**-proofs can be encoded in linear logic using sequents of the form $!\,B_1, \ldots, !\,B_n \vdash B_0$, where B_0 does not have ? as its top-level connective.

The ! operator can also be used to explain the behavior of the intuitionistic implication. Since the $\supset$R rule applied to the formula $B \supset C$ moves B to the left-hand side, it seems necessary to encode such an implication as, say, $(!\,B) \multimap C$, where $\multimap$ is the *linear implication.* Such an encoding ensures that ! is affixed to B as a new member of the left-hand side. This decomposition of the intuitionistic implication also explains the second restriction listed above. In particular, consider the following inference rule in which the conclusion is a single-conclusion sequent encoded as described above.

$$\frac{\Sigma : \Delta_1 \vdash \Gamma_1, !\,B \qquad \Sigma : C, \Delta_2 \vdash \Gamma_2}{\Sigma : (!\,B) \multimap C, \Delta_1, \Delta_2 \vdash \Gamma_1, \Gamma_2} \multimap\text{L}$$

As is described in more detail in Section 6.3.2, the right-introduction rule for ! when applied to the premise $\Delta_1 \vdash \Gamma_1, !\,B$ is only permitted if Δ_1 contains only !'ed formulas and Γ_1 contains only ?'ed formulas. Given our encoding, the right-hand side will have one formula that is not a top-level ?: thus, Γ_1 must be empty and Γ_2 must be that single formula. In this way, the second restriction on the structure of $\supset$L in **I**-proofs can be explained.

6.3 Sequent calculus proof systems for linear logic

The two-side proof system for linear logic is formed by putting together all of the inference rules in Figure 6.1, 6.2, 6.3, and 6.4. Before considering this full system, we first consider the following interesting subset of linear logic.

6.3.1 Multiplicative additive linear logic

Multiplicative additive linear logic or *MALL* for short is the subset of linear logic that results from collecting together the inference rules in Figure 6.1 and 6.2. MALL contains the additive and multiplicative versions of the classical disjunction, conjunction, and their units. Since MALL does not contain weakening or contraction, the additive and multiplicative versions of these connections are not inter-admissible within proofs (see Exercise 4.6). The eight logical connectives of MALL are listed in the following table by showing which is the additive or multiplicative variant of the associated classical connective.

Classical	Linear Additive	Linear Multiplicative
$\boldsymbol{t}$	$\top$	$\mathbf{1}$
$\boldsymbol{f}$	$\mathbf{0}$	$\bot$
$\wedge$	$\&$	$\otimes$
$\vee$	$\oplus$	$⅋$

$$\frac{\Sigma : \Gamma \vdash \Delta}{\Sigma : \Gamma, \mathbf{1} \vdash \Delta}\ \mathbf{1}L \qquad \frac{}{\Sigma : \cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{}{\Sigma : \Gamma \vdash \top, \Delta}\ \top R$$

$$\frac{}{\Sigma : \Gamma, \mathbf{0} \vdash \Delta}\ \mathbf{0}L \qquad \frac{}{\Sigma : \bot \vdash \cdot}\ \bot\mathrm{L} \qquad \frac{\Sigma : \Gamma \vdash \Delta}{\Sigma : \Gamma \vdash \bot, \Delta}\ \bot\mathrm{R}$$

$$\frac{\Sigma : \Gamma, B_i \vdash \Delta}{\Sigma : \Gamma, B_1 \,\&\, B_2 \vdash \Delta}\ \&\mathrm{L}\ (i = 1, 2) \qquad \frac{\Sigma : \Gamma \vdash B, \Delta \qquad \Sigma : \Gamma \vdash C, \Delta}{\Sigma : \Gamma \vdash B \,\&\, C, \Delta}\ \&\mathrm{R}$$

$$\frac{\Sigma : \Gamma, B \vdash \Delta \qquad \Sigma : \Gamma, C \vdash \Delta}{\Sigma : \Gamma, B \oplus C \vdash \Delta}\ \oplus\mathrm{L} \qquad \frac{\Sigma : \Gamma \vdash B_i, \Delta}{\Sigma : \Gamma \vdash B_1 \oplus B_2, \Delta}\ \oplus\mathrm{R}\ (i = 1, 2)$$

$$\frac{\Sigma : \Gamma, B_1, B_2 \vdash \Delta}{\Sigma : \Gamma, B_1 \otimes B_2 \vdash \Delta}\ \otimes\mathrm{L} \qquad \frac{\Sigma : \Gamma_1 \vdash B, \Delta_1 \qquad \Sigma : \Gamma_2 \vdash C, \Delta_2}{\Sigma : \Gamma_1, \Gamma_2 \vdash B \otimes C, \Delta_1, \Delta_2}\ \otimes\mathrm{R}$$

$$\frac{\Sigma : \Gamma_1, B \vdash \Delta_1 \qquad \Sigma : \Gamma_2, C \vdash \Delta_2}{\Sigma : \Gamma_1, \Gamma_2, B \parr C \vdash \Delta_1, \Delta_2}\ \parr\mathrm{L} \qquad \frac{\Sigma : \Gamma \vdash B, C, \Delta}{\Sigma : \Gamma \vdash B \parr C, \Delta}\ \parr\mathrm{R}$$

$$\frac{\Sigma : \Gamma \vdash B, \Delta}{\Sigma : \Gamma, B^\perp \vdash \Delta}\ (\cdot)^\perp L \qquad \frac{\Sigma : \Gamma, B \vdash \Delta}{\Sigma : \Gamma \vdash B^\perp, \Delta}\ (\cdot)^\perp R$$

Figure 6.1: The introduction rules for **L**

$$\frac{}{\Sigma : B \vdash B}\ \mathit{init} \qquad \frac{\Sigma : \Gamma \vdash B, \Delta \qquad \Sigma : \Gamma', B \vdash \Delta'}{\Sigma : \Gamma, \Gamma' \vdash \Delta, \Delta'}\ \mathit{cut}$$

Figure 6.2: The two identity rules for **L**

$$\frac{\Sigma : \Gamma, B[t/x] \vdash \Delta}{\Sigma : \Gamma, \forall x.B \vdash \Delta}\ \forall L \qquad \frac{y : \tau, \Sigma : \Gamma \vdash B[y/x], \Delta}{\Sigma : \Gamma \vdash \forall x_\tau.B, \Delta}\ \forall R$$

$$\frac{y : \tau, \Sigma : \Gamma, B[y/x] \vdash \Delta}{\Sigma : \Gamma, \exists x_\tau.B \vdash \Delta}\ \exists L \qquad \frac{\Sigma : \Gamma \vdash B[t/x], \Delta}{\Sigma : \Gamma \vdash \exists x.B, \Delta}\ \exists R$$

Figure 6.3: The introduction rules for quantifiers in **L**

$$\frac{\Sigma : \Gamma \vdash \Delta}{\Sigma : \Gamma, ! B \vdash \Delta}\ ! W \qquad \frac{\Sigma : \Gamma, ! B, ! B \vdash \Delta}{\Sigma : \Gamma, ! B \vdash \Delta}\ ! C \qquad \frac{\Sigma : \Gamma, B \vdash \Delta}{\Sigma : \Gamma, ! B \vdash \Delta}\ ! D$$

$$\frac{\Sigma : \Gamma \vdash \Delta}{\Sigma : \Gamma \vdash ?B, \Delta}\ ?W \qquad \frac{\Sigma : \Gamma \vdash ?B, ?B, \Delta}{\Sigma : \Gamma \vdash ?B, \Delta}\ ?C \qquad \frac{\Sigma : \Gamma \vdash B, \Delta}{\Sigma : \Gamma \vdash ?B, \Delta}\ ?D$$

$$\frac{\Sigma : !\Gamma, B \vdash ?\Delta}{\Sigma : !\Gamma, ?B \vdash ?\Delta}\ ?L \qquad \frac{\Sigma : !\Gamma \vdash B, ?\Delta}{\Sigma : !\Gamma \vdash !B, ?\Delta}\ !R$$

Figure 6.4: The rules for the exponentials in **L**

Here, $\mathbf{1}$ is the unit for $\otimes$, $\top$ is the unit for $\&$, $\bot$ is the unit for $⅋$, and $\mathbf{0}$ is the unit for $\oplus$. Our presentation of linear logic will also accept negation as a first-class connective, written as $(\cdot)^\perp$: the inference rules for negation in Figure 6.1 are the same as used by Gentzen (see Section 4.5). Keeping with the conventions described in Section 2.4, all binary logical connectives of linear logic have the type $o \to o \to o$, the units have the type o, and negation has the type $o \to o$.

Exercise 6.1. Let p, q, and r be propositional constants (constants of type o). Provide *MALL* proofs of the following sequents.

1. $\vdash p ⅋ p^\perp$
2. $(p \otimes q) \otimes r \vdash (r \otimes q) \otimes p$
3. $(p ⅋ q) ⅋ r \vdash (r ⅋ q) ⅋ p$
4. $p \otimes (q ⅋ r) \vdash (p \otimes q) ⅋ r$
5. $p \otimes (q ⅋ r) \vdash (p \otimes r) ⅋ q$
6. $r \vdash p ⅋ (p^\perp \otimes q) ⅋ (q^\perp \otimes r)$
7. $p^\perp \otimes q^\perp \vdash (p ⅋ q)^\perp$
8. $(p ⅋ q)^\perp \vdash p^\perp \otimes q^\perp$

Exercise 6.2. (‡) In the sequent $\vdash p \otimes q,\ p^\perp \otimes q,\ p \otimes q^\perp,\ p^\perp \otimes q^\perp$, every occurrence of the propositional constants p and q can be matched with an occurrence of its negation. Show, however, that this sequent is not provable in $\mathbf{L}$.

Although *MALL* is a propositional logic, it is an expressive and interesting logic on its own right. Deciding provability of *MALL* formulas is PSPACE-complete [Lincoln et al., 1992]. However, *MALL* is too weak to serve as the basis of a logic programming language since it is decidable and since it does not involve quantification, which is central to most views on logic programming. Adding the first-order quantifiers in Figure 6.3 to *MALL* does increase the expressiveness of the logic but the decidability of the resulting logic remains PSPACE-complete.

6.3.2 Linear logic as MALL plus exponentials

Full linear logic is the strengthening of MALL with the addition of the quantifiers $\forall$ and $\exists$ (whose inference rules in Figure 6.3 are essentially the same as the rules in classical and intuitionistic logics) and the addition of the two operators ! and ?, collectively called the *exponentials*. The exponentials reintroduce weakening and contraction into linear logic but only for formulas marked with these exponentials. In particular, there are four rules for each of these exponentials. Of those four, two permit weakening and contraction for the formulas they mark. The other two rules are essentially introduction rules. The *dereliction rules* $!D$ and $?D$ can be understood (reading rules from conclusion to premise) as saying that formulas that can be weakened and contracted can drop this privilege. The *promotion rules* $!R$ and $?L$ can similarly be read as saying that one way to show that a formula can gain the privilege of being weakened and contracted is to show that that formula can be proved in a context where every other formula has that privilege.

The proof system that arises from collecting together all the inference rules in Figures 6.1, 6.2, 6.3, and 6.4 is called the **L** proof system. Formulas that are built from the connectives explicitly mentioned in the **L** proof system are called ***L**-formulas*.

We extend the notion of logical equivalence $B \equiv C$ (see Section 4.3) to linear logic. In particular, two formulas B and C are *equivalent* in linear logic if the two sequents $B \vdash C$ and $C \vdash B$ are provable in **L**, which is the same as asserting that the sequent $\cdot \vdash (B \multimap C) \mathbin{\&} (C \multimap B)$ in provable in **L**.

Exercise 6.3. Show the following equivalences between the exponential, additive, and multiplicative connectives holds in linear logic. (These equivalences are inspired by the algebraic equation $x^{m+n} = x^m \times x^n$.)

$$!\top \equiv \mathbf{1} \qquad !(B \mathbin{\&} C) \equiv {!B} \otimes {!C} \qquad ?\mathbf{0} \equiv \bot \qquad ?(B \oplus C) \equiv {?B} \parr {?C}$$

Exercise 6.4. Let p be a propositional constant and let B be the formula $p \otimes !(p \multimap (p \otimes p)) \otimes !(p \multimap \mathbf{1})$. Show that the sequents $B \vdash B \otimes B$ and $B \vdash \mathbf{1}$ are provable in **L**.

Exercise 6.5. (‡) An *exponential prefix* is a finite sequence of zero or more occurrences of ! and ?. Let π be an exponential prefix. Prove that $\pi\pi B \equiv \pi B$ for all formulas B. Use that result to show that there are only seven exponential prefixes in linear logic up to equivalence: the empty prefix, !, ?, !?, ?!, !?!, and ?!?.

Exercise 6.6. Consider adding to linear logic a second tensor, say, $\hat{\otimes}$, that has the same inference rules as the original tensor. Prove that $B \otimes C$ is logically equivalent to $B \mathbin{\hat{\otimes}} C$. In this sense, the inference rules for tensor define it uniquely. Show that this is true for all logical connectives and quantifiers of linear logic except for the exponentials ! and ?.

6.3.3 Duality and polarity

The familiar De Morgan dualities of classical logic hold in a comprehensive fashion in linear logic. Not only do the binary connectives, units, and quantifiers have De Morgan duals, the exponentials do as well. We list here the De Morgan duals for all the logical connectives in linear logic.

connective	$\top$	$\&$	$\mathbf{1}$	$\otimes$	$\bot$	$⅋$	$\mathbf{0}$	$\oplus$	$!$	$?$	$\forall$	$\exists$
De Morgan dual	$\mathbf{0}$	$\oplus$	$\bot$	$⅋$	$\mathbf{1}$	$\otimes$	$\top$	$\&$	$?$	$!$	$\exists$	$\forall$

This table encodes several equivalences, of which we lists below a few.

$$(B ⅋ C)^\perp \equiv B^\perp \otimes C^\perp \qquad (B \,\&\, C)^\perp \equiv B^\perp \oplus C^\perp \qquad \top^\perp \equiv \mathbf{0}$$

$$(\exists x.B)^\perp \equiv \forall x.(B^\perp) \qquad (?\, B)^\perp \equiv\, !(B^\perp)$$

As a result of equivalences of this form, it is possible to rewrite every formula in linear logic into an equivalent formula in which negation has atomic scope. Such formulas are said to be in *negation normal form*. If we restrict our attention to only formulas in such normal forms, it is possible to give a one-sided sequent calculus proof system for linear logic, such as the one in Figure 6.5. By exploiting dualities, this proof system has about half the number of inference as the two-sided inference system for linear logic. Note that in Figure 6.5, the negation symbol that appears in *init* and *cut* is no longer a logical connective (since it has no introduction rules) but should be understood as the operator that negates its argument and then puts the result into negation normal form. We shall, however, make only limited use of this one-sided sequent system for linear logic. Instead, we shall continue to use two-sided sequents in what follows.

A important and exciting aspect of linear logic is the following. It is easy to confirm that in *MALL*, the right-introduction rule of a logical connective is invertible if and only if the left-introduction rule of that connective (or the right-introduction rule of its De Morgan dual) is not invertible. This observation leads to attributing a *polarity* to connectives. In particular, we say that a connective is *negative* if its right-introduction rule is invertible, and it is *positive* if its left-introduction rule is invertible. The negative connectives are $\bot$, $\top$, $⅋$, $\&$, and $\forall$. The positive connectives are $\mathbf{1}$, $\mathbf{0}$, $\otimes$, $\oplus$, and $\exists$.

Another perspective on the polarity of linear logic connectives is the following. If the right-introduction rule for a connective requires information from an oracle or its context, then that rule introduces a positive connective. For example, the $\oplus$R rule requires knowing which disjunct should be selected; the $\otimes$R rule needs to know how to split a context, the $\mathbf{1}R$ rule needs to know if its surrounding context is empty, and the $\exists R$ rule needs to be given a term. Dually, the right-introduction rules for negative connectives do not need any

$$\frac{}{\Sigma :\ \vdash \top, \Delta}\ \top R \qquad \frac{\Sigma :\ \vdash B, \Delta \qquad \Sigma :\ \vdash C, \Delta}{\Sigma :\ \vdash B \,\&\, C, \Delta}\ \& R$$

$$\frac{}{\Sigma :\ \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Sigma :\ \vdash B, \Delta_1 \qquad \Sigma :\ \vdash C, \Delta_2}{\Sigma :\ \vdash B \otimes C, \Delta_1, \Delta_2}\ \otimes\mathrm{R}$$

$$\frac{\Sigma :\ \vdash \Delta}{\Sigma :\ \vdash \bot, \Delta}\ \bot\mathrm{R} \qquad \frac{\Sigma :\ \vdash B, C, \Delta}{\Sigma :\ \vdash B \parr C, \Delta}\ \parr R$$

$$\frac{\Sigma :\ \vdash B_i, \Delta}{\Sigma :\ \vdash B_1 \oplus B_2, \Delta}\ \oplus\mathrm{R}\ (i = 1, 2)$$

$$\frac{y : \tau, \Sigma :\ \vdash B[y/x], \Delta}{\Sigma :\ \vdash \forall x_\tau . B, \Delta}\ \forall R \qquad \frac{\Sigma :\ \vdash B[t/x], \Delta}{\Sigma :\ \vdash \exists x . B, \Delta}\ \exists R$$

$$\frac{\Sigma :\ \vdash \Delta}{\Sigma :\ \vdash ?B, \Delta}\ ?W \qquad \frac{\Sigma :\ \vdash ?B, ?B, \Delta}{\Sigma :\ \vdash ?B, \Delta}\ ?C \qquad \frac{\Sigma :\ \vdash B, \Delta}{\Sigma :\ \vdash ?B, \Delta}\ ?D$$

$$\frac{\Sigma :\ \vdash B, ?\Delta}{\Sigma :\ \vdash !B, ?\Delta}\ !R$$

$$\frac{}{\Sigma :\ \vdash B, B^\perp}\ \mathit{init} \qquad \frac{\Sigma :\ \vdash B, \Delta \qquad \Sigma :\ \vdash B^\perp, \Delta'}{\Sigma :\ \vdash \Delta, \Delta'}\ \mathit{cut}$$

Figure 6.5: A one-sided sequent calculus proof system for linear logic

additional information for their successful application. (Note that the eigenvariable condition for the ∀R rule requires that the eigenvariable is not currently free in the sequent: however, it is a simple matter to organize things so that new names are always selected independently from the context.) In this latter sense, it is possible to then classify ! as a positive connective since its right rule (the promotion rule $!R$), requires the information from the context that all formulas in the context are marked appropriately with an exponential. As a result, we also consider ? (the De Morgan dual of !) as negative.

We say that the polarity of a non-atomic formula is negative or positive depending only on the polarity of its top-most connective. In order to extend the notion of polarity to all linear logic formulas, we adopt the convention that atoms have negative polarity.

Exercise 6.7. Let B be a linear logic formula. Prove that if the only occurrences of atomic formulas and negative connectives in B are in the scopes of occurrences of !, then $B \equiv\ !B$. Dually, prove that if the only occurrences of atomic formulas and positive connectives are in the scope of occurrences of ?, $B \equiv\ ?B$.

Exercise 6.8. Let B and C be two formulas for which $B \equiv\ !B$ and $C \equiv\ !C$.

Show that the following equivalences hold for the positive connectives.

$$\mathbf{1} \equiv\, !\mathbf{1} \qquad \mathbf{0} \equiv\, !\mathbf{0} \qquad B \otimes C \equiv\, !(B \otimes C) \qquad \exists x.B \equiv\, !\exists x.B \qquad B \oplus C \equiv\, !(B \oplus C)$$

Alternatively, let B and C be two formulas such that $B \equiv\, ?B$ and $C \equiv\, ?C$. Show that the following equivalences hold for the negative connectives.

$$\bot \equiv\, ?\bot \qquad \top \equiv\, ?\top \qquad B \parr C \equiv\, ?(B \parr C) \qquad B \,\&\, C \equiv\, ?(B \,\&\, C) \qquad \forall x.B \equiv\, ?\forall x.B$$

Exercise 6.9. Eventually, we will prove the cut-elimination theorem for the **L** proof system for linear logic. A simple consequence of that cut-elimination theorem is the proof that some introduction rules in **L** are invertible. For example, assume that the linear logic sequent $\Sigma : \Delta \vdash \Gamma, B \parr C$ has a proof, say Ξ. We want to prove that it has a cut-free proof in which the last inference rule is an introduction rule for this occurrence of $B \parr C$. This is proved by considering the result of eliminating cut from the following:

$$\dfrac{\dfrac{\begin{array}{c}\Xi \\ \Sigma : \Delta \vdash \Gamma, B \parr C\end{array} \qquad \begin{array}{c}\Xi' \\ \Sigma : B \parr C \vdash B, C\end{array}}{\Sigma : \Delta \vdash \Gamma, B, C}\ cut}{\Sigma : \Delta \vdash \Gamma, B \parr C}\ \parr R.$$

Here, Ξ' is the obvious proof of $\Sigma : B \parr C \vdash B, C$. Using an argument of this style, prove the invertibility of $\& R$, $\forall R$, $\otimes$L, $\oplus$L, and $\exists L$.

Exercise 6.10. Prove that if $\Sigma : \top \vdash B$ is provable in **L** then, for every multiset of Σ-formulas Δ, the sequent $\Sigma : \Delta \vdash B$ in provable in **L**.

Exercise 6.11. The following three entailments hold in classical logic.

Mix:	$A \wedge B$	$\vdash A \vee B$
Switch:	$(A \vee B) \wedge C$	$\vdash A \vee (B \wedge C)$
Medial:	$(A \wedge C) \vee (B \wedge D)$	$\vdash (A \vee B) \wedge (C \vee D)$

(The names for these entailments are taken from [Guglielmi, 2007].) Consider mapping the pair of classical logic connectives $\langle \wedge, \vee \rangle$ into one of the following four pairs of linear logic connectives

$$\langle \otimes, \parr \rangle, \quad \langle \otimes, \oplus \rangle, \quad \langle \&, \parr \rangle, \quad \langle \&, \oplus \rangle.$$

For each of the above three classical logic entailments, find which of these mappings of connectives yields an entailment provable in linear logic. For example, applying the first of these mappings to the Mix entailment yields $A \otimes B \vdash A \parr B$, which is not generally provable in linear logic.

Exercise 6.12. We define a new attribute, called *junctiveness*, of a MALL connectives as follows. The junctiveness of the connectives $\top$, $\&$, $\mathbf{1}$, or $\otimes$ is *conjunctive* while the junctiveness of the connectives $\bot$, $⅋$, $\mathbf{0}$, $\oplus$ is *disjunctive*. Thus, each connective has four attributes, namely, arity (0 for unit or 2 for binary connective), additive/multiplicative, polarity (positive/negative), and junctiveness (conjunctive/disjunctive). Show that if we fix the arity, then, given any two of the remaining three attributes, the third can be determined uniquely. For example, there is a unique binary connective that is conjunctive and positive (the multiplicative $\otimes$) and a unique unit that is disjunctive and additive (the positive $\mathbf{0}$). Show also that the De Morgan dual of a connective (see the beginning of Section 6.3.3) flips the junctiveness and polarity while leaving unchanged the other two attributes.

6.3.4 Introducing implications

Since implication plays a large role in the design of the logic programming languages we have seen in earlier chapters, we add implication as a logical connective into linear logic. In fact, there are two implications, namely the *linear implication* $\multimap$ and the *intuitionistic implication* $\Rightarrow$. The linear implication $B \multimap C$ can be defined as $B^\perp ⅋ C$ and the intuitionistic implication $B \Rightarrow C$ can be defined as $(!\,B) \multimap C$. Since both of these implications are based on the multiplicative disjunction $⅋$, these connectives are considered multiplicative and they have negative polarity.

The left and right-introduction rules for $\multimap$ are the following.

$$\frac{\Sigma : \Gamma_1 \vdash B, \Delta_1 \qquad \Sigma : \Gamma_2, C \vdash \Delta_2}{\Sigma : \Gamma_1, \Gamma_2, B \multimap C \vdash \Delta_1, \Delta_2} \multimap L \qquad \frac{\Sigma : \Gamma, B \vdash C, \Delta}{\Sigma : \Gamma \vdash B \multimap C, \Delta} \multimap\mathrm{R}$$

Exercise 6.13. Prove the following *curry/uncurry equivalences*.

$$\mathbf{1} \multimap H \equiv H \qquad (B \otimes C) \multimap H \equiv B \multimap C \multimap H$$

$$\mathbf{0} \multimap H \equiv \top \qquad (B \oplus C) \multimap H \equiv (B \multimap H) \,\&\, (C \multimap H)$$

$$(\exists x.B\ x) \multimap H \equiv \forall x.(B\ x \multimap H)$$

Many presentations of linear logic make little or no use of implications since they often focus on the rich symmetries allowed by the negation of linear logic. In particular, every logical connective of linear logic, except for the implications $\multimap$ and $\Rightarrow$, have other logical connectives that are their De Morgan duals. Another, more serious, problems with the intuitionistic implication is the nature of its left and right-introduction rules. For example, it is tempting to write the following candidate introduction rules for $\Rightarrow$.

$$\frac{\Delta, C \vdash\, !\,B, \Gamma \qquad \Delta, C \vdash \Gamma}{\Delta, B \Rightarrow C \vdash \Gamma} \qquad \frac{\Delta, !\,B \vdash C, \Gamma}{\Delta \vdash B \Rightarrow C, \Gamma}$$

These rules, however, break the usual pattern for introduction rules in sequent calculus: exactly one occurrence of a logical connective appears in the conclusion while no new occurrences of a logical connective appears in a premise. In both of these rules, the occurrence of ! in the premise violates this pattern. This pattern has already been violated, in principle, by the rules for the exponentials. In particular, the contraction rule $!\,C$ inserts two occurrences of ! into a premise while $!\,R$ requires possibly many occurrences of ! and ? to be present in the conclusion. We address these issues around the implications and the exponentials by introducing a new style of sequent calculus proof system in the next section.

6.4 Single conclusion sequents with two zones

One of our hopes with introducing linear logic is to provide a means to enrich the logic programming languages described in Chapter 5. Thus we will analyze goal-directed proofs, backchaining, and focused proof systems within linear logic. This analysis will lead to showing that *all* of linear logic can be presented as an abstract logic programming language. Before showing that result, we show how to relate proofs in linear logic with **I**-proofs and **C**-proofs.

If linear logic does serve as a more refined and low-level setting for both classical and intuitionistic logic, then we might expect that simply replacing the logical connectives in $\Downarrow \mathcal{L}_0$, namely $\{\boldsymbol{t}, \wedge, \supset, \forall\}$ (see Section 5.5), with the corresponding linear logic connectives $\{\top, \&, \Rightarrow, \forall\}$ should allow us to reproduce intuitionistic proofs within linear logic. If that is indeed the case, then adding $\multimap$ to this last set of connectives might well provide us with an extension to *fohh*. We will soon show to what extent that expectation is true.

Let $\mathcal{L}_1$ be the set of logical connectives $\{\top, \&, \multimap, \Rightarrow, \forall\}$. An $\mathcal{L}_1$*-formula* is any first-order formula all of whose logical connectives come from $\mathcal{L}_1$. Figure 6.6 presents an (unfocused) proof system **P** for the formulas taken from $\mathcal{L}_1$. In order to deal with the problem of specifying an introduction rule for $\Rightarrow$ mentioned at the end of the previous section, the **P** proof system features one new innovation: the left-hand context in sequents is divided into two *zones*. In particular, this proof system uses sequents of the form $\Sigma : \Delta; \Gamma \vdash B$. Here, both Δ and Γ are multisets of $\mathcal{L}_1$ formulas, and B is an $\mathcal{L}_1$ formula. We say that Δ is the *unbounded context* context while Γ is the *bounded context* of this sequent. The informal reading of the sequent $B_1, \ldots, B_n; C_1, \ldots, C_m \vdash E$ is given by the linear logic sequent

$$!\,B_1, \ldots, !\,B_n, C_1, \ldots, C_m \vdash E.$$

The $\&R$ rule is additive, meaning that the bounded and unbounded contexts are the same in the conclusion and in the sequents in the premises.

$$\dfrac{}{\Sigma : \Delta; A \vdash A}\ \mathit{init} \qquad \dfrac{\Sigma : \Delta, B; \Gamma, B \vdash C}{\Sigma : \Delta, B; \Gamma \vdash C}\ \mathit{absorb} \qquad \dfrac{}{\Sigma : \Delta; \Gamma \vdash \top}\ \top R$$

$$\dfrac{\Sigma : \Delta; \Gamma, B_i \vdash C}{\Sigma : \Delta; \Gamma, B_1 \,\&\, B_2 \vdash C}\ \&\mathrm{L} \qquad \dfrac{\Sigma : \Delta; \Gamma \vdash B \qquad \Sigma : \Delta; \Gamma \vdash C}{\Sigma : \Delta; \Gamma \vdash B \,\&\, C}\ \&\mathrm{R}$$

$$\dfrac{\Sigma : \Delta; \Gamma_1 \vdash B \qquad \Sigma : \Delta; \Gamma_2, C \vdash E}{\Sigma : \Delta; \Gamma_1, \Gamma_2, B \multimap C \vdash E}\ \multimap L \qquad \dfrac{\Sigma : \Delta; \Gamma, B \vdash C}{\Sigma : \Delta; \Gamma \vdash B \multimap C}\ \multimap\mathrm{R}$$

$$\dfrac{\Sigma : \Delta; \cdot \vdash B \qquad \Sigma : \Delta; \Gamma, C \vdash E}{\Sigma : \Delta; \Gamma, B \Rightarrow C \vdash E}\ \Rightarrow\mathrm{L} \qquad \dfrac{\Sigma : \Delta, B; \Gamma \vdash C}{\Sigma : \Delta; \Gamma \vdash B \Rightarrow C}\ \Rightarrow R$$

$$\dfrac{\Sigma : \Delta; \Gamma, B[t/x] \vdash C}{\Sigma : \Delta; \Gamma, \forall x.B \vdash C}\ \forall L \qquad \dfrac{y : \tau, \Sigma : \Delta; \Gamma \vdash B[y/x]}{\Sigma : \Delta; \Gamma \vdash \forall x_\tau.B}\ \forall R$$

$$\dfrac{\Sigma : \Delta; \Gamma_1 \vdash B \quad \Sigma : \Delta; \Gamma_2, B \vdash C}{\Sigma : \Delta; \Gamma_1, \Gamma_2 \vdash C}\ \mathit{cut} \qquad \dfrac{\Sigma : \Delta; \cdot \vdash B \quad \Sigma : \Delta, B; \Gamma \vdash C}{\Sigma : \Delta; \Gamma \vdash C}\ \mathit{cut}\,!$$

Figure 6.6: The single-conclusion, two-zone proof system $\mathbf{P}$ for $\mathcal{L}_1$.

However, the other rules with two premises treat their unbounded contexts additively while treating their bounded contexts multiplicatively: i.e., every formula occurrence in the bounded context of the conclusion occurs in the bounded context of exactly one premise. This hybrid behavior for the multiplicative inference rules is possible because contraction is available for the unbounded contexts. For example, as the following derivation illustrates, the multiplicative $\multimap L$ rule plus contraction (!L) can be used to justify the hybrid rule.

$$\begin{array}{c} \Delta; \Gamma_1 \vdash B \qquad \Delta; \Gamma_2 \vdash C \\ \hline \Delta, \Delta; \Gamma_1, \Gamma_2, B \multimap C \vdash E \\ \hline\hline \Delta; \Gamma_1, \Gamma_2, B \multimap C \vdash E \end{array}\ !C$$

There are two inference rules in Figure 6.6, namely $\Rightarrow$L and *cut*!, that require the bounded part of one of its premises to be empty. When that context is empty, as in $B_1, \ldots, B_n; \cdot \vdash E$, the corresponding linear logic sequent is $!\,B_1, \ldots, !\,B_n \vdash E$. When that sequent is provable in linear logic, then $!\,B_1, \ldots, !\,B_n \vdash !\,E$ is also provable (using the $!\,R$ rule in Figure 6.4). Thus, requiring a premise to have an empty bounded context can also guarantee that a (hidden) ! formula is proved from the unbounded context.

The following function translates formulas that may involve implications into formulas where those implications are replaced by their definitions. Let $B^\diamond$ be the result of repeatedly replacing within B all occurrences of $C_1 \Rightarrow C_2$ with $(!\,C_1)^\perp \parr C_2$ and all occurrences of $C_1 \multimap C_2$ with $C_1{}^\perp \parr C_2$. We also allow $^\diamond$ to be applied to a multiset of formulas which results in the multiset of $^\diamond$ applied to each member.

The following proposition relates the connection between the **P** and **L** proof systems.

Proposition 6.14. *Let B be a formula, Δ and Γ be multisets of formulas for linear logic with possible occurrences of $\multimap$ and $\Rightarrow$. The sequent $\Delta; \Gamma \vdash B$ has a* **P***-proof if and only if the sequent $!(\Delta^\diamond), \Gamma^\diamond \vdash B^\diamond$ has a linear logic proof.*

Proving the forward direction is a straightforward induction on the structure of proofs. Proving the converse is slightly more challenging but it can be more easily proved using the completeness of a focused proof system for linear logic given in Section 6.8. We shall not provide a proof of this proposition since we will consider a more general proof system in Section 6.7 and prove various properties of that proof system in Section 6.8. The proof of this proposition will follow immediately from those more general results.

Exercise 6.15. Let $\Delta; \Gamma \vdash B$ be an **P**-sequent in which there is no occurrence of $\multimap$. Assume also that Ξ is **P** proof of that sequent that does not have occurrences of the *cut* rule but may have occurrences of *cut*! rule. Then Γ is either empty or a singleton.

Although several properties of the **P** proof system could be stated and proved, this unfocused proof system is not the best for our needs to study generalizations of goal-directed search and backchaining. We now motivate a new, focused version of the **P** proof system.

As we did in Section 5.4, we organize the left-hand rules using the backchaining discipline. As we have done before, we illustrate this by presenting two different proof systems: the first using a focused formula using the $\Downarrow$ to denote the focus of the backchain rule, and a second proof system where backchaining is described as a single inference rule BC.

Figure 6.7 contains a proof system in which the application of the left-introduction rules is on a designated formula from the left (compare these rules to those in Figure 5.1). The new sequent, written as $\Sigma : \mathcal{P}; \Gamma \Downarrow D \vdash A$, is used to display that designated formula between the $\Downarrow$ and the $\vdash$. That displayed formula is the only one on which left-introduction rules may be applied. The two *decide* rules are used to turn the attempt to prove an atomic formula into an attempt to use a focused formula. The sequent $\Sigma : \mathcal{P}; \Gamma \vdash G$ or the sequent $\Sigma : \mathcal{P}; \Delta \Downarrow D \vdash A$ has a $\Downarrow\mathcal{L}_1$-proof if it has a proof using the rules in Figure 6.7.

Note that the rule for $\multimap L$ requires splitting the bounded context Γ_1, Γ_2 into two parts (when reading the rule bottom up). There are, of course, 2^n such splittings if that context has $n \geq 0$ distinct formulas.

The soundness and completeness of the $\Downarrow\mathcal{L}_1$ proof system for sequents using formulas only from $\mathcal{L}_1$ will following from a stronger result that we shall prove in some detail in Section 6.8.

$$\frac{}{\Sigma : \mathcal{P}; \Gamma \vdash \top}\ \top R \qquad \frac{\Sigma : \mathcal{P}; \Gamma \vdash B \qquad \Sigma : \mathcal{P}; \Gamma \vdash C}{\Sigma : \mathcal{P}; \Gamma \vdash B \,\&\, C}\ \&\mathrm{R}$$

$$\frac{\Sigma : \mathcal{P}; \Gamma, B \vdash C}{\Sigma : \mathcal{P}; \Gamma \vdash B \multimap C}\ \multimap\mathrm{R} \qquad \frac{\Sigma : \mathcal{P}, B; \Gamma \vdash C}{\Sigma : \mathcal{P}; \Gamma \vdash B \Rightarrow C}\ \Rightarrow R$$

$$\frac{y : \tau, \Sigma : \mathcal{P}; \Gamma \vdash B[y/x]}{\Sigma : \mathcal{P}; \Gamma \vdash \forall x_\tau . B}\ \forall R$$

$$\frac{\Sigma : \mathcal{P}, D; \Gamma \Downarrow D \vdash A}{\Sigma : \mathcal{P}, D; \Gamma \vdash A}\ \mathit{decide}\,! \qquad \frac{\Sigma : \mathcal{P}; \Gamma \Downarrow D \vdash A}{\Sigma : \mathcal{P}; \Gamma, D \vdash A}\ \mathit{decide}$$

$$\frac{}{\Sigma : \mathcal{P}; \cdot \Downarrow A \vdash A}\ \mathit{init} \qquad \frac{\Sigma \Vdash t : \tau \qquad \Sigma : \mathcal{P}; \Gamma \Downarrow D[t/x] \vdash A}{\Sigma : \mathcal{P}; \Gamma \Downarrow \forall_\tau x . D \vdash A}\ \forall\mathrm{L}$$

$$\frac{\Sigma : \mathcal{P}; \Gamma \Downarrow D_i \vdash A}{\Sigma : \mathcal{P}; \Gamma \Downarrow D_1 \,\&\, D_2 \vdash A}\ \&\mathrm{L}\ (i \in \{1, 2\})$$

$$\frac{\Sigma : \mathcal{P}; \Gamma_1 \vdash G \quad \Sigma : \mathcal{P}; \Gamma_2 \Downarrow D \vdash A}{\Sigma : \mathcal{P}; \Gamma_1, \Gamma_2 \Downarrow G \multimap D \vdash A}\ \multimap L$$

$$\frac{\Sigma : \mathcal{P}; \cdot \vdash G \quad \Sigma : \mathcal{P}; \Gamma \Downarrow D \vdash A}{\Sigma : \mathcal{P}; \Gamma \Downarrow G \Rightarrow D \vdash A}\ \Rightarrow\mathrm{L}$$

Figure 6.7: The focused proof system $\Downarrow\mathcal{L}_1$. In the $\forall$L rule, t is a Σ-term of type τ.

For a second (less proof-theoretic) description of backchaining, consider the following definition. Let the syntactic variable B range over $\mathcal{L}_1$-formulas. Then $\|B\|_\Sigma$ is the smallest set of triples of the form $\langle \mathcal{P}, \Gamma, B' \rangle$, where $\mathcal{P}$ and Γ are multisets of formulas, such that

1. $\langle \emptyset, \emptyset, B \rangle \in \|B\|_\Sigma$;

2. if $\langle \mathcal{P}, \Gamma, B_1 \& B_2 \rangle \in \|B\|_\Sigma$ then $\langle \mathcal{P}, \Gamma, B_1 \rangle \in \|B\|_\Sigma$ and $\langle \mathcal{P}, \Gamma, B_2 \rangle \in \|B\|_\Sigma$;

3. if $\langle \mathcal{P}, \Gamma, B_1 \Rightarrow B_2 \rangle \in \|B\|_\Sigma$ then $\langle \mathcal{P} \cup \{B_1\}, \Gamma, B_2 \rangle \in \|B\|_\Sigma$;

4. if $\langle \mathcal{P}, \Gamma, B_1 \multimap B_2 \rangle \in \|B\|_\Sigma$ then $\langle \mathcal{P}, \Gamma \uplus \{B_1\}, B_2 \rangle \in \|B\|_\Sigma$; and

5. if $\langle \mathcal{P}, \Gamma, \forall x_\tau . B' \rangle \in \|B\|_\Sigma$ and t is a Σ-term of type τ, then

$$\langle \mathcal{P}, \Gamma, B'[t/x] \rangle \in \|B\|_\Sigma.$$

Let $\Downarrow\mathcal{L}_1'$ be the proof system that results from replacing *init* and the four left-introduction rules in Figure 6.7 with the *backchaining* inference rule in Figure 6.8.

$$\frac{\Sigma:\mathcal{P};\cdot\vdash B_1\ \ldots\ \Sigma:\mathcal{P};\cdot\vdash B_n\quad \Sigma:\mathcal{P};\Gamma_1\vdash C_1\ \ldots\ \Sigma:\mathcal{P};\Gamma_m\vdash C_m}{\Sigma:\mathcal{P};\Gamma_1,\ldots,\Gamma_m,B\vdash A}\ \mathrm{BC}$$

provided $n, m \geq 0$, $\langle\{B_1,\ldots,B_n\},\{C_1,\ldots,C_m\},A\rangle \in \|B\|_\Sigma$, and A is atomic.

Figure 6.8: Backchaining for the linear logic fragment $\mathcal{L}_1$.

Proposition 6.16. *Let B be a formula and let Δ and Γ be multisets of formulas, all over the logical constants $\top, \&, \multimap, \Rightarrow$, and $\forall$. The sequent $\Sigma:\Delta;\Gamma\vdash B$ has a proof in $\Downarrow\mathcal{L}_1$ if and only if it has a proof in $\Downarrow\mathcal{L}_1'$.*

This proposition follows directly from the completeness of the $\Downarrow\mathcal{L}_1$ proof system, following the same lines used to prove the analogous results in Section 5.7.

It is now clear from the $\Downarrow\mathcal{L}_1$-proof system that the dynamics of proof search in this setting has improved beyond that described for *fohh* (Section 5.13). In particular, every sequent in a $\Downarrow\mathcal{L}_1$ proof of the sequent $\Sigma:\mathcal{P};\Gamma\vdash G$ is either of the form

$$\Sigma,\Sigma':\mathcal{P},\mathcal{P}';\Gamma'\vdash G' \qquad \text{or} \qquad \Sigma,\Sigma':\mathcal{P},\mathcal{P}';\Gamma'\Downarrow D\vdash A.$$

Just as with *fohh*, the signature can grown by the addition of Σ' and the unbounded context can grown by the addition of $\mathcal{P}'$. The bounded context, Γ', however, can change in much more general and arbitrary ways. Formulas in the bounded context that were present at the root of a proof may not necessarily be present later (higher) in the proof. As we shall see later, we can use formulas in the bounded context to represent, say, the state of a computation or a switch that is off but later on.

Exercise 6.17. Consider the set $\mathcal{L}_1\cup\{\bot\}$ of linear logic connectives. Show that this set of connectives is complete in the sense that all other logical connectives can be written in terms of these. In particular, describe how to encode

$$B^\perp \quad 0 \quad 1 \quad !B \quad B\oplus C \quad B\otimes C \quad \exists x.B \quad ?B \quad B \parr C$$

using only the connectives in $\mathcal{L}_1\cup\{\bot\}$. Use the **P** proof system to present the required proofs. Can you argue why it is the case that if $\mathcal{L}'$ is a proper subset of $\mathcal{L}_1$ then $\mathcal{L}'\cup\{\bot\}$ does not yield a complete set of connectives for linear logic.

6.5 Embedding *fohh* into linear logic

The abstract logic programming language $\langle \mathcal{L}_1, \mathcal{L}_1, \vdash_{\mathcal{L}} \rangle$ has been also called Lolli (after the lollipop shape of the $\multimap$). As a programming language, Lolli appears to be $\mathcal{L}_0$ with $\multimap$ added. To make this connection more precise, we should show how $\mathcal{L}_0$ can be embedded into Lolli (since, technically, they use different sets of connectives). Girard has presented a mapping of intuitionistic logic into linear logic that preserves not only provability but also proofs [Girard, 1987]. On the fragment of intuitionistic logic containing $\boldsymbol{t}$, $\wedge$, $\supset$, and $\forall$, his translation is given by:

$$\begin{gathered}
(A)^0 = A, \text{ where } A \text{ is atomic,} \\
(\boldsymbol{t})^0 = \top, \\
(B_1 \wedge B_2)^0 = (B_1)^0 \mathbin{\&} (B_2)^0, \\
(B_1 \supset B_2)^0 = (B_1)^0 \Rightarrow (B_2)^0, \\
(\forall x.B)^0 = \forall x.(B)^0.
\end{gathered}$$

However, if we are willing to focus attention on only cut-free proofs in intuitionistic logic and in linear logic, it is possible to define a "tighter" translation. Consider the following two translation functions.

$$\begin{gathered}
(A)^+ = (A)^- = A, \text{ where } A \text{ is atomic} \\
(\boldsymbol{t})^+ = \mathbf{1} \qquad (\boldsymbol{t})^- = \top \\
(B_1 \wedge B_2)^+ = (B_1)^+ \otimes (B_2)^+ \\
(B_1 \wedge B_2)^- = (B_1)^- \mathbin{\&} (B_2)^- \\
(B_1 \supset B_2)^+ = (B_1)^- \Rightarrow (B_2)^+ \\
(B_1 \supset B_2)^- = (B_1)^+ \multimap (B_2)^- \\
(\forall x.B)^+ = \forall x.(B)^+ \\
(\forall x.B)^- = \forall x.(B)^-
\end{gathered}$$

If we allow positive occurrences of $\vee$ and $\exists$ within cut-free proofs, as in proofs involving the hereditary Harrop formulas, we would also need the following two clauses.

$$\begin{gathered}
(B_1 \vee B_2)^+ = (B_1)^+ \oplus (B_2)^+ \\
(\exists x.B)^+ = \exists x.(B)^+
\end{gathered}$$

Proposition 6.18. *Let Σ be a signature, B be a Σ-formula and Δ a set of Σ-formulas, all over the logical constants $\boldsymbol{t}, \wedge, \supset$, and $\forall$. Define Δ^- to be the multiset $\{C^- \mid C \in \Delta\}$. Then, the sequent $\Sigma : \Delta \vdash B$ has an **I**-proof if and only if the sequent $\Sigma : \Delta^-; \cdot \vdash B^+$ has a cut-free proof in $\Downarrow\mathcal{L}_1$.*

This proposition is a consequence of the more general Proposition 6.45. In fact, if one considers $\Downarrow\mathcal{L}_0$-proofs instead of **I**-proofs, then $\Downarrow\mathcal{L}_0$-proofs of $\Sigma : \Delta \vdash B$ are essentially $\Downarrow\mathcal{L}_1$-proofs of $\Sigma : \Delta^-; \cdot \vdash B^+$. This suggests how to design the concrete syntax of a linear logic programming language so that the interpretation of Prolog and λProlog programs remains unchanged when embedded into this new setting. In particular, the Prolog syntax

$$A_0\ :-\ A_1, \ldots, A_n$$

is traditionally intended to denote (the universal closure of) the formula

$$(A_1 \wedge \ldots \wedge A_n) \supset A_0.$$

Given the negative translation above, such a Horn clause would then be translated to the linear logic formula

$$(A_1 \otimes \ldots \otimes A_n) \multimap A_0.$$

Thus, the comma in Prolog denotes $\otimes$ and $:-$ denotes the converse of $\multimap$.

For another example, the natural deduction rule for the introduction of implication, often expressed using the diagram

$$\dfrac{\begin{array}{c}(A)\\ \vdots \\ B\end{array}}{A \supset B},$$

can be written as the following first-order formula for axiomatizing a provability predicate:

$$\forall A \forall B((prov(A) \supset prov(B)) \supset prov(A\ imp\ B)),$$

where the domain of quantification is over propositional formulas of the object-language and *imp* is the object-level implication. This formula is written in λProlog using the syntax

```
prov (A imp B)  :-  prov A => prov B.
```

Given the above proposition, this formula can be translated to the formula

$$\forall A \forall B((prov\ A \Rightarrow prov\ B) \multimap prov\ (A\ imp\ B)),$$

which means that the λProlog symbol `=>` should denote $\Rightarrow$. Thus, in the implication introduction rule displayed above, the meta-level implication represented as three vertical dots can be interpreted as an intuitionistic implication while the meta-level implication represented as the horizontal bar can be interpreted as a linear implication.

In the next chapter, we will present numerous example of logic programs using $\mathcal{L}_1$ formulas that illustrate features of linear logic. We give a simple example here. Assume that we would like to move from, say, `step1` to `step2` in a computation (proof search) and in the process of making that change, we wish to flip a switch. In other words, we would like to write a logic specification that makes the following synthetic inference rules possible.

$$\frac{\Delta;\Gamma,\texttt{on}\vdash\texttt{step2}}{\Delta;\Gamma,\texttt{off}\vdash\texttt{step1}}\qquad\frac{\Delta;\Gamma,\texttt{off}\vdash\texttt{step2}}{\Delta;\Gamma,\texttt{on}\vdash\texttt{step1}}$$

Using the Prolog-style syntax described above, the following two clauses implement these synthetic rules.

```
step1 :- off, on  -o step2.
step1 :- on,  off -o step2.
```

To illustrate this, assume that the two (equivalent) formulas

$$\texttt{off}\multimap(\texttt{on}\multimap\texttt{step2})\multimap\texttt{step1},\quad\texttt{on}\multimap(\texttt{off}\multimap\texttt{step2})\multimap\texttt{step1}$$

are members of Δ. We have the following partial derivation in $\Downarrow\mathcal{L}_1$ to justify the second of the synthetic rules above.

$$\dfrac{\dfrac{\dfrac{}{\Delta;\cdot\Downarrow\texttt{on}\vdash\texttt{on}}\;\mathit{init}}{\Delta;\texttt{on}\vdash\texttt{on}}\;\mathit{decide}\qquad\dfrac{\dfrac{\Delta;\Gamma,\texttt{off}\vdash\texttt{step2}}{\Delta;\Gamma\vdash\texttt{off}\multimap\texttt{step2}}\multimap R\qquad\dfrac{}{\Delta;\cdot\Downarrow\texttt{step1}\vdash\texttt{step1}}\;\mathit{init}}{\Delta;\Gamma\Downarrow(\texttt{off}\multimap\texttt{step2})\multimap\texttt{step1}\vdash\texttt{step1}}\multimap L}{\dfrac{\Delta;\Gamma,\texttt{on}\Downarrow\texttt{on}\multimap(\texttt{off}\multimap\texttt{step2})\multimap\texttt{step1}\vdash\texttt{step1}}{\Delta;\Gamma,\texttt{on}\vdash\texttt{step1}}\;\mathit{decide}\,!}\multimap L$$

The two occurrences of $\multimap L$ require splitting the bounded context in their conclusion. There can be many possible splittings of these multisets, depending on the size of Γ. However, in this particular setting, the bound context can only be split one way: all other splitting would not have allowed for completing the phase and, thus, forming the synthetic rule. If $\Rightarrow$ replaced $\multimap$ in this example, the resulting synthetic rules would be

$$\frac{\Delta,\texttt{off},\texttt{on};\cdot\vdash\texttt{step2}}{\Delta,\texttt{off};\cdot\vdash\texttt{step1}}\qquad\frac{\Delta,\texttt{on},\texttt{off};\cdot\vdash\texttt{step2}}{\Delta,\texttt{on};\cdot\vdash\texttt{step1}}$$

Clearly, this would be a poor implementation of a switch.

6.6 A model of resource consumption

This text does not present many details about the implementation of proof search, but the following considerations seem high-level and useful to mention. As we mentioned in Section 6.4, attempt to apply the multiplicative inference rule $\multimap L$ from either Figure 6.6 or Figure 6.7 requires splitting a multiset of formulas into two multisets: in general, an exponential number of such splittings is possible. A better strategy than trying each possible splitting is

$$\dfrac{\texttt{subcontext}\ O\ I}{\Sigma:\mathcal{P};\ [I \parallel O] \vdash \top}\ \top R \qquad \dfrac{\Sigma:\mathcal{P};\ [I \parallel O] \vdash B \qquad \Sigma:\mathcal{P};\ [I \parallel O] \vdash C}{\Sigma:\mathcal{P};\ [I \parallel O] \vdash B \mathbin{\&} C}\ \&\mathrm{R}$$

$$\dfrac{\Sigma:\mathcal{P};\ [\langle B\rangle :: I \parallel \circ :: O] \vdash C}{\Sigma:\mathcal{P};\ [I \parallel O] \vdash B \multimap C}\ \multimap\mathrm{R} \qquad \dfrac{\Sigma:\mathcal{P}, B;\ [I \parallel O] \vdash C}{\Sigma:\mathcal{P};\ [I \parallel O] \vdash B \Rightarrow C}\ \Rightarrow R$$

$$\dfrac{y:\tau, \Sigma:\mathcal{P};\ [I \parallel O] \vdash B[y/x]}{\Sigma:\mathcal{P};\ [I \parallel O] \vdash \forall x_\tau . B}\ \forall R$$

$$\dfrac{\Sigma:\mathcal{P}, D; [I \parallel O] \Downarrow D \vdash A}{\Sigma:\mathcal{P}, D;\ [I \parallel O] \vdash A}\ \mathit{decide}\,!$$

$$\dfrac{\texttt{pick}\ I\ D\ M \qquad \Sigma:\mathcal{P}; [M \parallel O] \Downarrow D \vdash A}{\Sigma:\mathcal{P};\ [I \parallel O] \vdash A}\ \mathit{decide}$$

$$\dfrac{}{\Sigma:\mathcal{P}; [I \parallel I] \Downarrow A \vdash A}\ \mathit{init} \qquad \dfrac{\Sigma \Vdash t:\tau \qquad \Sigma:\mathcal{P}; [I \parallel O] \Downarrow D[t/x] \vdash A}{\Sigma:\mathcal{P}; [I \parallel O] \Downarrow \forall_\tau x.D \vdash A}\ \forall\mathrm{L}$$

$$\dfrac{\Sigma:\mathcal{P}; [I \parallel O] \Downarrow D_i \vdash A}{\Sigma:\mathcal{P}; [I \parallel O] \Downarrow D_1 \mathbin{\&} D_2 \vdash A}\ \&\mathrm{L}\ (i \in \{1,2\})$$

$$\dfrac{\Sigma:\mathcal{P};\ [I \parallel M] \vdash G \qquad \Sigma:\mathcal{P}; [M \parallel O] \Downarrow D \vdash A}{\Sigma:\mathcal{P}; [I \parallel O] \Downarrow G \multimap D \vdash A}\ \multimap\mathrm{L}$$

$$\dfrac{\Sigma:\mathcal{P};\ [I \parallel I] \vdash G \qquad \Sigma:\mathcal{P}; [I \parallel O] \Downarrow D \vdash A}{\Sigma:\mathcal{P}; [I \parallel O] \Downarrow G \Rightarrow D \vdash A}\ \Rightarrow\mathrm{L}$$

Figure 6.9: The IO proof system.

needed if the logic $\mathcal{L}_1$ is to be the foundations of a usable logic programming language. Such a strategy is possible and rests on two observations. First, instead of splitting the formulas in the bounded left context at the moment of applying the $\multimap$L rule, we can send all the formulas in the bounded context to the process searching for a proof of the left premise. If a proof of that premise is found, some of those bounded formulas are consumed. The remaining, unconsumed, bounded formulas can then be sent to the right premise to be consumed there. Second, the decide inference rule in Figure 6.7 actually consumes a bounded formula.

Figure 6.9 contains the IO proof system, which is a modification of the $\Downarrow\mathcal{L}_1$ proof system in Figure 6.7 in which the bounded context (written using the schematic variable Γ) is replaced by the pairing $[I \parallel O]$, where I and O denote, respectively, collections of *input* and *output* formulas. Since we need to support the process of *deleting* formulas from an input inorder to arrive at

```
kind opt           type -> type.
type none          opt A.
type some          A -> opt A.
type pick          list (opt A) -> A -> list (opt A) -> o.
type subcontext    list (opt A)      -> list (opt A) -> o.

pick (some B::I) B (none::I).
pick (C::I)      B (C::O)    :- pick I B O.

subcontext nil        nil.
subcontext (C::O)     (C::I)      :- subcontext O I.
subcontext (none::O)  (some B::I) :- subcontext O I.
```

Figure 6.10: The formal definition of the two predicates used in Figure 6.9.

an output, the structures encoding I and O will be lists of *option-formulas* and not just of formulas. By an option-formula we mean a term of the form $\langle B\rangle$, for B a formula, or $\circ$, which denotes that a formula has been deleted. The pick relation used in the *decide* rule in Figure 6.9 is used to select an occurrence of a formula from an input list and return the result of deleting that occurrence in the output list. The formal definition of the pick and subcontext predicates is given using the Horn clauses displayed in Figure 6.10.

There are several observations to make about the rules in Figure 6.9.

1. Most rules are such that when the pair $[I \parallel O]$ appears in the conclusion, it also appears in all its premises. The exceptions are described next.

2. When reading inference rules from conclusion to premises, the $\multimap$R rule can be seen as taking the pair $[I \parallel O]$ and giving to the premise the input $\langle B\rangle$. At the same time, the corresponding output structure contains $\circ$ which denotes the deletion of B. As a result, this modified rule indicates that the formula B must be consumed in the proof of the premise.

3. The *decide* rule employs the pick predicate to nondeterministically select a formula D from the input structure while making it deleted in the output structure.

4. The left premise of the $\Rightarrow$L rule contains the pairing $[I \parallel I]$. Such a pairing means that all formulas in the input are also in the output: that is, no formulas have been deleted. The *init* rule uses a similar pairing.

5. The condition subcontext O I appearing in the premise of the $\top$R is true if O results from deleting some formulas occurring in I.

Exercise 6.19. The predicate `subcontext` can be removed from the proof system in Figure 6.9 by making use of the `pick` predicate instead. In particular, show that the one rule in Figure 6.9 that references `subcontext` can be replaced by the following two rules.

$$\frac{}{\Sigma : \mathcal{P};\ [I \parallel I] \vdash \top}\ \top R \qquad \frac{\texttt{pick}\ \mathrm{I\ D\ M} \quad \Sigma : \mathcal{P};\ [M \parallel O] \vdash \top}{\Sigma : \mathcal{P};\ [I \parallel O] \vdash \top}\ \top R$$

In order to prove the correctness of the proof system in Figure 6.9, we define the formal difference, $I - O$ whenever it the case that `subcontext` $O\ I$ holds: in particular, $I - O$ is the multiset of formulas D such that $\langle D \rangle$ occurs in I and the corresponding position in O is the symbol $\circ$.

Lemma 6.20. *Given a list of option-formulas I, the difference $I - I$ is the empty multiset. Whenever `subcontext` $I\ M$ and `subcontext` $M\ O$ hold then `subcontext` $I\ O$ holds and $I - O$ is the multiset union of $I - M$ and $M - O$. Finally, if `pick` $I\ D\ O$ holds then $I - O$ is the multiset containing one occurrence of D.*

The following lemma is proved by a simple induction on the structure of IO-proofs.

Lemma 6.21. *If $\Sigma : \mathcal{P};\ [I \parallel O] \vdash G$ has an IO-proof then `subcontext` $O\ I$ holds. Similiarly, if $\Sigma{:}\mathcal{P};\ [I \parallel O] \Downarrow D \vdash G$ has an IO-proof then `subcontext` $O\ I$ holds.*

The following proposition shows that this approach to the lazy splitting of contexts is sound.

Proposition 6.22. *If $\Sigma{:}\mathcal{P};\ [I \parallel O] \vdash G$ has an IO-proof then $\Sigma{:}\mathcal{P}; I - O \vdash G$ has a $\Downarrow\mathcal{L}_1$-proof. Similiarly, if $\Sigma : \mathcal{P}; [I \parallel O] \Downarrow D \vdash G$ has an IO-proof then $\Sigma : \mathcal{P}; I - O \Downarrow D \vdash G$ has a $\Downarrow\mathcal{L}_1$-proof.*

Proof. Let Ξ be an IO-proof of $\Sigma : \mathcal{P};\ [I \parallel O] \vdash G$. We can convert Ξ to an $\Downarrow\mathcal{L}_1$-proof by simply replacing every occurrence of the pairing $[I \parallel O]$ in Ξ with the multiset $I - O$. For example, consider the IO inference rule

$$\frac{\Sigma : \mathcal{P};\ [I \parallel M] \vdash G \qquad \Sigma : \mathcal{P}; [M \parallel O] \Downarrow D \vdash A}{\Sigma : \mathcal{P};\ [I \parallel O] \Downarrow G \multimap D \vdash A}\ \multimap\mathrm{L}$$

If we set Γ_1 and Γ_2 to be, respectively, $I - M$ and $M - O$, then by Lemma 6.20, $I - O$ is the multiset union of Γ_1 and Γ_2. Thus, the rule above is converted to the $\Downarrow\mathcal{L}_1$ inference rule

$$\frac{\Sigma : \mathcal{P}; \Gamma_1 \vdash G \qquad \Sigma : \mathcal{P}; \Gamma_2 \Downarrow D \vdash A}{\Sigma : \mathcal{P}; \Gamma_1, \Gamma_2 \Downarrow G \multimap D \vdash A}\ \multimap\mathrm{L}$$

The remaining cases all follow as simply as this case. □

I should also prove the following completeness theorem eventhough it seems to be rather technical. The fact that I/O contexts are ordered seems to be the main catch.

Proposition 6.23. *If $\Sigma : \mathcal{P}; \Gamma \vdash G$ has a $\Downarrow\mathcal{L}_1$-proof then there are lists of option-formulas I and O such that* **subcontext** *O I holds and $\Sigma : \mathcal{P}; [I \parallel O] \vdash G$ has an IO-proof. Similiarly, if $\Sigma : \mathcal{P}; I - O \Downarrow D \vdash G$ has a $\Downarrow\mathcal{L}_1$-proof then there are lists of option-formulas I and O such that* **subcontext** *O I holds and $\Sigma : \mathcal{P}; [I \parallel O] \Downarrow D \vdash G$ has an IO-proof then*

6.7 Multiple conclusion uniform proofs

Our treatment of linear logic proof theory via goal directed search and backchaining is only able to capture a part of linear logic. As we saw in Exercise 6.17, if we extend the $\mathcal{L}_1$ collection of connectives with $\perp$, we can encode all of linear logic's connectives. This suggests adding the 0-ary, multiplicative disjunction might be interesting to consider, especially since it has negative polarity, like the other connectives in $\mathcal{L}_1$. In fact, it would seem sensible to add not just $\perp$ but also $⅋$ and $?$ since they are all negative polarity connectives and they represent the 0-ary, 2-ary, and "∞-ary" multiplicative disjunction. To that end, we define $\mathcal{L}_2$ to be the set of connectives

$$\mathcal{L}_2 = \{\top, \&, \multimap, \Rightarrow, \forall, \perp, ⅋, ?\}$$

and we say that an $\mathcal{L}_2$-formula is any first-order formula built using the $\mathcal{L}_2$ connectives. Of course, sequent calculus proofs involving these additional connectives forces us to consider multiple conclusion sequent calculus. This presentation of linear logic using the logical connectives in $\mathcal{L}_2$ is called the *Forum presentation of linear logic*.

The set of connectives $\mathcal{L}_2$ is redundant since we can remove $⅋$ and $?$ and still have a set of connectives that is complete for linear logic, as the following linear logic equivalences validate.

$$?\,B \equiv (B \multimap \perp) \Rightarrow \perp \qquad B ⅋ C \equiv (B \multimap \perp) \multimap C$$

While the addition of $⅋$ and $?$ is not strictly necessary, their presences will allow us to write natural specifications later one. Also, their presence does not seem to complicate the proof theory analysis we consider in the following section.

What should it mean to do goal-directed search when there are possibly several formulas on the right of a sequent? The key aspect of goal-directed search that we wish to maintain is that goal formulas (right-hand side formulas) are able to be introduced without any restriction, no matter what other formulas are on the left or right of the sequent arrow. Thus, it seems natural to expect that we should be able to *simultaneously* introduce all the logical connectives on the right of the sequent arrow. Although the sequent calculus cannot deal directly with simultaneous rule application, reference to *permutabilities* of inference rules can indirectly address simultaneity. That is, we can require that if two or more right-introduction rules can be used to derive a given sequent, then all possible orders of applying those right-introduction rules can, in fact, be done and the resulting proofs are all equal modulo permutations of introduction rules.

More precisely: A cut-free sequent proof Ξ is *uniform* if for every subproof Ξ' of Ξ and for every non-atomic formula occurrence B in the right-hand side of the end-sequent of Ξ', there is a proof Ξ'' that is equal to Ξ' up to a permutation of inference rules and is such that the last inference rule in Ξ'' introduces the top-level logical connective of B. Clearly this notion of uniform proof extends the one given in Section 5.1. We similarly extend the notion of *abstract logic programming language* to be a triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ such that for all sequents with formulas from $\mathcal{D}$ on the left and formulas from $\mathcal{G}$ on the right, that sequent has a proof if and only if it has a uniform proof.

The $\Downarrow\mathcal{L}_2$ proof system for the Forum presentation of linear logic, given in Figure 6.11, contains sequents having the form

$$\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon \quad \text{and} \quad \Sigma : \Psi; \Gamma \Downarrow B \vdash \Delta; \Upsilon,$$

where Σ is a signature, and Γ, Δ, Ψ and Υ are multiset of Σ-formulas from $\mathcal{L}_2$. The intended meanings of these two sequents in linear logic are

$$\Sigma : !\,\Psi, \Gamma \vdash \Delta, ?\,\Upsilon \quad \text{and} \quad \Sigma : !\,\Psi, \Gamma, B \vdash \Delta, ?\,\Upsilon,$$

respectively. The $\Downarrow\mathcal{L}_2$ proof system contains right rules only for sequents of the form $\Sigma{:}\Psi; \Gamma \vdash \Delta; \Upsilon$. The syntactic variable $\mathcal{A}$ used in Figure 6.11 denotes a multiset of atomic formulas. As we have seen before, left-introduction rules are applied only to the formula that is next to the $\Downarrow$ in its conclusion. Given that the $\mathcal{L}_2$ connectives have negative polarity, all occurrences of right-introduction rules in proofs involving them are invertible. This observation makes it an easy matter to prove that uniform proofs are complete.

The **L** proof system can serve as an (unfocused) proof system for $\mathcal{L}_2$: we simply need to replace the implications in $\mathcal{L}_2$-formulas with their definitions, using the $(\cdot)^\diamond$ function given with the statement of Proposition 6.14. Given the intended interpretation of sequents in $\Downarrow\mathcal{L}_2$, the following soundness theorem can be proved by simple induction on the structure of $\Downarrow\mathcal{L}_2$ proofs.

Theorem 6.24 (Soundness). *If the sequent $\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon$ has a $\Downarrow\mathcal{L}_2$ proof then $!\,\Psi^\diamond, \Gamma^\diamond \vdash \Delta^\diamond, ?\,\Upsilon^\diamond$ has a linear logic proof. If the sequent $\Sigma : \Psi; \Gamma \Downarrow B \vdash \mathcal{A}; \Upsilon$ has a $\Downarrow\mathcal{L}_2$ proof then $!\,\Psi^\diamond, \Gamma^\diamond, B^\diamond \vdash \Delta^\diamond, ?\,\Upsilon^\diamond$.*

$$\frac{}{\Sigma : \Psi; \Gamma \vdash \top, \Delta; \Upsilon}\ \top R$$

$$\frac{\Sigma : \Psi; \Gamma \vdash B, \Delta; \Upsilon \quad \Sigma : \Psi; \Gamma \vdash C, \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash B \mathbin{\&} C, \Delta; \Upsilon}\ \&\mathrm{R}$$

$$\frac{\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash \bot, \Delta; \Upsilon}\ \bot R \qquad \frac{\Sigma : \Psi; \Gamma \vdash B, C, \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash B \parr C, \Delta; \Upsilon}\ \parr R$$

$$\frac{\Sigma : \Psi; B, \Gamma \vdash C, \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash B \multimap C, \Delta; \Upsilon}\ \multimap R \qquad \frac{\Sigma : B, \Psi; \Gamma \vdash C, \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash B \Rightarrow C, \Delta; \Upsilon}\ \Rightarrow R$$

$$\frac{y : \tau, \Sigma : \Psi; \Gamma \vdash B[y/x], \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash \forall_\tau x.B, \Delta; \Upsilon}\ \forall R \qquad \frac{\Sigma : \Psi; \Gamma \vdash \Delta; B, \Upsilon}{\Sigma : \Psi; \Gamma \vdash ?\,B, \Delta; \Upsilon}\ ?\,R$$

$$\frac{\Sigma : \Psi; \Gamma \Downarrow B \vdash \mathcal{A}; \Upsilon}{\Sigma : \Psi; B, \Gamma \vdash \mathcal{A}; \Upsilon}\ \mathit{decide}$$

$$\frac{\Sigma : B, \Psi; \Gamma \Downarrow B \vdash \mathcal{A}; \Upsilon}{\Sigma : B, \Psi; \Gamma \vdash \mathcal{A}; \Upsilon}\ \mathit{decide}\,! \qquad \frac{\Sigma : \Psi; \Gamma \vdash \mathcal{A}, B; B, \Upsilon}{\Sigma : \Psi; \Gamma \vdash \mathcal{A}; B, \Upsilon}\ \mathit{decide}\,?$$

$$\frac{}{\Sigma : \Psi; \cdot \Downarrow A \vdash A; \Upsilon}\ \mathit{init} \qquad \frac{}{\Sigma : \Psi; \cdot \Downarrow A \vdash \cdot; A, \Upsilon}\ \mathit{init}\,?$$

$$\frac{}{\Sigma : \Psi; \cdot \Downarrow \bot \vdash \cdot; \Upsilon}\ \bot L \qquad \frac{\Sigma : \Psi; B \vdash \cdot; \Upsilon}{\Sigma : \Psi; \cdot \Downarrow ?\,B \vdash \cdot; \Upsilon}\ ?\,L$$

$$\frac{\Sigma : \Psi; \Gamma \Downarrow B_i \vdash \mathcal{A}; \Upsilon}{\Sigma : \Psi; \Gamma \Downarrow B_1 \mathbin{\&} B_2 \vdash \mathcal{A}; \Upsilon}\ \&\mathrm{L}_i \qquad \frac{\Sigma : \Psi; \Gamma \Downarrow B[t/x] \vdash \mathcal{A}; \Upsilon}{\Sigma : \Psi; \Gamma \Downarrow \forall_\tau x.B \vdash \mathcal{A}; \Upsilon}\ \forall L$$

$$\frac{\Sigma : \Psi; \Gamma_1 \Downarrow B \vdash \mathcal{A}_1; \Upsilon \quad \Sigma : \Psi; \Gamma_2 \Downarrow C \vdash \mathcal{A}_2; \Upsilon}{\Sigma : \Psi; \Gamma_1, \Gamma_2 \Downarrow B \parr C \vdash \mathcal{A}_1, \mathcal{A}_2; \Upsilon}\ \parr L$$

$$\frac{\Sigma : \Psi; \Gamma_1 \vdash \mathcal{A}_1, B; \Upsilon \quad \Sigma : \Psi; \Gamma_2 \Downarrow C \vdash \mathcal{A}_2; \Upsilon}{\Sigma : \Psi; \Gamma_1, \Gamma_2 \Downarrow B \multimap C \vdash \mathcal{A}_1, \mathcal{A}_2; \Upsilon}\ \multimap\mathrm{L}$$

$$\frac{\Sigma : \Psi; \cdot \vdash B; \Upsilon \quad \Sigma : \Psi; \Gamma \Downarrow C \vdash \mathcal{A}; \Upsilon}{\Sigma : \Psi; \Gamma \Downarrow B \Rightarrow C \vdash \mathcal{A}; \Upsilon}\ \Rightarrow\mathrm{L}$$

Figure 6.11: The $\Downarrow\mathcal{L}_2$ proof system. The rule ∀R has the proviso that y is not in the signature Σ, and the rule ∀L has the proviso that t is a Σ-term of type τ. In $\&L_i$, $i = 1$ or $i = 2$. Cut rules for $\Downarrow\mathcal{L}_2$ will be considered in Figure 6.12.

As a presentation of linear logic, Forum and its proof system $\Downarrow\mathcal{L}_2$ are rather odd. First, Forum's proof system does not contain the cut-rule whereas most presentation of linear logic are concerned with the dynamics of cut-elimination. Since we are interested in proof search instead of proof normalization, this dispensing with the cut-rule is understandable. Second, negation is not a primitive and the De Morgan dual of a logical connective in $\mathcal{L}_2$ is not, in fact, present in $\mathcal{L}_2$. Again, most proof systems for linear logic (even the one in Figure 6.4) are more symmetric in that if they contain a connective, they also contain its dual. Instead, Forum gives the two implications, $\multimap$ and $\Rightarrow$, a central role and this contributes to the asymmetric nature of Forum. On the other hand, the decision to use implications makes it easy for Forum to generalize logic programming based on Horn clauses, hereditary Harrop formulas, and Lolli. Although cut is not an inference rule and duality is not a feature of the logical connectives used in Forum, cut-elimination and duality will play a significant role in how one reasons about Forum specifications.

Exercise 6.25. Assume that a, b, c, d are all propositional constants (i.e., they have type o). Prove the following formulas using the $\Downarrow\mathcal{L}_2$ proof system. Note that proving B using $\Downarrow\mathcal{L}_2$ means to prove the sequent $\cdot : \cdot; \cdot \vdash B; \cdot$.

- $((a \multimap \bot) \multimap \bot) \multimap a$,
- $(d \multimap (a \parr b)) \multimap (\mathbf{1} \multimap (c \parr d)) \multimap (a \parr b \parr c)$
- $?\,b \multimap (b \multimap \bot) \Rightarrow \bot$ and $((b \multimap \bot) \Rightarrow \bot) \multimap ?\,b$
- $b \parr c \multimap (b \multimap \bot) \multimap c$ and $((b \multimap \bot) \multimap c) \multimap (b \parr c)$

Exercise 6.26. The proof rule in $\Downarrow\mathcal{L}_2$ for $?\,L$ is unlike the other left rules in that it does not maintain focus as one moves from the conclusion to a premise. Consider the following variation to that inference rule.

$$\frac{\Sigma : \Psi; \cdot \Downarrow B \vdash \cdot; \Upsilon}{\Sigma : \Psi; \cdot \Downarrow ?\,B \vdash \cdot; \Upsilon}\ ?\,L'$$

Show that if we replace $?\,L$ with $?\,L'$ then the resulting proof system is no longer complete. In particular, the formula $?(a \multimap b) \multimap ?(a \multimap b)$ does not have a proof.

Exercise 6.27. The $\mathcal{L}_2$ presentation of linear uses the 8 logical connectives $\{\top, \&, \multimap, \Rightarrow, \forall, \bot, \parr, ?\}$. Show that all the 64 pairings of the right-introduction rules for these 8 connectives permutes over each other.

6.8 Formal properties of Forum proofs

We shall now establish the main proof theory results regarding the Forum presentation of linear logic. This section follows roughly the outline of results that are given in Section 5.5 for the $\mathcal{L}_0$ subset of intuitionistic logic. The outline for this section is the following.

1. Define the notion of path in formulas and their associated sequent.
2. Use paths to describe the right-introduction and left-introduction phases.
3. Prove the admissibility of the non-atomic initial rule in $\Downarrow\mathcal{L}_2$.
4. Add three cut rules to $\Downarrow\mathcal{L}_2$ and then prove that they can be eliminated.
5. Prove the completeness of $\Downarrow\mathcal{L}_2$ with respect of the unfocused **L**.
6. Prove the cut-elimination theorem for the **L** proof system.

6.8.1 Paths and synthetic inference rules

We move the notion of path given in Section 5.5 from $\mathcal{L}_0$-formulas to $\mathcal{L}_2$-formulas. In particular, we define the relationship $\cdot \uparrow \cdot$ on $\mathcal{L}_2$-formulas as follows (here, A ranges over atomic formulas).

$$\frac{}{A \uparrow A} \qquad \frac{B_1 \uparrow P}{B_1 \mathbin{\&} B_2 \uparrow P} \qquad \frac{B_2 \uparrow P}{B_1 \mathbin{\&} B_2 \uparrow P} \qquad \frac{B \uparrow P}{C \Rightarrow B \uparrow C \Rightarrow P} \qquad \frac{B \uparrow P}{\forall_\tau x.B \uparrow \forall_\tau x.P}$$

$$\frac{}{\bot \uparrow \bot} \qquad \frac{}{?\,B \uparrow ?\,B} \qquad \frac{B \uparrow P}{C \multimap B \uparrow C \multimap P} \qquad \frac{B_1 \uparrow P_1 \quad B_2 \uparrow P_2}{B_1 \parr B_2 \uparrow P_1 \parr P_2}$$

The elimination of & from paths can be seen as justified using the following equivalences.

$$B \parr (C_1 \mathbin{\&} C_2) \equiv (B \parr C_1) \mathbin{\&} (B \parr C_2) \tag{6.1}$$
$$B \multimap (C_1 \mathbin{\&} C_2) \equiv (B \multimap C_1) \mathbin{\&} (B \multimap C_2) \tag{6.2}$$

Using these equivalences (and other equivalences related to $\Rightarrow$ and $\forall$), it is possible to pull all occurrences of & within a formula to the outside of the formula. That is, we have $B \equiv \bigwith_{B\uparrow P} P$.

In general, paths have a more complex structure in this setting than we saw in Section 5.5. Fortunately, paths have a reasonably simple normal form.

Using the equivalences

$$B \parr (\forall x.C) \equiv (\forall x.B \parr C) \tag{6.3}$$
$$B \multimap (\forall x.C) \equiv (\forall x.B \multimap C) \tag{6.4}$$
$$B \Rightarrow (\forall x.C) \equiv (\forall x.B \Rightarrow C), \tag{6.5}$$

a path can be written in the form $\forall x_1 \ldots \forall n_n.P'$ where $n \geq 0$ and every occurrence of $\forall$ in P' occurs in the scope of a ? or to the left of either $\multimap$ or $\Rightarrow$. Similarly, using the equivalences

$$(B \multimap C_1) \parr C_2 \equiv B \multimap (C_1 \parr C_2) \tag{6.6}$$
$$(B \Rightarrow C_1) \parr C_2 \equiv B \Rightarrow (C_1 \parr C_2) \tag{6.7}$$
$$B \multimap C \Rightarrow D \equiv C \Rightarrow B \multimap D \tag{6.8}$$

and the unit rules $\bot \parr B \equiv B \parr \bot \equiv B$ and the commutativity of $\parr$, all paths have the following normal form.

$$\forall \bar{x}[C_1 \Rightarrow \ldots \Rightarrow C_n \Rightarrow B_1 \multimap \ldots \multimap B_m \multimap A_1 \parr \ldots \parr A_p \parr ?\,E_1 \ldots \parr ?\,E_q]$$

where n, m, p, q are non-negative integers, $A_1, \ldots, A_p$ are atomic formulas, $B_1, \ldots, B_m, C_1, \ldots, C_n, E_1, \ldots, E_q$ are $\mathcal{L}_2$ formulas, and $\forall \bar{x}$ is a list of universally quantified variables. If a path P has the normal form above, then we say that the multiset $\{C_1, \ldots, C_n\}$ is its *intuitionistic arguments*, the multiset $\{B_1, \ldots, B_m\}$ is its *linear arguments*, the multiset $\{A_1, \ldots, A_p\}$ is its *atomic targets*, and the multiset $\{E_1, \ldots, E_q\}$ is its *?-targets*. Finally, $\bar{x}$ is the list of *bound variables* of P (we assume that all these bound variables are distinct). Since these various components to the normal form of a path are multisets, this decomposition of a path is unique. We shall also display this normal form as the sequent

$$\Sigma : C_1, \ldots, C_n; B_1, \ldots, B_m \vdash A_1, \ldots, A_p; E_1, \ldots, E_q.$$

Consider what the right-introduction phase and the left-introduction phase are when applied to the following formula

$$\forall \bar{x}(C \Rightarrow B_1 \multimap B_2 \multimap A_1 \parr A_2 \parr ?\,E),$$

which is its own path formula since it has no occurrences of &. The right-introduction phase can be written schematically as follows.

$$\frac{\bar{x} : C; B_1, B_2 \vdash A_1, A_2; E}{\cdot : \cdot; \cdot \vdash \forall \bar{x}(C \Rightarrow B_1 \multimap B_2 \multimap A_1 \parr A_2 \parr ?\,E); \cdot}$$

Note that the unique premise to this phase ends with the sequent representation associate to that path. Of course, if we place any items in any of the zones in the conclusion, they should also be placed into the same zone in the premise. Focusing on this example formula leads to the following derivation.

$$\dfrac{\Psi; \cdot \vdash \hat{C}; \Upsilon \qquad \Psi; \Gamma_1 \vdash \hat{B}_1, \mathcal{A}_1; \Upsilon \qquad \Psi; \Gamma_2 \vdash \hat{B}_2, \mathcal{A}_2; \Upsilon \qquad \Psi; \hat{E} \vdash \cdot; \Upsilon}{\Psi; \Gamma_1, \Gamma_2 \Downarrow \forall \bar{x}(C \Rightarrow B_1 \multimap B_2 \multimap A_1 \parr A_2 \parr ?\, E) \vdash \hat{A}_1, \hat{A}_2, \mathcal{A}_1, \mathcal{A}_2; \Upsilon}$$

Here, $\hat{A}_1, \hat{A}_2, \hat{B}_1, \hat{B}_2, \hat{C}, \hat{E}$ are the result of applying θ to the formulas in A_1, A_2, B_1, B_2, C, E, and θ is the substitution for the variables $\bar{x}$ that tabulates the substitutions used in the $\forall R$ rules.

To improve readability of sequents and derivations, we shall often not display signatures (such as Σ in the previous example). Furthermore, we shall often place a " in a particular zone of an occurrence of a sequent to means that the contents of that zone is taken from the sequent *below* it in a derivation.

We generalize the following two notions introduced in Section 5.8. A *border* sequent is a sequent of the form $\Sigma : \Psi; \Gamma \vdash \mathcal{A}; \Upsilon$: that is, they are four-zone sequents in which the right bounded context contains only atoms. (Since occurrences of Σ in sequent denoting binders, we shall not refer to it as a zone.) A *synthetic inference rule* is then the inference rule that results from moving from a border sequent upwards through a *decide* or *decide*! rule, followed by a left-introduction phase and then a right-introduction phase: if the latter has any open premises, these are necessarily border phases. Schematically, a synthetic inference rule can be seen as composed of focused inference rules as follows.

$$\dfrac{\dfrac{\dfrac{\ldots \quad \Sigma, \Sigma' : \Psi, \Psi'; \Gamma' \vdash \mathcal{A}'; \Upsilon, \Upsilon' \quad \ldots}{\vdots \quad \ldots \quad \vdots}\ \text{right-intro phase}}{\vdots \qquad \vdots \qquad \vdots}\ \text{left-intro phase}}{\Sigma : \Psi; \Gamma \vdash \mathcal{A}; \Upsilon}\ \textit{decide} \text{ or } \textit{decide}\,!$$

The *decide* ? rule can also generate synthetic inferences rule but the internal structure of such a rule has an empty left-introduction phase.

We can view the construction of the right-introduction phase as a rewriting process. The objects that we rewrite are multisets of sequents all of the form $\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon$. One-step rewriting is given as following. Select some member of this multiset: i.e., write the given multiset of sequents as $\mathcal{M} \cup \{\mathbf{S}\}$. Next, consider any right-introduction rule that has conclusion $\mathbf{S}$ and the multiset of premises $\mathcal{M}'$ (this multiset will contain 0, 1, or 2 elements). The multiset union $\mathcal{M} \cup \mathcal{M}'$ is the result of this rewrite. When this relation holds, we write

$$\mathcal{M} \cup \{\mathbf{S}\} \to \mathcal{M} \cup \mathcal{M}'$$

The following observations are easy to make about this notion of rewriting.

1. A multiset of border sequents does not rewrite. In this sense, collections of border sequents are normal forms.
2. Define the size of sequents of the form $\Sigma\colon\Psi;\Gamma\vdash\Delta;\Upsilon$ to be the number of occurrences of logical connectives in Δ, and define the size of a multiset $\mathcal{M}$ to be the sum of the sizes of all sequents in $\mathcal{M}$. The length of a series of rewritings starting with $\mathcal{M}$ is bounded by the size of $\mathcal{M}$. Thus, this rewriting system is always terminating.

What we really wish to prove is that every right-introduction phase with a fixed endsequent has the same multiset of premises. In terms of rewriting, we want to prove that our rewriting system is *confluent*. As is well-known, we only need to prove that our system is *locally confluence* in order to conclude that our terminating rewrite system is confluence. In our situation, proving local confluence means proving that if $\mathcal{M}$ rewrites in one step to $\mathcal{M}_1$ and to $\mathcal{M}_2$, then there exists $\mathcal{M}_0$ such that both $\mathcal{M}_1$ and $\mathcal{M}_2$ rewrite to $\mathcal{M}_0$.

Proposition 6.28. *The rewriting systems encoding the right-introduction phase is confluent.*

Proof. As we commented above, we only need to show local confluence. Thus, assume that $\mathcal{M}$ rewrites in one step to $\mathcal{M}_1$ and to $\mathcal{M}_2$. We now need to prove that there exists $\mathcal{M}_0$ such that both $\mathcal{M}_1$ and $\mathcal{M}_2$ rewrite to $\mathcal{M}_0$. In the event that the two rewrites $\mathcal{M}\to\mathcal{M}_1$ and $\mathcal{M}\to\mathcal{M}_2$ select two different sequents to apply introduction rules, then $\mathcal{M}_0$ is just the result of rewriting those two sequents in parallel. Otherwise, these two rewrite work on the same sequent in $\mathcal{M}$, say, $\Sigma\colon\Psi;\Gamma\vdash\Delta;\Upsilon$. Thus, there are two non-atomic formulas in Δ that are introduced. For example, the multiset

$$\mathcal{M}\cup\{\Sigma:\Psi;\Gamma\vdash B \parr C, D \mathbin{\&} E, \Delta';\Upsilon\}$$

can be rewritten to both

$$\mathcal{M}\cup\{\Sigma:\Psi;\Gamma\vdash B, C, D \mathbin{\&} E, \Delta';\Upsilon\}$$

and to

$$\mathcal{M}\cup\{\Sigma:\Psi;\Gamma\vdash B \parr C, D, \Delta';\Upsilon,\quad \Sigma:\Psi;\Gamma\vdash B \parr C, E, \Delta';\Upsilon\}$$

Since the right-introduction rules for $\parr$ and & permute over each other, the desired common redex $\mathcal{M}_0$ is simply

$$\mathcal{M}\cup\{\Sigma:\Psi;\Gamma\vdash B, C, D, \Delta';\Upsilon,\quad \Sigma:\Psi;\Gamma\vdash B, C, E, \Delta';\Upsilon\}$$

Thus, local confluence is guaranteed by the permutation of inference rules. All other cases to consider can be proved similarly since we know that all right-introduction rules for the $\Downarrow\mathcal{L}_2$ connectives permute over each other (Exercise 6.27). □

The following propositions follows from the rewriting argument just given: the right-introduction phase can select one particular formula to decompose entirely before considering other formulas in the endsequent.

Proposition 6.29. *Consider the sequent $\Sigma : \Psi; \Gamma \vdash G, \Delta; \Upsilon$. There is a right-introduction phase with this endsequent such that the formula G is decomposed first. More specially, that right-introduction phase can be written as*

$$\dfrac{\left\{ \begin{array}{c} \Xi_i \\ \Sigma, \Sigma_i : \Psi, \Psi_i; \Gamma, \Gamma_i \vdash \mathcal{A}_i, \Delta; \Upsilon, \Upsilon_i \end{array} \right\}_{G \uparrow P_i}}{\Sigma : \Psi; \Gamma \vdash G, \Delta; \Upsilon}$$

where we assume that the path P_i is associated with the sequent $\Sigma_i : \Psi_i; \Gamma_i \vdash \mathcal{A}_i; \Upsilon_i$ and where Ξ_i is the right-introduction phase of the i^{th} premise.

As regards left-introduction phases, we note that every premise of a left-introduction rule with endsequent $\Sigma : \Psi; \Gamma \Downarrow B \vdash \mathcal{A}; \Upsilon$ is such that the first two zones and the last zone are identical to the corresponding zones in the endsequent: that is, these sequents are of the form $\Sigma : \Psi; \Gamma' \vdash \Delta'; \Upsilon$, for some multisets Γ' and Δ'. Thus, it is only the zones immediately adjacent to the $\vdash$ that vary during the construction of the left-introduction phase.

Proposition 6.30. *Let B be an $\mathcal{L}_2$ formula. The sequent $\Sigma : \Psi; \Gamma \Downarrow B \vdash \mathcal{A}; \Upsilon$ is the endsequent of a left-introduction phase with a multiset of premises $\mathcal{P}$ if and only if*

1. *there is a path P in B for which*

   $$\Sigma' : C_1, \ldots, C_n; B_1, \ldots, B_m \vdash A_1, \ldots, A_p; E_1, \ldots, E_q$$

   is the associated sequent;

2. *there is a substitution θ that maps the variables in Σ' to Σ-terms;*

3. *$\mathcal{A}$ is equal to the multiset union $\{A_1\theta, \ldots, A_p\theta\} \cup \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_m$;*

4. *Γ is the multiset union $\Gamma_1 \cup \cdots \cup \Gamma_m$; and*

5. *$\mathcal{P}$ is the multiset union of the following three multisets,*

$$\begin{aligned} \{\ '' \ : \ '' \ ; \cdot \vdash C_i\theta; \ '' \ \}_{i=1}^{n} &\cup \{\ '' \ : \ '' \ ; \Gamma_i \vdash B_i\theta, \mathcal{A}_i; \ '' \ \}_{i=1}^{m} \\ &\cup \{\ '' \ : \ '' \ ; E_i\theta \vdash \cdot; \ '' \ \}_{i=1}^{q}. \end{aligned}$$

Proof. This equivalence is proved by induction on the structure of the $\mathcal{L}_2$ formula B in a fashion similar to that given in Proposition 5.18. □

6.8.2 Admissibility of the general initial rule

We can now prove the admissibility of generalized initial rules for Forum formulas.

Theorem 6.31 (Initial admissibility)**.** *Let Ψ and Υ be multisets of $\mathcal{L}_2$ Σ-formulas. Let B be a $\mathcal{L}_2$ Σ-formulas. The following general forms of the* init *and* init? *rules are admissible in $\Downarrow\mathcal{L}_2$.*

1. *The sequent $\Sigma : \Psi; B \vdash B; \Upsilon$ is provable.*
2. *If B is a member of Ψ then $\Sigma : \Psi; \cdot \vdash B; \Upsilon$ is provable.*
3. *If B is a member of Υ then $\Sigma : \Psi; B \vdash \cdot; \Upsilon$ is provable.*
4. *If B is a member of both Ψ and Υ then $\Sigma : \Psi; \cdot \vdash \cdot; \Upsilon$ is provable.*

Proof. We describe how to build a $\Downarrow\mathcal{L}_2$-proof of $\Sigma : \Psi; B \vdash B; \Upsilon$ by induction on the structure of the formula B. We first consider the right-introduction phase with the endsequent $\Sigma : \Psi; B \vdash B; \Upsilon$. By Proposition 5.17, for every path P in B, there is a premise sequent of that right-introduction phase of the form $\Sigma, \Sigma' : \Psi, \Psi'; B, \Gamma' \vdash \mathcal{A}'; \Upsilon, \Upsilon'$, where $\Sigma' : \Psi'; \Gamma' \vdash \mathcal{A}'; \Upsilon'$ is the sequent associated to P. (The bound variables in Σ' are chosen to be disjoint from Σ.) In order to complete the proof of all of these premises, use the *decide* rule to select the occurrence of B in the left-bounded context. By Proposition 6.30, there is a left-introduction phase that corresponds to P. By setting θ to the identity substitution on the variables in Σ', we have $\mathcal{A} = \mathcal{A}'\theta$ and $\mathcal{A}_i$ is empty for $i = 1, \ldots, m$ and the sequents

$$\begin{array}{l}\{\Sigma, \Sigma' : \Psi, \Psi'; \ \cdot \ \vdash C_i; \Upsilon, \Upsilon'\}_{i=1}^{n} \cup \\ \{\Sigma, \Sigma' : \Psi, \Psi'; B_i \vdash B_i; \Upsilon, \Upsilon'\}_{i=1}^{m} \cup \\ \{\Sigma, \Sigma' : \Psi, \Psi'; E_i \vdash \ \cdot \ ; \Upsilon, \Upsilon'\}_{i=1}^{q}.\end{array}$$

must all be provable. The middle group of sequents are proved by the inductive assumption. The first group is proved by first using the *decide*! rule, choosing $C_i \in \Psi'$, and then applying the inductive assumption. Similarly, the third group is proved by first using the *decide*? rule, choosing $E_i \in \Upsilon'$, and then applying the inductive assumption.

The remaining three claims of this proposition are proved exactly the same way except that for the second claim, one uses the *decide*! rule instead of the *decide* rule and for the third and fourth claims, one uses the *decide*? rule first to reduce their provability to the previous two cases. □

Exercise 6.32. Prove that the following pairs of sequents are provable in the $\Downarrow\mathcal{L}_2$ proof system for all Σ-formulas B.

$$\frac{\Sigma : \Psi; \cdot \vdash B; \Upsilon \qquad \Sigma : \Psi, B; \Gamma \vdash \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon}\ \mathit{cut}\,!$$

$$\frac{\Sigma : \Psi; \Gamma \vdash \Delta; B, \Upsilon \qquad \Sigma : \Psi; B \vdash \cdot; \Upsilon}{\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon}\ \mathit{cut}\,?$$

$$\frac{\Sigma : \Psi; \Gamma_1 \vdash B, \Delta_1; \Upsilon \qquad \Sigma : \Psi; \Gamma_2, B \vdash \Delta_2; \Upsilon}{\Sigma : \Psi; \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2; \Upsilon}\ \mathit{cut}$$

Figure 6.12: The two exponential cut rules and the non-exponential cut rule. The syntactic variable Δ denotes a multiset of formulas.

1. $\Sigma : \cdot; (B \multimap \bot) \multimap \bot \vdash B; \cdot$ and $\Sigma : \cdot; B \vdash (B \multimap \bot) \multimap \bot; \cdot$.
2. $\Sigma : \cdot; (B \Rightarrow \bot) \multimap \bot \vdash B; \cdot$ and $\Sigma : B; \cdot \vdash (B \Rightarrow \bot) \multimap \bot; \cdot$.
3. $\Sigma : \cdot; ?\,B \vdash \cdot; B$ and $\Sigma : \cdot; B \vdash ?\,B; \cdot$

[Hint: Theorem 6.31 is needed to prove some of these. A couple other sequents require a bit more work to prove.]

6.8.3 Cut rules and cut elimination

We next turn our attention to proving the cut-admissibility theorem for $\Downarrow\mathcal{L}_2$-proofs. Figure 6.12 introduces three cut rules for the $\Downarrow\mathcal{L}_2$ proof system. The first two inference rules are the *exponential cut rules* (*cut*!, *cut*?) and the remaining inference rule is (the non-exponential) *cut* rule. The formula B is the *cut-formula* in each of these rules. In all of these cut inference rules, the bounded contexts are treated multiplicatively while the unbounded contexts are treated additively. We call the proof system that combines the inference rules in Figure 6.11 and Figure 6.12 the $\Downarrow^{+}\mathcal{L}_2$ proof system and proofs in that system will be called $\Downarrow^{+}\mathcal{L}_2$-proofs.

The *height* of a $\Downarrow^{+}\mathcal{L}_2$-proof Ξ is the maximum number of inference rules on a path in Ξ: this number is greater than or equal to 1. The following two propositions can be proved by simple inductions on the structure of $\Downarrow\mathcal{L}_2$-proofs.

Proposition 6.33 (Weakening $\Downarrow^{+}\mathcal{L}_2$-proofs). *If $\Sigma : \Psi; \Gamma \vdash \mathcal{A}; \Upsilon$ has a $\Downarrow^{+}\mathcal{L}_2$-proof of height h then $\Sigma, \Sigma' : \Psi, \Psi'; \Gamma \vdash \mathcal{A}; \Upsilon, \Upsilon'$ has a $\Downarrow^{+}\mathcal{L}_2$-proof of height h.*

Proposition 6.34 (Substitution into $\Downarrow^{+}\mathcal{L}_2$-proofs). *Let Σ be a signature, x be a variable not declared in Σ, τ be a primitive type, and t be a Σ-term of type τ. If $\Sigma, x : \tau : \Psi; \Gamma \vdash \mathcal{A}; \Upsilon$ has a $\Downarrow^{+}\mathcal{L}_2$-proof of height h then $\Sigma : \Psi[t/x]; \Gamma[t/x] \vdash \mathcal{A}[t/x]; \Upsilon[t/x]$ has a $\Downarrow^{+}\mathcal{L}_2$-proof of height h.*

The following lemma states that if a formula occurrence in the unbounded zones of a sequent is never decided on within a proof of that sequent, then that occurrence can be removed from its zone and the result will still be a proof of the same height.

Lemma 6.35 (Strengthening $\Downarrow^+\mathcal{L}_2$-proofs)**.** *Assume that we have a $\Downarrow^+\mathcal{L}_2$ proof of height h of either*

$$\Sigma : \Psi, B; \Gamma \vdash \Delta; \Upsilon \quad \textit{or} \quad \Sigma : \Psi, B; \Gamma \Downarrow D \vdash \Delta; \Upsilon$$

in which there is no occurrence of decide! *used with the formula B. Then there is a $\Downarrow^+\mathcal{L}_2$ proof of height h or less of either (respectively)*

$$\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon \quad \textit{or} \quad \Sigma : \Psi; \Gamma \Downarrow D \vdash \Delta; \Upsilon,$$

Similarly, assume that we have a $\Downarrow^+\mathcal{L}_2$ proof of height h of either

$$\Sigma : \Psi; \Gamma \vdash \Delta; B, \Upsilon \quad \textit{or} \quad \Sigma : \Psi; \Gamma \Downarrow D \vdash \Delta; B, \Upsilon$$

in which there is no occurrence of decide? *used with the formula B. Then there is a $\Downarrow^+\mathcal{L}_2$ proof of height h or less of either (respectively)*

$$\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon \quad \textit{or} \quad \Sigma : \Psi; \Gamma \Downarrow D \vdash \Delta; \Upsilon.$$

Proof. Add some discussion, especially for the case where the cut formula is the B in the left storage zone. In the *cut*! case, we have

$$\frac{\Sigma : \Psi; \cdot \vdash B; \Upsilon \qquad \Sigma : \Psi, B; \Gamma \vdash \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon}\ \textit{cut}!$$

By induction, the right premise yield directly a proof of the conclusion and the left premise can be discarded. In this case, however, the height has gotten smaller (the only time that can happen, I think). □

The following lemma allow us to replace an occurrence of *cut*? on B with possibly several occurrences of *cut* on B. The proof of this lemma is immediate.

Lemma 6.36 (Replacing *decide*? with *cut*)**.** *If the sequent $\Sigma : \Psi; B \vdash \cdot; \Upsilon$ has a $\Downarrow^+\mathcal{L}_2$-proof, say, Ξ, then every derivation of the form*

$$\frac{\overset{\Xi'}{\Sigma, \Sigma' : \Psi, \Psi'; \Gamma \vdash \mathcal{A}, B; B, \Upsilon, \Upsilon'}}{\Sigma, \Sigma' : \Psi, \Psi'; \Gamma \vdash \mathcal{A}; B, \Upsilon, \Upsilon'}\ \textit{decide}?,$$

where the variables bound in Σ' are not bound in Σ and where Ψ' and Υ' are multisets, can be converted to the derivation

$$\frac{\overset{\Xi'}{\Sigma, \Sigma' : \Psi, \Psi'; \Gamma \vdash \mathcal{A}, B; B, \Upsilon, \Upsilon'} \qquad \overset{\Xi''}{\Sigma, \Sigma' : \Psi, \Psi'; B \vdash \cdot; \Upsilon, \Upsilon'}}{\Sigma, \Sigma' : \Psi, \Psi'; \Gamma \vdash \mathcal{A}; B, \Upsilon, \Upsilon'}\ \textit{cut}.$$

Here, Ξ'' is the result of weakening Ξ using Proposition 6.33.

This lemma means that we are getting rid of all occurrences of *decide* ? rules applied to the B formula occurrence. There may still be many occurrence of the *decide* ? rule applied to other formulas since both Ξ and Ξ'' might have many such occurrences of that rule.

Lemma 6.37 (Replacing *cut*? with *cut*)**.** *Let Ξ be a $\Downarrow^+\mathcal{L}_2$-proof. This proof can be transformed into a proof of the same sequent that does not contain any occurrences of the cut? rule.*

Proof. We do a simple, double induction. The outer induction involves the number of occurrences of *cut*? rule in Ξ. If there is such a cut rule, take one that is of minimal height. Now the inner induction transforms that exponential cut into a non-exponential cut as follows. Consider the following occurrence of the *cut*? rule.

$$\dfrac{\overset{\Xi_1}{\Sigma : \Psi; \Gamma \vdash \Delta; B, \Upsilon} \qquad \overset{\Xi_2}{\Sigma : \Psi; B \vdash \cdot; \Upsilon}}{\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon} \; cut?$$

By repeatedly applying Lemma 6.36, all occurrences of the *decide* ? rule on B in Ξ_1 can be replaced by applications of *cut*. This yields a proof of $\Sigma : \Psi; \Gamma \vdash \Delta; B, \Upsilon$ in which no applications of *decide* ? are applied to B. By Lemma 6.35, we have a $\Downarrow^+\mathcal{L}_2$ proof of $\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon$. Thus, we have replaced the above occurrence of *cut*? on B with possibly several instances of *cut* on B. Note that the height of the resulting proof is smaller than the height of the original proof. □

At this point in proving the cut-elimination theorem for $\Downarrow\mathcal{L}_2$-proofs, we introduce a second cut-like rule, called the *key cut* (compare this rule to the rule by the same name in Section 5.5).

$$\dfrac{\Sigma : \Psi; \Gamma_1 \vdash B, \Delta; \Upsilon \quad \Sigma : \Psi; \Gamma_2 \Downarrow B \vdash \mathcal{A}; \Upsilon}{\Sigma : \Psi; \Gamma_1, \Gamma_2 \vdash \Delta, \mathcal{A}; \Upsilon} \; cut_k$$

When there is an occurrence of the key cut on a non-atomic formula B, we know that the right-introduction phase that has the left premise as its endsequent and the left introduction phase that has the right premise as its endsequent both decompose B. We generalize the definition of the height of a proof to also include this inference rule. We will now show (i) how to replace occurrences of *cut* and *cut*! on the cut formula B with occurrences of cut_k on B, and (ii) how to replace cut_k on B with instances of *cut* on strict subformulas of B. Furthermore, we say that a proof is *cut-free* if it has no occurrences of any of the three cut rules in Figure 6.12 as well as cut_k. Obviously, a $\Downarrow^+\mathcal{L}_2$-proof that has no occurrences of a cut rule is a $\Downarrow\mathcal{L}_2$-proof.

Lemma 6.38 (Replace *cut* ! with *cut*$_k$)**.** *Consider the following occurrence of the* cut ! *rule*

$$\dfrac{\overset{\Xi_l}{\Sigma : \Psi; \cdot \vdash B; \Upsilon} \qquad \overset{\Xi_r}{\Sigma : \Psi, B; \Gamma \vdash \Delta; \Upsilon}}{\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon}\ \mathit{cut}\,!,$$

where Ξ_l and Ξ_r are cut-free proofs. We can replace this occurrence of cut ! *on B with a proof of the same endsequent that may contain possibly many occurrences of* cut$_k$ *on B.*

Proof. Consider a subderivation in Ξ_r of the form

$$\dfrac{\overset{\Xi_0}{\Sigma, \Sigma' : \Psi, \Psi', B; \Gamma \Downarrow B \vdash \mathcal{A}; \Upsilon, \Upsilon'}}{\Sigma, \Sigma' : \Psi, \Psi', B; \Gamma \vdash \mathcal{A}; \Upsilon, \Upsilon'}\ \mathit{decide}\,!,$$

where the variables bound in Σ' are not bound in Σ and where Ψ' and Υ' are multisets. This inference rule can be converted to the derivation

$$\dfrac{\overset{\Xi_l'}{\Sigma' : \Psi'; \cdot \vdash B; \Upsilon'} \qquad \overset{\Xi_0}{\Sigma' : \Psi', B; \Gamma \Downarrow B \vdash \mathcal{A}; \Upsilon'}}{\Sigma' : \Psi', B; \Gamma \vdash \mathcal{A}; \Upsilon'}\ \mathit{cut}_k.$$

Here, Ξ_l' is the result of weakening Ξ_l using Proposition 6.33. We can thus removed all occurrences of *decide* ! on B in Ξ_r to obtain the proof Ξ_r' of $\Sigma : \Psi, B; \Gamma \vdash \Delta; \Upsilon$. Using Proposition 6.35, we can strengthen Ξ_r' to get a proof of $\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon$ in which we have replaced one occurrence of *cut* ! with possibly many occurrences of *cut*$_k$. □

Lemma 6.39 (Replace *cut* with *cut*$_k$)**.** *Consider the following occurrence of the* cut *rule*

$$\dfrac{\overset{\Xi_l}{\Sigma : \Psi; \Gamma_1 \vdash B, \Delta_1; \Upsilon} \qquad \overset{\Xi_r}{\Sigma : \Psi; \Gamma_2, B \vdash \Delta_2; \Upsilon}}{\Sigma : \Psi; \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2; \Upsilon}\ \mathit{cut},$$

where Ξ_l and Ξ_r are cut-free proofs. We can replace this occurrence of cut *on B with possibly many occurrences of* cut$_k$ *on B.*

Proof. We proceed by induction on the structure of Ξ_r. If the endsequent of Ξ_r is not a border sequent, then Ξ_r ends with a right-introduction phase. This instance of *cut* can be permuted up through that entire right-introduction phase, leaving instances of *cut* with only border sequents. Since all of these occurrences of *cut* have shorter proofs of their rightmost premise, the inductive assumption can be applied.

Assume instead that the endsequent of Ξ_r is a border sequent: hence, the last inference rule of Ξ_r is an occurrence of either *decide*, *decide* !, or *decide* ?.

Assume the case that the first of these three choices is made. If that occurrence of *decide* selects B, then Ξ_r has the form

$$\dfrac{\begin{array}{c}\Xi_r'\\ \Sigma : \Psi; \Gamma_2 \Downarrow B \vdash \Delta_2; \Upsilon\end{array}}{\Sigma : \Psi; \Gamma_2, B \vdash \Delta_2; \Upsilon}\ \mathit{decide}.$$

In this case, the *cut* rule above can be changed directly to the following

$$\dfrac{\begin{array}{c}\Xi_l\\ \Sigma : \Psi; \Gamma_1 \vdash B, \Delta_1; \Upsilon\end{array} \qquad \begin{array}{c}\Xi_r'\\ \Sigma : \Psi; \Gamma_2 \Downarrow B \vdash \Delta_2; \Upsilon\end{array}}{\Sigma : \Psi; \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2; \Upsilon}\ \mathit{cut}_k.$$

The other case we need to consider is when the last inference rule of Ξ_r is an instance of the *decide* rule on a formula occurring in Γ_2: that is, Ξ_r has the form

$$\dfrac{\begin{array}{c}\Xi_r'\\ \Sigma : \Psi; \Gamma_3, B \Downarrow F \vdash \Delta_2; \Upsilon\end{array}}{\Sigma : \Psi; \Gamma_3, F, B \vdash \Delta_2; \Upsilon}\ \mathit{decide},$$

where Γ_2 decomposes to $\Gamma_3 \cup \{F\}$ and where Δ_2 contains only atomic formulas. By Proposition 6.30, since the sequent $\Sigma : \Psi; \Gamma_3, B \Downarrow F \vdash \Delta_2; \Upsilon$ is the endsequent of a left-introduction phase with a multiset of premises $\mathcal{P}$ there is a path P in F for which

$$\Sigma' : C_1, \ldots, C_n; B_1, \ldots, B_m \vdash A_1, \ldots, A_p; E_1, \ldots, E_q$$

is the associated sequent; there is a substitution θ that maps the variables in Σ' to Σ-terms; Δ_2 is equal to the multiset union $\{A_1\theta, \ldots, A_p\theta\} \cup \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_m$; $\Gamma_3 \cup \{B\}$ is the multiset union $\hat{\Gamma}_1 \cup \cdots \cup \hat{\Gamma}_m$; and $\mathcal{P}$ is the following multiset union of three multisets,

$$\begin{array}{l}\{\text{"}\ \ : \text{"}\ \ ; \cdot \vdash C_i\theta; \text{"}\ \ \}_{i=1}^{n} \cup \{\text{"}\ \ : \text{"}\ \ ; \hat{\Gamma}_i \vdash B_i\theta, \mathcal{A}_i; \text{"}\ \ \}_{i=1}^{m}\\ \qquad\qquad\qquad\qquad \cup \{\text{"}\ \ : \text{"}\ \ ; E_i\theta \vdash \cdot; \text{"}\ \ \}_{i=1}^{q}.\end{array}$$

The formula B occurs in at least one of the multisets $\hat{\Gamma}_1, \ldots, \hat{\Gamma}_m$: without loss of generality, we can assume that $\hat{\Gamma}_1$ is equal to $\hat{\Gamma}_1' \cup \{B\}$. We can now build the same left-introduction phase from these premises except that the one that corresponds to $\Sigma : \Psi; \hat{\Gamma}_1', B \vdash B_1\theta, \mathcal{A}_1; \Upsilon$ is replaced by

$$\dfrac{\Sigma : \Psi; \Gamma_1 \vdash B, \Delta_1; \Upsilon \quad \Sigma : \Psi; \hat{\Gamma}_1', B \vdash B_1\theta, \mathcal{A}_1; \Upsilon}{\Sigma : \Psi; \Gamma_1, \hat{\Gamma}_1' \vdash \Delta_1, B_1\theta, \mathcal{A}_1; \Upsilon}\ \mathit{cut}.$$

When this left-introduction phase is assembled, the result is a proof of $\Sigma : \Psi; \Gamma_3, \Gamma_1 \Downarrow F \vdash \Delta_1, \Delta_2; \Upsilon$. By applying the *decide* rule and remembering that $\Gamma_3 \cup \{F\}$ is Γ_2, we now have a proof of $\Sigma : \Psi; \Gamma_2, \Gamma_1 \vdash \Delta_1, \Delta_2; \Upsilon$ in which the height of the *cut* has been reduced.

The remaining cases to consider is then the last inference rule of Ξ_r is either *decide*! or *decide*?. If that rule is *decide*? then Ξ_r ends in a right-introduction phase and, as we have argued above, the *cut* rule can be permuted up through this phase. If that rule is *decide*! then Ξ_r has the form

$$\dfrac{\begin{array}{c}\Xi_r'\\ \Sigma : \Psi', C; \Gamma_2, B \Downarrow C \vdash \Delta_2; \Upsilon\end{array}}{\Sigma : \Psi', C; \Gamma_2, B \vdash \Delta_2; \Upsilon}\ \textit{decide}!.$$

where Γ_2 can be written as $\Psi' \cup \{C\}$. It is also the case that the cut rule can be permuted up through the resulting left-introduction phase in Ξ_r. ☐

Lemma 6.40 (Replacing $\textit{cut}_k$ with other cuts)**.** *Consider an occurrence of the* $\textit{cut}_k$ *rule of the form*

$$\dfrac{\begin{array}{c}\Xi_l\\ \Sigma : \Psi; \Gamma_1 \vdash B, \Delta; \Upsilon\end{array} \quad \begin{array}{c}\Xi_r\\ \Sigma : \Psi; \Gamma_2 \Downarrow B \vdash \mathcal{A}; \Upsilon\end{array}}{\Sigma : \Psi; \Gamma_1, \Gamma_2 \vdash \Delta, \mathcal{A}; \Upsilon}\ \textit{cut}_k,$$

where Ξ_l *and* Ξ_r *are (cut-free)* $\Downarrow\mathcal{L}_2$*-proofs. We can transform this proof into a proof of the same endsequent in which there are no occurrences of* $\textit{cut}_k$ *and the only occurrences of the cut, cut!, and cut? rules have cut-formulas that are strictly smaller than* B*.*

Proof. Consider the instance of the $\textit{cut}_k$ rule given in the assumptions of this lemma. If B is atomic, then $\mathcal{A}$ is the multiset containing exactly B and the result of eliminating $\textit{cut}_k$ is Ξ_l.

Now assume that B is not atomic. Thus, Ξ_l ends in a right-introduction phase and Ξ_r ends in a left-introduction phase. By Proposition 6.30, there is a path P in B that has the associated sequent representation

$$\mathcal{X} : C_1, \ldots, C_n; B_1, \ldots, B_m \vdash A_1, \ldots, A_p; E_1, \ldots, E_q$$

and there is a substitution θ that maps the variables in $\mathcal{X}$ to Σ-terms such that $\mathcal{A}'$ is the multiset union $\{A_1\theta, \ldots, A_p\theta\} \cup \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_m$, Γ is the multiset union $\Gamma_1 \cup \cdots \cup \Gamma_m$, and this phase has $n + m + q$ premises

$$\begin{array}{l}\{\text{"} : \text{"} ; \cdot \vdash C_i\theta; \text{"} \}_{i=1}^{n} \cup \{\text{"} : \text{"} ; \Gamma_i \vdash B_i\theta, \mathcal{A}_i; \text{"} \}_{i=1}^{m}\\ \qquad\qquad\qquad\qquad \cup \{\text{"} : \text{"} ; E_i\theta \vdash \cdot; \text{"} \}_{i=1}^{q}.\end{array}$$

By Proposition 6.29, Ξ_l ends with a right-introduction phase that contains a premise of the form

$$\begin{array}{c}\Xi_0\\ \vdots\\ \Sigma, \mathcal{X} : \Psi, C_1, \ldots, C_n; \Gamma, B_1, \ldots, B_m \vdash \mathcal{A}, A_1, \ldots, A_p; E_1, \ldots, E_q, \Upsilon\end{array}$$

By repeated application of Proposition 6.34, we know that the sequent

$$\Sigma, \mathcal{X} : \Psi, C_1\theta, \ldots, C_n\theta; \Gamma, B_1\theta, \ldots, \overset{\Xi'_0}{B_m\theta} \vdash \mathcal{A}, A_1\theta, \ldots, A_p\theta; E_1\theta, \ldots, E_q\theta, \Upsilon$$

has a $\Downarrow^{+}\mathcal{L}_2$ proof. We can take Ξ'_0 and use *cut*, *cut*!, and *cut*? with the proofs of the $n + m + q$ premises above to yield a proof with $n + m + q$ occurrences of these cut rules to provide a proof without occurrences of cut_k of the endsequent $\Sigma : \Psi; \Gamma, \Gamma' \vdash \Delta, \mathcal{A}; \Upsilon$. Note that the size of each of the cut formulas $C_1\theta, \ldots, C_n\theta, \Gamma, B_1\theta, \ldots, B_m\theta, E_1\theta, \ldots, E_q\theta$ are strictly smaller than the size of the original cut formula B. $\square$

Lemma 6.41. *An occurrence of either the cut or cut! rule with premises proved by cut-free proofs can be eliminated to yield a cut-free proof of the same sequent.*

Proof. Consider an occurrence of the *cut* inference rule

$$\frac{\Sigma : \Psi; \Gamma_1 \vdash B, \Delta_1; \Upsilon \quad \Sigma : \Psi; \Gamma_2, B \vdash \Delta_2; \Upsilon}{\Sigma : \Psi; \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2; \Upsilon}\ cut,$$

where the premises have cut-free $\Downarrow\mathcal{L}_2$-proofs. By applying Lemma 6.39, there is a proof Ξ of $\Sigma : \Psi; \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2; \Upsilon$ that contains no occurrences of *cut* but it might have several instances of the cut_k rule applied to the B formula. Similarly, consider an occurrence of the *cut*! inference rule

$$\frac{\Sigma : \Psi; \cdot \vdash B; \Upsilon \quad \Sigma : \Psi, B; \Gamma \vdash \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon}\ cut!,$$

where the premises have cut-free $\Downarrow\mathcal{L}_2$-proofs. By applying Lemma 6.38, there is a proof Ξ of $\Sigma : \Psi; \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2; \Upsilon$ that contains no occurrences of *cut*! but it might have several instances of the cut_k rule applied to the B formula. Thus, in either case, the proof Ξ contains no occurrences of *cut* or *cut*! while it may contain several occurrences of cut_k.

We now proceed by induction on the structure of the formula B. Assume that B is an atomic formula. The occurrences of cut_k can be eliminated by repeatedly replacing an upper occurrence of cut_k with its left premise. On the other hand, assume that B is not atomic. We can now do a second induction on the number of occurrence of cut_k in Ξ. If that number is 0 then the proof Ξ is the desired cut-free proof. Otherwise, there exists at least one occurrence of cut_k on B. If we pick an upper-most occurrence of cut_k and apply Lemma 6.40, we can convert that occurrence of cut_k to several occurrences of *cut*, *cut*!, and *cut*? on strictly smaller formulas than B. By applying Lemma 6.37, this proof can be converted to a proof without occurrences of the *cut*? rule. By applying Lemma 6.40, there is a proof of the same endsequent where the occurrences of *cut* and *cut*! are on strictly smaller formulas than B. By applying the inductive assumption, all of these occurrences of *cut* can be eliminated. We have now reduced the number of cut_k inference rules and, hence, we have completed our proof by the outer induction. □

We can bring these lemmas together to prove the main cut-elimination theorem for $\Downarrow^+\mathcal{L}_2$ proofs.

Theorem 6.42 (Elimination of cuts)**.** *If a sequent has a $\Downarrow^+\mathcal{L}_2$-proof then it has a (cut-free) $\Downarrow\mathcal{L}_2$-proof.*

Proof. Take a $\Downarrow^+\mathcal{L}_2$-proof of a sequent, say, $\mathbf{S}$. By applying Lemma 6.37, we can assume that all occurrences of *cut*? have been replaced. Thus, let Ξ be a proof of $\mathbf{S}$ that may contain occurrences of *cut* and *cut*!.

Our proof proceeds by a simple induction on the number of occurrences of *cut* and *cut*! inference rules in a proof. In particular, we first take an occurrence of a *cut* or *cut*! rule which is the endsequent of a subproof of minimal height: by Lemma 6.41, such a subproof has cut-free proofs of its conclusion. Thus, we have eliminated one occurrence of the *cut* or *cut*! rules and, hence, by the inductive argument, we can eliminate all cut rules. □

At the end of Section 6.1, we described an interaction between the rules of contraction and the cut rule in *LK* that would allow cut elimination to produce completely unrelated proofs of a given endsequent. In that example, the cut formula was weakened on both the left and right side of the premises of the cut rule. In the focused proof system $\Downarrow^+\mathcal{L}_2$, such a situation cannot happen. For example, consider the *cut*! inference rule.

$$\frac{\Sigma : \Psi; \cdot \vdash B; \Upsilon \qquad \Sigma : \Psi, B; \Gamma \vdash \Delta; \Upsilon}{\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon}\ cut!$$

The occurrence of the cut-formula B in the left premise cannot be weakened since it will be the subject of a right-introduction rule. The occurrence of B in the right premise can, however, be weakened (by an application of an initial rule). A similar statement holds for the *cut*? rule while for the *cut* rule, the occurrences of the cut formula in the premises cannot be weakened in either premise. As a result, the kind of problem arising from weakening and cut that can appear in *LK* is avoided in $\Downarrow^+\mathcal{L}_2$.

6.8.4 Soundness and completeness of the focused proof system

We now wish to show that the $\Downarrow\mathcal{L}_2$ proof system is not just some contrived proof system but that it can prove all the same theorems that the $\mathbf{L}$ proof system can prove. We would also like to go one more step and show that some of the proof theory of $\mathbf{L}$ can be inferred from the proof theory of $\Downarrow\mathcal{L}_2$. Since these two proof systems use different sets of logical connectives, we must first define a mapping from formulas used in the **L** proof system into $\mathcal{L}_2$-formulas.

Recall that the negatively polarized logical connectives of **L** are $\bot$, $\top$, $⅋$, $\&$, and $\forall$ while the positively polarized logical connectives are $\mathbf{1}$, $\mathbf{0}$, $\otimes$, $\oplus$, and $\exists$. We consider a formula that is a top-level negation as being neither positively or negatively polarized: one does not know the intended polarity of a negated formula until one considers the formula that is negated.

We define two functions, namely, $(\cdot)^\triangledown$ that maps **L** formulas into $\mathcal{L}_2$ formulas and $(\cdot)^\blacktriangledown$ that maps those formulas with a positively polarized top-level logical connective into $\mathcal{L}_2$ formulas. If A is an atomic formula, then $A^\triangledown = A$. These functions are defined for other formulas as follows.

$$\begin{aligned}
\top^\triangledown &= \top & \mathbf{0}^\blacktriangledown &= \top\\
\bot^\triangledown &= \bot & \mathbf{1}^\blacktriangledown &= \bot\\
(B ⅋ C)^\triangledown &= B^\triangledown ⅋ C^\triangledown & (B \otimes C)^\blacktriangledown &= B^\triangledown \multimap C^\triangledown \multimap \bot\\
(B \& C)^\triangledown &= B^\triangledown \& C^\triangledown & (B \oplus C)^\blacktriangledown &= (B^\triangledown \multimap \bot) \& (C^\triangledown \multimap \bot)\\
(\forall x.B)^\triangledown &= \forall x.(B)^\triangledown & (\exists x.B)^\blacktriangledown &= \forall x.(B^\triangledown \multimap \bot)\\
(?\,B)^\triangledown &= ?(B^\triangledown) & (!\,B)^\blacktriangledown &= (B^\triangledown) \Rightarrow \bot
\end{aligned}$$

For formulas P with a positively polarized top-level logical connective, set $(P)^\triangledown = (P)^\blacktriangledown \multimap \bot$. If the top-level connective is negation, then $(B^\perp)^\triangledown = B^\triangledown \multimap \bot$. If Γ is a multiset of **L** formulas then we write $\Gamma^\triangledown$ to denote the multiset of $\mathcal{L}_2$ formulas $\{B^\triangledown \mid B \in \Gamma\}$: assume a similar definition for $\Gamma^\blacktriangledown$ whenever all formulas in Γ have a positive polarity connective as their top-level connective.

For convenience, we use the notation $\Sigma : \Psi; \Gamma \vdash_{\Downarrow} \Delta; \Upsilon$ to denote the proposition that the sequent $\Sigma : \Psi; \Gamma \vdash_{\Downarrow} \Delta; \Upsilon$ has a $\Downarrow\mathcal{L}_2$-proof.

As one expects, the following soundness property for the $(\cdot)^\triangledown$ translation has a straightforward proof, even if there are many simple cases to consider.

Proposition 6.43 (Soundness of $\Downarrow\mathcal{L}_2$-proofs). *Let Γ and Δ be Σ-formulas in linear logic such that $\Sigma : \cdot; \Gamma^\triangledown \vdash \Delta^\triangledown; \cdot$ has a (cut-free) $\Downarrow\mathcal{L}_2$-proof. Then $\Sigma : \Gamma \vdash \Delta$ has a cut-free proof in **L**.*

Proof. We prove the following strengthening of this proposition. Let Θ be a multiset of Σ-formulas all of which have a top-level positive connective and let Γ, Δ, Ψ, and Υ be multisets of Σ-formulas in linear logic.

1. If $\Sigma : \Psi^\triangledown; \Gamma^\triangledown, \Theta^\blacktriangledown \vdash \Delta^\triangledown; \Upsilon^\triangledown$ has a $\Downarrow\mathcal{L}_2$-proof then $\Sigma : !\,\Psi, \Gamma \vdash \Theta, \Delta, ?\,\Upsilon$ has a cut-free proof in **L**.

2. If B is an **L** Σ-formula and $\Sigma : \Psi^\triangledown; \Gamma^\triangledown, \Theta^\blacktriangledown \Downarrow B^\triangledown \vdash \Delta^\triangledown; \Upsilon^\triangledown$ has a $\Downarrow\mathcal{L}_2$-proof then $\Sigma : !\,\Psi, \Gamma, B \vdash \Theta, \Delta, ?\,\Upsilon$ has a cut-free proof in **L**.

3. If B is an **L** Σ-formula with a top-level positive connective and $\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \Downarrow B^{\blacktriangledown} \vdash \Delta^{\triangledown}; \Upsilon^{\triangledown}$ has a $\Downarrow\mathcal{L}_2$-proof then $\Sigma : !\,\Psi, \Gamma \vdash B, \Theta, \Delta, ?\,\Upsilon$ has a cut-free proof in **L**.

We shall also assume that we only consider $\Downarrow\mathcal{L}_2$-proofs that satisfy the following invariant: every sequent in a $\Downarrow\mathcal{L}_2$-proof that has an occurrence of $\bot$ in the right-linear context is the conclusion of the $\bot$R inference rule. Given that all right-introduction rules permute over each other, this restriction on proofs is easily satisfied.

We proceed by mutual induction on the structure of $\Downarrow\mathcal{L}_2$-proofs of these three kind of sequents. First, let Ξ be $\Downarrow\mathcal{L}_2$-proof of $\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \vdash \Delta^{\triangledown}; \Upsilon^{\triangledown}$. The last inference rule in Ξ is either a right-introduction rule or one of the three decide rules. We consider the following cases.

- Assume that this last inference rule introduced a negative polarity **L** connective. For example, if that rule is $\invamp R$ then Δ can be written as $B \invamp C, \Delta'$ and that last inference rule is of the form

$$\frac{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \vdash B^{\triangledown}, C^{\triangledown}, \Delta^{\triangledown}; \Upsilon^{\triangledown}}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \vdash (B \invamp C)^{\triangledown}, \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \invamp R$$

  By the inductive hypothesis, $\Sigma : !\,\Psi, \Gamma \vdash B, C, \Theta, \Delta, ?\,\Upsilon$ has an **L** proof and, by the $\invamp R$ rule in **L**, we have an **L** proof of $\Sigma : !\,\Psi, \Gamma \vdash B \invamp C, \Theta, \Delta, ?\,\Upsilon$. The remaining negative polarity connectives are handled in such a simple and direct fashion.

- Assume that the last inference rule of Ξ is $\multimap R$. (Notice that $\Rightarrow R$ is not possible here.) Thus, Δ can be written as B, Δ' where B is either a negation or a top-level positive polarity connective. In the first case, write B as $C^{\bot}$ and the last two inference rules in Ξ are

$$\frac{\dfrac{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, C^{\triangledown}, \Theta^{\blacktriangledown} \vdash \Delta^{\triangledown}; \Upsilon^{\triangledown}}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, C^{\triangledown}, \Theta^{\blacktriangledown} \vdash \bot, \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \bot\text{R}}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \vdash C^{\triangledown} \multimap \bot, \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \multimap\text{R}$$

  By the inductive hypothesis, $\Sigma : !\,\Psi, \Gamma, C \vdash \Theta, \Delta, ?\,\Upsilon$ has an **L** proof and, by the $(\cdot)^{\bot}R$ rule in **L**, we have an **L** proof of $\Sigma : !\,\Psi, \Gamma \vdash C^{\bot}, \Theta, \Delta, ?\,\Upsilon$. The other case to consider is when B is a top-level positive polarity connective, in which case, the last two inference rules of Ξ are

$$\frac{\dfrac{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, B^{\blacktriangledown}, \Theta^{\blacktriangledown} \vdash \Delta^{\triangledown}; \Upsilon^{\triangledown}}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, B^{\blacktriangledown}, \Theta^{\blacktriangledown} \vdash \bot, \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \bot\text{R}}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \vdash B^{\blacktriangledown} \multimap \bot, \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \multimap R$$

  By the inductive hypothesis, $\Sigma : !\,\Psi, \Gamma \vdash B, \Theta, \Delta, ?\,\Upsilon$ has an **L** proof, which also serves as the desired proof for this case.

- Assume that the last inference rule of Ξ is one of the decide rules. In the case of the *decide* ? inference rule, that rule translates directly to the uses of the contraction and dereliction rules ($?\,C$ and $?\,D$) for ?. In the case of the *decide* rule, the desired **L** proof follows immediate from the mutual inductive hypothesis. Finally, in the case of the *decide* ! rule, the desired **L** proof follows from the mutual inductive hypothesis as well as the contraction and dereliction rules ($!\,C$ and $!\,D$) for !.

Now consider the second mutually inductive statement. Assume that Ξ is a $\Downarrow\mathcal{L}_2$-proof of $\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \Downarrow B^{\triangledown} \vdash \Delta^{\triangledown}; \Upsilon^{\triangledown}$. Again, there are three cases to consider for B. If B has a top-level negative polarity logical connective then the corresponding inference rule to use with the inductive assumption is the **L** left introduction rule for that connective. If B is the negation $C^{\perp}$, then the last two inference rules of Ξ are

$$\dfrac{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \vdash C^{\triangledown}, \Delta^{\triangledown}; \Upsilon^{\triangledown} \qquad \dfrac{}{\Sigma : \Psi^{\triangledown}; \Downarrow \perp \vdash; \Upsilon^{\triangledown}}\ \perp\mathrm{L}}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \Downarrow C^{\triangledown} \supset \perp \vdash \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \multimap L$$

By the inductive assumption, $\Sigma : !\,\Psi, \Gamma \vdash C, \Theta, \Delta, ?\,\Upsilon$ has a cut-free proof in **L**. The desired final proof is built using the $(\cdot)^{\perp}L$ rule. The final case to consider for B is when it has a top-level positive logical connective. In this case, Ξ is of the form

$$\dfrac{\begin{matrix}\Xi' \\ \Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \vdash B^{\blacktriangledown}, \Delta^{\triangledown}; \Upsilon^{\triangledown}\end{matrix} \qquad \dfrac{}{\Sigma : \Psi^{\triangledown}; \Downarrow \perp \vdash; \Upsilon^{\triangledown}}\ \perp\mathrm{L}}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \Downarrow B^{\blacktriangledown} \supset \perp \vdash \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \multimap L$$

It is here that the definition of $(\cdot)^{\blacktriangledown}$ matters. We illustrate this with B being $B_1 \otimes B_2$ (the other cases are similar). In this case, Ξ' must be of the form

$$\dfrac{\dfrac{\dfrac{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, B_1^{\triangledown}, B_2^{\triangledown}, \Theta^{\blacktriangledown} \vdash \Delta^{\triangledown}; \Upsilon^{\triangledown}}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, B_1^{\triangledown}, B_2^{\triangledown}, \Theta^{\blacktriangledown} \vdash \perp, \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \perp\mathrm{L}}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, B_1^{\triangledown}, \Theta^{\blacktriangledown} \vdash B_2^{\triangledown} \multimap \perp, \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \multimap L}{\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \vdash B_1^{\triangledown} \multimap B_2^{\triangledown} \multimap \perp, \Delta^{\triangledown}; \Upsilon^{\triangledown}}\ \multimap L$$

By the inductive hypothesis, we know that the sequent $\Sigma : !\,\Psi, \Gamma, B_1, B_2 \vdash \Theta, \Delta, ?\,\Upsilon$ has a cut-free **L** proof. The desired **L** proof for this case follows from applying the $\otimes$L rule of **L**.

Now consider the third and final mutually inductive statement. Assume that Ξ is a $\Downarrow\mathcal{L}_2$-proof of $\Sigma : \Psi^{\triangledown}; \Gamma^{\triangledown}, \Theta^{\blacktriangledown} \Downarrow B^{\blacktriangledown} \vdash \Delta^{\triangledown}; \Upsilon^{\triangledown}$. Again, the definition of $(\cdot)^{\blacktriangledown}$ matters and we illustrate it for $\otimes$: the other cases are done similarly. Let B be $B_1 \otimes B_2$. Thus, Ξ be of the form

$$\dfrac{\Psi^{\triangledown}; \Gamma_1^{\triangledown}, \Theta_1^{\blacktriangledown} \vdash B_1^{\triangledown}, \Delta_1^{\triangledown}; \Upsilon^{\triangledown} \qquad \dfrac{\Psi^{\triangledown}; \Gamma_2^{\triangledown}, \Theta_2^{\blacktriangledown} \vdash B_2^{\triangledown}, \Delta_2^{\triangledown}; \Upsilon^{\triangledown} \qquad \dfrac{}{\Psi^{\triangledown}; \cdot \Downarrow \perp \vdash \cdot; \Upsilon^{\triangledown}}}{\Psi^{\triangledown}; \Gamma_2^{\triangledown}, \Theta_2^{\blacktriangledown} \Downarrow B_2^{\triangledown} \multimap \perp \vdash \Delta_2^{\triangledown}; \Upsilon^{\triangledown}}}{\Psi^{\triangledown}; \Gamma_1^{\triangledown}, \Gamma_2^{\triangledown}, \Theta_1^{\blacktriangledown}, \Theta_2^{\blacktriangledown} \Downarrow B_1^{\triangledown} \multimap B_2^{\triangledown} \multimap \perp \vdash \Delta_1^{\triangledown}, \Delta_2^{\triangledown}; \Upsilon^{\triangledown}}$$

where Γ, Δ, and Θ are split into their respective pairs of multisets (the signature binder is dropped for readability). By the inductive hypothesis, there are cut-free **L** proofs for $\Sigma : !\,\Psi, \Gamma_1 \vdash B_1, \Theta_1, \Delta_1, ?\,\Upsilon$ and $\Sigma : !\,\Psi, \Gamma_2 \vdash B_2, \Theta_2, \Delta_2, ?\,\Upsilon$. The $\otimes$R rule of **L** provides the final, desired **L** proof of $\Sigma : !\,\Psi, \Gamma_2 \vdash B_1 \otimes B_2, \Theta_2, \Delta_2, ?\,\Upsilon$. □

Recalling from Section 6.1, an inference rule is invertible if whenever its conclusion is provable, its premises are provable. We state an inversion lemma for $\Downarrow\mathcal{L}_2$-proofs.

Lemma 6.44. *All the right-introduction rules of $\Downarrow\mathcal{L}_2$ are invertible. Furthermore, the following equivalences hold.*

$$\Sigma : \Psi; \Gamma, (B \Rightarrow \bot) \multimap \bot \vdash_{\Downarrow} \Delta; \Upsilon \quad \textit{if and only if} \quad \Sigma : \Psi, B; \Gamma \vdash_{\Downarrow} \Delta; \Upsilon.$$

$$\Sigma : \Psi; \Gamma \vdash_{\Downarrow} ?\,B, \Delta; \Upsilon \quad \textit{if and only if} \quad \Sigma : \Psi; \Gamma \vdash_{\Downarrow} \Delta; \Upsilon, B.$$

Proof. The proofs that the eight right rules are invertible all follow the same pattern (see Exercise 6.9). We illustrate that pattern with two examples. Consider the $?\,R$ rule. Assume that $\Sigma : \Psi; \Gamma \vdash_{\Downarrow} \Delta, ?\,B; \Upsilon$. Since the sequent $\Sigma : \cdot; ?\,B \vdash \cdot; B$ has a $\Downarrow\mathcal{L}_2$-proof, then the *cut* rule and cut elimination theorem yields a $\Downarrow\mathcal{L}_2$-proof of $\Sigma : \Psi; \Gamma \vdash_{\Downarrow} \Delta; B, \Upsilon$. For a second example, consider the $\&R$ rule. Assume that $\Sigma : \Psi; \Gamma \vdash_{\Downarrow} \Delta, B_1 \mathbin{\&} B_2; \Upsilon$. Since the sequents $\Sigma : \cdot; B_1 \mathbin{\&} B_2 \vdash B_i; \cdot$ have $\Downarrow\mathcal{L}_2$-proofs (for $i = 1$ and $i = 2$), then the cut rule and cut elimination theorem yields $\Downarrow\mathcal{L}_2$-proofs of $\Sigma : \Psi; \Gamma \vdash \Delta; B_1, \Upsilon$ and $\Sigma : \Psi; \Gamma \vdash \Delta; B_2, \Upsilon$.

Now consider the first equivalence. If we assume that $\Sigma : \Psi; \Gamma, (B \Rightarrow \bot) \multimap \bot \vdash_{\Downarrow} \Delta; \Upsilon$ then, using the *cut* rule with a proof of $\Sigma : B; \cdot \vdash (B \Rightarrow \bot) \multimap \bot; \cdot$ (see also Exercise 6.32), we have (after apply cut-elimination) a $\Downarrow\mathcal{L}_2$-proof of $\Sigma : \Psi, B; \Gamma \vdash \Delta; \Upsilon$. Conversely, assume that $\Sigma : \Psi, B; \Gamma \vdash \Delta; \Upsilon$ has a $\Downarrow\mathcal{L}_2$-proof Ξ. This proof ends with a right-introduction phase and we list the $n \geq 0$ premises of that phase as the sequents $\Sigma, \Sigma_i : \Psi, \Psi_i, B; \Gamma_i \vdash \mathcal{A}_i; \Upsilon, \Upsilon_i$, for $1 \leq i \leq n$. Given all of these $\Downarrow\mathcal{L}_2$-proofs, we can build the following n additional proofs (for $1 \leq i \leq n$).

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma, \Sigma_i : \Psi, \Psi_i, B; \Gamma_i \vdash \mathcal{A}_i; \Upsilon, \Upsilon_i}{\Sigma, \Sigma_i : \Psi, \Psi_i, B; \Gamma_i \vdash \bot, \mathcal{A}_i; \Upsilon, \Upsilon_i}\ \bot\text{R}}{\Sigma, \Sigma_i : \Psi, \Psi_i; \Gamma_i \vdash B \Rightarrow \bot, \mathcal{A}_i; \Upsilon, \Upsilon_i}\ \Rightarrow R \qquad \dfrac{}{\Sigma, \Sigma_i : \cdot; \bot \Downarrow \cdot \vdash \cdot;}\ \bot\text{L}}{\Sigma, \Sigma_i : \Psi, \Psi_i; \Gamma_i \Downarrow (B \Rightarrow \bot) \multimap \bot \vdash \mathcal{A}_i; \Upsilon, \Upsilon_i}\ \multimap L}{\Sigma, \Sigma_i : \Psi, \Psi_i; \Gamma_i, (B \Rightarrow \bot) \multimap \bot \vdash \mathcal{A}_i; \Upsilon, \Upsilon_i}\ \textit{decide}$$

We can now build a proof of $\Sigma : \Psi; \Gamma, (B \Rightarrow \bot) \multimap \bot \vdash \Delta; \Upsilon$ by attaching the right phase at the end of Ξ to these other premises.

Now consider the second equivalence. From $\Sigma : \Psi; \Gamma \vdash_{\Downarrow} \Delta; \Upsilon, B$ we immediate conclude $\Sigma : \Psi; \Gamma \vdash_{\Downarrow} \Delta, ?\,B; \Upsilon$ by using the $?\,R$ rule. Conversely, assume

$\Sigma : \Psi; \Gamma \vdash_{\Downarrow} \Delta, ?B; \Upsilon$. Since all right-introduction rules permute over each other, we can assume that the $?R$ has been applied first (reading the proof bottom-up) which has the premise $\Sigma : \Psi; \Gamma \vdash \Delta; \Upsilon, B$. □

Theorem 6.45 (Completeness of $\Downarrow\mathcal{L}_2$-proofs)**.** *Let Δ and Γ be multisets of* ***L*** *formulas. If $\Sigma : \Gamma \vdash \Delta$ has a* ***L*** *proof then $\Sigma : \cdot; \Gamma^{\triangledown} \vdash \Delta^{\triangledown}; \cdot$ has a $\Downarrow\mathcal{L}_2$-proof.*

Proof. We prove completeness by showing that the inference rules of the **L** proof system are all admissible (via the $(\cdot)^{\triangledown}$ mapping) in the $\Downarrow\mathcal{L}_2$-proof system. Assume that $\Sigma : \Delta \vdash \Gamma$ has a **L** proof Ξ. We proceed by induction on the structure of Ξ.

In the case that Ξ is an instance of the initial rule, Δ and Γ are equal and contain the single element B. By Proposition 6.31, $\Sigma : \cdot; B^{\triangledown} \vdash_{\Downarrow} B^{\triangledown}; \cdot$. In the case that the last inference rule is an instance of the cut rule

$$\frac{\Sigma : \Gamma_1 \vdash B, \Delta_1 \qquad \Sigma : \Gamma_2, B \vdash \Delta_2}{\Sigma : \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ cut,$$

we are allowed to assume that $\Sigma : \cdot; \Gamma_1^{\triangledown} \vdash_{\Downarrow} B^{\triangledown}, \Delta_1^{\triangledown}; \cdot$ and $\Sigma : \cdot; \Gamma_2^{\triangledown}, B^{\triangledown} \vdash_{\Downarrow} \Delta_2^{\triangledown}; \cdot$. Using the *cut* rule of $\Downarrow^{+}\mathcal{L}_2$ and the cut elimination theorem (Theorem 6.42), we know that $\Sigma : \cdot; \Gamma_1^{\triangledown}, \Gamma_2^{\triangledown} \vdash_{\Downarrow} \Delta_1^{\triangledown}, \Delta_2^{\triangledown}; \cdot$.

Since the right-introduction rules for the connectives $\{\top, \&, \forall, \bot, \invamp\}$ are essentially the same in **L** and $\Downarrow\mathcal{L}_2$ proof systems, it is immediate to treat the case where the proof Ξ is a right-introduction rule for one of these connectives. On the other hand, the left introduction rules for these connectives can be applied even when the right is not a collection of atomic formulas. In these cases, we proceed by using the cut elimination result for $\Downarrow^{+}\mathcal{L}_2$ proofs. For example, assume that the last inference rule for Ξ is

$$\frac{\Sigma : \Gamma, B_i \vdash \Delta}{\Sigma : \Gamma, B_1 \,\&\, B_2 \vdash \Delta}\ \&L\ (i = 1, 2).$$

By the inductive hypothesis, we know that $\Sigma : \cdot; \Gamma^{\triangledown}, B_i^{\triangledown} \vdash_{\Downarrow} \Delta^{\triangledown}; \cdot$. By Proposition 6.31 we know that $\Sigma : \cdot; B_1^{\triangledown} \,\&\, B_2^{\triangledown} \vdash B_1^{\triangledown} \,\&\, B_2^{\triangledown}; \cdot$ has a $\Downarrow\mathcal{L}_2$-proof. Immediate subproofs of that proof are proofs of $\Sigma : \cdot; B_1^{\triangledown} \,\&\, B_2^{\triangledown} \vdash B_i^{\triangledown}; \cdot$ for $i = 1$ and $i = 2$. Using the cut elimination result (Theorem 6.42), we can conclude that $\Sigma : \cdot; \Gamma^{\triangledown}, B_1^{\triangledown} \,\&\, B_2^{\triangledown} \vdash_{\Downarrow} \Delta^{\triangledown}; \cdot$. The left-introduction rules for $\{\top, \forall, \bot, \invamp\}$ can be done similarly, invoking an application of the cut elimination theorem.

To illustrate how to show that the introduction rules for the positive connectives $\{\mathbf{0}, \oplus, \exists, \mathbf{1}, \otimes\}$ are treated, we illustrate the cases where the last inference rule of Ξ is $\oplus R$ and $\oplus$L.

$$\frac{\Sigma : \Gamma \vdash B_i, \Delta}{\Sigma : \Gamma \vdash B_1 \oplus B_2, \Delta}\ \oplus\mathrm{R}\ (i = 1, 2)$$

By the inductive hypothesis, we can assume that $\Sigma : \cdot; \Gamma^\nabla \vdash_\Downarrow B_i^\nabla, \Delta^\nabla; \cdot$. Also note that the sequent $\Sigma : \cdot; B_i^\nabla, (B_1^\nabla \multimap \bot) \mathbin{\&} (B_2^\nabla \multimap \bot) \vdash \cdot; \cdot$ has a $\Downarrow\mathcal{L}_2$-proof (an observation that requires the use of Theorem 6.31). These $\Downarrow\mathcal{L}_2$-proofs can be brought together to prove the $(\cdot)^\nabla$ translation of the sequent $\Sigma : \Gamma \vdash B_1 \oplus B_2, \Delta$.

$$\dfrac{\dfrac{\dfrac{\Sigma : \cdot; \Gamma^\nabla \vdash B_i^\nabla, \Delta^\nabla; \cdot \quad \Sigma : \cdot; B_i^\nabla, (B_1^\nabla \multimap \bot) \mathbin{\&} (B_2^\nabla \multimap \bot) \vdash \cdot; \cdot}{\Sigma : \cdot; \Gamma^\nabla, (B_1^\nabla \multimap \bot) \mathbin{\&} (B_2^\nabla \multimap \bot) \vdash \Delta^\nabla; \cdot}\ \mathit{cut}}{\Sigma : \cdot; \Gamma^\nabla, (B_1^\nabla \multimap \bot) \mathbin{\&} (B_2^\nabla \multimap \bot) \vdash \bot, \Delta^\nabla; \cdot}\ \bot\mathrm{R}}{\Sigma : \cdot; \Gamma^\nabla \vdash ((B_1^\nabla \multimap \bot) \mathbin{\&} (B_2^\nabla \multimap \bot)) \multimap \bot, \Delta^\nabla; \cdot}\ \multimap\mathrm{R}$$

Next, consider the case in which the final inference rule of Ξ is

$$\dfrac{\Sigma : \Gamma, B \vdash \Delta \qquad \Sigma : \Gamma, C \vdash \Delta}{\Sigma : \Gamma, B \oplus C \vdash \Delta}\ \oplus\mathrm{L}.$$

By the inductive assumption, we have both $\Sigma : \cdot; \Gamma^\nabla, B^\nabla \vdash_\Downarrow \Delta^\nabla; \cdot$ and $\Sigma : \cdot; \Gamma^\nabla, C^\nabla \vdash_\Downarrow \Delta^\nabla; \cdot$. Attaching the $\Downarrow\mathcal{L}_2$-proofs of these two sequents to the following derivation finishes the proof for the $\oplus$L introduction rule.

$$\dfrac{\dfrac{\dfrac{\Sigma : \cdot; \Gamma^\nabla, B_1^\nabla \vdash \Delta^\nabla; \cdot}{\Sigma : \cdot; \Gamma^\nabla, B_1^\nabla \vdash \bot, \Delta^\nabla; \cdot}}{\Sigma : \cdot; \Gamma^\nabla \vdash B_1^\nabla \multimap \bot, \Delta^\nabla; \cdot} \quad \dfrac{\dfrac{\Sigma : \cdot; \Gamma^\nabla, B_2^\nabla \vdash \Delta^\nabla; \cdot}{\Sigma : \cdot; \Gamma^\nabla, B_2^\nabla \vdash \bot, \Delta^\nabla; \cdot}}{\Sigma : \cdot; \Gamma^\nabla \vdash B_2^\nabla \multimap \bot, \Delta^\nabla; \cdot}}{\Sigma : \cdot; \Gamma^\nabla \vdash (B_1^\nabla \multimap \bot) \mathbin{\&} (B_2^\nabla \multimap \bot), \Delta^\nabla; \cdot}$$

Since the sequent

$$\Sigma : \cdot; (B_1^\nabla \multimap \bot) \mathbin{\&} (B_2^\nabla \multimap \bot), ((B_1^\nabla \multimap \bot) \mathbin{\&} (B_2^\nabla \multimap \bot)) \multimap \bot \vdash \cdot; \cdot$$

has a $\Downarrow\mathcal{L}_2$-proof, we can use the cut-elimination theorem to obtain a proof of the $(\cdot)^\nabla$ translation of $\Sigma : \Gamma, B_1 \oplus B_2 \vdash \Delta$.

The introduction rules for **0**, **1**, $\otimes$, and $\exists$, can be done similarly, invoking an application of the cut elimination theorem. Thus, the remaining rules in **L** that need to be considered are the exponentials. We consider the four rules for ! in the $\Downarrow\mathcal{L}_2$ proof systems.

Assume that the last inference rule of Ξ is

$$\dfrac{\Sigma : \Gamma \vdash \Delta}{\Sigma : \Gamma, ! B \vdash \Delta}\ !W$$

By the inductive hypothesis, we know that $\Sigma : \cdot; \Gamma^\nabla \vdash_\Downarrow \Delta^\nabla; \cdot$. By Proposition 6.33, we can weaken this sequent and conclude that $\Sigma : B^\nabla; \Gamma^\nabla \vdash_\Downarrow \Delta^\nabla; \cdot$. By applying Lemma 6.44, we have $\Sigma : \cdot; \Gamma^\nabla, (B^\nabla \Rightarrow \bot) \multimap \bot \vdash_\Downarrow \Delta^\nabla; \cdot$, which completes this case.

Assume that the last inference rule of Ξ is

$$\dfrac{\Sigma : \Gamma, !\,B, !\,B \vdash \Delta}{\Sigma : \Gamma, !\,B \vdash \Delta}\ !\,C$$

By the inductive hypothesis, we know that $\Sigma : \cdot; \Gamma^{\triangledown}, (!\,B)^{\triangledown}, (!\,B)^{\triangledown} \vdash_{\Downarrow} \Delta^{\triangledown}; \cdot$. Using cut-elimination on the following proof (where the proofs of the two left premises is guaranteed by Exercise 6.32),

$$\dfrac{\Sigma : B^{\triangledown}; \cdot \vdash (!\,B)^{\triangledown}; \cdot \qquad \dfrac{\Sigma : B^{\triangledown}; \cdot \vdash (!\,B)^{\triangledown}; \cdot \quad \Sigma : \cdot; \Gamma^{\triangledown}, (!\,B)^{\triangledown}, (!\,B)^{\triangledown} \vdash \Delta^{\triangledown}; \cdot}{\Sigma : B^{\triangledown}; \Gamma^{\triangledown}, (!\,B)^{\triangledown} \vdash \Delta^{\triangledown}; \cdot}\ \mathit{cut}}{\Sigma : B^{\triangledown}; \Gamma^{\triangledown} \vdash \Delta^{\triangledown}; \cdot}\ \mathit{cut}$$

we have $\Sigma : B^{\triangledown}; \Gamma^{\triangledown} \vdash_{\Downarrow} \Delta^{\triangledown}; \cdot$. Using Lemma 6.44, we can conclude that $\Sigma : \cdot; \Gamma^{\triangledown}, (B^{\triangledown} \Rightarrow \bot) \multimap \bot \vdash_{\Downarrow} \Delta^{\triangledown}; \cdot$.

The case when the last inference rule of Ξ is

$$\dfrac{\Sigma : \Gamma, B \vdash \Delta}{\Sigma : \Gamma, !\,B \vdash \Delta}\ !\,D$$

follows simply from a use of the *cut* rule and a proof of $\Sigma : \cdot; (!\,B)^{\triangledown} \vdash B; \cdot$ (Exercise 6.32).

Assume that the last rule of Ξ is

$$\dfrac{\Sigma : !\,\Gamma \vdash B, ?\,\Delta}{\Sigma : !\,\Gamma \vdash !\,B, ?\,\Delta}\ !\mathrm{R}$$

By the inductive hypothesis, we know that $\Sigma : \cdot; (!\,\Gamma)^{\triangledown} \vdash_{\Downarrow} B^{\triangledown}, (?\,\Delta)^{\triangledown}; \cdot$. By repeatedly applying Lemma 6.44, we can conclude that $\Sigma : \Gamma^{\triangledown}; \cdot \vdash_{\Downarrow} B^{\triangledown}, (?\,\Delta)^{\triangledown}; \cdot$. Since all the right rules permute over each other, we can assume that the $?\,R$ rule are applied below the rules related to B, leading us to $\Sigma : \Gamma^{\triangledown}; \cdot \vdash_{\Downarrow} B^{\triangledown}; \Delta^{\triangledown}$. With a proof of that sequent, we now build the following proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma : \Gamma^{\triangledown}; \cdot \vdash B^{\triangledown}; \Delta^{\triangledown} \qquad \dfrac{}{\Sigma : \Gamma^{\triangledown}; \cdot \Downarrow \bot \vdash \cdot; \Delta^{\triangledown}}\ \bot\mathrm{L}}{\Sigma : \Gamma^{\triangledown}; \cdot \Downarrow B^{\triangledown} \Rightarrow \bot \vdash \cdot; \Delta^{\triangledown}}\ {\Rightarrow}\mathrm{L}}{\Sigma : \Gamma^{\triangledown}; B^{\triangledown} \Rightarrow \bot \vdash \cdot; \Delta^{\triangledown}}\ \mathit{decide}}{\Sigma : \Gamma^{\triangledown}; B^{\triangledown} \Rightarrow \bot \vdash \bot; \Delta^{\triangledown}}\ \bot\mathrm{R}}{\Sigma : \Gamma^{\triangledown}; \cdot \vdash (B^{\triangledown} \Rightarrow \bot) \multimap \bot; \Delta^{\triangledown}}\ {\multimap}\mathrm{R}$$

By repeated application of Lemma 6.44, we can conclude

$$\Sigma : \cdot; (!\,\Gamma)^{\triangledown} \vdash_{\Downarrow} (B^{\triangledown} \Rightarrow \bot) \multimap \bot; \Delta^{\triangledown}$$

and by repeated application of the $?\,R$ rule, we have

$$\Sigma : \cdot; (!\,\Gamma)^{\triangledown} \vdash_{\Downarrow} (B^{\triangledown} \Rightarrow \bot) \multimap \bot, (?\,\Delta)^{\triangledown}; \cdot,$$

which provides a proof of our desired sequent.

The only remaining **L** rules to consider are the four rules for the ?-exponential. Since ? is translated directly to ? by $(\cdot)^\nabla$, the proofs involving ? are similar but simpler than for the !-exponential. We do not include these cases here. $\square$

A simple consequence of cut-elimination for $\Downarrow^+\mathcal{L}_2$-proofs is that cut can be eliminated from the **L** system.

Theorem 6.46. *A sequent provable in* **L** *can be proved without the cut rule.*

Proof. We first show that a sequent in **L** that is the conclusion of the cut rule applied to two cut-free proofs can be proved by a cut-free proof. Once this is done, a simply induction can remove all instances of the cut rule from a proof. Thus, assume that $\Sigma : B, \Delta_1 \vdash \Gamma_1$ and $\Sigma : \Delta_2 \vdash \Gamma_2, B$ have cut-free **L** proofs. By the completeness of $\Downarrow\mathcal{L}_2$-proofs (Theorem 6.45), we know that $\Sigma : \cdot; B^\nabla, \Delta_1^\nabla \vdash \Gamma_1^\nabla; \cdot$ and $\Sigma : \cdot; \Delta_2^\nabla \vdash B^\nabla, \Gamma_2^\nabla; \cdot$ have $\Downarrow\mathcal{L}_2$-proofs. Using the *cut* inference rule of $\Downarrow\mathcal{L}_2$, we know that $\Sigma : \cdot; \Delta_1^\nabla, \Delta_2^\nabla \vdash_\Downarrow \Gamma_1^\nabla, \Gamma_2^\nabla; \cdot$ has $\Downarrow^+\mathcal{L}_2$-proof. By the cut-elimination theorem for $\Downarrow^+\mathcal{L}_2$-proofs (Theorem 6.42), we know that this sequent also has a (cut-free) $\Downarrow\mathcal{L}_2$-proof. By the soundness theorem of $\Downarrow\mathcal{L}_2$-proofs (Theorem 6.43) we finally know that $\Sigma : \Delta_1, \Delta_2 \vdash \Gamma_1, \Gamma_2$ has a cut-free proof. $\square$

6.9 Bibliographic notes

More observations about interactions between the structural rules and cut-elimination are given by Danos et al. [1997] and Lafont in [Girard et al., 1989].

The notion of the *polarity* of logical connectives that we have used here is due to Andreoli [1992] and Girard [1991a]. Those two papers also introduced the notion of multi-zone sequents for the treatment of bounded and unbounded contexts in sequents for linear logic.

A one-sided sequent calculus proof system for linear logic is given in Figure 6.5. The focused variant of that proof system is given in Figure 6.13. This proof system is due to Andreoli [1992]. The main difference between Andreoli's original system and the one given here is that the zone between $\vdash$ and $\Uparrow$ is a list in his system while it is a multiset in Figure 6.13. The D_1 rule corresponds to the *decide* rule while the D_2 rule corresponds to the *decide* ! rule. Similarly, the I_1 rule corresponds to the *init* rule while the I_2 rule corresponds to the *init* ? rule. The rules $[R \Uparrow]$ and $[R \Downarrow]$ are not needed in $\Downarrow\mathcal{L}_2$-proofs given our use of two-sided sequents and implications.

The first major result that one usually attempts to prove about focused proof systems is that they are complete with respect to their unfocused version. Andreoli proved this result using a permutation argument in which unfocused

$$\frac{\Sigma \vdash \Gamma \Uparrow \Delta; \Upsilon}{\Sigma \vdash \bot, \Gamma \Uparrow \Delta; \Upsilon}\,[\bot] \quad \frac{\Sigma \vdash F, G, \Gamma \Uparrow \Delta; \Upsilon}{\Sigma \vdash F \mathbin{\rotatebox[origin=c]{180}{\&}} G, \Gamma \Uparrow \Delta; \Upsilon}\,[\mathbin{\rotatebox[origin=c]{180}{\&}}] \quad \frac{\Sigma \vdash \Gamma \Uparrow \Delta; \Upsilon, F}{\Sigma \vdash ?F, \Gamma \Uparrow \Delta; \Upsilon}\,[?]$$

$$\frac{}{\Sigma \vdash \top, \Gamma \Uparrow \Delta; \Upsilon}\,[\top] \qquad \frac{\Sigma \vdash F, \Gamma \Uparrow \Delta; \Upsilon \quad \Sigma \vdash G, \Gamma \Uparrow \Delta; \Upsilon}{\Sigma \vdash F \mathbin{\&} G, \Gamma \Uparrow \Delta; \Upsilon}\,[\&]$$

$$\frac{y : \tau, \Sigma \vdash B[y/x], \Gamma \Uparrow \Delta; \Upsilon}{\Sigma \vdash \forall_\tau x.B, \Gamma \Uparrow \Delta; \Upsilon}\,[\forall] \qquad \frac{}{\Sigma \vdash \mathbf{1} \Downarrow \cdot; \Upsilon}\,[\mathbf{1}]$$

$$\frac{\Sigma \vdash F \Downarrow \Delta_1; \Upsilon \quad \Sigma \vdash G \Downarrow \Delta_2; \Upsilon}{\Sigma \vdash F \otimes G \Downarrow \Delta_1, \Delta_2; \Upsilon}\,[\otimes] \qquad \frac{\Sigma \vdash F \Uparrow \cdot; \Upsilon}{\Sigma \vdash !F \Downarrow \cdot; \Upsilon}\,[!]$$

$$\frac{\Sigma \vdash F_i \Downarrow \Delta; \Upsilon}{\Sigma \vdash F_1 \oplus F_2 \Downarrow \Delta; \Upsilon}\,[\oplus_i] \qquad \frac{\Sigma \Vdash t : \tau \quad \Sigma \vdash B[t/x] \Downarrow \Delta; \Upsilon}{\Sigma \vdash \exists_\tau x.B \Downarrow \Delta; \Upsilon}\,[\exists]$$

$$\frac{\Sigma \vdash \Gamma \Uparrow \Delta, F; \Upsilon}{\Sigma \vdash F, \Gamma \Uparrow \Delta; \Upsilon}\,[R\Uparrow] \quad \text{provided that } F \text{ is a literal or a positive formula}$$

$$\frac{\Sigma \vdash F \Uparrow \Delta; \Upsilon}{\Sigma \vdash F \Downarrow \Delta; \Upsilon}\,[R\Downarrow] \quad \text{provided that } F \text{ is a negative formula}$$

$$\frac{}{\Sigma \vdash A^\perp \Downarrow A; \Upsilon}\,[I_1] \qquad \frac{}{\Sigma \vdash A^\perp \Downarrow \cdot; \Upsilon, A}\,[I_2]$$

$$\frac{\Sigma \vdash F \Downarrow \Delta; \Upsilon}{\Sigma \vdash \cdot \Uparrow \Delta, F; \Upsilon}\,[D_1] \qquad \frac{\Sigma \vdash F \Downarrow \Delta; \Upsilon, F}{\Sigma \vdash \cdot \Uparrow \Delta; \Upsilon, F}\,[D_2]$$

Figure 6.13: The $\mathcal{J}$ proof system. The rule $[\forall]$ has the usual proviso that y is not in Σ. In $[\oplus_i]$, $i = 1$ or $i = 2$.

proofs could be made progressively more focused. The proof of the completeness of $\Downarrow\mathcal{L}_2$-proofs given in [Miller, 1996] directly relied on Andreoli's proof of completeness.

A direct proof of cut-elimination for a focused proof system for linear logic was given by Bruscoli and Guglielmi [2006] and Guglielmi [1996] for the subset of Forum that does not include the (redundant) ? exponential and in which formulas were limited to what we call paths here. Their proof described cut-elimination at the level of synthetic inference rules.

The style of completeness proof given here first proves that the generalized initial rule and the cut rule are admissible in the focused proof system. Given those results, it is then a simple matter to conclude completeness of focusing. This approach to proving properties about focused proof systems was given in [Chaudhuri, 2006; Chaudhuri et al., 2008b] for intuitionistic linear logic and was later extended by Liang and Miller [2011, 2022] to intuitionistic and classical logics. Further developments of this style of proof, along with a formal verification, is given by Simmons [2014] for propositional intuitionistic logic.

As Exercise 6.6 shows, it is possible for linear logic to have a collection of different exponentials in linear logic. A presentation of such additional operators, including a cut-elimination theorem, was first given in [Danos et al., 1993]. Since these additional operators do not necessarily need to permit weakening and contraction, these additional operators do not necessarily allow one to prove the exponential laws (as described in Exercise 6.3). For these reasons, such additional operators have been called *subexponentials* in [Nigam and Miller, 2009]: that paper also illustrates how subexponentials can be used to enhance the expressiveness of proof search specifications based on linear logic (see also [Chaudhuri, 2018; Liang and Miller, 2015; Olarte et al., 2015]).

When Girard [1987] introduced linear logic, he also introduced *proof-nets* as a proof system specifically designed to capture the parallelism in proofs better than sequent calculus proofs. Here we have stressed using focused proof system as an improvement to sequent calculus. Focused proof systems can be extended with the notion of *multi-focusing* in which focusing can be made on more than one formula within the left-introduction phase [Delande and Miller, 2008]. Such an extension provides another method for capturing parallel actions within a proof structure [Chaudhuri et al., 2008a, 2016].

Exercise 6.8 illustrated a property of formulas B for which $B \equiv\, !B$ holds. If we restrict B to come from MALL, then very few formulas have this property. In full linear logic, any formula of the form $!C$ has this property since $!C \equiv\, !!C$. If one extends MALL with least fixed points and term equality (thus moving linear logic closer to model checking and arithmetic), then there are many other formulas that satisfy that equivalence: see [Baelde, 2012; Baelde and Miller, 2007; Heath and Miller, 2019].

An implementation of programming language based on $\mathcal{L}_1$ was described in [Hodas and Tamura, 2001]. Forum has been given a couple of implementations: see [López and Pimentel, 1998; Urban, 1997]. An important part of these implementation is the *lazy splitting of multisets* during proof search, a technique that is described in Section 6.6. This technique was first presented in [Hodas and Miller, 1991, 1994] and extended in [Cervesato et al., 2000b, 1996; Hodas et al., 1998].

Chapter 7

Linear logic programming

In this chapter, we present several, small logic programs: the first examples use only the Lolli fragment and later example use the full Forum presentation of linear logic.

7.1 Encoding multisets as formulas

Consider the following encoding of multisets of terms as formulas in linear logic. Let token *item* be a predicate of one argument: the linear logic atomic formula *item* x will denote the multiset containing just the one element x occurring once. There are two natural encoding of multisets into formulas using this predicate. The *conjunctive* encoding uses $\mathbf{1}$ for the empty multiset and $\otimes$ to combine two multisets. For example, the multiset $\{1, 2, 2\}$ is encoded by the linear logic formula *item* $1 \otimes$ *item* $2 \otimes$ *item* 2. Proofs search using this style encoding places multiset on the left of the sequent arrow. This approach is favored when an intuitionistic subset of linear logic is used, such as in the $\mathcal{L}_1$ subset of linear logic (Section 6.4). The dual encoding, the *disjunctive* encoding, uses $\perp$ for the empty multiset and $\parr$ to combine two multisets. Proofs search using this style encoding places multisets on the right of the sequent arrow and multiple conclusion sequents are now required, such as in the $\mathcal{L}_2$ presentation of linear logic (Section 6.7).

Exercise 7.1.(‡) Let M_1 and M_2 be two multisets of natural numbers and let P_1 and P_2 be their conjunctive encoding, respectively. Show that $\vdash P_1 \multimap P_2$ implies $\vdash P_2 \multimap P_1$.

Exercise 7.2. Redo Exercise 7.1 but this time assuming that P_1 and P_2 are the disjunctive encoding M_1 and M_2.

Let S and T be the two formulas *item* $s_1 \parr \cdots \parr$ *item* s_n and *item* $t_1 \parr \cdots \parr$ *item* t_m, respectively ($n, m \geq 0$). Exercise 7.2 allows us to conclude that

$\vdash S \multimap T$ if and only if $\vdash T \multimap S$ if and only if the two multisets $\{s_1, \ldots, s_n\}$ and $\{t_1, \ldots, t_m\}$ are equal. Consider now the following two ways for encoding the multiset inclusion $S \sqsubseteq T$.

- $S \parr 0 \multimap T$. This formula mixes multiplicative connectives with the additive connective 0: the latter allows items that are not matched between S and T to be deleted.

- $\exists q(S \parr q \multimap T)$. This formula mixes multiplicative connectives with a higher-order quantifier. Intuitively, we would like to consider the instantiation for q to be the multiset difference of S from T, such a restriction on p is not part of this formula: specifically, q could be instantiate with any linear logic formula.

As it turns out, these two approaches are equivalent in linear logic, in the sense that

$$\forall S \forall T[(S \parr 0 \multimap T) \equiv \exists q(S \parr q \multimap T)].$$

holds in **L**. Recall from Section 6.3.2 that the equivalence $B \equiv C$ is defined to hold when the formula $(B \multimap C) \,\&\, (C \multimap B)$ is provable in linear logic.

7.2 A syntax for Lolli programs

In order to present several examples in this chapter, we extend Prolog and λProlog syntax to accommodate Lolli logic programs. As we have already indicated in Section 6.5, the symbols `=>` and `:-` of Prolog and λProlog are used to represent $\Rightarrow$, and the converse of $\multimap$, respectively. We shall also write `-o` and `<=` to represent the $\multimap$ and the converse of $\Rightarrow$. Given these connectives we can define (in the sense described in Section 5.9) the symbols `true`, `,` (comma), `;` (semicolon), `exists`, and `bang` which represent the linear logic connectives **1**, $\otimes$, $\oplus$, $\exists$, and !, respectively. These definitions can be written as follows.

```
type true    o.
type ,       o -> o -> o.
type ;       o -> o -> o.
type exists (A -> o) -> o.
type bang    o -> o.

true.
(P , Q) :- P :- Q.
(P ; Q) :- P.
(P ; Q) :- Q.
exists B :- (B T).
bang G <= G.
```

These clause encode only the right-introduction rules for their respective logical connective. We also allow the symbols `&` and `erase` to denote, respectively, & and $\top$.

7.3 Permuting a list

Since the bounded part of contexts in $\mathcal{L}$-proofs are multisets, it is a simple matter to permute a list of items by first loading the list's members into the bounded part of a context and then unloading them. The latter operation is nondeterministic and can succeed once for each permutation of the loaded list. Consider the following simple program:

```
kind list             type -> type.
type nil              list A.
type ::               A -> list A -> list A.
type load, unload     list A -> list A -> o.

load nil K      :- unload K.
load (X::L) K   :- (item X -o load L K).
unload nil.
unload (X::L)   :- item X, unload L.
```

Here, `nil` denotes the empty list and `::` the list constructor. The meaning of `load` and `unload` is dependent on the contents of the bounded part of the context, so the correctness of these clauses must be stated relative to a context. Let Γ be a set of formulas containing the four formulas displayed above and any other formulas that do not contain either `item`, `load`, or `unload` as their head symbol. (The *head symbol* of a clause of the form A or $G \multimap A$ is the predicate symbol that is the head of the atom A.) Let Δ be the multiset containing exactly the atomic formulas

$$\texttt{item } a_1\texttt{, ..., item } a_n.$$

We shall say that such a context *encodes* the multiset $\{a_1, \ldots, a_n\}$. It is now an easy matter to prove the following two assertions about `load` and `unload`:

- The goal `(unload K)` is provable from $\Gamma; \Delta$ if and only if `K` is a list containing the same elements with the same multiplicity as the multiset encoded in Δ.

- The goal `(load L K)` is provable from $\Gamma; \Delta$ if and only if `K` is a list containing the same elements with the same multiplicity as in the list `L` together with the multiset encoded in the context Δ.

In order for `load` and `unload` to correctly permute the elements of a list, we must guarantee two things about the context: first, the predicates `item`, `load`, and `unload` 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 quantifiers over the predicate names `item`, `load`, and `unload` (we discuss such "higher-order quantification" elsewhere). The second condition — that the unbounded part of a context is empty — can be managed by making use of the modal nature of !, which we now discuss in more detail.

Consider proving the sequent $\Gamma; \Delta \longrightarrow\ !G_1 \otimes G_2$, where Γ and Δ are program clauses and G_1 and G_2 are goal formulas. Given the completeness of uniform proofs for the system $\mathcal{L}'$, 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. In a sense, since bounded resources can come and go within contexts during a computation, 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 two lists given the example program above: add either the formula

```
perm L K  :-  bang(load L K).
```

or, equivalently, the formula

```
perm L K  <=  load L K.
```

to those defining `load` and `unload`. Thus attempting to prove `(perm L K)` will result in an attempt to prove `(load L K)` with an empty bounded context. From the description of `load` above, `L` and `K` must be permutations of each other.

Exercise 7.3. Let Γ_0 be the collection of $\mathcal{L}_1$-formulas given in Section 7.2 for defining various symbols denoting logical connectives, and let Γ be a collection of $\mathcal{L}_1$-formulas that do not define those same symbols. Prove the following about provability in $\Downarrow\mathcal{L}_1$. The sequent $\Gamma_0, \Gamma; \Delta \vdash$ `bang` G is provable if and only if $\Gamma_0, \Gamma; \Delta \vdash$ `one` $\& G$ is provable if and only if Δ is empty and $\Gamma_0, \Gamma; \cdot \vdash G$ is provable.

7.4 Multiset rewriting

The ideas presented in the permutation example can easily 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 $M \Longrightarrow_H N$ 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 $\Longrightarrow_H^*$ be the reflexive and transitive closure of $\Longrightarrow_H$.

Given a rewriting system H, we wish to specify 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 $\Longrightarrow_H^*$ relation. Let Γ_0 be the following set of formulas (these are independent of H):

```
rewrite L K      <= load L K.

load (X::L) K    :- (item X -o load L K).
load nil    K    :- rew K

rew K            :- unload K.

unload (X::L)    :- item X, unload L.
unload nil.
```

Taken alone, these clauses give a slightly different version of the `permute` program of the last example. The only addition is the binary predicate `rew`, which will be used as a socket into which we can plug a particular rewrite system.

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 pair $\langle\{a_1, \ldots, a_n\}, \{b_1, \ldots, b_m\}\rangle$ then this pair is encoded as the clause:

```
rew K :- item a1,   ...,   item an,
        (item b1 -o ... -o item bm -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. For example, if $H = \{\langle\{a, b\}, \{b, c\}\rangle, \langle\{a, a\}, \{a\}\rangle\}$ then Γ_H is the set of clauses:

```
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 $M \Longrightarrow_H^* N$ if and only if the goal `(rewrite L K)` is provable from the context $\Gamma_0, \Gamma_H; \emptyset$.

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

Exercise 7.4 (`maxa` revisited)**.** (‡) Consider again Exercise 5.42 in which it was argued that computing the maximum of a multiset of natural numbers was not possible if that multiset was encoded as atomic formulas in the left-side of sequents in **I**-proofs. It is possible to write such a program when using $\mathcal{L}_1$ formulas: in fact, the bounded sequents of $\Downarrow\mathcal{L}_1$-proofs can be used to start and compute with such a multiset. Write a logic program $\mathcal{P}$ using $\mathcal{L}_1$-formula such the following holds. If N is a set of natural numbers $\{n_1, \ldots, n_k\}$ and $k \geq 1$ then the $\Downarrow\mathcal{L}_1$-sequent $\mathcal{P}; a\ n_1, \ldots, a\ n_k \vdash maxa\ m$ is provable if and only if m is the maximum of $\{n_1, \ldots, n_k\}$.

Exercise 7.5. (‡) As in Exercise 7.4, let $k \geq 1$ and let N be a set of natural numbers $\{n_1, \ldots, n_k\}$. Write a logic program $\mathcal{P}$ that computes the sum $n_1 + \cdots + n_k$. More precisely, the $\Downarrow\mathcal{L}_1$-sequent $\mathcal{P}; a\ n_1, \ldots, a\ n_k \vdash sumup\ m$ should be provable if and only if $m = n_1 + \cdots + n_k$. Contrast this exercise with the predicate `sumup` in Figure 5.4.

Exercise 7.6. Represent the finite graph $G = (N, E)$, with nodes N and edges $E \subseteq N \times N$, as the two sets of atomic formulas

$$\mathcal{N} = \{node(x) \mid x \in N\} \quad \text{and} \quad \mathcal{E} = \{edge(x, y) \mid \langle x, y\rangle \in E\}.$$

Consider the logic program $\mathcal{P}$ that consists of the following declarations and clauses.

```
kind node               type.
type connected, loop    o.
type node, nd           node -> o.

connected :- node u, (nd u => loop).
loop.
loop :- nd u, edge u v, node v, (nd v => loop).
```

Show that the sequent $\mathcal{P}, \mathcal{E}; \mathcal{N} \vdash connected$ is provable in $\Downarrow\mathcal{L}_1$ if and only if the graph G is connected.

Exercise 7.7 (No notconnected)**.** Represent the finite graph $G = (N, E)$, with nodes N and edges $E \subseteq N \times N$, as the set of atomic formulas

$$\mathcal{G} = \{node(x) \mid x \in N\} \cup \{edge(x, y) \mid \langle x, y\rangle \in E\}.$$

Argue why it is impossible to write a logic program $\mathcal{P}$ in first-order hereditary Harrop formulas that specifies the predicate $nc(x, y)$ such that for all $x, y \in N$, x and y are *not* connected by a path in the graph G if and only if the sequent $\mathcal{G}, \mathcal{P} \vdash nc(x, y)$ is provable.

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

Figure 7.1: A specification of an intuitionistic propositional object-logic

7.5 Context management in a theorem prover

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

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

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

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

This formula states that in order to prove some object-level formula `G`, first check to see if there is a conjunctive hypothesis, say `(A and B)`, in the context and, if so, attempt a proof of `G` from the context extended with the two hypotheses `A` and `B`. Other introduction and elimination rules can be specified similarly. Finally, the formula

```
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)` from the assumptions `(hyp H1)`, . . ., `(hyp Hi)` in the meta-logic if and only if there is a proof of `G` from the assumptions `H1`, . . ., `Hi` in the object-logic.

$$\frac{\Gamma, A, B \vdash G}{\Gamma, A, A \supset B \vdash G} \supset\!L_1,\ A \text{ atomic} \qquad \frac{\Gamma, C \supset D \supset B \vdash G}{\Gamma, (C \wedge D) \supset B \vdash G} \supset\!L_2$$

$$\frac{\Gamma, C \supset B, D \supset B \vdash G}{\Gamma, (C \vee D) \supset B \vdash G} \supset\!L_3 \qquad \frac{\Gamma \vdash G}{\Gamma, \bot \supset B \vdash G} \supset\!L_5$$

$$\frac{\Gamma, D \supset B \vdash C \supset D \qquad \Gamma, B \vdash G}{\Gamma, (C \supset D) \supset B \vdash G} \supset\!L_4$$

Figure 7.2: Replacements for the $\supset$L Rule

Unfortunately, an intuitionistic meta-logic does not permit the natural specification of 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 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 7.1. In that specification, a hypothesis is both "read from" and "written into" a context during the elimination of implications. All other elimination rules simply "read from" the context; they do not "write back." The formulas represented by the last two clauses in Figure 7.1 use a $\otimes$ with $\top$: this allows for all unused hypotheses to be erased, since the object logic has no restrictions on weakening.

It should be noted that this specification cannot be used effectively with a depth-first interpreter because when the implication left rule can be used once, it can be used any number of times: this can cause such an interpreter to loop. Fortunately, an alternative presentation of the implication left-introduction rule can solve this particular problem. For example, the proof system given by Dyckhoff [1992] and Hudelmaier [1992] can be expressed directly in this setting. In their papers, the left-introduction rule for implication can be replaced by the five rules in Figure 7.2. Thus, consider modifying the specification in Figure 7.1 by replacing its one formula specifying implication elimination with the five clauses for implication elimination in Figure 7.3 (derived from Figure 7.2), along with the (partial) axiomatization of object-level atomic formulas. Executing this linear logic program in a depth-first interpreter can

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

isatom p.
isatom q.
isatom r.
```

Figure 7.3: A contraction-free formulation of ⊃L.

yield a decision procedure for propositional intuitionistic logic.

7.6 Multiset rewriting in Forum

Since Forum contains Lolli, the techniques for rewriting multisets by using the bounded left-side zone can be used in Forum as well. However, it is also possible to use the bounded right-side zone as well. To illustrate that approach, consider the clause

$$a \parr b \circ\!\!- c \parr d \parr e.$$

When presenting examples of Forum specification we continue the habit of using $\circ\!\!-$ and $\Leftarrow$ as the converses of $\multimap$ and $\Rightarrow$ since they provide a more natural operational reading of clauses (similar to the use of `:-` in Prolog). Here, $\parr$ binds tighter than $\circ\!\!-$ and $\Leftarrow$. Consider the $\Downarrow\mathcal{L}_2$ sequent $\Sigma : \Psi; \Delta \vdash a, b, \Gamma; \Upsilon$ where the above clause is a member of Ψ. A proof for this sequent can proceed as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma : \Psi; \Delta \vdash c, d, e, \Gamma; \Upsilon}{\Sigma : \Psi; \Delta \vdash c, d \parr e, \Gamma; \Upsilon}}{\Sigma : \Psi; \Delta \vdash c \parr d \parr e, \Gamma; \Upsilon} \quad \dfrac{\dfrac{}{\Sigma : \Psi; \cdot \Downarrow a \vdash a; \Upsilon} \quad \dfrac{}{\Sigma : \Psi; \cdot \Downarrow b \vdash b; \Upsilon}}{\Sigma : \Psi; \cdot \Downarrow a \parr b \vdash a, b; \Upsilon}}{\Sigma : \Psi; \Delta \Downarrow c \parr d \parr e \multimap a \parr b \vdash a, b, \Gamma; \Upsilon}}{\Sigma : \Psi; \Delta \vdash a, b, \Gamma; \Upsilon}$$

We can interpret this fragment of a proof as a reduction of the multiset a, b, Γ to the multiset c, d, e, Γ by backchaining on the clause displayed above.

Of course, a clause may have multiple, top-level implications. In this case, the surrounding context must be manipulated properly to prove the sub-goals that arise in backchaining. Consider using the *decide* rule on the formula

$$A_1 \parr A_2 \Leftarrow G_4 \multimapinv G_3 \Leftarrow G_2 \multimapinv G_1$$

to prove the sequent $\Sigma : \Psi; \Delta \vdash A_1, A_2, \mathcal{A}; \Upsilon$. An attempt to prove this sequent would then lead to the attempt to prove the four sequents

$$\begin{array}{ll} \Sigma : \Psi; \Delta_1 \vdash G_1, \mathcal{A}_1; \Upsilon & \Sigma : \Psi; \cdot \vdash G_2; \Upsilon \\ \Sigma : \Psi; \Delta_2 \vdash G_3, \mathcal{A}_2; \Upsilon & \Sigma : \Psi; \cdot \vdash G_4; \Upsilon \end{array}$$

where Δ is the multiset union of Δ_1 and Δ_2, and $\mathcal{A}$ is the multiset union of $\mathcal{A}_1$ and $\mathcal{A}_2$. In other words, those subgoals immediately to the right of an $\Leftarrow$ are attempted with empty bounded contexts: the bounded contexts, here Δ and $\mathcal{A}$, are divided up and used in attempts to prove those goals immediately to the right of $\multimapinv$.

For an example of computing using multisets on the right of $\Downarrow\mathcal{L}_2$ sequents, consider again computing the sum of a multiset of natural numbers. Assume that we take the encoding of natural numbers and addition (`sum`) given in Figure 5.3, and make them available as $\mathcal{L}_2$ formulas. Now add to these formulas the following two formulas.

```
sumall M     :- acc M -o acc z.
acc N || a M :- sum N M S, acc S.
```

Exercise 7.8. Let Σ and Ψ be the signature and logic programs given above for *sumall* and *acc*. Show that the sequent

$$\Sigma : \Psi; \cdot \vdash a\ n_1 \parr a\ n_2 \parr \cdots \parr a\ n_i \parr sumall\ m; \cdot$$

is provable if and only if m is the sum of $n_1, \ldots, n_i$.

Many more examples of specifications written using the Forum presentation of linear logic appear in Chapters 10, 12, and 13.

7.7 Specification of sequent calculus proof systems

Given the proof-theoretic motivations of Forum and its inclusion of quantification at higher-order types, it is not surprising that it can be used to specify proof systems for various object-level logics. Below we illustrate how sequent calculus proof systems can be specified using the multiple conclusion aspect of Forum and show how properties of linear logic can be used to infer properties of the object-level proof systems. We shall use the terms *object-level logic* and *meta-level logic* to distinguish between the logic whose proof system is being specified and the logic of Forum.

Consider the well known, two-sided sequent proof systems for classical, intuitionistic, and linear logic. As we have described in Section 4.1, the distinction between sequents in these logics can be described by where the structural rules of thinning and contraction can be applied. In classical logic, these structural rules are allowed on both sides of the sequent arrow; in intuitionistic logic, no structural rules are allowed on the right of the sequent arrow; and in linear logic, they are not allowed on either side of the arrow. This suggests the following representation of sequents in these three systems. Let *bool* be the type of object-level propositional formulas and let $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ be two meta-level predicates of type $bool \to o$. Sequents in these four logics can be specified as follows: object-logic sequents will be two-sided and the left and right will be paired using $\longrightarrow$ (following Gentzen's original notation [1935]).

Linear: The sequent $B_1, \ldots, B_n \longrightarrow C_1, \ldots, C_m$ $(n, m \geq 0)$ can be represented by the meta-level formula

$$\lfloor B_1\rfloor \parr \cdots \parr \lfloor B_n\rfloor \parr \lceil C_1\rceil \parr \cdots \parr \lceil C_m\rceil.$$

Intuitionistic: The sequent $B_1, \ldots, B_n \longrightarrow C$ $(n \geq 0)$ can be represented by the meta-level formula

$$?\lfloor B_1\rfloor \parr \cdots \parr ?\lfloor B_n\rfloor \parr \lceil C\rceil.$$

Classical: The sequent $B_1, \ldots, B_n \longrightarrow C_1, \ldots, C_m$ $(n, m \geq 0)$ can be represented by the meta-level formula

$$?\lfloor B_1\rfloor \parr \cdots \parr ?\lfloor B_n\rfloor \parr ?\lceil C_1\rceil \parr \cdots \parr ?\lceil C_m\rceil.$$

The $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ predicates are used to identify which object-level formulas appear on which side of the sequent arrow, and the ? exponential is used to mark the formulas to which weakening and contraction can be applied.

We shall limit our attention to dealing only with a propositional, intuitionistic object-level logic and proof system. To denote first-order object-level formulas, we will reuse the binary, infix symbols $\wedge$, $\vee$, and $\supset$ at type $bool \to bool \to bool$ (although these were used in, for example, Chapter 4 at a different type, there will be no confusion in this section since we use linear logic connectives for the meta-logic).

Figure 7.4 is a specification of intuitionistic logic provability using the above style of sequent encoding for just the connectives $\wedge$ and $\supset$. Expressions displayed as they are in Figure 7.4 are abbreviations for closed formulas: the intended formulas are those that result by applying ! to their universal closure.

$$
\begin{array}{ll}
(\supset R) & \lceil A \supset B \rceil \multimapinv ?\lfloor A \rfloor \parr \lceil B \rceil. \\
(\supset L) & \lfloor A \supset B \rfloor \Leftarrow \lceil A \rceil \multimapinv ?\lfloor B \rfloor. \\
(\wedge R) & \lceil A \wedge B \rceil \multimapinv \lceil A \rceil \multimapinv \lceil B \rceil. \\
(\wedge L_1) & \lfloor A \wedge B \rfloor \multimapinv ?\lfloor A \rfloor. \\
(\wedge L_2) & \lfloor A \wedge B \rfloor \multimapinv ?\lfloor B \rfloor. \\
(\mathit{Initial}) & \lceil B \rceil \parr \lfloor B \rfloor. \\
(\mathit{cut}) & \bot \multimapinv ?\lfloor B \rfloor \Leftarrow \lceil B \rceil.
\end{array}
$$

Figure 7.4: The *LJ* specification of a sequent calculus for intuitionistic logic.

$$
\frac{\Gamma, A \supset B \longrightarrow B \quad \Gamma, A \supset B, B \longrightarrow E}{\Gamma, A \supset B \longrightarrow E} \supset L \qquad \frac{A, \Gamma \longrightarrow B}{\Gamma \longrightarrow A \supset B} \supset R
$$

$$
\frac{\Gamma, A \longrightarrow E}{\Gamma, A \wedge B \longrightarrow E} \wedge L \qquad \frac{\Gamma, B \longrightarrow E}{\Gamma, A \wedge B \longrightarrow E} \wedge L \qquad \frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B} \wedge R
$$

$$
\frac{}{\Gamma, B \longrightarrow B}\ \mathit{Initial} \qquad \frac{\Gamma \longrightarrow C \quad C, \Gamma \longrightarrow B}{\Gamma \longrightarrow B}\ \mathit{cut}
$$

Figure 7.5: The inference rules encoded using *LJ*

Let *LJ* be the set of clauses displayed in Figure 7.4 and let Σ_1 be the set of constants containing object-logical connectives $\supset$ and $\wedge$ along with the two predicates $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$.

We now examine the synthetic inference rules that result from using the *decide* ! rule with a formula in *LJ*. Let Γ be a multiset of object-level formulas (terms of type *bool*) and let $\lfloor \Gamma \rfloor$ be the multiset $\{\lfloor B \rfloor \mid B \in \Gamma\}$. The synthetic inference rule resulting from using *decide* ! with the $(\supset R)$ clause in *LJ* is

$$
\frac{\Sigma_1 : LJ; \cdot \vdash \lceil B \rceil; \lfloor A \rfloor, \lfloor \Gamma \rfloor}{\Sigma_1 : LJ; \cdot \vdash \lceil A \supset B \rceil; \lfloor \Gamma \rfloor}\ .
$$

Thus, this synthetic inference rule captures exactly the object-level inference: that is, proving the object-level sequent $\Gamma \longrightarrow A \supset B$ has been successfully reduced to proving the sequent $A, \Gamma \longrightarrow B$ (see the $\supset R$ rule in Figure 7.5).

It is simple matter to compute the synthetic inference rule that arises from using *decide* ! on the (*cut*) clause, namely,

$$
\frac{\Sigma_1 : LJ; \cdot \vdash \lceil C \rceil; \mathcal{L} \qquad \Sigma_1 : LJ; \cdot \vdash \lceil B \rceil; \lfloor C \rfloor, \mathcal{L}}{\Sigma_1 : LJ; \cdot \vdash \lceil B \rceil; \mathcal{L}}\ .
$$

This meta-level synthetic rule captures the object-level inference rule called *cut* in Figure 7.5. Note that the occurrence of $\Leftarrow$ in the specification of (cut) is important here: consider the following modification of the specification of the object-level cut inference rule.

$$(cut') \quad \bot \multimapinv ?\lfloor B \rfloor \multimapinv \lceil B \rceil.$$

There are two synthetic inference rules that result in using *decide*! on this formula, namely, the one display above as well as the following.

$$\frac{\Sigma_1 : LJ; \cdot \vdash \lceil B \rceil, \lceil C \rceil; \mathcal{L} \qquad \Sigma_1 : LJ; \cdot \vdash \cdot; \lfloor C \rfloor, \mathcal{L}}{\Sigma_1 : LJ; \cdot \vdash \lceil B \rceil; \mathcal{L}}$$

This additional synthetic rule correspond to the following object-level inference rule.

$$\frac{\Gamma \longrightarrow B, C \qquad C, \Gamma \longrightarrow \cdot}{\Gamma \longrightarrow B}$$

In other words, the specification of (cut') is not able to specify that the occurrence of B on the right in the conclusion should be moved only to the right side of the right premise of the cut rule. It is possible to prove that if B moves to the right-side of the left premise, then that left premise will not ultimately be provable. None-the-less, we wish to have exactly one synthetic inference rule arising from our meta-level specification of the cut rule. Hence, the (cut) rule and the $(\supset L)$ rules both have occurrences of $\Leftarrow$. Recall that the first of the reflections in Section 6.1 points out that both (cut) and $(\supset L)$ are different from other sequent calculus rules: in *LJ*, that difference is captured in by the use of $\Leftarrow$ instead of $\multimapinv$ in the specification of these two rules (see also Proposition 4.2).

7.8 Bibliographic notes

The example of Lolli logic programs in Sections 7.3, 7.4, and 7.5 are taken from [Hodas and Miller, 1994]. The examples of Forum logic programs in Sections 7.6 and 7.7 are taken from [Miller, 1996]. The analysis of object-level sequent systems using linear logic as a meta-theory can be significantly extended beyond what is in Section 7.7: see, for example, [Miller and Pimentel, 2004, 2013; Nigam et al., 2014].

It is not surprising that a programming language directly exploiting proof theory ideas and techniques can be used to implement a sequent calculus (as in Section 7.7) and a theorem prover (as in Section 7.5). We shall see in subsequent chapters (starting with Chapter 12) several other application of linear logic programming in domains that are not overtly connected with logic and proof theory.

Linear logic programming has found useful applications in the parsing of natural language sentences. In particular, both Pareschi and Miller [1990] and Hodas [1994, 1999] have shown how phenomena such as *gap threading* can be captured, at least in part, by linear logic specifications such as those provided by Lolli.

Chapter 10

Collection analysis for Horn clauses

In this chapter we use both proof theory and linear logic to provide a certain kind of *static checking*—called *collection analysis* here—of Horn clause logic programs.

10.1 Introduction

Static analysis of logic programs can provide useful information for programmers and compilers. Type checking, an example of a static analysis, is valuable during the development of code since type errors often represent program errors that are caught at compile time when they are easier to find and fix than at runtime when they are much harder to locate. Static type information also provides valuable documentation of code since it provides a concise approximation to what the code does.

To illustrate an example of what is called *collection analysis*, consider a Horn clause specification of list sorting that maintains duplicates of elements (see, for example, Figure 5.6). Part of the correctness of a sort program includes the fact that if the atomic formula $(sort\ t\ s)$ is provable, then s is a permutation of t that is in-order. The proof of such a property is likely to involve inductive arguments requiring the invention of invariants: in other words, this is not likely to be a property that can be inferred statically during compile time. On the other hand, if the lists t and s are approximated by multisets (that is, if we forget the order of items in lists), then it might be possible to establish that if the atomic formula $(sort\ t\ s)$ is provable, then the multiset associated to s is equal to the multiset associated to t. If that is so, then it is immediate that the lists t and s are, in fact, permutations of one another (in other words, no elements were dropped, duplicated, or created during sorting). As we shall see, such properties based on using multisets to approximate lists can often be established statically. As a result, at least part of the correctness of the sort specification can be established automatically. Besides lists, other data structures, such as trees, can be approximated by various kinds of collections of the items that they contain. Such approximations can be used to provide partial correctness properties of Horn clause logic programs.

We present a scheme by which such collection analysis can be structured and automated. Central to this scheme is the use of linear logic as a computational logic underlying the logic of Horn clauses.

10.2 The undercurrents

There are various themes that underlie our approach to inferring properties of Horn clause programs. We list them explicitly below. The rest of this chapter can be seen as a particular example of how these themes can be developed.

10.2.1 If typing is important, why use only one type system?

Types and other static properties of programming languages have proved important on several levels. Typing can be useful for programmers: they can offer important invariants and document for code. Static analysis can also be used by compilers to uncover useful structures that allow compilers to make choices that can improve execution. While compilers might make use of multiple static analysis regimes, programmers do not usually have convenient access to multiple static analyzes for the code that they are composing. Sometimes, a programming language provides no static analysis, as is the case with Lisp and Prolog. Other programming languages offer exactly one typing discipline, such as the polymorphic typing disciplines of Standard ML and λProlog (SML also statically determines if a given function defined over concrete data structures cover all possible input values). It seems clear, however, that such analysis of code, if it can be done quickly and incrementally, might have significant benefits for programmers during the process of writing code. For example, a programmer might find it valuable to know that a recursive program that she has just written has linear or quadratic runtime complexity, or that a relation she just specified actually defines a function. Having an open set of properties and analysis tools is an interesting direction for the design of a programming language. The collection analysis we discuss here could be just one such analysis tool.

10.2.2 Viewing constants and variables as one

The inference rule of $\forall$-generalization states that if B is provable then $\forall x.B$ is provable (with appropriate provisos if the proof of B depends on hypotheses). If we are in a first-order logic, then the free first-order variable x of B becomes bound in $\forall x.B$ by this inference rule.

Observe the following two things about this rule. First, if we are in an untyped setting, then we can, in principle, quantify over any variable in any expression, even those that play the role of predicates or functions. Mixing such rich abstractions with logic is well known to be inconsistent so when we propose such rich abstractions in logic, we must accompany it with some discipline (such as typing) that will yield consistency.

Second, we need to observe that differences between constants and variables can be seen as one of "scope," at least from a syntactic, proof theoretic, and computational point of view. For example, variables are intended as syntactic objects that can "vary." During the computation of, say, the relation of appending lists, universal quantified variables surrounding Horn clauses change via substitution (via backchaining and unification) but the constructors for the empty and non-empty lists as well as the symbol denoting the append relation do not change and, hence, can be seen as constants. But from a compiling and linking point-of-view, the append predicate might be considered something that varies: if append is in a module of Prolog that is separately compiled, the append symbol might denote a particular object in the compiled code that is later changed when the code is loaded and linked. In a similar fashion, we shall allow ourselves to instantiate constants with expression during static analysis.

Substituting for constants allows us to "split the atom:" that is, by substituting for the predicate p in the atom $p(t_1, \ldots, t_n)$, we replace that atom with a formula, which, in this chapter, will be a linear logic formula.

10.2.3 Linear logic underlies computational logic

Linear logic [Girard, 1987] is able to explain the proof theory of usual Horn clause logic programming (and even richer logic programming languages [Hodas and Miller, 1994]). It is also able to provide means to reason about resources, such as items in multisets and sets. Thus, linear logic will allow us to sit within one declarative framework to describe both usual logic programming as well as "sub-atomic" reasoning about the resources implicit in the arguments of predicates.

10.3 Abstraction and substitution in proof theory

A sequent is a triple of the form $\Sigma : \Gamma \vdash \Delta$ were Σ, the signature, is a list of non-logical constants and eigenvariables paired with a simple type, and where both Γ and Δ are multisets of Σ-formulas (i.e., formulas all of whose non-logical symbols are in Σ). The rules for linear logic are the standard ones [Girard, 1987], except here signatures have been added to sequents. The rules for quantifier introduction are the only rules that require the signature and they are reproduced here:

$$\frac{\Sigma, y:\tau; B[y/x], \Gamma \vdash \Delta}{\Sigma; \exists x_\tau.B, \Gamma \vdash \Delta}\ \exists L \qquad \frac{\Sigma \vdash t:\tau \quad \Sigma; \Gamma \vdash B[t/x], \Delta}{\Sigma; \Gamma \vdash \exists x_\tau.B, \Delta}\ \exists R$$

$$\frac{\Sigma \vdash t:\tau \quad \Sigma; B[t/x], \Gamma \vdash \Delta}{\Sigma; \forall x_\tau.B, \Gamma \vdash \Delta}\ \forall L \qquad \frac{\Sigma, y:\tau; \Gamma \vdash B[y/x], \Delta}{\Sigma; \Gamma \vdash \forall x_\tau.B, \Delta}\ \forall R$$

The premise $\Sigma \vdash t : \tau$ is the judgment that the term t has the (simple) type τ given the typing declaration contained in Σ.

We now outline three ways to instantiate items appearring within the sequent calculus.

10.3.1 Substituting for types

Although we think of formulas and proofs as untyped expressions, we shall use simple typing within sequents to control the kind of formulas that are present. A signature is used to bind and declare typing for (eigen)variables and non-logical constants within a sequent. Simple types are, formally speaking, also a simple class of untyped λ-terms: the type o is used to denote formulas (following Church [Church, 1940]). In a sequent calculus proof, simple type expressions are global and admit no bindings. As a result, it is an easy matter to show that if one takes a proof with a type constant σ and replaces everywhere σ with some type, say, τ, one gets another valid proof. We shall do this later when we replace a list by a multiset that approximates it: since we are using linear logic, we shall use formulas to encode multisets and so we shall replace the type constant `list` with `o`.

10.3.2 Substituting for non-logical constants

Consider the sequent

$$\Sigma, p : \tau; !\, D_1, !\, D_2, !\, \Gamma \vdash p(t_1, \ldots, t_m)$$

where the type τ is a predicate type (that is, it is of the form $\tau_1 \to \cdots \to \tau_m \to o$) and where p appears in, say, D_1 and D_2 and in no formula of Γ. The linear logic exponential ! is used here to encode the fact that the formulas

D_1 and D_2 are available for arbitrary reuse within a proof (the usual case for program clauses). Using the right introduction rules for implication and the universal quantifier, it follows that the sequent

$$\Sigma; !\,\Gamma \vdash \forall p[D_1 \Rightarrow D_2 \Rightarrow p(t_1, \ldots, t_m)]$$

is also provable. Since this is a universal quantifier, there must be proofs for all instances of this quantifier. Let θ be the substitution $[p \mapsto \lambda x_1 \ldots \lambda x_m.S]$, where S is a term over the signature $\Sigma \cup \{x_1, \ldots, x_m\}$ of type o. A consequence of the proof theory of linear logic is that there is a proof also of

$$\Sigma; !\,\Gamma \vdash D_1\theta \Rightarrow D_2\theta \Rightarrow S[t_1/x_1, \ldots, t_m/x_m]$$

and of the sequent

$$\Sigma; !\,D_1\theta, !\,D_2\theta, !\,\Gamma \vdash S[t_1/x_1, \ldots, t_m/x_m].$$

As this example illustrates, it is possible to instantiate a predicate (here p) with an abstraction of a formula (here, $\lambda x_1 \ldots \lambda x_m.\ S$). Such instantiation carries a provable sequent to a provable sequent.

10.3.3 Substituting for assumptions

An instance of the cut-rule (mentioned earlier) is the following:

$$\frac{\Sigma; \Gamma_1 \vdash B \qquad \Sigma; B, \Gamma_2 \vdash C}{\Sigma; \Gamma_1, \Gamma_2 \vdash C}$$

This inference rule (especially when associated with the cut-elimination procedure) provides a way to merge (substitution) the proof of a formula (here, B) with a use of that formula as an assumption. For example, consider the following situation. Given the example in the Section 10.3.2, assume that we can prove

$$\Sigma; !\,\Gamma \vdash !\,D_1\theta \quad \text{and} \quad \Sigma; !\,\Gamma \vdash !\,D_2\theta.$$

Using two instances of the cut rule and the proofs of these sequent, it is possible to obtain a proof of the sequent

$$\Sigma; !\,\Gamma \vdash S[t_1/x_1, \ldots, t_m/x_m]$$

(contraction on the left for !'ed formulas must be applied).

Thus, by a series of instantiations of proofs, it is possible to move from a proof of, say,

$$\Sigma, p : \tau; !\,D_1, !\,D_2, !\,\Gamma \vdash p(t_1, \ldots, t_m)$$

to a proof of

$$\Sigma; !\,\Gamma \vdash S[t_1/x_1, \ldots, t_m/x_m].$$

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

split X nil nil nil.
split X (A::R) (A::S) B :- leq A X, split X R S B.
split X (A::R) S (A::B) :- gr  A X, split X R S B.
sort nil nil.
sort (F::R) S:- split F R Sm B, sort Sm SS, sort B BS,
   append SS (F::BS) S.
```

Figure 10.1: Some Horn clauses for specifying a sorting relation.

$$\begin{gathered}
\forall K(\bot \parr K \multimapboth K)\\
\forall X, L, K, M(L \parr K \multimapboth M) \Rightarrow (\mathit{item}\ X \parr L \parr K \multimapboth \mathit{item}\ X \parr M)\\
\forall X(\bot \parr \bot \multimapboth \bot)\\
\forall X, A, B, R, S.(S \parr B \multimapboth R) \Rightarrow \mathbf{1} \Rightarrow (\mathit{item}\ A \parr S \parr B \multimapboth \mathit{item}\ A \parr R)\\
\forall X, A, B, R, S.(S \parr B \multimapboth R) \Rightarrow \mathbf{1} \Rightarrow (S \parr \mathit{item}\ A \parr B \multimapboth \mathit{item}\ A \parr R)\\
(\bot \multimapboth \bot)\\
\forall F, R, S, Sm, Bg, SS, BS.(Sm \parr B \multimapboth R) \,\&\, (Sm \multimapboth SS) \,\&\, (B \multimapboth BS) \,\&\\
(SS \parr \mathit{item}\ F \parr BS \multimapboth S) \Rightarrow (\mathit{item}\ F \parr R \multimapboth S)
\end{gathered}$$

Figure 10.2: The result of instantiating various non-logical constants in the above Horn clauses.

We shall see this style of reasoning about proofs several times below. This allows us to "split an atom" $p(t_1, \ldots, t_m)$ into a formula $S[t_1/x_1, \ldots, t_m/x_m]$ and to transform proofs of the atom into proofs of that formula. In what follows, the formula S will be a linear logic formula that provides an encoding of some judgment about the data structures encoded in the terms $t_1, \ldots, t_m$.

10.4 Multisets approximations

A *multiset expression* is a formula in linear logic built from the predicate symbol *item* (denoting the singleton multiset), the linear logic multiplicative disjunction $\parr$ (for multiset union), and the unit $\bot$ for $\parr$ (used to denote the empty multiset). We shall also allow a predicate variable (a variable of type o) to be used to denote a (necessarily open) multiset expression. An example of an open multiset expression is $\mathit{item}\ f(X) \parr \bot \parr Y$, where Y is a variable of type o, X is a first-order variable, and f is some first-order term constructor.

Let S and T be two multiset expressions. The two *multiset judgments* that we wish to capture are multiset inclusion, written as $S \sqsubseteq T$, and equality, written as $S \overset{m}{=} T$. We shall use the syntactic variable ρ to range over these two judgments, which are formally binary relations of type $o \to o \to o$. A *multiset statement* is a formula of the form

$$\forall \bar{x}[S_1 \; \rho_1 \; T_1 \,\&\, \cdots \,\&\, S_n \; \rho_n \; T_n \Rightarrow S_0 \; \rho_0 \; T_0]$$

where the quantified variables $\bar{x}$ are either first-order or of type o and formulas $S_0, T_0, \ldots, S_n, T_n$ are possibly open multiset expressions.

If S and T are closed multiset expressions, then we write $\models_m S \sqsubseteq T$ whenever the multiset (of closed first-order terms) denoted by S is contained in the multiset denoted by T, and we write $\models_m S \overset{m}{=} T$ whenever the multisets denoted by S and T are equal. Similarly, we write

$$\models_m \forall \bar{x}[S_1 \; \rho_1 \; T_1 \,\&\, \cdots \,\&\, S_n \; \rho_n \; T_n \Rightarrow S_0 \; \rho_0 \; T_0]$$

if for all closed substitutions θ such that $\models_m S_i\theta \; \rho_i \; T_i\theta$ for all $i = 1, \ldots, n$, it is the case that $\models_m S_0\theta \; \rho_0 \; T_0\theta$.

The following Proposition is central to our use of linear logic to establish multiset statements for Horn clause programs.

Proposition 10.1. *Let $S_0, T_0, \ldots, S_n, T_n$ $(n \geq 0)$ be multiset expressions all of whose free variables are in the list of variables $\bar{x}$. For each judgment $s \; \rho \; t$ we write $s \; \hat{\rho} \; t$ to denote $\exists q(s \parr q \multimap t)$ if ρ is $\sqsubseteq$ and $t \multimapboth s$ if ρ is $\overset{m}{=}$. If*

$$\forall \bar{x}[S_1 \; \hat{\rho}_1 \; T_1 \,\&\, \ldots \,\&\, S_n \; \hat{\rho}_n \; T_n \Rightarrow S_0 \; \hat{\rho}_0 \; T_0]$$

is provable in linear logic, then

$$\models_{ms} \forall \bar{x}[S_1 \; \rho_1 \; T_1 \,\&\, \cdots \,\&\, S_n \; \rho_n \; T_n \Rightarrow S_0 \; \rho_0 \; T_0]$$

This proposition shows that linear logic can be used in a sound way to infer valid multiset statement. On the other hand, the converse (completeness) does not hold: the statement

$$\forall x \forall y.(x \sqsubseteq y) \,\&\, (y \sqsubseteq x) \Rightarrow (x \overset{m}{=} y)$$

is valid but its translation into linear logic is not provable.

To illustrate how deduction in linear logic can be used to establish the validity of a multiset statement, consider the first-order Horn clause program in Figure 10.1. The signature for this collection of clauses can be given as follows:

```
split X nil nil nil.
split X (X::R) S B :- split X R S B.
split X (A::R) (A::S) B :- lt A X, split X R S B.
split X (A::R) S (A::B) :- gr A X, split X R S B.
```

Figure 10.3: A change in the specification of splitting lists to drop duplicates.

$$\begin{array}{l}
\forall X.(?\,\mathbf{0} \multimap ?(\mathit{item}\ X \oplus \mathbf{0} \oplus \mathbf{0}))\\
\forall X, B, R, S.\ (?\,R \multimap ?(\mathit{item}\ X \oplus S \oplus B)) \Rightarrow\\
\qquad (?(\mathit{item}\ X \oplus R) \multimap ?(\mathit{item}\ X \oplus S \oplus B))\\
\forall X, A, B, R, S.\ \mathbf{1}\ \&\ (?\,R \multimap ?(\mathit{item}\ X \oplus S \oplus B)) \Rightarrow\\
\qquad (?(\mathit{item}\ A \oplus R) \multimap ?(\mathit{item}\ X \oplus \mathit{item}\ A \oplus S \oplus B))\\
\forall X, A, B, R, S.\ \mathbf{1}\ \&\ (?\,R \multimap ?(\mathit{item}\ X \oplus S \oplus B)) \Rightarrow\\
\qquad (?(\mathit{item}\ A \oplus R) \multimap ?(\mathit{item}\ X \oplus S \oplus \mathit{item}\ A \oplus B))
\end{array}$$

Figure 10.4: The result of substituting set approximations into the `split` program.

```
type nil      list
type ::       int -> list -> list
type append   list -> list -> list -> o
type split    int -> list -> list -> list -> o
type sort     list -> list -> o
type leq      int -> int -> o
type gr       int -> int -> o
```

The first two declarations provide constructors for empty and non-empty lists, the next three are predicates whose Horn clause definition is presented in Figure 10.1, and the last two are order relations that are apparently defined elsewhere.

If we think of lists as collections of items, then we might want to check that the sort program as written does not drop, duplicate, or create any elements. That is, if the atom (`sort` s t) is provable then the multiset of items in the list denoted by s is equal to the multiset of items in the list denoted by t. If this property holds then t and s are lists that are permutations of each other: of course, this does not say that it is the correct permutation but this more simple fact is one that, as we show, can be inferred automatically.

Computing this property of our example logic programming follows the following three steps.

First, we provide an approximation of lists as being, in fact, multiset: more precisely, as *formulas* denoting multisets. The first step, therefore, must be to substitute `o` for `list` in the signature above. Now we can now interpret the constructors for lists using the substitution

$$\texttt{nil} \mapsto \bot \qquad \texttt{::} \mapsto \lambda x \lambda y.\ \mathit{item}\ x \parr y.$$

Under such a mapping, the list `(1::3::2::nil)` is mapped to the multiset expression $\mathit{item}\ 1 \parr \mathit{item}\ 3 \parr \mathit{item}\ 2 \parr \bot$.

Second, we associate with each predicate in Figure 10.1 a multiset judgment that encodes an invariant concerning the multisets denoted by the predicate's arguments. For example, if ($\texttt{append}\ r\ s\ t$) or ($\texttt{split}\ u\ t\ r\ s$) is provable then the multiset union of the items in r with those in s is equal to the multiset of items in t, and if ($\texttt{sort}\ s\ t$) is provable then the multisets of items in lists s and t are equal. This association of multiset judgments to atomic formulas can be achieved formally using the following substitutions for constants:

$$\begin{aligned} \texttt{append} &\mapsto \lambda x \lambda y \lambda z.\ (x \parr y) \multimapboth z \\ \texttt{split} &\mapsto \lambda u \lambda x \lambda y \lambda z.\ (y \parr z) \multimapboth x \\ \texttt{sort} &\mapsto \lambda x \lambda y.\ x \multimapboth y \end{aligned}$$

The predicates `leq` and `gr` (for the least-than-or-equal-to and greater-than relations) make no statement about collections of items, so that they can be mapped to a trivial tautology via the substitution

$$\texttt{leq} \mapsto \lambda x \lambda y.\ \mathbf{1} \qquad \texttt{gr} \mapsto \lambda x \lambda y.\ \mathbf{1}$$

Figure 10.2 presents the result of applying these mappings to Figure 10.1.

Third, we must now attempt to prove each of the resulting formulas. In the case of Figure 10.2, all the displayed formulas are trivial theorems of linear logic.

Having taken these three steps, we now claim that we have proved the intended collection judgments associate to each of the logic programming predicates above: in particular, we have now shown that our particular sort program computes a permutation.

10.5 Formalizing the method

The formal correctness of this three stage approach is easily justified given the substitution properties we presented in Section 10.3 for the sequent calculus presentation of linear logic.

Let Γ denote a set of formulas that contains those in Figure 10.1. Let θ denote the substitution described above for the type `list`, for the constructors `nil` and `cons`, and for the predicates in Figure 10.1. If Σ is the signature for Γ then split Σ into the two signatures Σ_1 and Σ_2 so that Σ_1 is the domain of the substitution θ and let Σ_3 be the signature of the range of θ (in this case, it just contains the constant *item*). Thus, $\Gamma\theta$ is the set of formula in Figure 10.2.

Assume now that $\Sigma_1, \Sigma_2; \Gamma \vdash sort(t, s)$ is provable. Given the discussion in Sections 10.3.1 and 10.3.2, we know that

$$\Sigma_1, \Sigma_3; \Gamma\theta \vdash t\theta \multimapboth s\theta$$

is provable. Since the formulas in $\Gamma\theta$ are provable, we can use substitution into proofs (Section 10.3.3) to conclude that $\Sigma_1, \Sigma_3; \vdash t\theta \multimapboth s\theta$. Given Proposition 10.1, we can conclude that $\models_m t\theta \overset{m}{=} s\theta$: that is, that $t\theta$ and $s\theta$ encode the same multiset.

Consider the following model theoretic argument for establishing similar properties of Horn clauses. Let $\mathcal{M}$ be the Herbrand model that captures the invariants that we have in mind. In particular, $\mathcal{M}$ contains the atoms (`append` $r\ s\ t$) and (`split` $u\ t\ r\ s$) if the items in the list r added to the items in list s are the same as the items in t. Furthermore, $\mathcal{M}$ contains all closed atoms of the form (`leq` $t\ s$) and (`gr` $t\ s$), and closed atoms (`sort` $s\ t$) where s and t are lists that are permutations of one another. One can now show that $\mathcal{M}$ satisfies all the Horn clauses in Figure 10.1. As a consequence of the soundness of first-order classical logic, any atom provable from the clauses in Figure 10.1, must be true in $\mathcal{M}$. By construction of $\mathcal{M}$, this means that the desired invariant holds for all atoms proved from the program.

The approach suggested here using linear logic and deduction remains syntactic and proof theoretic: in particular, showing that a model satisfies a Horn clause is replaced by a deduction within linear logic.

10.6 Sets approximations

It is rather easy to encode sets and the equality and subset judgments on sets into linear logic. In fact, the transition to set from multiset is provided by the use of the linear logic exponential: since we are using disjunctive encoding of collections (see the discussion in Section 7.1), we use the ? exponential (if we were using the conjunctive encoding, we would use the ! exponential).

The expression $?\,item\ t$ can be seen as describing the presence of an item for which the exact multiplicity does not matter: this formula represents the capacity to be used any number of times. Thus, the set $\{x_1, \ldots, n_n\}$ can be encoded as $?\,item\ x_1 \parr \cdots \parr ?\,item\ x_n$. Using logical equivalences of linear logic, this formula is also equivalent to the formula $?(item\ x_1 \oplus \cdots \oplus item\ x_n)$. This latter encoding is the one that we shall use for building our encoding of sets.

A *set expression* is a formula in linear logic built from the predicate symbol *item* (denoting the the singleton set), the linear logic additive disjunction $\oplus$

(for set union), and the unit $\mathbf{0}$ for $\oplus$ (used to denote the empty set). We shall also allow a predicate variable (a variable of type o) to be used to denote a (necessarily open) set expression. An example of an open multiset expression is $(\mathit{item}\ f(X)) \oplus \mathbf{0} \oplus Y$, where Y is a variable of type o, X is a first-order variable, and f is some first-order term constructor.

Let S and T be two set expressions. The two *set judgments* that we wish to capture are set inclusion, written as $S \subseteq T$, and equality, written as $S \overset{s}{=} T$. We shall use the syntactic variable ρ to range over these two judgments, which are formally binary relations of type $o \to o \to o$. A *set statement* is a formula of the form

$$\forall \bar{x}[S_1\ \rho_1\ T_1\ \&\ \cdots\ \&\ S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0]$$

where the quantified variables $\bar{x}$ are either first-order or of type o and formulas $T_0, S_0, \ldots, T_n, S_n$ are possibly open set expressions.

If S and T are closed set expressions, then we write $\models_s S \subseteq T$ whenever the set (of closed first-order terms) denoted by S is contained in the set denoted by T, and we write $\models_s S \overset{s}{=} T$ whenever the sets denoted by S and T are equal. Similarly, we write

$$\models_s \forall \bar{x}[S_1\ \rho_1\ T_1\ \&\ \cdots\ \&\ S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0]$$

if for all closed substitutions θ such that $\models_s S_i\theta\ \rho_i\ T_i\theta$ for all $i = 1, \ldots, n$, it is the case that $\models_s S_0\theta\ \rho_0\ T_0\theta$.

The following Proposition is central to our use of linear logic to establish set statements for Horn clause programs.

Proposition 10.2. *Let $S_0, T_0, \ldots, S_n, T_n$ $(n \geq 0)$ be set expressions all of whose free variables are in the list of variables $\bar{x}$. For each judgment $s\ \rho\ t$ we write $s\ \hat{\rho}\ t$ to denote $?\,s \multimap ?\,t$ if ρ is $\subseteq$ and $(?\,s \multimap ?\,t)\ \&\ (?\,t \multimap ?\,s)$ if ρ is $\overset{s}{=}$. If*

$$\forall \bar{x}[S_1\ \hat{\rho}_1\ T_1\ \&\ \ldots\ \&\ S_n\ \hat{\rho}_n\ T_n \Rightarrow S_0\ \hat{\rho}_0\ T_0]$$

is provable in linear logic, then

$$\models_s \forall \bar{x}[S_1\ \rho_1\ T_1\ \&\ \cdots\ \&\ S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0]$$

Lists can be approximated by sets by using the following substitution:

$$\texttt{nil} \mapsto \mathbf{0} \qquad \texttt{::} \mapsto \lambda x \lambda y.\ \mathit{item}\ x \oplus y.$$

Under such a mapping, the list `(1::2::2::nil)` is mapped to the set expression $\mathit{item}\ 1 \oplus \mathit{item}\ 2 \oplus \mathit{item}\ 2 \oplus \mathbf{0}$. This expression is equivalent ($\multimapboth$) to the set expression $\mathit{item}\ 1 \oplus \mathit{item}\ 2$.

For a simple example of using set approximates, consider modifying the sorting program provided before so that duplicates are not kept in the sorted

$$\frac{}{\Gamma; A_i \vdash A_1 \oplus \cdots \oplus A_n}\ \oplus\mathrm{R}$$

$$\frac{\Gamma; A_1 \vdash C \quad \ldots \quad \Gamma; A_n \vdash C}{\Gamma; A_1 \oplus \cdots \oplus A_n \vdash C}\ \oplus\mathrm{L}$$

$$\frac{\Gamma; B_1 \oplus \cdots \oplus B_m \vdash C}{\Gamma; A \vdash C}\ \mathrm{BC}$$

Here, $n, m \geq 0$ and in the BC (backchaining) inference rule, the formula $?(A_1 \oplus \cdots \oplus A_n) \multimap ?(B_1 \oplus \cdots \oplus B_m)$ must be a member of Γ and $A \in \{A_1, \ldots, A_n\}$.

Figure 10.5: Specialized proof rules for proving set statements.

list. Do this modification by replacing the previous definition for splitting a list with the clauses in Figure 10.3. That figure contains a new definition of splitting that contains three clauses for deciding whether or not the "pivot" for the splitting `X` is equal to, less than (using the `lt` predicate), or greater than the first member of the list being split. Using the following substitutions for predicates

$$\begin{aligned}
\texttt{append} &\mapsto \lambda x \lambda y \lambda z.\ ?(x \oplus y) \multimapboth ?\,z \\
\texttt{split} &\mapsto \lambda u \lambda x \lambda y \lambda z.\ ?\,x \multimap ?(\mathit{item}\ u \oplus y \oplus z) \\
\texttt{sort} &\mapsto \lambda x \lambda y.\ ?\,x \multimapboth ?\,y
\end{aligned}$$

(as well as the trivial substitution for `lt` and `ge`), we can show that sort relates two lists only if those lists are approximated by the same set.

In the case of determining the validity of a set statement, the use of linear logic here appears to be rather weak when compared to the large body of results for solving set-based constraint systems [Aiken, 1994; Pacholski and Podelski, 1997].

10.7 Automation of deduction

We describe how automation of proof for the linear logic translations of set and multiset statements given in Propositions 10.1 and 10.2 can be performed.

In order to understand how to automatically prove the required formulas, we first provide a normal form theorem for the fragment of linear logic for which we are interested. The key result of linear logic surrounding the search for cut-free proofs is given by the completeness of *focused proofs* [Andreoli, 1992]. Focused proofs are a normal form that significantly generalizes standard completeness results in logic programming, including the completeness of SLD-resolution and uniform proofs as well as various forms of bottom-up and top-down reasoning.

We first analyze the nature of proof search for the linear logic translation of set statements. Note that when considering provability of set statements, there is no loss of generality if the only set judgment it contains is the subset judgment since set equality can be expressed as two inclusions. We now prove that the proof system in Figure 10.5 is sound and complete for proving set statements.

Proposition 10.3. *Let $S_0, T_0, \ldots, S_n, T_n$ $(n \geq 0)$ be set expressions all of whose free variables are in the list of variables $\bar{x}$. The formula*

$$\forall \bar{x}[(?\, S_1 \multimap ?\, T_1) \,\&\, \ldots \,\&\, (?\, S_n \multimap ?\, T_n) \Rightarrow (?\, S_0 \multimap ?\, T_0)]$$

is provable in linear logic if and only if the sequent

$$(?\, S_1 \multimap ?\, T_1), \ldots, (?\, S_n \multimap ?\, T_n); S_0 \vdash T_0$$

is provable using the proof system in Figure 10.5.

Proof. The soundness part of this proposition ("if") is easy to show. For completeness ("only if"), we use the completeness of focused proofs in [Andreoli, 1992]. In order to use this result of focused proofs, we need to give a polarity to all atomic formulas. We do this by assigning all atomic formulas (those of the form *item* $(\cdot)$ and those symbols in $\bar{x}$ of type o) negative polarity. Second, we need to translation the two sided sequent $\Gamma; S \vdash T$ to $\Gamma^\perp; T \Uparrow S^\perp$ when S is not atomic (that is, its top-level logical connective is $\oplus$) and to $\Gamma^\perp, T; S^\perp \Uparrow \cdot$ when S is a atom. Completeness then follows directly from the structure of focused proofs. □

Notice that the resulting proofs are essentially bottom-up: one reasons from formulas on the left of the sequent arrow to formulas on the right.

We can now conclude that it is decidable to determine whether or not the linear logic translation of a set statement is provable. Notice that in a proof built using the inference rules in Figure 10.5, if the endsequent is $\Gamma; S \vdash T$ then all sequents in the proof have the form $\Gamma; S' \vdash T$, for some S'. Thus, the search for a proof either succeeds (proof search ends by placing $\oplus$ R on top), or fails to find a proof, or it cycles, a case we can always detect since there is only a finite number of atomic formulas that can be S'.

The proof system in Figure 10.6 can be used to characterize the structure of proofs of the linear logic encoding of multiset statements. Let

$$\forall \bar{x}[S_1 \;\hat{\rho}_1\; T_1 \,\&\, \ldots \,\&\, S_n \;\hat{\rho}_n\; T_n \Rightarrow S_0 \;\hat{\rho}_0\; T_0]$$

$$\frac{}{\Gamma; A_1 \parr \cdots \parr A_n \vdash A_1, \ldots, A_n} \ \parr \mathrm{L}$$

$$\frac{\Gamma; S \vdash T_1, T_2, \Delta}{\Gamma; S \vdash T_1 \parr T_2, \Delta} \ \parr \mathrm{R}$$

$$\frac{\Gamma; S \vdash A_1, \ldots, A_n, \Delta}{\Gamma; S \vdash B_1, \ldots, B_m, \Delta} \ \mathrm{BC}$$

Here, $n, m \geq 0$ and in the BC (backchaining) inference rule, it must be the case that the formula

$$(A_1 \parr \cdots \parr A_n) \multimap (B_1 \parr \cdots \parr B_m)$$

is a member of Γ.

Figure 10.6: Specialized proof rules for proving multiset statements.

be the translation of a multiset statement into linear logic. Provability of this formula can be reduced to attempting to prove $S_0 \ \hat{\rho}_0 \ T_0$ from assumptions of the form

$$(A_1 \parr \cdots \parr A_n) \multimap (B_1 \parr \cdots \parr B_m),$$

where $A_1, \ldots, A_n, B_1, \ldots, B_m$ are atomic formulas. Such formulas can be called *multiset rewriting clauses* since backchaining on such clauses amounts to rewriting the right-hand-side multiset of a sequent (see rule BC in Figure 10.6). Such rewriting clauses are particularly simple since they do not involve quantification.

Proposition 10.4. *Let S_0 and T_0 be multiset expressions all of whose free variables are in the list of variables $\bar{x}$ and let Γ be a set of multiset rewriting rules. The formula $S_0 \multimap T_0$ is a linear logic consequence of Γ if and only if the sequent $\Gamma; S_0 \vdash T_0$ is provable using the inference rules in Figure 10.6.*

Proof. The soundness part of this proposition ("if") is easy to show. Completeness ("only if") is proved elsewhere, for example, in [Miller, 1993, Proposition 2]. It is also an easy consequence of the the completeness of focused proofs in [Andreoli, 1992]: fix the polarity to all atomic formulas to be positive. □

Notice that the proofs using the rules in Figure 10.6 are straight line proofs (no branching) and that they are top-down (or goal-directed). Given these observation, it follows that determining if $S_0 \multimap T_0$ is provable from a set of multiset rewriting clauses is decidable, since this problem is contained within the reachability problem of Petri Nets [Esparza and Nielsen, 1994]. Proving a multiset inclusion judgment $\exists q(S_0 \parr q \multimap T_0)$ involves first instantiating this higher-order quantifier. In principle, this instantiation can be delayed until attempting to apply the sole instance of the $\parr$ L rule (Figure 10.6).

10.8 List approximations

We now consider using lists as approximations. Since lists have more structure than sets and multisets, it is more involved to encode and reason with them. We only illustrate their use and do not follow a full formal treatment for them.

Since the order of elements in a list is important, the encoding of lists into linear logic must involve a connective that is not commutative. (Notice that both $\parr$ and $\oplus$ are commutative.) Linear implication provides a good candidate for encoding the order used in lists. For example, consider proof search with the formula

$$\mathit{item}\ a \multimapinv (p \multimapinv (\mathit{item}\ b \multimapinv (p \multimapinv \bot)))$$

on the right. (This formula is equivalent to $\mathit{item}\ a \parr (p^\perp \otimes (\mathit{item}\ b \parr p^\perp))$.) Such a formula can be seen as describing a process that is willing to output the item a then go into input mode waiting for the atomic formula p to appear. If that formula appears, then item b is output and again it goes into input waiting mode looking for p. If another occurrence of p appears, this process becomes the inactive process. Clearly, a is output prior to when b is output: this ordering is faithfully captured by proof search in linear logic.

The example above suggests that lists and list equality can be captured directly in linear logic using the following encoding:

$$\texttt{nil} \mapsto \lambda l.\bot \qquad \texttt{cons} \mapsto \lambda x \lambda R \lambda l.\ \mathit{item}\ x \multimapinv (l \multimapinv (R\ l))$$

The encoding of the list, say ($\texttt{cons}\ a\ (\texttt{cons}\ b\ \texttt{nil})$), is given by the λ-abstraction

$$\lambda l.\mathit{item}\ a \multimapinv (l \multimapinv (\mathit{item}\ b \multimapinv (l \multimapinv \bot))).$$

The following proposition can be proved by induction on the length of the list t.

Proposition 10.5. *Let s and t be two lists (built using* `nil` *and* `cons`*) and let S and T be the translation of those lists into expressions of type $o \to o$ via the substitution above. Then $\forall l.(Sl) \multimapboth (Tl)$ is provable in linear logic if and only if s and t are the same list.*

This presentation of lists can be "degraded" to multisets simply by applying the translation of a list to the formula $\bot$. For example, applying the translation of ($\texttt{cons}\ a\ (\texttt{cons}\ b\ \texttt{nil})$) to $\bot$ yields the formulas

$$\mathit{item}\ a \multimapinv (\bot \multimapinv (\mathit{item}\ b \multimapinv (\bot \multimapinv \bot)))$$

```
traverse emp null.
traverse (bt N emp R) (N::S) :- traverse R S.
traverse (bt N (bt M L1 L2) R) S :-
  traverse (bt M L1 (bt N L2 R)) S.
```

Figure 10.7: Traversing a binary tree to produce a list.

$$
\begin{gathered}
\forall W.\forall w.W w \multimapboth W w \\
\forall N \forall R \forall S \forall W \forall w.(\mathit{item}\ N \multimapinv (w \multimapinv R\ W\ \ w)) \multimapboth \\
(\mathit{item}\ N \multimapinv (w \multimapinv S\ W\ \ w)) \multimapinv (\forall W \forall w.R\ W\ \ w \multimapboth S\ W\ \ w) \\
\forall N \forall M \forall L_1 \forall L_2 \forall R \forall S \forall W \forall w \\
L_1(\lambda k.\mathit{item} M \multimapinv (k \multimapinv L_2(\lambda l.\mathit{item} N \multimapinv (l \multimapinv R\ W\ \ l))k))w \multimapboth S\ W\ \ w \multimapinv \\
\forall W \forall w.L_1(\lambda k.\mathit{item} M \multimapinv (k \multimapinv L_2(\lambda l.\mathit{item} N \multimapinv (l \multimapinv R\ W\ \ l))k))w \multimapboth S\ W\ \ w
\end{gathered}
$$

Figure 10.8: Linear logic formulas arising from a difference list approximation.

which is linear logically equivalent to *item a* ⅋ *item b*.

Given this presentation of lists, there appears to be no simple combinator for, say, list concatenation and, as a result, there is no direct way to express the judgments of prefix, suffix, sublist, etc. Thus, beyond equality of lists (by virtual of Proposition 10.5) there are few natural judgments that can be stated for list. More can be done, however, by considering difference lists.

10.9 Difference list approximations

Since our framework includes λ-abstractions, it is natural to represent difference lists as a particular kind of list abstraction over a list. For example, in λProlog a difference list is naturally represented as a λ-term of the form

$$\lambda L.\texttt{cons}\ x_1\ (\texttt{cons}\ x_2\ (\ldots(\texttt{cons}\ x_n\ L)\ldots)).$$

Such abstracted lists are appealing since the simple operation of composition encodes the concatenation of two lists. Given concatenation, it is then easy to encode the judgments of prefix and suffix. To see other example of computing on difference lists described in fashion, see [Brisset and Ridoux, 1991].

Lists can be encoded using the difference list notion with the following mapping into linear logic formulas.

$$\texttt{nil} \mapsto \lambda L \lambda l.\ L\ l$$

$$\texttt{cons} \mapsto \lambda x \lambda R \lambda L \lambda l.\ \mathit{item}\ x \multimapinv (l \multimapinv (R\ L\ l))$$

The encoding of the list, say (`cons` a (`cons` b `nil`)), is given by the λ-abstraction

$$\lambda L \lambda l.\mathit{item}\ a \multimapinv (l \multimapinv (\mathit{item}\ b \multimapinv (l \multimapinv L\ l))).$$

In Figure 10.7, a predicate for traversing a binary tree is given. Binary trees are encoded using the type `btree` and are constructed using the constructors `emp`, for the empty tree, and `bt` of type `int` $\rightarrow$ `btree` $\rightarrow$ `btree` $\rightarrow$ `btree`, for building non-empty trees. A useful invariant of this program is that the list of items approximating the binary tree structure in the first argument of `traverse` is equal to the list of items in the second argument. Linear logic formulas for computing that approximation can be generated using the following approximating substitution.

$$\begin{aligned}
\texttt{btree} &\mapsto \texttt{o}\\
\texttt{emp} &\mapsto \lambda L \lambda l.\ L\ l\\
\texttt{bt} &\mapsto \lambda x \lambda R \lambda S \lambda L \lambda l.(R\ (\lambda l.\mathit{item}\ x \multimapinv (l \multimapinv (S\ L\ l)))\ l))
\end{aligned}$$

The result of applying that substitution (as well as the one above for `nil` and `cons`) is displayed in Figure 10.8. While these formulas appear rather complex, they are all, rather simple theorems of higher-order linear logic: these theorems are essentially trivial since the $\beta\eta$-conversions used to build the formulas from the data structures has done all the essential work in organizing the items into a list. Establishing these formulas proves that the order and multiplicity of elements in the binary tree and in the list in a provable traverse computation are the same.

10.10 Future work

Various extensions of the basic scheme described here are natural to consider. In particular, it should be easy to consider approximating data structures that contain items of differing types: each of these types could be mapped into different $\mathit{item}_\alpha(\cdot)$ predicates, one for each type α.

It should also be simple to construct approximating mappings given the *polymorphic* typing of a given constructor's type. For example, if we are given the following declaration for binary tree (written here in λProlog syntax),

```
kind btree    type -> type.
type emp      btree A.
type bt       A -> btree A -> btree A -> btree A.
```

it should be possible to automatically construct the mapping

$$\texttt{btree} \mapsto \lambda x.\texttt{o}$$

$$\begin{aligned}\texttt{emp} &\mapsto \bot\\ \texttt{bt} &\mapsto \lambda x \lambda y \lambda z.item_A(x) \parr x \parr y\end{aligned}$$

that can, for example, approximate a binary tree with the multiset of the labels for internal nodes.

Abstract interpretation [Cousot and Cousot, 1977] can associate to a program an approximation to its semantics. Such approximations can help to determine various kinds of properties of programs. It will be interesting to see how well the particular notions of collection analysis can be related to abstract interpretation. More challenging would be to see to what extent the general methodology described here – the substitution into proofs (computation traces) and use of linear logic – can be related to the very general methodology of abstract interpretation.

10.11 Bibliographic notes

Typing in λProlog is described in [Nadathur and Miller, 1988; Nadathur and Pfenning, 1992].

The Ciao system preprocessor [Hermenegildo et al., 2005] provides for such functionality by allowing a programmer to write various properties about code that the preprocessor attempts to verify.

Encoding of asynchronous process calculi into linear logic has been explored in several papers: see, for example, [Kobayashi and Yonezawa, 1995; Miller, 2003], as well as in Chapter 7.

Most of the material in this chapter are based the paper [Miller, 2006].

Chapter 12

Encoding security protocols

By extending the encoding of multiset rewriting in linear logic that was presented in Section 7.6, we find a natural setting to encode some features of communicating processes that are communicating securely over a public communication structure.

12.1 Communicating processes

The left side of Figure 12.1 represents a common view of a data structure based on pointers. If I have access to the pointer on the top left then I have access to the resource A *and* to the resource B (memory is a good example of a resource). It is, of course, tempting to apply linear logic's negation to diagram and to the conjunction. To this end, consider the right side of this figure. Here, arrows have been inverted and the static resource (something that is *accessed*) is dualized into a process (the thing that does the *accessing*). The operational interpretation of this right-hand diagram is that the two processes P and Q meet (synchronize) around the $\parr$ and afterwards, they are replaced by a new process. Such an interpretation is exactly the intended meaning of a clause of the form

$$P \parr Q \multimapinv R,$$

where R is the result of P and Q meeting. Thus, the $\parr$ connective provides a location, a *forum*, for processes to meet: it is this aspect of $\parr$ that gave the Forum language in Chapter 6 its name.

To illustrate this approach to encoding processes using linear logic as a logic programming language, we consider here briefly the π-calculus. The principle computation mechanism of the π-calculus is the synchronization of two agents during which there is a transfer of a name from one agent to another. The expression $\bar{x}z.P$ describes an agent that is willing to transmit the name z on

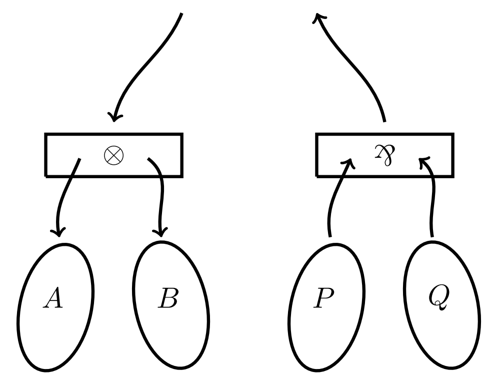

Figure 12.1: Illustrating how to interpret the operational reading of the dual connectives $\otimes$ and $⅋$.

the wire with name x. The expression $x(y).Q$ denotes an agent that is willing to receive a name on wire x and formally bind that value to y. The bound variable y in this expression is scoped over Q. The central computational step of the π-calculus is the reduction of the parallel composition $\bar{x}z.P \mid x(y).Q$ to the expression $P \mid Q[z/y]$. The agents P and $Q[z/y]$ are now able to continue their interactions with their environment independently.

Another important aspect of the π-calculus is the notion of scope restriction: in the agent expression $(x)P$, x is bound and invisible to the outside. The scoped value x, however, can be communicated outside its scope, providing a phenomenon known as "scope extrusion." For example, $(z)(\bar{x}z.P \mid Q) \mid x(y).R$ is structurally equivalent to $(z)(\bar{x}z.P \mid Q \mid x(y).R)$, provided that z is not free in $x(y).R$. This scope restriction is always easy to accommodate since we shall assume that α-conversion is available for changing the name of bound variables. This expression can be reduced to $(z)(P \mid Q \mid R[z/y])$, where the scope of the restriction (z) is larger since it contains the agent $R[z/y]$ in which z may be free. This mechanism of generating new names (using α-conversion) and sending them outside their scope is an important part of the computational power of the π-calculus.

For an example, consider the following process expression where a, b, x are free constants of type *name*.

$$(x(y).\bar{y}a.\bar{y}b.nil) \mid (z)(\bar{x}z.z(u).z(v).\bar{u}v.nil)$$

Given the informal description of how a π-calculus expression evolves, the scope of the (z) restriction enlarges to yield the expression

$$(z)\Big((x(y).\bar{y}a.\bar{y}b.nil) \mid (\bar{x}z.z(u).z(v).\bar{u}v.nil)\Big)$$

Next, a communication can take place within the scope of the restriction, yielding the expression

$$(z)\Big((\bar{z}a.\bar{z}b.nil) \mid (z(u).z(v).\bar{u}v.nil)\Big)$$

Two more internal communication steps yields that expression

$$(z)\Big(nil \mid (\bar{a}b.nil)\Big)$$

Since z is not free in the scope of the restriction (z) and since nil is the unit of parallel composition, this last expression is essentially the same as the expression $(\bar{a}b.nil)$.

We encode some of the behavior of the π-calculus as proof search within Forum using the following primitive type and four non-logical symbols.

```
kind name     type.
type or       o -> o -> o.
type send     name -> name -> o -> o.
type get      name -> (name -> o) ->  o.
type match    name -> name -> o -> o.
```

As is clear from these types, we make use of higher-order types and λ-abstractions to smooth the treatment of bound variables and variable scoping. The following mapping translates some π-calculus expressions into linear logic.

$$\langle\!\langle P \mid Q \rangle\!\rangle = \langle\!\langle P \rangle\!\rangle \parr \langle\!\langle Q \rangle\!\rangle \qquad \langle\!\langle (x)P \rangle\!\rangle = \forall x \langle\!\langle P \rangle\!\rangle \qquad \langle\!\langle nil \rangle\!\rangle = \bot$$

$$\langle\!\langle \bar{x}y.P \rangle\!\rangle = \texttt{send}\ x\ y\ \langle\!\langle P \rangle\!\rangle \qquad \langle\!\langle x(y).P \rangle\!\rangle = \texttt{get}\ x\ \lambda y \langle\!\langle P \rangle\!\rangle$$

$$\langle\!\langle P + Q \rangle\!\rangle = \texttt{or}\ \langle\!\langle P \rangle\!\rangle\ \langle\!\langle Q \rangle\!\rangle \qquad \langle\!\langle [x = y]P \rangle\!\rangle = \texttt{match}\ x\ y\ \langle\!\langle P \rangle\!\rangle$$

To describe the meaning of the five non-logical constants, we have the following Forum specification.

```
get X R || send X Y Q :- R Y || Q.
match X X P :- P.
or P Q :- P.
or P Q :- Q.
```

Note that these axioms are higher-order in the sense that they allow quantification over predicate symbols (such as `P` and `Q`) as well as variables of type $name \to o$ (such as `R`).

Exercise 12.1. Show that the informal reduction of π-calculus expressions given above can be reproduced in the Forum proof of the sequent $\Sigma : \Psi; P_0 \vdash P_1; \cdot$ where Σ collects the constants declared above along with the declarations that `a` and `b` are names, Ψ is the multiset of the six formulas listed above, P_1 is the expression

```
get x (y\ send y a (send y b bot)) ||
pi z\ (send x z (get z u\ (get z v\ send u v bot)))
```

and P_0 is the expression `(send a b bot)`.

Exercise 12.2. Let Q be the expression

```
get x y (or (match y a (send x a bot))
            (match y b (send x b bot)))
```

Also let P_a, P_b, and P_c be the processes `(send x a bot)`, `(send x b bot)`, and `(send x c bot)`, respectively. Show that the two Forum sequents $\Sigma : \Psi; P_a \vdash P_a \,|\, Q; \cdot$ and $\Sigma : \Psi; P_b \vdash P_b \,|\, Q; \cdot$ are provable but that $\Sigma : \Psi; P_c \vdash P_c \,|\, Q; \cdot$ is not provable.

Clearly, a goal of this kind of encoding of process calculus into linear logic would be to identify the notion of "process P reduces to Q" with the provability of the Forum sequent $\Sigma : \Psi; \langle\!\langle Q \rangle\!\rangle \vdash \langle\!\langle P \rangle\!\rangle; \cdot$. While this encoding into linear logic captures some of the nature of computation and communication in the π-calculus, there is also a serious flaws in this encoding. The first suggestion of such a flaw concerns that fact that only some combinators of the π-calculus are translated into linear logic connectives while others are encoded using non-logical constants. Why not encode, for example, the π-calculus $+$ using the linear logic $\oplus$? While the right-introduction rules for $\oplus$ in linear logic do encode the nondeterministic choice that is intended for the π-calculus reduction, the left-introduction rule for $\oplus$ would force us to accept the following reduction strategy: if P reduces to Q_1 and to Q_2, then P reduces to $Q_1 + Q_2$, which is a principle that is not generally seen as a proper reduction in the π-calculus literature. It is for this reason that the encoding of $+$ is made with a non-logical symbol since backchaining on its axiomatization mimics the right-hand introduction rule for $\oplus$ but the left-hand introduction is not available using that axiomatization.

Just as the left-rule for $\oplus$ rules out using that connective to encode the π-calculus $+$, the left-rule for $\forall$ is also problematic. Note that $\forall x \forall y. Pxy \vdash \forall x. Pxx$ is provable in every quantificational logic we have considered in this book. In the setting of the π-calculus, this would mean that we would need to accept the reduction of $(x)\bar{x}a.\bar{x}b.nil$ to the process $(x)(y)\bar{x}a.\bar{y}b.nil$, which is again not an accepted reduction in the π-calculus.

We will provide a different encoding of the π-calculus in Chapter 13 in which process expressions are not encoded as formulas but as terms. A much greater precision with the π-calculus can be achieve with that encoding.

In the rest of this chapter, we shall consider a calculus for communication that is, in some senses, weaker than that of the π-calculus. In this weaker setting, provability in linear logic is much more accurate and flexible.

Message 1	$A \longrightarrow S$: A, B, n_A
Message 2	$S \longrightarrow A$: $\{n_A, B, k_{AB}, \{k_{AB}, A\}_{k_{BS}}\}_{k_{AS}}$
Message 3	$A \longrightarrow B$: $\{k_{AB}, A\}_{k_{BS}}$
Message 4	$B \longrightarrow A$: $\{n_B\}_{k_{AB}}$
Message 5	$A \longrightarrow B$: $\{n_B, \mathit{Secret}\}_{k_{AB}}$

Figure 12.2: The conventional presentation of the Needham-Schroeder Shared Key Protocol.

12.2 A conventional presentation of protocols

Let us assume that Alice and Bob want to make use of a trusted server to help them establish their own private channel for communications. Both Alice and Bob have private encryption keys that allow them to communicate securely with a server. At the end of this protocol's execution, Alice and Bob should be sharing an encryption key that allows them to securely exchange messages between themselves, without any additional need of the trusted server.

Figure 12.2 is a presentation of the *Needham-Schroeder Shared Key Protocol* (abbreviated NS) using a standard kind of description. Here, A, B, and S denote the agents Alice, Bob, and server, respectively. In addition, encryption keys and nonces are denoted by the schematic variables k and n, respectively.

One of our goals is to replace this specific syntax with one that is based on a direct use of logic. We do this now by identifying a sequence of aspects of the conventional presentation that we might see as possible features of Forum.

Emphasize using a public network The notation $A \longrightarrow B : M$ is a bit misleading since it seems to indicate a "three-way synchronization" between Alice, Bob, and a message M. However, it is important to see that communication is, in fact, asynchronous, in the sense that Alice is meant to put the message M into a public network (say, the internet) and that at some time later, Bob is meant to retrieve that message from that network. It should be possible to these two actions to be interleaved with some intruder who might read, delete, and/or modify the message M. Thus, a better syntax is inspired by multiset rewriting (we use $\mathsf{N}\,\cdot$ to denote network messages).

$$\begin{array}{rcl} A & \longrightarrow & A' \mid \mathsf{N}\, M \\ B \mid \mathsf{N}\, M & \longrightarrow & B' \\ & \vdots & \\ E \mid \mathsf{N}\, M & \longrightarrow & E' \mid \mathsf{N}\, M \end{array}$$

Here, an eavesdropper E might read and rewrite the message while storing part of it in it internal memory. More generally, we can image that the action of an agent could be described more generally as

$$(A\ Memory) \mid \mathsf{N}\, M_1 \mid \cdots \mid \mathsf{N}\, M_p \longrightarrow (A'\ Memory') \mid \mathsf{N}\, P_1 \mid \cdots \mid \mathsf{N}\, P_q$$

where $p, q \geq 0$. The agent can be missing from the left (agent creation) or can be missing from the right (agent deletion). If agent is missing from both sides, messages might simply mutate into other messages. Multiset rewriting and, hence, linear logic can easily capture such dynamics.

Static distribution of keys Consider a protocol containing the following steps.

$$\begin{array}{lll} & \vdots & \\ \text{Message } i & \text{A} \longrightarrow \text{S: } \{M\}_k \\ \text{Message } j & \text{S} \longrightarrow \text{A: } \{P\}_k \\ & \vdots & \end{array}$$

In the general setting, we need to declare exactly which agents have access to which keys: in the steps above, we know two places where the k is used but we must separately declare, for example, that the key is not known to any other agents. This declaration is critical for modularity and for establishing correctness later: it can also be made statically by using a **local** declaration, such as the following.

$$local\ k. \quad \left\{ \begin{array}{rl} & \vdots \\ A & \longrightarrow A' \mid \ \mathsf{N}\{M\}_k \\ S \mid \ \mathsf{N}\{P\}_k & \longrightarrow S' \\ & \vdots \end{array} \right\}$$

This declarations appears to be similar to a quantifier. The intention is that we can statically examine all occurrences of the bound variable k in the scope of this quantifier and thereby know which agents do and do not contain occurrences of this key.

Dynamic creation of new symbols During the execution of a protocol, new symbols, representing nonces (used to help guarantee "freshness") and keys for encryption and session management, are needed in protocols. Using the syntax in Figure 12.2, one needs to explicitly point out that, for example, n_A, n_b, and k_{AB} need to be generated a fresh, new symbols during the execution of this protocol. We introduce a more explicit syntax for this purpose.

$$a_1\ S \longrightarrow new\ k.\ (a_2\ k\ S) \mid \mathsf{N}\{M\}_k$$

This *new* operator resembles, of course, a quantifier: it should support α-conversion and seems to be a bit like reasoning generically. The scope of *new* is over the body of this rule. This quantifier will also be used when we need to generate a nonce.

Mapping the conventional notation into linear logic There are two approaches to view the new notation we have introduced as logical connectives.

	$\mid$	unit	$\longrightarrow$	new	local
disjunctive	$⅋$	$\bot$	$\circ\!-$	$\forall$	$\exists$
conjunctive	$\otimes$	$\mathbf{1}$	$-\!\circ$	$\exists$	$\forall$

The disjunctive approach allows protocols to be seen as Forum specifications:: that is, it fits into the "logic programming as goal-directed search" paradigm. The conjunctive approach is also popular and has been used in, say, the MSR system [Cervesato et al., 1999]. From the linear logic perspective, these two approaches yield essentially the same dynamics when doing proof search: the only difference is that what happens in the right-hand side of sequents using the disjunctive approach happens essentially unchanged on the left-hand side using the conjunctive approach.

Encrypted data as an abstract data type A final step of encoding of the conventional syntax into Forum requires dealing with encryption keys and encrypted data. We shall assume that an encryption key is a symbolic function, say, k of type $d \to d$ and that the encrypted message $\{M\}_k$ is encode as the simple application $(k\ M)$. If an agent has access to the data constructor that is an encryption key, then via a simple matching operation within logic, decryption can take place. If, however, the encryption key is not available to the agent, then decryption is impossible. Thus, we are representing encrypted data as an *abstract data type*.

In order for encryption keys to be inserted into data object, we introduce the postfix coercion constructor $(\cdot)^\circ$ of type $(d \to d) \to d$. The use of higher-order types means that we will also use the equations of $\alpha\beta\eta$-conversion when processing encrypted data. Thus, we can write linear logic expressions of the following form.

$$\exists\, k. \left[\begin{array}{r} a_1\ S \circ\!- \forall n.\ a_2\ \langle k^\circ, S\rangle ⅋ \mathsf{N}\, k\ n \\ a_2\ \langle k^\circ, S\rangle ⅋ \mathsf{N}\, k\ M \circ\!- \ldots \qquad\qquad\qquad \end{array}\right]$$

12.3 A linear logic formulation

For the rest of this chapter, we assume that the primitive types are $\mathcal{S} = \{o, d\}$. We use the type d to encode messages. For convenience, we shall assume that all strings are included in this type. The tupling operator $\langle\cdot,\cdot\rangle$, for pairing data together, has type $d \to d \to d$. Expressions such as $\langle\cdot,\cdot,\ldots,\cdot\rangle$ denote pairing associated to the right. As mentioned in the previous section, we also need the constructor $(\cdot)^\circ$ of type $(d \to d) \to d$ in order to allow an encryption key to be considered a data item.

We encode a public communication medium as a multiset of *network messages* that are encoded as an atomic formulas of the form $\mathsf{N}\,t$, where $\mathsf{N}\,\cdot$ is a predicate of type $d \to o$ and t (of type d) is the actual encoding of a message. For example, the following are examples of network messages.

$$\mathsf{N}\,\langle\texttt{"alice"},\texttt{"account34"}\rangle \qquad \mathsf{N}\,\langle\texttt{"bob"},\texttt{"45euros"}\rangle$$

Such network messages could be used to facilitate a financial transaction. Since we will model the public network as an evolving multiset of atomic formulas with the N predicate, many actors (encoded as processes) other than Alice and Bob can access and read these messages: it is likely that we do not intend these financial transactions to be viewable and mutable by just anyone with access to the network.

In order to encode actors, such as Alice and Bob, participating in a communication protocol, we make the following few definitions. A *role identifier* is a symbol, say, ρ. [1] For some number $n \geq 1$ and for $i = 1,\ldots,n$, the pair ρ_i of a role identifier and an index is a *role state predicate* of type $d \to \cdots d \to o$ of some (possibly zero) arity. These state predicates are used to encode internal states of a role as a protocol progresses. A *role state atom* is an atomic formula of the form $(\rho_i\ t_1\ \cdots\ t_m)$ where $t_1,\ldots,t_m$ are terms of type d and ρ_i is a role state predicate. A *role clause* is a linear logic formula of the form

$$\forall x_1 \ldots \forall x_i [a_1 \parr \cdots \parr a_m \multimapinv \forall y_1 \ldots \forall y_j [b_1 \parr \cdots \parr b_n]]$$

where $m \geq 1$ and $i, j, n \geq 0$. Here, the *head* of such a clause is the formula $a_1 \parr \cdots \parr a_m$ and the *body* is $\forall y_1 \ldots \forall y_j [b_1 \parr \cdots \parr b_n]$. Role clauses also have the following restrictions: all the atoms $a_1,\ldots,a_m,b_1,\ldots,b_n$ are either network messages or role state atoms such that the following hold.

1. There must be exactly one role state atom in the head and at most one in the body.

2. If the role state atom in the head is $(\rho_i\ \bar{t})$ and if there is any role state atom in the body, say, $(\rho'_j\ \bar{s})$, then ρ and ρ' must be the same role identifier and $i < j$.

Thus, a role clause involves at most a single role (and possibly network messages): this implies that roles cannot synchronize with other roles directly and

[1] Should I use the term "agent" instead of "role"?

that one role cannot evolve into another role. It is allowed for a role to be deleted since no role state atom must appear in the body. It is also the case that all roles have finite runs.

A *role theory* is a linear logic formula of the form

$$\exists x_1 \ldots \exists x_r \; [C_1 \otimes \cdots \otimes C_s],$$

where $r, s \geq 0$, $C_1, \ldots, C_s$ are role clauses, where $x_1, \ldots, x_r$ are variables of type d or $d \to d$, and whenever C_i and C_j have the same role state predicate in their head then $i = j$. This latter condition will imply that agents in protocols are deterministic. This is a condition that can easily be relaxed within linear logic if nondeterministic agents are of interest.

Many other restrictions or generalization could be considered here for the definition of roles theory and role clauses, but for our simple considerations here, this definition is sufficient. Ultimately, we will introduce a different syntax for roles that will not need to use these rather awkward role state predicates. Existential quantification like that surrounding role theories are used in logic programming (see Section 9.6) to provide for abstract data-types and here they will serve as local constants shared by certain role clauses. In particular, shared keys between, say Alice and a trusted server, will be existentially quantified in this way with a variable of type $d \to d$. The use of existential quantifier at type $d \to d$ is explained next.

12.4 Encryption as an abstract data type

As we have mentioned, encryption keys are encoded using symbols of type $d \to d$. These keys can be given static scope in a role theory using existential quantification around role clauses in such a theory. They can also be generated as new using a universal quantifier in the body of a role clause.

Consider the following specification that contains three occurrences of encryption keys.

$$\begin{array}{lll} \exists k_{as} \exists k_{bs} [& a_1 \; \langle M, S \rangle & \multimapinv a_2 \; S \parr \mathsf{N}\,(k_{as} \; M). \\ & b_1 \; T \parr \mathsf{N}\,(k_{bs} \; M) & \multimapinv b_2 \; M \; T. \\ & s_1 \; \parr \mathsf{N}\,(k_{as} \; P) & \multimapinv \mathsf{N}\,(k_{bs} \; P). \qquad] \end{array}$$

(Here as elsewhere, quantification of capital letter variables is universal with scope limited to the clause in which the variable appears.) In this example, Alice (a_1, a_2) communicates with Bob (b_1, b_2) via a server (s_1). To make the communications secure, Alice uses the key k_{as} while Bob uses the key k_{bs}. The server is deleted immediately after it translates one message encrypted for Alice to a message encrypted for Bob. The use of the existential quantifiers helps establish that the occurrences of keys, say, between Alice and the server

$$\begin{aligned}
&\exists k_{as} \exists k_{bs} \{ \\
a_1\ S &\multimapinv \forall na.\ a_2\ na\ S \parr \mathsf{N}\,\langle alice, bob, na\rangle. \\
a_2\ N\ S \parr \mathsf{N}\,(k_{as}\langle N, bob, K, En\rangle) &\multimapinv a_3\ N\ K\ S \parr \mathsf{N}\,En. \\
a_3\ Na\ Key^{\circ}\ S \parr \mathsf{N}\,(Key\ Nb) &\multimapinv a_4\ \parr \mathsf{N}\,(Key\langle Nb, S\rangle). \\
b_1\ \parr \mathsf{N}\,(k_{bs}\langle Key^{\circ}, alice\rangle) &\multimapinv \forall nb.\ b_2\ nb\ Key^{\circ} \parr \mathsf{N}\,(Key\ nb). \\
b_2\ Nb\ Key^{\circ} \parr \mathsf{N}\,(Key\langle Nb, S\rangle) &\multimapinv b_3\ S. \\
s_1 \parr \mathsf{N}\,\langle alice, bob, N\rangle &\multimapinv \forall k.\mathsf{N}\,(k_{as}\langle N, bob, k^{\circ}, (k_{bs}\langle k^{\circ}, alice\rangle)\rangle). \\
&\}
\end{aligned}$$

Figure 12.3: Encoding the NS protocol.

and Bob and the server, are the only occurrences of that key. Even if more principals are added to this system, these occurrences are still the only ones for these keys. Thus, the existential quantifier helps in determining the static or lexical scope of key distribution. Of course, as protocols are evaluated (that is, a proof is searched for), keys may extrude their scope and move freely onto the network. This dynamic notion of scope extrusion is similar to that found in the π-calculus [Milner et al., 1992a] and is modeled here similar to an encoding of the π-calculus into linear logic found in [Miller, 1993].

Example 12.3. Figure 12.3 contains a linear logic implementation of the NS protocol contained in Figure 12.2. Let $C_1, \ldots, C_6$ be the six role clauses in Figure 12.3 (remembering that there are implicit universal quantifiers around role clauses). It is a simple matter to show that this protocol implements the specification

$$\forall x[a_1\ x \parr b_1 \parr s_1 \multimapinv a_4 \parr b_3\ x]$$

in the sense that there is a simple proof of the Forum sequent

$$\Sigma, k_{as}, k_{bs} : C_1, \ldots, C_6; \cdot \vdash \forall x[a_4 \parr b_3\ x \multimap a_1\ x \parr b_1 \parr s_1]; \cdot$$

That is, this protocol is able to transform the initial states of Alice (with some secret), Bob, and the server to the final states of Alice and Bob (now with the secret).

Example 12.4. Consider the following two clauses for Alice.

$$a\ K^{\circ} \parr \mathsf{N}\,(K\ M) \multimapinv a'\ M. \qquad (3.1)$$
$$a\ \parr \mathsf{N}\,(K\ M) \multimapinv a'\ M. \qquad (3.2)$$

In the first case, Alice possesses an encryption key and uses it to decrypt a network message. In the second case, it appears that she is decrypting a message without knowing the key, an inappropriate behavior, of course. Note that (3.2) is logically equivalent (and, hence, operationally indistinguishable using proof search) to both of the formulas

$$\forall M \forall X[a \parr \mathsf{N}\, X \multimap\!\!- a'\ M] \quad \text{and} \quad \forall X[a \parr \mathsf{N}\, X \multimap\!\!- \exists M. a'\ M].$$

This last clause clearly illustrates that Alice is not actually decoding an existing message but is simply guessing (using $\exists$) at some data value M, and continues with that guess as $a'\ M$. If one thinks operationally instead of declaratively about proof search involving clause (3.2), we would consider possible unifiers for matching the pattern $(K\ M)$ with a network message, say, $(k\ s)$, for two constants k and s. Unification on simply typed λ-terms yields exactly the following three distinct unifiers:

$$[M \mapsto (k\ s), K \mapsto \lambda w.w] \quad [M \mapsto s, K \mapsto k] \quad [M \mapsto M, K \mapsto \lambda w.(k\ s)]$$

Thus, M can be bound to either $(k\ s)$ or s or any term: in other words, M can be bound to any expression of type d.

Exercise 12.5. The logical entailment can help in reasoning about role clauses and theories: such entailments are strengthened by the presence of quantification at type $d \to d$. Consider the two clauses

$$a_1 \multimap\!\!- \forall k. \mathsf{N}\,(k\ m) \quad \text{and} \quad a_1 \multimap\!\!- \forall k. \mathsf{N}\,(k\ m').$$

Both of these clauses specify that Alice can take a step that generates a new encryption key and then outputs a message (either m or m') using that encryption key. Since Alice has no continuation, no one, not even Alice will be able to decode this message. It should be the case that these two clauses are "operationally" similar since they both generate a "junk message." Show that these formulas are, in fact, logically equivalent.

What is missing here is a kind of converse to the claim in Exercise 12.3.

12.5 Abstracting internal states

The following example illustrates that using existential quantification over *predicates* (in particular, role state predicates) allows interesting rewriting of the structure of role theories.

Example 12.6 (Reducing n-way to 2-way synchronization)**.** General n-way synchronization ($n \geq 3$) can be rewritten using 2-way synchronization by the introduction of new, intermediate, and hidden predicates. For example, the following two formulas are logically equivalent.

$$\exists l_1 \exists l_2. \left\{ \begin{array}{r} a \parr b \multimapinv l_1 \\ l_1 \parr c \multimapinv l_2 \parr e \\ l_2 \multimapinv d \parr f \end{array} \right\} \quad \dashv\vdash \quad a \parr b \parr c \multimapinv d \parr e \parr f$$

The clause on the right specifies a 3-way synchronization and the spawning of 3 new atoms whereas the formula on the left is limited to rewriting at most two atoms into at most 2 atoms. The proof of the forward entailment in linear logic is straightforward while the proof of the reverse entailment involves the two higher-order substitutions of $a \parr b$ for $\exists l_1$ and $d \parr f$ for $\exists l_2$. As long as we are using logical entailment, these two formulas are indistinguishable and can be used interchangeably in all contexts. If instead we could observe possible failures in the search for proofs, then it is possible to distinguish these formulas: consider the search for a proof of a sequent containing a and b but not c. The proof theory of linear logic we have presented here does not observe such failures since that proof theory is generally involved with reasoning about *complete* proofs.

Existential quantification over program clauses can also be used to hide predicates encoding roles. In fact, one might argue that the various restrictions on sets of process clauses (no synchronization directly with atoms encoding roles and no role changing into another role) might all be considered a way to enforce locality (i.e., hiding) of predicates. Existential quantification can, however, achieve this same notion of locality but much more directly.

Example 12.7 (Hiding role state predicates)**.** The following two formulas are logically equivalent:

$$\exists\ a_2, a_3. \left\{ \begin{array}{l} a_1 \parr \mathsf{N}\, m_0 \multimapinv a_2 \parr \mathsf{N}\, m_1 \\ a_2 \parr \mathsf{N}\, m_2 \multimapinv a_3 \parr \mathsf{N}\, m_3 \\ a_3 \parr \mathsf{N}\, m_4 \multimapinv a_4 \parr \mathsf{N}\, m_5 \end{array} \right\} \quad \dashv\vdash$$

$$a_1 \parr \mathsf{N}\, m_0 \multimapinv (\mathsf{N}\, m_1 \multimapinv (\mathsf{N}\, m_2 \multimapinv (\mathsf{N}\, m_3 \multimapinv (\mathsf{N}\, m_4 \multimapinv (\mathsf{N}\, m_5 \parr a_4)))))$$

The changing of polarity that occurs when moving to the body of a $\multimapinv$ flips expressions from output (e.g., $\mathsf{N}\, m_1$) to input (e.g., $\mathsf{N}\, m_2$), etc.

We develop the observation made in this example to a larger extent in the next section.

12.6 Roles as nested implications

The observation that abstracting over internal states results in an equivalent syntax with nested $\multimapinv$ suggests an alternative syntax for roles. Consider the following two syntactic categories of linear logic formulas:

$$H ::= A \mid \bot \mid H \parr H \qquad\qquad K ::= H \mid H \multimapinv K \mid \forall x.K$$

Here, A denotes the class of atomic formulas encoding network messages (in particular, formulas of the form $\mathsf{N}\,\cdot$). Formulas belonging to the class H denote bundles of messages that are used as either input or output to the network. Formulas belonging to the class K can have deep nesting of implications. As we shall see, the nesting of $\multimapinv$ causes an alternation between a process that is willing to output a message to one that is willing to input a message.

To see this mechanism in the proof search setting, consider a sequent $\Delta \longrightarrow \Gamma$, where Δ is a multiset of K formulas and Γ are multisets of K formulas (here, we elide the signature associated to a sequent). The right-hand side of sequents involve asynchronous behavior (output) and left-hand side of sequents involve synchronous behavior (input). The two rules involving proof search with implications can be written as follows:

$$\frac{\Delta, K \longrightarrow \Gamma, H, \mathcal{A}}{\Delta \longrightarrow H \multimapinv K, \Gamma, \mathcal{A}} \qquad \frac{H \longrightarrow \mathcal{A}_1 \qquad \Delta \longrightarrow K, \mathcal{A}_2}{\Delta, H \multimapinv K \longrightarrow \mathcal{A}_1, \mathcal{A}_2}$$

Here, $\mathcal{A}$ denotes a multiset of atoms (i.e., network messages). Note that we can assume that the left-introduction rule for $\multimapinv$ is only done when the right-hand side of the concluding sequent contains at most atomic formulas.

Figure 12.4 contains three formulas are displayed: the first represents the role of Alice, the second Bob, and the final one the server. (All agents in this figure are written at the same polarity, in this case, in output mode: since Bob and the server essentially start with inputs, these two agents are negated, meaning they first output nothing and then move to input mode.) These formulas are a second way to encode the NS protocol within linear logic. If the three formulas in Figure 12.4 are placed on the right-hand side of a sequent arrow (with no formulas on the left) then the role formula for Alice will output a message and move to the left-side of the sequent arrow (reading inference rules bottom up). Bob and the server output nothing and move to the left-hand side as well. At that point, the server will need to be chosen for a $\multimap$L inference rule, which will cause it to input the message that Alice sent and then move its continuation to the right-hand side. It will then immediately output another message, and so on. If a role for the server should be permanent, then the first line of Figure 12.4 for the server could be simply changed by replacing $\multimapinv$ with $\Leftarrow$.

Various equivalences familiar from the study of asynchronous communication are found in linear logic. For example, if one skips a phase, the two phases can be contracted as follows:

$$p \multimapinv (\bot \multimapinv (q \multimapinv k)) \equiv p \parr q \multimapinv k$$

(Out)	$\forall na.\ \mathsf{N}\,\langle alice,\ bob,\ na\rangle \mathrel{\circ\!\!-}$
(In)	$\quad(\forall K \forall En.\ \mathsf{N}\,(k_{as}\langle na,\ bob,\ K^\circ,\ En\rangle) \mathrel{\circ\!\!-}$
(Out)	$\quad\quad(\mathsf{N}\,En \mathrel{\circ\!\!-}$
(In)	$\quad\quad\quad(\forall N.\ \mathsf{N}\,(KN) \mathrel{\circ\!\!-}$
(Out)	$\quad\quad\quad\quad(\mathsf{N}\,(Kab\langle N,\ secret\rangle) \mathrel{\circ\!\!-}$
(Cont)	$\quad\quad\quad\quad\quad a_4))).$

THE ROLE FOR ALICE

(Out)	$\perp \mathrel{\circ\!\!-}$
(In)	$\quad(\forall Key.\ \mathsf{N}\,k_{bs}(Key^\circ,\ alice) \mathrel{\circ\!\!-}$
(Out)	$\quad\quad(\forall nb.\ \mathsf{N}\,(Key\ nb) \mathrel{\circ\!\!-}$
(In)	$\quad\quad\quad(\mathsf{N}\,(Key\langle nb,\ secret\rangle) \mathrel{\circ\!\!-}$
(Cont)	$\quad\quad\quad\quad b_3\ secret))).$

THE ROLE FOR BOB

(Out)	$\perp \mathrel{\circ\!\!-}$
(In)	$\quad(\forall N.\ \mathsf{N}\,\langle alice,\ bob,\ N\rangle \mathrel{\circ\!\!-}$
(Out)	$\quad\quad(\forall k.\ \mathsf{N}\,k_{as}\langle N,\ bob,\ k^\circ,\ k_{bs}(k^\circ,\ alice)\rangle)).$

THE ROLE FOR THE SERVER

Figure 12.4: The roles of Alice, Bob, and a server

$$p \mathrel{\circ\!\!-} (\perp \mathrel{\circ\!\!-} \forall x(q\ x \mathrel{\circ\!\!-} k\ x)) \equiv \forall x(p \parr q\ x \mathrel{\circ\!\!-} k\ x)).$$

While the nested presentation of roles is in some sense, more complicated syntax than the form using role clauses, this presentation certainly has its advantages. For example, there is only one predicate, namely $\mathsf{N}\,\cdot$, involved in writing out security protocols: role identifiers and role state predicates have disappeared. A role can now be seen as simply a formula and a role theory as simply an existentially quantified tensor of roles.

Do the following proposition and proof more carefully.

Proposition 12.8. *For every role theory in which only the predicate $\mathsf{N}\cdot$ is free, there is a collection of role formulas to which it is provably equivalent.*

Proof. This proposition is proved by showing how to remove the existentially quantified role state predicate with maximal index by generating the appropriate higher-order substitution (similar to those produced in Example 12.6). When no more quantified role state predicates remain, the resulting theory is the desired collection of role formulas. □

The style of specification given in Figure 12.4 is similar to that of process calculus: in particular, the implication $\circ\!-$ is syntactically similar to the dot prefix in, say, CCS. Universal quantification can appear in two modes: in output mode it is used to generate new eigenvariables (similar to the π-calculus restriction operator) and in input mode it is used for variable binding (similar to value-passing CCS). The formula $a \circ\!- (b \circ\!- (c \circ\!- (d \circ\!- k)))$ can denote processes described as

$$\bar{a} \,||\, (b.\ (\bar{c} \,||\, (d.\ \ldots))) \quad \text{or} \quad a.\ (\bar{b} \,||\, (c.\ (\bar{d} \,||\, \ldots)))$$

depending on which polarity it is being used. This formula and it's negation can also be written without linear implications as follows:

$$a \parr (b^{\perp} \otimes (c \parr (d^{\perp} \otimes \ldots))) \text{ resp, } \; a^{\perp} \otimes (b \parr (c^{\perp} \otimes (d \parr \ldots))).$$

Once a process with a continuation (that is, one that has an implication) has done an output (input), its continuation is an input (output) process.

The following two examples illustrate a difference between the abstractions available in logic with those available in the π-calculus and the spi-calculus.

Example 12.9 (Comparison with the π-calculus)**.** The π-calculus expression

$$(x)(\bar{x}m \mid x(y).Py)$$

is (weakly) bisimilar to the expression (Pm). This example is used to show that communication over a hidden channel provides no possible means for the environment to interact. A similar expression can be written as the following expression in linear logic:

$$\forall K[Km \parr (\forall x(Px \multimap Kx) \multimap \perp)].$$

Here, we have abstracted the *predicate* K: in a sense, we have abstracted the communication medium itself, and as such, the medium is available only for the particular purpose of communicating the message m from one process to another that is willing to do an input. This expression is logically equivalent to (Pm): the proof that (Pm) implies the displayed formula involves a use of equality (easy to add to the underlying logic in several ways) and the higher-order substitution $\lambda w.(w = m) \multimap \perp$ for K.

Example 12.10 (Comparison with the spi-calculus)**.** In the spi-calculus [Abadi and Gordon, 1999], a "public" channel can be used for communicating. To ensure that messages are only "understood" by the appropriate parties, messages are encrypted with keys that are given specific scopes. For example, the expression

$$(k)(\bar{q}(\{m\}_k) \mid q(y).\text{let } \{x\}_k = y \text{ in } Px)$$

describes a process that is willing to output an encrypted message $\{m\}_k$ on a public channel q and to also input such a message and decode it. The key k is given a scope similar to that given in the π-calculus expression. The linear logic expression, call it B,

$$\forall k[\mathsf{N}\,(k\ m) \parr (\forall x(Px \multimap \mathsf{N}\,(k\ x))) \multimap \bot]$$

is most similar to this spi-calculus expression: here, the network $\mathsf{N}\,\cdot$ corresponds to the public channel q. It is not the case, however, that B is logically equivalent to Pm since linear logic can observe that B can output something on the public channel, that is, $\forall y(\top \multimap \mathsf{N}\,y) \vdash B$ whereas it is not necessarily true that $\forall y(\top \multimap \mathsf{N}\,y) \vdash Pm$ is provable.

12.7 Bibliographic notes

Many of the examples from this chapter were taken from [Miller, 2003] and some of those have been inspired by material on encoding security protocols in MSR (multiset rewriting) found in [Cervesato et al., 1999, 2000a; Cervesato and Stehr, 2007].

While high-level specifications of secure channels in systems like the π-calculus or proof theory are elegant to use, it is possible to provide lower level implementations using encryption of such high-level constructs [Abadi et al., 2002].

Andreoli used a compilation method [Andreoli, 1992] collection of *bipolar* formulas. Applying his compiling technique to the formula in Figure 12.4 yields the formulas in Figure 12.3: the new constants introduced by compilation are the names used to denote role continuation.

Chapter 13

Formalizing operational semantic

In this chapter, we show how to use logic programming languages to specify the operational semantics of various programming and specification languages. Establishing these links between logic and operational semantics has many advantages for operational semantics: implementations from automated deduction can be used to animate semantic specifications; the proof-theoretic treatment of term-level binding structures can be used to address binding structures in the syntax of programs; and the declarative nature of logical specifications provides broad avenues for reasoning about semantic specifications. We shall illustrate all of these advantages in this chapter.

This chapter will use the term "logic specification" interchangeably with "logic program" and "theory". Additionally, when we speak of "programming languages" we include specification languages such as the λ-calculus and the π-calculus.

13.1 Three frameworks for operational semantics

Several formalisms have been used to specify what and how programming languages compute. If one wishes to build on top of such formalisms such things as *concepts* (e.g., observational equivalence and static analysis) and *tools* (e.g., interpreters, model checkers, and theorem provers), then the quality of such encodings is essential. Here, we shall use logic to directly encode operational semantics instead of other formal devices such as, for example, complete partial orders, algebras, games, and Petri nets. Proof search will provide logical specifications with dynamics that are able to capture a range of operational specifications. We focus on three popular frameworks for specifying operational semantics and describe the logic frameworks most naturally associated with that framework. These three frameworks are described below.

Structural operational semantics First introduced by Milner [1980] and by Plotkin [1981, 2004], *structural operational semantics* (SOS) was used to describe programming language features ranging from concurrency to functional computation to stateful computations. This style of specification, now commonly referred to as *small-step SOS*, allows for a natural treatment of concurrency via interleaving. *Big-step SOS*, introduced by Kahn [1987], is convenient for specifying, say, non-concurrent settings such as functional programming. Both of these forms of operational semantics define relations using inductive systems described by inference rules. As we shall see, Horn clauses are usually appropriate for encoding such inference rules.

Abstract machines A certain kind of term rewriting can be viewed as encoding *abstract machines* in which objects like code, stacks of arguments, etc, are directly manipulated. The SECD machine of Landin [1964] is an early example of an abstract machine. Such abstract machines can often be represented using *binary clauses*, which are degenerate Horn clauses in which the body of a clause contains one atomic formula. Proof search with such clauses are tail recursive and naturally specify simple, iterative algorithms. Arbitrary Horn clause programs can also be transformed into binary clauses using a continuation-passing style transformation. As such, binary clauses can be seen as capturing a thread of computation that contains a sequence of instructions or commands. While binary clauses represent a retreat from logic in the sense that they employ fewer logical constants (such as conjunction) than general Horn clauses, they do provide two things in exchange: (1) a way to explicitly specify the order of computation and (2) a basis for an extension to linear logic in which concurrency and imperative features can be naturally captured in a big-step-style semantic specification.

Multiset rewriting Specifying computations by computing directly on multisets was proposed in the 1990s with the Gamma programming language [Banâtre and Métayer, 1993] and the chemical abstract machine [Berry and Boudol, 1992], as well as later with the specification of security protocols [Bistarelli et al., 2005; Cervesato et al., 1999; Durgin et al., 2004]. In addition, Petri net specifications can be encoded directly using multiset rewriting [Delzanno, 2002]. Since the proof theory of linear logic employs multisets, it is a simple matter to capture multiset rewriting using proof search in linear (see Section 7.1 and [Gehlot and Gunter, 1990; Kanovich, 1995]).

13.2 The abstract syntax of programs as terms

In order to encode a programming language, we first map the syntactic expressions used to form programs into logic-level terms. Since almost all interesting programming languages contain binding constructions, we will capture those binding constructions directly using the λ-binding available within Church's STT [Church, 1940] (see Section 2.3). Constructors of the programming language are mapped to term constructors and the latter are typed by the *syntactic categories* of the objects that are used in the construction. As is common, the term constructor is modeled as an *application* of the constructor to arguments. Similarly, binders in the programming language domain are mapped to λ-*abstractions* of variables over the encoding of their scope. We illustrate more aspects of this style of encoding with two examples that we shall return to again later.

13.2.1 Encoding the untyped λ-calculus

The untyped λ-calculus (see Section 2.1) can be encoded as simply typed λ-terms using one syntactic type, say tm, and two constructors for application and abstraction. If we use the constructor app for building applications then its typing is given as $app : tm \to tm \to tm$: that is, app takes two untyped λ-terms and returns their application. If we use the constructor abs for building an untyped λ-abstraction, its type is $(tm \to tm) \to tm$. Using λProlog syntax, these tokens are declared as follows.

```
kind tm     type.
type app    tm -> tm -> tm.
type abs    (tm -> tm) -> tm.
```

Note that abs is applied to a term-level abstraction: the argument type $tm \to tm$ acts as the syntactic type of term abstractions over terms. The following is a list of some untyped λ-terms along with their encoding as a simply type λ-term of type tm.

$\lambda x.x$	$(abs\ \lambda x\ x)$
$\lambda x \lambda y.x$	$(abs\ \lambda x\,(abs\ \lambda y\ x))$
$\lambda x.\,(x\ x)$	$(abs\ \lambda x\,(app\ x\ x))$
$\lambda x \lambda y \lambda z.\,(x\ z)\,(y\ z)$	$(abs\ \lambda x\,(abs\ \lambda y\,(abs\ \lambda z\,((app\ \ (app\ x\ z)\ \ (app\ y\ z))))))$

It is important to observe that two untyped λ-terms are α convertible if and only if their encodings as terms of type tm are $\beta\eta$-convertible.

13.2.2 Encoding the π-calculus expressions

Processes in the finite π-calculus are describe by the grammar

$$\mathtt{P} ::= 0 \mid \bar{x}y.\mathtt{P} \mid x(y).\mathtt{P} \mid \tau.\mathtt{P} \mid (x)\mathtt{P} \mid [x = y]\mathtt{P} \mid \mathtt{P}|\mathtt{P} \mid \mathtt{P} + \mathtt{P}.$$

```
kind  n, p          type.

type  null          p.
type  taup          p -> p.
type  plus, par     p -> p -> p.
type  match, out    n -> n -> p -> p.
type  nu            (n -> p) -> p.
type  in            n -> (n -> p) -> p.
```

Figure 13.1: Declarations of the primitive types and constructors for the finite π-calculus.

(Many treatments of the π-calculus also include a replication operator or recursion: their absence here is why we are describing the *finite* π-calculus.) We use the symbols `P` and `Q` to denote processes and lower case letters, e.g., x, y, z to denote names. The occurrences of y in the processes $x(y).\mathtt{P}$ and $(y)\mathtt{P}$ are binding occurrences with `P` as their scope. The notion of free and bound variables is the usual one and we consider processes to be syntactically equal if they are equal up to α-conversion.

We encode π-calculus expressions as terms using the declarations of two primitives types, for names and processes, and process constructors in Figure 13.1. The precise translation of π-calculus syntax into simply typed λ-terms is given using the following function $[\![.]\!]$ that translates process expressions to $\beta\eta$-long normal terms of type p.

$$\begin{array}{rcl@{\qquad}rcl}
[\![0]\!] &=& \mathtt{null} & [\![\mathtt{P}+\mathtt{Q}]\!] &=& \mathtt{plus}\ [\![\mathtt{P}]\!]\ [\![\mathtt{Q}]\!] \\
[\![\tau.\mathtt{P}]\!] &=& \mathtt{taup}\ [\![\mathtt{P}]\!] & [\![\mathtt{P}|\mathtt{Q}]\!] &=& \mathtt{par}\ [\![\mathtt{P}]\!]\ [\![\mathtt{Q}]\!] \\
[\![x(y).\mathtt{P}]\!] &=& \mathtt{in}\ x\ (\lambda y.[\![\mathtt{P}]\!]) & [\![\bar{x}y.\mathtt{P}]\!] &=& \mathtt{out}\ x\ y\ [\![\mathtt{P}]\!] \\
[\![(x)\mathtt{P}]\!] &=& \mathtt{nu}\ (\lambda x.[\![\mathtt{P}]\!]) & [\![[x=y]\mathtt{P}]\!] &=& \mathtt{match}\ x\ y\ [\![\mathtt{P}]\!]
\end{array}$$

For example, the π-calculus expression $\nu y.[x = y]\bar{x}z.0$ is translated into the expression `(nu y\ match x y (out x z null))` which contains the free names `x` and `z`.

13.3 Big step semantics: call-by-value evaluation

Figure 13.2 contains a common, *big-step semantic specification* of *call-by-value evaluation* for the λ-calculus: this specification is given as both inference rules as well as Horn clause specification (using λProlog syntax). The (infix) predicate symbol $\Downarrow$ is of type $tm \to tm \to o$ is written simply as `eval` in these

$$\frac{}{\lambda x.R \searrow \lambda x.R} \qquad \frac{M \searrow (\lambda x.R) \qquad N \searrow U \qquad R[U/x] \searrow V}{(M\ N) \searrow V}$$

```
type eval   tm -> tm -> o.

eval (abs R) (abs R).
eval (app M N) V :-
     eval M (abs R), eval N U, eval (R U) V.
```

Figure 13.2: Big step specification of the call-by-value evaluation of the untyped λ-calculus.

clauses. The encoding of the atomic evaluation judgment $R[U/x] \searrow V$ in Figure 13.2 is simply `(eval (R U) V)` in the clausal specification: that is, the logic expression simply forms the expression $(R\ U)$ and once R is instantiated with a λ-abstractions, the logic's built-in treatment of β-reduction performs the necessary substitution.

Such a specification is referred to as "big step" since the predicate $\searrow$ relates an expression to its final value. In contrast, as we now illustrate, "small step" specifications encode just a single step in a possibly long series of transitions.

13.4 Small step semantics: π-calculus transitions

Figure 13.3 contains a specification of the late transition semantics for the π-calculus. Figure 13.5 contains the corresponding specification using λProlog. This encoding uses three constructors for actions: $\tau : a$ (for the silent action) and the two constants $\downarrow$ and $\uparrow$, both of type $n \to n \to a$ (for building input and output actions, respectively). Note that τ is overloaded by being used as a constructor of actions and of processes. The free output action $\bar{x}y$, is encoded as $\uparrow xy$ while the bound output action $\bar{x}(y)$ is encoded as $\lambda y\ (\uparrow xy)$ (or the η-equivalent term $\uparrow x$). The free input action xy, is encoded as $\downarrow xy$ while the bound input action $x(y)$ is encoded as $\lambda y\ (\downarrow xy)$ (or simply $\downarrow x$). Note that bound input and bound output actions have type $n \to a$ instead of a.

The relation of one-step (late) transition [Milner et al., 1992b] for the π-calculus is denoted by `P` $\xrightarrow{\alpha}$ `Q`, where `P` and `Q` are processes and α is an action. Our encoding splits this relation into two relations, depending on whether or not the transition involves a free or a bound action. The relation between two processes, P and Q, and an action A is encoded using the arrow $P \xrightarrow{A} Q$. This relation is captured using the predicate `one` of type `p -> a -> p -> o` in Figure 13.5. The relation between a process P, an abstracted process Q, and

$$\frac{}{\tau.\mathtt{P} \xrightarrow{\tau} \mathtt{P}}\ \textsc{tau} \qquad \frac{}{x(y).\mathtt{P} \xrightarrow{x(w)} \mathtt{P}[w/y]}\ \textsc{in}, w \notin fn((y)\mathtt{P})$$

$$\frac{}{\bar{x}y.\mathtt{P} \xrightarrow{\bar{x}y} \mathtt{P}}\ \textsc{out} \qquad \frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}{[x = x]\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}\ \textsc{match}$$

$$\frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}{\mathtt{P} + \mathtt{Q} \xrightarrow{\alpha} \mathtt{P}'}\ \textsc{sum} \qquad \frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}{\mathtt{P} \mid \mathtt{Q} \xrightarrow{\alpha} \mathtt{P}' \mid \mathtt{Q}}\ \textsc{par}, bn(\alpha) \cap fn(\mathtt{Q}) = \emptyset$$

$$\frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}{(y)\mathtt{P} \xrightarrow{\alpha} (y)\mathtt{P}'}\ \textsc{res}, y \notin n(\alpha)$$

$$\frac{\mathtt{P} \xrightarrow{\bar{x}y} \mathtt{P}'}{(y)\mathtt{P} \xrightarrow{\bar{x}(w)} \mathtt{P}'[w/y]}\ \textsc{open}, y \neq x, w \notin fn((y)\mathtt{P}')$$

$$\frac{\mathtt{P} \xrightarrow{\bar{x}(w)} \mathtt{P}' \quad \mathtt{Q} \xrightarrow{x(w)} \mathtt{Q}'}{\mathtt{P} \mid \mathtt{Q} \xrightarrow{\tau} (w)(\mathtt{P}' \mid \mathtt{Q}')}\ \textsc{close} \qquad \frac{\mathtt{P} \xrightarrow{\bar{x}y} \mathtt{P}' \quad \mathtt{Q} \xrightarrow{x(z)} \mathtt{Q}'}{\mathtt{P} \mid \mathtt{Q} \xrightarrow{\tau} \mathtt{P}' \mid \mathtt{Q}'[y/z]}\ \textsc{comm}$$

Figure 13.3: The late transition rules for the finite π-calculus.

an bound action A is encoded using the harpoon $P \overset{A}{\rightharpoonup} Q$. This relation is captured using the predicate `oneb` of type `p -> (n -> a) -> (n -> p) -> o` in Figure 13.5.

One-step transition judgments are translated to atomic formulas as follows (we overload the symbol $[\![.]\!]$ from Section 13.2.2).

$$\begin{array}{rcl@{\qquad}rcl}
[\![\mathtt{P} \xrightarrow{xy} \mathtt{Q}]\!] & = & [\![\mathtt{P}]\!] \xrightarrow{\downarrow xy} [\![\mathtt{Q}]\!] & [\![\mathtt{P} \xrightarrow{x(y)} \mathtt{Q}]\!] & = & [\![\mathtt{P}]\!] \overset{\downarrow x}{\rightharpoonup} \lambda y.[\![\mathtt{Q}]\!] \\
[\![\mathtt{P} \xrightarrow{\bar{x}y} \mathtt{Q}]\!] & = & [\![\mathtt{P}]\!] \xrightarrow{\uparrow xy} [\![\mathtt{Q}]\!] & [\![\mathtt{P} \xrightarrow{\bar{x}(y)} \mathtt{Q}]\!] & = & [\![\mathtt{P}]\!] \overset{\uparrow x}{\rightharpoonup} \lambda y.[\![\mathtt{Q}]\!] \\
[\![\mathtt{P} \xrightarrow{\tau} \mathtt{Q}]\!] & = & [\![\mathtt{P}]\!] \xrightarrow{\tau} [\![\mathtt{Q}]\!] & & &
\end{array}$$

Figure 13.3 contains a set of clauses, called $\mathbf{D}_\pi$, that encodes the operational semantics of the late transition system for the finite π-calculus. In this specification, free variables are schema variables that are assumed to be universally quantified over the clause in which they appear. These schema variables have primitive types such as a, n, and p as well as arrow types such as $n \rightarrow a$ and $n \rightarrow p$.

Note that, as a consequence of using λ-tree syntax for this specification, the usual side conditions in the original specifications of the π-calculus [Milner et al., 1992b] are no longer present. For example, the side condition that $X \neq y$

$$
\begin{array}{rccl}
\text{TAU:} & \top & \supset & \tau\ P \xrightarrow{\tau} P \\
\text{IN:} & \top & \supset & \texttt{in}\ X\ M \xrightharpoonup{\downarrow X} M \\
\text{OUT:} & \top & \supset & \texttt{out}\ x\ y\ P \xrightarrow{\uparrow xy} P \\
\text{MATCH:} & P \xrightarrow{A} Q & \supset & \texttt{match}\ x\ x\ P \xrightarrow{A} Q \\
 & P \xrightharpoonup{A} Q & \supset & \texttt{match}\ x\ x\ P \xrightharpoonup{A} Q \\
\text{SUM:} & P \xrightarrow{A} R \vee Q \xrightarrow{A} R & \supset & P + Q \xrightarrow{A} R \\
 & P \xrightharpoonup{A} R \vee Q \xrightharpoonup{A} R & \supset & P + Q \xrightharpoonup{A} R \\
\text{PAR:} & P \xrightarrow{A} P' & \supset & P \mid Q \xrightarrow{A} P' \mid Q \\
 & Q \xrightarrow{A} Q' & \supset & P \mid Q \xrightarrow{A} P \mid Q' \\
 & P \xrightharpoonup{A} M & \supset & P \mid Q \xrightharpoonup{A} \lambda n (M\,n \mid Q) \\
 & Q \xrightharpoonup{A} N & \supset & P \mid Q \xrightharpoonup{A} \lambda n (P \mid N\,n) \\
\text{RES:} & \forall n (Pn \xrightarrow{A} Qn) & \supset & \nu n.Pn \xrightarrow{A} \nu n.Qn \\
 & \forall n (Pn \xrightharpoonup{A} P'n) & \supset & \nu n.Pn \xrightharpoonup{A} \lambda m\ \nu n.P'nm \\
\text{OPEN:} & \forall y (My \xrightarrow{\uparrow Xy} M'y) & \supset & \nu y.My \xrightharpoonup{\uparrow X} M' \\
\text{CLOSE:} & P \xrightharpoonup{\downarrow X} M \wedge Q \xrightharpoonup{\uparrow X} N & \supset & P \mid Q \xrightarrow{\tau} \nu y.(My \mid Ny) \\
 & P \xrightharpoonup{\uparrow X} M \wedge Q \xrightharpoonup{\downarrow X} N & \supset & P \mid Q \xrightarrow{\tau} \nu y.(My \mid Ny) \\
\text{COM:} & P \xrightharpoonup{\downarrow X} M \wedge Q \xrightarrow{\uparrow XY} Q' & \supset & P \mid Q \xrightarrow{\tau} MY \mid Q' \\
 & P \xrightarrow{\uparrow XY} P' \wedge Q \xrightharpoonup{\downarrow X} N & \supset & P \mid Q \xrightarrow{\tau} P' \mid NY
\end{array}
$$

Figure 13.4: The inference rules in Figure 13.3 as logical formulas.

```
kind   a              type.
type   tau            a.
type   dn, up         n -> n -> a.

type   one            p -> a -> p -> o.
type   oneb           p -> (n -> a) -> (n -> p) -> o.

oneb (in X M) (dn X) M.
one (out X Y P) (up X Y) P.
one  (taup P) tau P.
one  (match X X P) A Q :- one P A Q.
oneb (match X X P) A M :- oneb P A M.
one  (plus P Q) A R :- one  P A R.
one  (plus P Q) A R :- one  Q A R.
oneb (plus P Q) A M :- oneb P A M.
oneb (plus P Q) A M :- oneb Q A M.
one  (par P Q) A (par P1 Q) :- one P A P1.
one  (par P Q) A (par P Q1) :- one Q A Q1.
oneb (par P Q) A (x\par (M x) Q) :- oneb P A M.
oneb (par P Q) A (x\par P (N x)) :- oneb Q A N.
one  (nu P) A (nu Q) :- pi x\ one  (P x) A (Q x).
oneb (nu P) A (y\ nu x\Q x y) :-
  pi x\ oneb (P x) A (Q x).
oneb (nu M) (up X) N :-
  pi y\ one (M y) (up X y) (N y).
one (par P Q) tau (nu y\ par (M y) (N y)) :-
  oneb P (dn X) M , oneb Q (up X) N.
one (par P Q) tau (nu y\ par (M y) (N y)) :-
  oneb P (up X) M , oneb Q (dn X) N.
one (par P Q) tau (par (M Y) T) :-
  oneb P (dn X) M, one Q (up X Y) T.
one (par P Q) tau (par R (M Y)) :-
  oneb Q (dn X) M,  one P (up X Y) R.
```

Figure 13.5: The λProlog specification of for the finite π-calculus.

in the open rule is implicit, since X is outside the scope of y and therefore cannot be instantiated with y (substitutions into logical expressions cannot capture bound variable names). The adequacy of our encoding is stated in the following proposition (the proof of this proposition can be found in [Tiu, 2004]).

Proposition. Let `P` and `Q` be processes and α an action. Let $\bar{n}$ be a list of free names containing the free names in `P`, `Q`, and α. The transition $\mathtt{P} \xrightarrow{\alpha} \mathtt{Q}$ is derivable in the π-calculus if and only if $\forall\bar{n}.[\![\mathtt{P} \xrightarrow{\alpha} \mathtt{Q}]\!]$ is provable from the logical theory $\mathbf{D}_\pi$.

13.5 Binary clauses

A reduced class of Horn clause, called *binary clauses*, can play an important role in modeling computation. As we argue below, they can be used to explicitly order computations whose order is left unspecified in Horn clauses: such an explicit ordering is important if one wishes to use the framework of big-step semantics to capture side-effects and concurrency. They can also be used to capture the notion of abstract machines, a common device for specifying operational semantics.

13.5.1 Continuation passing in logic programming

Continuation-passing style specifications are possible in logic programming using quantification over the type of formulas [Tarau, 1992]. In fact, it is possible to *cps transform* Horn clauses into binary clauses as follow. First, for every predicate p of type $\tau_1 \to \ldots \to \tau_n \to o$ $(n \geq 0)$, we provide a second predicate $\hat{p}$ of type $\tau_1 \to \ldots \to \tau_n \to o \to o$: that is, an additional argument of type o (the type of formulas) is added to predicate p. Thus, the atomic formula A of the form $(p\ t_1\ \ldots\ t_n)$ is similarly transformed to the term $\hat{A} = (\hat{p}\ t_1\ \ldots\ t_n)$ of type $o \to o$. Using these conventions, the cps transformation of the formula

$$\forall z_1 \ldots \forall z_m\ [(A_1 \wedge \ldots \wedge A_n) \supset A_0] \quad (m \geq 0,\ n > 0)$$

is the binary clause

$$\forall z_1 \ldots \forall z_m \forall k\ [(\hat{A}_1\ (\hat{A}_2(\cdots(\hat{A}_n\ k)\cdots))) \supset (\hat{A}_0\ k)].$$

Similarly, the cps transformation of the formula

$$\forall z_1 \ldots \forall z_m\ [A_0] \qquad \text{is} \qquad \forall z_1 \ldots \forall z_m \forall k\ [k \supset (\hat{A}_0\ k)].$$

If $\mathcal{P}$ is a finite set of Horn clauses and $\hat{\mathcal{P}}$ is the result of applying this cps transformation to all clauses in $\mathcal{P}$, then $\mathcal{P} \vdash A$ if and only if $\hat{\mathcal{P}} \vdash (\hat{A}\ \top)$.

$$(((abs\ R) \searrow (abs\ R))\ ;\ K) \supset K.$$
$$((M \searrow (abs\ R))\ ;\ (N \searrow U)\ ;\ ((R\ U) \searrow V)\ ;\ K) \supset ((app\ M\ N) \searrow V)\ ;\ K.$$

Figure 13.6: Binary version of call-by-value evaluation.

Consider again the presentation of call-by-value evaluation given by the Figure 13.2. In order to add side-effecting features, this specification must be made more explicit: in particular, the exact order in which M, N, and $(R\ U)$ are evaluated must be specified. The cps transformation of that specification is given in Figure 13.6: there, evaluation is denoted by a ternary predicate which is written using both the $\searrow$ arrow and a semicolon: e.g., the relation "M evaluates to V with the continuation K" is denoted by $(M \searrow V)\ ;\ K$. If we write this evaluation predicate as `evalc` then the λProlog specification of the formulas in Figure 13.6 can be written as follows.

```
type evalc    term -> term -> o -> o.

evalc (abs R) (abs R) K :- K.
evalc (app M N) V K :- evalc M (abs R) (evalc N U
                                       (evalc (R U) V K)).
```

In this specification, goals are now sequenced in the sense that bottom-up proof search is forced to construct a proof of one evaluation pair before others such pairs. The goal $((M \searrow V)\ ;\ \top)$ is provable if and only if V is the call-by-value result of M. The order in which evaluation is executed is now forced not by the use of logical connectives but by the use of the non-logical constant $(\cdot \searrow \cdot)\ ;\ \cdot$.

13.5.2 Abstract Machines

Abstract machines, which are often used to specify operational semantics, can be encoded naturally using binary clauses. To see this, consider the following definition of *Abstract Evaluation System* (AES) which generalizes the notion of abstract machines [Hannan and Miller, 1992].

Recall that a term rewriting system is a pair (Σ, R) such that Σ is a signature and R is a set of directed equations $\{l_i \Rightarrow r_i\}_{i \in I}$ with $l_i, r_i \in T_\Sigma(X)$ and $\mathcal{V}(r_i) \subseteq \mathcal{V}(l_i)$. Here, $T_\Sigma(X)$ denotes the set of first-order terms with constants from the signature Σ and free variables from X, and $\mathcal{V}(t)$ denotes the set of free variables occurring in t. An *abstract evaluation system* is a quadruple (Σ, R, ρ, S) such that the pair $(\Sigma, R \cup \{\rho\})$ is a term rewriting system, ρ is not a member of R, and $S \subseteq R$.

$$M \Rightarrow \langle\, \mathit{nil},\ M,\ \mathit{nil}\,\rangle$$

$$\begin{array}{l}
\langle\, E,\ \lambda M,\ X :: S\rangle \Rightarrow \langle X :: E,\ M,\ S\rangle \\
\langle\, E,\ M \,\hat{}\, N,\ S\rangle \Rightarrow \langle\, E,\ M,\ \{E, N\} :: S\rangle \\
\langle \{E', M\} :: E,\ 0,\ S\rangle \Rightarrow \langle\, E',\ M,\ S\rangle \\
\langle\, X :: E,\ n+1,\ S\rangle \Rightarrow \langle\, E,\ n,\ S\rangle
\end{array}$$

$$\langle\, E,\ \lambda M,\ \mathit{nil}\,\rangle \Rightarrow \{E, \lambda M\}$$

$$M \Rightarrow \langle \mathit{nil},\ \mathit{nil},\ M :: \mathit{nil},\ \mathit{nil}\,\rangle$$

$$\begin{aligned}
\langle S,\ E,\ \lambda M :: C,\ D\rangle &\Rightarrow \langle \{E, \lambda M\} :: S,\ E,\ C,\ D\rangle \\
\langle S,\ E,\ (M \,\hat{}\, N) :: C,\ D\rangle &\Rightarrow \langle S,\ E,\ M :: N :: ap :: C,\ D\rangle \\
\langle S,\ E,\ n :: C,\ D\rangle &\Rightarrow \langle nth(n, E) :: S,\ E,\ C,\ D\rangle \\
\langle X :: \{E', \lambda M\} :: S,\ E,\ ap :: C,\ D\rangle &\Rightarrow \langle \mathit{nil},\ X :: E',\ M :: \mathit{nil},\ (S, E, C) :: D\rangle \\
\langle X :: S,\ E,\ \mathit{nil},\ (S', E', C') :: D\rangle &\Rightarrow \langle X :: S',\ E',\ C',\ D\rangle
\end{aligned}$$

$$\langle X :: S,\ E,\ \mathit{nil},\ \mathit{nil}\rangle \Rightarrow X$$

Figure 13.7: The Krivine machine (top) and SECD machine (bottom).

Evaluation in an AES is a sequence of rewriting steps with the following restricted structure. The first rewrite rule must be an instance of the ρ rule. This rule can be understood as "loading" the machine to an initial state given an input expression. The last rewrite step must be an instance of a rule in S: these rules denote the successful termination of the machine and can be understood as "unloading" the machine and producing the answer or final value. All other rewrite rules are from R. We also make the following significant restriction to the general notion of term rewriting: all rewriting rules must be applied to a term at its root. This restriction significantly simplifies the computational complexity of applying rewrite rules during evaluation in an AES. A term $t \in T_\Sigma(\emptyset)$ *evaluates* to the term s (with respect to the AES (Σ, R, ρ, S)) if there is a series of rewriting rules satisfying the restrictions above that rewrites t into s.

The SECD machine [Landin, 1964] and Krivine machine [Curien, 1990] are both AESs and variants of these are given in Figure 13.7. There, the syntax for λ-terms uses de Bruijn notation with $\hat{}$ (infix) and λ as the constructors for application and abstraction, respectively, and $\{E, M\}$ denotes the closure of term M with environment E. The first rule given for each machine is the "load" rule or ρ of their AES description. The last rule given for each is the "unload" rule. (In each of these cases, the set S is a singleton.) The remaining rules are state transformation rules, each one moving the machine through a computation step.

A state in the Krivine machine is a triple $\langle E, M, S\rangle$ in which E is an environment, M is a single term to be evaluated and S is a stack of arguments. A state in the SECD machine is a quadruple $\langle S, E, C, D\rangle$ in which S is a stack of computed values, E is an environment (here just a list of terms), C is a list of commands (terms to be evaluated) and D is a dump or saved state. The expression $nth(n, E)$, used to access variables in an environment, is treated as a function that returns the $n + 1^{st}$ element of the list E. Although Landin's original description of the SECD machine used variables names, our use of de Bruijn numerals does not change the essential mechanism of that machine.

There is a natural and immediate way to see a given AES as a set of binary clauses. Let *load*, *unload*, and *rewrite* be three predicates of one argument each. Given the AES (Σ, R, ρ, S) let $\mathcal{B}$ be the set of binary clauses composed of the following three groups of formulas:

1. $\forall\hat{x}$ [*rewrite* $r \supset$ *load* l] where ρ is the rewrite rule $l \Rightarrow r$,
2. for every rewrite rule $l \Rightarrow r$ in R, add one clause of the form $\forall\hat{x}$ [*rewrite* $r \supset$ *rewrite* l], and
3. for every rewrite rule $l \Rightarrow r$ in S, add one clause of the form $\forall\hat{x}$ [*unload* $r \supset$ *rewrite* l].

It is then easy to show that if we start with term t and evaluate it to get s (this can be a nondeterministic relationship) then from the set of clauses $\mathcal{B}$ we can prove *unload* $t \supset$ *load* s. In particular, if this implication is provable from $\mathcal{B}$ then it has a proof of the form displayed in Figure 13.8: there, only synthetic inference rules (Section 5.8) are displayed. The transitions of the abstract machine can be read directly from this proof: given the term s, the machine's state is initialized to be s_1, which is then repeatedly rewritten yielding the sequence of terms $s_2, \ldots, s_n$, at which point the machine is unloaded to get the value t. For more about translating SOS specifications directly into abstract machines, see [Hannan and Miller, 1992].

In order to motivate our next operational semantic framework, consider the problem of using binary clauses to specifying side-effects, exceptions, and concurrent (multi-threaded) computation. Since all the dynamics of computation is represented via term structures (say, within $s, s_1, \ldots, s_n, t$) all the information about these threads, reference cells, exceptions, etc., must be maintained as, say, lists within these terms. Such an approach to specifying these features

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathit{unload}\ t \vdash \mathit{unload}\ t}}{\mathit{unload}\ t \vdash \mathit{rewrite}\ s_n}}{\vdots}}{\mathit{unload}\ t \vdash \mathit{rewrite}\ s_i}}{\vdots}}{\mathit{unload}\ t \vdash \mathit{rewrite}\ s_1}}{\mathit{unload}\ t \vdash \mathit{load}\ s}$$

Figure 13.8: A proof involving synthetic rules based on the formulas in $\mathcal{B}$ related to the execution of an abstract machine.

of a programming language lacks modularity and makes little use of logic. We now consider extending binary clauses so that these additional features have a much more natural and modular specification.

13.6 Linear logic

In Sections 7.4 and 7.6, we illustrated how linear logic can be used to capture multiset rewriting. Given that many aspects of computation can be captured using multiset rewriting, it is possible to describe a subset of linear logic that includes binary clauses but provides a natural means to capture side effects and concurrency. The examples in this section are adapted from [Miller, 1996].

13.6.1 Adding a counter to evaluation

Consider again the binary clause example given in Figure 13.6. As we showed in Section 6.5, the top-level intuitionistic implication $\Rightarrow$ of Horn clauses can be rewritten as the linear implication $\multimap$ without changing the operational reading of proof search. With this change, the binary clauses in that figure are also an example of multiset rewriting: in particular, one atom is repeatedly replace by another atom (until the atom is replaced by a final continuation). In this way, binary clauses can be seen as modeling single-threaded computation. Now that we have embedded binary clauses within the richer setting of linear logic, it is easy to see how "multi-threaded" computations might be organized. We present a couple of examples here.

Consider adding to the untyped λ-calculus a single global counter that can be read and incremented. In particular, we shall place all integers into type *tm* and add two additional constructors of type *tm*, namely *get* and *inc*. The intended operational semantics of these two constants is that evaluating the first returns the current value of the counter and evaluating the second increments the counter's value and returns the counter's old value. We also assume that integers are values: that is, for every integer i the clause $\forall k(k \multimap (i \searrow i) \mathbin{;} k)$ is part of the evaluator's specification. The multiset rewriting specification of these two additional constructors can be given as the two formulas

$$\forall K \forall V\ (r\ V \parr K \multimap ((\mathit{get} \searrow V) \mathbin{;} K) \parr r\ V) \text{ and}$$
$$\forall K \forall V\ (r\ (V+1) \parr K \multimap ((\mathit{inc} \searrow V) \mathbin{;} K) \parr r\ V).$$

Here, the atom of the form $(r\ x)$ denotes the "r-register" with value x. Let $\mathcal{D}$ contain the two formulas in Figure 13.6, the two formulas displayed above, and the formulas mentioned above describing the evaluation of integers. Then $\mathcal{D}$ is a specification of the call-by-value evaluator with one global counter in the sense that the logical judgment

$$!\,\mathcal{D} \vdash ((M \searrow V) \mathbin{;} \top) \parr r\ 0$$

holds exactly when we expect the program M to evaluation to V in the setting when the register r is initialized to 0.

Of course, the name of the register should not be a part of the specification of a counter. As described in Section 9.6, higher-order quantification over r makes it possible to hide the name of this register. In Figure 13.9 there are three specifications, E_1, E_2, and E_3, of a counter: all three specifications store the counter's value in an atomic formula as the argument of the predicate r. In these three specifications, the predicate r is existentially quantified over the specification in which it is used so that the atomic formula that stores the counter's value is itself local to the counter's specification. The first two specifications store the counter's value on the right of the sequent arrow, and reading and incrementing the counter occurs via a synchronization between an $\searrow$-atom and an r-atom. In the third specification, the counter is stored as a linear assumption on the left of the sequent arrow, and synchronization is not used: instead, the linear assumption is "destructively" read and then rewritten in order to specify the *get* and *inc* functions (similar to the examples in Section 7.4). Finally, in the first and third specifications, evaluating the *inc* symbol causes 1 to be added to the counter's value. In the second specification, evaluating the *inc* symbol causes 1 to be subtracted from the counter's value: to compensate for this unusual implementation of *inc*, reading a counter in the second specification returns the negative of the counter's value.

Although these three specifications of a global counter are different, they should be equivalent in the sense that the process of evaluating terms cannot tell them apart. Although there are several ways that the equivalence of such counters can be argued, the specifications of these counters are, in fact, *logically* equivalent.

$$\begin{aligned}
E_1 = \exists r[\ \ & (r\ 0)^\perp \otimes \\
& !\forall K \forall V\ (r\ V \parr K \multimap ((get \searrow V)\ ;\ K) \parr r\ V) \otimes \\
& !\forall K \forall V\ (r\ (V+1) \parr K \multimap ((inc \searrow V)\ ;\ K) \parr r\ V)]
\end{aligned}$$

$$\begin{aligned}
E_2 = \exists r[\ \ & (r\ 0)^\perp \otimes \\
& !\forall K \forall V\ (r\ V \parr K \multimap ((get \searrow (-V))\ ;\ K) \parr r\ V) \otimes \\
& !\forall K \forall V\ (r\ (V-1) \parr K \multimap ((inc \searrow (-V))\ ;\ K) \parr r\ V)]
\end{aligned}$$

$$\begin{aligned}
E_3 = \exists r[\ \ & (r\ 0) \otimes \\
& !\forall K \forall V\ (r\ V \otimes (r\ V \multimap K) \multimap ((get \searrow V)\ ;\ K)) \otimes \\
& !\forall K \forall V\ (r\ V \otimes (r\ (V+1) \multimap K) \multimap ((inc \searrow V)\ ;\ K))]
\end{aligned}$$

Figure 13.9: Three specifications of a global counter.

Proposition 13.1. *The three entailments $E_1 \vdash E_2$, $E_2 \vdash E_3$, and $E_3 \vdash E_1$ are provable in linear logic.*

Proof. The proof of each of these entailments proceeds (in a bottom-up fashion) by choosing an eigenvariable to instantiate the existential quantifier on the left-hand side and then instantiating the right-hand existential quantifier with some term involving that eigenvariable. Assume that in all three cases, the eigenvariable selected is the predicate symbol s. Then the first entailment is proved by instantiating the right-hand existential with $\lambda x.s\ (-x)$; the second entailment is proved using the substitution $\lambda x.(s\ (-x))^\perp$; and the third entailment is proved using the substitution $\lambda x.(s\ x)^\perp$. The proof of the first two entailments must also use the identities $-0 = 0$, $-(x+1) = -x-1$, and $-(x-1) = -x+1$. The proof of the third entailment requires no such identities. $\square$

Clearly, logical equivalence is a strong equivalence: it immediately implies that evaluation cannot tell the difference between any of these different specifications of a counter. For example, assume $E_1 \vdash (M \searrow V)\ ;\ \top$. Then by the cut inference rule (modus ponens) and the above proposition, we have $E_2 \vdash (M \searrow V)\ ;\ \top$.

It is possible to generalize a bit the previous example involving a single global counter to languages that have the ability to generate references dynamically, such as those found in, say, Algol or Standard ML [Chirimar, 1995; Miller, 1996].

$$\begin{aligned}
K &\multimap (none \searrow none) \mathbin{;} K.\\
(E \searrow V) \mathbin{;} K &\multimap ((guard\ E) \searrow (guard\ V)) \mathbin{;} K.\\
(E \searrow V) \mathbin{;} K &\multimap ((poll\ E) \searrow (poll\ V)) \mathbin{;} K.\\
(E \searrow V) \mathbin{;} K &\multimap ((receive\ E) \searrow (receive\ V)) \mathbin{;} K.\\
(E \searrow V) \mathbin{;} K &\multimap ((some\ E) \searrow (some\ V)) \mathbin{;} K.\\
(E \searrow U) \mathbin{;} ((F \searrow V) \mathbin{;} K) &\multimap ((choose\ E\ F) \searrow (choose\ U\ V)) \mathbin{;} K.\\
(E \searrow U) \mathbin{;} ((F \searrow V) \mathbin{;} K) &\multimap ((transmit\ E\ F) \searrow (transmit\ U\ V)) \mathbin{;} K.\\
(E \searrow U) \mathbin{;} ((F \searrow V) \mathbin{;} K) &\multimap ((wrap\ E\ F) \searrow (wrap\ U\ V)) \mathbin{;} K.
\end{aligned}$$

Figure 13.10: These CML-like constructors evaluate to themselves.

13.6.2 Specification of Concurrency primitives

The concurrency primitives found in Concurrent ML (CML) [Reppy, 1991] can also be specified in linear logic. We assume that the reader has some familiarity with this extension to ML.

Consider extending the untyped λ-calculus with the following constructors.

$$\begin{aligned}
none &: tm.\\
guard, poll, receive, some, sync &: tm \to tm.\\
choose, transmit, wrap &: tm \to tm \to tm.\\
spawn, newchan &: (tm \to tm) \to tm.
\end{aligned}$$

The meaning of these constructors is then given using the linear logic formulas in Figure 13.10 and Figure 13.11. The clauses in Figure 13.10 specify the straightforward evaluation rules for the eight data constructors. In Figure 13.11, the predicate *event* is of type $tm \to tm \to o \to o$ and is used to store in the multiset *events*, a technical aspect of this semantic specification. The first three clauses of that figure defined the meaning of the three special forms *sync*, *spawn*, and *newchan*. The remaining clauses specify the *event* predicate.

The formulas in Figure 13.11 allow for multiple threads of evaluation. Evaluation of the *spawn* function initiates a new evaluation thread. The *newchan* function causes a new eigenvariable to be picked (via the $\forall c$ quantification) and then to assume that that eigenvariable is a value (via the assumption $\forall I(I \multimap (c \searrow c) \mathbin{;} I)$): such a new value can be used to designate new channels for use in synchronization. The *sync* primitive allows for synchronization between threads: its use causes an "evaluation thread" to become an "event thread." The behaviors of event threads are described by the remaining clauses in Figure 13.11. The primitive events are *transmit* and *receive* and they rep-

$$\begin{array}{r}
\mathit{eval}\ E\ U\ (\mathsf{event}\ U\ V\ K) \multimap ((\mathit{sync}\ E) \searrow V)\ ;\ K.\\
(((R\ \mathit{unit}) \searrow \mathit{unit})\ ;\ \bot) \parr K \multimap ((\mathit{spawn}\ R) \searrow \mathit{unit})\ ;\ K.\\
\forall c(\forall I(I \multimap (c \searrow c)\ ;\ I) \Rightarrow ((R\ c) \searrow V)\ ;\ K) \multimap ((\mathit{newchan}\ R) \searrow V)\ ;\ K.
\end{array}$$

$$K \parr L \multimap \mathsf{event}\ (\mathit{receive}\ C)\ V\ K \parr \mathsf{event}\ (\mathit{transmit}\ C\ V)\ \mathit{unit}\ L$$

$$\begin{array}{r}
\mathsf{event}\ E\ V\ K \multimap \mathsf{event}\ (\mathit{choose}\ E\ F)\ V\ K.\\
\mathsf{event}\ F\ V\ K \multimap \mathsf{event}\ (\mathit{choose}\ E\ F)\ V\ K.\\
\mathsf{event}\ E\ U\ (((\mathit{app}\ F\ U) \searrow V)\ ;\ K) \multimap \mathsf{event}\ (\mathit{wrap}\ E\ F)\ V\ K.\\
((\mathit{app}\ F\ \mathit{unit}) \searrow U)\ ;\ (\mathsf{event}\ U\ V\ K) \multimap \mathsf{event}\ (\mathit{guard}\ F)\ V\ K.\\
(\mathsf{event}\ E\ U\ \top)\ \&\ K \multimap \mathsf{event}\ (\mathit{poll}\ E)\ (\mathit{some}\ E)\ K.\\
K \multimap \mathsf{event}\ (\mathit{poll}\ E)\ \mathit{none}\ K.
\end{array}$$

Figure 13.11: Specifications of some primitives similar to those in Concurrent ML.

resent two halves of a synchronization between two event threads. Note that the clause describing their meaning is the only clause in Figure 13.11 that has a head with more than one atom. The non-primitive events *choose*, *wrap*, *guard*, and *poll* are reduced to other calls to *event* and $\searrow$. The *choice* event is implemented as a local, nondeterministic choice. (Specifying global choice, as in CCS [Milner, 1989], would be much more involved.) The *wrap* and *guard* events chain together evaluation and synchronization but in direct orders.

The only use of *additive* linear logic connectives, in particular & and $\top$, in any of our semantic specifications is in the specification of polling: in an attempt to synchronize with (*poll E*) (with the continuation K) the goal

$$(\mathsf{event}\ E\ U\ \top)\ \&\ K$$

is attempted (for some unimportant term U). Thus, a copy of the current evaluation threads is made and ($\mathsf{event}\ E\ U\ \top$) is attempted in one of these copies. This atom is provable if and only if there is a complementary event for E in the current environment, in which case, the continuation $\top$ brings us to a quick completion and the continuation K is attempted in the original and unspoiled context of threads. If such a complementary event is not present, then the other clause for computing a polling event can be used, in which case, the result of the poll is *none*, which signals such a failure. The semantics of polling, unfortunately, is not exactly as intended in CML since it is possible to have a polling event return *none* even if the event being tested could be synchronized. This analysis of polling is similar to the analysis of testing in process calculus as described in [Miller, 1993].

The PhD thesis of Chirimar [Chirimar, 1995] presents a linear logic specification of a programming language motivated by Standard ML [Milner et al., 1990]. In particular, a specification for the call-by-value λ-calculus is provided, and then modularly extended with the specifications of references, exceptions, and continuations: each of these features is specified without complicating the specifications of other the features.

13.7 Bibliographic notes

A distinction is often made between the *static semantics* and the *dynamic semantics* of programming languages (see, for example, Clement et al. [1986]). Static semantics refers to properties of program text that can be inferred by a compiler: typing is a typical example of a static semantics. Dynamic semantics refers to properties of programs that can be inferred by executing programs: termination is a typical examples of dynamic semantics. In this chapter, we have limited ourselves to the specification of dynamic semantics of some simple programming languages and to the π-calculus. Logic programming and its concomitant technologies of unification and proof search has also had an important role in specifying the static semantics of programming languages, particular, in type checking and type inference: see, for example, the so-call Hindley-Milner approach to type inference [Hindley, 1969; Milner, 1978].

There has been long-standing interest in being able to formally specify and reason about the operational semantic descriptions of programming languages. Early dynamic semantics used both small-step specifications and big-step semantics have been formalized as logical specifications [Despeyroux, 1988; Hannan, 1993]. Such logical specifications have also been used to develop the formal metatheory of those programming languages: see, for example, [Despeyroux, 1986; Hannan and Pfenning, 1992; McDowell and Miller, 2002; Pfenning and Schürmann, 1999]. One explicit approach to reasoning about static and dynamic semantics is the two-level logic approach [McDowell and Miller, 2002; Miller, 2010; Gacek et al., 2012] which has been formally supported in the Abella proof assistant Baelde et al. [2014].

The specification of the π-calculus Milner et al. [1992a] in Figure 13.3 is taken from [Miller and Palamidessi, 1999]. The general outline of this chapter is based on the short article [Miller, 2008].

Solutions to selected exercises

Solution to Exercise 2.3 (page 15). E_2 normalizes to the Church encoding of 16. In general, E_n has the λ-normal form that encodes the number

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\Big\}{n+1}$$

There are $n+1$ occurrences of 2 in this expression.

Solution to Exercise 2.4 (page 15). The abstraction $(\lambda x.w)$ is vacuous, i.e., x is not free in its scope (which is just the variable w). Since substitution is capture-avoiding, every instance of that term remains a vacuous abstraction. Since the term $\lambda y.y$ is not a vacuous abstraction, no such expression for N is possible.

Solution to Exercise 2.5 (page 17). The proof of uniqueness is a simple induction on the structure of typing judgment proofs. For the second part of this question, let Σ be the empty signature, let t be the λ-term $\lambda x.x$, and assume that S contains two different primitive sorts a and b. Then we have both $\Sigma \Vdash t : a \to a$ and $\Sigma \Vdash t : b \to b$.

Solution to Exercise 3.2 (page 29). The multiplicative version of the $\wedge$R rule is

$$\frac{\Sigma : \Gamma \vdash \Delta, B \qquad \Sigma : \Gamma' \vdash \Delta', C}{\Sigma : \Gamma, \Gamma' \vdash \Delta, \Delta', B \wedge C}\ \wedge\mathrm{R}^m.$$

The following derivation shows that weakening and the additive $\wedge$R rule can be used to derive the multiplicative $\wedge\mathrm{R}^m$ rule.

$$\dfrac{\dfrac{\Sigma : \Gamma \vdash \Delta, B}{\Sigma : \Gamma, \Gamma' \vdash \Delta, \Delta', B}\,wR, wL \qquad \dfrac{\Sigma : \Gamma \vdash \Delta, C}{\Sigma : \Gamma, \Gamma' \vdash \Delta, \Delta', C}\,wR, wL}{\Sigma : \Gamma, \Gamma' \vdash \Delta, \Delta', B \wedge C}\,\wedge\mathrm{R}$$

The following derivation shows that contraction and the multiplicative $\wedge\mathrm{R}^m$ rule can be used to derive the additive $\wedge$R rule.

$$\dfrac{\dfrac{\Sigma : \Gamma \vdash \Delta, B \qquad \Sigma : \Gamma \vdash \Delta, C}{\Sigma : \Gamma, \Gamma \vdash \Delta, \Delta, B \wedge C}\ \wedge\mathrm{R}^m}{\Sigma : \Gamma \vdash \Delta, B \wedge C}\ cR, cL$$

Similarly, the $\wedge\mathrm{L}^m$ rule can be derived from the $\wedge$L rule with contraction.

$$\dfrac{\dfrac{\dfrac{\Sigma : \Gamma, B, C \vdash \Delta}{\Sigma : \Gamma, B \wedge C, C \vdash \Delta}\ \wedge\mathrm{L}}{\Sigma : \Gamma, B \wedge C, B \wedge C \vdash \Delta}\ \wedge\mathrm{L}}{\Sigma : \Gamma, B \wedge C \vdash \Delta}\ cL$$

Finally, the $\wedge$L rule can be derived from the $\wedge\mathrm{L}^m$ rule with weakening.

$$\dfrac{\dfrac{\Sigma : \Gamma, B \vdash \Delta}{\Sigma : \Gamma, B, C \vdash \Delta}\ wL}{\Sigma : \Gamma, B \wedge C \vdash \Delta}\ \wedge\mathrm{R}^m$$

Solution to Exercise 4.1 (page 40). Since $\sqrt{2}^{\sqrt{2}}$ is either rational or irrational we have two cases to consider. In the case that $\sqrt{2}^{\sqrt{2}}$ is rational, then set $a = b = \sqrt{2}$. In the case that $\sqrt{2}^{\sqrt{2}}$ is irrational, then set $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. A more satisfying proof of this fact results from assigning $a = \sqrt{2}$ and $b = \log_2 9$. R. Kuzmin [1930] proved that $\sqrt{2}^{\sqrt{2}}$ is transcendental.

Solution to Exercise 4.3 (page 43). Of these examples, (3), (4), (5), (6), and (7) all have **C**-proofs but no **I**-proofs. A **C**-proof of (5) is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\ init}{p \vdash q, p}\ wR}{\vdash p \supset q, p}\ \multimap\mathrm{R} \qquad \dfrac{}{p \vdash p}\ init}{(p \supset q) \supset p \vdash p, p}\ \supset\mathrm{L}}{(p \supset q) \supset p \vdash p}\ cR}{\cdot \vdash ((p \supset q) \supset p) \supset p}\ \multimap\mathrm{R}$$

Solution to Exercise 4.5 (page 44). The list of pairs for which entailment is provable in classical logic is

$$\{\langle A, \neg\neg A\rangle, \langle \neg\neg A, A\rangle, \langle \neg A, \neg\neg\neg A\rangle, \langle \neg\neg\neg A, \neg A\rangle, \}$$

The list of pairs for which entailment is provable in intuitionistic logic is the same list except that the pair $\langle \neg\neg A, A\rangle$ is removed.

Solution to Exercise 4.7 (page 44). Assume that S contains the primitive types i and j. The following is an **I**-proof.

$$\dfrac{\dfrac{f : i \to j, y : i \Vdash (f\ y) : j \qquad \dfrac{}{f : i \to j, y : i : \cdot \vdash \boldsymbol{t}}\ \boldsymbol{t}\mathrm{R}}{\dfrac{f : i \to j, y : i : \cdot \vdash \exists_j x\, \boldsymbol{t}}{f : i \to j, y : i : \cdot \vdash \forall_i y \exists_j x\, \boldsymbol{t}}\ \forall\mathrm{R}}\ \exists\mathrm{R}}{f : i \to j : \cdot \vdash (\exists_j x\, \boldsymbol{t}) \vee (\forall_i y \exists_j x\, \boldsymbol{t})}\ \vee\mathrm{R}$$

The following is an **C**-proof.

$$f : i \to j : \cdot \vdash (\exists_j x\, \boldsymbol{t}) \vee (\forall_i x\, \boldsymbol{f})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{f : i \to j, x : i \Vdash (f\ x) : j \qquad \dfrac{}{f : i \to j, x : i : \cdot \vdash \boldsymbol{t}, \boldsymbol{f}}\ \boldsymbol{t}\mathrm{R}}{f : i \to j, x : i : \cdot \vdash \exists_j x\, \boldsymbol{t}, \boldsymbol{f}}\ \exists\mathrm{R}}{f : i \to j : \cdot \vdash \exists_j x\, \boldsymbol{t}, \forall_i x\, \boldsymbol{f}}\ \forall\mathrm{R}}{f : i \to j : \cdot \vdash (\exists_j x\, \boldsymbol{t}) \vee (\forall_i x\, \boldsymbol{f}), (\exists_j x\, \boldsymbol{t}) \vee (\forall_i x\, \boldsymbol{f})}\ \vee\mathrm{R} \times 2}{f : i \to j : \cdot \vdash (\exists_j x\, \boldsymbol{t}) \vee (\forall_i x\, \boldsymbol{f})}\ cR$$

There is no **I**-proof of this sequent since the contraction of the right is necessary to complete a proof. In both this example and in Exercise 4.3(4), completing a proof requires two subformulas separated by a disjunction to "communicate" in the sense that one disjunction puts into the sequent context some item (here, an eigenvariable and in Exercise 4.3(4) an assumption) that the other disjunct needs. This communication can happen in the proof if that disjunction is contracted on the right.

Solution to Exercise 4.9 (page 45). We provide a high-level outline of the proof: various details need to be filled in.

For one direction, we shall show how to transform a **C**-proof with a generalized restart rule to a **C**-proof without restart. Since **I**-proofs are **C**-proofs, this establishes the forward implication. Restarts can be removed one-by-one via the following transformation.

$$\dfrac{\dfrac{\dfrac{\overset{\Xi}{\Sigma : \Gamma \vdash B, \Delta}}{\Sigma : \Gamma \vdash C, \Delta}\ \mathit{Restart}}{\vdots}}{\Sigma' : \Gamma' \vdash B, \Delta'} \qquad \Longrightarrow \qquad \dfrac{\dfrac{\dfrac{\dfrac{\overset{\Xi}{\Sigma : \Gamma \vdash B, \Delta}}{\Sigma : \Gamma \vdash C, B, \Delta}\ wR}{\vdots}}{\Sigma' : \Gamma' \vdash B, B, \Delta'}\ cR}{\Sigma' : \Gamma' \vdash B, \Delta'}$$

That is, the restart rule can be implemented using a contraction and a weakening on the right. It is easy to confirm that the formula B can be added to all possible inference rules below this occurrence of the restart rule.

For a sketch of the converse direction, consider a **C**-proof. Mark a formula on the right-hand side of every sequent as follows. The single formula on the right of the endsequent is marked (assuming that we start proof search with a single formula to prove). If the last inference rule of the proof is a left-introduction rule, then the marked occurrence of the formula in the conclusion is also marked in all the premises. If the last inference rule is a right-introduction rule, then we have two cases: If the introduced formula is already marked, then mark its subformulas that appear in the right-hand side of any premise (for example, if the marked formula is $A \Rightarrow B$ then mark B in the premise; if the marked formula is $A \wedge B$ then mark A in one premise and B in the other; etc). Otherwise, the right-hand formula introduced is not marked, in which case, we have a *marking break*, and we mark in the premises of the inference rules the subformulas of the right-hand formula introduced and continue. The only other rules that might be applied are: cL, in which case the marked formula on the right persists from conclusion to premise; cL, in which case, if the marked formula is the one contracted then select one of its copies to mark in the premise, otherwise, the marked formula persists in the premise; and *init*, in which case, if the marked formula on the right is not the same as the formula on the left, then this occurrence of the *init* rule is also a marking break.

To illustrate this notion of marking formulas, consider the following annotated **C**-proof.

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{init^*}
\UnaryInfC{$p \vdash p, q^*, p \supset q, p \vee (p \supset q)$}
\RightLabel{$\multimap$R}
\UnaryInfC{$\vdash p, (p \supset q)^*, p \supset q, p \vee (p \supset q)$}
\RightLabel{cR}
\UnaryInfC{$\vdash p, (p \supset q)^*, p \vee (p \supset q)$}
\RightLabel{$\vee$R*}
\UnaryInfC{$\vdash p^*, p \vee (p \supset q), p \vee (p \supset q)$}
\RightLabel{cR}
\UnaryInfC{$\vdash p^*, p \vee (p \supset q)$}
\RightLabel{$\vee$R}
\UnaryInfC{$\vdash p \vee (p \supset q)^*, p \vee (p \supset q)$}
\RightLabel{cR}
\UnaryInfC{$\vdash p \vee (p \supset q)^*$}
\end{prooftree}
$$

Here, an asterisk is used to indicate marked formulas and to indicate which inference rules correspond to marking gaps.

Now the **I**-proof with Restart is built as follows. For sequents that are the conclusion of a rule that is not a marking break, delete all non-marked formula on the right. For sequents that are the conclusion of a rule that is a marking break, then this one inference rule become two: an instance of the Restart rule must be inserted and then the version of the inference rule corresponding to the marking break is put into the proof with the non-marked right-hand formulas deleted.

For example, performing this transformation on the **C**-proof yields the following structure.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\,\mathit{init}}{p \vdash q}\,\mathit{Restart}}{\vdash p \supset q}\,{\multimap}\mathrm{R}}{\vdash p \supset q}\,\mathit{cR}}{\vdash p \lor (p \supset q)}\,\lor\mathrm{R}}{\vdash p}\,\mathit{Restart}}{\vdash p}\,\mathit{cR}}{\vdash p \lor (p \supset q)}\,\lor\mathrm{R}}{\vdash p \lor (p \supset q)}\,\mathit{cR}$$

This sequence of rules is not yet an **I**-proof: there are three occurrences of cR that are not allowed in **I**-proofs: these can either be deleted or reclassified as Restart rules.

Solution to Exercise 4.15 (page 49). Let Π_1 and Π_2 be the following proofs of $p \vdash \boldsymbol{f}$ and $\vdash p$, respectively.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\,\mathit{init} \qquad \dfrac{}{\boldsymbol{f} \vdash \boldsymbol{f}}\,\mathit{init}}{p, p \supset \boldsymbol{f} \vdash \boldsymbol{f}}\,{\supset}\mathrm{L}}{p, p \vdash \boldsymbol{f}}\,\mathit{defL}}{p \vdash \boldsymbol{f}}\,\mathit{cL} \qquad \dfrac{\dfrac{\begin{matrix}\Pi_1 \\ p \vdash \boldsymbol{f}\end{matrix}}{\vdash p \supset \boldsymbol{f}}\,{\multimap}\mathrm{R}}{\vdash p}\,\mathit{defR}$$

Clearly, by defining p to be $\neg p$ (hence, the equivalence $p \equiv \neg p$ is provable), one is asking for trouble. It turns out that if the ambient logic does not have the contraction rules (such as in linear logic), it is not possible for such a problematic definition to yield an inconsistency [Girard, 1992; Schroeder-Heister, 1993].

Solution to Exercise 4.17 (page 50). Let D_k be the formula $\forall x(p\ x \supset p\ (f^{2^k} x)$ sequent $(k > 1)$. Prove that D_{k+1} can be proved from D_k. Show how these lemmas can be organized into a complete proof of, for example, $p(f^{256}a)$.

Solution to Exercise 4.20 (page 54). The following inference rules can used to prove the invertibility of $\lor$L and $\forall$R. The remaining two cases can be proved in a similar fashion.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\,\mathit{init}}{B \vdash B \lor C}\,\lor\mathrm{R} \qquad \begin{matrix}\Xi \\ \Gamma, B \lor C \vdash \Delta\end{matrix}}{\Gamma, B \vdash \Delta}\,\mathit{cut} \qquad \dfrac{\dfrac{\dfrac{}{C \vdash C}\,\mathit{init}}{C \vdash B \lor C}\,\lor\mathrm{R} \qquad \begin{matrix}\Xi \\ \Gamma, B \lor C \vdash \Delta\end{matrix}}{\Gamma, C \vdash \Delta}\,\mathit{cut}}{\Gamma, B \lor C \vdash \Delta}\,\lor\mathrm{L}$$

$$\cfrac{\cfrac{\begin{array}{c}\Xi\\ \Sigma' : \Gamma \vdash \forall_\tau x.B, \Delta\end{array} \quad \cfrac{\cfrac{}{\Sigma' : B[c/x] \vdash B[c/x]}\, \mathit{init}}{\Sigma' : \forall_\tau x.B \vdash B[c/x]}\, \forall\mathrm{L}}{\Sigma' : \Gamma \vdash B[c/x], \Delta}\, \mathit{cut}}{\Sigma : \Gamma \vdash \forall_\tau x.B, \Delta}\, \forall\mathrm{R}.$$

Here, Σ' is the signature $\Sigma, c : \tau$ and c is not declared in Σ. Note that if we start with a proof Ξ of the sequent $\Sigma : \Gamma \vdash \forall_\tau x.B, \Delta$ then it is a simple matter to view Ξ as a proof of $\Sigma' : \Gamma \vdash \forall_\tau x.B, \Delta$.

Solution to Exercise 5.7 (page 64). Let the Σ-formulas $D_0, \ldots, D_n$ $(n \geq 0)$ be Horn clauses using description (5.3). Thus, D_0 is of the form

$$\forall \bar{x}_1.(A_1 \supset \cdots \supset (\forall \bar{x}_m.A_m \supset \forall \bar{x}_0.A_0))$$

where $m \geq 0$ and $\bar{x}_0, \ldots \bar{x}_m$ are lists of variables, all of which are distinct. It is an easy matter to show that $\supset$R and $\forall$R are invertible rules within **C**-proofs. In particular, the sequent $\Sigma : D_1, \ldots, D_n \vdash D_0$ has a **C**-proof if and only if

$$\Sigma, \bar{x}_0, \bar{x}_1, \ldots \bar{x}_m : D_1, \ldots, D_n, A_1, \ldots, A_m \vdash A_0$$

has a **C**-proof. Since all the formulas on the left-hand side of this sequent are Horn clauses, the result follows directly from Proposition 5.6. We can also allow Horn clauses using description (5.2): we would simply need to prove the invertibility of additional introduction rules. The result that the classical entailment among Horn clauses implies their intuitionistic entailment can be generalized to geometric formulas, in which case that result is often referred to as the Barr Theorem [Negri, 2016].

Solution to Exercise 5.8 (page 64). Exercise 4.3(5) provides a **C**-proof of $((p \supset q) \supset p) \supset p$. It is easy to see that there is no **I**-proof (and, hence, no uniform proof) of this formula. Now assume that there is another formula, say, A which only contains implications and is strictly smaller while also having a **C**-proof but no **I**-proof. Thus B contains 2 or fewer occurrences of implications. Thus, B is of clausal order 2 or less and is of the form $(A_1 \supset (A_2 \supset A_3))$ or $((A_1 \supset A_2) \supset A_3)$ where A_1, A_2, A_3 are atomic formulas. Thus attempting a cut-free proof of B leads to attempting proofs of either $A_1, A_2 \vdash A_3$ or $A_1 \supset A_2 \vdash A_3$. In either case, we have a sequent involving only Horn clauses and, as a result of Proposition 5.6, if it is classically provable it is also intuitionistically provable. This is a contradiction.

Solution to Exercise 5.28 (page 81). Let Γ_1, Γ_2 be multisets of $\Downarrow\mathcal{L}_0$ formulas and let B and C be $\Downarrow\mathcal{L}_0$ formulas. Assume that $\Sigma : \Gamma_1 \vdash B$ and $\Sigma : B, \Gamma_2 \vdash C$ have cut-free **I**-proofs. By completeness of $\Downarrow\mathcal{L}_0$-proofs, these sequents also have $\Downarrow\mathcal{L}_0$-proofs. By the admissibility of weakening (Proposition 5.21), we have $\Sigma : B, \Gamma_1, \Gamma_2 \vdash C$ and $\Sigma : \Gamma_1, \Gamma_2 \vdash B$ have $\Downarrow\mathcal{L}_0$-proofs. By the admissibility of cut (Theorem 5.26), the sequent $\Sigma : \Gamma_1, \Gamma_2 \vdash C$ has an $\Downarrow\mathcal{L}_0$-proof. Finally, by the soundness of $\Downarrow\mathcal{L}_0$-proofs (Theorem 5.15), we have $\Sigma : \Gamma_1, \Gamma_2 \vdash C$ has a cut-free **I**-proof.

Solution to Exercise 5.42 (page 95). Assume that there is an *fohh* program Γ that satisfies the following specification: for every set $k \geq 1$ and $\{n_1, \ldots, n_k\}$, we have $\mathcal{A}, \Gamma \vdash_I maxa\ n$ if and only if n is the maximum of the set $\{n_1, \ldots, n_k\}$ and $\mathcal{A}$ is the set of atomic formulas $\{a\ n_1, \ldots, a\ n_k\}$. Let $\mathcal{A}$ be the set of atoms $\{a\ z, a\ (s\ z)\}$ and let $\mathcal{A}'$ be the set of atoms $\{a\ z, a\ (s\ z), a\ (s\ (s\ z))\}$. Thus, it must be the case that $\mathcal{A}, \Gamma \vdash_I maxa\ (s\ z)$. But by the monotonicity property of intuitionistic provability, $\mathcal{A}', \Gamma \vdash_I maxa\ (s\ z)$ but this is a contradiction to the choice of Γ, since $(s\ z)$ is not the maximum of the set of numbers encoded in $\mathcal{A}'$.

Solution to Exercise 5.43 (page 96). Assume that the logic program Γ defines the `notconnected` predicate. Using the graph described in Figure 5.5, it must be the case that `notconnected a e` is provable. But if we add `adj a e` to the logic program, the monotonicity property must force `notconnected a e` to be provable in that extended program. But this contradicts the assumption about `notconnected`.

Solution to Exercise 5.45 (page 97). Assume that there is a *fohh*-logic specifications P over the signature Σ_P. Also assume that this signature contains the constants $a : i$ and $f : i \to i \to i$. Also, assume that the constants $d : i$ and $e : i$ are not declared in Σ_P. By the specification of *subAll*, it is the case that

$$d : i, e : i, \Sigma_S \vdash_I subAll\ d\ a\ (f\ \ d\ \ e)\ (f\ \ a\ \ e).$$

By Proposition 5.44 and using the substitution of e for d, we know that

$$e : i, \Sigma_S \vdash_I subAll\ e\ a\ (f\ \ e\ \ e)\ (f\ \ a\ \ e).$$

But this contradicts the specification for *subAll*.

Solution to Exercise 6.2 (page 107). Assume that there is a cut-free proof of

$$\vdash p \otimes q,\ p^\perp \otimes q,\ p \otimes q^\perp,\ p^\perp \otimes q^\perp$$

Because of the symmetry of replacing p with $p^\perp$ and q with $q^\perp$, we can assume without loss of generality that this sequent is proved by the following occurrence of the $\otimes R$ rule.

$$\frac{\vdash p, \Delta \qquad \vdash q, \Delta'}{\vdash p \otimes q,\ p^\perp \otimes q,\ p \otimes q^\perp,\ p^\perp \otimes q^\perp}\ \otimes R$$

Here, Δ and Δ' are multisets whose union is the three element multiset $p^\perp \otimes q,\ p \otimes q^\perp,\ p^\perp \otimes q^\perp$. Note first that neither Δ nor Δ' can be empty. Note also that neither Δ nor Δ' can be a singleton: a simple case analysis show that if one of these multisets is a singleton then the corresponding premise is not provable. We have reached a contradiction when we note that every possible partition of 3 elements must contain either an empty or singleton partition.

Solution to Exercise 6.5 (page 108). It is an easy matter to show that for every prefix π ranging from the empty prefix, to $!$, $?$, $!?$, $?!$, $!?!$, and $?!?$ satisfies the equivalence $\pi\pi B \equiv \pi B$ for all formulas B. For example, the case for $\pi = ?\,!$ leads to proving the following two entailments.

$$\dfrac{\dfrac{\dfrac{}{?\,!\,B \vdash ?\,!\,B}\ \mathit{init}}{!\,?\,!\,B \vdash ?\,!\,B}\ !\,D}{?\,!\,?\,!\,B \vdash ?\,!\,B}\ ?\,L \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{!\,B \vdash !\,B}\ \mathit{init}}{!\,B \vdash ?\,!\,B}\ ?\,D}{!\,B \vdash !\,?\,!\,B}\ !\,R}{!\,B \vdash ?\,!\,?\,!\,B}\ ?\,D}{?\,!\,B \vdash ?\,!\,?\,!\,B}\ ?\,L$$

In the case that $\pi = !\,?\,!$, similar proofs can be give, although the following chain of equivalences

$$!\,?\,!\,!\,?\,!\,B \equiv !\,?\,!\,?\,!B \equiv !\,?\,!\,B.$$

is more convincing (rewriting subformulas by logically equivalent subformulas is justified using the cut-elimination result: see Section 4.3)

Here, we assume that the equivalences associated with $!$ and with $?\,!$ have already been proved. We can now prove that any prefix that has length 4 or more must be equivalence to one of shorter length. Let π be a prefix of length 4 or more, thus we can write it as $b_1b_2b_3b_4\pi'$ where the b_i's are either $!$ or $?$. These first four position must alternate between these two flavors of exponentials since otherwise they must contain either $!\,!$ or $?\,?$ (which can be shortened). Thus, π must be either $!\,?\,!\,?\,\pi'$ or $?\,!\,?\,!\,\pi'$. In the first case, we repeat $!\,?$ and in the second case we repeat $?\,!$. In either case, these repeated patterns can be shortened.

Solution to Exercise 6.17 (page 117). We use the six linear logic connectives $\{\top, \&, \bot, \multimap, \Rightarrow, \forall\}$ to define the remaining connectives.

$$B^\perp \equiv B \multimap \bot \quad 0 \equiv \top \multimap \bot \quad 1 \equiv \bot \multimap \bot \quad B \parr C \equiv (B \multimap \bot) \multimap C$$

$$B \oplus C \equiv ((B \multimap \bot) \mathbin{\&} (C \multimap \bot)) \multimap \bot \quad B \otimes C \equiv (B \multimap C \multimap \bot) \multimap \bot$$

$$\exists x.B \equiv (\forall x(B \multimap \bot)) \multimap \bot$$

$$!\,B \equiv (B \Rightarrow \bot) \multimap \bot \qquad ?\,B \equiv (B \multimap \bot) \Rightarrow \bot$$

Solution to Exercise 7.1 (page 153). Prove by induction on n that if Γ is a multiset of atoms and P is a tensor of atoms $A_1 \otimes \cdots \otimes A_n$ $(n \geq 0)$ then $\Gamma \vdash P$

is provable if and only if Γ is equal to the multiset $\{A_1, \ldots, A_n\}$. If $n = 0$ then this case is immediate since P is $\mathbf{1}$ and Γ is empty. Now, assume that $n > 0$ and that P is $(A_1 \otimes \cdots \otimes A_i) \otimes (A_{i+1} \otimes \cdots \otimes A_n)$. If $\Gamma \vdash P$ is provable then there is a multiset partition of Γ into Γ_1 and Γ_2 such that both sequents $\Gamma_1 \vdash A_1 \otimes \cdots \otimes A_i$ and $\Gamma_2 \vdash A_{i+1} \otimes \cdots \otimes A_n$ are provable. By induction, we have that Γ_1 is $\{A_1, \ldots, A_i\}$ and Γ_2 is $\{A_{i+1}, \ldots, A_n\}$ and, hence, Γ is $\{A_1, \ldots, A_n\}$. For the converse, assume that Γ_1 and Γ_2 are the multiset of atomic formula occurrences in P_1 and P_2, respectively. By induction, the sequents $\Gamma_1 \vdash P_1$ and $\Gamma_2 \vdash P_2$ are provable and, hence, so is $\Gamma \vdash P$.

Solution to Exercise 7.4 (page 158). Let the program $\mathcal{P}$ be the result of adding the declarations and clauses for `leq` from Figure 5.3 to the following declarations and clauses.

```
type maxa    nat -> o.

maxa M :- a M.
maxa M :- a N, a P, leq N P, (a P -o maxa M).
```

Solution to Exercise 7.5 (page 158). Let the program $\mathcal{P}$ be the result of adding the declarations and clauses for `sum` from Figure 5.3 to the following declarations and clauses.

```
type sumall    nat -> o.

sumall M :- a M.
sumall M :- a N, a P, sum N P S, (a S -o sumall M).
```

Bibliography

Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999. (Cited on page 239.)

Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002. (Cited on page 240.)

Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993. (Cited on page 7.)

Alexander Aiken. Set constraints: results, applications, and future directions. In *PPCP94: Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 171–179, 1994. (Cited on page 212.)

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992. doi:10.1093/logcom/2.3.297. (Cited on pages 98, 149, 212, 213, 214, 240, and 263.)

Peter B. Andrews. Provability in elementary type theory. *Zeitschrift fur Mathematische Logic und Grundlagen der Mathematik*, 20:411–418, 1974. (Cited on page 13.)

Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986. (Cited on pages 21 and 182.)

Andrew W. Appel and Amy P. Felty. Polymorphic lemmas and definitions in λProlog and Twelf. *Theory and Practice of Logic Programming*, 4(1-2): 1–39, 2004. doi:10.1017/S1471068403001698. (Cited on page 22.)

K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. of the ACM*, 29(3):841–862, 1982. (Cited on pages 11 and 98.)

Ali Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École Polytechnique, September 2015. (Cited on page 261.)

Matthias Baaz and Alexander Leitsch. Cut-elimination and Redundancy-elimination by Resolution. *Journal of Symbolic Computation*, 29(2):149–176, 2000. (Cited on page 200.)

David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1):2:1–2:44, April 2012. doi:10.1145/2071368.2071370. (Cited on page 151.)

David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 4790 of *LNCS*, pages 92–106, 2007. doi:10.1007/978-3-540-75560-9_9. (Cited on page 151.)

David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014. doi:10.6092/issn.1972-5787/4650. (Cited on pages 21, 258, and 277.)

Jean-Pierre Banâtre and Daniel Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, January 1993. (Cited on page 242.)

Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, New York, revised edition, 1984. (Cited on page 21.)

Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. (Cited on page 21.)

C. Benzmüller, C. E. Brown, and M. Kohlhase. Cut-simulation and impredicativity. *Logical Methods in Computer Science*, 5(1):1–21, 2009. doi:10.2168/LMCS-5(1:6)2009. (Cited on page 200.)

Christoph Benzmüller and Peter Andrews. Church's Type Theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2019 edition, 2019. (Cited on page 11.)

G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992. (Cited on page 242.)

Katalin Bimbó. *Proof Theory: Sequent Calculi and Related Formalisms*. CRC Press, 2015. (Cited on pages 36 and 58.)

Stefano Bistarelli, Iliano Cervesato, Gabriele Lenzini, and Fabio Martinelli. Relating multiset rewriting and process algebras for security protocol analysis. *Journal of Computer Security*, 13(1):3–47, 2005. (Cited on page 242.)

Roberto Blanco and Dale Miller. Proof outlines as proof certificates: a system description. In Iliano Cervesato and Carsten Schürmann, editors, *Proceedings First International Workshop on Focusing*, volume 197 of *Electronic Proceedings in Theoretical Computer Science*, pages 7–14. Open Publishing Association, November 2015. doi:10.4204/EPTCS.197.2. URL `http://www.eprover.org/EVENTS/IWIL-2015.html`. (Cited on page 263.)

Kenneth A. Bowen. Programming with full first-order logic. In Hayes, Michie, and Pao, editors, *Machine Intelligence 10*, pages 421–440. Ellis Horwood and John Wiley, 1982. (Cited on page 276.)

Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979. (Cited on page 261.)

Pascal Brisset and Olivier Ridoux. Naïve reverse can be linear. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, Paris, France, June 1991. MIT Press. (Cited on page 216.)

Paola Bruscoli and Alessio Guglielmi. On structuring proof search for first order linear logic. *Theoretical Computer Science*, 360(1-3):42–76, 2006. doi:10.1016/j.tcs.2005.11.047. (Cited on page 150.)

M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994. doi:10.1016/0743-1066(94)90032-9. (Cited on page 99.)

Iliano Cervesato and Mark-Oliver Stehr. Representing the MSR cryptoprotocol specification language in an extension of rewriting logic with dependent types. *Higher-Order Symbolic Computation*, 20:3–35, 2007. doi:10.1007/s10990-007-9003-3. (Cited on page 240.)

Iliano Cervesato, Joshua Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In Roy Dyckhoff, Heinrich Herre, and Peter Schroeder-Heister, editors, *7th Workshop on Extensions to Logic Programming*, LNAI, pages 28–30, Leipzig, Germany, March 1996. Springer. (Cited on page 151.)

Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A meta-notation for protocol analysis. In R. Gorrieri, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop — CSFW'99*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press. (Cited on pages 231, 240, and 242.)

Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Relating strands and multiset rewriting for security protocol analysis. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop — CSFW'00*, pages 35–51, Cambridge, UK, 3–5 July 2000a. IEEE Computer Society Press. (Cited on page 240.)

Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1-2):133–163, 2000b. (Cited on page 151.)

Kaustuv Chaudhuri. *The Focused Inverse Method for Linear Logic*. PhD thesis, Carnegie Mellon University, December 2006. Technical report CMU-CS-06-162. (Cited on page 150.)

Kaustuv Chaudhuri. Encoding additives using multiplicatives and subexponentials. *Math. Structures in Computer Science*, 28(5):651–666, 2018. doi:10.1017/S0960129516000293. URL `http://chaudhuri.info/papers/draft15mallmsel.pdf`. (Cited on page 151.)

Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In G. Ausiello, J. Karhumäki, G. Mauri, and L. Ong, editors, *Fifth International Conference on Theoretical Computer Science*, volume 273 of *IFIP*, pages 383–396. Springer, September 2008a. doi:10.1007/978-0-387-09680-3_26. (Cited on page 151.)

Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *J. of Automated Reasoning*, 40(2-3):133–177, 2008b. doi:10.1007/s10817-007-9091-0. (Cited on pages 150 and 263.)

Kaustuv Chaudhuri, Stefan Hetzl, and Dale Miller. A multi-focused proof system isomorphic to expansion proofs. *J. of Logic and Computation*, 26 (2):577–603, 2016. doi:10.1093/logcom/exu030. (Cited on page 151.)

Zakaria Chihani and Dale Miller. Proof certificates for equality reasoning. In Mario Benevides and René Thiemann, editors, *Post-proceedings of LSFA 2015: 10th Workshop on Logical and Semantic Frameworks, with Applications. Natal, Brazil.*, number 323 in ENTCS, pages 93–108. Elsevier, 2016. doi:10.1016/j.entcs.2016.06.007. (Cited on page 270.)

Zakaria Chihani, Dale Miller, and Fabien Renaud. Foundational proof certificates in first-order logic. In Maria Paola Bonacina, editor, *CADE 24: Conference on Automated Deduction 2013*, number 7898 in LNAI, pages 162–177, 2013. doi:10.1007/978-3-642-38574-2_11. (Cited on pages 270 and 271.)

Zakaria Chihani, Tomer Libal, and Giselle Reis. The proof certifier Checkers. In Hans De Nivelle, editor, *Proceedings of the 24th Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, number 9323 in LNCS, pages 201–210. Springer, 2015. doi:10.1007/978-3-319-24312-2_14. (Cited on page 271.)

Zakaria Chihani, Dale Miller, and Fabien Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 59(3):287–330, 2017. doi:10.1007/s10817-016-9380-6. (Cited on pages 259, 263, 270, and 271.)

Jawahar Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, February 1995. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/chirimar/phd.ps`. (Cited on pages 255 and 258.)

Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940. doi:10.2307/2266170. (Cited on pages 5, 6, 13, 17, 167, 200, 204, 243, and 269.)

D. Clement, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. In *K. Fuchi and M. Nivat, editors, proceedings of the France-Japan AI and CS Symposium, ICOT, Japan*, pages 49–89, 1986. URL `http://www.inria.fr/rrrt/rr-0416.html`. also Technical Memorandum PL-86-6 Information Processing Society of Japan and Rapport de recherche #0416, INRIA. (Cited on page 258.)

Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007. (Cited on page 261.)

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977. (Cited on page 218.)

P.-L. Curien. The $\lambda\rho$-calculus: An abstract framework for environment machines. Technical report, LIENS–CNRS, 1990. (Cited on page 251.)

V. Danos, J.-B. Joinet, and H. Schellinx. LKT and LKQ: sequent calculi for second order logic based upon dual linear decompositions of classical implication. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Note Series, pages 211–224. Cambridge University Press, 1995. doi:10.1017/CBO9780511629150. (Cited on page 263.)

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. The structure of exponentials: Uncovering the dynamics of linear logic proofs. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Kurt Gödel Colloquium*, volume 713 of *LNCS*, pages 159–171. Springer, 1993. (Cited on page 151.)

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. A new deconstructive logic: Linear logic. *Journal of Symbolic Logic*, 62(3):755–807, 1997. doi:10.2307/2275572. (Cited on page 149.)

Dedukti. The Dedukti system. `https://deducteam.github.io/`, 2013. (Cited on page 260.)

Olivier Delande and Dale Miller. A neutral approach to proof and refutation in MALL. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 498–508. IEEE Computer Society Press, 2008. doi:10.1016/j.apal.2009.07.017. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/lics08b.pdf`. (Cited on page 151.)

Giorgio Delzanno. An overview of MSR(C): A CLP-based framework for the symbolic verification of parameterized concurrent systems. *Electron. Notes Theor. Comput. Sci*, 76:65–82, 2002. doi:10.1016/S1571-0661(04)80786-2. (Cited on page 242.)

Joëlle Despeyroux. Proof of translation in natural semantics. In *1st Symp. on Logic in Computer Science*, pages 193–205, Cambridge, Mass, June 1986. IEEE. (Cited on page 258.)

Thierry Despeyroux. TYPOL: A formalism to implement natural semantics. Research Report 94, INRIA, Rocquencourt, France, March 1988. (Cited on page 258.)

Henry DeYoung and Frank Pfenning. Substructural proofs as automata. In *Asian Symposium on Programming Languages and Systems*, pages 3–22. Springer, 2016. (Cited on page 223.)

Roberto Di Cosmo and Dale Miller. Linear logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2019 edition, 2019. (Cited on page 11.)

Nancy A. Durgin, Patrick Lincoln, and John C. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *J. Comput. Secur*, 12(2): 247–311, 2004. doi:10.3233/JCS-2004-12203. (Cited on page 242.)

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. of Symbolic Logic*, 57(3):795–807, September 1992. doi:10.2307/2275431. (Cited on page 160.)

Roy Dyckhoff and Stephane Lengrand. Call-by-value λ-calculus and LJQ. *J. of Logic and Computation*, 17(6):1109–1134, 2007. doi:10.1093/logcom/exm037. (Cited on page 263.)

Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. of the ACM*, 23(4):733–742, 1976. (Cited on pages 11 and 98.)

Javier Esparza and Mogens Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994. (Cited on page 214.)

William M. Farmer. *Simple Type Theory: A Practical Logic for Expressing and Reasoning About Mathematical Ideas*. Springer Nature, 2023. (Cited on pages 21 and 182.)

Melvin Fitting. Tableaus for logic programming. *Journal of Automated Reasoning*, 13(2):175–188, 1994. (Cited on page 276.)

Melvin C. Fitting. *Intuitionistic Logic Model Theory and Forcing*. North-Holland, 1969. (Cited on page 44.)

Dov M. Gabbay. N-Prolog: An extension of Prolog with hypothetical implication II—logical foundations, and negation as failure. *Journal of Logic Programming*, 2(4):251–283, December 1985. (Cited on page 45.)

Dov M. Gabbay and Nicola Olivetti. *Goal-Directed Proof Theory*, volume 21 of *Applied Logic Series*. Kluwer Academic Publishers, August 2000. (Cited on page 98.)

Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49 (2):241–273, 2012. doi:10.1007/s10817-011-9218-1. URL `http://arxiv.org/abs/0911.2993`. (Cited on pages 258 and 277.)

Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986. (Cited on pages 11, 36, 58, and 98.)

Vijay Gehlot and Carl Gunter. Normal process representatives. In *5th Symp. on Logic in Computer Science*, pages 200–207, Philadelphia, Pennsylvania, June 1990. IEEE Computer Society Press. doi:10.1109/LICS.1990.113746. (Cited on page 242.)

Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935. doi:10.1007/BF01201353. Translation of articles that appeared in 1934-35. Collected papers appeared in 1969. (Cited on pages 6, 20, 36, 40, 44, 57, 58, 99, 102, 163, 262, 267, and 276.)

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987. doi:10.1016/0304-3975(87)90045-4. (Cited on pages 6, 118, 151, 167, 203, 204, and 263.)

Jean-Yves Girard. On the unity of logic. Technical Report 26, Université Paris VII, June 1991a. (Cited on page 149.)

Jean-Yves Girard. A new constructive logic: classical logic. *Math. Structures in Comp. Science*, 1:255–296, 1991b. doi:10.1017/S0960129500001328. (Cited on page 263.)

Jean-Yves Girard. A fixpoint theorem in linear logic. An email posting archived at `https://www.seas.upenn.edu/~sweirich/types/archive/1992/msg00030.html` to the linear@cs.stanford.edu mailing list, February 1992. (Cited on pages 99 and 283.)

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989. (Cited on pages 36, 58, and 149.)

Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *8th Asian Symposium on Computer Mathematics*, volume 5081 of *LNCS*, page 333. Springer, 2007. (Cited on page 259.)

Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979. doi:10.1007/3-540-09724-4. (Cited on page 261.)

Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–186. MIT Press, 2000. (Cited on page 21.)

Alessio Guglielmi. *Abstract Logic Programming in Linear Logic—Independence and Causality in a First Order Calculus*. PhD thesis, Università di Pisa, 1996. (Cited on page 150.)

Alessio Guglielmi. A system of interaction and structure. *ACM Trans. on Computational Logic*, 8(1):1–64, January 2007. doi:10.1145/1182613.1182614. (Cited on pages 58 and 111.)

Thomas C. Hales. A proof of the Kepler conjecture. *Annals of Mathematics*, 162(3):1065–1185, 2005. (Cited on page 259.)

Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. II. Programs as definitions. *J. of Logic and Computation*, 1(5):635–660, October 1991. doi:10.1093/logcom/1.5.635. (Cited on page 58.)

John Hannan. Extended natural semantics. *J. of Functional Programming*, 3 (2):123–152, April 1993. doi:10.1017/S0956796800000666. (Cited on page 258.)

John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992. doi:10.1017/S0960129500001559. (Cited on pages 250 and 252.)

John Hannan and Frank Pfenning. Compiler verification in LF. In *7th Symp. on Logic in Computer Science*, Santa Cruz, California, June 1992. IEEE Computer Society Press. (Cited on page 258.)

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060. (Cited on page 261.)

R. Harrop. Concerning formulas of the types $A \to B \vee C,\ A \to (Ex)B(x)$ in intuitionistic formal systems. *J. of Symbolic Logic*, 25:27–32, 1960. (Cited on page 99.)

Quentin Heath and Dale Miller. A proof theory for model checking. *J. of Automated Reasoning*, 63(4):857–885, 2019. doi:10.1007/s10817-018-9475-3. (Cited on pages 99 and 151.)

Hugo Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes*. PhD thesis, Université Paris 7, 1995. URL `https://tel.archives-ouvertes.fr/tel-00382528`. (Cited on page 263.)

Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Sci. Comput. Program.*, 58(1-2):115–140, 2005. (Cited on page 218.)

R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. (Cited on page 258.)

Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In G. Kahn, editor, *6th Symp. on Logic in Computer Science*, pages 32–42, Amsterdam, July 1991. IEEE. (Cited on page 151.)

Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. doi:10.1006/inco.1994.1036. (Cited on pages 151, 165, and 203.)

Joshua Hodas, Kevin Watkins, Naoyuki Tamura, and Kyoung-Sun Kang. Efficient implementation of a linear logic programming language. In Joxan Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 145–159, 1998. (Cited on page 151.)

Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, May 1994. (Cited on page 166.)

Joshua S. Hodas. A linear logic treatment of phrase structure grammars for unbounded dependencies. In Alain Lecomte, Françoise Lamarche, and Guy Perrier, editors, *Proceedings of the 2nd International Conference on Logical Aspects of Computational Linguistics (LACL-97)*, volume 1582 of *LNAI*, pages 160–179, Berlin, September 1999. Springer. (Cited on page 166.)

Joshua S. Hodas and Naoyuki Tamura. lolliCop — A linear logic implementation of a lean connection-method theorem prover for first-order classical logic. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of IJCAR: International Joint Conference on Automated Reasoning*, number 2083 in LNCS, pages 670–684. Springer, 2001. (Cited on page 151.)

Jacob M. Howe. *Proof Search Issues in Some Non-Classical Logics.* PhD thesis, University of St Andrews, December 1998. Available as University of St Andrews Research Report CS/99/1. (Cited on page 263.)

Jörg Hudelmaier. Bounds on cut-elimination in intuitionistic propositional logic. *Archive for Mathematical Logic*, 31:331–353, 1992. (Cited on page 160.)

Gérard P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975. doi:10.1016/0304-3975(75)90011-0. (Cited on page 9.)

Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer, March 1987. doi:10.1007/BFb0039592. (Cited on page 242.)

Max I. Kanovich. Petri nets, Horn programs, Linear Logic and vector games. *Annals of Pure and Applied Logic*, 75(1–2):107–135, 1995. doi:10.1017/S0960129500001328. (Cited on page 242.)

Stephen Cole Kleene. Permutability of inferences in Gentzen's calculi LK and LJ. *Memoirs of the American Mathematical Society*, 10:1–26, 1952. (Cited on page 37.)

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR)*, pages 207–220, Big Sky, MT, October 2009. ACM SIGOPS. (Cited on page 259.)

Naoki Kobayashi and Akinori Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 7(2):113–149, 1995. doi:10.1007/BF01211602. (Cited on page 218.)

R. A. Kowalski. Algorithm = Logic + Control. *Communications of the Association for Computing Machinery*, 22:424–436, 1979. (Cited on page 9.)

S. Kripke. A completeness theorem in modal logic'. *J. of Symbolic Logic*, 24 (1):1–14, 1959. (Cited on page 99.)

S. A. Kripke. Semantical analysis of intuitionistic logic I. In J. N. Crossley and M. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. (Proc. 8th Logic Colloq. Oxford 1963) North-Holland, Amsterdam, 1965. (Cited on pages 39 and 99.)

Jean-Louis Krivine. *Lambda-Calcul : Types et Modèles*. Etudes et Recherches en Informatique. Masson, 1990. (Cited on page 21.)

R. Kuzmin. Sur une nouvelle classe de nombres transcendants. *Bulletin de l'Académie des Sciences de l'URSS*, pages 585–597, 1930. (Cited on page 280.)

Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing a notion of modules in the logic programming language λProlog. In Evelina Lamma and Paola Mello, editors, *4th Workshop on Extensions to Logic Programming*, volume 660 of *LNAI*, pages 359–393. Springer, 1993. (Cited on page 99.)

P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6 (5):308–320, 1964. (Cited on pages 242 and 251.)

Olivier Laurent. *Etude de la polarisation en logique*. PhD thesis, Université Aix-Marseille II, March 2002. (Cited on page 263.)

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52 (7):107–115, 2009. doi:10.1145/1538788.1538814. (Cited on page 259.)

Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009. doi:10.1016/j.tcs.2009.07.041. Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi. (Cited on pages 263 and 267.)

Chuck Liang and Dale Miller. A focused approach to combining logics. *Annals of Pure and Applied Logic*, 162(9):679–697, 2011. doi:10.1016/j.apal.2011.01.012. (Cited on page 150.)

Chuck Liang and Dale Miller. On subexponentials, synthetic connectives, and multi-level delimited control. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, number 9450 in LNCS, November 2015. doi:10.1007/978-3-662-48899-7_21. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/subdelimlncs.pdf`. (Cited on page 151.)

Chuck Liang and Dale Miller. Focusing Gentzen's LK proof system. In Thomas Piecha and Kai Wehmeier, editors, *Peter Schroeder-Heister on Proof-Theoretic Semantics*, Outstanding Contributions to Logic. Springer, 2022. URL `https://hal.archives-ouvertes.fr/hal-03457379`. To appear. (Cited on page 150.)

P. Lincoln, J. Mitchell, A. Scedrov, and N. Shankar. Decision problems for propositional linear logic. *Annals of Pure and Applied Logic*, 56:239–311, 1992. (Cited on page 107.)

John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 2 edition, 1987. ISBN 3-540-18199-7. (Cited on pages 11 and 98.)

Pablo López and Ernesto Pimentel. The UMA Forum linear logic programming language. implementation, January 1998. (Cited on page 151.)

Sonia Marin, Dale Miller, and Marco Volpe. A focused framework for emulating modal proof systems. In Lev Beklemishev, Stéphane Demri, and András Máté, editors, *11th Conference on Advances in Modal Logic*, number 11 in Advances in Modal Logic, pages 469–488, Budapest, Hungary, August 2016. College Publications. URL `https://hal.archives-ouvertes.fr/hal-01379624`. (Cited on page 271.)

Sonia Marin, Dale Miller, Elaine Pimentel, and Marco Volpe. From axioms to synthetic inference rules via focusing. *Annals of Pure and Applied Logic*, 173(5):1–32, 2022. doi:10.1016/j.apal.2022.103091. (Cited on page 99.)

Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland. (Cited on page 7.)

John McCarthy. Artificial intelligence, logic and formalizing common sense. In Richmond Thomason, editor, *Philosophical Logic and Artificial Intelligence*. Kluwer Academic, 1989. URL `http://www-formal.stanford.edu/jmc/ailogic.dvi`. (Cited on page 92.)

Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000. doi:10.1016/S0304-3975(99)00171-1. (Cited on page 178.)

Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1): 80–136, 2002. doi:10.1145/504077.504080. (Cited on pages 58, 258, and 277.)

Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 294(3): 411–437, 2003. doi:10.1016/S0304-3975(01)00168-2. (Cited on page 99.)

Jia Meng. *The integration of higher order interactive proof with first order automatic theorem proving*. PhD thesis, University of Cambridge, Computer Laboratory, 2015. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-872.pdf`. (Cited on page 261.)

Dale Miller. A theory of modules for logic programming. In Robert M. Keller, editor, *Third Annual IEEE Symposium on Logic Programming*, pages 106–114, Salt Lake City, Utah, September 1986. (Cited on page 98.)

Dale Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989a. MIT Press. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/iclp89.pdf`. (Cited on page 181.)

Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):79–108, January 1989b. doi:10.1016/0743-1066(89)90031-9. (Cited on page 99.)

Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/AbsInLP.pdf.pdf`. (Cited on page 263.)

Dale Miller. Unification of simply typed lambda-terms as logic programming. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press. (Cited on page 99.)

Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First Russian Conference on Logic Programming, 14-18 September 1990*, number 592 in LNAI, pages 322–337. Springer, 1992. (Cited on page 99.)

Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *3rd Workshop on Extensions to Logic Programming*, number 660 in LNCS, pages 242–265, Bologna, Italy, 1993. Springer. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/pic.pdf`. (Cited on pages 214, 234, and 257.)

Dale Miller. A proposal for modules in λProlog. In R. Dyckhoff, editor, *4th Workshop on Extensions to Logic Programming*, number 798 in LNCS, pages 206–221. Springer, 1994. (Cited on page 99.)

Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996. doi:10.1016/0304-3975(96)00045-X. (Cited on pages 150, 165, 253, and 255.)

Dale Miller. Higher-order quantification and proof search. In Hélène Kirchner and Christophe Ringeissen, editors, *Proceedings of AMAST 2002*, number 2422 in LNCS, pages 60–74, 2002. (Cited on page 200.)

Dale Miller. Encryption as an abstract data-type: An extended abstract. In Iliano Cervesato, editor, *Proceedings of FCS'03: Foundations of Computer Security*, volume 84 of *ENTCS*, pages 18–29. Elsevier, 2003. doi:10.1016/S1571-0661(04)80841-7. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/fcs03.pdf`. (Cited on pages 218 and 240.)

Dale Miller. Collection analysis for Horn clause programs. In *Proceedings of PPDP 2006: 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 179–188, July 2006. doi:10.1145/1140335.1140357. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/ppdp06.pdf`. (Cited on page 218.)

Dale Miller. Formalizing operational semantic specifications in logic. *Concurrency Column of the Bulletin of the EATCS*, October 2008. (Cited on page 258.)

Dale Miller. Reasoning about computations using two-levels of logic. In K. Ueda, editor, *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS'10)*, number 6461 in LNCS, pages 34–46, 2010. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/aplas10.pdf`. (Cited on page 258.)

Dale Miller. A proposal for broad spectrum proof certificates. In J.-P. Jouannaud and Z. Shao, editors, *CPP: First International Conference on Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 54–69, 2011. doi:10.1007/978-3-642-25379-9_6. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/cpp11.pdf`. (Cited on page 270.)

Dale Miller. Communicating and trusting proofs: The case for broad spectrum proof certificates. In P. Schroeder-Heister, W. Hodges, G. Heinzmann, and P. E. Bour, editors, *Logic, Methodology, and Philosophy of Science. Proceedings of the Fourteenth International Congress*, pages 323–342. College Publications, 2014. (Cited on page 260.)

Dale Miller. Proof checking and logic programming. *Formal Aspects of Computing*, 29(3):383–399, 2017. doi:10.1007/s00165-016-0393-z. URL `http://dx.doi.org/10.1007/s00165-016-0393-z`. (Cited on page 259.)

Dale Miller. Reciprocal influences between logic programming and proof theory. *Philosophy & Technology*, 34(1):75–104, March 2021. doi:10.1007/s13347-019-00370-x. (Cited on page 11.)

Dale Miller. A survey of the proof-theoretic foundations of logic programming. *Theory and Practice of Logic Programming*, 22(6):859–904, October 2022. doi:10.1017/S1471068421000533. Published online November 2021. (Cited on page 11.)

Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, volume 225 of *LNCS*, pages 448–462, London, June 1986. Springer. doi:10.1007/3-540-16492-8_94. (Cited on page 98.)

Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012. doi:10.1017/CBO9781139021326. (Cited on pages 21, 22, 98, 99, and 269.)

Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. *ACM Computing Surveys*, 31, September 1999. doi:10.1145/333580.333590. (Cited on page 258.)

Dale Miller and Elaine Pimentel. Linear logic as a framework for specifying sequent calculus. In Jan van Eijck, Vincent van Oostrom, and Albert Visser, editors, *Logic Colloquium '99: Proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic*, Lecture Notes in Logic, pages 111–135. A K Peters Ltd, 2004. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/lc99.pdf`. (Cited on pages 58 and 165.)

Dale Miller and Elaine Pimentel. A formal framework for specifying sequent calculus proof systems. *Theoretical Computer Science*, 474:98–116, 2013. doi:10.1016/j.tcs.2012.12.008. URL `http://hal.inria.fr/hal-00787586`. (Cited on pages 37, 58, and 165.)

Dale Miller and Marco Volpe. Focused labeled proof systems for modal logic. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, number 9450 in LNCS, pages 266–280, November 2015. doi:10.1007/978-3-662-48899-7_19. (Cited on page 271.)

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1–2):125–157, 1991. doi:10.1016/0168-0072(91)90068-W. (Cited on pages 184 and 263.)

Robin Milner. A theory of type polymorphism in programming. *J. of Computer and System Sciences*, 17(3):348–375, 1978. (Cited on page 258.)

Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, New York, NY, 1980. (Cited on page 242.)

Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989. ISBN 978-0-13-115007-2. (Cited on page 257.)

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990. (Cited on page 258.)

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, 100(1):1–40, September 1992a. doi:10.1016/0890-5401(92)90008-4. (Cited on pages 234 and 258.)

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, 100(1):41–77, 1992b. doi:10.1016/0890-5401(92)90009-5. (Cited on pages 245 and 246.)

John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):99–124, 1991. (Cited on page 99.)

Joan Moschovakis. Intuitionistic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2021 edition, 2021. (Cited on page 11.)

Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming*. PhD thesis, University of Pennsylvania, May 1987. (Cited on page 186.)

Gopalan Nadathur and Dale Miller. An Overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/iclp88.pdf`. (Cited on page 218.)

Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990. doi:10.1145/96559.96570. (Cited on page 186.)

Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992. (Cited on pages 22 and 218.)

George C. Necula and Shree Prakash Rahul. Oracle-based checking of untrusted software. In Chris Hankin and Dave Schmidt, editors, *28th ACM Symp. on Principles of Programming Languages*, pages 142–154. ACM, 2001. (Cited on page 270.)

Sara Negri. Proof analysis beyond geometric theories: from rule systems to systems of rules. *Journal of Logic and Computation*, 26(2):513–537, 2016. doi:10.1093/LOGCOM/EXU037. (Cited on page 284.)

Sara Negri and Jan von Plato. *Structural Proof Theory*. Cambridge University Press, 2001. (Cited on pages 36, 58, and 99.)

Vivek Nigam and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In António Porto and Francisco Javier López-Fraguas, editors, *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 129–140. ACM, 2009. doi:10.1145/1599410.1599427. (Cited on page 151.)

Vivek Nigam, Elaine Pimentel, and Giselle Reis. An extended framework for specifying and reasoning about proof systems. *J. of Logic and Computation*, 2014. doi:10.1093/logcom/exu029. (Cited on page 165.)

Carlos Olarte, Vivek Nigam, and Elaine Pimentel. Subexponential concurrent constraint programming. *Theoretical Computer Science*, 606:98–120, November 2015. doi:10.1016/j.tcs.2015.06.031. (Cited on page 151.)

Leszek Pacholski and Andreas Podelski. Set constraints: A pearl in research on constraints. In *Principles and Practice of Constraint Programming - CP97*, number 1330 in LNCS, pages 549–562. Springer, 1997. (Cited on page 212.)

Remo Pareschi and Dale Miller. Extending definite clause grammars with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 373–389. MIT Press, June 1990. (Cited on page 166.)

Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in Science & Business Media. Springer, 1994. (Cited on page 21.)

Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *4th Symp. on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989. IEEE. (Cited on page 22.)

Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000. (Cited on page 37.)

Frank Pfenning. Church and Curry: Combining intrinsic and extrinsic typing. In Christoph Benzmüller, Chad E. Brown, Jörg Siekmann, and Richard Statman, editors, *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*, number 17 in Studies in Logic, pages 303–338. College Publications, 2008. (Cited on page 21.)

Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer. doi:10.1007/3-540-48660-7_14. (Cited on pages 22 and 258.)

Jan von Plato. The development of proof theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, winter 2018 edition, 2018. (Cited on page 11.)

Jan von Plato and G. Gentzen. Gentzen's proof of normalization for natural deduction. *Bulletin of Symbolic Logic*, 14(2):240–257, June 2008. URL `http://www.jstor.org/stable/20059973`. (Cited on page 58.)

Gordon D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981. (Cited on page 242.)

Gordon D. Plotkin. A structural approach to operational semantics. *J. of Logic and Algebraic Programming*, 60-61:17–139, 2004. (Cited on page 242.)

Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965. (Cited on page 44.)

A. N. Prior. The runabout inference-ticket. *Analysis*, 21(2):38–39, December 1960. (Cited on page 49.)

Michael Rathjen and Wilfried Sieg. Proof theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2020 edition, 2020. (Cited on page 11.)

David W. Reed and Donald W. Loveland. A comparison of three Prolog extensions. *Journal of Logic Programming*, 12(1 & 2):25–50, January 1992. (Cited on page 276.)

John H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305, June 1991. (Cited on page 256.)

J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12:23–41, January 1965. (Cited on page 8.)

Harold Schellinx. Some syntactical observations on linear logic. *J. of Logic and Computation*, 1(4):537–559, September 1991. (Cited on page 179.)

Wolfgang Schönfeld. PROLOG extensions based on tableau calculus. In *Proceedings of IJCAI 85*, pages 730–732, 1985. (Cited on page 276.)

Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *8th Symp. on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993. doi:10.1109/LICS.1993.287585. (Cited on pages 99 and 283.)

Robert J. Simmons. Structural focalization. *ACM Trans. on Computational Logic*, 15(3):21, 2014. doi:10.1145/2629678. (Cited on page 151.)

Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic*. Elsevier, 2006. (Cited on page 21.)

Richard Statman. Bounds for proof-search and speed-up in the predicate calculus. *Annals of Mathematical Logic*, 15:225–287, 1978. (Cited on page 37.)

Paul Tarau. Program transformations and WAM-support for the compilation of definite metaprograms. In *Proceedings of the First and Second Russian Conference on Logic Programming*, number 592 in LNAI, pages 462–473. Springer, 1992. (Cited on page 249.)

Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004. URL `http://etda.libraries.psu.edu/theses/approved/WorldWideIndex/ETD-479/`. (Cited on page 249.)

Alwen Tiu and Dale Miller. A proof search specification of the π-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, volume 138 of *ENTCS*, pages 79–101, 2005. doi:10.1016/j.entcs.2005.05.006. URL `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/fguc04workshop.pdf`. (Cited on page 58.)

Anne Sjerp Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer, 1973. (Cited on page 44.)

Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics*, volume 1. North-Holland, 1988. (Cited on page 39.)

Christian Urban. Forum and its implementations. Master's thesis, University of St. Andrews, December 1997. (Cited on page 151.)

Nathan Wetzler, Marijn J. H. Heule, and Jr. Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014. doi:10.1007/978-3-319-09284-3_31. (Cited on page 262.)

Index