Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language

Amy P. Felty

A dissertation
in
Computer and Information Science

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

1989

Dale Miller
Supervisor of Dissertation

Jean Gallier
Graduate Group Chairperson

Abstract

Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language

Amy P. Felty

Supervisor: Dale Miller

We argue that a logic programming language with a higher-order intuitionistic logic as its foundation can be used both to naturally specify and implement theorem provers. The language extends traditional logic programming languages by replacing first-order terms with simply-typed λ-terms, replacing first-order unification with higher-order unification, and allowing implication and universal quantification in queries and the bodies of clauses. Inference rules for a variety of proof systems can be naturally specified in this language. The higher-order features of the language contribute to a concise specification of provisos concerning variable occurrences in formulas and the discharge of assumptions present in many proof systems. In addition, abstraction in meta-terms allows the construction of terms representing object level proofs which capture the notions of abstractions found in many proof systems. The operational interpretations of the connectives of the language provide a set of basic search operations which describe goal-directed search for proofs.

To emphasize the generality of the meta-language, we compare it to another general specification language: the Logical Framework (LF). We describe a translation which compiles a specification of a logic in LF to a set of formulas of our meta-language, and prove this translation correct.

A direct specification of inference rules provides a declarative account of a proof system and a specification of the process of searching for proofs, but generally does not implement a theorem prover that can be executed directly. We show that it is sometimes possible to obtain a theorem prover that is complete under depth-first control by making only slight modifications to a specification. For the purpose of general theorem proving, we show how tactics and tacticals, which provide a framework for high-level control over search, can be directly implemented in our extended language. This framework serves as a starting point for implementing theorem provers and proof systems that can integrate many diversified operations on formulas and proofs for various logics.

We present an extensive set of examples that have been implemented in the higher-order logic programming language λProlog.

Table of Contents

1 Introduction **1**
- 1.1 Outline of the Dissertation . . . 3
  - 1.1.1 Specification . . . 4
  - 1.1.2 Implementation . . . 4

2 A Higher-Order Logic Programming Language **6**
- 2.1 The Simply Typed λ-Calculus . . . 7
- 2.2 Definite Clauses and Goal Formulas . . . 9
- 2.3 A Non-Deterministic Interpreter . . . 10
- 2.4 A Deterministic Interpreter . . . 12
- 2.5 Syntax for Logic Programs . . . 13

3 Specifying Sequent and Natural Deduction Style Theorem Provers for First-Order Intuitionistic Logic **17**
- 3.1 Specifying A Sequential Proof System . . . 17
- 3.2 A Specification of Natural Deduction in First-Order Logic . . . 24
- 3.3 Explicit vs. Implicit Representation of Assumptions in Natural Deduction . . . 27
- 3.4 Proof Terms for Natural Deduction . . . 28
- 3.5 A Theorem Prover That Constructs Normal N_I Proofs . . . 33

4 Specifying Other Logics **40**
- 4.1 Specification of Proof Systems for Classical Logic . . . 40
- 4.2 Specifying the Untyped and Simply-Typed λ-Calculus . . . 41
- 4.3 Correctness of Specifications . . . 45
  - 4.3.1 Mappings Between Object Terms and Meta-Terms . . . 46
  - 4.3.2 Correctness of the Specification of $\beta\eta$-Convertibility . . . 54
- 4.4 Specification of a Higher-Order Logic . . . 63
- 4.5 Discussion . . . 67

5 LF Signatures as Logic Programs **70**
- 5.1 Canonical LF . . . 70
- 5.2 A Specification of β-Convertibility for LF . . . 80
- 5.3 Translating LF Signatures to Logic Programs . . . 92
- 5.4 Translation of a Signature for Natural Deduction . . . 111

6 Executing Specifications Directly **116**
- 6.1 A Depth-First Strategy for Proof Checking in First-Order Logic . . . 117
- 6.2 Depth-First Theorem Proving in First-Order Logic . . . 119
- 6.3 Depth-First Search With a Higher-Order Object Logic . . . 127

7 Implementing Interpreters for Theorem Provers **130**
- 7.1 Defining and Interpreting Goal Structures . . . 130
  - 7.1.1 Goal Structures . . . 130

7.1.2 Inference Rules as a Relation on Goals . . . 132
7.1.3 Interpreting Compound Goal Structures . . . 133
7.2 N_I Inference Rules as Tactics . . . 134
7.3 Some Simple Interpreters . . . 139
7.3.1 A Depth-First Interpreter . . . 139
7.3.2 A Depth-First Iterative Deepening Interpreter . . . 140
7.4 A Tactic Interpreter . . . 142
7.5 Interactive Component for the Tactic Interpreter . . . 145
7.6 A Tactic Theorem Prover for Natural Deduction . . . 151
7.7 A Session With a Natural Deduction Tactic Prover . . . 153
7.8 Use of Meta-Language Features in Tactic Provers . . . 155
7.9 A Contrast to ML Tactic Theorem Provers . . . 156

8 Operations on Proof Terms **158**
8.1 Transforming L_I Proofs to N_I Proofs . . . 158
8.2 Proof Normalization in N_I . . . 165
8.3 Some Tactics for Proof by Analogy . . . 172

9 Conclusion and Future Work **179**
9.1 Future Work . . . 181
A λProlog Programs for Manipulating Lists . . . 184
B Infix Operators Used in λProlog Modules . . . 185

List of Figures

2.1 A Complete Set of Inference Rules for the Meta-Language 11

3.1 The L_I Sequent Proof System for First-Order Intuitionistic Logic 19
3.2 The N_I Natural Deduction Proof System for First-Order Intuitionistic Logic 25
3.3 Modified Rules for a Precise Formulation of the N_I Proof System 29
3.4 N_I Proof of $\forall x q(x) \supset \exists x q(x)$ 30
3.5 N_I Proofs of $p \supset (p \supset p)$ and $\forall x(q(x) \wedge p) \supset \forall x q(x)$ 31
3.6 Some Example Fragments of N_I Deductions 35

4.1 The L_C Sequent Proof System for First-Order Classical Logic 41
4.2 A Proof System for $\beta\eta$-Convertibility of λ-Terms 42
4.3 Type Assignment for the Simply-Typed λ-Calculus 44
4.4 Encoding of Untyped Terms 47
4.5 Decoding of Untyped Terms 52
4.6 Quantifier Rules for Higher-Order Logic 66

5.1 β-Convertibility in LF 72
5.2 The Edinburgh Logical Framework 73
5.3 Application Rules for C-LF 79
5.4 Encoding of LF Terms 81
5.5 Decoding of LF Terms 82
5.6 Negative Translation of LF Judgments to Definite Clauses 93
5.7 Positive Translation of LF Judgments to Goal Formulas 94
5.8 Translation of Arbitrary LF Assertions 111
5.9 LF Signature for a Fragment of N_I 111

7.1 The Import Structure of a Tactic Theorem Prover for N_I 153

8.1 L_I and N_I Proofs of $\forall x q(x) \supset \exists x q(x)$ 161
8.2 Reductions for Proof Normalization in N_I 166
8.3 Reductions for Removing Redundant Applications of $\vee$-E or $\exists$-E 170
8.4 A Tactic Theorem Prover for N_I With Proof By Analogy Tactics 176
8.5 N_I Proofs of $(\forall x q(x) \vee p) \supset \forall x(q(x) \vee p)$ and $\forall x(q(x) \wedge p) \supset (\forall x q(x) \wedge p)$. 177

B.1 Infix Operators Used in λProlog Modules 185

List of Modules

3.1 `fol`: Logical Connectives for First-Order Logic . . . 18
3.2 `nprf`: Proof Term Constructors for N_I . . . 33
3.3 `niprover`: Specification of N_I . . . 34
3.4 `ninormal`: Specification of N_I that Constructs Normal Deductions . . . 39
4.1 `convert`: $\beta\eta$-Convertibility in the Simply-Typed λ-Calculus . . . 46
5.1 `lfsig`: Signature for LF Terms, Types, and Kinds . . . 81
5.2 `lfconv`: β-Convertibility in LF . . . 85
5.3 `lfnorm`: β-Normalization in LF . . . 86
5.4 `lf_fol`: Translation of LF Signature for First-Order Logic . . . 112
5.5 `lf_ni`: A Slight Simplification of the Translation of an LF Signature for Natural Deduction . . . 114
6.1 `lprf`: Proof Term Constructors for L_I and L_C . . . 122
6.2 `lc_prove`: Main Search Component for Automatic Theorem Prover for L_C . 123
6.3 `lc_iter`: Iteration Loop for Automatic Theorem Prover for L_C . . . 124
6.4 `lc_auto`: Root Module for Automatic Theorem Prover for L_C . . . 125
7.1 `goals`: Goal Constructors for Meta-Goals . . . 131
7.2 `maptac`: Interpreting Compound Goal Structures . . . 133
7.3 `ndgoal`: Primitive Goal Constructors for N_I . . . 134
7.4 `ndtac` Part I: Tactics for N_I . . . 137
7.5 `ndtac` Part II: Tactics for N_I . . . 138
7.6 `dfs`: A Depth-First Interpreter . . . 140
7.7 `idfs`: An Iterative Deepening Interpreter . . . 141
7.8 `tacticals`: Some Common Tacticals . . . 143
7.9 `goalred`: Simplifying Compound Goals . . . 145
7.10 `inter_tacs`: Interactive Component for Tactic Interpreter . . . 150
7.11 `ndprint`: Output Program for Interactive Proof Search for N_I . . . 151
7.12 `nd`: Root Module for N_I Tactic Theorem Prover . . . 152
8.1 `liprover`: Specification of the L_I Inference Rules Without Cut . . . 162
8.2 `ninormal`: Explicit Context Specification of N_I that Constructs Normal Deductions . . . 163
8.3 `lniprover`: Proof Transformer from L_I Proofs to N_I Deductions . . . 164
8.4 `ndredex`: Reductions for Proof Normalization in N_I . . . 171
8.5 `ndnormalize`: Proof Normalization for N_I . . . 172
8.6 `mapcopy`: Processing Compound Goals in Proof By Analogy . . . 174
8.7 `copy`: Some Basic Tactics for Proof by Analogy . . . 174
8.8 `ndcopy`: Tactics for Proof By Analogy in N_I . . . 176
A.1 `lists`: Some Simple Operations on Lists . . . 184

Chapter 1

Introduction

Logic programming languages have many characteristics that indicate that they should serve both as good specification and implementation languages for theorem provers. First, in terms of specification, at the foundation of any logic programming language is a given logic in which programs are specified as a set of declarative propositions. The language Prolog [SS86], for instance, has the classical, first-order theory of Horn clauses as its foundation. Propositions in such languages are generally clauses with a top-level implication where a clause *body* implies its *head*. Proof systems that are defined by a set of inference figures should be easily specified as clauses of this form: the head of a clause specifies the conclusion of a rule, while the body specifies its premises. Second, in terms of implementation, a central mechanism of computation in logic programming is search. Search in logic programming languages is generally goal-directed, and is specified by a small set of operations. Search is also fundamental to theorem proving. The process of discovering a proof involves traversing an often very large and complex search space in some controlled manner. Finally, unification is an important mechanism in logic programming which is immediately and elegantly accessible in most implementations. This mechanism can be very useful in theorem proving in the manipulation of formulas and proofs, and in determining which inference rules can be applied and producing the proper instances of these rules.

The functional programming language ML was originally developed as the meta-language for the implementation of theorem provers and has been used extensively for this purpose [GMW79, Gor85, C$^+$86, Pau88]. The language contains many features that are useful for the design of theorem provers. It has a secure typing scheme, and is higher-order, allowing complex operations to be composed easily. In addition, it contains some unification capabilities and provisions for sophisticated manipulation of data objects. While ML has been used with much success in implementing theorem provers, many of the characteristics of logic programming languages suggest that such languages are worth investigating as an alternative in which certain basic operations such as search and unification are available more directly.

The basic data structure of traditional logic programming languages such as Prolog is first-order terms. As argued in [MN87], such terms are not adequate for representing quantified formulas in first-order logic, or in any other logic that contains quantifiers. First-order terms cannot adequately characterize the notions of variables and the scopes of variable bindings in such formulas. Of course, quantification can be specially encoded. For example, in Prolog, we can represent abstractions in formulas by representing bound variables as either Prolog free variables or constants. The formula $\forall x \exists y\ P(x, y)$, for instance, could be written as the first-order term `forall(X,exists(Y,p(X,Y)))` or `forall(x,exists(y,p(x,y)))` (where capital letters represent free variables and lower case letters represent constants). In either representation, occurrences of variables inside the scope of quantifiers must be distinguished from those outside it, and thus the substitution and unification that is available on free variables in Prolog is not available for these terms. In other words, Prolog's unification cannot provide unification at the object level. The programmer would have to write special procedures that accomplish these tasks for the encoded representation. By manipulating such an encoding, much of the declarative nature of logic programs is lost.

For the purposes of this dissertation, we use a higher-order logic programming language based on *higher-order hereditary Harrop formulas* [MNS87, MNPS]. This language replaces first-order terms with simply typed λ-terms. These terms can be used to elegantly express the *higher-order abstract syntax* of object logics [PE88]. For example, the abstractions built into λ-terms can be used to naturally represent quantification. We will see that the operations of quantifier instantiation and substitution are very naturally specified in terms of application of λ-terms. Abstraction in λ-terms also allows us to represent notions of abstraction found in many proof systems. For example, in natural deduction there is a notion of a variable bound inside a proof [Pra71]. In addition, in natural deduction, a proof of an implication $A \supset B$ can be considered a function from proofs of A to proofs of B. As we will see, terms representing proofs can be constructed in which these notions are captured. The construction of proof terms will in fact be an important aspect of all the theorem provers presented in this dissertation.

Our extended language also permits queries and the bodies of clauses to be both implications and universally quantified. We shall show how universal quantification can be used to naturally specify the provisos on inference rules in many proof systems concerning the occurrences of variables in formulas. Such uses of universal quantification are in fact essential for the correct implementation of various kinds of theorem provers for these logics. In addition, we will see that implication is very useful for specifying the discharge of assumptions in natural deduction systems.

The characteristics of higher-order hereditary Harrop formulas described so far suggest that this logic is well-suited to the representation of formulas and proofs and the declarative

specification of inference rules. We will also see that many of the operational aspects of the logic programming language with this logic as its foundation are well-suited to the implementation of theorem provers and the organization and implementation of proof systems in general. By a proof system[1], we mean here a system which implements not only theorem proving but also many other operations on formulas and proofs for potentially many logics.

For example, quantification over higher-order objects such as predicates provides a mechanism for writing procedures which take other procedures as arguments. In theorem proving, this capability will be useful in writing procedures to implement basic control mechanisms for proof search, for instance. Such procedures will take as parameters the various primitive operations of a particular theorem prover and compose them in various ways to form more complex operations and proof search strategies.

The operational interpretation of implicational goals provides support for modular programming. A modular organization is beneficial if not necessary for the implementation of potentially complex proof systems. Separating control mechanisms from the specification of primitive operations of a theorem prover is one example of modularization that may be useful. It is also desirable to separate domain specific information for one domain from that of others. Modules may contain, for example, search strategies geared toward a particular logic or domain, or store libraries of definitions and theorems for a given domain. By taking such a modular approach, not only will each individual module be conceptually simpler, but it will also be possible to integrate many potentially diverse operations on formulas and proofs into one unified framework.

1.1 Outline of the Dissertation

In Chapter 2, we present our extended logic programming language. We first present the language of higher-order hereditary Harrop (hohh) formulas, and then describe a non-deterministic interpreter for this language in terms of several simple search operations which correspond closely to the connectives of the logic. We also describe the deterministic interpreter λProlog, which adopts depth-first control, and for which prototype implementations exist [MN88, EP89]. The remainder of the dissertation is divided into two parts: specification and implementation.

[1] In contrast, note the other use of the term proof system to mean a set of inference figures for constructing proofs in a particular logic. The two uses of the term are quite different and it will always be clear from context which is meant.

1.1.1 Specification

To specify a theorem prover, we begin with a given logic and a proof system for that logic, and specify the inference rules as a set of hohh formulas. Such specifications provide a declarative account of the content of the proof systems. In addition, each specification will also have an operational reading with respect to the non-deterministic interpreter. In general, we will discuss both the declarative and operational readings of specifications.

In Chapter 3, we illustrate the specification of theorem provers using first-order intuitionistic logic as an example. We specify both sequent and natural deduction proof systems. We show that there are many ways to specify natural deduction, and include a specification that constructs only normal proofs. In Chapter 4, we specify several proof systems for other logics including classical first-order logic, the simply-typed λ-calculus, and a higher-order logic. One consequence of writing programs that are declarative and easy to read is that correctness proofs for such programs should be relatively easy to establish. As an example, we establish the correctness of a specification for $\beta\eta$-conversion in the untyped λ-calculus. We first prove the correctness of the representation of untyped terms as terms in the meta-language. This result illustrates in general the correspondence between first-order and higher-order abstract syntax for object logics. Once this result is established it is then easy to show the correctness of the specification of $\beta\eta$-conversion with respect to the non-deterministic interpreter described in Chapter 2.

The Edinburgh Logical Framework (LF) is a logic developed to provide a general theory of inference systems that captures many uniformities across different logics [HHP89]. In Chapter 5, we formalize the correspondence between specifying logics in LF and specifying logics as hohh formulas. We show that LF signatures can be naturally "compiled" into hohh formulas.

1.1.2 Implementation

We have used the term specification to mean a set of hohh formulas representing the inference rules of a given proof system, and have indicated that in addition to declarative content, specifications have an operational reading with respect to a non-deterministic interpreter. Yet specifications do not in general serve as complete implementations with respect to some deterministic control. In the second half of this dissertation, we are concerned with building theorem provers of practical import, and thus good execution behavior will become an important consideration. Since we use the same meta-language for both specification and implementation, we distinguish between the two by using the term implementation to mean a program that is intended to have the additional property that it has good behavior with respect to a deterministic interpreter, in particular, for the purposes of this dissertation, with respect to the λProlog interpreter described in

Chapter 2.

In Chapter 6, we analyze the operational behavior of the specifications of Chapters 3 and 4 with respect to the deterministic interpreter. We will see that many of the specifications, exactly as they are presented, are complete implementations of proof checkers. As theorem provers, not surprisingly, the programs often do not behave well with respect to the deterministic interpreter. In some cases, as we will discuss in Chapter 6, slight modifications to the specifications can provide complete implementations of automatic theorem provers.

In Chapter 7, we consider more generally the question of developing good implementations for theorem provers and proof systems. We argue that λProlog serves as a good meta-language for implementing interpreters for theorem provers. We implement several, each consisting of a small set of basic control mechanisms that behave well under depth-first control. We then show that by adding a set of clauses specifying the inference rules of a particular logic, we can obtain a theorem prover for that logic. These inference rule clauses serve as the set of basic operations to the interpreter. In this chapter, we concentrate mainly on the implementation of an interpreter based on tactics and tacticals. As an example, we present a complete theorem prover for natural deduction for first-order intuitionistic logic. We show that the tactic interpreter provides an environment that can be extended modularly to include other logics and operations such as interactive proof search, and later in Chapter 8, capabilities for proof by analogy.

As stated earlier, the construction of terms representing proofs in a particular proof system will be an important aspect of all the theorem provers we present. In Chapter 8, we present several operations involving such proof terms. Proof by analogy is one such operation where proof terms play a central role. We also present a program which translates proof terms representing proofs in a cut-free sequent system for first-order intuitionistic logic to proof terms for normal natural deduction proofs. This program combines two separate specifications for these two proof systems and provides an illustration of the correspondence between them. Also in this chapter is a program for proof normalization in natural deduction. The proof reductions of the proof normalization theorem in [Pra71] are specified quite naturally as clauses relating two proof terms.

Finally, we summarize and discuss future research directions in Chapter 9.

Chapter 2

A Higher-Order Logic Programming Language

The logic programming language used in this dissertation extends traditional logic programming languages by enriching the underlying logical foundation. Higher-order hereditary Harrop formulas extend Horn clauses in essentially two ways. The first extension permits richer logical expressions in both queries (goals) and the bodies of program clauses. In particular, this extension provides for implications, disjunctions, and universally and existentially quantified formulas, as well as conjunction. The addition of disjunctions and existential quantifiers into the bodies of clauses does not depart much from the usual presentation of Horn clauses since such extended clauses are classically equivalent to Horn clauses. The addition of implications and universal quantifiers, however, makes a significant departure. The second extension to Horn clauses makes this language *higher-order* in the sense that it is possible to quantify over predicate and function symbols. For a complete realization of this kind of extension, several other features must be added. In order to instantiate predicate and function variables with terms, first-order terms are replaced by more expressive simply typed λ-terms. The application of λ-terms is handled by λ-conversion, while the unification of λ-terms is handled by higher-order unification.

There are four major components to our extended logic programming language: types, λ-terms, definite clauses, and goal formulas. Types and terms are essentially those of the simply typed λ-calculus [HS86]. In Section 2.1, we give the basic definitions for this calculus. Then, in Section 2.2, we present the formulas and clauses of the logic programming language. In Section 2.3, we discuss a simple non-deterministic interpreter for the language, and then in Section 2.4, a deterministic version is presented. This interpreter is a description of the logic programming language λProlog. Finally, in Section 2.5, we present a syntax for terms and formulas of the logic that will be used in the remainder of this dissertation.

2.1 The Simply Typed λ-Calculus

We assume that a certain set of base types is provided. This set must contain the type symbol o which will denote the type of logic programming propositions. Function types are built in the usual way using the arrow constructor $\rightarrow$: if τ_1 and τ_2 are types then so is $\tau_1 \rightarrow \tau_2$. The arrow type constructor associates to the right: $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ is read as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

For each type τ we assume that there are denumerably many constants and variables of that type. λ-terms can then be built up using constants, variables, applications, and abstractions in the usual way, as specified by the following inductive definition where M and N are syntactic variables for terms, $x : \tau$ is a variable of type τ, and $c : \tau$ is a constant of type τ.

$$M := c : \tau \mid x : \tau \mid MN \mid \lambda x : \tau.M$$

When writing a variable or constant, we often omit the type when it can be inferred from context. We assume the usual definitions for bound and free variables, and closed and open terms.

Terms are assigned types as follows: a constant or variable of type τ is a term of type τ, an abstraction $\lambda x : \tau.M$ is a term of type $\tau \rightarrow \tau'$ if M is a term of type τ', and an application MN is a term of type τ' if M is a term of type $\tau \rightarrow \tau'$ and N is a term of type τ. If a term can be assigned a type in this manner it is said to be *well-typed*.

In this dissertation, we will denote a substitution, σ, as a set of pairs. The set $\{\langle x_1, M_1\rangle, \ldots, \langle x_n, M_n\rangle\}$ denotes the function that for $i = 1, \ldots, n$ maps x_i to M_i. Given a term N, $\sigma(N)$ denotes the term obtained by simultaneously replacing free occurrences of $x_1, \ldots, x_n$ in N with $M_1, \ldots, M_n$, respectively, systematically renaming bound variables when necessary in order to avoid variable capture. We write $\mathrm{dom}(\sigma)$ to denote the domain of substitution σ and $\mathrm{cod}(\sigma)$ to denote the codomain of σ.[1] We define an updating operation on substitutions. Given substitution σ, $\langle x, M\rangle + \sigma$ denotes the substitution obtained by adding the pair $\langle x, M\rangle$ to σ, such that if x already appears on the left of a pair in σ, this pair is overwritten. Following convention, we sometimes write $[N_1/x_1, \ldots, N_n/x_n]M$ to denote the term $\sigma(M)$ where σ is the substitution $\{\langle x_1, N_1\rangle, \ldots, \langle x_n, N_n\rangle\}$.

Two terms M and N are said to be *$\beta\eta$-convertible*, written $M =_{\beta\eta} N$ if they are equivalent modulo the following three equations.

(α)	$\lambda x.M = \lambda y.[y/x]M$	if y is not free in M
(β)	$(\lambda x.M)N = [N/x]M$	
(η)	$\lambda x.Mx = M$	if x is not free in M

Equality between λ-terms in our meta-language is taken to mean $\beta\eta$-convertible.

[1] Throughout this dissertation, dom and cod will be used to denote the domain and codomain of functions in general.

A *β-redex* is a term of the form $(\lambda x.M)N$ and an *η-redex* is a term of the form $\lambda x.Mx$ where x does not occur free in M. A term is in *$\beta\eta$-normal form* if it contains no β or η-redexes. A term in $\beta\eta$-normal form has the form $\lambda x_1 \ldots \lambda x_n.(xM_1 \ldots M_m)$ where $n, m \geq 0$, x is either a constant or variable and $M_1, \ldots, M_m$ are in $\beta\eta$-normal form. We say that a term is in *$\beta\eta$-long normal form* (or just $\beta\eta$-long form) if it has the form $\lambda x_1 \ldots \lambda x_n.(xM_1 \ldots M_m)$ where x is a variable or constant of type $\tau_1 \to \cdots \to \tau_m \to \tau$ where τ is a base type, and $M_1, \ldots, M_m$ are in $\beta\eta$-long form. We will also say that a term is *$\beta\eta$-normal* or *$\beta\eta$-long* if it is in $\beta\eta$-normal form or $\beta\eta$-long form, respectively. All well-typed terms are $\beta\eta$-convertible to a term in $\beta\eta$-normal form and a term in $\beta\eta$-long form that are unique up to the (α) equation above. Given two terms M and N, if $M =_{\beta\eta} N$ and N is $\beta\eta$-normal or $\beta\eta$-long, we say that N is the $\beta\eta$-normal form or $\beta\eta$-long form, respectively, of N.

We assume that the reader is familiar with the basic properties of the simply typed λ-calculus, in particular, those found in Chapters 1,7,13, and 15 of [HS86]. The following result, which will be needed in later chapters, follows from basic properties of substitution and α-conversion. We write $M =_\alpha N$ to denote the equivalence of two terms modulo the (α) equation above.

Lemma 2.1 Let σ be a substitution and let M and N be terms.

1. If $\sigma(\lambda x.M) =_\alpha \lambda y.N$ then $(\langle x, y\rangle + \sigma)(M) =_\alpha N$.
2. If $(\langle x, y\rangle + \sigma)(M) =_\alpha N$, and y is not free in $\sigma(\lambda x.M)$ whenever y is different from x, then $\sigma(\lambda x.M) =_\alpha \lambda y.N$.

Proof: The proof of (1) is by induction on the structure of the term M.

Base: If M is a constant c, then N is also the constant c. Clearly $(\langle x, y\rangle + \sigma)(c) =_\alpha c$. If M is the variable x, then N is the variable y, and clearly $(\langle x, y\rangle + \sigma)(x) =_\alpha y$. If M is a variable z where z is different from x, then $\sigma(\lambda x.z) =_\alpha \lambda x'.(\langle x, x'\rangle + \sigma)(z)$ for some x' not free in the terms in $\mathrm{cod}(\sigma)$ or z. Thus N is $(\langle x, x'\rangle + \sigma)(z)$ where y is not free in $(\langle x, x'\rangle + \sigma)(z)$. Since x is not free in z, $(\langle x, y\rangle + \sigma)(z) =_\alpha (\langle x, x'\rangle + \sigma)(z)$.

Induction Step: If M has the form P_1P_2, then N has the form Q_1Q_2 and $\sigma(\lambda x.P_1P_2) =_\alpha \lambda y.Q_1Q_2$. Thus $\sigma(\lambda x.P_1) =_\alpha \lambda y.Q_1$ and $\sigma(\lambda x.P_2) =_\alpha \lambda y.Q_2$. By the induction hypothesis, $(\langle x, y\rangle + \sigma)(P_1) =_\alpha Q_1$ and $(\langle x, y\rangle + \sigma)(P_2) =_\alpha Q_2$. Clearly,

$$((\langle x, y\rangle + \sigma)(P_1))((\langle x, y\rangle + \sigma)(P_2)) =_\alpha (\langle x, y\rangle + \sigma)(P_1P_2).$$

Thus $(\langle x, y\rangle + \sigma)(P_1P_2) =_\alpha Q_1Q_2$.

If M has the form $\lambda w.P$ then N has the form $\lambda z.Q$ and $\sigma(\lambda x.\lambda w.P) =_\alpha \lambda y.\lambda z.Q$. For some x' not free in the terms in $\mathrm{cod}(\sigma)$ or in $\lambda w.P$,

$$\sigma(\lambda x.\lambda w.P) =_\alpha \lambda x'.(\langle x, x'\rangle + \sigma)(\lambda w.P).$$

Since y is not free in $\lambda y.\lambda z.Q$, y is not free in $\sigma(\lambda x.\lambda w.P)$. Thus $\lambda x'.(\langle x, x'\rangle + \sigma)(\lambda w.P) =_\alpha \lambda y.(\langle x, y\rangle + \sigma)(\lambda w.P)$. Hence $(\langle x, y\rangle + \sigma)(\lambda w.P) =_\alpha \lambda z.Q$.

The proof of (2) is also by induction on the structure of M.

Base: If M is a constant c, then N is also the constant c. Clearly $\sigma(\lambda x.c) =_\alpha \lambda y.c$. If M is the variable x, then N is the variable y, and clearly $\sigma(\lambda x.x) =_\alpha \lambda y.y$. Next consider the case when M is the variable y different from x and y is not free in $\sigma(\lambda x.y)$. Since y is not free in $\sigma(\lambda x.y)$, there must be a pair $\langle y, z\rangle \in \sigma$ such that z is different from y and $(\langle x, y\rangle + \sigma)(y) =_\alpha z$. Thus N is the variable z, and hence $\sigma(\lambda x.y) =_\alpha \lambda y.z$. Finally, if M is a variable z where z is different from both x and y, then $(\langle x, y\rangle + \sigma)(z) =_\alpha \sigma(z)$, and N is $\sigma(z)$. Thus $\sigma(\lambda x.z) =_\alpha \lambda y.\sigma(z)$.

Induction Step: If M has the form P_1P_2, then N has the form Q_1Q_2 and $(\langle x, y\rangle + \sigma)(P_1P_2) =_\alpha Q_1Q_2$. Since,

$$(\langle x, y\rangle + \sigma)(P_1P_2) =_\alpha ((\langle x, y\rangle + \sigma)(P_1))((\langle x, y\rangle + \sigma)(P_2)),$$

it follows that $(\langle x, y\rangle + \sigma)(P_1) =_\alpha Q_1$ and $(\langle x, y\rangle + \sigma)(P_2) =_\alpha Q_2$. Thus by the induction hypothesis, $\sigma(\lambda x.P_1) =_\alpha \lambda y.Q_1$ and $\sigma(\lambda x.P_2) =_\alpha \lambda y.Q_2$. Thus $\sigma(\lambda x.P_1P_2) =_\alpha \lambda y.Q_1Q_2$.

If M has the form $\lambda w.P$ then N has the form $\lambda z.Q$ and $(\langle x, y\rangle + \sigma)(\lambda w.P) =_\alpha \lambda z.Q$. We must show $\sigma(\lambda x.\lambda w.P) =_\alpha \lambda y.\lambda z.Q$. For the case when x is y, we know that $\lambda x.(\langle x, x\rangle + \sigma)(\lambda w.P) =_\alpha \lambda x.\lambda z.Q$. Clearly $\lambda x.(\langle x, x\rangle + \sigma)(\lambda w.P) =_\alpha \sigma(\lambda x.\lambda w.P)$. Thus $\sigma(\lambda x.\lambda w.P) =_\alpha \lambda x.\lambda z.Q$. It remains to be shown that the equivalence holds when x is different from y. For some x' not free in the terms in $\mathrm{cod}(\sigma)$ or in $\lambda w.P$, $\sigma(\lambda x.\lambda w.P) =_\alpha \lambda x'.(\langle x, x'\rangle + \sigma)(\lambda w.P)$. Since x is different from y, we know that y is not free in $\sigma(\lambda x.\lambda w.P)$. Thus, either x' is y or y is not free in $(\langle x, x'\rangle + \sigma)(\lambda w.P)$. In either case,

$$\lambda x'.(\langle x, x'\rangle + \sigma)(\lambda w.P) =_\alpha \lambda y.(\langle x, y\rangle + \sigma)(\lambda w.P).$$

Since $(\langle x, y\rangle + \sigma)(\lambda w.P) =_\alpha \lambda z.Q$, it follows that $\lambda y.(\langle x, y\rangle + \sigma)(\lambda w.P) =_\alpha \lambda y.\lambda z.Q$. Thus $\sigma(\lambda x.\lambda w.P) =_\alpha \lambda y.\lambda z.Q$. ■

2.2 Definite Clauses and Goal Formulas

Logical connectives and quantifiers are introduced into λ-terms by introducing suitable constants as in [Chu40]. In particular, the constants $\wedge, \vee, \supset$ are all assumed to have type $o \to o \to o$, and the constants $\forall$ and $\exists$ are given type $(\alpha \to o) \to o$ for each type replacing the "type variable" α. (Negation is not used in this programming language.) The expressions $\forall\lambda x.A$ and $\exists\lambda x.A$ are abbreviated to be $\forall xA$ and $\exists xA$, respectively. $\wedge, \vee$, and $\supset$ will be written as infix constants. A λ-term which is of type o is called a *proposition*. A function symbol whose target type is o will be considered a *predicate*.

Given a set of base types $\mathcal{B}$, a *signature (over $\mathcal{B}$)* is a finite set Σ of constants and variables such that there is at least one constant or variable of every base type. A *λ-term over a signature* Σ is a λ-term built using the terms in Σ and the logical constants. Given a signature Σ, we define $H(\Sigma)$ to be the set of λ-terms over Σ that do not contain the logical constant $\supset$. A proposition in $H(\Sigma)$ in $\beta\eta$-normal form whose head is not a logical constant will be called an *atomic formula*.

We now define two new classes of propositions over a signature Σ, called *goal formulas* and *definite clauses* (or just *clauses*). Let A be a syntactic variable for atomic formulas, G a syntactic variable for goal formulas, and D a syntactic variable for definite clauses. These two classes of formulas are defined by the following mutual recursion.

$$G := A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid D \supset G \mid \forall x G$$

$$D := A \mid G \supset A \mid \forall x D$$

Note that the top-level form of a definite clause is either $\forall x_1 \ldots \forall x_n A$ or $\forall x_1 \ldots \forall x_n (G \supset A)$ where $n \geq 0$. In either case, the atomic formula A is called the *head* of the clause, and G is called the *body*. A universal instance of the body will be called a *subgoal*. There is one final restriction on definite clauses: the head of a definite clause must have a constant as its head. The heads of atomic goal formulas on the other hand may be either variable or constant. The set of definite clauses are also called *higher-order hereditary Harrop formulas* (see [MNPS]), which we abbreviate to hohh. Goal formulas will also be called *queries*.

A *logic program* or just simply a *program* is a pair $\Sigma; \mathcal{P}$ where Σ is a signature and $\mathcal{P}$ is a finite set of clauses over Σ. We will often refer to a set $\mathcal{P}$ of definite clauses alone as a program. A signature can usually be inferred from any type information given and the occurrences of constants in the definite clauses.

2.3 A Non-Deterministic Interpreter

Provability for the logic described in the previous section is given in terms of a sequent calculus [Gen69]. A *sequent* in this system is a triple $\Sigma; \mathcal{P} \longrightarrow G$, where Σ is a signature over a set of base types $\mathcal{B}$, $\mathcal{P}$ is a set of definite clauses over Σ, and G is a goal formula over Σ. The inference rules for this system are given in Figure 2.1. The set $|\mathcal{P}|_\Sigma$ is defined to be the smallest set of clauses over Σ such that $\mathcal{P} \subseteq |\mathcal{P}|_\Sigma$ and if $\forall x D \in |\mathcal{P}|_\Sigma$ and $M \in H(\Sigma)$, then $[M/x]D \in |\mathcal{P}|_\Sigma$. Trees are constructed using these inference rules as schemas in the usual way up to $\beta\eta$-convertibility of the goal formula, *i.e.*, for every node $\Sigma; \mathcal{P} \longrightarrow G$ that is an instance of a premise of one rule, if it is not a leaf node, then there is some G' such that $G =_{\beta\eta} G'$, and $\Sigma; \mathcal{P} \longrightarrow G'$ is an instance of the conclusion of the preceeding rule. Building trees in this way avoids the need for an explicit rule for $\beta\eta$-convertibility. A proof of the sequent $\Sigma; \mathcal{P} \longrightarrow G$ is a finite tree such that the root is labeled with $\Sigma; \mathcal{P} \longrightarrow G$

$$\frac{\Sigma;\mathcal{P} \longrightarrow G_1 \qquad \Sigma;\mathcal{P} \longrightarrow G_2}{\Sigma;\mathcal{P} \longrightarrow G_1 \wedge G_2}\ \wedge\text{-R}$$

$$\frac{\Sigma;\mathcal{P} \longrightarrow G_1}{\Sigma;\mathcal{P} \longrightarrow G_1 \vee G_2}\ \vee\text{-R} \qquad \frac{\Sigma;\mathcal{P} \longrightarrow G_2}{\Sigma;\mathcal{P} \longrightarrow G_1 \vee G_2}\ \vee\text{-R}$$

$$\frac{\Sigma;\mathcal{P} \longrightarrow [M/x]G}{\Sigma;\mathcal{P} \longrightarrow \exists x:\tau.G}\ \exists\text{-R} \qquad \frac{\Sigma;\mathcal{P} \cup \{D\} \longrightarrow G}{\Sigma;\mathcal{P} \longrightarrow D \supset G}\ \supset\text{-R}$$

$$\frac{\Sigma \cup \{y:\tau\};\mathcal{P} \longrightarrow [y/x]G}{\Sigma;\mathcal{P} \longrightarrow \forall x:\tau.G}\ \forall\text{-R} \qquad \frac{\Sigma;\mathcal{P} \longrightarrow G}{\Sigma;\mathcal{P} \longrightarrow A}\ \text{backchain}$$

The $\exists$-R rule has the proviso that M is a term of type τ in $H(\Sigma)$.
The $\forall$-R rule has the proviso that the variable y is not in Σ.
The backchain rule has the proviso that $G \supset A$ is in $|\mathcal{P}|_\Sigma$.

Figure 2.1: A Complete Set of Inference Rules for the Meta-Language

and the leaves are labeled with *initial sequents*, that is, sequents $\Sigma';\mathcal{P}' \longrightarrow A'$ such that A' is atomic and $A' \in |\mathcal{P}'|_{\Sigma'}$.

One important property of this proof system [MNPS] is that given a signature Σ, a set of definite clauses $\mathcal{P}$ over Σ, and a goal formula G over Σ, the sequent $\Sigma;\mathcal{P} \longrightarrow G$ is provable in the inference system in Figure 2.1 iff G is intuitionistically provable from $\mathcal{P}$. This logic allows for quantification over arbitrary predicates, a feature which we will not make use of until Chapter 7. Quantification over function symbols on the other hand will be used extensively in our examples although it will generally be restricted to order two. Note that in any node in a proof, if the goal formula on the right of the sequent has a top-level connective, the sequent must be the conclusion of the inference rule that introduces that connective. If the goal formula is atomic, the sequent is either an initial sequent or the conclusion of the backchain rule. Based on this property (called *uniformity* [MNPS]), a simple *non-deterministic* logic programming interpreter can be described. We represent a state of such an interpreter as a triple $\langle \Sigma, \mathcal{P}, G\rangle$ where Σ is the current signature, $\mathcal{P}$ is the current program consisting of clauses over Σ, and G is the current goal, a goal formula over Σ. $\Sigma;\mathcal{P} \vdash_I G$ will denote the proposition that the interpreter succeeds given the current signature Σ, program $\mathcal{P}$, and goal G. Judgments of this form will be called *hohh judgments*. A high level description of an interpreter is given by the following six *search operations*.

AND	$\Sigma; \mathcal{P} \vdash_I G_1 \wedge G_2$ only if $\Sigma; \mathcal{P} \vdash_I G_1$ and $\Sigma; \mathcal{P} \vdash_I G_2$.
OR	$\Sigma; \mathcal{P} \vdash_I G_1 \vee G_2$ only if $\Sigma; \mathcal{P} \vdash_I G_1$ or $\Sigma; \mathcal{P} \vdash_I G_2$.
INSTANCE	$\Sigma; \mathcal{P} \vdash_I \exists x : \tau.G$ only if there is some term M in $H(\Sigma)$ of type τ such that $\Sigma; \mathcal{P} \vdash_I [M/x]G$.
AUGMENT	$\Sigma; \mathcal{P} \vdash_I D \supset G$ only if $\Sigma; \mathcal{P} \cup \{D\} \vdash_I G$.
GENERIC	$\Sigma; \mathcal{P} \vdash_I \forall x : \tau.G$ only if $\Sigma \cup \{y : \tau\}; \mathcal{P} \vdash_I [y/x]G$ where y is a variable or constant such that $y \notin \Sigma$.
BACKCHAIN	$\Sigma; \mathcal{P} \vdash_I A$ (where A is atomic) if either $A \in \lvert\mathcal{P}\rvert_\Sigma$ or $G \supset A \in \lvert\mathcal{P}\rvert_\Sigma$ and $\Sigma; \mathcal{P} \vdash_I G$.

Note that the AUGMENT search operation extends the current program, while the GENERIC search operation extends the current signature. We allow this new signature item to be either a variable or constant. As we will see in Section 4.3 and Chapter 5, in establishing various results about logic programs it will be convenient to think of these new signature items as variables. On the other hand, in presenting example clauses and programs and describing their operational behavior, we will generally think of them as constants to avoid confusion with "logic variables" used as an implementation technique for handling substitutions and unification.

Also, note that the set of terms from which substitution terms are chosen in the INSTANCE operation is the same set that was used in defining $|\mathcal{P}|_\Sigma$ used in the BACKCHAIN operation.

2.4 A Deterministic Interpreter

In order to implement a deterministic interpreter, it is important to make choices which are left unspecified by the high-level description of the non-deterministic interpreter. The choices made here are those that were made in implementing the λProlog systems LP2.7 [MN88] and eLP [EP89], many of which are similar to those routinely used in Prolog.

The order in which conjuncts and disjuncts are attempted and the order for backchaining over definite clauses is determined exactly as in conventional Prolog: conjuncts and disjuncts are attempted in the order they are presented. Definite clauses are backchained over in the order they are listed in $\mathcal{P}$ using a depth-first search paradigm to handle failures. In the extended language, clauses can be added dynamically by the AUGMENT operation. We specify that new clauses get added to the top of the list.

The non-determinism in the INSTANCE operation is extreme. Generally when an existential goal is attempted, there is very little information available as to what closed λ-term should be inserted. Instead, the Prolog implementation technique of instantiating the existential quantifier with a logic (free) variable which is later "filled in" using unification is

employed. Thus instead of picking a term M from $H(\Sigma)$, the INSTANCE search operation will introduce a new logic variable as the substitution term. A similar use of logic variables is made in implementing BACKCHAIN: instead of choosing a clause from $|\mathcal{P}|_\Sigma$, a clause from $\mathcal{P}$ is chosen and an instance is made by replacing all outermost universally quantified variables with new logic variables. Such logic variables are not part of the meta-language and thus are distinct from the variables that occur in Σ.

The addition of logic variables in our setting, however, forces the following extensions to conventional Prolog implementations. First, higher-order unification becomes necessary since these logic variables can occur inside λ-terms. Also the equality of terms is not a simple syntactic check but a more complex check of $\beta\eta$-conversion. This equality check is decidable, although if the terms being compared are large, this check can be very expensive: β-reduction can greatly increase the size of λ-terms. Since higher-order unification is not in general decidable and since most general unifiers do not necessarily exist when unifiers do exist, unification can contribute to the search aspects of the full interpreter. λProlog addresses this by implementing a depth-first version of the unification search procedure described in [Hue75, SG88]. It was shown in [MNPS] that such unification is sufficient for determining substitutions, and in [Nad87, NM88], that this unification procedure can be smoothly integrated into the usual backtracking mechanism of logic programming languages. The higher-order unification problems we shall encounter in this dissertation are all rather simple. In fact all such problems are decidable. In addition, the presence of logic variables requires that GENERIC be implemented slightly differently than is described above. In particular, if the goal $\forall x G$ or the current program $\mathcal{P}$ contains logic variables, the new signature item y must not appear in the terms eventually instantiated for those logic variables. Several ways of handling the constraints on unification imposed by the GENERIC operation are discussed in [Mil88]. Without these checks, logic variables would not be a sound implementation technique.

2.5 Syntax for Logic Programs

Since much of this dissertation is concerned with how to specify and implement theorem provers using the class of hereditary Harrop formulas presented in this chapter, we shall need to present many such formulas. We will make such presentations by using the syntax adopted by the eLP [EP89] implementation of λProlog, which itself borrows from conventional Prolog systems.

Variables are represented by tokens with an upper case initial letter and constants are represented by tokens with a lower case initial letter. Function application is represented by juxtaposing two terms of suitable types. Application associates to the left, except for infix constants and then normal infix conventions are adopted. λ-abstraction is represented

using backslash as an infix symbol: a term of the form $\lambda x.M$ is written as `X\M`. Terms are most accurately thought of as being representatives of $\beta\eta$-conversion equivalence classes of terms. For example, the terms `X\(f X)`, `Y\(f Y)`, `(F\Y\(F Y) f)` and `f` all represent the same class of terms. In the programs in this dissertation, when a logic variable `A` has functional type, say $\tau \to \tau'$ where τ' is a base type, we sometimes write its η-long form `X\(A X)` when we want to make explicit the fact that `A` is an abstraction.

The symbols `,` and `;` represent $\wedge$ and $\vee$ respectively, and `,` binds tighter than `;`. The symbol `:-` denotes "implied-by" while `=>` denotes the converse "implies." The first symbol is often used to write the top-level connective of definite clauses as in Prolog: the clause $G \supset A$ can be written `A :- G`. Implications in goals and the bodies of clauses are always written using `=>`. Free variables in a definite clause are assumed to be universally quantified, while free variables in a goal are assumed to be existentially quantified. Universal and existential quantification within goals and definite clauses are written using the constants `pi` and `sigma` in conjunction with a λ-abstraction.

Below is an example of a (first-order) program using this syntax.

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

The goal `(sterile j)`, for example, follows from these clauses.

To specify a signature for a program, *kind declarations* are used to introduce new base types, and *type declarations* are used to introduce constants and give them types. In λProlog, types are assigned either explicitly by user declarations or by automatically inferring them from their use in programs. Explicit typings are made by adding to program clauses declarations such as:

```
kind  jar         type.
kind  insect      type.

type  sterile     jar -> o.
type  in          insect -> jar -> o.
```

Notice that from this declaration, the types of the variables and other constants in the example clauses above can easily be inferred.

λProlog permits a degree of polymorphism by allowing type declarations to contain type variables (written as capital letters). For example, `pi` is given the polymorphic typing `(A -> o) -> o`. It is also convenient to be able to build new "primitive" types from other types. This is done using type constructors. In this paper, we will need to have only one such type constructor, `list`. For example, `(list jar)` would be the type of lists all of whose entries are of type `jar`. Lists are represented as in the programming language ML by the following construction: `nil` represents an empty list of polymorphic type `(list A)`, and if `X` is of type `A` and `L` is of type `(list A)` then `X::L` represents a list

of type `(list A)` whose first element is `X` and whose tail is `L`. The programs throughout this dissertation will make use of various operations on lists. We illustrate two versions of a program for testing list membership here. These two, and all other list functions used in this dissertation are displayed in Appendix A for reference. We introduce two predicates, `memb` and `member`, both of polymorphic type `A -> (list A) -> o`, used to implement standard list membership programs. The code for these programs is as follows.

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

member X (X::L) :- !.
member X (Y::L) :- member X L.
```

The `member` program illustrates one of the few non-logical features of λProlog used in this dissertation: the cut (!). The cut is a goal which always succeeds and commits the interpreter to all choices made since the parent goal was unified with the head of the clause in which the cut occurs (see [SS86]). For example, if `L` is the list `(1::2::3::nil)` and `A` is a logic variable, the goal `(memb A L)` will succeed in three ways with `1`, `2`, `3` as the successive instances of `A`, while the goal `(member A L)` will succeed only once with `A` as `1`.

Two other non-logical features of the logic programming language that will be used in implementing theorem provers are the `write` and `read` predicates. As in Prolog, `(write A)` prints the current binding of `A` to the screen and will always succeed. The read predicate has polymorphic type `(A -> o) -> o`. A goal of the form `read (X\G)` prompts the user for input of some term `M`, and then solves the goal `((X\G) M)`. When G fails, the read also fails.

In λProlog, sets of type declarations and clauses are organized into modules. In the implementation section of this dissertation, for illustration purposes, we will present modules as a whole using the syntax of eLP. Also, in the specification section it will sometimes be helpful to present whole modules, particularly when they are imported by programs appearing in later chapters. A module consists of five parts: a name, a list of other modules imported by the module, kind declarations, type declarations, and finally, program clauses. The `lists` module in Appendix A, for example, illustrates most of this format. During an execution, all subgoals generated by clauses in a module will have access to the clauses of the current environment plus any in the modules directly imported by this module. Thus if module `a` imports module `b`, the clauses in `b` can be used in solving subgoals generated by clauses in `a`. If `b` imports `c`, the clauses of `c` are not directly available to `a`. Thus only one level in the import hierarchy is used to determine which clauses are available. Kind and type declarations on the other hand are visible all the way up the hierarchy. Thus the signature items in `c` are also signature items for `a`.

In eLP, all infix symbols must be specified in a grammar which is loaded upon entering the interpreter. All infix symbols used in this dissertation are listed in Appendix B

according to their order of precedence. In assigning types to symbols that appear in the grammar, the symbols must be quoted. For example, the list constructor `::` is declared as follows in the `lists` module.

```
type  '::'    A -> (list A) -> (list A).
```

In Chapter 1, we distinguished between specifications and implementations. Note that both specifications and implementations are programs since a program is simply a set of definite clauses associated with a signature. As stated, specifications will have both a declarative reading and an operational reading with respect to the non-deterministic interpreter described in Section 2.3. Implementations are programs that we additionally expect to execute with respect to the deterministic interpreter described in Section 2.4.

Chapter 3

Specifying Sequent and Natural Deduction Style Theorem Provers for First-Order Intuitionistic Logic

In this chapter and the next, we will illustrate the specification of theorem provers and proof checkers in hohh using several examples from both first-order and higher-order logics. We begin in this chapter by considering the specification of both sequent style and natural deduction proof systems for first-order intuitionistic logic. Since we will be specifying logics within a logic, to avoid confusion we will refer to hohh as the *meta-logic* and the logic being specified as the *object logic*.

There are two parts to a specification of a theorem prover in our language. First, we specify the syntax of a given object logic by introducing typed constants to represent the constants and connectives of the logic. Then, we give a set of clauses which encode the inference rules of a particular proof system for the object logic. Such a set of clauses provides a declarative account of the content of the proof system and, with respect to the non-deterministic interpreter given in the previous chapter, provides a specification for a theorem prover. In addition, we will specify the construction of object-level proofs. For this task, we introduce typed constants which serve as proof constructors. Programs containing proofs will serve as specifications of both theorem provers and proof checkers.

3.1 Specifying A Sequential Proof System

To represent a first-order logic, we introduce two primitive types: `form` for the object-level formulas and `i` for first-order individuals. The new type `form` serves to distinguish formulas

of the object logic from formulas of the meta-logic (which have type `o`). The connectives of the meta-logic have a set meaning, for example as given by the non-deterministic interpreter of the previous chapter, while the object-level connectives will have only the meaning attributed to them by the programs that use them. Given these new primitive types, we introduce constants for the object level connectives. These constants are declared with their types in the `fol` module. For example, the infix constant `and` is introduced for

```
module fol.

kind    i       type.
kind    form    type.

type    'and'   form -> form -> form.
type    'or'    form -> form -> form.
type    'imp'   form -> form -> form.
type    neg     form -> form.
type    forall  (i -> form) -> form.
type    exists  (i -> form) -> form.
type    false   form.
```

Module `fol`: Logical Connectives for First-Order Logic

conjunction and is given the type `form -> form -> form`. It is a constructor that takes two formulas as arguments and forms their conjunction. Often, a fixed set of constants, function symbols, propositions, and predicates is given for a particular first-order logic. We must also introduce constants for these objects, and give them types. For example, for a logic containing a constant c, a binary function symbol f, a binary predicate p, a unary predicate q, and a proposition r, we give the following declarations.

```
type    c       i.
type    f       i -> i -> i.
type    p       i -> i -> form.
type    q       i -> form.
type    r       form.
```

Using these definitions, the first-order formula $\forall x \exists y (P(x, y) \supset Q(f(x, y)))$, for example, is represented by the λ-term:

```
(forall X\ (exists Y\ ((p X Y) imp (q (f X Y)))))
```

By declaring `forall` and `exists` to take functional arguments, we have defined object-level binding of variables by quantifiers in terms of lambda abstraction, the meta-level binding operator. Thus, bound variables of the object language are identified with bound variables of the meta-language (of type `i`). Note that a similar use of λ-terms to represent formulas is also adopted in the meta-language. There, the quantifiers `pi` and `sigma` have polymorphic type `(A -> o) -> o`. As stated in Chapter 2, this representation of formulas

was introduced by Church [Chu40], and in fact has been adopted by many others [Pau86, MN87, HHP89, CH88] to express the *higher-order abstract syntax* of object logics [PE88].

In this chapter and the next, we will specify sequent style and natural deduction proof systems for both first-order intuitionistic and classical logic. The proof systems we specify will be variants of the L and N systems originally given in [Gen69]. The formulations used here will be called L_I, L_C, N_I, and N_C, for an intuitionistic sequent calculus, a classical sequent calculus, an intuitionistic natural deduction proof system, and a classical natural deduction proof system, respectively.

We begin the specification of theorem provers for first-order logic with the sequent system L_I, whose formulation is very similar to the L system in [Dum77]. In this system a sequent is written $\Gamma \longrightarrow A$ where Γ is a *set* of formulas, and A is a formula. Following convention, we write A, Γ to denote the set $\Gamma \cup \{A\}$. An *initial* sequent has the form $\Gamma \longrightarrow A$ where $A \in \Gamma$. The inference rules are given in Figure 3.1. There are no structural

$$\frac{\Gamma \longrightarrow A \qquad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\ \wedge\text{-R} \qquad \frac{A, B, \Gamma \longrightarrow C}{A \wedge B, \Gamma \longrightarrow C}\ \wedge\text{-L}$$

$$\frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B}\ \vee\text{-R} \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}\ \vee\text{-R} \qquad \frac{A, \Gamma \longrightarrow C \qquad B, \Gamma \longrightarrow C}{A \vee B, \Gamma \longrightarrow C}\ \vee\text{-L}$$

$$\frac{A, \Gamma \longrightarrow B}{\Gamma \longrightarrow A \supset B}\ \supset\text{-R} \qquad \frac{\Gamma \longrightarrow A \qquad B, \Gamma \longrightarrow C}{A \supset B, \Gamma \longrightarrow C}\ \supset\text{-L}$$

$$\frac{A, \Gamma \longrightarrow \bot}{\Gamma \longrightarrow \neg A}\ \neg\text{-R} \qquad \frac{\Gamma \longrightarrow A}{\neg A, \Gamma \longrightarrow \bot}\ \neg\text{-L}$$

$$\frac{\Gamma \longrightarrow [y/x]A}{\Gamma \longrightarrow \forall x A}\ \forall\text{-R} \qquad \frac{[t/x]A, \Gamma \longrightarrow C}{\forall x A, \Gamma \longrightarrow C}\ \forall\text{-L}$$

$$\frac{\Gamma \longrightarrow [t/x]A}{\Gamma \longrightarrow \exists x A}\ \exists\text{-R} \qquad \frac{[y/x]A, \Gamma \longrightarrow C}{\exists x A, \Gamma \longrightarrow C}\ \exists\text{-L}$$

$$\frac{\Gamma \longrightarrow \bot}{\Gamma \longrightarrow A}\ \bot\text{-R} \qquad \frac{\Gamma \longrightarrow A \qquad A, \Gamma \longrightarrow C}{\Gamma \longrightarrow C}\ \text{cut}$$

The $\forall$-R and $\exists$-L rules have the proviso that the variable y cannot appear free in the lower sequent.

Figure 3.1: The L_I Sequent Proof System for First-Order Intuitionistic Logic

rules in this presentation. The $\bot$-R rule is as in the specification of sequent systems in [Pra65], and corresponds to the usual rule for thinning on the right. We also include the cut rule. We will call the formula to which a rule is applied (*e.g.* $A \wedge B$ in the $\wedge$-L rule) the *principle* formula.

We will represent sets using lists where the order and number of copies of each element is not significant. We introduce a new primitive type `seq` for sequents and make the

following declaration.

```
type    '-->'   (list form) -> form -> seq.
```

The constant `-->` is an infix operator whose antecedent is a list of formulas and succedent is a single formula.

In this example, we will retain proofs as they are built, so we introduce another primitive type `lprf` for the type of sequential proofs. We will see that there are many choices in representing and constructing proofs. The examples given here serve merely to illustrate. Of course, a theorem prover need not build explicit proofs at all.

The basic relation between a sequent and its proofs will be represented as a binary relation at the meta-level by the infix constant `>-` declared as follows.

```
type    '>-'    lprf -> seq -> o.
```

The inference rules of the sequent calculus will be expressed as simple declarative facts about this relation. Operationally, `>-` can be viewed as the theorem proving predicate. In presenting the clauses for the inference rules, we will discuss both their declarative and operational meanings.

First, consider the $\wedge$-R inference rule in Figure 3.1 which introduces a conjunction on the right side of the sequent. The declarative reading of this inference rule is captured by the following definite clause.[1]

```
(and_r Q1 Q2) >- (Gamma --> (A and B)) :- Q1 >- (Gamma --> A), Q2 >- (Gamma --> B).
```

This clause may be read as: if `Q1` is a proof of `(Gamma --> A)` and `Q2` is a proof of `(Gamma --> B)`, then `(and_r Q1 Q2)` is a proof of `(Gamma --> (A and B))`. The rule can also be viewed as defining the constant `and_r`: it is a function from two proofs (the premises of the $\wedge$-R rule) to a new proof (its conclusion). Its logic program type is `lprf -> lprf -> lprf`.

Operationally, this rule can be employed when the sequent to be proved has a conjunction on the right of the arrow. Using the BACKCHAIN search command, the sequent and proof of the query must unify with the proof in the head of this clause. If there is a match, the AND search operation is used to verify the two new subgoals in the body of this clause. The unification here is essentially first-order.

Next, consider the two inference rules for proving disjunctions, the $\vee$-R rules in Figure 3.1. These rules have a very natural rendering as the following definite clause.

```
(or_r Q) >- (Gamma --> (A or B)) :- Q >- (Gamma --> A); Q >- (Gamma --> B).
```

Declaratively, this clause specifies the meaning of a proof of a disjunction. For `(or_r Q)` to be a proof of `(Gamma --> (A or B))`, `Q` must be a proof of either `(Gamma --> A)` or

[1] Many programs in this dissertation build or manipulate sequent-style or natural deduction proofs. We adopt the convention that `Q`, `Q1`, `Q2`, etc. will be used for sequent proof variables, and `P`, `P1`, `P2`, etc. for natural deduction proof variables.

`(Gamma --> B)`. Operationally, this clause would cause an OR search operation to be used to determine which of the subgoals in the body should succeed.

Alternatively, we could choose to specify the two rules for $\vee$-R with two clauses, and introduce two constructors `or_r1` and `or_r2`, which serve to indicate which instance of the rule is used. The corresponding definite clauses would then be as follows.

```
(or_r1 Q) >- (Gamma --> (A or B)) :- Q >- (Gamma --> A).
(or_r2 Q) >- (Gamma --> (A or B)) :- Q >- (Gamma --> B).
```

Introductions of logical constants into the antecedent of a sequent can be achieved similarly. The main difference here is that the antecedent is a list instead of a single formula. Consider the implication introduction rule, the $\supset$-L rule in Figure 3.1. This rule could be specified as the following definite clause.

```
(imp_l Q1 Q2) >- (Gamma --> C) :- memb (A imp B) Gamma,
                                  Q1 >- (Gamma --> A),
                                  Q2 >- ((B::Gamma) --> C).
```

Here, `memb` is the version of the procedure for testing list membership that does not use cut. (See Appendix A.)

Note that in the $\supset$-L rule in Figure 3.1 the principle formula $A \supset B$ appears in the set $A \supset B, \Gamma$, but in any particular instance of the rule, this formula may or may not appear in the premises. In contrast, in the definite clause specification, the most general form of the rule is used, where the formula `(A imp B)` always appears in the list `Gamma` in the subgoals.

All propositional rules for Gentzen sequential systems can be very naturally understood as combining a first-order unification step with possibly an AND or an OR search operation. We now look at specifying quantifier introduction rules. Here, the operational reading of definite clauses will use the INSTANCE and GENERIC search operations and second-order unification. Consider the $\exists$-R inference rule which can be written as the following definite clause.

```
(exists_r Q) >- (Gamma --> (exists A)) :- sigma T\ (Q >- (Gamma --> (A T))).
```

The existential formula of the conclusion of this rule is written `(exists A)` where the logic variable `A` has functional type `i -> form`. Thus `A` is an abstraction over individuals and `(A T)` represents the formula that is obtained by substituting `T` for the bound variable in `A`. Note the use of β-conversion at the meta-level to specify substitution at the object level. Declaratively, this clause reads: if there exists a term `T` (of type `i`) such that `Q` is a proof of `(Gamma --> (A T))`, then `(exists_r Q)` is a proof of `(Gamma --> (exists A))`. Operationally, we rely on second-order unification to instantiate the logic variable `A`. The existential instance `(A T)` is obtained via the interpreter's operation of β-reduction. Of course, the implementation of INSTANCE will choose a logic variable with which to instantiate `T`. By making `T` a logic variable, we do not need to commit to a specific term for the

substitution. It will later be assigned a value through unification if there is such a value which results in a proof.

The use of `sigma` in the above definite clause makes explicit the correspondence of existential introduction at the meta-level with existential introduction for this particular object logic. Note, though, that its use is not required here. The ∃-R rule could alternatively be specified as below:

```
(exists_r Q) >- (Gamma --> (exists A)) :- Q >- (Gamma --> (A T)).
```

where a meta-level negative occurrence of an existential quantifier is replaced with a positive occurrence of a meta-level universal quantifier (not shown explicitly here since, by convention, we assume universal closure at the top level). The two hohh formulas are equivalent.

This rule provides another example where there are other options in specifying the proof object. For example, it might be sensible to store inside the proof the actual substitution term used. In this case, the `exists_r` constant could be given the type `i -> nprf -> nprf`, and the inference rule specified as follows.

```
(exists_r T Q) >- (Gamma --> (exists A)) :- Q >- (Gamma --> (A T)).
```

Here, since `T` appears in both the head and body of the clause, it must be in the scope of a universal quantifier over the whole clause.

Now we consider the ∀-R rule which has the additional proviso that y is not free in Γ or $\forall x A$. Although our programming language does not contain a check for "not free in" it is still possible to specify this inference rule. This proviso is handled by using a universal quantifier at the meta-level.

```
(forall_r Q) >- (Gamma --> (forall A)) :- pi Y\ ((Q Y) >- (Gamma --> (A Y))).
```

Again `A` has functional type. In this case, so does `Q`, and the type of `forall_r` is `(i -> lprf) -> lprf`. Declaratively, this clause reads: if we have a function `Q` that maps arbitrary terms `Y` to proofs `(Q Y)` of the sequent `(Gamma --> (A Y))`, then `(forall_r Q)` is a proof of `(Gamma --> (forall A))`.

Operationally, the GENERIC search operation is used to insert a new constant of type `i` into the sequent. Since that constant will not be permitted to appear in `Gamma` or `A` the proviso will be satisfied. In the case when proof objects are defined so that they may contain formulas or first-order terms inside them, this new constant may appear in the proof term of the subgoal. Thus, it is necessary to introduce a λ-abstraction over type `i` into proof objects as we have done. Note that if the representation of proofs was such that neither formulas or terms ever appeared inside proof terms, such an abstraction would not be necessary. In this case, the clause could be specified as follows:

```
(forall_r Q) >- (Gamma --> (forall A)) :- pi Y\ (Q >- (Gamma --> (A Y))).
```

and `forall_r` would have type `lprf -> lprf`.

The remaining quantifier rules are specified similarly. The final rules are the $\bot$-R and cut rules which may be specified as follows.

```
(false_r Q) >- (Gamma --> A) :- Q >- (Gamma --> false).
(cut Q1 Q2) >- (Gamma --> C) :- Q1 >- (Gamma --> A), Q2 >- ((A::Gamma) --> C).
```

The specification is completed by the following simple definite clause for initial sequents, that is, a sequent whose succedent appears as one of the formulas in the antecedent.

```
(initial A) >- (Gamma --> A) :- memb A Gamma.
```

Here the constant `initial` is of type `form -> lprf`.

Note that this clause represents the only way in which formulas get placed inside proof terms in this specification. Depending on the later use made of proofs, it may be desirable to store other formulas inside proofs. For example, the principle formula in any of the inference rules could be stored inside proof terms making this information more directly accessible to programs manipulating these terms.

It also might be desirable, again depending on how proofs are used, for proof objects to contain less information than we have specified. For example, we may want a single sequential proof to be a proof of many different sequents, that is, the proof terms should be polymorphic. In that case, it might be desirable for the `initial` proof term not to store a formula within the proof. Instead, `initial` could have the simpler type `lprf`. In this case, it records that we have an initial sequent, but not which one. For example, the proof term `(imp_r (and_l initial))` represents a proof of $\Gamma \longrightarrow p \wedge q \supset p$ and $\Gamma \longrightarrow p \wedge q \supset q$ for any Γ, p, and q.

Of course, as was stated earlier, proof objects do not need to be built at all. The predicate `-->` could be replaced with a similar predicate, say `provable` or `true`, of type `seq -> o`.

In specifying L_I, we made use of the built-in lists of the meta-language, and the auxiliary `memb` predicate for extracting formulas of a particular form. Alternatively, we could have represented lists directly as λ-terms, in the manner used by Huet and Lang in [HL78] and by Paulson in the Isabelle theorem prover. For each type τ, a new primitive type $\bar{\tau}$ is introduced. A list of elements of type τ will have functional type $\bar{\tau} \rightarrow \bar{\tau}$. A constant C^τ of type $\tau \rightarrow \bar{\tau} \rightarrow \bar{\tau}$ serves as the list constructor, and a list of elements $a_1, \ldots, a_n$ of type τ is denoted $\lambda u.C^\tau a_1(C^\tau a_2 \ldots (C^\tau a_n u)\ldots)$. The set of lists containing the constant A is then represented as the pattern $\lambda u.l_1(C^\tau A(l_2 u))$ where l_1 and l_2 are variables of type $\bar{\tau} \rightarrow \bar{\tau}$. To specify L_I using this list representation, we add the new primitive type `lform` to be used in constructing lists of terms of type `form`. We also introduce the constant `cons` as the list constructor, and modify the type of `-->` accordingly as follows.

```
type  cons    form -> lform -> lform.
type  -->     (lform -> lform) -> form -> seq.
```

Using this list representation, the definite clauses for rules that introduce formulas on the right of the sequent arrow remain unchanged. For the rules that introduce formulas on the left, we replace the call to the `memb` program with a pattern for a list containing a formula of the appropriate shape. For example, the $\supset$-L rule will now be specified by the following clause.

```
(imp_l Q1 Q2) >- (U\(L1 (cons (A imp B) (L2 U))) --> C) :-
  Q1 >- (U\(L1 (cons (A imp B) (L2 U))) --> A),
  Q2 >- (U\(cons B (L1 (cons (A imp B) (L2 U)))) --> C).
```

Second-order matching is now required to match `U\(L1 (cons (A imp B) (L2 U)))` to an arbitrary list containing `(A imp B)`. By specifying all the rules that introduce connectives on the left of the sequent arrow in this fashion, we obtain a specification of L_I that does not require any auxiliary predicates. For readability, in the examples that follow, we will continue to use the built-in lists of the meta-language, and auxiliary predicates for manipulating them. In each case though, such lists can be replaced by λ-terms as above, and auxiliary predicates can be replaced by pattern matching on these λ-terms.

3.2 A Specification of Natural Deduction in First-Order Logic

We next consider specifying inference rules for the natural deduction N_I system, which we take to be the I system as presented in [Pra65]. The inference rules for N_I are given in Figure 3.2.

We introduce a new constant `nprf` to be the type of proofs constructed by this theorem prover. Here, the basic proof relation is between proofs and formulas (instead of sequents). We again introduce a binary infix operator, in this case `#`, with the following declaration.

```
type    '#'     nprf -> form -> o.
```

Several of the introduction rules for this system resemble rules that apply to succedents in the sequential system just considered. Those that correspond to the example clauses given in the previous section can be specified naturally as the definite clauses below.

```
(and_i P1 P2) # (A and B) :- P1 # A, P2 # B.
(or_i P) # (A or B) :- P # A; P # B.
(exists_i P) # (exists A) :- sigma T\ (P # (A T)).
(forall_i P) # (forall A) :- pi Y\ (P # (A Y)).
```

Note that the $\forall$-I rule also has a proviso, in this case, that y cannot appear free in $\forall x A$, or in any assumption on which $[y/x]A$ depends. Again, a universal quantifier at the meta-level is used. The condition that y cannot appear in $\forall x A$ is similar to the proviso on the $\forall$-R rule in the sequent system in Figure 3.1, and it is easy to see that this part of the proviso is similarly handled by meta-level universal quantification. We will see shortly that this use of meta-level universal quantification also handles the restriction on assumptions. In

$$\frac{A \qquad B}{A \wedge B}\ \wedge\text{-I} \qquad \frac{A \wedge B}{A}\ \wedge\text{-E} \qquad \frac{A \wedge B}{B}\ \wedge\text{-E}$$

$$\frac{A}{A \vee B}\ \vee\text{-I} \qquad \frac{B}{A \vee B}\ \vee\text{-I} \qquad \frac{A \vee B \qquad \overset{(A)}{C} \qquad \overset{(B)}{C}}{C}\ \vee\text{-E}$$

$$\frac{\overset{(A)}{B}}{A \supset B}\ \supset\text{-I} \qquad \frac{A \qquad A \supset B}{B}\ \supset\text{-E}$$

$$\frac{\overset{(A)}{\bot}}{\neg A}\ \neg\text{-I} \qquad \frac{A \qquad \neg A}{\bot}\ \neg\text{-E}$$

$$\frac{[y/x]A}{\forall x A}\ \forall\text{-I} \qquad \frac{\forall x A}{[t/x]A}\ \forall\text{-E}$$

$$\frac{[t/x]A}{\exists x A}\ \exists\text{-I} \qquad \frac{\exists x A \qquad \overset{([y/x]A)}{B}}{B}\ \exists\text{-E}$$

$$\frac{\bot}{A}\ \bot_I$$

The $\forall$-I rule has the proviso that the variable y cannot appear free in $\forall x A$, or in any assumption on which $[y/x]A$ depends.

The $\exists$-E rule has the proviso that the variable y cannot appear free in $\exists x A$, in B, or in any assumption on which the upper occurrence of B depends.

Figure 3.2: The N_I Natural Deduction Proof System for First-Order Intuitionistic Logic

this specification, we have assumed that there will be no way for either formulas or terms to appear inside proofs. Thus `P` in the clause for $\forall$-I is not an abstraction. Again, we have choices in specifying these rules. For example, we can specify $\vee$-I as two definite clauses as we did for $\vee$-R in the previous section. We can also include substitution terms in the clause for $\exists$-I as we did for $\exists$-R. If we do so, then first-order terms may appear in proofs, and $\forall$-I must be re-specified so that `P` is an abstraction over terms. The clauses reflecting these modifications are as follows.

```
(or_i1 P) # (A or B) :- P # A.
(or_i2 P) # (A or B) :- P # B.
(exists_i T P) # (exists A) :- P # (A T).
(forall_i P) # (forall A) :- pi Y\ ((P Y) # (A Y)).
```

In natural deduction, unlike sequential systems, we have the additional task of specifying the operation of discharging assumptions. Consider the implication introduction rule.

This rule can very naturally be specified using the definite clause below.

```
(imp_i P) # (A imp B) :- pi PA\ ((PA # A) => ((P PA) # B)).
```

This clause represents the fact that if `P` is a "proof function" which maps an arbitrary proof of `A`, say `PA`, to a proof of `B`, namely `(P PA)`, then `(imp_i P)` is a proof of `(A imp B)`. Here, the proof of an implication is represented by a function from proofs to proofs. The constant `imp_i` is declared with the following type:

```
type    imp_i   (nprf -> nprf) -> nprf.
```

Notice that while sequential proofs only contain abstractions of type `i`, natural deduction proofs may contain abstractions of both types `i` and `nprf`.

Operationally, the AUGMENT search operation plays a role in implementing the discharge of assumptions. In this case, to solve the subgoal `(pi PA\((PA # A) => ((P PA) # B)))`, the GENERIC operation is used to choose a new object, say `pa`, to play the role of a proof of the formula `A`. The AUGMENT goal is used to add this assumption about `A` and `pa`, that is `(pa # A)`, to the current set of program clauses. This clause is then available to use in the search for a proof of `B`, *i.e.*, in solving the subgoal `((P pa) # B)`. The proof of `B` will most likely contain instances of the proof of `A` (the term `pa`). The function `P` is then the result of abstracting `pa` out of the proof of `B`.

As was stated, the proviso on the $\forall$-I rule requires that the variable y does not appear in any assumptions on which the premise depends. We now can see how this restriction is handled by a universal quantifier at the meta-level. At any point in the construction of a proof, the current available assumptions will be in the form of program clauses (where the assumed formula will be associated with its proof). When the GENERIC search operation introduces a new constant for `Y`, this constant will not appear in any program clauses (in addition to not occurring in the current goal). Thus, the restriction on the occurrences of y in assumptions will be enforced.

Elimination rules are specified similarly to the introduction rules. We give two examples, the $\supset$-E and the $\exists$-E rules which are specified by the following definite clauses.

```
(imp_e P1 P2) # B :- P1 # A, P2 # (A imp B).
(exists_e P1 P2) # B :- P1 # (exists A),
                        pi Y\ (pi P\ ((P # (A Y)) => ((P2 P) # B))).
```

The $\exists$-E rule contains both a proviso handled by a universal quantifier at the meta-level, and the discharge of an assumption, again handled by meta-level universal quantification and implication. Here, `P2` is an abstraction over just the proof term `P`. The proof constructor `exists_e` is declared as follows.

```
type    exists_e    nprf -> (nprf -> nprf) -> nprf.
```

Again, if terms or formulas appear in proofs, `P2` would have to be an abstraction over `Y` also. Then the proof constructor `exists_e` would declared as follows:

```
type    exists_e    nprf -> (i -> nprf -> nprf) -> nprf.
```

and the definite clause for ∃-E would be as below.

```
(exists_e P1 P2) # B :- P1 # (exists A),
                        pi Y\ (pi P\ ((P # (A Y)) => ((P2 Y P) # B))).
```

The final rule of N_I is the $\perp_I$ rule specified below, completing the specification of this proof system.

```
(false_i P) # A :- P # false.
```

3.3 Explicit vs. Implicit Representation of Assumptions in Natural Deduction

In specifying N_I we showed that it was quite natural to specify the discharge of assumptions using universal quantification and implication at the meta-level. This representation of assumptions is implicit in the sense that assumptions are manipulated by the programming language, and no explicit programmer control is necessary. It is also possible to explicitly keep track of assumptions by storing them in a list and making the manipulation of these lists explicit in the definite clauses specifying the inference rules. Such assumption lists will contain pairs of formulas associated with their proofs, and each definite clause will have an extra argument corresponding to the current list of assumptions at the time the rule is applied. The constant `#` will still be a relation between a formula and its proof, but now will be declared as below using the new primitive type `judg` since it represents the basic judgment for natural deduction.

```
type    '#'     nprf -> form -> judg.
```

In this specification, we use the sequent arrow `-->` as our theorem proving predicate as in the sequential system, but now it will be used to form "judgment sequents," declared with the following type.

```
type    '-->'   (list judg) -> judg -> o.
```

A list of assumptions associated with their proofs appears on the left of the arrow, and the formula to be proved and its proof appear on the right. We will call such lists of pairs *contexts*. We obtain the explicit context specification of N_I via a systematic modification of the definite clauses of the "implicit context" specification of Section 3.2. For those clauses that do not involve the discharge of assumptions, we simply add a list and sequent arrow to form a judgment sequent in the head and subgoals of each clause. For example, the clauses for ∧-I and ∧-E are as follows.

```
Gamma --> (and_i P1 P2) # (A and B) :- Gamma --> P1 # A, Gamma --> P2 # B.

Gamma --> (and_e P) # A :- Gamma --> P # (A and B); Gamma --> P # (B and A).
```

The discharge of assumptions as in the $\supset$-I rule is specified as below where the new assumption gets added to the context rather than the program.

```
Gamma --> (imp_i P) # (A imp B) :- pi PA\ (((PA # A)::Gamma) --> (P PA) # B).
```

The remaining rules that discharge assumptions are given below. They are obtained by similarly modifying the corresponding definite clause of the implicit context specification described in Section 3.2.

```
Gamma --> (neg_i P) # (neg A) :- pi PA\ (((PA # A)::Gamma) --> (P PA) # false).

Gamma --> (or_e P P1 P2) # C :- Gamma --> P # (A or B),
   pi PA\ (((PA # A)::Gamma) --> (P1 PA) # C),
   pi PB\ (((PB # B)::Gamma) --> (P2 PB) # C).

Gamma --> (exists_e P1 P2) # B :- Gamma --> P1 # (exists A),
   pi Y\ (pi P\ (((P # (A Y))::Gamma) --> (P2 P) # B)).
```

In addition we will need the following clause to complete proofs.

```
Gamma --> P # A :- memb (P # A) Gamma.
```

The membership test in the above clause is equivalent to unification of a goal with an atomic clause in the implicit context specification of the N_I proof system.

As a result of these simple modifications we obtain a new specification that is equivalent to the previous one. In fact, the procedure outlined above provides a mechanical method of formulating a corresponding sequent system for any natural deduction system. We will see in Chapter 8 that a further modification of the explicit context specification for the N_I system will correspond to the L_I sequent proof system.

In Chapter 7, we will see that the use of lists to represent assumptions has an additional advantage. For implementation purposes, it will often be desirable to have explicit control over the manipulation of assumptions. For example, removing an assumption when it is known that it is no longer needed can reduce the size of the search space. For this task, list manipulation provides extra flexibility that meta-level implication cannot.

3.4 Proof Terms for Natural Deduction

In discussing the specification of the inference rules for L_I and N_I in the previous sections, we indicated that there were often many choices in how rules were represented as definite clauses. Although, as stated earlier, the choice of proof term will ultimately depend on what the proofs will be used for, in this section we examine the choices for N_I in more detail, leading up to a representation that, given a proof term, allows us to fully recover the deduction tree that it represents. We first give a precise definition of deductions in N_I, so that we may better see the correspondence between proof terms and the deduction trees they represent. The definition of deduction that we present is based on the definition

in [Pra65] that uses discharge functions. First, we slightly modify the inference rules of Figure 3.2 to make them more precise. Those rules which are modified are given in Figure 3.3. In particular the two versions of the $\wedge$-E and $\vee$-I rules are each given different

$$\frac{A \wedge B}{A}\ \wedge\text{-E}_1 \qquad \frac{A \wedge B}{B}\ \wedge\text{-E}_2$$

$$\frac{A}{A \vee B}\ \vee\text{-I}_1 \qquad \frac{B}{A \vee B}\ \vee\text{-I}_2$$

$$\frac{[y/x]A}{\forall x A}\ \forall\text{-I}(y) \qquad \frac{\forall x A}{[t/x]A}\ \forall\text{-E}(t)$$

$$\frac{[t/x]A}{\exists x A}\ \exists\text{-I}(t) \qquad \frac{\exists x A \qquad \overset{([y/x]A)}{\overset{\vdots}{B}}}{B}\ \exists\text{-I}(y)$$

Figure 3.3: Modified Rules for a Precise Formulation of the N_I Proof System

names, to avoid ambiguous applications of either, *e.g.*, concluding $A \vee A$ from A by an application of $\vee$-I. Also the quantifier rules are parameterized by the substitution term or variable t or y, so that when the variable bound by quantification doesn't appear in the formula, it is still possible to retrieve the substitution term.

In the rules of N_I, discharge of assumptions is indicated by parentheses. For example, in the $\supset$-I rule, (A) indicates the discharge of zero or more occurrences of A. The role of discharge functions is to make explicit which hypotheses are discharged by which rule application in a given deduction. To formalize this notion, we introduce several definitions based on those in [Pra65]. First, we say that the premise of an elimination rule containing the connective for which the rule is named is the *major premise*, and other premises are *minor premises*. In the $\supset$-E rule, for example, $A \supset B$ is the major premise, and A is the minor premise. We call a tree constructed using these inference rules as schemas in the obvious way a *deduction tree*. The formula occurring at the root in a deduction tree is the *end-formula*. We distinguish between formulas and occurrences of formulas as they appear in deduction trees by defining an *occurrence* to be a pair consisting of a tree address and a formula occurring at that address. An *assumption* is an occurrence that appears as a leaf in a deduction. Given a deduction tree Π, a *discharge function* $\mathcal{F}$ on Π is a partial function mapping assumptions to occurrences in Π.

An N_I *deduction of* C *depending on* Γ is then defined to be a pair $(\Pi, \mathcal{F})$ such that the following conditions hold.

1. Π is a deduction tree whose end-formula is C.

2. $\mathcal{F}$ is a discharge function on Π such that whenever $\mathcal{F}(\mathcal{A}) = \mathcal{B}$, $\mathcal{A}$ is a leaf in the subtree whose root is $\mathcal{B}$, and either

   (a) $\mathcal{B}$ is the premise of $\supset$-I whose conclusion is $A \supset B$, and $\mathcal{A}$ is an occurrence of A,

   (b) $\mathcal{B}$ is the middle premise of $\vee$-E whose major premise is $A \vee B$, and $\mathcal{A}$ is an occurrence of A,

   (c) $\mathcal{B}$ is the rightmost premise of $\vee$-E whose major premise is $B \vee A$, and $\mathcal{A}$ is an occurrence of A,

   (d) $\mathcal{B}$ is the minor premise of $\exists$-E(y) whose major premise is $\exists x A$, and $\mathcal{A}$ is an occurrence of $[y/x]A$.

3. Γ is the set of assumptions in Π that are not in the domain of $\mathcal{F}$.

Whenever $\mathcal{F}(\mathcal{A}) = \mathcal{B}$, we say that $\mathcal{A}$ is *discharged at* $\mathcal{B}$ *in* Π. A deduction $(\Pi, \mathcal{F})$ is an N_I *proof of* A if $(\Pi, \mathcal{F})$ is a deduction of A depending on $\emptyset$. In other words, $(\Pi, \mathcal{F})$ is a proof when $\mathcal{F}$ is a total function on the assumptions in Π.

Consider the proof in Figure 3.4 of $\forall x q(x) \supset \exists x q(x)$ in a language that contains at least a constant c, a proposition p, and a unary predicate q, with a discharge function that maps p to itself and $\forall x q(x)$ to the premise of the lower application of $\supset$-I: $\exists x q(x)$. Using

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x q(x)}{q(c)}\ \forall\text{-E}(c) \qquad \dfrac{p}{p \supset p}\ \supset\text{-I}}{q(c) \wedge (p \supset p)}\ \wedge\text{-I}}{q(c)}\ \wedge\text{-E}_1}{\exists x q(x)}\ \exists\text{-I}(c)}{\forall x q(x) \supset \exists x q(x)}\ \supset\text{-I}$$

Figure 3.4: N_I Proof of $\forall x q(x) \supset \exists x q(x)$

the representation for proof terms of Section 3.2 that did not include substitution terms, the proof term for this deduction is:

```
(imp_i P\(exists_i (and_e1 (and_i (forall_e P) (imp_i Q\Q))))).
```

This representation is minimal in the sense that proofs contain enough information to uniquely determine which definite clause was used at each step in its construction, but no more. Using this representation, there will be proof terms that correspond to many deductions. For example, from the above proof term, it is impossible to know that the subterm `(imp_i Q\Q)` is a proof of $p \supset p$. It could also be a proof of $q(c) \supset q(c)$ for example. In fact, the above proof term represents any deduction of the above form with the substitution term c replaced by any first-order term, the proposition p replaced by any formula, and the formula $q(x)$ replaced by any formula.

There are various ways in which we can add information to proof terms to distinguish proof terms for different deductions from one another. One way to make proof terms more precise is to include substitution terms, as discussed in Section 3.1 and 3.2. We modify the proof term given previously as follows.

```
(imp_i P\(exists_i c (and_e1 (and_i (forall_e c P) (imp_i Q\Q)))))
```

By including this information in proof terms, deduction trees of the same form that contain different substitution terms will no longer be identified. Including formulas in proofs will provide further information. For example, the proof term below provides the missing conjunct in the application of $\wedge$-E in the above deduction.

```
(imp_i P\(exists_i c (and_e1 (p imp p) (and_i (forall_e c P) (imp_i Q\Q)))))
```

Including this information allows us to determine that the subproof `(imp_i Q\Q)` is a proof of `(p imp p)`.

One way to include information to uniquely determine the end-formula of a deduction is to pair each proof term with the term of type `form` representing the end-formula. Continuing our example, the above proof may be paired with the following formula.

```
((forall X\ (q X)) imp (exists X\ (q X)))
```

For this example, the above tree is now the unique deduction corresponding to this proof term/formula pair.

Note that in the above proof term, the meta-variables `P` and `Q` represent proofs of the assumptions $\forall x q(x)$ and p, respectively. The fact that `P` is bound in the argument to the outer occurrence of `imp_i` corresponds to the fact that $\forall x q(x)$ is discharged at the premise of the lower application of $\supset$-I and similarly for `Q` bound by the inner occurrence of `imp_i`, and the corresponding assumption p. Figure 3.5 (a) illustrates a deduction in which there are two possible discharge functions that make it a proof. The assumption p could be

$$\dfrac{\dfrac{p}{p \supset p}\,\supset\text{-I}}{p \supset (p \supset p)}\,\supset\text{-I} \qquad\qquad \dfrac{\dfrac{\dfrac{\dfrac{\forall x(q(x) \wedge p)}{q(y) \wedge p}\,\forall\text{-E}(y)}{q(y)}\,\wedge\text{-E}_1}{\forall x q(x)}\,\forall\text{-I}(y)}{\forall x(q(x) \wedge p) \supset \forall x q(x)}\,\supset\text{-I}$$

(a) (b)

Figure 3.5: N_I Proofs of $p \supset (p \supset p)$ and $\forall x(q(x) \wedge p) \supset \forall x q(x)$

discharged by either application of $\supset$-I. The following two proof terms correspond to the discharge of p at the lower and upper application of $\supset$-I respectively.

```
(imp_i P\(imp_i Q\P))
(imp_i P\(imp_i Q\Q))
```

As another example, consider the proof in Figure 3.5 (b) of $\forall x(q(x) \wedge p) \supset \forall x q(x)$. The proviso on the $\forall$-I rule states that the variable y used to instantiate the formula $\forall x q(x)$ cannot appear in the conclusion of the rule or in any assumptions that are not discharged in the subtree above the application of this rule. As a result we can consider the variable y as being bound in the subtree in which it is introduced. We identify deductions up to the names of these "bound variables," *i.e.* the variables appearing as arguments to $\forall$-I and $\exists$-E. For example, we want to identify the proof in Figure 3.5 (b) with all proof trees that can be obtained by replacing all occurrences of y in the subtree above the application of $\forall$-I(y) with any other variable. (The same identification is made in [Pra71].) This identification corresponds to identifying α-convertible proof terms. For example, the renaming of y in the above tree corresponds to the renaming of `Y` in the corresponding proof term:

```
(imp_i P\(forall_i Y\(and_e1 p (forall_e Y P)))).
```

The first example illustrated that the `and_e1` proof constructor must take as an argument the conjunct that is dropped in applying the rule if proof terms are to be constructed in such a way that the deduction trees they represent can be recovered. In general, in order for proof terms to stand in a one-to-one relation to deductions, up to the equivalence just described, other proof constructors for the elimination rules must take formulas as additional arguments. The constants used in constructing such proof terms and their types are given in the `nprf` module on page 33. This module imports the `fol` module which introduces the primitive types `i` and `form` and contains declarations for the connectives of first-order logic (see page 18).

The use of this proof representation requires a slight modification to the clauses for the elimination rules to include certain formulas inside proof terms. The clauses for the introduction and $\bot_I$ rules need not be modified from those given in the previous section. The complete set of clauses is given in the `niprover` module on page 34. Note that there are two clauses for $\vee$-I and $\wedge$-E, and that clauses include substitution information. Also, as discussed in Section 3.2, since substitution terms are included in proof terms, the argument to `forall_i` and the third argument to `exists_e` must be abstractions over type `i`.

An alternate approach to representing proofs so that they stand in a one-to-one relation to deductions, up to the desired equivalence is to introduce a "maximal" proof representation, in the sense that proof terms contain even more formulas, enough so that proof terms alone (without an associated formula) stand in a one-to-one relation to deductions. We will see in Chapter 5 that in compiling LF signatures specifying various object logics into definite clauses, we obtain such a maximal representation for proofs. With such a representation it is always possible to determine the end-formula of a deduction by extracting the information directly from the proof term. In contrast, in the "intermediate" representation we have described, the proof term must be paired with a formula in order to recover in full the deduction which it represents.

```
module nprf.

import  fol.

kind    nprf        type.

type    and_i       nprf -> nprf -> nprf.
type    and_e1      form -> nprf -> nprf.
type    and_e2      form -> nprf -> nprf.
type    or_i1       nprf -> nprf.
type    or_i2       nprf -> nprf.
type    or_e        form -> form -> nprf ->
                      (nprf -> nprf) -> (nprf -> nprf) -> nprf.
type    imp_i       (nprf -> nprf) -> nprf.
type    imp_e       form -> nprf -> nprf -> nprf.
type    neg_i       (nprf -> nprf) -> nprf.
type    neg_e       form -> nprf -> nprf -> nprf.
type    exists_i    i -> nprf -> nprf.
type    exists_e    (i -> form) -> nprf -> (i -> nprf -> nprf) -> nprf.
type    forall_i    (i -> nprf) -> nprf.
type    forall_e    i -> (i -> form) -> nprf -> nprf.
type    false_i     nprf -> nprf.
```

Module `nprf`: Proof Term Constructors for N_I

3.5 A Theorem Prover That Constructs Normal N_I Proofs

Note that in the specification of L_I, we can leave out the clause for the cut rule since, by Gentzen's cut-elimination result [Gen69], the proof system remains complete without it. Without this clause, we have a specification of an L_I theorem prover that builds cut-free proofs. A corresponding notion to cut-free proofs in L systems is normal proofs in N systems. In this section, we specify a theorem prover that only builds normal natural deduction proofs. In N_I it is not a matter of simply adding or removing clauses, but of specifying the rules in a different manner.

The main condition for an N_I deduction to be normal is that it must contain no *maximal formula*, that is, a formula that is the conclusion of an I-rule or $\perp_I$ and the major premise of an E-rule, since such applications are redundant (as in the example in Figure 3.6 (a) on page 35). For the fragment of N_I without the $\vee$ and $\exists$ connectives, this condition is taken as the definition of normal. With these connectives in the logic the condition must be made slightly stronger, and requires some further definitions. A *segment* in a deduction is defined to be a sequence of occurrences $A_1, \ldots, A_n$ such that A_1 is not the conclusion of an application of $\vee$-E or $\exists$-E, for $i = 1, \ldots, n-1$, A_i is a minor premise of an application of $\vee$-E or $\exists$-E, and A_n is not the minor premise of an application of $\vee$-E or $\exists$-E. All occurrences in a segment are occurrences of the same formula. The deduction in

```
module niprover.

import  nprf.

type    '#'     nprf -> form -> o.

(and_i P1 P2) # (A and B) :- P1 # A, P2 # B.
(or_i1 P) # (A or B) :- P # A.
(or_i2 P) # (A or B) :- P # B.
(imp_i P) # (A imp B) :- pi PA\ ((PA # A) => ((P PA) # B)).
(neg_i P) # (neg A) :- pi PA\ ((PA # A) => ((P PA) # false)).
(exists_i T P) # (exists A) :- P # (A T).
(forall_i P) # (forall A) :- pi Y\ ((P Y) # (A Y)).
(false_i P) # A :- P # false.
(and_e1 B P) # A :- P # (A and B).
(and_e2 A P) # B :- P # (A and B).
(or_e A B P P1 P2) # C :- P # (A or B),
                         pi PA\ ((PA # A) => ((P1 PA) # C)),
                         pi PB\ ((PB # B) => ((P2 PB) # C)).
(imp_e A P1 P2) # B :- P1 # A, P2 # (A imp B).
(neg_e A P1 P2) # false :- P1 # A, P2 # (neg A).
(exists_e A P1 P2) # B :- P1 # (exists A),
                         pi Y\ (pi P\ ((P # (A Y)) => ((P2 Y P) # B))).
(forall_e T A P) # (A T) :- P # (forall A).
```

Module `niprover`: Specification of N_I

Figure 3.6 (b) contains a segment of length 3 of occurrences of $A \wedge B$. As in [Pra65], we will say a segment is the premise of an application of a rule when its last formula is the premise of the rule. A *maximal segment* is a segment that begins with a conclusion of an application of an I-rule or $\perp_I$ and ends with a major premise of an E-rule. The segment of length 3 in Figure 3.6 (b) is in fact maximal. Note that a maximal formula is a special case of a maximal segment. A *normal* deduction is then defined to be a deduction that contains no maximal segment.

Normal deductions can be characterized in terms of the form of certain sequences of formulas called paths. A *path* in a deduction is a sequence of occurrences $A_1, \ldots, A_n$ such that the following conditions hold. (1) A_1 is a leaf that is not discharged by an application of $\vee$-E or $\exists$-E. (2) For $i = 1, \ldots, n-1$, A_i is not the minor premise of an application of $\supset$-E or $\neg$-E. If A_i is the major premise of an application of $\vee$-E or $\exists$-E, then A_{i+1} is an assumption discharged by this application. Otherwise, A_{i+1} is the conclusion of the rule for which A_i is a premise. (3) A_n is either the minor premise of $\supset$-E or $\neg$-E or the end-formula of the deduction. In a normal deduction, each path contains a series of segments divided into three parts, an *E-part* followed by a *minimum segment*, followed by an *I-part*. The E-part and I-part may be empty. Each segment in the E-part is a major premise of an

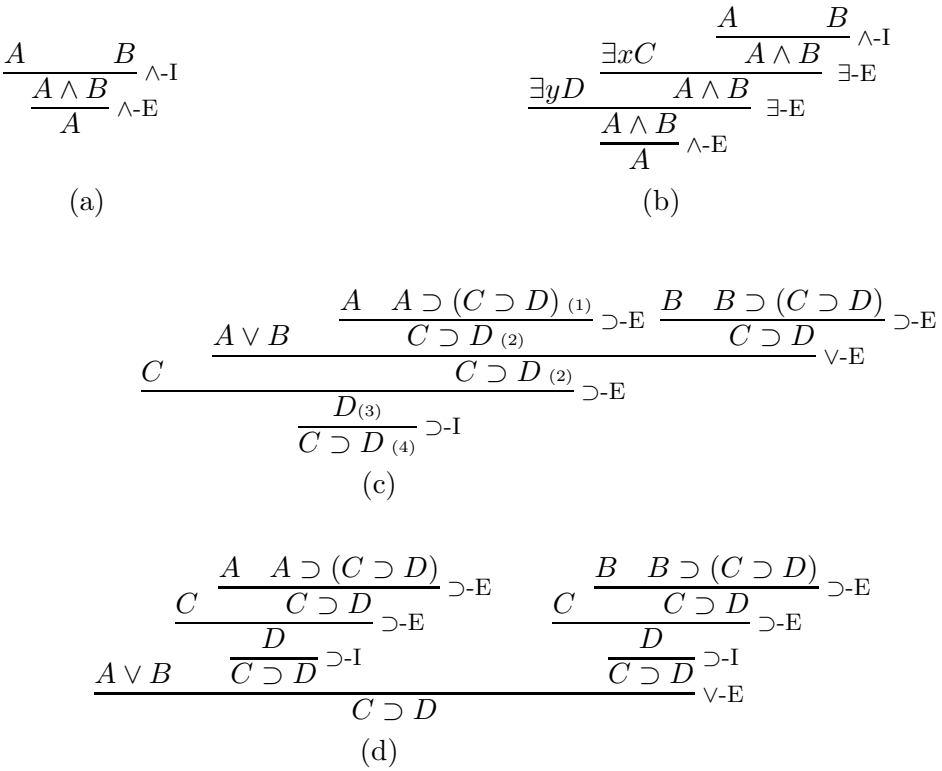

Figure 3.6: Some Example Fragments of N_I Deductions

E-rule. The minimum segment is either the last segment or the premise of an I-rule or $\perp_I$. Each segment in the I-part is a conclusion of an I-rule. Figure 3.6 (c) illustrates a path of length 5 (containing 4 segments) in a normal deduction. The other paths in Figure 3.6 (c) are (1) a similar path starting with $B \supset (C \supset D)$, (2) $A \wedge B, A$ (3) $A \wedge B, B$ and (4) C. Note that since applications of $\vee$-E and $\exists$-E occur in the middle of segments, their minor premises and conclusion may appear in the E-part, I-part, or the minimum segment of paths through them. In Figure 3.6 (c), the application of $\vee$-E occurs in the E-part of the segment shown there.

We define an *E-part deduction* to be a deduction whose end-formula is in the minimum segment of all paths through it. In other words, all paths through the end-formula only have a minimum segment and possibly an E-part, but no I-part. For example, the subtree rooted at D in Figure 3.6 (c) is an E-part deduction. It is the minimum segment of the following 2 paths: (1) $A \supset (C \supset D), C \supset D, C \supset D, D$ and (2) $B \supset (C \supset D), C \supset D, C \supset D, D$.

We present several variations of specifications that build normal N_I deductions. The division of normal deductions into E-parts and I-parts will be reflected in these specifications. The first specification we present here can be viewed as a modification of the `niprover` module presented in Section 3.4 (see page 34). As in `niprover`, we will adopt

the "intermediate" proof representation, which includes substitution terms and some formulas in proofs and provides two clauses each for the $\wedge$-E and $\vee$-I rules. We will use two relations in specifying the definite clauses for the inference rules, one for building E-parts, and one for building I-parts. We introduce the constant `#e` for the E-part relation, and also again use `#`, this time to relate a formula with a normal deduction. Both have type `(nprf -> form -> o)`. If a goal of the form `(P #e A)` is provable, then `P` represents an E-part deduction of `A`. A goal of the form `(P # A)` is provable if `P` is a normal deduction of `A`. Operationally, the clauses for the `#e` relation will apply E-rules, and the clauses for the `#` relation will apply I-rules and join I-parts and E-parts at the minimum segment.

The introduction rules and $\perp_I$ are specified below. They are all specified as before in `niprover`, except that discharged assumptions are added as facts about the `#e` relation since they will occur at the leaves in deductions and will always occur in the E-parts or minimum segments of paths through them.

```
(and_i P1 P2) # (A and B) :- P1 # A, P2 # B.
(or_i1 P) # (A or B) :- P # A.
(or_i2 P) # (A or B) :- P # B.
(imp_i P) # (A imp B) :- pi PA\ ((PA #e A) => ((P PA) # B)).
(neg_i P) # (neg A) :- pi PA\ ((PA #e A) => ((P PA) # false)).
(forall_i P) # (forall A) :- pi Y\ ((P Y) # (A Y)).
(exists_i T P) # (exists A) :- P # (A T).
(false_i P) # A :- P # false.
```

The elimination rules except for $\vee$-E and $\exists$-E are specified as follows using the `#e` predicate since the major premise of these rules can only appear in E-parts of proofs.

```
(and_e1 B P) #e A :- P #e (A and B).
(and_e2 A P) #e B :- P #e (A and B).
(imp_e A P1 P2) #e B :- P1 # A, P2 #e (A imp B).
(neg_e A P1 P2) #e false :- P1 # A, P2 #e (neg A).
(forall_e T A P) #e (A T) :- P #e (forall A).
```

Note that the minor premise of the $\supset$-E rule (and similarly for $\neg$-E) is specified as the subgoal `(P1 # A)`. This reflects the fact that the subproof at the minor premise can be an arbitrary normal proof. By the definition of path, the root of this subproof will always be the last formula occurrence in the paths through it.

$\vee$-E and $\exists$-E are each specified by two definite clauses, the first corresponding to when the minor premises and conclusion occur in the I-parts (or minimum segment) of paths, and the second when they occur in E-parts (or minimum segment).

```
(or_e A B P P1 P2) # C :- P #e (A or B),
                          pi PA\ ((PA #e A) => ((P1 PA) # C)),
                          pi PB\ ((PB #e B) => ((P2 PB) # C)).

(or_e A B P P1 P2) #e C :- P #e (A or B),
                           pi PA\ ((PA #e A) => ((P1 PA) #e C)),
```

```
                          pi PB\ ((PB #e B) => ((P2 PB) #e C)).

(exists_e A P1 P2) # B :- P1 #e (exists A),
                          pi Y\ (pi P\ ((P #e (A Y)) => ((P2 Y P) # B))).

(exists_e A P1 P2) #e B :- P1 #e (exists A),
                           pi Y\ (pi P\ ((P #e (A Y)) => ((P2 Y P) #e B))).
```

We must also add the additional clause below which serves to join I-parts and E-parts at the minimum segment.

```
P # A :- P #e A.
```

Its declarative reading is that an E-part deduction is a normal deduction.

We now discuss several modifications that can be made to this specification. Our goals in this presentation are twofold. The first is simply to present alternative ways of specifying the rules of N_I so that only normal proofs get constructed. The second is to obtain a specification that corresponds to the clauses in the specification of L_I presented in Section 3.1 which, without the clause for the cut rule, constructs cut-free proofs. In Chapter 8, we will see that from such a specification, we can rather directly obtain a program that translates L_I proofs to N_I proofs.

In any N_I deduction, any occurrence of $\bot$ is always in the minimum segment of paths through it, since it cannot be the major premise of an E-rule or the conclusion of an I-rule. As a result, we have the option of specifying the $\neg$-E rule as follows where the only difference is the use of `#` instead of `#e` in the head of the clause.

```
(neg_e A P1 P2) # false :- P1 # A, P2 #e (neg A).
```

With this clause replacing the one given earlier, the role of the clauses for the `#` relation is to build I-parts in a goal directed fashion, and possibly also add the last segment in the E-parts of paths through `false`. Clauses for `#e` will then add the remaining E-parts.

We can simplify the above specification of the N_I rules if we consider the following refinement of normal deductions. We define an *E-segment* in a normal deduction to be a segment whose last occurrence is the conclusion of an application of $\vee$-E or $\exists$-E and the major premise of an E-rule. Note that maximal segments are a special case of E-segments. As pointed out to Prawitz by Martin-Löf [Pra71], the definition of normal can be sharpened to require that normal deductions contain no E-segments or maximal formulas. We call such deductions *E-normal* deductions. In an E-normal deduction all minor premises and conclusions of applications of $\vee$-E and $\exists$-E will appear only in I-parts or minimum segments of paths. Note that the deduction in Figure 3.6 (c), although it is normal, is not E-normal. Figure 3.6 (d) contains a modification of the proof in (c) that meets the extra restriction on segments to make it E-normal.

We can modify the above clauses to reflect this sharpened normal form by simply eliminating the clauses for $\vee$-E and $\exists$-E with the `#e` relation in the head, those for applications

such that the minor premises and conclusion appear in E-parts. By eliminating these two clauses, we obtain a theorem prover that builds E-normal deductions.

We now illustrate an alternative specification that builds E-normal proofs and will correspond to the specification of L_I that builds cut-free proofs. This specification will only have clauses for the `#` predicate. Atomic clauses for the `#e` predicate will associate formulas with E-part deductions and will only get added dynamically during program execution. Note that the specification that builds E-normal deductions removed two clauses with `#e` in the head, and the alternate specification of the $\neg$-E rule illustrated how to remove another and replace it with a clause with `#` in the head. The remaining clauses with `#e` in the head are those specifying the $\wedge$-E, $\supset$-E, and $\forall$-E rules. These rules can also be modified. The two rules for $\wedge$-E, for example, can be specified by the following clause with `#` in the head.

```
PC # C :- P #e (A and B),
          (((and_e1 B P) #e A) => (((and_e2 A P) #e B) => (PC # C))).
```

Operationally, in attempting to find a normal proof for any formula `C`, this clause will look for an E-part deduction of a conjunction `(A and B)`, apply both versions of the $\wedge$-E rule to it to obtain two new E-part deductions, add the new subproofs as atomic program clauses, and attempt to find a normal proof for `C` in the environment extended with these new assumptions. We can similarly specify the $\supset$-E and $\forall$-E rules as the following clauses.

```
PC # C :- P2 #e (A imp B), P1 # A,
          (((imp_e A P1 P2) #e B) => (PC # C)).

PC # C :- P #e (forall A),
          (((forall_e T A P) #e (A T)) => (PC # C)).
```

The full specification of N_I with these modifications is given in the `ninormal` module.

As stated and proved in [Pra65], normal deductions have the *subformula property*, that is, every formula occurring in a normal deduction of A from Γ is a subformula of A or of some formula in Γ. In fact, every formula occurring in an E-part of a path is a subformula of a formula in Γ, every formula occurring in an I-part of a path is a subformula of A, and every formula occurring in a minimum segment is a subformula of both A and some formula in Γ. This property is reflected in the following operational description of the `ninormal` module. The clauses for the I-rules apply the rules in a backward direction so that the formulas in the subgoals (which correspond to the premises) are always subformulas of the formula in the head of the clause (the conclusion). In contrast, the clauses for the E-rules (except $\vee$-E and $\exists$-E) apply the rules in a forward direction from the assumptions so that the conclusion is always a subformula of the major premise. In applying E-rules, new E-part deductions get built from existing E-part deductions and are then added to the program as new facts about the `#e` relation.

```
module ninormal.

import  nprf.

type    '#'     nprf -> form -> o.
type    '#e'    nprf -> form -> o.

P # A :- P #e A.

(and_i P1 P2) # (A and B) :- P1 # A, P2 # B.
(or_i1 P) # (A or B) :- P # A.
(or_i2 P) # (A or B) :- P # B.
(imp_i P) # (A imp B) :- pi PA\ ((PA #e A) => ((P PA) # B)).
(neg_i P) # (neg A) :- pi PA\ ((PA #e A) => ((P PA) # false)).
(forall_i P) # (forall A) :- pi Y\ ((P Y) # (A Y)).
(exists_i T P) # (exists A) :- P # (A T).
(false_i P) # A :- P # false.

PC # C :- P #e (A and B),
          (((and_e1 B P) #e A) => (((and_e2 A P) #e B) => (PC # C))).

PC # C :- P2 #e (A imp B), P1 # A,
          (((imp_e A P1 P2) #e B) => (PC # C)).

PC # C :- P #e (forall A),
          (((forall_e T A P) #e (A T)) => (PC # C)).

(neg_e A P1 P2) # false :- P1 # A, P2 #e (neg A).

(or_e A B P P1 P2) # C :- P #e (A or B),
                          pi PA\ ((PA #e A) => ((P1 PA) # C)),
                          pi PB\ ((PB #e B) => ((P2 PB) # C)).

(exists_e A P1 P2) # B :- P1 #e (exists A),
                          pi Y\ (pi P\ ((P #e (A Y)) => ((P2 Y P) # B))).
```

Module `ninormal`: Specification of N_I that Constructs Normal Deductions

Chapter 4

Specifying Other Logics

All of the example specifications presented so far have been for first-order intuitionistic logic. In this chapter, we examine several others including classical logic, the λ-calculus, and higher-order logic.

4.1 Specification of Proof Systems for Classical Logic

Classical logic can be specified similarly to intuitionistic logic. We begin with a sequential system L_C. L_C can be obtained from the L_I system in Figure 3.1 by allowing more than one formula on the right of a sequent, and modifying the inference rules accordingly. The complete proof system appears in Figure 4.1. In this system, an initial sequent has the form $\Gamma \longrightarrow \Delta$ where Γ and Δ contain a common formula. The $\perp$-I rule as in Figure 3.1 is no longer needed, and the $\vee$-R rule becomes the dual of the $\wedge$-L rule.

To specify this system, the sequent arrow `-->` will have the modified type `(list form) -> (list form) -> seq` to reflect the fact that the right of the sequent also contains a set of formulas. All rules that apply to formulas on the right must now test for list membership also. For example, the $\wedge$-R rule becomes:

```
(and_r P1 P2) >- (Gamma --> Delta) :- memb (A and B) Delta,
                                      P1 >- (Gamma --> [A| Delta]),
                                      P2 >- (Gamma --> [B| Delta]).
```

A similar modification to all of the rules that operate on the succedent provides a complete specification for this proof system. We will return to this example in Chapter 6 where we implement an automatic theorem prover for classical first-order logic based on the L_C proof system.

$$\frac{\Gamma \longrightarrow A, \Delta \qquad \Gamma \longrightarrow B, \Delta}{\Gamma \longrightarrow A \wedge B, \Delta}\ \wedge\text{-R} \qquad \frac{A, B, \Gamma \longrightarrow \Delta}{A \wedge B, \Gamma \longrightarrow \Delta}\ \wedge\text{-L}$$

$$\frac{\Gamma \longrightarrow A, B, \Delta}{\Gamma \longrightarrow A \vee B, \Delta}\ \vee\text{-R} \qquad \frac{A, \Gamma \longrightarrow \Delta \qquad B, \Gamma \longrightarrow \Delta}{A \vee B, \Gamma \longrightarrow \Delta}\ \vee\text{-L}$$

$$\frac{A, \Gamma \longrightarrow B, \Delta}{\Gamma \longrightarrow A \supset B, \Delta}\ \supset\text{-R} \qquad \frac{\Gamma \longrightarrow A, \Delta \qquad B, \Gamma \longrightarrow \Delta}{A \supset B, \Gamma \longrightarrow \Delta}\ \supset\text{-L}$$

$$\frac{A, \Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \neg A, \Delta}\ \neg\text{-R} \qquad \frac{\Gamma \longrightarrow A, \Delta}{\neg A, \Gamma \longrightarrow \Delta}\ \neg\text{-L}$$

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

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

$$\frac{\Gamma \longrightarrow A, \Delta \qquad A, \Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \text{cut}$$

The $\forall$-R and $\exists$-L rules have the proviso that the variable y cannot appear free in the lower sequent.

Figure 4.1: The L_C Sequent Proof System for First-Order Classical Logic

The natural deduction system N_C for classical logic differs from N_I only in the replacement of the $\perp_I$ rule with the $\perp_C$ rule below. N_C is the same as the C system in [Pra65].

$$\frac{\begin{array}{c}(\neg A)\\ \perp\end{array}}{A}\ \perp_C$$

We simply replace the definite clause for $\perp_I$ presented in Section 3.2 with the following clause for $\perp_C$ to obtain the specification of a natural deduction theorem prover for classical logic.

```
(false_c P) # A :- pi P1\ ((P1 # (neg A)) => ((P P1) # false)).
```

4.2 Specifying the Untyped and Simply-Typed λ-Calculus

We now turn to an example that is somewhat different from previous ones. We begin in this section by specifying untyped λ-terms and certain operations on them. We then specify a proof system for type-checking and inference in the simply-typed λ-calculus.

We introduce the type `tm` for untyped terms, and two constants `abs` and `app` declared below for encoding terms.

```
type    abs     (tm -> tm) -> tm.
type    app     tm -> tm -> tm.
```

Using these two constants all closed untyped terms can be encoded. For example, the term $\lambda f \lambda n.(f(fn))$ is represented as

```
(abs F\ (abs N\ (app F (app F N)))).
```

This encoding is essentially the one used in [Mey81]: `abs` corresponds to the function Ψ, for coercing functions into terms, and `app` corresponds to the function Φ for coercing terms into functions.

By making the argument to `abs` functional, we use abstraction at the meta-level to represent abstraction at the object-level, and thus the bound variables of each level are identified, just as bound variables in first-order logic were identified with meta-variables in previous specifications. As a result, this representation does not distinguish between α-convertible terms at the object-level.

$$(\lambda x.M)N \to [N/x]M \quad (\beta) \qquad\qquad \lambda x.Mx \to M \quad (\eta)$$

$$\frac{M \to N}{\lambda x.M \to \lambda x.N}\,\xi \qquad \frac{M \to P}{MN \to PN}\,\text{CONG1} \qquad \frac{N \to P}{MN \to MP}\,\text{CONG2}$$

$$\frac{M \to N}{M =_{\beta\eta} N}\,\text{RED} \qquad\qquad M =_{\beta\eta} M \quad (\text{REFL})$$

$$\frac{M =_{\beta\eta} N}{N =_{\beta\eta} M}\,\text{SYM} \qquad\qquad \frac{M =_{\beta\eta} P \qquad P =_{\beta\eta} N}{M =_{\beta\eta} N}\,\text{TRANS}$$

The η axiom has the proviso that the variable x cannot appear free in M.

Figure 4.2: A Proof System for $\beta\eta$-Convertibility of λ-Terms

We now illustrate that it is straightforward to specify $\beta\eta$-convertibility for untyped λ-terms. Figure 4.2 provides a formulation of $\beta\eta$-convertibility. $\to$ specifies the relation "reduces in one step to," while $=_{\beta\eta}$ is the reflexive, symmetric, transitive closure of $\to$. Our specification will reflect the formulation given by these proof rules. We first introduce the following three predicates.

```
type    redex   tm -> tm -> o.
type    red1    tm -> tm -> o.
type    conv    tm -> tm -> o.
```

The `redex` predicate is use to specify the first two axioms in Figure 4.2 which reduce top-level β and η redexes. These axioms are specified by the following definite clauses.

```
redex (app (abs M) N)  (M N).
redex (abs X\(app M X)) M.
```

Operationally, if a term unifies with the pattern in the left term in the first clause, then it is a β-redex whose reduced form appears on the right. The substitution of `N` for the bound variable in `M` is specified by application of λ-terms at the meta-level `(M N)`. The second clause illustrates an aspect of higher-order abstract syntax that we have not yet seen. Consider the instances of the variable `M` in the first argument. If such an instance were to contain a free occurrence of the variable `X`, the bound variable name in the above pattern would have to be changed to avoid capture. Thus, any instance of `M` will not contain any free occurrences of the bound variable in the above pattern, and so the proviso on this rule will be satisfied. The redex of a term matching this pattern is simply `M`. Note that the $\beta\eta$-long form of this clause is given above. It could also be written in its $\beta\eta$-normal form:

```
redex (abs (app M)) M.
```

The predicate `red1` relates two λ-terms if one arises from the other by replacing exactly one redex, and is defined by the following clauses.

```
red1 M N :- redex M N.
red1 (abs M) (abs N) :- pi X\ (red1 (M X) (N X)).
red1 (app M N) (app P N) :- red1 M P.
red1 (app M N) (app M P) :- red1 N P.
```

The first clause states that a term can be reduced in one step if it is a β or η redex. The second clause specifies the ξ rule for reducing inside the scope of an abstraction. Operationally, if a term matches with `(abs M)` on the left of the arrow, the clause must then descend through the abstraction. The universal quantifier (`pi`) is used to generate a new meta-level signature item, say `c`. β-reduction at the meta-level of `(M c)` performs the substitution of the new item for the object level abstracted variable in `M`. In effect, the new signature item plays the role of the name of the object level bound variable. If the subgoal succeeds, then `(N c)` is some term reachable in one step from `(M c)`. `N` is the abstraction not containing `c`. Declaratively, the clause reads: `(abs M)` reduces in one step to `(abs N)` if for arbitrary term `X`, `M` applied to `X` reduces in one step to `N` applied to `X`. The last two clauses specify the CONG1 and CONG2 rules for reducing an application. An application reduces in one step if either the term on the left or the term on the right reduces in one step.

Finally the reflexive, symmetric, and transitive closure of `red1` is specified as follows using the `conv` predicate.

```
conv M N :- red1 M N.
conv M M.
conv M N :- conv N M.
conv M N :- conv M P, conv P N.
```

Note that this specification of convertibility doesn't build proof terms. Such terms could be included as an extra argument to the `redex`, `red1` and `conv` predicates, and used to record the sequence of steps used to convert one term to the other.

Next, we consider associating simple types with terms, and illustrate how to specify a type assignment system which can be used to recognize the subclass of untyped terms that make up the simply-typed λ-terms. We introduce the new meta-type `ty`, and the infix arrow `-->`, used to construct functional types in the usual way, declared as follows.

```
type    '-->'   ty -> ty -> ty.
```

Note that both object-level types and terms will be represented by terms of the meta-language. To avoid confusion we will refer to types and terms of the meta-language as *meta-types* and *meta-terms*.

We introduce the infix predicate `#t` to represent the basic relation between a term and its type,

```
type    '#t'    tm -> ty -> o.
```

with the intended meaning that the atomic proposition `(M #t S)` holds when the term `M` has type `S`.

$$\frac{\begin{array}{c}(x:\tau_1)\\ M:\tau_2\end{array}}{\lambda x.M:\tau_1\to\tau_2}\ \text{ABS} \qquad\qquad \frac{M:\tau_1\to\tau_2 \qquad N:\tau_1}{MN:\tau_2}\ \text{APP}$$

The ABS rule has the proviso that the variable x cannot appear free in any assumption on which $M:\tau_2$ depends.

Figure 4.3: Type Assignment for the Simply-Typed λ-Calculus

Figure 4.3 contains two natural deduction style rules for type-checking and inference in the simply-typed λ-calculus. They are typical of type assignment systems such as that found in [HS86]. These rules are specified by the following clauses.

```
(abs M) #t (R --> S) :- pi X\ ((X #t R) => ((M X) #t S)).
(app M N) #t S :- M #t (R --> S), N #t R.
```

The first formula encodes the fact that an abstraction `(abs M)` has functional type `R --> S` if for arbitrary term `X`, `X` has type `R` implies that `(M X)` has type `S`. Operationally, this clause would introduce a new signature item, say `c` of meta-type `tm`, then extend the `#t` relation with the assumption that `c` has object-level type `R`, `(c #t R)`, and then attempt to prove that the β-normal form of `(M c)`, the result of replacing the bound variable of `M` with the name `c`, has type `S`. Note that both the proviso and the discharge of assumptions is handled as in the clauses for natural deduction in Section 3.2. The GENERIC search

operation introduces a new constant `c` that cannot occur in the current program. Thus it will not occur in the clauses that represent the current assumptions. The AUGMENT search operation adds the new assumption `(c #t R)` to the program. The second formula specifies the rule for application. An application `(app M N)` has type `S` if `M` has functional type `(R --> S)` and `N` has type `R`.

To provide a set of base types, we introduce meta-constants of type `ty`. We can then introduce object-level constants whose types are built up from these base types. To do so, we introduce meta-constants of type `tm` and include a type assignment clause for each constant. For example, if `i` is a type, and `f` is a function of type `(i --> i) --> i`, we include the following declarations and atomic clause.

```
type    i       ty.
type    f       tm.

f #t (i --> i) --> i.
```

Note that we could also use a formulation of the simply-typed λ-calculus where type information is explicitly attached to bound variables. To do this, we simply include the type of the bound variable as an extra argument to an abstraction:

```
type    abs     ty -> (tm -> tm) -> tm.
```

and modify the clause for type-checking abstractions as follows.

```
(abs R M) #t (R --> S) :- pi X\ ((X #t R) => ((M X) #t S)).
```

Such a representation that includes explicit type information inside terms will be adopted in the next chapter when we specify the LF type system.

4.3 Correctness of Specifications

In the examples in this and the previous chapter, terms, formulas, and proofs of an object logic were specified as simply typed λ-terms, while inference rules were specified as hohh formulas. In the specifications, there was always a clear correspondence between objects of the object language and objects in the meta-language. We now illustrate how to formalize this connection using the specification in the previous section of $\beta\eta$-convertibility in the untyped λ-calculus as an example. We will define a bijective mapping from untyped λ-terms to terms of the meta-language of type `tm`, and use this bijection to prove the correctness of our representation for untyped terms. We then prove the correctness of the program for $\beta\eta$-convertibility given by the clauses specifying the proof system of Figure 4.2 in the previous section. This specification is repeated in the `convert` module below.

This presentation serves as an illustration for establishing correctness results in general for specifications of object logics in hohh. We will see that the encoding of λ-terms can

```
module convert.

kind    tm      type.

type    abs     (tm -> tm) -> tm.
type    app     tm -> tm -> tm.

type    redex   tm -> tm -> o.
type    red1    tm -> tm -> o.
type    conv    tm -> tm -> o.

redex (app (abs M) N)  (M N).                          % beta
redex (abs X\(app M X)) M.                             % eta

red1 M N :- redex M N.
red1 (abs M) (abs N) :- pi X\ (red1 (M X) (N X)).      % xi
red1 (app M N) (app P N) :- red1 M P.                  % cong1
red1 (app M N) (app M P) :- red1 N P.                  % cong2

conv M N :- red1 M P.                                  % red
conv M M.                                              % refl
conv M N :- conv N M.                                  % sym
conv M N :- conv M P, conv P N.                        % trans
```

Module `convert`: $\beta\eta$-Convertibility in the Simply-Typed λ-Calculus

be adapted quite directly to encode formulas in any of the example logics we have presented. One reason for choosing this proof system as our example in this section is that in Chapter 5, we extend the encoding of λ-terms to an encoding on LF terms, and specify β-conversion for LF. The results presented here easily extend to this richer calculus.

4.3.1 Mappings Between Object Terms and Meta-Terms

We begin by formalizing the connection between terms of the object language and meta-terms of type `tm`. In the proofs in this section there are several notions of equality at both the object and meta-level. We use the $=$ symbol to denote syntactic equality between terms of both the object and meta-language. In either language, when we want to denote that two terms M and N are α-convertible, we write explicitly $M =_\alpha N$. We continue to use $=_{\beta\eta}$ for $\beta\eta$-convertibility in both languages, though we consider only meta-terms that are in $\beta\eta$-long normal form in this section. Finally, we use the symbol $\equiv$ to denote definitional equality. It is important to note that in contrast to previous sections, in this section we use first-order syntax for meta-terms since one goal of the proofs below is to show the correctness of the "higher-order" notion of syntax.

We assume the object language consists of a fixed set of constants and a countably infinite set of variables, and that terms are built up from these constants and variables in the

usual way. We define an encoding between untyped terms and terms in the meta-language of type `tm` in $\beta\eta$-long normal form. We will see that the domain of the encoding is actually the α-equivalence classes of untyped terms, while the codomain is the α-equivalence classes of meta-terms in $\beta\eta$-long normal form of type `tm`. To encode constants, we assume the existence of a mapping Φ, called a *constant mapping*, from the constants of the object language to some fixed set of constants of the meta-language of type `tm`. We assume that Φ is bijective, so that each constant is mapped to a distinct meta-constant. In all of the results that follow, we assume a fixed Φ.

We must also consider the encoding of variables. For this task, we need several definitions. We define a *variable encoding* ρ to be a bijective function whose domain is a set of untyped variables, and whose codomain is a set of variables of the meta-language of type `tm`. As we do for substitutions, we represent a variable encoding ρ as a set of pairs of the form $\langle x, \mathtt{X}\rangle$. Here `X` is the encoding of x. Like Φ, ρ is assumed to be bijective so that distinct variables will have distinct encodings. Updating of variable encodings is defined and denoted similarly to the updating of substitutions. In the definitions that follow, in updating a variable encoding, *e.g.*, $\langle x, \mathtt{X}\rangle + \rho$, it will always be the case that `X` is a new variable of the meta-language not already in $\mathrm{cod}(\rho)$, so that the updating operation does not alter the bijectivity of ρ. We say that a variable encoding ρ is *well-defined* on term M if all of the free variables in M are in $\mathrm{dom}(\rho)$. We say that a variable encoding is well-defined on a set of terms if it is well-defined on every member of the set. Since for any two terms M and N such that $M =_\alpha N$, M and N have the same free variables, it is easy to see that for any variable encoding ρ, ρ is well-defined of M iff it is well-defined on N. $\langle\!\langle M \rangle\!\rangle_\rho$ will denote the encoding of M with respect to a variable encoding ρ. When we write $\langle\!\langle M \rangle\!\rangle_\rho$, it will often be assumed without explicitly stating that ρ is well-defined on M. The full encoding is defined inductively in Figure 4.4. It is easy to see that for

$$
\begin{array}{rcl}
\langle\!\langle c \rangle\!\rangle_\rho & := & \Phi(c) \;\text{ for constant } c \in \mathrm{dom}(\Phi) \\
\langle\!\langle x \rangle\!\rangle_\rho & := & \rho(x) \;\text{ for variable } x \in \mathrm{dom}(\rho) \\
\langle\!\langle MN \rangle\!\rangle_\rho & := & \mathtt{(app}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho\mathtt{)} \\
\langle\!\langle \lambda x.M \rangle\!\rangle_\rho & := & \mathtt{(abs\ X\backslash}\ \langle\!\langle M \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}\mathtt{)} \quad \mathtt{X} \notin \mathrm{cod}(\rho)
\end{array}
$$

Figure 4.4: Encoding of Untyped Terms

any untyped term M, and variable encoding ρ well-defined on M, $\langle\!\langle M \rangle\!\rangle_\rho$ is a meta-term in $\beta\eta$-long form.

Given a set of variables $\mathcal{V}$, we denote the set of untyped terms whose free variables are in $\mathcal{V}$ as $UT(\mathcal{V})$. At the meta-level, given a set of variables $\mathcal{V}$ of type `tm`, we denote the set of terms of type `tm` in $\beta\eta$-long normal form built up from the the constants `app` and `abs`, the constants in $\mathrm{cod}(\Phi)$, and variables of type `tm` whose free variables are in $\mathcal{V}$ as $T(\mathcal{V})$. Note that for any $\mathcal{V}$, all of the variables and constants other than `app` and `abs` appearing

in a term in $T(\mathcal{V})$ have type `tm`.

Lemma 4.1 Let M be a term, and ρ a variable encoding that is well-defined on M. Then $\langle\!\langle M \rangle\!\rangle_\rho$ is a term in $T(\mathrm{cod}(\rho))$, *i.e.*, a term of type `tm` in $\beta\eta$-long form whose free variables are in $\mathrm{cod}(\rho)$.

Proof: The proof is by induction on the structure of M.

Base: If M is a constant c, c must be in $\mathrm{dom}(\Phi)$ and $\langle\!\langle c \rangle\!\rangle_\rho = \Phi(c)$, and by definition of a constant mapping $\Phi(c)$ is a constant of type `tm`.

If M is a variable x, x must be in $\mathrm{dom}(\rho)$. $\langle\!\langle x \rangle\!\rangle_\rho = \rho(x)$, and by definition of variable encodings, $\rho(x)$ is a variable of type `tm`.

Case: If M has the form NP, then $\langle\!\langle NP \rangle\!\rangle_\rho \equiv$ `(app` $\langle\!\langle N \rangle\!\rangle_\rho$ $\langle\!\langle P \rangle\!\rangle_\rho$`)`. By the induction hypothesis, $\langle\!\langle N \rangle\!\rangle_\rho$ and $\langle\!\langle P \rangle\!\rangle_\rho$ are terms of type `tm` in $\beta\eta$-long form whose free variables are in $\mathrm{cod}(\rho)$. Thus, so is `(app` $\langle\!\langle N \rangle\!\rangle_\rho$ $\langle\!\langle P \rangle\!\rangle_\rho$`)`.

Case: If M is a term of the form $\lambda x.N$, then $\langle\!\langle \lambda x.N \rangle\!\rangle_\rho \equiv$ `(abs X\` $\langle\!\langle N \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$`)` where `X` is a variable of type `tm` such that `X` $\notin \mathrm{cod}(\rho)$. Since the free variables of $\lambda x.N$ are in $\mathrm{dom}(\rho)$, the free variables of N are all in $\mathrm{dom}(\langle x, \mathtt{X}\rangle + \rho)$, so by the induction hypothesis, $\langle\!\langle N \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$ is a term of type `tm` in $\beta\eta$-long form whose free variables are in $\mathrm{cod}(\langle x, \mathtt{X}\rangle + \rho)$. Thus, `(abs X\` $\langle\!\langle N \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$`)` is a term of type `tm` in $\beta\eta$-long form whose free variables are in $\mathrm{cod}(\rho)$. ■

The following lemma will be used often in the arguments that follow. It is proved by a simple induction on the structure of terms.

Lemma 4.2 Let M be an untyped term, and ρ a variable encoding that is well-defined on M.

1. If $y \in \mathrm{dom}(\rho)$, then $\rho(y)$ is free in $\langle\!\langle M \rangle\!\rangle_\rho$ iff y is free in M.
2. If `Y` is a variable of type `tm` such that `Y` $\notin \mathrm{cod}(\rho)$, then `Y` is not free in $\langle\!\langle M \rangle\!\rangle_\rho$.

We now prove that the encoding on terms is well-defined, *i.e.*, the encodings of two α-convertible untyped terms are α-convertible meta-terms. This proof will require substitutions at the object and meta-level. To distinguish between the two, we call substitutions of the object language *object variable substitutions*, and substitutions of the meta-language *meta-variable substitutions*. For the special case when the domain of a meta-variable substitution contains only variables of some type `a`, we call the substitution an `a`-*substitution*. In this section, meta-variable substitutions will always be `tm`-substitutions.

Given an object variable substitution σ, if ρ_1 is a variable encoding well-defined on $\mathrm{dom}(\sigma)$ and ρ_2 is a variable encoding well-defined on $\mathrm{cod}(\sigma)$, then σ_{ρ_1,ρ_2} denotes the following `tm`-substitution.

$$\sigma_{\rho_1,\rho_2} = \{\langle \rho_1(x), \langle\!\langle P \rangle\!\rangle_{\rho_2} \rangle | \langle x, P\rangle \in \sigma\}$$

The well-definedness of the encoding on terms is a corollary of the following lemma.

Lemma 4.3 Let σ be an object variable substitution, and let M and N be terms such that $\sigma(M) =_\alpha N$. Let ρ_1 be a variable encoding well-defined on M and $\mathrm{dom}(\sigma)$, and ρ_2 a variable encoding well-defined on N and $\mathrm{cod}(\sigma)$ such that ρ_1 and ρ_2 agree on common domain elements that are free in both $\sigma(M)$ and N. Then

$$\sigma_{\rho_1,\rho_2}(\langle\!\langle M \rangle\!\rangle_{\rho_1}) =_\alpha \langle\!\langle N \rangle\!\rangle_{\rho_2}.$$

Proof: The proof is by induction on the structure of M.

Base: If M is a constant, then N is the same constant. $\sigma_{\rho_1,\rho_2}(\langle\!\langle M \rangle\!\rangle_{\rho_1}) \equiv \sigma_{\rho_1,\rho_2}(\Phi(M)) = \Phi(M)$ and $\langle\!\langle N \rangle\!\rangle_{\rho_2} \equiv \Phi(N)$. Since $M = N$, clearly $\Phi(M) = \Phi(N)$.

If M is a variable, then either $\langle M, N\rangle \in \sigma$ or $M \notin \mathrm{dom}(\sigma)$ and N is M. If $\langle M, N\rangle \in \sigma$ then $\langle \rho_1(M), \langle\!\langle N \rangle\!\rangle_{\rho_2}\rangle \in \sigma_{\rho_1,\rho_2}$ and thus

$$\sigma_{\rho_1,\rho_2}(\langle\!\langle M \rangle\!\rangle_{\rho_1}) \equiv \sigma_{\rho_1,\rho_2}(\rho_1(M)) =_\alpha \langle\!\langle N \rangle\!\rangle_{\rho_2}.$$

If $M \notin \mathrm{dom}(\sigma)$ and N is M, then ρ_1 and ρ_2 must map M to the same meta-variable. Thus $\rho_1(M) = \rho_2(N)$ and $\rho_1(M) \notin \mathrm{dom}(\sigma_{\rho_1,\rho_2})$. Thus,

$$\sigma_{\rho_1,\rho_2}(\langle\!\langle M \rangle\!\rangle_{\rho_1}) \equiv \sigma_{\rho_1,\rho_2}(\rho_1(M)) = \rho_1(M) = \rho_2(N) \equiv \langle\!\langle N \rangle\!\rangle_{\rho_2}.$$

Thus in either case $\sigma_{\rho_1,\rho_2}(\langle\!\langle M \rangle\!\rangle_{\rho_1}) =_\alpha \langle\!\langle N \rangle\!\rangle_{\rho_2}$.

Case: If M is P_1P_2, then N has the form Q_1Q_2 and $\sigma(P_1P_2) =_\alpha Q_1Q_2$. We must show that $\sigma_{\rho_1,\rho_2}(\langle\!\langle P_1P_2 \rangle\!\rangle_{\rho_1}) =_\alpha \langle\!\langle Q_1Q_2 \rangle\!\rangle_{\rho_2}$. By properties of substitution, and α-conversion, $\sigma(P_1P_2) =_\alpha \sigma(P_1)\sigma(P_2)$, and thus $\sigma(P_1) =_\alpha Q_1$ and $\sigma(P_2) =_\alpha Q_2$. Then by the induction hypothesis, $\sigma_{\rho_1,\rho_2}(\langle\!\langle P_1 \rangle\!\rangle_{\rho_1}) =_\alpha \langle\!\langle Q_1 \rangle\!\rangle_{\rho_2}$ and $\sigma_{\rho_1,\rho_2}(\langle\!\langle P_2 \rangle\!\rangle_{\rho_1}) =_\alpha \langle\!\langle Q_2 \rangle\!\rangle_{\rho_2}$. Thus

$$\begin{aligned}
&\sigma_{\rho_1,\rho_2}(\langle\!\langle P_1P_2 \rangle\!\rangle_{\rho_1}) \\
&\quad \equiv \sigma_{\rho_1,\rho_2}(\texttt{app}\ \langle\!\langle P_1 \rangle\!\rangle_{\rho_1}\ \langle\!\langle P_2 \rangle\!\rangle_{\rho_1}) && \text{by definition of the encoding} \\
&\quad =_\alpha (\texttt{app}\ \sigma_{\rho_1,\rho_2}(\langle\!\langle P_1 \rangle\!\rangle_{\rho_1})\ \sigma_{\rho_1,\rho_2}(\langle\!\langle P_2 \rangle\!\rangle_{\rho_1}) && \text{by substitution} \\
&\quad =_\alpha (\texttt{app}\ \langle\!\langle Q_1 \rangle\!\rangle_{\rho_2}\ \langle\!\langle Q_2 \rangle\!\rangle_{\rho_2}) && \text{by the induction hypothesis} \\
&\quad \equiv \langle\!\langle Q_1Q_2 \rangle\!\rangle_{\rho_2} && \text{by definition of the encoding.}
\end{aligned}$$

Case: If M has the form $\lambda x.P$, then N has the form $\lambda y.Q$ and $\sigma(\lambda x.P) =_\alpha \lambda y.Q$. We must show that $\sigma_{\rho_1,\rho_2}(\langle\!\langle \lambda x.P \rangle\!\rangle_{\rho_1}) =_\alpha \langle\!\langle \lambda y.Q \rangle\!\rangle_{\rho_2}$. Let $\texttt{X}$ and $\texttt{Y}$ be variables of type $\texttt{tm}$ such that $\texttt{X} \notin \mathrm{cod}(\rho_1)$ and $\texttt{Y} \notin \mathrm{cod}(\rho_2)$. Then by definition of the encoding,

$$\sigma_{\rho_1,\rho_2}(\langle\!\langle \lambda x.P \rangle\!\rangle_{\rho_1}) \equiv \sigma_{\rho_1,\rho_2}(\texttt{abs X\}\langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho_1}) \quad \text{and}$$

$$\langle\!\langle \lambda y.Q \rangle\!\rangle_{\rho_2} \equiv (\texttt{abs Y\}\langle\!\langle Q \rangle\!\rangle_{\langle y,\texttt{Y}\rangle+\rho_2}).$$

Since $\sigma(\lambda x.P) =_\alpha \lambda y.Q$, by Lemma 2.1 (1) it follows that $(\langle x, y\rangle + \sigma)(P) =_\alpha Q$. Then by the induction hypothesis,

$$(\langle x, y\rangle + \sigma)_{\langle x,\texttt{X}\rangle+\rho_1,\langle y,\texttt{Y}\rangle+\rho_2}(\langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho_1}) =_\alpha \langle\!\langle Q \rangle\!\rangle_{\langle y,\texttt{Y}\rangle+\rho_2}.$$

$(\langle x,y\rangle+\sigma)_{\langle x,\mathtt{X}\rangle+\rho_1,\langle y,\mathtt{Y}\rangle+\rho_2}$ differs from σ_{ρ_1,ρ_2} in that (a) the pair $\langle \mathtt{X},\mathtt{Y}\rangle$ appears in the former and not in the latter, (b) if $x \in \mathrm{dom}(\sigma)$, the pair $\langle \rho_1(x), \langle\!\langle (\sigma(x)) \rangle\!\rangle_{\rho_2}\rangle$ appears in the latter but not the former, and (c) if $y \in \mathrm{dom}(\rho_2)$, every occurrence of $\rho_2(y)$ in the terms in $\mathrm{cod}(\sigma_{\rho_1,\rho_2})$ is replaced by $\mathtt{Y}$ in $\mathrm{cod}((\langle x,y\rangle+\sigma)_{\langle x,\mathtt{X}\rangle+\rho_1,\langle y,\mathtt{Y}\rangle+\rho_2})$. Let σ_1 be the following subset of σ_{ρ_1,ρ_2},

$$\sigma_1 = \{\langle \rho_1(z), \langle\!\langle T \rangle\!\rangle_{\rho_2}\rangle | \langle z,T\rangle \in \sigma, y \text{ is free in } T, z \neq x\}$$

and σ_2 be the following subset of $(\langle x,y\rangle+\sigma)_{\langle x,\mathtt{X}\rangle+\rho_1,\langle y,\mathtt{Y}\rangle+\rho_2}$,

$$\sigma_2 = \{\langle \rho_1(z), \langle\!\langle T \rangle\!\rangle_{\langle y,\mathtt{Y}\rangle+\rho_2}\rangle | \langle z,T\rangle \in \sigma, y \text{ is free in } T, z \neq x\}.$$

Then $(\langle x,y\rangle+\sigma)_{\langle x,\mathtt{X}\rangle+\rho_1,\langle y,\mathtt{Y}\rangle+\rho_2}$ is the substitution $((\langle \mathtt{X},\mathtt{Y}\rangle+\sigma_{\rho_1,\rho_2})-\sigma_1)\cup\sigma_2$. Thus

$$(\langle x,y\rangle+\sigma)_{\langle x,\mathtt{X}\rangle+\rho_1,\langle y,\mathtt{Y}\rangle+\rho_2}(\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}) \quad \text{and} \quad (((\langle \mathtt{X},\mathtt{Y}\rangle+\sigma_{\rho_1,\rho_2})-\sigma_1)\cup\sigma_2)(\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1})$$

are the same term. Since y is not free in $\lambda y.Q$ and $\sigma(\lambda x.P) =_\alpha \lambda y.Q$, y is also not free in $\sigma(\lambda x.P)$. For $\langle z,T\rangle \in \sigma$ such that y is free in T, z is not free in $\lambda x.P$. Assume that z is different from x. Then z is not free in P, and thus, by Lemma 4.2 (1) $(\langle x,\mathtt{X}\rangle+\rho_1)(z)$ is not free in $\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}$. Since z is different from x, $\rho_1(z)$ is not free in $\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}$. Thus, the variables in $\mathrm{dom}(\sigma_1)$ and $\mathrm{dom}(\sigma_2)$ are not free in $\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}$. Then,

$$(((\langle \mathtt{X},\mathtt{Y}\rangle+\sigma_{\rho_1,\rho_2})-\sigma_1)\cup\sigma_2)(\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}) =_\alpha (\langle \mathtt{X},\mathtt{Y}\rangle+\sigma_{\rho_1,\rho_2})(\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}).$$

Thus we can conclude that

$$(\langle \mathtt{X},\mathtt{Y}\rangle+\sigma_{\rho_1,\rho_2})(\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}) =_\alpha \langle\!\langle Q \rangle\!\rangle_{\langle y,\mathtt{Y}\rangle+\rho_2}.$$

In order to apply Lemma 2.1 (2), we must show that if $\mathtt{X}$ is different from $\mathtt{Y}$, then $\mathtt{Y}$ is not free in $\sigma_{\rho_1,\rho_2}(\mathtt{X}\backslash\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1})$. Since $\mathtt{Y} \notin \mathrm{dom}(\rho_2)$, by Lemma 4.2 (2), $\mathtt{Y}$ does not appear free in the terms in $\mathrm{cod}(\sigma_{\rho_1,\rho_2})$. Thus for any term $\mathtt{T}$, there will be no new free occurrences of $\mathtt{Y}$ in $\sigma_{\rho_1,\rho_2}(\mathtt{T})$ that weren't already in $\mathtt{T}$. There are two subcases depending on whether or not $\mathtt{Y} \in \mathrm{cod}(\langle x,\mathtt{X}\rangle+\rho_1)$. If $\mathtt{Y} \notin \mathrm{cod}(\langle x,\mathtt{X}\rangle+\rho_1)$, by Lemma 4.2 (2), $\mathtt{Y}$ is not free in $\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}$. Thus $\mathtt{Y}$ is not free in $\sigma_{\rho_1,\rho_2}(\mathtt{X}\backslash\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1})$. On the other hand, if $\mathtt{Y} \in \mathrm{cod}(\langle x,\mathtt{X}\rangle+\rho_1)$, then there is a z different from x such that $\langle z,\mathtt{Y}\rangle \in \rho_1$. There are two subcases depending on whether or not $z \in \mathrm{dom}(\sigma)$. If $z \in \mathrm{dom}(\sigma)$, then $\mathtt{Y} \in \mathrm{dom}(\sigma_{\rho_1,\rho_2})$, and thus for any term $\mathtt{T}$, $\mathtt{Y}$ does not appear free in $\sigma_{\rho_1,\rho_2}(\mathtt{T})$. If $z \notin \mathrm{dom}(\sigma)$, we can show that z is not free in P. If it were, then since z is different from x, z is also free in $\lambda x.P$. Since $z \notin \mathrm{dom}(\sigma)$, z is free in $\sigma(\lambda x.P)$. Since $\sigma(\lambda x.P) =_\alpha \lambda y.Q$, z is free in $\lambda y.Q$. Since ρ_2 is well-defined on $\lambda y.Q$, $z \in \mathrm{dom}(\rho_2)$. Since ρ_1 and ρ_2 agree on common domain elements that are free in $\sigma(\lambda x.P)$ and $\lambda y.Q$, $\rho_2(z) = \mathtt{Y}$, a contradiction since $\mathtt{Y} \notin \mathrm{cod}(\rho_2)$. Thus z is not free in P. Thus, by Lemma 4.2 (2), $\mathtt{Y}$ is not free in $\langle\!\langle P \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}$. Thus $\mathtt{Y}$ is not free

in $\sigma_{\rho_1,\rho_2}(\mathtt{X}\backslash\langle\!\langle P\rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1})$. We have shown that in any case, when X is different from Y, Y is not free in $\sigma_{\rho_1,\rho_2}(\mathtt{X}\backslash\langle\!\langle P\rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1})$. So, by Lemma 2.1 (2),

$$\sigma_{\rho_1,\rho_2}(\mathtt{X}\backslash\langle\!\langle P\rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}) =_\alpha \mathtt{Y}\backslash\langle\!\langle Q\rangle\!\rangle_{\langle y,\mathtt{Y}\rangle+\rho_2}.$$

Thus

$$\sigma_{\rho_1,\rho_2}(\mathtt{abs\ X}\backslash\langle\!\langle P\rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho_1}) =_\alpha (\mathtt{abs\ Y}\backslash\langle\!\langle Q\rangle\!\rangle_{\langle y,\mathtt{Y}\rangle+\rho_2})$$

and hence by the definition of the encoding,

$$\sigma_{\rho_1,\rho_2}(\langle\!\langle \lambda x.P\rangle\!\rangle_{\rho_1}) =_\alpha \langle\!\langle \lambda y.Q\rangle\!\rangle_{\rho_2}.$$

■

Corollary 4.4 (Well-Definedness of Encoding for Untyped Terms)
Let M and N be two terms such that $M =_\alpha N$. Let ρ_1 and ρ_2 be two variable encodings that are well-defined on M and N, and that agree on the free variables of M and N. Then $\langle\!\langle M\rangle\!\rangle_{\rho_1} =_\alpha \langle\!\langle N\rangle\!\rangle_{\rho_2}$.

Corollary 4.5 (Substitution Commutes with Encoding)
Let M and P be terms, and x a variable. Let ρ_1 be a variable encoding well-defined on M and x, and ρ_2 a variable encoding well-defined on $[P/x]M$ and P such that ρ_1 and ρ_2 agree on common domain elements free in $[P/x]M$. Then

$$[\langle\!\langle P\rangle\!\rangle_{\rho_2}/\langle\!\langle x\rangle\!\rangle_{\rho_1}]\langle\!\langle M\rangle\!\rangle_{\rho_1} =_\alpha \langle\!\langle [P/x]M\rangle\!\rangle_{\rho_2}.$$

Proof: The proof follows from Lemma 4.3 with $\sigma = \{\langle x, P\rangle\}$ and $N = [P/x]M$. ■

We also define the reverse operation, which we call a *decoding*, from terms in $\beta\eta$-long form of type tm whose free variables have type tm to untyped terms. As was the case for the encoding, the decoding is up to the $=_\alpha$ relation on both the object and meta-language. To decode constants, we will use the inverse of the constant mapping Φ. To decode variables, we will use a *variable decoding* where the domain is a set of meta-variables of type tm and the codomain is a set of variables of the object language. We will use a bar, *e.g.*, $\bar{\rho}$, to denote variable decodings. $\bar{\rho}$ is well-defined on M if all of the free variables in M are in $\mathrm{dom}(\bar{\rho})$. $\bar{\rho}$ is well-defined on a set of terms if it is well-defined on every member of the set. The decoding of M with respect to variable decoding $\bar{\rho}$ is denoted by $\|\mathtt{M}\|_{\bar{\rho}}$. The full decoding is defined in Figure 4.5.

Given a tm-substitution σ, if $\bar{\rho}_1$ is a variable decoding well-defined on $\mathrm{dom}(\sigma)$ and $\bar{\rho}_2$ is a variable decoding well-defined on $\mathrm{cod}(\sigma)$, then $\sigma_{\bar{\rho}_1,\bar{\rho}_2}$ denotes the following object variable substitution.

$$\sigma_{\bar{\rho}_1,\bar{\rho}_2} = \{\langle \bar{\rho}_1(\mathtt{X}), \|\mathtt{P}\|_{\bar{\rho}_2}\rangle | \langle \mathtt{X}, \mathtt{P}\rangle \in \sigma\}$$

We now state the analogues of Lemmas 4.1, 4.2, 4.3 and Corollaries 4.4 and 4.5 for the decoding. The proofs are analogous to the corresponding proofs for the encoding.

$$
\begin{array}{rcl}
\|\mathtt{c}\|_{\bar\rho} & := & \Phi^{-1}(\mathtt{c}) \ \text{ for } \mathtt{c} \in \mathrm{cod}(\Phi) \\
\|\mathtt{X}\|_{\bar\rho} & := & \bar\rho(\mathtt{X}) \ \text{ for } \mathtt{X} \in \mathrm{dom}(\bar\rho) \\
\|\mathtt{app\ M\ N}\|_{\bar\rho} & := & \|\mathtt{M}\|_{\bar\rho}\ \|\mathtt{N}\|_{\bar\rho} \\
\|\mathtt{abs\ P}\|_{\bar\rho} & := & (\lambda x.\|\mathtt{M}\|_{\langle \mathtt{X},x\rangle+\bar\rho}) \\
 & & \text{where } \mathtt{X} \text{ is the bound variable in } \mathtt{P},\ \mathtt{M} \text{ is the body, and } x \notin \mathrm{cod}(\bar\rho)
\end{array}
$$

Figure 4.5: Decoding of Untyped Terms

Lemma 4.6 Let M be a term of type tm in $\beta\eta$-long form, and $\bar\rho$ a variable decoding that is well-defined on M. Then $\|\mathtt{M}\|_{\bar\rho}$ is a term in $UT(\mathrm{cod}(\bar\rho))$, *i.e.*, an untyped term whose free variables are in $\mathrm{cod}(\bar\rho)$.

Lemma 4.7 Let M be a term of type tm in $\beta\eta$-long form, and let $\bar\rho$ be a variable decoding that is well-defined on M.

1. If $\mathtt{Y} \in \mathrm{dom}(\bar\rho)$, then $\bar\rho(\mathtt{Y})$ is free in $\|\mathtt{M}\|_{\bar\rho}$ iff Y is free in M.
2. If y is a variable such that $y \notin \mathrm{cod}(\bar\rho)$, then y is not free in $\|\mathtt{M}\|_{\rho}$.

Lemma 4.8 Let σ be a tm-substititution, and let M and N be terms of type tm in $\beta\eta$-long form such that $\sigma(\mathtt{M}) =_\alpha \mathtt{N}$. Let $\bar\rho_1$ be a variable decoding well-defined on M and $\mathrm{dom}(\sigma)$ and $\bar\rho_2$ a variable decoding well-defined on N and $\mathrm{cd}(\sigma)$ such that $\bar\rho_1$ and $\bar\rho_2$ agree on common domain elements that are free in both $\sigma(\mathtt{M})$ and N. Then

$$\sigma_{\bar\rho_1,\bar\rho_2}(\|\mathtt{M}\|_{\bar\rho_1}) =_\alpha \|\mathtt{N}\|_{\bar\rho_2}.$$

Corollary 4.9 (Well-Definedness of Decoding for Untyped Terms)
Let M and N be terms of type tm in $\beta\eta$-long form such that $\mathtt{M} =_\alpha \mathtt{N}$. Let $\bar\rho_1$ and $\bar\rho_2$ be two variable decodings that are well-defined on M and N, and that agree on the free variables of M and N. Then $\|\mathtt{M}\|_{\bar\rho_1} =_\alpha \|\mathtt{N}\|_{\bar\rho_2}$.

Corollary 4.10 (Substitution Commutes with Decoding)
Let M and P be terms of type tm in $\beta\eta$-long form and X a variable of type tm. Let $\bar\rho_1$ be a variable decoding well-defined on M and X, and $\bar\rho_2$ a variable decoding well-defined on [P/X]M and P such that $\bar\rho_1$ and $\bar\rho_2$ agree on common domain elements free in [P/X]M. Then

$$[\|\mathtt{P}\|_{\bar\rho_2}/\|\mathtt{X}\|_{\bar\rho_1}]\|\mathtt{M}\|_{\bar\rho_1} =_\alpha \|[\mathtt{P}/\mathtt{X}]\mathtt{M}\|_{\bar\rho_2}.$$

Note that given a variable encoding ρ, $UT(\mathrm{dom}(\rho))$ is exactly the set of untyped terms over which ρ is well-defined. Similarly, given a variable decoding $\bar{\rho}$, $T(\mathrm{dom}(\bar{\rho}))$ is exactly the set of meta-terms over which $\bar{\rho}$ is well-defined.

Theorem 4.11 (Correctness of Encoding and Decoding of Untyped Terms)
Let ρ be a variable encoding. The encoding $\langle\!\langle \rangle\!\rangle_\rho$ is a bijection from the α-equivalence classes of $UT(\mathrm{dom}(\rho))$ to the α-equivalence classes of $T(\mathrm{cod}(\rho))$, *i.e.*, a bijective mapping from sets of untyped terms with free variables in $\mathrm{dom}(\rho)$, to sets of simply-typed terms of type `tm` in $\beta\eta$-long form with free variables in $\mathrm{cod}(\rho)$. Furthermore, the decoding $\|\|_{\rho^{-1}}$ is the inverse of $\langle\!\langle \rangle\!\rangle_\rho$.

Proof: We proceed by induction on the structure of terms. By Lemma 4.1, we know that $\langle\!\langle \rangle\!\rangle_\rho$ maps terms in $UT(\mathrm{dom}(\rho))$ to meta-terms in $T(\mathrm{cod}(\rho))$. By Lemma 4.6, we know that $\|\|_{\rho^{-1}}$ maps meta-terms in $T(\mathrm{cod}(\rho))$ to terms in $UT(\mathrm{dom}(\rho))$. We show that for an untyped term M whose free variables are in $\mathrm{dom}(\rho)$, $\|\langle\!\langle M \rangle\!\rangle_\rho\|_{\rho^{-1}} =_\alpha M$. The proof that for a meta-term `M` of type `tm` in $\beta\eta$-long form whose free variables are in $\mathrm{cod}(\rho)$, $\langle\!\langle \|\mathtt{M}\|_{\rho^{-1}} \rangle\!\rangle_\rho =_\alpha \mathtt{M}$ is analogous.

Base: If M is a constant, then $\|\langle\!\langle M \rangle\!\rangle_\rho\|_{\rho^{-1}} \equiv \|\Phi(M)\|_{\rho^{-1}} \equiv \Phi^{-1}(\Phi(M)) = M$.
If M is a variable, then $\|\langle\!\langle M \rangle\!\rangle_\rho\|_{\rho^{-1}} \equiv \|\rho(M)\|_{\rho^{-1}} \equiv \rho^{-1}(\rho(M)) = M$.

Case: If M has the form PQ, then

$$\|\langle\!\langle PQ \rangle\!\rangle_\rho\|_{\rho^{-1}} \equiv \|(\mathtt{app}\ \langle\!\langle P \rangle\!\rangle_\rho\ \langle\!\langle Q \rangle\!\rangle_\rho)\|_{\rho^{-1}} \equiv \|\langle\!\langle P \rangle\!\rangle_\rho\|_{\rho^{-1}} \|\langle\!\langle Q \rangle\!\rangle_\rho\|_{\rho^{-1}}.$$

By the induction hypothesis, $\|\langle\!\langle P \rangle\!\rangle_\rho\|_{\rho^{-1}} =_\alpha P$, and $\|\langle\!\langle Q \rangle\!\rangle_\rho\|_{\rho^{-1}} =_\alpha Q$. So the latter expression above is α-equivalent to PQ.

Case: If M has the form $\lambda x.N$, assume $x \notin \mathrm{dom}(\rho)$, otherwise rename the bound variable in $\lambda x.N$. Such renaming is allowed by Corollary 4.4. Let `X` be a variable of type `tm` such that $\mathtt{X} \notin \mathrm{cod}(\rho)$.

$$\|\langle\!\langle \lambda x.N \rangle\!\rangle_\rho\|_{\rho^{-1}} \equiv \|(\mathtt{abs}\ \mathtt{X}\backslash\langle\!\langle N \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho})\|_{\rho^{-1}} \equiv \lambda x.\|\langle\!\langle N \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}\|_{\langle \mathtt{X},x\rangle+\rho^{-1}}.$$

By the induction hypothesis, $\|\langle\!\langle N \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}\|_{\langle \mathtt{X},x\rangle+\rho^{-1}} =_\alpha N$. Thus

$$\lambda x.\|\langle\!\langle N \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}\|_{\langle \mathtt{X},x\rangle+\rho^{-1}} = \lambda x.N.$$

■

This result establishes the correctness of identifying abstraction at the object-level with abstraction at the meta-level. The same result can easily be established for our representation of formulas which identifies variables bound by quantification in formulas with variables bound by λ-abstraction at the meta-level. For example, the two clauses below illustrate the encoding of universally and existentially quantified formulas in first-order logic.

$\langle\!\langle \forall x A \rangle\!\rangle_\rho$:= `(forall X\` $\langle\!\langle A \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$ `)` $\mathtt{X} \notin \mathrm{cod}(\rho)$

$\langle\!\langle \exists x A \rangle\!\rangle_\rho$:= `(exists X\` $\langle\!\langle A \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$ `)` $\mathtt{X} \notin \mathrm{cod}(\rho)$

In such an encoding, the codomain of variable encodings would be meta-variables of type `i`. Of course, a decoding can be defined similarly. Such an encoding for first-order terms and formulas is essentially the same as the encoding of first-order terms and formulas in LF given in [HHP89]. There, the encoding of terms and formulas is defined within the sublanguage of LF that corresponds to the simply-typed λ-calculus. As a result, the above proof is similar to the proofs for Adequacy of Syntax, I and II where $\beta\eta$-long forms in this presentation correspond to LF canonical forms. (See Chapter 5 for a definition of canonical.)

4.3.2 Correctness of the Specification of $\beta\eta$-Convertibility

We are now ready to prove the correctness of the `conv` program in the `convert` module. Here, we establish a correspondence between provable atomic goals of the form `(conv M N)` and $\beta\eta$-convertible λ-terms. We divide the result into two parts, showing the correspondence in each direction. In this subsection, since Theorem 4.11 established the fact that the domain and codomain of the encoding and decoding are α-equivalence classes of terms, we will no longer distinguish between two α-convertible terms. Now, when we write $M = N$ for object level terms M and N, we will assume that M is α-convertible to N, and similarly for terms at the meta-level.

Let ρ be a variable encoding, and let Ξ be a proof of $M =_{\beta\eta} N$. We say that ρ is *well-defined* on Ξ if for every node of the form $P =_{\beta\eta} Q$ in Ξ, ρ is well-defined on P and Q. In the results below, we take Σ_0 to be the set of declarations of constants and their types that appears in the `convert` module. Let $\mathcal{V}$ be a set of variables of type `tm`. The correspondence between the subset of $H(\Sigma_0 \cup \mathcal{V})$ of terms of type `tm` and the set $T(\mathcal{V})$ can be described as follows: $H(\Sigma_0 \cup \mathcal{V})$ contains $\beta\eta$-equivalence classes of terms of type `tm` while $T(\mathcal{V})$ contains the $\beta\eta$-long representatives from each class. This correspondence is important in the proofs below.

Theorem 4.12 (Correctness IA)

Let Ξ be a proof of $M =_{\beta\eta} N$ and let ρ be a variable encoding well-defined on Ξ. Let Σ be the signature $\Sigma_0 \cup \mathrm{cod}(\rho)$. Then the following judgment holds.

$$\Sigma; \mathtt{convert} \vdash_I \mathtt{conv}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho$$

Proof: The proof is by induction on the height of a proof of $M =_{\beta\eta} N$ in the proof system of Figure 4.2. We prove simultaneously that if Ξ is a proof of $M \to N$ and ρ a variable encoding well-defined on M and N, then the following judgment holds.

$$\Sigma; \mathtt{convert} \vdash_I \mathtt{red1}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho$$

Case: (REFL) $M =_{\beta\eta} N$ is provable in one step by the REFL axiom. Then N is M, and $\langle\!\langle M \rangle\!\rangle_\rho = \langle\!\langle N \rangle\!\rangle_\rho$. Thus $\texttt{(conv}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho\texttt{)}$ is an instance of the atomic clause specifying the REFL axiom, so clearly the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho$$

Case: (TRANS) The last step in a proof of $M =_{\beta\eta} N$ is an application of the TRANS rule whose premises are $M =_{\beta\eta} P$ and $P =_{\beta\eta} N$. We must show that:

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho.$$

Since ρ is well-defined on Ξ, ρ is well-defined on P. Thus $\langle\!\langle P \rangle\!\rangle_\rho$ is in $T(\mathrm{cod}(\rho))$, and hence is also in $H(\Sigma)$. The above judgment holds if by BACKCHAIN on the clause specifying the TRANS rule, the following judgment holds,

$$\Sigma; \texttt{convert} \vdash_I \texttt{(conv}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle P \rangle\!\rangle_\rho\texttt{)}, \texttt{(conv}\ \langle\!\langle P \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho\texttt{)}$$

and by the AND search operation, the following judgments hold.

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle P \rangle\!\rangle_\rho \qquad\qquad \Sigma; \texttt{convert} \vdash_I \texttt{conv}\ \langle\!\langle P \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho$$

These judgments hold by the `conv` induction hypothesis.

Case: (SYM) The last step in a proof of $M =_{\beta\eta} N$ is an application of the SYM rule whose premise is $N =_{\beta\eta} M$. We must show that:

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho.$$

This judgment holds if by BACKCHAIN on the clause specifying the SYM rule, the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv}\ \langle\!\langle N \rangle\!\rangle_\rho\ \langle\!\langle M \rangle\!\rangle_\rho$$

This judgment holds by the `conv` induction hypothesis.

Case: (RED) The last step in a proof of $M =_{\beta\eta} N$ is an application of the RED rule whose premise is $M \to N$. We must show that

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho$$

This judgment holds if by BACKCHAIN on the clause specifying the RED rule, the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{red1}\ \langle\!\langle M \rangle\!\rangle_\rho\ \langle\!\langle N \rangle\!\rangle_\rho$$

This judgment holds by the `red1` induction hypothesis.

Case: (β) $M \to N$ is provable in one step by the β axiom. Then M has the form $(\lambda x.P)Q$, N is $[Q/x]P$, and $(\lambda x.P)Q \to [Q/x]P$. We must show that:

$$\Sigma; \texttt{convert} \vdash_I \texttt{red1}\ \langle\!\langle (\lambda x.P)Q \rangle\!\rangle_\rho\ \langle\!\langle [Q/x]P \rangle\!\rangle_\rho.$$

This judgment holds if by BACKCHAIN on the first `red1` clause, the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{redex}\ \langle\!\langle (\lambda x.P)Q \rangle\!\rangle_\rho\ \langle\!\langle [Q/x]P \rangle\!\rangle_\rho$$

Let `X` be a variable such that $\texttt{X} \notin \mathrm{cod}(\rho)$. By the definition of the encoding

$$\langle\!\langle (\lambda x.P)Q \rangle\!\rangle_\rho \equiv \texttt{(app (abs X\ } \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{)} \ \langle\!\langle Q \rangle\!\rangle_\rho \texttt{)}.$$

The variable encodings ρ and $\langle x, \texttt{X}\rangle + \rho$ agree on common domain elements that are free in $[Q/x]P$. Thus, by Corollary 4.5,

$$\langle\!\langle [Q/x]P \rangle\!\rangle_\rho = [\langle\!\langle Q \rangle\!\rangle_\rho / \texttt{X}] \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho}.$$

By β-conversion at the meta-level,

$$[\langle\!\langle Q \rangle\!\rangle_\rho / \texttt{X}] \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} =_{\beta\eta} \texttt{(X\} \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho}\ \langle\!\langle Q \rangle\!\rangle_\rho \texttt{)}.$$

Thus, the above judgment is equivalent to:

$$\Sigma; \texttt{convert} \vdash_I \texttt{redex (app (abs X\ } \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{)}\ \langle\!\langle Q \rangle\!\rangle_\rho \texttt{) (X\} \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho}\ \langle\!\langle Q \rangle\!\rangle_\rho \texttt{)}.$$

This judgment holds since it is an instance of the atomic clause specifying the β axiom.

Case: (η) $M \to N$ is provable in one step by the η axiom. Then M has the form $\lambda x.Px$, N is P, x is a variable that does not appear free in P, and $\lambda x.Px \to P$. We must show that:

$$\Sigma; \texttt{convert} \vdash_I \texttt{red1}\ \langle\!\langle \lambda x.Px \rangle\!\rangle_\rho\ \langle\!\langle P \rangle\!\rangle_\rho.$$

This judgment holds if by BACKCHAIN on the first `red1` clause, the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{redex}\ \langle\!\langle \lambda x.Px \rangle\!\rangle_\rho\ \langle\!\langle P \rangle\!\rangle_\rho$$

Let `X` be a variable such that $\texttt{X} \notin \mathrm{cod}(\rho)$. By the definition of the encoding

$$\langle\!\langle \lambda x.Px \rangle\!\rangle_\rho \equiv \texttt{(abs X\(app } \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho}\ \texttt{X))}.$$

Since x does not appear free in P, it follows by Corollary 4.4 that $\langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} = \langle\!\langle P \rangle\!\rangle_\rho$. Thus the above judgment is equivalent to the following judgment.

$$\Sigma; \texttt{convert} \vdash_I \texttt{redex (abs X\(app } \langle\!\langle P \rangle\!\rangle_\rho\ \texttt{X))}\ \langle\!\langle P \rangle\!\rangle_\rho$$

Since $\texttt{X} \notin \mathrm{cod}(\rho)$, by Lemma 4.2 (2), `X` does not appear free in $\langle\!\langle P \rangle\!\rangle_\rho$. Thus the above subgoal is an instance of the atomic clause specifying the η axiom.

Case: (CONG1) The last step in a proof of $M \to N$ is an application of the CONG1 rule. Then M has the form PT, N has the form QT, the conclusion of the rule is $PT \to QT$ and the premise is $P \to Q$. We must show that:

$$\Sigma; \texttt{convert} \vdash_I \texttt{red1}\ \langle\!\langle PT \rangle\!\rangle_\rho\ \langle\!\langle QT \rangle\!\rangle_\rho.$$

By the definition of the encoding,

$$\langle\!\langle PT \rangle\!\rangle_\rho \equiv \texttt{(app } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle T \rangle\!\rangle_\rho \texttt{)} \quad \text{and} \quad \langle\!\langle QT \rangle\!\rangle_\rho \equiv \texttt{(app } \langle\!\langle Q \rangle\!\rangle_\rho \ \langle\!\langle T \rangle\!\rangle_\rho \texttt{)}.$$

Thus, the above judgment is equivalent to:

$$\Sigma; \texttt{convert} \vdash_I \texttt{red1 (app } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle T \rangle\!\rangle_\rho \texttt{) (app } \langle\!\langle Q \rangle\!\rangle_\rho \ \langle\!\langle T \rangle\!\rangle_\rho \texttt{)}.$$

This judgment holds if by BACKCHAIN on the clause specifying the CONG1 rule, the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{red1} \ \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho$$

This judgment holds by the `red1` induction hypothesis.

Case: (CONG2) This case is similar to the above case for CONG1.

Case: (ξ) The last step in a proof of $M \to N$ is an application of the ξ rule. Then M has the form $\lambda x.P$, N has the form $\lambda x.Q$, the conclusion of the rule is $\lambda x.P \to \lambda x.Q$ and the premise is $P \to Q$. We must show that:

$$\Sigma; \texttt{convert} \vdash_I \texttt{red1} \ \langle\!\langle \lambda x.P \rangle\!\rangle_\rho \ \langle\!\langle \lambda x.Q \rangle\!\rangle_\rho.$$

Let `X` be a variable such that $\texttt{X} \notin \mathrm{cod}(\rho)$. $\langle x, \texttt{X} \rangle + \rho$ is well-defined on P and Q. By the definition of the encoding,

$$\langle\!\langle \lambda x.P \rangle\!\rangle_\rho \equiv \texttt{(abs X\ } \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{)} \quad \text{and} \quad \langle\!\langle \lambda x.Q \rangle\!\rangle_\rho \equiv \texttt{(abs X\ } \langle\!\langle Q \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{)}.$$

Thus, the above judgment is equivalent to:

$$\Sigma; \texttt{convert} \vdash_I \texttt{red1 (abs X\ } \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{) (abs X\ } \langle\!\langle Q \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{)}.$$

This judgment holds if by BACKCHAIN on the clause specifying the ξ rule, the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{pi X\(red1 (X\ } \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{X) (X\ } \langle\!\langle Q \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{X))}$$

`X` does not appear in Σ, so this judgment holds if by AUGMENT the following judgment holds.

$$\Sigma \cup \{\texttt{X} : \texttt{tm}\}; \texttt{convert} \vdash_I \texttt{red1 (X\ } \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{X) (X\ } \langle\!\langle Q \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{X)}$$

By β-conversion at the meta-level,

$$\texttt{(X\ } \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{X)} =_{\beta\eta} \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \quad \text{and} \quad \texttt{(X\ } \langle\!\langle Q \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \texttt{X)} =_{\beta\eta} \langle\!\langle Q \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho}.$$

Thus the above judgment is equivalent to the following judgment.

$$\Sigma \cup \{\texttt{X} : \texttt{tm}\}; \texttt{convert} \vdash_I \texttt{red1} \ \langle\!\langle P \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \ \langle\!\langle Q \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho}$$

This judgment holds by the `red1` induction hypothesis. ■

Note that in the above result, we had to start with a particular proof and then use a variable encoding ρ that was well-defined on the proof. This was necessary to insure that in each application of the (TRANS) rule, the term P that appears in the premises of the rule but not the conclusion is in $UT(\mathrm{dom}(\rho))$. Then it can be encoded with respect to ρ and its encoding will be in $T(\mathrm{cod}(\rho))$ and hence in $H(\Sigma)$. We can remove this dependency on proofs by making use of the Church-Rosser property for untyped terms. This property states that for any two terms such that $M =_{\beta\eta} N$, there is a term P such that $M \rightarrow^* P$ and $N \rightarrow^* P$ where $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$. Based on this property, we can prove the following lemma and stronger correctness result.

Lemma 4.13 Let ρ be a variable encoding and M and N terms in $UT(\mathrm{dom}(\rho))$. Let Σ be the signature $\Sigma_0 \cup \mathrm{cod}(\rho)$.

1. If $M \rightarrow^* N$, then $\Sigma; \texttt{convert} \vdash_I \texttt{conv} \langle\!\langle M \rangle\!\rangle_\rho \langle\!\langle N \rangle\!\rangle_\rho$ holds.
2. If $M \rightarrow N$, then $\Sigma; \texttt{convert} \vdash_I \texttt{red1} \langle\!\langle M \rangle\!\rangle_\rho \langle\!\langle N \rangle\!\rangle_\rho$ holds.

Proof: If $M \rightarrow^* N$, we can construct a proof of $M =_{\beta\eta} N$ such that there are no applications of (SYM) and for every application of the (TRANS) rule with conclusion $M' =_{\beta\eta} N'$ and premises $M' =_{\beta\eta} P'$ and $P' =_{\beta\eta} N'$, it is the case that $M' \rightarrow^* P'$ and $P' \rightarrow^* N'$. We prove (1) and (2) by simultaneous induction on a proof of $M =_{\beta\eta} N$ of this form, and on any proof of $M \rightarrow N$. All cases are similar to Theorem 4.12 except the case when the last step in the proof is an application of the (TRANS) rule. For this case, we must show that the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv} \langle\!\langle M \rangle\!\rangle_\rho \langle\!\langle N \rangle\!\rangle_\rho$$

In this case, if $M =_{\beta\eta} P$ and $P =_{\beta\eta} N$ are the premises of this application of (TRANS), then we know that $M \rightarrow^* P$ and $P \rightarrow^* N$. For any terms Q_1 and Q_2, if $Q_1 \rightarrow Q_2$ (or $Q_1 \rightarrow^* Q_2$), then the free variables in Q_2 are a subset of the free variables in Q_1. Thus the free variables in P are a subset of the free variables in M, so ρ is well-defined on P, and $\langle\!\langle P \rangle\!\rangle_\rho$ is in $T(\mathrm{cod}(\rho))$ and hence in $H(\Sigma)$. Thus the above judgment holds if by BACKCHAIN on the clause specifying the (TRANS) rule followed by AND search, the following judgments hold.

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv} \langle\!\langle M \rangle\!\rangle_\rho \langle\!\langle P \rangle\!\rangle_\rho \qquad\qquad \Sigma; \texttt{convert} \vdash_I \texttt{conv} \langle\!\langle P \rangle\!\rangle_\rho \langle\!\langle N \rangle\!\rangle_\rho$$

These judgments hold by the induction hypothesis for (1). ■

Corollary 4.14 (Correctness IB)

Let ρ be a variable encoding and M and N terms in $UT(\mathrm{dom}(\rho))$. Let Σ be the signature $\Sigma_0 \cup \mathrm{cod}(\rho)$. If $M =_{\beta\eta} N$, then the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv} \langle\!\langle M \rangle\!\rangle_\rho \langle\!\langle N \rangle\!\rangle_\rho$$

Proof: By the Church-Rosser property, there is a term P such that $M \rightarrow^* P$ and $N \rightarrow^* P$. We must show that the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv} \langle\!\langle M \rangle\!\rangle_\rho \langle\!\langle N \rangle\!\rangle_\rho$$

Since the free variables in P are a subset of the free variables in M and N, ρ is well-defined on P and $\langle\!\langle P \rangle\!\rangle_\rho$ is in $T(\mathrm{cod}(\rho))$ and hence in $H(\Sigma)$. Thus the above judgment holds if by BACKCHAIN on the clause specifying the (TRANS) rule followed by AND search, the following judgments hold.

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv} \langle\!\langle M \rangle\!\rangle_\rho \langle\!\langle P \rangle\!\rangle_\rho \qquad \Sigma; \texttt{convert} \vdash_I \texttt{conv} \langle\!\langle P \rangle\!\rangle_\rho \langle\!\langle N \rangle\!\rangle_\rho$$

The first judgment holds by Lemma 4.13. The second judgment holds if by BACKCHAIN on the clause specifying the (SYM) rule, the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv} \langle\!\langle N \rangle\!\rangle_\rho \langle\!\langle P \rangle\!\rangle_\rho$$

This judgment also holds by Lemma 4.13. ■

Theorem 4.15 (Correctness II)

Let $\bar{\rho}$ be a variable decoding and `M` and `N` terms in $T(\mathrm{dom}(\bar{\rho}))$. Let Σ be the signature $\Sigma_0 \cup \mathrm{dom}(\bar{\rho})$. If the judgment $\Sigma; \texttt{convert} \vdash_I \texttt{conv M N}$ holds, then $\|\texttt{M}\|_{\bar{\rho}} =_{\beta\eta} \|\texttt{N}\|_{\bar{\rho}}$.

Proof: The proof is by induction on the height of a proof of $\Sigma; \texttt{convert} \vdash_I \texttt{conv M N}$. We prove simultaneously that if the judgment $\Sigma; \texttt{convert} \vdash_I \texttt{red1 M N}$ holds, then $\|\texttt{M}\|_{\bar{\rho}} \rightarrow \|\texttt{N}\|_{\bar{\rho}}$.

Case: (REFL) The last step in a proof of `(conv M N)` is a BACKCHAIN on the clause specifying the REFL axiom. Then `N` is `M`, and $\|\texttt{M}\|_{\bar{\rho}} = \|\texttt{N}\|_{\bar{\rho}}$. Thus, by the REFL axiom, $\|\texttt{M}\|_{\bar{\rho}} =_{\beta\eta} \|\texttt{N}\|_{\bar{\rho}}$.

Case: (TRANS) The last step in a proof of `(conv M N)` is a BACKCHAIN on the clause specifying the TRANS rule. By BACKCHAIN on this clause followed by AND search, the following judgments hold,

$$\Sigma; \texttt{convert} \vdash_I \texttt{conv M P} \qquad \text{and} \qquad \Sigma; \texttt{convert} \vdash_I \texttt{conv P N}$$

where P is a term of type tm in $H(\Sigma)$. Thus, its $\beta\eta$-long form is in $T(\mathrm{dom}(\bar{\rho}))$. By the induction hypothesis for conv, $\|\mathtt{M}\|_{\bar{\rho}} =_{\beta\eta} \|\mathtt{P}\|_{\bar{\rho}}$ and $\|\mathtt{P}\|_{\bar{\rho}} =_{\beta\eta} \|\mathtt{N}\|_{\bar{\rho}}$. Thus, by an application of the TRANS rule $\|\mathtt{M}\|_{\bar{\rho}} =_{\beta\eta} \|\mathtt{N}\|_{\bar{\rho}}$.

Case: (SYM) This case is similar to the above case for TRANS (without the AND search operation).

Case: (RED) The last step in a proof of (conv M N) is a BACKCHAIN on the clause specifying the RED rule. Thus the judgment $\Sigma;\mathtt{convert} \vdash_I \mathtt{(red1\ M\ N)}$ holds. By the induction hypothesis for red1, $\|\mathtt{M}\|_{\bar{\rho}} \rightarrow \|\mathtt{N}\|_{\bar{\rho}}$. Thus, by an application of the RED rule $\|\mathtt{M}\|_{\bar{\rho}} =_{\beta\eta} \|\mathtt{N}\|_{\bar{\rho}}$.

Case: (β) The last step in a proof of (red1 M N) is a BACKCHAIN on the first red1 clause, and the last step in the proof of the resulting subgoal (redex M N) is a BACKCHAIN on the redex clause specifying the β axiom. Then M has the form (app (abs R) Q) where X is the bound variable in R and P is the body, and N is the $\beta\eta$-long form of (R Q). By β-conversion at the meta-level (R Q) $=_{\beta\eta}$ [Q/X]P. Since R and Q are in $\beta\eta$-long form and Q has base type tm, it is easy to see that [Q/X]P is in $\beta\eta$-long form. We must show that

$$\|\mathtt{(app\ (abs\ R)\ Q)}\|_{\bar{\rho}} \rightarrow \|\mathtt{[Q/X]P}\|_{\bar{\rho}}.$$

Let x be a variable such that $x \notin \mathrm{cod}(\bar{\rho})$. By the definition of the decoding

$$\|\mathtt{(app\ (abs\ R)\ Q)}\|_{\bar{\rho}} \equiv (\lambda x.\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}})\|\mathtt{Q}\|_{\bar{\rho}}$$

By the β axiom of the object language,

$$(\lambda x.\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}})\|\mathtt{Q}\|_{\bar{\rho}} \rightarrow [\|\mathtt{Q}\|_{\bar{\rho}}/x]\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}}.$$

The variable decodings $\bar{\rho}$ and $\langle \mathtt{X}, x\rangle + \bar{\rho}$ agree on common domain elements that are free in [Q/X]P. Thus, by Corollary 4.10,

$$[\|\mathtt{Q}\|_{\bar{\rho}}/x]\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}} = \|\mathtt{[Q/X]P}\|_{\bar{\rho}}.$$

Thus

$$\|\mathtt{(app\ (abs\ R)\ Q)}\|_{\bar{\rho}} \rightarrow \|\mathtt{[Q/X]P}\|_{\bar{\rho}}.$$

Case: (η) The last step in a proof of (red1 M N) is a BACKCHAIN on the first red1 clause, and the last step in the proof of the resulting subgoal (redex M N) is a BACKCHAIN on the redex clause specifying the η axiom. Then M has the form (abs X\(app P X)) and N is P, and as discussed in Section 4.2, X does not occur in P. We must show that

$$\|\mathtt{(abs\ X\backslash(app\ P\ X))}\|_{\bar{\rho}} \rightarrow \|\mathtt{P}\|_{\bar{\rho}}.$$

Let x be a variable such that $x \notin \mathrm{cod}(\bar{\rho})$. By the definition of the decoding

$$\|\mathtt{(abs\ X\backslash(app\ P\ X))}\|_{\bar{\rho}} \equiv \lambda x.(\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}})x$$

By Lemma 4.7 (1), x does not occur free in $\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}}$. Thus, by the η axiom of the object level,

$$\lambda x.(\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}})x \to \|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}}.$$

Since x does not occur free in $\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}}$, by Corollary 4.9, $\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}} = \|\mathtt{P}\|_{\bar{\rho}}$. Thus

$$\|\texttt{(abs X\(app P X))}\|_{\bar{\rho}} \to \|\mathtt{P}\|_{\bar{\rho}}.$$

Case: (CONG1) The last step in a proof of `(red1 M N)` is a BACKCHAIN on the clause specifying the CONG1 rule. Then `M` has the form `(app P T)`, `N` has the form `(app Q T)`, and by BACKCHAIN the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{(red1 P Q)}$$

By the definition of the decoding,

$$\|\texttt{(app P T)}\|_{\bar{\rho}} \equiv \|\mathtt{P}\|_{\bar{\rho}}\|\mathtt{T}\|_{\bar{\rho}} \quad \text{and} \quad \|\texttt{(app Q T)}\|_{\bar{\rho}} \equiv \|\mathtt{Q}\|_{\bar{\rho}}\|\mathtt{T}\|_{\bar{\rho}}.$$

By the induction hypothesis for `red1`,

$$\|\mathtt{P}\|_{\bar{\rho}} \to \|\mathtt{Q}\|_{\bar{\rho}}.$$

By an application of the CONG1 rule,

$$\|\mathtt{P}\|_{\bar{\rho}}\|\mathtt{T}\|_{\bar{\rho}} \to \|\mathtt{Q}\|_{\bar{\rho}}\|\mathtt{T}\|_{\bar{\rho}}.$$

Thus,

$$\|\texttt{(app P T)}\|_{\bar{\rho}} \to \|\texttt{(app Q T)}\|_{\bar{\rho}}.$$

Case: (CONG2) This case is similar to the above case for CONG1.

Case: (ξ) The last step in a proof of `(red1 M N)` is a BACKCHAIN on the clause specifying the ξ rule. Then `M` has the form `(abs R)` where `X` is the bound variable in `R` and `P` is the body, and `N` has the form `(abs S)` where `X` is the bound variable in `S` and `Q` is the body. We assume that `X` does not appear in Σ, otherwise it can be renamed in `R` and `S`. By BACKCHAIN the following judgment holds.

$$\Sigma; \texttt{convert} \vdash_I \texttt{pi X\(red1 (R X) (S X))}$$

By the GENERIC search operation, the following judgment holds.

$$\Sigma \cup \{\mathtt{X} : \mathtt{tm}\}; \texttt{convert} \vdash_I \texttt{red1 (R X) (S X)}$$

By β-conversion at the meta-level `(R X)` $=_{\beta\eta}$ `P` and `(S X)` $=_{\beta\eta}$ `Q`, and since `R` and `S` are $\beta\eta$-long forms, so are `P` and `Q`. Let x be a variable such that $x \notin \mathrm{cod}(\bar{\rho})$. Then $\langle \mathtt{X}, x\rangle + \bar{\rho}$

is well-defined on `P` and `Q` and these terms are in $T(\mathrm{dom}(\langle \mathtt{X}, x\rangle + \bar{\rho}))$. By the induction hypothesis for `red1`,

$$\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}} \rightarrow \|\mathtt{Q}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}}.$$

By an application of the ξ rule,

$$\lambda x.\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}} \rightarrow \lambda x.\|\mathtt{Q}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}}.$$

By the definition of the decoding,

$$\|\texttt{(abs R)}\|_{\bar{\rho}} \equiv \lambda x.\|\mathtt{P}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}} \qquad \text{and} \qquad \|\texttt{(abs S)}\|_{\bar{\rho}} \equiv \lambda x.\|\mathtt{Q}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}}.$$

Thus,

$$\|\texttt{(abs R)}\|_{\bar{\rho}} \rightarrow \|\texttt{(abs S)}\|_{\bar{\rho}}.$$

■

Note that these proofs illustrated more than just correctness of the `conv` program. They in fact illustrated a step-by-step correspondence between proofs in the object language and proofs in the meta-language. Each application of a rule at the object level corresponds to a BACKCHAIN on a particular clause at the meta-level. Similar results for any of the other specifications in this and the previous chapter can be established. One aspect that this example did not illustrate is the discharge of assumptions that occurs in natural deduction style proof systems. In specifications for proof systems that include this operation, not only signature items as in the above example but also program clauses get added via the AUGMENT operation as the proof proceeds.

In the specifications that construct proof terms, the step-by-step correspondence between object and meta-proofs gets recorded more precisely, since the constant at the head of the proof term determines which clause was used at each step of the meta-proof. When proof terms contain enough information, it is possible to define an encoding and decoding between proof terms and object-level proof trees, and proceed to establish correctness by showing the bijectivity of these functions. For example, in Section 3.4, a proof representation was given that should be sufficient to establish a one-to-one correspondence between proof terms constructed by the `niprover` specification and natural deduction trees in the style of [Pra65] as defined in Section 3.4 up to an equivalence which identifies deductions that are the same except for the names of parameters to the $\forall$-I and $\exists$-E rules.

As we will see, the specification of β-conversion in LF in Chapter 5 is quite similar to the `convert` module. Thus the results proven in this section extend rather directly to a proof of correctness of that specification. In addition, in that chapter, we also will consider not only convertibility but also normalization of λ-terms.

4.4 Specification of a Higher-Order Logic

In Chapter 3, we saw many specifications for first-order logic. We now demonstrate the specification of a higher-order logic. In the specification of a first-order logic, it was convenient to introduce `i` as the type of all first-order individuals, and then define the quantifiers in terms of meta-level abstractions over this type. Thus, both `forall` and `exists` had type `(i -> form) -> form`. In fact, we could easily have extended such a specification to a many-sorted logic (with a finite set of sorts), declaring one primitive type corresponding to each sort of the object logic, thus identifying types of the object logic with types of the meta-language. We would then need to introduce quantifiers over each sort, *i.e.*, for each primitive type `s`, we declare:

```
type    exists_s    (s -> form) -> form.
type    forall_s    (s -> form) -> form.
```

In addition, we would need quantifier inference rules for each sort.

In higher-order logic, on the other hand, we need to quantify over not only primitive types but also types at any functional level. For example, a second-order formula could contain a quantification over a predicate, say of arity 2, which in our formulation is an object of type `i -> i -> form`. One approach would be to declare polymorphic quantifiers as follows:

```
type    exists      (S -> form) -> form.
type    forall      (S -> form) -> form.
```

where the type variable `S` indicates that the argument to `exists` or `forall` can be an abstraction over an object of any meta-level type. There are several reasons to avoid the use of type variables here. Consider the following specification of the ∃-I rule.

```
(exists_i P) # (exists X\(A X)) :- P # (A T).
```

When `exists` has polymorphic type `(S -> form) -> form`, `A` has polymorphic type `(S -> form)`. Thus, this clause allows instances of `X\(A X)` where the type of `X` is any one of an infinite number of types such as `i`, `i --> i`, `i --> form`, etc. Operationally, in backchaining over this clause, even if the term unifying with `(A T)` were a closed term, a correct implementation of unification would branch infinitely in finding unifiers for `A` and `T`. In a particular unification problem, not only are there an infinite number of instantces of `S`, there may be infinitely many solutions for some fixed instance of `S`. For example, consider unifying the closed term `(p c)` with `(A T)` where `p` has type `i -> form` and `c` has type `i`. When the type variable `S` is instantiated with base type `i`, there are just two solutions to this unification problem: (1) `A` is `Y\(p Y)` and `T` is `c`, or (2) `A` is `Y\(p c)` and `T` is itself. But when `S` is the functional type `i -> i`, we have the following infinite set of solutions: `A` is `F\(p (F`^{n} `c))` and `T` is `Y\Y`, where $n \geq 0$. In addition, the type for `X` is allowed to be any meta-type including types we may not intend to quantify over, such

as the type of formulas of the meta-language (type `o`), or more complicated types such as `(list o -> o)`.

Instead, as in the specification of the λ-calculus, we will specify object-level types as terms of the meta-language having meta-type `ty`. Now, instead of introducing constants such as `i` and `form` as primitive meta-types as in the specification of first-order logic, we declare them as constants of type `ty` as follows.

```
type    i       ty.
type    form    ty.
```

Formulas are now a subclass of terms of meta-type `tm`, and we declare the connectives as object-level term constructors as follows.

```
type    'and'   tm -> tm -> tm.
type    'or'    tm -> tm -> tm.
type    'imp'   tm -> tm -> tm.
type    neg     tm -> tm.
type    forall  ty -> (tm -> tm) -> tm.
type    exists  ty -> (tm -> tm) -> tm.
type    false   tm.
```

We also include the `abs` and `app` constants and their types as in Section 4.2:

```
type    abs     (tm -> tm) -> tm.
type    app     tm -> tm -> tm.
```

The quantifiers `exists` and `forall` are specified so that the type of the quantified object appears as an extra argument in the quantified expression. Thus the first argument to `exists` and `forall` is a term of meta-type `ty`. For operations such as quantifier instantiation, we must now check that the (object-level) type of the substitution term matches the type of the quantified variable before performing the operation. In addition, formulas must be type-checked in order to insure that they are well-formed. The type-checking of the meta-language only insures that they have meta-type `tm`. They must also have object-level type `form`. We implement a type-checking program to perform such checks. This program will include the two clauses in the previous section for type-checking λ-terms. In addition, we will need one clause for each logical connective and one for each non-logical constant of the object language. The clauses for the logical connectives are as follows.

```
(A and B) #t form :- A #t form, B #t form.
(A or B) #t form :- A #t form, B #t form.
(A imp B) #t form :- A #t form, B #t form.
(neg A) #t form :- A #t form.
(exists S A) #t form :- pi X\ ((X #t S) => ((A X) #t form)).
(forall S A) #t form :- pi X\ ((X #t S) => ((A X) #t form)).
```

Formulas with a propositional connective at the top-level are well-formed if their subformulas are well-formed. In the quantified formula `(exists S A)`, `S` is the object-level type of the variable bound by λ-abstraction in `A`. The clause above for type checking terms of

this form reads: for arbitrary `X` of type `S`, if `(A X)` is a formula, then `(exists S A)` is a formula. As in the clause in the previous section for `abs`, a new constant of meta-type `tm` is introduced for `X` and substituted for the bound variable in `A`. An assumption about the object-level type of this new object is added to the program, which may be used in subsequent type checking subgoals. Type checking for `forall` is similar.

Consider a language with a constant c of type i, functions f and g having types $i \to i \to i$ and $(i \to i) \to i \to i$, respectively, and a predicate q that takes one argument of type i. We introduce meta-constants `c`, `f`, `g`, and `q` which are declared with the following types.

```
type    c       tm.
type    f       tm -> tm -> tm.
type    g       (tm -> tm) -> tm -> tm.
type    q       tm -> tm.
```

The following clauses are needed to type-check terms built from these constants.

```
c #t i.
(f X Y) #t i :- X #t i, Y #t i.
(g F X) #t i :- pi Y\ ((Y #t i) => ((F Y) #t i)), X #t i.
(q X) #t form :- X #t i.
```

Clauses such as these for the non-logical constants of the object language, together with the clauses above for the connectives and the two clauses in the previous section for type-checking λ-terms make up the complete type-checking program.

We will specify a natural deduction proof system for higher-order logic, and as in previous examples, introduce the meta-type `nprf` for proofs, and an infix constant `#p` to represent the basic relation between a formula and its proofs, declared as follows.

```
type    '#p'    nprf -> tm -> o.
```

This constant is analogous to `#` in the first-order natural deduction specification, but tagged here by `p` to distinguish it from the `#t` relation between terms and types. We introduce `#` as a top-level predicate, with the same type as `#p`, whose function is to insure that formulas are well-formed in addition to checking if they are provable. This relation is defined by the single definite clause below.

```
P # A :- A #t form, P #p A.
```

In specifying the inference rules, we will assume that the term on the right side of `#p` in the head of each clause has type `form`. As a result, very few type-checking subgoals will be needed in these clauses.

The propositional rules for this logic are the same as those for natural deduction in first-order logic (see Figure 3.2). In fact, the clauses for the introduction rules for the propositional connectives and for the $\bot$-I rules need no type-checking subgoals, and thus can be specified similarly to the corresponding rules for first-order logic. Clauses for these rules are given below. The only difference between these clauses and clauses seen in Section 3.2 is the use of the `#p` predicate instead of the `#` predicate.

$$\frac{\begin{array}{c}(x:\tau)\\ [y/x]A\end{array}}{\forall x:\tau.A}\ \forall\text{-I} \qquad \frac{\forall x:\tau.A \qquad t:\tau}{[t/x]A}\ \forall\text{-E}$$

$$\frac{t:\tau \qquad [t/x]A}{\exists x:\tau.A}\ \exists\text{-I} \qquad \frac{\exists x:\tau.A \qquad \begin{array}{c}(y:\tau)\ ([y/x]A)\\ B\end{array}}{B}\ \exists\text{-E}$$

The $\forall$-I rule has the proviso that the variable y cannot appear free in $\forall xA$, or in any assumption on which $[y/x]A$ depends.

The $\exists$-E rule has the proviso that the variable y cannot appear free in $\exists xA$, in B, or in any assumption on which the upper occurrence of B depends.

Figure 4.6: Quantifier Rules for Higher-Order Logic

```
(and_i P1 P2) #p (A and B) :- P1 #p A, P2 #p B.
(or_i1 P) #p (A or B) :- P #p A.
(or_i2 P) #p (A or B) :- P #p B.
(imp_i P) #p (A imp B) :- pi PA\ ((PA #p A) => ((P PA) #p B)).
(neg_i P) #p (neg A) :- pi PA\ ((PA #p A) => ((P PA) #p false)).
(false_i P) #p A :- P #p false.
```

The clauses for the elimination rules for the propositional connectives are also similar to the corresponding clauses for first-order logic. In these clauses, some additional type-checking subgoals are needed to insure all formulas have type `form`. These clauses are specified as follows.

```
(and_e1 P) #p A :- P #p (A and B), B #t form.
(and_e2 P) #p B :- P #p (A and B), A #t form.

(or_e P P1 P2) #p C :- P #p (A or B), (A or B) #t form,
                       pi PA\ ((PA #p A) => ((P1 PA) #p C)),
                       pi PB\ ((PB #p B) => ((P2 PB) #p C)).

(imp_e P1 P2) #p B :- P1 #p A, A #t form,
                      P2 #p (A imp B).

(neg_e P1 P2) #p false :- P1 # A, A #t form,
                          P2 #p (neg A).
```

The quantifier rules of higher-order logic are given in Figure 4.6. They are also similar to those for first-order logic except that the quantified object can be of any type, and additional subproofs are needed to insure that substitution terms in the $\exists$-I and $\forall$-E rules match the types of the quantified variable. The clause for $\exists$-I is as follows:

```
(exists_i S T P) #p (exists S A) :- T #t S, P #p (A T).
```

Here the first subgoal checks that the substitution term `T` has type `S`. The second, as in the first-order case, checks that `P` is a proof of `(A T)`, the formula resulting from substituting `T` for the bound variable in `A`. As in the first-order case, there are many choices concerning what information to include in proof terms. Here we have included both the substitution term `T` and its type `S`.

The following definite clause specifies the ∀-I rule for higher-order logic.

```
(forall_i S P) #p (forall S A) :- pi Y\ ((Y #t S) => ((P Y) #p (A Y))).
```

As in the first-order case, universal quantification at the meta-level is used to handle the proviso on this rule. Here, meta-level implication is also necessary to add an assumption about the type of the new signature item introduced for `Y`. This assumption may be used in subsequent type-checking subgoals such as those generated by applications of the ∃-I and ∀-E rules. Like the clause for ∃-I, the object-level type `S` is included in the proof term. The remaining quantifier rules, ∀-E and ∃-E, are specified similarly. They also require subgoals for type-checking formulas.

```
(forall_e S T P) #p (A T) :- T #t S, (forall S A) #t form,
                                P #p (forall S A).
(exists_e P1 P2) #p B :-
   P1 #p (exists S A), (exists S A) #t form,
   pi Y\ (pi P\ ((Y #t S) => ((P #p (A Y)) => ((P2 Y P) #p B)))).
```

In the higher-order version of ∃-E, two assumptions are discharged, one containing the type information for the variable used in substitution, and the other associating a proof with the discharged formula. Since we include substitution terms in proofs, `P2` must be an abstraction over `Y` and `P`.

In higher-order logic, there is usually some notion of equality between terms, and as a result, two different terms of type `form` may represent the same formula. If we choose to define equality up to $\beta\eta$-convertibility, we can make use of the `conv` program in the previous section and conclude our specification of higher-order logic with the following simple clause.

```
(convert A B P) #p A :- B #t form, P #p B, conv A B.
```

Here, `(convert A B P)` is a proof of formula `A` if `B` has type `form`, `P` is a proof of `B`, and `A` converts to `B`.

4.5 Discussion

In this and the previous chapter, we have presented several examples in support of our claim that the meta-language hohh can be used to naturally specify theorem provers for a variety of object logics. Certainly, many others can be specified. For example, it is also straightforward to specify inference systems for various modal and temporal logics. In

addition, various other type systems such as the second-order polymorphic λ-calculus, LF, and the calculus of constructions can be specified. (See [FM89].) Such type checking specifications are similar in spirit to the specification of the simply-typed λ-calculus presented in Section 4.2.

Several of the higher-order features of this meta-language have been extremely valuable in the specification of certain aspects of object logics. Yet we have only made limited use of some of these features. For example, we have used no predicate quantification so far, and quantification over functions has been at most second-order. For instance, the following clause from the `niprover` module on page 34 for the $\forall$-E rule of N_I illustrates the most complex use of function quantification in any of the specifications.

```
(forall_e T A P) # (A T) :- P # (forall A).
```

In this clause the variable `A` has second-order type `i -> form`. A similar use of quantification at the meta-level was used in all of the other specifications. The use of `pi` and `sigma` in the subgoals of clauses is even more restricted. In our examples, such quantification has been over only base types.

Operationally, the simple uses of function variables result in fairly simple unification problems. In all of the specifications, second-order unification is the most that is needed. In theorem proving, deciding which inference rule can be applied requires unifying the formula or sequent to be proved with the formula or sequent in the heads of definite clauses. When the sequent or formula to be proved is fully specified, deciding which clauses can be used will require only second-order matching. In the above clause for example, in unifying a formula with `(A T)`, there may be more than one unifier for `A` and `T`, but finding the unifiers will be relatively simple.

In [FM89], a sublanguage of hohh called L_λ is described. In this language, quantification over predicates is not allowed and quantification over function variables is greatly restricted. As a result, only very simple β-redexes occur in programs, and hence unification problems are very simple. In fact, for this sublanguage, unification is decidable and most general unifiers always exist. All of the specifications in this and the previous chapter with only minor modification, do in fact fall within this sublanguage.

In specifying various object logics, certain identifications were made between aspects of the object and meta-logic. For example, in first-order logic, bound variables in each language were identified, and first-order terms and formulas were identified with meta-terms of type `i` and `form`, respectively. The specification of higher-order logic illustrated that more complex logics can also be naturally specified as hohh formulas, yet their specification is not quite as direct as that for simpler logics. For example, we demonstrated the need for an encoding of terms and formulas as meta-terms of type `tm`. Object level types were represented as meta-terms also. As a result, auxiliary type-checking subgoals were often needed. In addition, a program for $\beta\eta$-conversion had to be employed since unification

at the meta-level could not be used to solve object level equations. On the other hand, the encoding of any term or formula of higher-order logic is a meta-term of order two or less, and thus unification problems were no more complex than those occurring in the specifications for first-order logics.

The Isabelle theorem prover [Pau88] contains a specification language based on a fragment of higher-order logic with implication and universal quantification. This language is used to specify inference rules for various object logics. That fragment is essentially a subset of higher-order hereditary Harrop formulas. Hence, many of the object logics we have specified here could be very similarly specified in the specification language of Isabelle. More will be said about Isabelle in the discussion of the implementation of theorem provers in Chapter 7. The LF type theory [HHP89] is another language in which object logics can be specified. In the next chapter, we examine in more detail the correspondence between specifying a logic in LF and in hohh.

Chapter 5

LF Signatures as Logic Programs

The Edinburgh Logical Framework (LF) [HHP89] is a typed λ-calculus developed for the purpose of specifying a wide class of logics so that commonalities among these logics can be exploited in implementing theorem provers and proof systems. In this chapter, we illustrate how logics specified in LF can be specified as logic programs. We present a translation for "compiling" LF signatures into logic programs and LF judgments into logic programming queries. First, in Section 5.1, we present a modified form of the LF system. Proofs in this modified system will correspond closely to proofs constructed by the non-deterministic interpreter in executing the programs obtained from compiled LF signatures. In Section 5.2, we provide a specification of β-normalization in LF. We represent terms in a manner similar to the representation of untyped λ-terms in Section 4.2. In fact, the specification of β-conversion is quite similar to the `convert` program for the untyped λ-calculus, and thus the correctness proofs of Section 4.3 easily extend to the program given here. In addition, we consider not only conversion but also normalization of LF terms. Normalization will play an important role in the execution of compiled LF signatures. In Section 5.3, we present the translation from LF signatures to logic programs and prove it correct, and finally in Section 5.4, we illustrate the translation on a subset of an LF signature specifying natural deduction for first-order intuitionistic logic.

5.1 Canonical LF

The LF system is a typed λ-calculus with dependent products. We begin here by presenting the simplified algorithmic version of LF as defined in [HHP89]. One difference in the richer system is that it contains both signatures which associate constants with their types, and contexts which associate variables with their types. The simplified system unifies these two as contexts. We continue, however, to use the term signature to mean a set of variables associated with their types that specify an object logic.

The system has three levels of terms: objects (often called just terms), types and families of types, and kinds. The syntax of LF is given by the following classes of objects.

$$\begin{array}{rcl} \Gamma & := & \langle\rangle \mid \Gamma, x : K \mid \Gamma, x : A \\ K & := & \mathsf{Type} \mid \Pi x : A.K \\ A & := & x \mid \Pi x : A.B \mid \lambda x : A.B \mid AM \\ M & := & x \mid \lambda x : A.M \mid MN \end{array}$$

Here Γ represents a context, M and N range over expressions for objects, A and B over types and families of types, K over kinds, and x over variables. We will use P and Q to range over arbitrary objects, families, or kinds. We call a variable that is a type a *type variable*, and a variable that is an object a *term variable*. A type or kind of the form $\Pi x : A.P$ will often be abbreviated $A \to P$ when x does not occur in P.

The three kinds of *assertions* in LF are as follows.

$\Gamma \vdash K \Rightarrow \text{kind}$	(K is a kind in Γ)
$\Gamma \vdash A \Rightarrow K$	(A has kind K in Γ)
$\Gamma \vdash M \Rightarrow A$	(M has type A in Γ)

We write $\Gamma \vdash \alpha$ for an arbitrary assertion.

Equality of terms, types, and kinds in LF is up to β-conversion, specified by the proof system in Figure 5.1. A *normal form* is a term, type, or kind with no subterms of the form $(\lambda x : A.P)M$. It is shown in [HHP89] that the Church-Rosser property holds for LF, and that well-typed expressions are strongly normalizing. We will write P^β to denote the β-normal form of term, type, or kind P, and similarly for families and kinds. We say that a context Γ is in β-normal form, if for every item $x : P$ in Γ, P is in β-normal form. We right Γ^β to denote the β-normal form of a context Γ. Normal forms have the following characterization.

1. A kind K is a normal form iff it has the form $\Pi x_1 : A_1 \ldots \Pi x_n : A_n.\mathsf{Type}$ where $n \geq 0$ and for $i = 1, \ldots, n$, A_i is a normal form type.
2. A type A is a normal form iff it has the form

   $$\lambda x_1 : A_1 \ldots \lambda x_n : A_n.\Pi y_1 : B_1 \ldots \Pi y_m : B_m.xM_1 \ldots M_k$$

   where $n, m, k \geq 0$, for $i = 1, \ldots, n$, A_i is a normal form type, for $i = 1, \ldots, m$, B_i is a normal form type, x is a type variable, and for $i = 1, \ldots, k$, M_i is a normal form term.
3. A term M is a normal form iff it has the form $\lambda x_1 : A_1 \ldots \lambda x_n : A_n.xM_1 \ldots M_k$ where $n, k \geq 0$, for $i = 1, \ldots, n$, A_i is a normal form type, x is a term variable, and for $i = 1, \ldots, k$, M_i is a normal form term.

$$(\lambda x : A.M)N \rightarrow_\beta [N/x]M \quad (\beta\text{-OBJ}) \qquad (\lambda x : A.B)N \rightarrow_\beta [N/x]B \quad (\beta\text{-FAM})$$

$$\frac{M \rightarrow_\beta M'}{\lambda x : A.M \rightarrow_\beta \lambda x : A.M'}\ (\xi\text{-ABS-OBJ}) \qquad \frac{B \rightarrow_\beta B'}{\lambda x : A.B \rightarrow_\beta \lambda x : A.B'}\ (\xi\text{-ABS-FAM})$$

$$\frac{A \rightarrow_\beta A'}{\lambda x : A.M \rightarrow_\beta \lambda x : A'.M}\ (\text{B-ABS-OBJ}) \qquad \frac{A \rightarrow_\beta A'}{\lambda x : A.B \rightarrow_\beta \lambda x : A'.B}\ (\text{B-ABS-FAM})$$

$$\frac{B \rightarrow_\beta B'}{\Pi x : A.B \rightarrow_\beta \Pi x : A.B'}\ (\xi\text{-PI-FAM}) \qquad \frac{K \rightarrow_\beta K'}{\Pi x : A.K \rightarrow_\beta \Pi x : A.K'}\ (\xi\text{-PI-KIND})$$

$$\frac{A \rightarrow_\beta A'}{\Pi x : A.B \rightarrow_\beta \Pi x : A'.B}\ (\text{B-PI-FAM}) \qquad \frac{A \rightarrow_\beta A'}{\Pi x : A.K \rightarrow_\beta \Pi x : A'.K}\ (\text{B-PI-KIND})$$

$$\frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N}\ (\text{CONG1-OBJ}) \qquad \frac{N \rightarrow_\beta N'}{MN \rightarrow_\beta MN'}\ (\text{CONG2-OBJ})$$

$$\frac{A \rightarrow_\beta A'}{AN \rightarrow_\beta A'N}\ (\text{CONG1-FAM}) \qquad \frac{N \rightarrow_\beta N'}{AN \rightarrow_\beta AN'}\ (\text{CONG2-FAM})$$

$$\frac{P_1 \rightarrow_\beta P_2}{P_1 =_\beta P_2}\ (\text{RED}) \qquad P =_\beta P \quad (\text{REFL})$$

$$\frac{P_1 =_\beta P_2}{P_2 =_\beta P_1}\ (\text{SYM}) \qquad \frac{P_1 =_\beta P_2 \qquad P_2 =_\beta P_3}{P_1 =_\beta P_3}\ (\text{TRANS})$$

Figure 5.1: β-Convertibility in LF

The inference rules of LF are given in Figure 5.2. The set of variables on the left of the colon in a context Γ is denoted as $\mathrm{dom}(\Gamma)$. We often write $\Gamma \vdash \alpha$ to mean that the indicated assertion is provable in the system. A proof of an assertion of the form $\Gamma \vdash \mathsf{Type} \Rightarrow \mathrm{kind}$ proves that Γ is a *valid context*.

$$\vdash \mathsf{Type} \Rightarrow \mathrm{kind} \quad \text{(A-TYPE-KIND)}$$

$$\frac{\Gamma \vdash K \Rightarrow \mathrm{kind} \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : K \vdash \mathsf{Type} \Rightarrow \mathrm{kind}} \text{(A-K-VAR)}$$

$$\frac{\Gamma \vdash A \Rightarrow \mathsf{Type} \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash \mathsf{Type} \Rightarrow \mathrm{kind}} \text{(A-T-VAR)}$$

$$\frac{\Gamma \vdash A \Rightarrow \mathsf{Type} \qquad \Gamma, x : A \vdash K \Rightarrow \mathrm{kind}}{\Gamma \vdash \Pi x : A.K \Rightarrow \mathrm{kind}} \text{(A-PI-KIND)}$$

$$\frac{\Gamma \vdash \mathsf{Type} \Rightarrow \mathrm{kind} \qquad x : K \in \Gamma}{\Gamma \vdash x \Rightarrow K^{\beta}} \text{(A-VAR-FAM)}$$

$$\frac{\Gamma \vdash A \Rightarrow \mathsf{Type} \qquad \Gamma, x : A \vdash B \Rightarrow \mathsf{Type}}{\Gamma \vdash \Pi x : A.B \Rightarrow \mathsf{Type}} \text{(A-PI-FAM)}$$

$$\frac{\Gamma \vdash A \Rightarrow \mathsf{Type} \qquad \Gamma, x : A \vdash B \Rightarrow K}{\Gamma \vdash \lambda x : A.B \Rightarrow \Pi x : A^{\beta}.K} \text{(A-ABS-FAM)}$$

$$\frac{\Gamma \vdash A \Rightarrow \Pi x : B.K \qquad \Gamma \vdash M \Rightarrow B}{\Gamma \vdash AM \Rightarrow ([M/x]K)^{\beta}} \text{(A-APP-FAM)}$$

$$\frac{\Gamma \vdash \mathsf{Type} \Rightarrow \mathrm{kind} \qquad x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A^{\beta}} \text{(A-VAR-OBJ)}$$

$$\frac{\Gamma \vdash A \Rightarrow \mathsf{Type} \qquad \Gamma, x : A \vdash M \Rightarrow B}{\Gamma \vdash \lambda x : A.M \Rightarrow \Pi x : A^{\beta}.B} \text{(A-ABS-OBJ)}$$

$$\frac{\Gamma \vdash M \Rightarrow \Pi x : A.B \qquad \Gamma \vdash N \Rightarrow A}{\Gamma \vdash MN \Rightarrow ([N/x]B)^{\beta}} \text{(A-APP-OBJ)}$$

Figure 5.2: The Edinburgh Logical Framework

LF is based on the *judgments-as-types* principle in which a basic judgment of an object logic such as "the formula A is provable" is encoded as a type. An LF term inhabiting such a type represents an object level proof. This is the view of LF terms that we are interested in, since it corresponds to the use of terms of our meta-language (the simply typed λ-terms) in previous chapters to represent object level proofs. Notice that the proof terms we presented were always in normal form. For example, in Section 3.4, the meta-term `(imp_i P\(imp_i Q\P))` was given as a natural deduction proof term for $p \supset (p \supset p)$. Since terms in the meta-language are equated up to $\beta\eta$-convertibility, the term `(imp_i P\(imp_i Q\((W\W) P)))`, for example, will represent the same proof. Yet, normal forms were always used in presentation since that is the form in which proof terms can be seen to correspond to the object level proofs which they represent, and in which manipulations of these terms can be described. Note that the term `(imp_i P\(imp_i Q\Q))` is a different meta-term and represents a different object-level proof of the same formula. In LF, as in logic programming, there will be different meta-proofs to construct different object-level proofs, but there may also be different meta-proofs to construct proof terms that are β-equivalent. Since we are concerned with the interpretation of LF terms as object level proofs, we will consider β-equivalent terms to represent the same object level proof, and will thus restrict our attention to LF terms in normal form. Note that normal proofs at the object level (in N_I for example) is a different concept. Even non-normal object-level proofs have been represented thus far as normal meta-terms. Determining whether or not a proof term represents an object level normal proof involves examining the structure of the term. The remainder of this section is devoted to the presentation of a modified form of LF that builds proofs of only normal terms. In fact, we will only consider terms in *canonical* form, a further restriction which corresponds to $\beta\eta$-long form as defined for the simply typed λ-calculus. Proofs in this modified LF will in fact correspond closely to meta-level proofs built by executing programs derived from LF signatures.

Several more definitions from [HHP89] are required. We define the *arity* of a type or kind to be the number of Πs in the prefix of its normal form. The arity of a variable with respect to a context is the arity of its type in that context. The arity of a bound variable occurrence in a term is the arity of the type label attached to its binding occurrence. We say that an occurrence of a variable x in a term is *fully applied* with respect to a context if it occurs in a subterm of the form $xM_1, \ldots M_n$, where n is the arity of x. A term P is canonical with respect to a context Γ if every variable occurrence in P is fully applied with respect to Γ. A term has a canonical form if its normal form is canonical. We say that a context Γ is in or has canonical form if for every item $x : P$ in Γ, P is in or has canonical form with respect to Γ.

The following characterization of canonical forms will be useful in the proofs in this chapter.

1. If K is a canonical kind with respect to Γ, then it is of the form

$$\Pi x_1 : A_1 \ldots \Pi x_n : A_n.\mathsf{Type}$$

where for $i = 1, \ldots, n$, A_i is canonical with respect to $\Gamma, x_1 : A_1, \ldots, x_{i-1} : A_{i-1}$.

2. If A is a canonical form with canonical kind

$$\Pi x_1 : A_1 \ldots \Pi x_n : A_n.\mathsf{Type}$$

with respect to Γ, then A is of the form

$$\lambda x_1 : A_1 \ldots \lambda x_n : A_n.\Pi y_1 : B_1 \ldots \Pi y_m : B_m.xM_1 \ldots M_k$$

where k is the arity of x, for $i = 1, \ldots, m$, B_i is canonical with respect to

$$\Gamma, x_1 : A_1, \ldots, x_n : A_n, y_1 : B_1, \ldots, y_{i-1} : B_{i-1},$$

and for $i = 1, \ldots, k$, M_i is canonical with respect to

$$\Gamma, x_1 : A_1, \ldots, x_n : A_n, y_1 : B_1, \ldots, y_m : B_m.$$

3. If M is a canonical form with canonical type

$$\Pi x_1 : A_1 \ldots \Pi x_n : A_n.xM_1 \ldots M_k$$

with respect to Γ, then M is of the form

$$\lambda x_1 : A_1 \ldots \lambda x_n : A_n.yN_1 \ldots N_l$$

where l is the arity of y, and for $i = 1, \ldots, l$, N_i is canonical with respect to

$$\Gamma, x_1 : A_1, \ldots, x_n : A_n.$$

We make use of the following results about LF. The first is from [HHP89], and the second is a simple observation about the judgments that can be proved in LF. We note here that this latter result does not hold in more general presentations of LF, but is important for our purposes.

Theorem 5.1 (Subject Reduction)

1. If $\Gamma \vdash K \Rightarrow \text{kind}$ and $K \rightarrow_\beta K'$, then $\Gamma \vdash K' \Rightarrow \text{kind}$.
2. If $\Gamma \vdash A \Rightarrow K$ and $A \rightarrow_\beta A'$, then $\Gamma \vdash A' \Rightarrow K$.
3. If $\Gamma \vdash M \Rightarrow A$ and $M \rightarrow_\beta M'$, then $\Gamma \vdash M' \Rightarrow A$.

Lemma 5.2

1. If $\Gamma \vdash A \Rightarrow K$, then K is normal.
2. If $\Gamma \vdash M \Rightarrow A$, then A is normal.

Proof: The proof is by induction on the height of a proof of an LF assertion. By inspection of the inference rules, it is easy to see that whenever the terms and types on the right of $\Rightarrow$ in the premises are normal, then the term or type on the right of $\Rightarrow$ in the conclusion must be normal. ■

We will say that a proof of an LF assertion is *normal* if there is no assertion which is the conclusion of (A-ABS-FAM) and the left premise of (A-APP-FAM), and no assertion which is the conclusion of (A-ABS-OBJ) and the left premise of (A-APP-OBJ).

Theorem 5.3 (Existence of Normal Proofs)
Let Γ be a context in normal form.

1. If $\Gamma \vdash K \Rightarrow \text{kind}$, then there is a normal proof of $\Gamma \vdash K^\beta \Rightarrow \text{kind}$.
2. If $\Gamma \vdash A \Rightarrow K$, then there is a normal proof of $\Gamma \vdash A^\beta \Rightarrow K^\beta$.
3. If $\Gamma \vdash M \Rightarrow A$, then there is a normal proof of $\Gamma \vdash M^\beta \Rightarrow A^\beta$.

Proof: By Lemma 5.2, if $\Gamma \vdash A \Rightarrow K$ holds, then K is normal, and if $\Gamma \vdash M \Rightarrow A$ holds, then A is normal. By Theorem 5.1 and the strong normalization property for well-typed objects, one of the following holds.

1. $\Gamma \vdash K^\beta \Rightarrow \text{kind}$
2. $\Gamma \vdash A^\beta \Rightarrow K$
3. $\Gamma \vdash M^\beta \Rightarrow A$

The proof is by simultaneous induction on the height of a proof of the above assertions. We show the case when the last inference is an application of (A-APP-OBJ). The case for (A-APP-FAM) is similar. For all other cases, by assuming that there is a normal proof of the premises, we obtain a normal proof of the conclusion by an application of the corresponding rule.

The conclusion of (A-APP-OBJ) is $\Gamma \vdash MN \Rightarrow ([N/x]B)^\beta$. Since MN is normal, M is not an abstraction. By the induction hypothesis for the premises, $\Gamma \vdash M \Rightarrow \Pi x : A.B$ and $\Gamma \vdash N \Rightarrow A$ have normal proofs. Since M is not an abstraction, the last rule in the proof of $\Gamma \vdash M \Rightarrow \Pi x : A.B$ is not (A-ABS-OBJ). Then by an application of (A-APP-OBJ), we obtain a normal proof of $\Gamma \vdash MN \Rightarrow ([N/x]B)^\beta$. ■

Corollary 5.4 (Form of Normal Proofs)
A normal proof has the form given by the following criteria.

1. The application of rules (A-VAR-OBJ) and (A-APP-OBJ) occur only in proof fragments of the following form:

   (a) $\Gamma \vdash x \Rightarrow \Pi x_1 : A_1 \ldots \Pi x_n : A_n.B$ $(n \geq 0)$ is the conclusion of an application of (A-VAR-OBJ) and the left premise of a series of n applications of (A-APP-OBJ).

   (b) For $i = 1, \ldots, n-1$, the conclusion of the i^{th} application and the left premise of the $(i+1)^{st}$ application of (A-APP-OBJ) is

   $$\Gamma \vdash xN_1 \ldots N_i \Rightarrow ([N_1/x_1, \ldots, N_i/x_i]\Pi x_{i+1} : A_{i+1} \ldots \Pi x_n : A_n.B)^{\beta}$$

   The conclusion of the n^{th} application of (A-APP-OBJ) is

   $$\Gamma \vdash xN_1 \ldots N_n \Rightarrow ([N_1/x_1, \ldots, N_n/x_n]B)^{\beta}.$$

   (c) For $i = 1, \ldots, n$, the right premise of the i^{th} application of (A-APP-OBJ) is

   $$\Gamma \vdash N_i \Rightarrow ([N_1/x_1, \ldots, N_{i-1}/x_{i-1}]A_i)^{\beta}.$$

2. Applications of rules (A-VAR-FAM) and (A-APP-FAM) occur similarly.

Proof: The proof is by induction on the height of a normal proof. All cases follow by a simple application of the induction hypothesis except for (A-APP-FAM) and (A-APP-OBJ). We show the case for (A-APP-OBJ). (A-APP-FAM) is similar. When the last step in a proof is (A-APP-OBJ), by definition of normal, the last step in the subproof of the left premise cannot be (A-ABS-OBJ). It must be either (A-VAR-OBJ) or (A-APP-OBJ). By the induction hypothesis, we know this subproof has the form specified by (1)-(2) above.

In the case when the last step in the subproof of the left premise is (A-VAR-OBJ), the conclusion of this subproof must have the form $\Gamma \vdash x \Rightarrow \Pi z : A.B$. Thus the right premise of the application of (A-APP-OBJ) must have the form $\Gamma \vdash N \Rightarrow A$ and its conclusion is $\Gamma \vdash xN \Rightarrow ([N/z]B)^{\beta}$. Thus the entire proof has the form satisfying (1)-(2), where the root is the conclusion of a fragment of the above form with a single application of (A-APP-OBJ).

In the case when the last step in the subproof of the left premise is (A-APP-OBJ), since we know that its proof has the form specified by (1)-(2) above, the conclusion of this subproof must have the form:

$$\Gamma \vdash xN_1 \ldots N_n \Rightarrow ([N_1/x_1, \ldots, N_n/x_n]B)^{\beta}$$

where $n \geq 1$. In fact, since it is followed by another application of (A-APP-OBJ), B must have the form $\Pi z : A.B'$. The type $([N_1/x_1, \ldots, N_n/x_n]\Pi z : A.B')^{\beta}$ is equal to:

$$\Pi z : ([N_1/x_1, \ldots, N_n/x_n]A)^{\beta}.([N_1/x_1, \ldots, N_n/x_n]B')^{\beta}$$

so the right premise of the last application of (A-APP-OBJ) must have the form

$$\Gamma \vdash N \Rightarrow ([N_1/x_1, \ldots, N_n/x_n]A)^\beta,$$

and the conclusion has the form

$$\Gamma \vdash xN_1 \ldots N_n N \Rightarrow ([N/z]([N_1/x_1, \ldots, N_n/x_n]B')^\beta)^\beta.$$

The left premise of the first application of (A-APP-OBJ) has the form

$$\Gamma \vdash x \Rightarrow \Pi x_1 : A_1 \ldots \Pi x_n : A_n . \Pi z : A . B'.$$

We can assume that $x_1, \ldots, x_n, z$ do not appear in $N_1, \ldots, N_n, N$. Otherwise, they can be renamed in the above assertion. Thus $([N/z]([N_1/x_1, \ldots, N_n/x_n]B')^\beta)^\beta$ is equal to $([N_1/x_1, \ldots, N_n/x_n, N/z]B')^\beta$. Hence the entire proof has the form satisfying (1)-(2), where the root is the conclusion of a series of $n+1$ applications of (A-APP-OBJ). ■

Based on the form of normal proofs described by Corollary 5.4, we define *Canonical* LF (abbreviated C-LF) to be the proof system obtained from LF by replacing rules (A-VAR-FAM) and (A-APP-FAM) with the (C-APP-FAM) rule in Figure 5.3, and replacing (A-VAR-OBJ) and (A-APP-OBJ) with the (C-APP-OBJ) rule in Figure 5.3. A type A is said to be a *base type* if its normal form is not an abstraction or a product.

Theorem 5.5

1. If $\Gamma \vdash K \Rightarrow$ kind in C-LF then Γ and K are canonical.
2. If $\Gamma \vdash A \Rightarrow K$ in C-LF then Γ, A, K are canonical.
3. If $\Gamma \vdash M \Rightarrow A$ in C-LF then Γ, M, A are canonical.

Proof: The proof is by induction on the height of the proof of the C-LF assertion, and uses the fact that if N is a canonical LF term, P is a canonical type or kind, and x is a term variable, then $([N/x]P)^\beta$ is canonical. Note that the head of the type or term in the conclusion of the (C-APP-FAM) and (C-APP-OBJ) rules is always fully applied. By inspection of the inference rules, it is easy to see that whenever the terms, types, and kinds in the premises are canonical, then the terms, types, and kinds in the conclusion must be canonical. ■

Theorem 5.6 (Soundness of C-LF)
If $\Gamma \vdash \alpha$ in C-LF, then $\Gamma \vdash \alpha$ in LF.

Proof: The proof is by induction on the height of the C-LF proof. All cases for the rules that occur in both C-LF and LF follow by a simple application of the induction hypothesis. If the last rule in the proof is (C-APP-OBJ), we can apply the induction hypothesis to obtain

$$\frac{\begin{array}{l} x : \Pi x_1 : A_1 \ldots \Pi x_n : A_n.\mathsf{Type} \in \Gamma \\ \Gamma \vdash \mathsf{Type} \Rightarrow \mathrm{kind} \\ \Gamma \vdash N_1 \Rightarrow A_1 \\ \Gamma \vdash N_2 \Rightarrow ([N_1/x_1]A_2)^\beta \\ \vdots \\ \Gamma \vdash N_n \Rightarrow ([N_1/x_1, \ldots, N_{n-1}/x_{n-1}]A_n)^\beta \end{array}}{\Gamma \vdash xN_1 \ldots N_n \Rightarrow \mathsf{Type}} \text{ (C-APP-FAM)}$$

$$\frac{\begin{array}{l} x : \Pi x_1 : A_1 \ldots \Pi x_n : A_n.B \in \Gamma \\ \Gamma \vdash \mathsf{Type} \Rightarrow \mathrm{kind} \\ \Gamma \vdash N_1 \Rightarrow A_1 \\ \Gamma \vdash N_2 \Rightarrow ([N_1/x_1]A_2)^\beta \\ \vdots \\ \Gamma \vdash N_n \Rightarrow ([N_1/x_1, \ldots, N_{n-1}/x_{n-1}]A_n)^\beta \end{array}}{\Gamma \vdash xN_1 \ldots N_n \Rightarrow ([N_1/x_1, \ldots, N_n/x_n]B)^\beta} \text{ (C-APP-OBJ)}$$

In (C-APP-OBJ) B is a base type, and in both rules $n \geq 0$.

Figure 5.3: Application Rules for C-LF

LF proofs of the premises. From these proofs, we can build a proof fragment in LF of the form described by (1) in Corollary 5.4 where the first two premises of (C-APP-OBJ) become the premises of the application of (A-VAR-OBJ), and the latter n premises become the n successive right premises of applications of (A-APP-OBJ). The case when the last rule in the proof is (C-APP-FAM) is similar. ■

Theorem 5.7 (Completeness of C-LF)
Let Γ be a canonical context.

1. If $\Gamma \vdash K \Rightarrow \mathrm{kind}$ in LF and K has a canonical form, then $\Gamma \vdash K^\beta \Rightarrow \mathrm{kind}$ in C-LF.
2. If $\Gamma \vdash A \Rightarrow K$ in LF and A and K have canonical forms, then $\Gamma \vdash A^\beta \Rightarrow K^\beta$ in C-LF.
3. If $\Gamma \vdash M \Rightarrow A$ in LF and M and A have canonical forms, then $\Gamma \vdash M^\beta \Rightarrow A^\beta$ in C-LF.

Proof: By Theorem 5.3, we know that there is a normal proof in LF of (1) $\Gamma \vdash K^\beta \Rightarrow \mathrm{kind}$, (2) $\Gamma \vdash A^\beta \Rightarrow K^\beta$, or (3) $\Gamma \vdash M^\beta \Rightarrow A^\beta$. Note that since M, A, K have canonical forms, $M^\beta, A^\beta, K^\beta$ are canonical. The proof is by induction on the height of such a normal proof. All cases for the rules that occur in both C-LF and LF follow by a simple application of the induction hypothesis. If the last rule in a normal proof of (3) is (A-VAR-OBJ) or

(A-APP-OBJ) its root occurs in a proof fragment of the form specified by Corollary 5.4 (1), and has the form:

$$\Gamma \vdash xN_1 \dots N_n \Rightarrow ([N_1/x_1, \dots, N_n/x_n]B)^\beta.$$

This fragment contains normal subproofs of the following assertions:

$\Gamma \vdash x \Rightarrow \Pi x_1 : A_1 \dots \Pi x_n : A_n.B$ where $x : \Pi x_1 : A_1 \dots \Pi x_n : A_n.B \in \Gamma$

$\Gamma \vdash \mathsf{Type} \Rightarrow \mathrm{kind}$

$\Gamma \vdash N_1 \Rightarrow A_1$

$\Gamma \vdash N_2 \Rightarrow ([N_1/x_1]A_2)^\beta$

$\vdots$

$\Gamma \vdash N_n \Rightarrow ([N_1/x_1, \dots, N_{n-1}/x_{n-1}]A_n)^\beta$

Since Γ is canonical, $\Pi x_1 : A_1 \dots \Pi x_n : A_n.B$ is a canonical type, and thus $A_1, \dots, A_n, B$ are canonical types. Since $xN_1 \dots N_n$ is a canonical term $N_1, \dots, N_n$ are canonical terms. Hence, $A_1, ([N_1/x_1]A_2)^\beta, \dots, ([N_1/x_1, \dots, N_{n-1}/x_{n-1}]A_n)^\beta$ are canonical types. Thus, we can apply the induction hypothesis to all of the assertions above to obtain C-LF proofs. Since $xN_1 \dots N_n$ is a canonical term that is not an abstraction, $([N_1/x_1, \dots, N_n/x_n]B)^\beta$ must be a base type. Thus we can apply (C-APP-OBJ) to these subproofs, and obtain a proof in C-LF of:

$$\Gamma \vdash xN_1 \dots N_n \Rightarrow ([N_1/x_1, \dots, N_n/x_n]B)^\beta.$$

The case when the last rule in a normal proof of (2) is (A-VAR-FAM) or (A-APP-FAM) is similar. ■

In the translation of LF signatures to logic programs, each signature item will be compiled into a definite clause. We will see that an application of (C-APP-OBJ) or (A-APP-FAM) in a C-LF proof corresponds to a BACKCHAIN on the corresponding program clause.

5.2 A Specification of β-Convertibility for LF

Several of the C-LF rules, including (C-APP-OBJ) and (C-APP-FAM) involve substitution and β-reduction. In the logic programs obtained from the translation of LF signatures, these operations will correspond to a need to execute a normalization program on the encoded LF objects. In this section, we represent LF terms, types, and kinds and specify β-conversion in much the same way that we represented terms and specified $\beta\eta$-conversion in Section 4.2 for the untyped λ-calculus. The proofs that the representation and specification are correct are analogous to the results in Section 4.3 for untyped terms. We then extend the specification of β-conversion to a program for β-normalization, and prove it correct.

We define an encoding between LF terms, types, and kinds, and terms in the simply-typed λ-calculus of type `tm`, `ty`, and `ki`, respectively. As in the encoding of Section 4.3, this encoding will be with respect to a *variable encoding*, which in this case will be a bijective

function between term variables and meta-variables of type `tm`, and type variables and meta-variables of type `ty`. A variable encoding ρ is well-defined on a term, type, or kind P if all of the free term and type variables in P are in $\mathrm{dom}(\rho)$ and are mapped to variables of type `tm` and `ty`, respectively. The encoding of P with respect to variable encoding ρ is denoted $\langle\!\langle P \rangle\!\rangle_\rho$ and defined in Figure 5.4. The types of the meta-constants used to

$$
\begin{array}{rcl}
\langle\!\langle x \rangle\!\rangle_\rho & := & \rho(x) \ \text{ for variable } x \in \mathrm{dom}(\rho) \\
\langle\!\langle \mathsf{Type} \rangle\!\rangle_\rho & := & \texttt{typ} \\
\langle\!\langle \Pi x : A.K \rangle\!\rangle_\rho & := & \texttt{(prod\_ki X\ } \langle\!\langle K \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \ \ \langle\!\langle A \rangle\!\rangle_\rho \texttt{)} \\
 & & \text{where X is a variable of type tm such that X} \notin \mathrm{cod}(\rho). \\
\langle\!\langle \Pi x : A.B \rangle\!\rangle_\rho & := & \texttt{(prod\_ty X\ } \langle\!\langle B \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \ \ \langle\!\langle A \rangle\!\rangle_\rho \texttt{)} \\
 & & \text{where X is a variable of type tm such that X} \notin \mathrm{cod}(\rho). \\
\langle\!\langle \lambda x : A.B \rangle\!\rangle_\rho & := & \texttt{(abs\_ty X\ } \langle\!\langle B \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \ \ \langle\!\langle A \rangle\!\rangle_\rho \texttt{)} \\
 & & \text{where X is a variable of type tm such that X} \notin \mathrm{cod}(\rho). \\
\langle\!\langle AM \rangle\!\rangle_\rho & := & \texttt{(app\_ty } \langle\!\langle A \rangle\!\rangle_\rho \ \ \langle\!\langle M \rangle\!\rangle_\rho \texttt{)} \\
\langle\!\langle \lambda x : A.M \rangle\!\rangle_\rho & := & \texttt{(abs\_tm X\ } \langle\!\langle M \rangle\!\rangle_{\langle x,\texttt{X}\rangle+\rho} \ \ \langle\!\langle A \rangle\!\rangle_\rho \texttt{)} \\
 & & \text{where X is a variable of type tm such that X} \notin \mathrm{cod}(\rho). \\
\langle\!\langle MN \rangle\!\rangle_\rho & := & \texttt{(app\_tm } \langle\!\langle M \rangle\!\rangle_\rho \ \ \langle\!\langle N \rangle\!\rangle_\rho \texttt{)}
\end{array}
$$

Figure 5.4: Encoding of LF Terms

represent abstraction, products, and application are given in the `lfsig` module below. Also included in this module are the declarations for two predicates which will be used later in the translation of LF signatures to definite clauses.

```
module lfsig.

kind    ki              type.
kind    ty              type.
kind    tm              type.

type    typ             ki.
type    prod_ki         (tm -> ki) -> ty -> ki.
type    prod_ty         (tm -> ty) -> ty -> ty.
type    abs_ty          (tm -> ty) -> ty -> ty.
type    abs_tm          (tm -> tm) -> ty -> tm.
type    app_ty          ty -> tm -> ty.
type    app_tm          tm -> tm -> tm.

type    is_type         ty -> o.
type    has_type        tm -> ty -> o.
```

Module `lfsig`: Signature for LF Terms, Types, and Kinds

Similarly, we can define a decoding as we did in Section 4.3 for untyped λ-terms. Again

the decoding is with respect to a *variable decoding*, which in this case is a bijective function between meta-variables of type `tm` and term variables, and meta-variables of type `ty` and type variables. A variable decoding $\bar{\rho}$ is well-defined on a term `P` of type `tm`, `ty`, or `ki` if all of the free variables of type `tm` and `ty` in `P` are in $\mathrm{dom}(\bar{\rho})$ and are mapped to term and type variables, respectively. The decoding of `P` with respect to variable decoding $\bar{\rho}$ is denoted $\|\mathtt{P}\|_{\bar{\rho}}$ and defined in Figure 5.5.

$$
\begin{array}{rcl}
\|\mathtt{X}\|_{\bar{\rho}} & := & \bar{\rho}(\mathtt{X}) \ \text{ for variable } \mathtt{X} \in \mathrm{dom}(\bar{\rho}) \\
\|\mathtt{typ}\|_{\bar{\rho}} & := & \mathsf{Type} \\
\|\texttt{(prod\_ki P A)}\|_{\bar{\rho}} & := & \Pi x : \|\mathtt{A}\|_{\bar{\rho}}.\|\mathtt{K}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}} \\
 & & \text{where } \mathtt{X} \text{ is the bound variable in } \mathtt{P}, \mathtt{K} \text{ is the body, and} \\
 & & x \text{ is a term variable such that } x \notin \mathrm{cod}(\bar{\rho}). \\
\|\texttt{(prod\_ty P A)}\|_{\bar{\rho}} & := & \Pi x : \|\mathtt{A}\|_{\bar{\rho}}.\|\mathtt{B}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}} \\
 & & \text{where } \mathtt{X} \text{ is the bound variable in } \mathtt{P}, \mathtt{B} \text{ is the body, and} \\
 & & x \text{ is a term variable such that } x \notin \mathrm{cod}(\bar{\rho}). \\
\|\texttt{(abs\_ty P A)}\|_{\bar{\rho}} & := & \lambda x : \|\mathtt{A}\|_{\bar{\rho}}.\|\mathtt{B}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}} \\
 & & \text{where } \mathtt{X} \text{ is the bound variable in } \mathtt{P}, \mathtt{B} \text{ is the body, and} \\
 & & x \text{ is a term variable such that } x \notin \mathrm{cod}(\bar{\rho}). \\
\|\texttt{(app\_ty A M)}\|_{\bar{\rho}} & := & \|\mathtt{A}\|_{\bar{\rho}} \ \ \|\mathtt{M}\|_{\bar{\rho}} \\
\|\texttt{(abs\_tm P A)}\|_{\bar{\rho}} & := & \lambda x : \|\mathtt{A}\|_{\bar{\rho}}.\|\mathtt{M}\|_{\langle \mathtt{X},x\rangle+\bar{\rho}} \\
 & & \text{where } \mathtt{X} \text{ is the bound variable in } \mathtt{P}, \mathtt{M} \text{ is the body, and} \\
 & & x \text{ is a term variable such that } x \notin \mathrm{cod}(\bar{\rho}). \\
\|\texttt{(app\_tm M N)}\|_{\bar{\rho}} & := & \|\mathtt{M}\|_{\bar{\rho}} \ \ \|\mathtt{N}\|_{\bar{\rho}}
\end{array}
$$

Figure 5.5: Decoding of LF Terms

We need several other concepts analogous to those in Section 4.3. Given a set of LF term and type variables $\mathcal{V}$, we denote the set of LF terms, types, and kinds whose free variables are in $\mathcal{V}$ as $LF(\mathcal{V})$. At the meta-level, given a set of variables $\mathcal{V}$ of type `tm` and `ty`, we denote the set of terms of type `tm`, `ty`, and `ki` in $\beta\eta$-long normal form built up from the constants in the `lfsig` module, whose free variables are in $\mathcal{V}$, as $T(\mathcal{V})$.

We will again need object variable and meta-variable substitutions. Here object variable substitutions will contain only term variables, and meta-variable substitutions will always be `tm`-substitutions. Recall the following definitions. Given an object variable substitution σ, if ρ_1 is a variable encoding well-defined on $\mathrm{dom}(\sigma)$ and ρ_2 is a variable encoding well-defined on $\mathrm{cod}(\sigma)$, then σ_{ρ_1,ρ_2} denotes the following `tm`-substitution.

$$\sigma_{\rho_1,\rho_2} = \{\langle \rho_1(x), \langle\!\langle P \rangle\!\rangle_{\rho_2} \rangle | \langle x, P\rangle \in \sigma\}$$

Given a `tm`-substitution σ, if $\bar{\rho}_1$ is a variable decoding well-defined on $\mathrm{dom}(\sigma)$ and $\bar{\rho}_2$ is a variable decoding well-defined on $\mathrm{cod}(\sigma)$, then $\sigma_{\bar{\rho}_1,\bar{\rho}_2}$ denotes the following object variable

substitution.

$$\sigma_{\bar{\rho}_1,\bar{\rho}_2} = \{\langle \bar{\rho}_1(\mathtt{X}), \|\mathtt{P}\|_{\bar{\rho}_2}\rangle | \langle \mathtt{X}, \mathtt{P}\rangle \in \sigma\}$$

We state the following results for the encoding and decoding which correspond to Lemma 4.1, Lemma 4.3, Corollary 4.4, Corollary 4.5, Lemma 4.6, Lemma 4.8, Corollary 4.9, Corollary 4.10, and Theorem 4.11 in Section 4.3. As in that section, we write explicitly $P =_\alpha Q$ when two terms, types, or kinds are α-convertible, and similarly for terms at the meta-level. We omit the proofs since they are analogous to the corresponding proofs for the encoding and decoding of untyped λ-terms in Section 4.3. The main differences are that now we have three classes of objects at the object level, terms of three primitive types at the meta-level, and two kinds of binding operators in the object language: products and abstractions. It is easy to see that the encoding maps kinds to meta-terms of type `ki`, types to meta-terms of type `ty`, and terms to meta-terms of type `tm`, and conversely for the decoding.

Lemma 5.8 Let P be an LF term, type, or kind, and ρ a variable encoding that is well-defined on P. Then $\langle\!\langle P \rangle\!\rangle_\rho$ is a term in $T(\mathrm{cod}(\rho))$, *i.e.*, a term of type `tm`, `ty`, or `ki`, respectively, in $\beta\eta$-long form whose free variables are in $\mathrm{cod}(\rho)$.

Lemma 5.9 Let σ be an object variable substitution, and let P and Q be LF terms, types, or kinds such that $\sigma(P) =_\alpha Q$. Let ρ_1 be a variable encoding well-defined on P and $\mathrm{dom}(\sigma)$, and ρ_2 a variable encoding well-defined on Q and $\mathrm{cod}(\sigma)$ such that ρ_1 and ρ_2 agree on common domain elements that are free in both $\sigma(P)$ and Q. Then

$$\sigma_{\rho_1,\rho_2}(\langle\!\langle P \rangle\!\rangle_{\rho_1}) =_\alpha \langle\!\langle Q \rangle\!\rangle_{\rho_2}.$$

Corollary 5.10 (Well-Definedness of Encoding for LF Terms, Types, and Kinds)
Let P and Q be two LF terms, types, or kinds such that $P =_\alpha Q$. Let ρ_1 and ρ_2 be two variable encodings that are well-defined on P and Q, and that agree on the free variables of P and Q. Then $\langle\!\langle P \rangle\!\rangle_{\rho_1} =_\alpha \langle\!\langle Q \rangle\!\rangle_{\rho_2}$.

Corollary 5.11 (Substitution Commutes with Encoding)
Let N be an LF term, P an LF term, type, or kind, and x a term variable. Let ρ_1 be a variable encoding well-defined on P and x, and ρ_2 a variable encoding well-defined on $[N/x]P$ and N such that ρ_1 and ρ_2 agree on common domain elements free in $[N/x]P$. Then

$$[\langle\!\langle N \rangle\!\rangle_{\rho_2}/\langle\!\langle x \rangle\!\rangle_{\rho_1}]\langle\!\langle P \rangle\!\rangle_{\rho_1} =_\alpha \langle\!\langle [N/x]P \rangle\!\rangle_{\rho_2}.$$

Lemma 5.12 Let $\mathtt{P}$ be a term of type $\mathtt{tm}$, $\mathtt{ty}$, or $\mathtt{ki}$ in $\beta\eta$-long form, and $\bar{\rho}$ a variable decoding that is well-defined on $\mathtt{P}$. Then $\|\mathtt{P}\|_{\bar{\rho}}$ is a term in $LF(\mathrm{cod}(\bar{\rho}))$, *i.e.*, a term, type, or kind, respectively, whose free variables are in $\mathrm{cod}(\bar{\rho})$.

Lemma 5.13 Let σ be a $\mathtt{tm}$-substitution, and let $\mathtt{P}$ and $\mathtt{Q}$ be terms of type $\mathtt{tm}$, $\mathtt{ty}$, or $\mathtt{ki}$ in $\beta\eta$-long form such that $\sigma(\mathtt{P}) =_\alpha \mathtt{Q}$. Let $\bar{\rho}_1$ be a variable decoding well-defined on $\mathtt{P}$ and $\mathrm{dom}(\sigma)$ and $\bar{\rho}_2$ a variable decoding well-defined on $\mathtt{Q}$ and $\mathrm{cod}(\sigma)$ such that $\bar{\rho}_1$ and $\bar{\rho}_2$ agree on common domain elements that are free in both $\sigma(\mathtt{P})$ and $\mathtt{Q}$. Then

$$\sigma_{\bar{\rho}_1,\bar{\rho}_2}(\|\mathtt{P}\|_{\bar{\rho}_1}) =_\alpha \|\mathtt{Q}\|_{\bar{\rho}_2}.$$

Corollary 5.14 (Well-Definedness of Decoding for LF Terms, Types, and Kinds)
Let $\mathtt{P}$ and $\mathtt{Q}$ be terms of type $\mathtt{tm}$, $\mathtt{ty}$, or $\mathtt{ki}$ in $\beta\eta$-long form such that $\mathtt{P} =_\alpha \mathtt{Q}$. Let $\bar{\rho}_1$ and $\bar{\rho}_2$ be two variable decodings that are well-defined on $\mathtt{P}$ and $\mathtt{Q}$, and that agree on the free variables of $\mathtt{P}$ and $\mathtt{Q}$. Then $\|\mathtt{P}\|_{\bar{\rho}_1} =_\alpha \|\mathtt{Q}\|_{\bar{\rho}_2}$.

Corollary 5.15 (Substitution Commutes with Decoding)
Let $\mathtt{N}$ be a term of type $\mathtt{tm}$ in $\beta\eta$-long form, $\mathtt{P}$ a term of type $\mathtt{tm}$, $\mathtt{ty}$, or $\mathtt{ki}$ in $\beta\eta$-long form, and $\mathtt{X}$ a variable of type $\mathtt{tm}$. Let $\bar{\rho}_1$ be a variable decoding well-defined on $\mathtt{P}$ and $\mathtt{X}$, and $\bar{\rho}_2$ a variable decoding well-defined on $[\mathtt{N}/\mathtt{X}]\mathtt{P}$ and $\mathtt{N}$ such that $\bar{\rho}_1$ and $\bar{\rho}_2$ agree on common domain elements free in $[\mathtt{N}/\mathtt{X}]\mathtt{P}$. Then

$$[\|\mathtt{N}\|_{\bar{\rho}_2}/\|\mathtt{X}\|_{\bar{\rho}_1}]\|\mathtt{P}\|_{\bar{\rho}_1} =_\alpha \|[\mathtt{N}/\mathtt{X}]\mathtt{P}\|_{\bar{\rho}_2}.$$

Theorem 5.16 (Correctness of Encoding and Decoding of LF Terms, Types, and Kinds)
Let ρ be a variable encoding. The encoding $\langle\!\langle\rangle\!\rangle_\rho$ is a bijection from the α-equivalence classes of $LF(\mathrm{dom}(\rho))$ to the α-equivalence classes of $T(\mathrm{cod}(\rho))$, *i.e.*, a bijective mapping from sets of LF terms, types, and kinds with free variables in $\mathrm{dom}(\rho)$, to sets of simply-typed terms of type $\mathtt{tm}$, $\mathtt{ty}$, and $\mathtt{ki}$, respectively, in $\beta\eta$-long form whose free variables are in $\mathrm{cod}(\rho)$. Furthermore, the decoding $\|\|_{\rho^{-1}}$ is the inverse of $\langle\!\langle\rangle\!\rangle_\rho$.

The `lfconv` module on page 85 specifies β-convertibility for LF as given by the proof system in Figure 5.1. The clauses are similar to those in the `convert` module on page 46 specifying $\beta\eta$-convertibility for the untyped λ-calculus, and so we do not discuss it further here.

```
module lfconv.

import lfsig.

type    redex_ty        ty -> ty -> o.
type    redex_tm        tm -> tm -> o.
type    red1_ki         ki -> ki -> o.
type    red1_ty         ty -> ty -> o.
type    red1_tm         tm -> tm -> o.
type    conv_ki         ki -> ki -> o.
type    conv_ty         ty -> ty -> o.
type    conv_tm         tm -> tm -> o.

redex_ty (app_ty (abs_ty B A) N) (B N).
redex_tm (app_tm (abs_tm M A) N) (M N).

red1_ki (prod_ki K1 A) (prod_ki K2 A) :- pi X\ (red1_ki (K1 X) (K2 X)).
red1_ki (prod_ki K A1) (prod_ki K A2) :- red1_ty A1 A2.

red1_ty A1 A2 :- redex_ty A1 A2.
red1_ty (prod_ty B1 A) (prod_ty B2 A) :- pi X\ (red1_ty (B1 X) (B2 X)).
red1_ty (prod_ty B A1) (prod_ty B A2) :- red1_ty A1 A2.
red1_ty (abs_ty B1 A) (abs_ty B2 A) :- pi X\ (red1_ty (B1 X) (B2 X)).
red1_ty (abs_ty B A1) (abs_ty B A2) :- red1_ty A1 A2.
red1_ty (app_ty A1 N) (app_ty A2 N) :- red1_ty A1 A2.
red1_ty (app_ty A N1) (app_ty A N2) :- red1_tm N1 N2.

red1_tm M1 M2 :- redex_tm M1 M2.
red1_tm (abs_tm M1 A) (abs_tm M2 A) :- pi X\ (red1_tm (M1 X) (M2 X)).
red1_tm (abs_tm M A1) (abs_tm M A2) :- red1_ty A1 A2.
red1_tm (app_tm M1 N) (app_tm M2 N) :- red1_tm M1 M2.
red1_tm (app_tm M N1) (app_tm M N2) :- red1_tm N1 N2.

conv_ki K K.
conv_ki K L :- conv_ki L K.
conv_ki K L :- conv_ki K Q, conv_ki Q L.
conv_ki K L :- red1_ki K L.

conv_ty A A.
conv_ty A B :- conv_ty B A.
conv_ty A B :- conv_ty A C, conv_ty C B.
conv_ty A B :- red1_ty A B.

conv_tm M M.
conv_tm M N :- conv_tm N M.
conv_tm M N :- conv_tm M P, conv_tm P N.
conv_tm M N :- red1_tm M N.
```

Module `lfconv`: β-Convertibility in LF

```
module lfnorm.

import lfconv.

type    normal_ki       ki -> o.
type    normal_ty       ty -> o.
type    normal_ty1      ty -> o.
type    normal_ty2      ty -> o.
type    normal_tm       tm -> o.
type    normal_tm1      tm -> o.
type    norm_ki         ki -> ki -> o.
type    norm_ty         ty -> ty -> o.
type    norm_tm         tm -> tm -> o.

normal_ki typ.
normal_ki (prod_ki K A) :- normal_ty A, pi X\ (normal_tm1 X => normal_ki (K X)).

normal_ty (abs_ty B A) :- normal_ty A, pi X\ (normal_tm1 X => normal_ty (B X)).
normal_ty A :- normal_ty1 A.

normal_ty1 (prod_ty B A) :- normal_ty A,
                            pi X\ (normal_tm1 X => normal_ty1 (B X)).
normal_ty1 A :- normal_ty2 A.

normal_ty2 (app_ty A N) :- normal_ty2 A, normal_tm N.

normal_tm (abs_tm M A) :- normal_ty A, pi X\ (normal_tm1 X => normal_tm (M X)).
normal_tm M :- normal_tm1 M.

normal_tm1 (app_tm M N) :- normal_tm1 M, normal_tm N.

norm_tm M N :- conv_tm M N, normal_tm N.
norm_ty A B :- conv_ty A B, normal_ty B.
norm_ki K L :- conv_ki K L, normal_ki L.
```

Module `lfnorm`: β-Normalization in LF

The `lfnorm` module above specifies β-normalization for terms, types, and kinds. The last three clauses are the top-level clauses of the normalization programs for each class of objects. Consider the following clause from that module, which specifies normalization for LF terms.

```
norm_tm M N :- conv_tm M N, normal_tm N.
```

Its declarative reading is that `N` is the normal form of `M` if `M` converts to `N`, and `N` is in normal form. Operationally, this clause makes use of the `conv_tm` program in the `lfconv` module, and then examines the form of `N` to see if it is normal. The `normal_tm` program is specified by the following clauses.

```
normal_tm (abs_tm M A) :- normal_ty A, pi X\ (normal_tm1 X => normal_tm (M X)).
```

```
normal_tm M :- normal_tm1 M.

normal_tm1 (app_tm M N) :- normal_tm1 M, normal_tm N.
```

It uses two predicates: `normal_tm` and `normal_tm1`. A goal of the form `(normal_tm M)` succeeds if `M` is normal, while a goal of the form `(normal_tm1 M)` succeeds under slightly stronger conditions: `M` must be a normal term that contains no outermost abstractions. Recall that a normal LF term has the form $\lambda x_1 : A_1 \ldots \lambda x_n : A_n . x M_1 \ldots M_k$ where $n, k \geq 0$, for $i = 1, \ldots, n$, A_i is a normal form type, x is a term variable, and for $i = 1, \ldots, k$, M_i is a normal form term. The above program reflects this characterization. Consider the declarative reading of the first clause. An abstraction `(abs_tm M A)` is normal if `A` is a normal type, and for arbitrary term `X`, if `X` is a normal term that contains no outer abstractions, then the term resulting from substituting the bound variable in `M` with `X` is normal. Operationally, the GENERIC search operation introduces a new signature item of type `tm`, and the AUGMENT search operation adds the assumption that this new item is a normal term that contains no abstractions. β-conversion at the meta-level is used to replace the bound variable in `M` with the new signature item. The subgoal `(normal_tm (M X))` ensures that this subterm in normal. The second `normal_tm` clause states that a normal term with no outer abstractions is normal, and the last clause reads that an application is normal if the first term is normal and has no outer abstractions, and the second term is an arbitrary normal term. Operationally, the `normal_tm` program strips off outer abstractions, while the `normal_tm1` program handles the inner applications and variables. The `normal_ty` and `normal_ki` programs are similar. The `normal_ty` program requires two auxiliary predicates. An atomic formula of the form `(normal_ty1 A)` states that `A` is a normal type that is not an abstraction, while a formula of the form `(normal_ty2 A)` states that `A` is a normal type that is not an abstraction or product.

Correctness results similar to Theorem 4.12, Lemma 4.13, Corollary 4.14, and Theorem 4.15 hold for the `conv_ki`, `conv_ty`, and `conv_tm` programs. The theorems below extend these results to the `norm_ki`, `norm_ty`, and `norm_tm` programs. In these theorems, we take Σ_0 to be the set of declarations of constants and their types that appears in the `lfsig`, `lfconv`, and `lfnorm` modules, and $\mathcal{P}_0$ to be the set of clauses in `lfconv` and `lfnorm`. We state the analogue of Lemma 4.13 (1) since it will be needed below. Its proof is similar to the proof of Lemma 4.13 (1) and so is not repeated. Here, $\rightarrow_\beta^*$ is the reflexive transitive closure of $\rightarrow_\beta$.

Lemma 5.17 Let ρ be a variable encoding and P and Q LF terms, types, or kinds in $LF(\mathrm{dom}(\rho))$ such that $P \rightarrow_\beta^* Q$. Let Σ be the signature $\Sigma_0 \cup \mathrm{cod}(\rho)$.

1. If P and Q are terms, then $\Sigma; \texttt{lfconv} \vdash_I \texttt{conv\_tm}\ \langle\!\langle P \rangle\!\rangle_\rho\ \langle\!\langle Q \rangle\!\rangle_\rho$ holds.
2. If P and Q are types, then $\Sigma; \texttt{lfconv} \vdash_I \texttt{conv\_ty}\ \langle\!\langle P \rangle\!\rangle_\rho\ \langle\!\langle Q \rangle\!\rangle_\rho$ holds.
3. If P and Q are kinds, then $\Sigma; \texttt{lfconv} \vdash_I \texttt{conv\_ki}\ \langle\!\langle P \rangle\!\rangle_\rho\ \langle\!\langle Q \rangle\!\rangle_\rho$ holds.

We will need the following definition in the proofs below. Given a set $\mathcal{V}$ of variables of type `tm` and `ty`, we define $\mathcal{P}_\mathcal{V}$ to be the following set of clauses:

$$\mathcal{P}_\mathcal{V} := \{\texttt{normal\_ty2 X}|\texttt{X} : \texttt{ty} \in \mathcal{V}\} \cup \{\texttt{normal\_tm1 X}|\texttt{X} : \texttt{tm} \in \mathcal{V}\}$$

These clauses state that a variable of type `ty` represents an LF type in normal form that is not an abstraction or product, and a variable of type `tm` represents an LF term in normal form that is not an abstraction. These clauses will be needed in checking whether or not terms that contain these variable are in normal form.

Theorem 5.18 (Correctness I)

Let ρ be a variable encoding and P and Q LF terms, types, or kinds in $LF(\mathrm{dom}(\rho))$ such that $P =_\beta Q$, and Q is β-normal. Let Σ be the signature $\Sigma_0 \cup \mathrm{cod}(\rho)$, and let $\mathcal{P}$ be the program $\mathcal{P}_0 \cup \mathcal{P}_{\mathrm{cod}(\rho)}$.

1. If P and Q are terms, then $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_tm } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho$ holds.
2. If P and Q are types, then $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ty } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho$ holds.
3. If P and Q are kinds, then $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ki } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho$ holds.

Proof: (1) The judgment $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_tm } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho$ holds if by BACKCHAIN and AND search, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \texttt{conv\_tm } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho \tag{1}$$

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_tm } \langle\!\langle Q \rangle\!\rangle_\rho \tag{2}$$

(2) The judgment $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ty } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho$ holds if by BACKCHAIN and AND search, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \texttt{conv\_ty } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho \tag{3}$$

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ty } \langle\!\langle Q \rangle\!\rangle_\rho \tag{4}$$

(3) The judgment $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ki } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho$ holds if by BACKCHAIN and AND search, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \texttt{conv\_ki } \langle\!\langle P \rangle\!\rangle_\rho \ \langle\!\langle Q \rangle\!\rangle_\rho \tag{5}$$

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ki } \langle\!\langle Q \rangle\!\rangle_\rho \tag{6}$$

Since Q is normal, $P \rightarrow_\beta^* Q$. Thus (1), (3), or (5) holds by Lemma 5.17. We prove (2), (4), or (6) by simultaneous induction on the structure of Q. We show the cases when Q is a type. The cases when when Q is a term or kind are similar.

Base: Q is a type variable. (4) holds if by BACKCHAIN on the second clause for `normal_ty` followed by BACKCHAIN on the second clause for `normal_ty1`, the following judgment holds.

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ty2 } \langle\!\langle Q \rangle\!\rangle_\rho$$

Since Q is a variable, Q must be in $\mathrm{dom}(\rho)$, and thus $\langle\!\langle Q \rangle\!\rangle_\rho \equiv \rho(Q)$. The term $\rho(Q)$ is a variable of type `ty`, and the clause `(normal_ty2` $\rho(Q)$`)` is in $\mathcal{P}_{\mathrm{cod}(\rho)}$, so the above judgment holds.

Case: Q has the form $\Pi x : A.B$. Let `X` be a variable of type `ty` that does not appear $\mathrm{cod}(\rho)$. By the definition of the encoding $\langle\!\langle \Pi x : A.B \rangle\!\rangle_\rho \equiv$ `(prod_ty X\`$\langle\!\langle B \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$ $\langle\!\langle A \rangle\!\rangle_\rho$`)`. We must show that the following judgment holds.

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ty (prod\_ty X\}\langle\!\langle B \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho} \ \langle\!\langle A \rangle\!\rangle_\rho\texttt{)}$$

This judgment holds if by BACKCHAIN the following judgment holds.

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ty1 (prod\_ty X\}\langle\!\langle B \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho} \ \langle\!\langle A \rangle\!\rangle_\rho\texttt{)}$$

The above judgment holds if by BACKCHAIN, and AND search, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ty } \langle\!\langle A \rangle\!\rangle_\rho$$

$$\Sigma; \mathcal{P} \vdash_I \texttt{pi X\(normal\_ty X => normal\_ty1 (X\}}\langle\!\langle B \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}\texttt{ X))}$$

The first judgment holds by the induction hypothesis. Since $\mathtt{X} \notin \mathrm{cod}(\rho)$, `X` is not in Σ. The second judgment holds if by GENERIC followed by AUGMENT, the following judgment holds.

$$\Sigma \cup \{\mathtt{X} : \mathtt{tm}\}; \mathcal{P} \cup \{\texttt{normal\_tm1 X}\} \vdash_I \texttt{normal\_ty1 (X\}}\langle\!\langle B \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}\texttt{ X)} \tag{7}$$

By β-conversion at the meta-level, `(X\`$\langle\!\langle B \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$ `X)` $\equiv \langle\!\langle B \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$. By the induction hypothesis, the following judgment holds.

$$\Sigma \cup \{\mathtt{X} : \mathtt{tm}\}; \mathcal{P} \cup \{\texttt{normal\_tm1 X}\} \vdash_I \texttt{normal\_ty } \langle\!\langle B \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$$

Since Q is normal, we know that B is not an abstraction. Thus the constant at the head of the term $\langle\!\langle B \rangle\!\rangle_{\langle x,\mathtt{X}\rangle+\rho}$ is not `abs_ty`. Thus the last step in a proof of the latter judgment must be a BACKCHAIN on the second clause for `normal_ty`. Hence (7) holds.

Case: Q has the form $\lambda x : A.B$. This case is similar to the previous one.

Case: Q has the form AM. By the definition of the encoding

$$\langle\!\langle AM \rangle\!\rangle_\rho \equiv \texttt{(app\_ty } \langle\!\langle A \rangle\!\rangle_\rho \ \ \langle\!\langle M \rangle\!\rangle_\rho\texttt{)}.$$

We must show that the following judgment holds.

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ty (app\_ty } \langle\!\langle A \rangle\!\rangle_\rho \ \ \langle\!\langle M \rangle\!\rangle_\rho\texttt{)}$$

This judgment holds if by a BACKCHAIN on the second clause for `normal_ty`, then a BACKCHAIN on the second clause for `normal_ty1`, then a BACKCHAIN on the first clause for `normal_ty2`, followed by AND search, the following judgments hold.

$$\begin{aligned} &\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ty2 } \langle\!\langle A \rangle\!\rangle_\rho \\ &\Sigma; \mathcal{P} \vdash_I \texttt{normal\_tm } \langle\!\langle M \rangle\!\rangle_\rho \end{aligned} \tag{8}$$

The latter judgment holds by the induction hypothesis. The following judgment also holds by the induction hypothesis.

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ty}\ \langle\!\langle A \rangle\!\rangle_\rho$$

Since AM is normal, we know that A is not an abstraction or product. Thus the constant at the head of the term $\langle\!\langle A \rangle\!\rangle_\rho$ is not `abs_ty` or `prod_ty`. Thus the last steps in a proof of the former judgment must be a BACKCHAIN on the second clause for `normal_ty`, followed by a BACKCHAIN on the second clause for `normal_ty1`. Hence (8) holds. ■

Theorem 5.19 (Correctness II)

Let $\bar{\rho}$ be a variable decoding and `P` and `Q` terms in $T(\mathrm{dom}(\bar{\rho}))$. Let Σ be the signature $\Sigma_0 \cup \mathrm{dom}(\bar{\rho})$, and let $\mathcal{P}$ be the program $\mathcal{P}_0 \cup \mathcal{P}_{\mathrm{dom}(\bar{\rho})}$. If one of the following judgments hold,

1. $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_tm P Q}$
2. $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ty P Q}$
3. $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ki P Q}$

then $\|\texttt{P}\|_\rho =_\beta \|\texttt{Q}\|_\rho$ and $\|\texttt{Q}\|_\rho$ is β-normal.

Proof: (1) By BACKCHAIN, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \texttt{conv\_tm P Q} \tag{9}$$

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_tm Q} \tag{10}$$

(2) By BACKCHAIN, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \texttt{conv\_ty P Q} \tag{11}$$

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ty Q} \tag{12}$$

(3) By BACKCHAIN, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \texttt{conv\_ki P Q} \tag{13}$$

$$\Sigma; \mathcal{P} \vdash_I \texttt{normal\_ki Q} \tag{14}$$

From (9), (11), or (13) we can show that $\|\texttt{P}\|_{\bar{\rho}} =_\beta \|\texttt{Q}\|_{\bar{\rho}}$ by a proof similar to Theorem 4.15. To show that $\|\texttt{Q}\|_{\bar{\rho}}$ is a normal form, we prove the following statements by simultaneous induction on the height of the meta-proofs of (10), (12), and (14).

1. If (10) holds, then Q is a normal term.
2. If (12) holds, then Q is a normal type.
3. If (14) holds, then Q is a normal kind.

4. If $\Sigma; \mathcal{P} \vdash_I$ `normal_tm1 Q` holds, then Q is a normal term that is a not an abstraction.
5. If $\Sigma; \mathcal{P} \vdash_I$ `normal_ty1 Q` holds, then Q is a normal type that is a not an abstraction.
6. If $\Sigma; \mathcal{P} \vdash_I$ `normal_ty2 Q` holds, then Q is a normal type that is a not an abstraction or product.

We show the cases for (5) and (6). The others are similar.

Base: $\Sigma; \mathcal{P} \vdash_I$ `normal_ty2 Q` is provable in one step. Then the clause `(normal_ty2 Q)` is in $\mathcal{P}_{\mathrm{dom}(\bar{\rho})}$. Thus `Q` is in $\mathrm{dom}(\bar{\rho})$. Since `Q` has type `ty`, $\|\mathtt{Q}\|_{\bar{\rho}}$ is a type variable, and hence `Q` is in normal form and is not an abstraction or product.

Case: The last step in a proof of $\Sigma; \mathcal{P} \vdash_I$ `normal_ty2 Q` is a BACKCHAIN on the clause in `lfnorm` for `normal_ty2`. Then `Q` has the form `(app_ty A N)`. By the decoding $\|\mathtt{(app\_ty\ A\ N)}\|_{\bar{\rho}} \equiv \|\mathtt{A}\|_{\bar{\rho}}\ \|\mathtt{N}\|_{\bar{\rho}}$. By BACKCHAIN, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \mathtt{normal\_ty2\ A}$$

$$\Sigma; \mathcal{P} \vdash_I \mathtt{normal\_tm\ N}$$

By the induction hypothesis $\|\mathtt{A}\|_{\bar{\rho}}$ is a normal type that is not an abstraction or product, and $\|\mathtt{N}\|_{\bar{\rho}}$ is normal term. Thus $\|\mathtt{A}\|_{\bar{\rho}}\ \|\mathtt{N}\|_{\bar{\rho}}$ is a normal type.

Case: The last step in a proof of $\Sigma; \mathcal{P} \vdash_I$ `normal_ty1 Q` is a BACKCHAIN on the second clause for `normal_ty1`. Thus $\Sigma; \mathcal{P} \vdash_I$ `normal_ty2 Q` holds. By the induction hypothesis, $\|\mathtt{Q}\|_{\bar{\rho}}$ is normal type that is not an abstraction or product.

Case: The last step in a proof of $\Sigma; \mathcal{P} \vdash_I$ `normal_ty1 Q` is a BACKCHAIN on the first clause for `normal_ty1`. Then `Q` has the form `(prod_ty B A)`, where `X` is the bound variable in `B` and `C` is the body. We assume that `X` does not appear in Σ, otherwise we rename it. Let x be a variable that does not occur in $\mathrm{cod}(\bar{\rho})$. By the definition of the decoding $\|\mathtt{(prod\_ty\ B\ A)}\|_{\bar{\rho}} \equiv \Pi x : \|\mathtt{A}\|_{\bar{\rho}}.\|\mathtt{C}\|_{\langle \mathtt{X}, x\rangle + \bar{\rho}}$. By BACKCHAIN and AND search, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \mathtt{normal\_ty\ A}$$

$$\Sigma; \mathcal{P} \vdash_I \mathtt{pi\ X\backslash(normal\_tm1\ X\ =>\ normal\_ty1\ (B\ X))}$$

By the induction hypothesis $\|\mathtt{A}\|_{\bar{\rho}}$ is a normal type. By GENERIC followed by AUGMENT on the latter judgment, the following holds.

$$\Sigma \cup \{\mathtt{X} : \mathtt{tm}\}; \mathcal{P} \cup \{\mathtt{normal\_tm1\ X}\} \vdash_I \mathtt{normal\_ty1\ (B\ X)}$$

By β-conversion at the meta-level $(\mathtt{B\ X}) =_{\beta\eta} \mathtt{C}$. Thus by the induction hypothesis, $\|\mathtt{C}\|_{\langle \mathtt{X}, x\rangle + \bar{\rho}}$ is a normal type that is not an abstraction. Thus $\Pi x : \|\mathtt{A}\|_{\bar{\rho}}.\|\mathtt{C}\|_{\langle \mathtt{X}, x\rangle + \bar{\rho}}$ is a normal type. ■

Corollary 5.20 Let ρ be a variable encoding. Let P be an LF term, type, or kind in $LF(\mathrm{dom}(\rho))$, and `P` a term of type `tm`, `ty`, or `ki` in $T(\mathrm{cod}(\rho))$. Let Σ be the signature $\Sigma_0 \cup \mathrm{cod}(\rho)$, and let $\mathcal{P}$ be the program $\mathcal{P}_0 \cup \mathcal{P}_{\mathrm{cod}(\rho)}$. If one of the following judgments hold,

1. $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_tm}\ \langle\!\langle P \rangle\!\rangle_\rho\ \texttt{P}$
2. $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ty}\ \langle\!\langle P \rangle\!\rangle_\rho\ \texttt{P}$
3. $\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ki}\ \langle\!\langle P \rangle\!\rangle_\rho\ \texttt{P}$

then P is $\langle\!\langle P^\beta \rangle\!\rangle_\rho$.

Proof: By Theorem 5.16, showing that P is $\langle\!\langle P^\beta \rangle\!\rangle_\rho$ is equivalent to showing that $\|\texttt{P}\|_{\rho^{-1}}$ is P^β. By Lemma 5.8, $\langle\!\langle P \rangle\!\rangle_\rho$ is in $T(\mathrm{cod}(\rho))$. Thus we can apply Theorem 5.19 to obtain the fact that $\|\texttt{P}\|_{\rho^{-1}}$ is the normal form of $\|\langle\!\langle P \rangle\!\rangle_\rho\|_{\rho^{-1}}$. By Theorem 5.16, $\|\langle\!\langle P \rangle\!\rangle_\rho\|_{\rho^{-1}} = P$, and hence we have our result. ■

Note that Theorems 5.18 and 5.19, and Corollary 5.20 will hold when the program $\mathcal{P}$ is larger, but does contain any interfering clauses, *i.e.*, clauses for any of the normalization or convertibility predicates in `lfconv` or `lfnorm` other than those that are already in those modules. In applying these results in the next section, programs will often contain additional clauses for the `has_type` and `is_type` predicates.

5.3 Translating LF Signatures to Logic Programs

In Section 4.2, after specifying $\beta\eta$-convertibility for untyped λ-terms, we specified a type assignment system for the simply typed λ-calculus. Similarly, for LF, we could specify the inference rules of Figure 5.2 as a set of clauses, and obtain a program for type checking LF assertions. Such an approach is taken in [FM89]. Here, we will instead start with an LF signature, and compile each signature item into a definite clause. Using this approach, no auxiliary clauses specifying the LF inference rules are needed in solving goals obtained by translating LF judgments of the form $K \Rightarrow \mathrm{kind}$, $A \Rightarrow K$, or $M \Rightarrow A$ The only auxiliary clauses required are those for normalization in `lfconv` and `lfnorm`.

We define both a "negative" translation and a "positive" translation. The negative translation maps canonical signature items to definite clauses, while the positive translation maps canonical judgments (*i.e.*, $P \Rightarrow Q$ or $P \Rightarrow \mathrm{kind}$ where P and Q are canonical) to goal formulas. Like term encodings, the translation is with respect to a variable encoding. A variable encoding ρ is well-defined on a context item, or a judgment if it is well-defined on the terms on the left and right of the colon or arrow. We use double brackets subscripted by a variable encoding and superscripted by a sign $[\![]\!]^-_\rho$ and $[\![]\!]^+_\rho$ for the negative and positive translations respectively. We say that a variable encoding is well-defined on a context Γ, if it is well-defined on all of the pairs in Γ. A variable encoding is well-defined on a judgment $P \Rightarrow Q$ if it is well defined on both P and Q. We write $[\![\Gamma]\!]^-_\rho$ to denote the set of clauses containing $[\![x : P]\!]^-_\rho$ for every $x : P \in \Gamma$. The negative translation is defined in Figure 5.6 and the positive translation is defined in Figure 5.7. Note that these definitions are mutually recursive.

$$[\![B : \Pi x : A.K]\!]^-_\rho \quad := \quad \texttt{pi X\(}\Big([\![x \Rightarrow A]\!]^+_{\langle x,\texttt{X}\rangle+\rho} \texttt{ => } [\![Bx : K]\!]^-_{\langle x,\texttt{X}\rangle+\rho}\Big)$$
where X is a variable of type tm such that $\texttt{X} \notin \mathrm{cod}(\rho)$.

$$[\![A : \mathsf{Type}]\!]^-_\rho \quad := \quad \texttt{is\_type } \langle\!\langle A \rangle\!\rangle_\rho \quad \text{where } A \text{ is a base type.}$$

$$[\![M : \Pi x : A.B]\!]^-_\rho \quad := \quad \texttt{pi X\(}\Big([\![x \Rightarrow A]\!]^+_{\langle x,\texttt{X}\rangle+\rho} \texttt{ => } [\![Mx : B]\!]^-_{\langle x,\texttt{X}\rangle+\rho}\Big)$$
where X is a variable of type tm such that $\texttt{X} \notin \mathrm{cod}(\rho)$.

$$[\![M : A]\!]^-_\rho \quad := \quad \texttt{pi B\(norm\_ty } \langle\!\langle A \rangle\!\rangle_\rho \texttt{ B => has\_type } \langle\!\langle M \rangle\!\rangle_\rho \texttt{ B)}$$
where A is a base type and
B is a variable of type ty such that $\texttt{B} \notin \mathrm{cod}(\rho)$.

Figure 5.6: Negative Translation of LF Judgments to Definite Clauses

The negative translation maps a signature item to a formula of the form:

$$\texttt{pi X}_1\texttt{\(G}_1 \texttt{ => ... pi X}_n\texttt{\(G}_n \texttt{ => D)...).}$$

Such formulas can also be written in the following two ways where we assume that for $i = 1, \ldots, n$, $\texttt{X}_{i+1}, \ldots, \texttt{X}_n$ do not appear in $\texttt{G}_i$ or D.

$$\texttt{pi X}_1\texttt{\...pi X}_n\texttt{\((G}_1\texttt{,...,G}_n\texttt{) => D)}$$

$$\texttt{D :- G}_1\texttt{,...,G}_n\texttt{.}$$

The latter form is the way we have been writing definite clauses, where the universal quantification is implicit. The formulas obtained by translating signature items are clauses for the has_type and is_type predicates. Note that these clauses will contain normalization subgoals. These subgoals reflect the normalization that occurs in the inference rules of C-LF after substituting terms for variables inside types.

$$[\![\Pi x : A.K \Rightarrow \text{kind}]\!]^+_\rho := [\![A \Rightarrow \mathsf{Type}]\!]^+_\rho \texttt{, pi X\(normal\_tm1 X =>} [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \texttt{ => } [\![K \Rightarrow \text{kind}]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\texttt{)}$$
where X is a variable of type tm such that $\mathtt{X} \notin \text{cod}(\rho)$.

$$[\![\mathsf{Type} \Rightarrow \text{kind}]\!]^+_\rho := \texttt{true}$$

$$[\![\Pi x : A.B \Rightarrow \mathsf{Type}]\!]^+_\rho := [\![A \Rightarrow \mathsf{Type}]\!]^+_\rho \texttt{, pi X\(normal\_tm1 X =>} [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \texttt{ => } [\![B \Rightarrow \mathsf{Type}]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\texttt{)}$$
where X is a variable of type tm such that $\mathtt{X} \notin \text{cod}(\rho)$.

$$[\![\lambda x : A.B \Rightarrow \Pi x : A.K]\!]^+_\rho := [\![A \Rightarrow \mathsf{Type}]\!]^+_\rho \texttt{, pi X\(normal\_tm1 X =>} [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \texttt{ => } [\![B \Rightarrow K]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\texttt{)}$$
where X is a variable of type tm such that $\mathtt{X} \notin \text{cod}(\rho)$.

$$[\![B \Rightarrow \Pi x : A.K]\!]^+_\rho := [\![A \Rightarrow \mathsf{Type}]\!]^+_\rho \texttt{, pi X\(normal\_tm1 X =>} [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \texttt{ => } [\![Bx \Rightarrow K]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\texttt{)}$$
where B is not an abstraction and
X is a variable of type tm such that $\mathtt{X} \notin \text{cod}(\rho)$.

$$[\![A \Rightarrow \mathsf{Type}]\!]^+_\rho := \texttt{sigma B\(norm\_ty } \langle\!\langle A \rangle\!\rangle_\rho \texttt{ B, is\_type B)}$$
where A is a base type and
B is a variable of type ty such that $\mathtt{B} \notin \text{cod}(\rho)$.

$$[\![\lambda x : A.M \Rightarrow \Pi x : A.B]\!]^+_\rho := [\![A \Rightarrow \mathsf{Type}]\!]^+_\rho \texttt{, pi X\(normal\_tm1 X =>} [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \texttt{ => } [\![M \Rightarrow B]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\texttt{)}$$
where X is a variable of type tm such that $\mathtt{X} \notin \text{cod}(\rho)$.

$$[\![M \Rightarrow \Pi x : A.B]\!]^+_\rho := [\![A \Rightarrow \mathsf{Type}]\!]^+_\rho \texttt{, pi X\(normal\_tm1 X =>} [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \texttt{ => } [\![Mx \Rightarrow B]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\texttt{)}$$
where M is not an abstraction and
X is a variable of type tm such that $\mathtt{X} \notin \text{cod}(\rho)$.

$$[\![M \Rightarrow A]\!]^+_\rho := \texttt{sigma N\(sigma B\(norm\_tm } \langle\!\langle M \rangle\!\rangle_\rho \texttt{ N, norm\_ty } \langle\!\langle A \rangle\!\rangle_\rho \texttt{ B, has\_type N B))}$$
where A is a base type, N is a variable of type tm and
B is variables of type ty such that $\mathtt{N}, \mathtt{B} \notin \text{cod}(\rho)$.

Figure 5.7: Positive Translation of LF Judgments to Goal Formulas

To illustrate this translation, we consider a simple example from an LF signature specifying natural deduction for first-order logic. The following is a declaration which introduces the constant for universal quantification and gives it a type: $\forall : (i \rightarrow form) \rightarrow form$. Let ρ be a variable encoding that contains $\{\langle i, \texttt{i}\rangle, \langle form, \texttt{form}\rangle, \langle \forall, \texttt{forall}\rangle\}$. The translation of this signature item is as follows.

$[\![\forall : (i \rightarrow form) \rightarrow form]\!]^{-}_{\rho} \equiv$

```
  pi A\([[i => Type]]^+_{⟨A,A⟩+ρ},
        pi X\(normal_tm1 X =>[[x : i]]^-_{⟨x,X⟩+⟨A,A⟩+ρ} => [[Ax => form]]^+_{⟨x,X⟩+⟨A,A⟩+ρ}) =>
        [[∀A => form]]^-_{⟨A,A⟩+ρ}) ≡
```

```
has_type (app_tm forall A) B :-
  norm_ty form B,
  sigma B1\ (norm_ty i B1, is_type B1),
  pi X\ (normal_tm1 X => (pi B2\ (norm_ty i B2 => has_type X B2)) =>
         (sigma N\ sigma B3\ (norm_tm (app_tm A X) N, norm_ty form B3,
                              has_type N B3))).
```

Clearly this clause can be simplified. We defer further discussion until Section 5.4 where we present, in full, a program obtained by translating a signature specifying natural deduction for a subset of first-order intutionistic logic.

We will now proceed to prove the correctness of the translations. Before doing so, we prove a lemma demonstrating that in the clauses and goals resulting from the translation, terms and types are normalized (via `norm_ty` and `norm_tm`) appropriately. To see why these normalization subgoals are necessary, consider the (C-APP-OBJ) rule. Each application of this rule involves a signature item of the form $x : \Pi x_1 : A_1 \ldots \Pi x_n : A_n.B$. In this signature item, for $i = 2, \ldots, n$, the type A_i may have occurrences of the bound variables $x_1, \ldots, x_{i-1}$. In an application of the rule, these bound variables $x_1, \ldots, x_n$ get "instantiated" with the terms $N_1, \ldots, N_n$ and the resulting types are then normalized. In the corresponding logic program, the "instantiation" step corresponds to instantiating the universally quantified variables in backchaining on a particular clause. The calls to the `norm_ty` program are then needed to normalize the instantiated types. We will see that the calls to `norm_tm` simply replace bound variables with free variables. While normalization can clearly be used to perform this task (as is done here), a simpler operation could in fact be implemented instead.

It is interesting to note that if the equations for β-reduction for LF terms as specified in the `lfconv` program were built into the unifier of the meta-language, the positive and negative translations could be defined so that they are the same translation. The definition of the translation above must differentiate between normalization subgoals generated by the positive translation that must be solved by the interpreter, and normalization subgoals generated by the negative translation which appear in the bodies of clauses.

Recall that in proving the correctness of the encodings of object level terms as meta-terms, we proved that substitution commutes with the encoding operation (and similarly for decodings). Here we actually prove that both substitution and normalization commute with the translation operation. For example, consider a given LF judgment for which we perform a substitution of terms for variables and normalize at the object level, and then perform the translation on the resulting judgment obtaining a provable goal. We will show that if we first translate the original judgment, encode the terms used in the substitution separately, perform the corresponding instantiation of the encoded terms for the meta-variables at the meta-level, and then do normalization via the normalization programs, the result will also be a provable goal.

We need the following definition for the induction in this lemma. We define the *order* of a type in β-normal form as follows: the order of a base type is 0, and the order of a type of the form $\lambda x_1 : A_1 \ldots \lambda x_n : A_n.\Pi y_1 : B_1 \ldots \Pi y_m : B_m.C$ where C is a base type is one greater than the maximum order of the types $A_1, \ldots, A_n, B_1, \ldots, B_m$. Similarly, we define the order of a kind in β-normal form. The order of Type is 0, and the order of a kind of the form $\Pi x_1 : A_1 \ldots \Pi x_n : A_n.\mathsf{Type}$ is one greater than the maximum order of the types $A_1, \ldots, A_n$. The order of an arbitrary type or kind is the order of its normal form.

As in Section 4.3, we no longer distinguish between two α-convertible terms. Now, when we write $P = Q$, it is understood that P is α-convertible to Q, and similarly for terms at the meta-level.

Lemma 5.21 Let Γ be a context. Let $N_1, \ldots, N_n$ be canonical LF terms with respect to Γ, and let $x, x_1, \ldots, x_n$ be variables. Let N be the variable x or a canonical LF term with respect to Γ. Let σ be the substitution

$$\{\langle x, N\rangle, \langle x_1, N_1\rangle, \ldots, \langle x_n, N_n\rangle\}.$$

Let A be a canonical LF type with respect to Γ. Let ρ_1 be a variable encoding well-defined on A and $\mathrm{dom}(\sigma)$, and ρ_2 a variable encoding well-defined on $\sigma(A)$ and $\mathrm{cod}(\sigma)$, such that ρ_1 and ρ_2 agree on common domain elements that are free in $\sigma(A)$. Let Σ be a signature that includes the declarations of `lfsig`, `lfconv`, `lfnorm`, and the variables in $\mathrm{cod}(\rho_2)$. Let $\mathcal{P}$ be a set of definite clauses that includes `lfconv`, `lfnorm`, $\mathcal{P}_{\mathrm{cod}(\rho_2)}$, and possibly clauses for the `is_type` and `has_type` predicates all of whose constants are in Σ. Then the following hold.

1. $\Sigma; \mathcal{P} \vdash_I [\![N \Rightarrow \sigma(A)^\beta]\!]^+_{\rho_2}$ iff $\Sigma; \mathcal{P} \vdash_I \sigma_{\rho_1,\rho_2}([\![x \Rightarrow A]\!]^+_{\rho_1})$.
2. $\Sigma; \mathcal{P} \vdash_I [\![\sigma(A)^\beta \Rightarrow \mathsf{Type}]\!]^+_{\rho_2}$ iff $\Sigma; \mathcal{P} \vdash_I \sigma_{\rho_1,\rho_2}([\![A \Rightarrow \mathsf{Type}]\!]^+_{\rho_1})$.
3. If y is a variable that is not in $\mathrm{dom}(\sigma)$, but is in $\mathrm{dom}(\rho_1)$ and $\mathrm{dom}(\rho_2)$ and $\rho_1(y) = \rho_2(y)$, and G is any goal formula, then

$$\Sigma; \mathcal{P}, [\![y : \sigma(A)^\beta]\!]^-_{\rho_2} \vdash_I G \quad \text{iff} \quad \Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I G.$$

Proof: We prove the forward and backward direction of (1), (2), and (3) by simultaneous induction on the order of A. We show only the cases for the forward direction. The other direction is similar.

Base: (1) A is a base type. Let `N` be a variable of type `tm` and `B` a variable of type `ty` that do not appear in $\mathrm{cod}(\rho_1)$ or $\mathrm{cod}(\rho_2)$. By the translation:

$$\begin{aligned}[\![N \Rightarrow \sigma(A)^\beta]\!]^+_{\rho_2} \equiv \texttt{sigma N\(sigma B\(norm\_tm } &\langle\!\langle N \rangle\!\rangle_{\rho_2} \texttt{ N,} \\ &\texttt{norm\_ty } \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \texttt{ B,} \\ &\texttt{has\_type N B))}\end{aligned}$$

By Corollary 5.20 we know that the instances of `N` and `B` for which this goal is provable are $\langle\!\langle N \rangle\!\rangle_{\rho_2}$ and $\langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}$, respectively. Thus, by 2 applications of the INSTANCE operation, followed by 2 applications of AND search, the following judgments hold.

$$\begin{aligned}&\Sigma; \mathcal{P} \vdash_I \texttt{norm\_tm } \langle\!\langle N \rangle\!\rangle_{\rho_2} \ \langle\!\langle N \rangle\!\rangle_{\rho_2} \\ &\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ty } \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \\ &\Sigma; \mathcal{P} \vdash_I \texttt{has\_type } \langle\!\langle N \rangle\!\rangle_{\rho_2} \ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \end{aligned} \tag{15}$$

Since $\langle x, N \rangle \in \sigma$, $\langle \langle\!\langle x \rangle\!\rangle_{\rho_1}, \langle\!\langle N \rangle\!\rangle_{\rho_2} \rangle \in \sigma_{\rho_1,\rho_2}$. Thus, $\langle\!\langle N \rangle\!\rangle_{\rho_2} = \sigma_{\rho_1,\rho_2}(\langle\!\langle x \rangle\!\rangle_{\rho_1})$. Since N is canonical, by Theorem 5.18 we can conclude that the following judgment holds.

$$\Sigma; \mathcal{P} \vdash_I \texttt{norm\_tm } \sigma_{\rho_1,\rho_2}(\langle\!\langle x \rangle\!\rangle_{\rho_1}) \ \langle\!\langle N \rangle\!\rangle_{\rho_2} \tag{16}$$

By Lemma 5.9, $\langle\!\langle \sigma(A) \rangle\!\rangle_{\rho_2} = \sigma_{\rho_1,\rho_2}(\langle\!\langle A \rangle\!\rangle_{\rho_1})$. Since $\sigma(A) = \sigma(A)^\beta$, by Theorem 5.18, we can conclude that the following judgment holds.

$$\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ty } \sigma_{\rho_1,\rho_2}(\langle\!\langle A \rangle\!\rangle_{\rho_1}) \ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \tag{17}$$

By the translation (with the substitution moved inward as far as possible):

$$\begin{aligned}\sigma_{\rho_1,\rho_2}([\![x \Rightarrow A]\!]^+_{\rho_1}) \equiv \texttt{sigma N\(sigma B\(norm\_tm } &\sigma_{\rho_1,\rho_2}(\langle\!\langle x \rangle\!\rangle_{\rho_1}) \texttt{ N,} \\ &\texttt{norm\_ty } \sigma_{\rho_1,\rho_2}(\langle\!\langle A \rangle\!\rangle_{\rho_1}) \texttt{ B,} \\ &\texttt{has\_type N B))}\end{aligned}$$

From (16), (17), and (15), respectively, it follows that the three conjuncts of this goal are provable from program $\Sigma; \mathcal{P}$ with instances $\langle\!\langle N \rangle\!\rangle_{\rho_2}$ and $\langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}$ for `N` and `B` respectively.

Base: (2) A is a base type. Let `B` be a variable of type `ty` that does not appear in $\mathrm{cod}(\rho_1)$ or $\mathrm{cod}(\rho_2)$. By the translation:

$$[\![\sigma(A)^\beta \Rightarrow \mathsf{Type}]\!]^+_{\rho_2} \equiv \texttt{sigma B\(norm\_ty } \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \texttt{ B, is\_type B)}$$

By Corollary 5.20 we know that the instance of `B` for which this goal is provable is $\langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}$. Thus, by the INSTANCE operation, followed by AND search, the following

judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ty}\ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}\ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}$$
$$\Sigma; \mathcal{P} \vdash_I \texttt{is\_type}\ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \tag{18}$$

By Lemma 5.9, $\langle\!\langle \sigma(A) \rangle\!\rangle_{\rho_2} = \sigma_{\rho_1,\rho_2}(\langle\!\langle A \rangle\!\rangle_{\rho_1})$. Since $\sigma(A) =_\beta \sigma(A)^\beta$, by Theorem 5.18, we can conclude that the following judgment holds.

$$\Sigma; \mathcal{P} \vdash_I \texttt{norm\_ty}\ \sigma_{\rho_1,\rho_2}(\langle\!\langle A \rangle\!\rangle_{\rho_1})\ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \tag{19}$$

By the translation (with the substitution moved inward as far as possible):

$$\sigma_{\rho_1,\rho_2}([\![A \Rightarrow \mathsf{Type}]\!]^+_{\rho_1}) \equiv \texttt{sigma B\textbackslash(norm\_ty}\ \sigma_{\rho_1,\rho_2}(\langle\!\langle A \rangle\!\rangle_{\rho_1})\ \texttt{B, is\_type B)}$$

From (19) and (18), respectively, it follows that the two conjuncts of this goal are provable from program $\Sigma; \mathcal{P}$ with instance $\langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}$ for `B`.

Base: (3) A is a base type. We prove this case by a second induction on the height of a proof of:

$$\Sigma; \mathcal{P}, [\![y : \sigma(A)^\beta]\!]^-_{\rho_2} \vdash_I G.$$

All cases follow by a simple application of the subinduction hypothesis except for the case when the last step in the proof of the above judgment is a BACKCHAIN on $[\![y : \sigma(A)^\beta]\!]^-_{\rho_2}$. Let `B` be a variable of type `ty` that does not appear in $\mathrm{cod}(\rho_1)$ or $\mathrm{cod}(\rho_2)$. By the translation:

$$[\![y : \sigma(A)^\beta]\!]^-_{\rho_2} \equiv \texttt{pi B\textbackslash(norm\_ty}\ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}\ \texttt{B => has\_type}\ \langle\!\langle y \rangle\!\rangle_{\rho_2}\ \texttt{B)}$$

Thus G has the form `(has_type` $\langle\!\langle y \rangle\!\rangle_{\rho_2}$ `C)`, where `C` is an instance of `B`. By BACKCHAIN, the following judgment holds.

$$\Sigma; \mathcal{P}, [\![y : \sigma(A)^\beta]\!]^-_{\rho_2} \vdash_I \texttt{norm\_ty}\ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}\ \texttt{C}$$

By Corollary 5.20, `C` must be $\langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}$. By Lemma 5.9, $\langle\!\langle \sigma(A) \rangle\!\rangle_{\rho_2} = \sigma_{\rho_1,\rho_2}(\langle\!\langle A \rangle\!\rangle_{\rho_1})$. Since $\sigma(A) =_\beta \sigma(A)^\beta$, by Theorem 5.18, we can conclude that the following judgment holds.

$$\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I \texttt{norm\_ty}\ \sigma_{\rho_1,\rho_2}(\langle\!\langle A \rangle\!\rangle_{\rho_1})\ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \tag{20}$$

By the translation (with the substitution moved inward as far as possible):

$$\sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \equiv \texttt{pi B\textbackslash(norm\_ty}\ \sigma_{\rho_1,\rho_2}(\langle\!\langle A \rangle\!\rangle_{\rho_1})\ \texttt{B => has\_type}\ \sigma_{\rho_1,\rho_2}(\langle\!\langle y \rangle\!\rangle_{\rho_1})\ \texttt{B)}$$

From (20), by BACKCHAIN on this clause (in the forward direction) with $\langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}$ as the instance of `B` we can conclude that the following judgment holds.

$$\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I \texttt{has\_type}\ \sigma_{\rho_1,\rho_2}(\langle\!\langle y \rangle\!\rangle_{\rho_1})\ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2} \tag{21}$$

Since $y \notin \mathrm{dom}(\sigma)$, $\rho_1(y) \notin \mathrm{dom}(\sigma_{\rho_1,\rho_2})$. We also know that $\rho_1(y) = \rho_2(y)$. Thus $\sigma_{\rho_1,\rho_2}(\langle\!\langle y \rangle\!\rangle_{\rho_1}) = \langle\!\langle y \rangle\!\rangle_{\rho_1} = \langle\!\langle y \rangle\!\rangle_{\rho_2}$. Hence the goal formula $\texttt{(has\_type } \sigma_{\rho_1,\rho_2}(\langle\!\langle y \rangle\!\rangle_{\rho_1}) \ \langle\!\langle \sigma(A)^\beta \rangle\!\rangle_{\rho_2}\texttt{)}$ is the same as the goal formula $\texttt{(has\_type } \langle\!\langle y \rangle\!\rangle_{\rho_2} \ \texttt{C)}$, which is the goal formula G that we set out to prove.

This completes the three base cases. We now show that (1), (2), and (3) hold when the type A has order n, assuming that they hold for types of order less than n. We will refer to these assumptions as induction hypotheses (1), (2), and (3).

Case: (1) A is $\Pi z_1 : A_1 \ldots \Pi z_m : A_m.B$ where B is a base type. We assume that $z_1, \ldots, z_m$ do not appear in $\mathrm{dom}(\sigma)$ or $\mathrm{cod}(\sigma)$, otherwise we rename them. Then

$$\sigma(A)^\beta = \Pi z_1 : \sigma(A_1)^\beta \ldots \Pi z_m : \sigma(A_m)^\beta.\sigma(B)^\beta.$$

We show the case when N is a canonical term. The case when N is x is simpler. In order to apply the translation N must have the form

$$\lambda z_1 : \sigma(A_1)^\beta \ldots \lambda z_m : \sigma(A_m)^\beta.P.$$

Let $\mathtt{Z}_1, \ldots, \mathtt{Z}_m, \mathtt{N}$ be distinct variables of type `tm` and `B` a variable of type `ty` that do not occur in $\mathrm{cod}(\rho_1)$ or $\mathrm{cod}(\rho_2)$. Note that these variables also do not appear in Σ. For $i = 1, \ldots, m$, let

$$\begin{aligned}
\rho_1^i &:= \langle z_1, \mathtt{Z}_1 \rangle + \cdots + \langle z_i, \mathtt{Z}_i \rangle + \rho_1 \\
\rho_2^i &:= \langle z_1, \mathtt{Z}_1 \rangle + \cdots + \langle z_i, \mathtt{Z}_i \rangle + \rho_2 \\
\Sigma_i &:= \Sigma \cup \{\mathtt{Z}_1\texttt{:tm}, \ldots, \mathtt{Z}_i\texttt{:tm}\} \\
\mathcal{P}_i &:= \mathcal{P} \cup \{\texttt{normal_tm1 } \mathtt{Z}_1, \ldots, \texttt{ normal_tm1 } \mathtt{Z}_i\}
\end{aligned}$$

For $i = 1, \ldots, m$, note that $\rho_1^i \subseteq \rho_1$, $\rho_2^i \subseteq \rho_2$, and $z_i \notin \mathrm{dom}(\sigma)$. Thus the subsitution $\sigma_{\rho_1^i,\rho_2^i}$ is the same substitution as σ_{ρ_1,ρ_2}. By the translation:

$$\begin{aligned}
&[\![\lambda z_1 : \sigma(A_1)^\beta \ldots \lambda z_m : \sigma(A_m)^\beta.P \Rightarrow \Pi z_1 : \sigma(A_1)^\beta \ldots \Pi z_m : \sigma(A_m)^\beta.\sigma(B)^\beta]\!]^+_{\rho_2} \equiv \\
&\quad [\![\sigma(A_1)^\beta \Rightarrow \mathsf{Type}]\!]^+_{\rho_2}\texttt{, pi Z}_1\texttt{\textbackslash(normal_tm1 Z}_1\texttt{ => } [\![z_1 : \sigma(A_1)^\beta]\!]^-_{\rho_2^1}\texttt{ =>} \\
&\qquad\qquad\qquad\qquad \vdots \\
&\quad [\![\sigma(A_m)^\beta \Rightarrow \mathsf{Type}]\!]^+_{\rho_2^{m-1}}\texttt{, pi Z}_m\texttt{\textbackslash(normal_tm1 Z}_m\texttt{ => } [\![z_m : \sigma(A_m)^\beta]\!]^-_{\rho_2^m}\texttt{ =>} \\
&\quad \texttt{sigma N\textbackslash(sigma B\textbackslash(norm_tm } \langle\!\langle P \rangle\!\rangle_{\rho_2^m}\texttt{ N,} \\
&\qquad\qquad\qquad \texttt{norm_ty } \langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m}\texttt{ B,} \\
&\qquad\qquad\qquad \texttt{has_type N B)...)}
\end{aligned}$$

By m alternate applications of AND, GENERIC, and AUGMENT twice each time, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I [\![\sigma(A_1)^\beta \Rightarrow \mathsf{Type}]\!]^+_{\rho_2}$$

$$\Sigma_1; \mathcal{P}_1, [\![z_1 : \sigma(A_1)^\beta]\!]^-_{\rho_2^1} \vdash_I [\![\sigma(A_2)^\beta \Rightarrow \mathsf{Type}]\!]^+_{\rho_2^1}$$

$$\vdots$$

$$\Sigma_{m-1}; \mathcal{P}_{m-1}, [\![z_1 : \sigma(A_1)^\beta]\!]^-_{\rho_2^1}, \ldots, [\![z_{m-1} : \sigma(A_{m-1})^\beta]\!]^-_{\rho_2^{m-1}} \vdash_I [\![\sigma(A_m)^\beta \Rightarrow \mathsf{Type}]\!]^+_{\rho_2^{m-1}}$$

$$\Sigma_m; \mathcal{P}_m, [\![z_1 : \sigma(A_1)^\beta]\!]^-_{\rho_2^1}, \ldots, [\![z_m : \sigma(A_m)^\beta]\!]^-_{\rho_2^m} \vdash_I \texttt{sigma N\(sigma B\}$$
$$\texttt{(norm\_tm } \langle\!\langle P \rangle\!\rangle_{\rho_2^m} \texttt{ N, norm\_ty } \langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m} \texttt{ B, has\_type N B))}$$

By induction hypothesis (2) in the forward direction, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \sigma_{\rho_1,\rho_2}([\![A_1 \Rightarrow \mathsf{Type}]\!]^+_{\rho_1})$$
$$\Sigma_1; \mathcal{P}_1, [\![z_1 : \sigma(A_1)^\beta]\!]^-_{\rho_2^1} \vdash_I \sigma_{\rho_1^1,\rho_2^1}([\![A_2 \Rightarrow \mathsf{Type}]\!]^+_{\rho_1^1})$$

$$\vdots$$

$$\Sigma_{m-1}; \mathcal{P}_{m-1}, [\![z_1 : \sigma(A_1)^\beta]\!]^-_{\rho_2^1}, \ldots, [\![z_{m-1} : \sigma(A_{m-1})^\beta]\!]^-_{\rho_1^{m-1}} \vdash_I \sigma_{\rho_1^{m-1},\rho_2^{m-1}}([\![A_m \Rightarrow \mathsf{Type}]\!]^+_{\rho_2^{m-1}})$$

By repeated applications of induction hypothesis (3) in the forward direction, the following judgments hold.

$$\Sigma; \mathcal{P} \vdash_I \sigma_{\rho_1,\rho_2}([\![A_1 \Rightarrow \mathsf{Type}]\!]^+_{\rho_1}) \tag{22}$$
$$\Sigma_1; \mathcal{P}_1, \sigma_{\rho_1^1,\rho_2^1}([\![z_1 : A_1]\!]^-_{\rho_1^1}) \vdash_I \sigma_{\rho_1^1,\rho_2^1}([\![A_2 \Rightarrow \mathsf{Type}]\!]^+_{\rho_1^1})$$

$$\vdots$$

$$\Sigma_{m-1}; \mathcal{P}_{m-1}, \sigma_{\rho_1^1,\rho_2^1}([\![z_1 : A_1]\!]^-_{\rho_1^1}), \ldots, \sigma_{\rho_1^{m-1},\rho_2^{m-1}}([\![z_{m-1} : A_{m-1}]\!]^-_{\rho_1^{m-1}})$$
$$\vdash_I \sigma_{\rho_1^{m-1},\rho_2^{m-1}}([\![A_m \Rightarrow \mathsf{Type}]\!]^+_{\rho_1^{m-1}})$$
$$\Sigma_m; \mathcal{P}_m, \sigma_{\rho_1^1,\rho_2^1}([\![z_1 : A_1]\!]^-_{\rho_1^1}), \ldots, \sigma_{\rho_1^m,\rho_2^m}([\![z_m : A_m]\!]^-_{\rho_1^m}) \vdash_I \texttt{sigma N\(sigma B\}$$
$$\texttt{(norm\_tm } \langle\!\langle P \rangle\!\rangle_{\rho_2^m} \texttt{ N, norm\_ty } \langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m} \texttt{ B, has\_type N B))} \tag{23}$$

By Corollary 5.20 we know that the instances of `N` and `B` for which (23) hold are $\langle\!\langle P \rangle\!\rangle_{\rho_2^m}$ and $\langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m}$, respectively. Let $\mathcal{P}'$ be the following set of clauses.

$$\mathcal{P}' := \{\sigma_{\rho_1,\rho_2}([\![z_1 : A_1]\!]^-_{\rho_1^1}), \ldots, \sigma_{\rho_1,\rho_2}([\![z_m : A_m]\!]^-_{\rho_1^m})\}.$$

Thus, by 2 applications of the INSTANCE operation, followed by 2 applications of AND search, the following judgments hold.

$$\Sigma_m; \mathcal{P}_m, \mathcal{P}' \vdash_I \texttt{norm\_tm } \langle\!\langle P \rangle\!\rangle_{\rho_2^m} \ \langle\!\langle P \rangle\!\rangle_{\rho_2^m}$$
$$\Sigma_m; \mathcal{P}_m, \mathcal{P}' \vdash_I \texttt{norm\_ty } \langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m} \ \langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m}$$
$$\Sigma_m; \mathcal{P}_m, \mathcal{P}' \vdash_I \texttt{has\_type } \langle\!\langle P \rangle\!\rangle_{\rho_2^m} \ \langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m} \tag{24}$$

By assumption, $\langle x, \lambda z_1 : \sigma(A_1)^\beta \ldots \lambda z_m : \sigma(A_m)^\beta.P\rangle \in \sigma$, so by β-conversion and substitution:

$$P =_\beta (\lambda z_1 : \sigma(A_1)^\beta \ldots \lambda z_m : \sigma(A_m)^\beta.P)z_1 \ldots z_m = \sigma(xz_1 \ldots z_m).$$

Since P is canonical, by Theorem 5.18, the following holds.

$$\Sigma_m; \mathcal{P}_m, \mathcal{P}' \vdash_I \texttt{norm\_tm}\ \langle\!\langle \sigma(xz_1 \ldots z_m) \rangle\!\rangle_{\rho_2^m}\ \langle\!\langle P \rangle\!\rangle_{\rho_2^m}$$

By Lemma 5.9, $\langle\!\langle \sigma(xz_1 \ldots z_m) \rangle\!\rangle_{\rho_2^m} = \sigma_{\rho_1^m,\rho_2^m}(\langle\!\langle xz_1 \ldots z_m \rangle\!\rangle_{\rho_1^m})$. Also $\sigma_{\rho_1^m,\rho_2^m} = \sigma_{\rho_1,\rho_2}$, so the above judgment is equivalent to:

$$\Sigma_m; \mathcal{P}_m, \mathcal{P}' \vdash_I \texttt{norm\_tm}\ \sigma_{\rho_1,\rho_2}(\langle\!\langle xz_1 \ldots z_m \rangle\!\rangle_{\rho_1^m})\ \langle\!\langle P \rangle\!\rangle_{\rho_2^m} \tag{25}$$

By Lemma 5.9, $\langle\!\langle \sigma(B) \rangle\!\rangle_{\rho_2^m} = \sigma_{\rho_1^m,\rho_2^m}(\langle\!\langle B \rangle\!\rangle_{\rho_1^m})$. Since $\sigma(B) =_\beta \sigma(B)^\beta$, and $\sigma_{\rho_1^m,\rho_2^m} = \sigma_{\rho_1,\rho_2}$, it follows by Corollary 5.20 that the following judgment holds.

$$\Sigma_m; \mathcal{P}_m, \mathcal{P}' \vdash_I \texttt{norm\_ty}\ \sigma_{\rho_1,\rho_2}(\langle\!\langle B \rangle\!\rangle_{\rho_1^m})\ \langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m} \tag{26}$$

We must show that the following judgment holds.

$$\Sigma; \mathcal{P} \vdash_I \sigma_{\rho_1,\rho_2}([\![x \Rightarrow \Pi z_1 : A_1 \ldots \Pi z_m : A_m.B]\!]^+_{\rho_1})$$

By the translation $\sigma_{\rho_1,\rho_2}([\![x \Rightarrow A]\!]^+_{\rho_1}) \equiv$

$$\begin{aligned}
\sigma_{\rho_1,\rho_2}(\ & [\![A_1 \Rightarrow \mathsf{Type}]\!]^+_{\rho_1}\texttt{, pi Z}_1\texttt{\textbackslash(normal_tm1 Z}_1\texttt{ => }[\![z_1 : A_1]\!]^-_{\rho_1^1}\texttt{ =>} \\
& \qquad\qquad \vdots \\
& [\![A_n \Rightarrow \mathsf{Type}]\!]^+_{\rho_1^{m-1}}\texttt{, pi Z}_n\texttt{\textbackslash(normal_tm1 Z}_1\texttt{ => }[\![z_m : A_m]\!]^-_{\rho_1^m}\texttt{ =>} \\
& \texttt{sigma N\textbackslash(sigma B\textbackslash(norm_tm }\langle\!\langle xz_1 \ldots z_m \rangle\!\rangle_{\rho_1^m}\texttt{ N,} \\
& \qquad\qquad \texttt{norm_ty }\langle\!\langle B \rangle\!\rangle_{\rho_1^m}\texttt{ B,} \\
& \qquad\qquad \texttt{has_type N B)...))}
\end{aligned}$$

Since $\texttt{Z}_1, \ldots, \texttt{Z}_m, \texttt{N}, \texttt{B}$ do not appear in $\mathrm{cod}(\rho_1)$ or $\mathrm{cod}(\rho_2)$, they do not appear in σ_{ρ_1,ρ_2}, so we can move the substitution as far inward as possible. Thus the above judgment holds if by m alternate applications of AND, GENERIC, and AUGMENT twice each time, the following judgments hold.

$\Sigma; \mathcal{P} \vdash_I \sigma_{\rho_1,\rho_2}([\![A_1 \Rightarrow \mathsf{Type}]\!]^+_{\rho_1})$

$\Sigma_1; \mathcal{P}_1, \sigma_{\rho_1,\rho_2}([\![z_1 : A_1]\!]^-_{\rho_1^1}) \vdash_I \sigma_{\rho_1,\rho_2}([\![A_2 \Rightarrow \mathsf{Type}]\!]^+_{\rho_1^1})$

$\vdots$

$\Sigma_{m-1}; \mathcal{P}_{m-1}, \sigma_{\rho_1,\rho_2}([\![z_1 : A_1]\!]^-_{\rho_1^1}), \ldots, \sigma_{\rho_1,\rho_2}([\![z_{m-1} : A_{m-1}]\!]^-_{\rho_1^{m-1}})$
$\quad \vdash_I \sigma_{\rho_1,\rho_2}([\![A_m \Rightarrow \mathsf{Type}]\!]^+_{\rho_1^{m-1}})$

$\Sigma_m; \mathcal{P}_m, \sigma_{\rho_1,\rho_2}([\![z_1 : A_1]\!]^-_{\rho_1^1}), \ldots, \sigma_{\rho_1,\rho_2}([\![z_m : A_m]\!]^-_{\rho_1^m}) \vdash_I \texttt{sigma N\textbackslash(sigma B\textbackslash}$
$\quad \texttt{(norm\_tm }\sigma_{\rho_1,\rho_2}(\langle\!\langle xz_1 \ldots z_m \rangle\!\rangle_{\rho_1^m})\texttt{ N, norm\_ty }\sigma_{\rho_1,\rho_2}(\langle\!\langle B \rangle\!\rangle_{\rho_1^m})\texttt{ B, has\_type N B))}$

We know that for $i = 1, \ldots, m$, it is the case that $\sigma_{\rho_1,\rho_2} = \sigma_{\rho_1^i,\rho_2^i}$. Thus, the first m judgments are the same as the m judgments starting with (22) that were shown to hold above. It follows from (25), (26), and (24), respectively, with instances $\langle\!\langle P \rangle\!\rangle_{\rho_2^m}$ and $\langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m}$ for `N` and `B` respectively, that the three conjuncts of the latter judgment hold.

Case: (2) The inductive case for (2) is similar to the inductive case for (1).

Case: (3) A is $\Pi z_1 : A_1 \ldots \Pi z_m : A_m.B$ where B is a base type. Again, we prove this case by a second induction on the height of a proof of:

$$\Sigma; \mathcal{P}, [\![y : \sigma(A)^\beta]\!]^-_{\rho_2} \vdash_I G.$$

The induction hypothesis of this subinduction will be called (3a). All cases follow by a simple application of induction hypothesis (3a) except for the case when the last step in the proof of the above judgment is a BACKCHAIN on $[\![y : \sigma(A)^\beta]\!]^-_{\rho_2}$. We assume that $z_1, \ldots, z_m$ do not appear free in $\mathrm{dom}(\sigma)$ or $\mathrm{cod}(\sigma)$, and for $i = 1, \ldots, m$, $z_i, \ldots, z_m$ do not appear free in A_i, otherwise we rename them. Then

$$\sigma(A)^\beta = \Pi z_1 : \sigma(A_1)^\beta \ldots \Pi z_m : \sigma(A_m)^\beta.\sigma(B)^\beta.$$

Let $\mathtt{Z}_1, \ldots, \mathtt{Z}_m$ be distinct variables of type `tm` and `B` a variable of type `ty` that do not occur in $\mathrm{cod}(\rho_1)$, $\mathrm{cod}(\rho_2)$. For $i = 1, \ldots, m$, let

$$\rho_1^i = \langle z_1, \mathtt{Z}_1 \rangle + \cdots + \langle z_i, \mathtt{Z}_i \rangle + \rho_1 \qquad \text{and} \qquad \rho_2^i = \langle z_1, \mathtt{Z}_1 \rangle + \cdots + \langle z_i, \mathtt{Z}_i \rangle + \rho_2.$$

By the translation:

$[\![y : \Pi z_1 : \sigma(A_1)^\beta \ldots \Pi z_m : \sigma(A_m)^\beta.\sigma(B)^\beta]\!]^-_{\rho_2} \equiv$

```
pi Z_1\([[z_1 ⇒ σ(A_1)^β]]^+_{ρ_2^1} =>
                ⋮
    pi Z_m\([[z_m ⇒ σ(A_m)^β]]^+_{ρ_2^m}
        pi B\(norm_ty ⟪σ(B)^β⟫_{ρ_2^m} B =>
            has_type (app_tm ... (app_tm ρ_2^m(y) Z_1) ... Z_m) B)...)
```

Thus for some instances $\mathtt{M}_1, \ldots, \mathtt{M}_m, \mathtt{C}$ of $\mathtt{Z}_1, \ldots, \mathtt{Z}_m, \mathtt{B}$, G is

```
(has_type (app_tm ... (app_tm ρ_2^m(y) M_1) ... M_m) C)
```

The terms $\mathtt{M}_1, \ldots, \mathtt{M}_m$ are in $H(\Sigma)$ and thus their $\beta\eta$-long forms are in $T(\mathrm{cod}(\rho_2))$. Hence, there are terms $M_1, \ldots, M_m$ such that for $i = 1, \ldots, m$ $\langle\!\langle M_i \rangle\!\rangle_{\rho_2} = \mathtt{M}_i$. Let σ' be the substitution

$$\{\langle z_1, M_1 \rangle, \ldots, \langle z_m, M_m \rangle\}.$$

Then

$$\sigma'_{\rho_2^m,\rho_2} = \{\langle \mathtt{Z}_1, \mathtt{M}_1 \rangle, \ldots, \langle \mathtt{Z}_m, \mathtt{M}_m \rangle\} \quad \text{and} \quad \sigma'_{\rho_1^m,\rho_2} = \{\langle \mathtt{Z}_1, \mathtt{M}_1 \rangle, \ldots, \langle \mathtt{Z}_m, \mathtt{M}_m \rangle\}.$$

Then for $j = 1$ or 2, G can be written:

$$\sigma'_{\rho_j^m,\rho_2}(\texttt{has\_type (app\_tm ... (app\_tm } \rho_2^m(y)\ \texttt{Z}_1\texttt{) ... Z}_m\texttt{) C}).$$

By BACKCHAIN on the above clause, followed by m applications of AND search, the following judgments hold.

$$\begin{array}{l}
\Sigma; \mathcal{P}, [\![y : \sigma(A)^\beta]\!]^-_{\rho_2} \vdash_I \sigma'_{\rho_2^m,\rho_2}([\![z_1 \Rightarrow \sigma(A_1)^\beta]\!]^+_{\rho_2^1}) \\
\vdots \\
\Sigma; \mathcal{P}, [\![y : \sigma(A)^\beta]\!]^-_{\rho_2} \vdash_I \sigma'_{\rho_2^m,\rho_2}([\![z_m \Rightarrow \sigma(A_m)^\beta]\!]^+_{\rho_2^m}) \\
\Sigma; \mathcal{P}, [\![y : \sigma(A)^\beta]\!]^-_{\rho_2} \vdash_I \texttt{norm_ty}\ \sigma'_{\rho_2^m,\rho_2}(\langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m})\ \texttt{C}
\end{array}$$

By Lemma 5.9,

$$\sigma'_{\rho_2^m,\rho_2}(\langle\!\langle \sigma(B)^\beta \rangle\!\rangle_{\rho_2^m}) = \langle\!\langle \sigma'(\sigma(B)^\beta) \rangle\!\rangle_{\rho_2}.$$

By Corollary 5.20, C must be $\langle\!\langle \sigma'(\sigma(B))^\beta \rangle\!\rangle_{\rho_2}$. Since $z_1, \ldots, z_m$ do not appear free in $\mathrm{dom}(\sigma)$ or $\mathrm{cod}(\sigma)$, $\sigma'(\sigma(B))^\beta = ((\sigma' \cup \sigma)(B))^\beta$. By induction hypothesis (3a) the following hold.

$$\begin{array}{l}
\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I \sigma'_{\rho_2^m,\rho_2}([\![z_1 \Rightarrow \sigma(A_1)^\beta]\!]^+_{\rho_2^1}) \\
\vdots \\
\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I \sigma'_{\rho_2^m,\rho_2}([\![z_m \Rightarrow \sigma(A_m)^\beta]\!]^+_{\rho_2^m})
\end{array}$$

By induction hypothesis (1) in the backward direction, the following hold.

$$\begin{array}{l}
\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I [\![M_1 \Rightarrow \sigma'(\sigma(A_1))^\beta]\!]^+_{\rho_2} \\
\vdots \\
\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I [\![M_m \Rightarrow \sigma'(\sigma(A_m))^\beta]\!]^+_{\rho_2})
\end{array}$$

Since $z_1, \ldots, z_m$ do not appear free in $\mathrm{dom}(\sigma)$ or $\mathrm{cod}(\sigma)$, for $i = 1, \ldots, m$,

$$\sigma'(\sigma(A_i))^\beta = ((\sigma' \cup \sigma)(A_i))^\beta.$$

So, by induction hypothesis (1) in the forward direction, the following hold.

$$\begin{array}{l}
\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I (\sigma' \cup \sigma)_{\rho_1^m,\rho_2}([\![z_1 \Rightarrow A_1]\!]^+_{\rho_1^m}) \\
\vdots \\
\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I (\sigma' \cup \sigma)_{\rho_1^m,\rho_2}([\![z_m \Rightarrow A_m]\!]^+_{\rho_1^m})
\end{array} \tag{27}$$

We must show that the following judgment holds.

$$\begin{array}{l}
\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \\
\quad \vdash_I \texttt{(has_type (app_tm ... (app_tm } \rho_2^m(y)\ \texttt{M}_1\texttt{) ... M}_m\texttt{)}\ \langle\!\langle ((\sigma' \cup \sigma)(B))^\beta \rangle\!\rangle_{\rho_2}\texttt{)}
\end{array} \tag{28}$$

We can do so by backchaining on $\sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1})$. By the translation (with the substitution moved inward as far as possible):

$$\sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) = \sigma_{\rho_1,\rho_2}([\![y : \Pi z_1 : A_1 \ldots \Pi z_m : A_m.B]\!]^-_{\rho_1}) \equiv$$

```
pi Z_1\(σ_{ρ1,ρ2}([[z_1 ⇒ A_1]]^+_{ρ_1^1}) =>
                    ⋮
    pi Z_m\(σ_{ρ1,ρ2}([[z_m ⇒ A_m]]^+_{ρ_1^m})
        pi B\(norm_ty σ_{ρ1,ρ2}(⟪B⟫_{ρ_1^m}) B =>
            has_type (app_tm ... (app_tm ρ_1^m(y) Z_1) ... Z_m) B)...)
```

Since $\rho_1(y) = \rho_2(y)$, it follows that $\rho_1^m(y) = \rho_2^m(y)$. Thus, (28) holds if, by BACKCHAIN on the above clause with instances $\mathtt{M}_1, \ldots, \mathtt{M}_m, \langle\!\langle((\sigma' \cup \sigma)(B))^\beta\rangle\!\rangle_{\rho_2}$ for $\mathtt{Z}_1, \ldots, \mathtt{Z}_m, \mathtt{B}$, followed by m applications of AND search, the following judgments hold.

$$\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I \sigma'_{\rho_1^m,\rho_2}(\sigma_{\rho_1,\rho_2}([\![z_1 \Rightarrow A_1]\!]^+_{\rho_1^1}))$$

$$\vdots$$

$$\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I \sigma'_{\rho_1^m,\rho_2}(\sigma_{\rho_1,\rho_2}([\![z_m \Rightarrow A_m]\!]^+_{\rho_1^m}))$$

$$\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I \mathtt{norm\_ty}\ \sigma'_{\rho_1^m,\rho_2}(\sigma_{\rho_1,\rho_2}(\langle\!\langle B\rangle\!\rangle_{\rho_1^m}))\ \langle\!\langle((\sigma' \cup \sigma)(B))^\beta\rangle\!\rangle_{\rho_2}$$

It is an easy consequence of Lemma 5.9 that since for $i = 1, \ldots, m$, $z_i, \ldots, z_m$ do not appear free in A_i, $[\![z_i \Rightarrow A_i]\!]^+_{\rho^i} = [\![z_i \Rightarrow A_i]\!]^+_{\rho_1^m}$. Also, $\sigma_{\rho_1,\rho_2} = \sigma_{\rho_1^m,\rho_2}$, and the variables $\mathtt{Z}_1, \ldots, \mathtt{Z}_m$ do not appear free in $\mathrm{dom}(\sigma_{\rho_1^m,\rho_2})$ or $\mathrm{cod}(\sigma_{\rho_1^m,\rho_2})$. Thus for $i = 1, \ldots, m$,

$$\sigma'_{\rho_1^m,\rho_2}(\sigma_{\rho_1,\rho_2}([\![z_i \Rightarrow A_i]\!]^+_{\rho_1^m})) = ((\sigma'_{\rho_1^m,\rho_2} \cup \sigma_{\rho_1^m,\rho_2})([\![z_i \Rightarrow A_i]\!]^+_{\rho_1^m}))$$
$$= ((\sigma' \cup \sigma)_{\rho_1^m,\rho_2}([\![z_i \Rightarrow A_i]\!]^+_{\rho_1^m})).$$

By the same argument, $\sigma'_{\rho_1^m,\rho_2}(\sigma_{\rho_1,\rho_2}(\langle\!\langle B\rangle\!\rangle_{\rho_1^m})) = ((\sigma' \cup \sigma)_{\rho_1^m,\rho_2}(\langle\!\langle B\rangle\!\rangle_{\rho_1^m}))$. Thus the above judments can be rewritten as:

$$\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I (\sigma' \cup \sigma)_{\rho_1^m,\rho_2}([\![z_1 \Rightarrow A_1]\!]^+_{\rho_1^m})$$

$$\vdots$$

$$\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I (\sigma' \cup \sigma)_{\rho_1^m,\rho_2}([\![z_m \Rightarrow A_m]\!]^+_{\rho_1^m})$$

$$\Sigma; \mathcal{P}, \sigma_{\rho_1,\rho_2}([\![y : A]\!]^-_{\rho_1}) \vdash_I \mathtt{norm\_ty}\ (\sigma' \cup \sigma)_{\rho_1^m,\rho_2}(\langle\!\langle B\rangle\!\rangle_{\rho_1^m})\ \langle\!\langle((\sigma' \cup \sigma)(B))^\beta\rangle\!\rangle_{\rho_2}$$

From the m judgments starting with (27), it follows that the first m judgments above hold. By Lemma 5.9,

$$(\sigma' \cup \sigma)_{\rho_1^m,\rho_2}(\langle\!\langle B\rangle\!\rangle_{\rho_1^m}) = \langle\!\langle(\sigma' \cup \sigma)(B)\rangle\!\rangle_{\rho_2}.$$

Since $(\sigma' \cup \sigma)(B) =_\beta ((\sigma' \cup \sigma)(B))^\beta$, by Theorem 5.18, we can conclude that the latter judgment holds. ■

Theorem 5.22 (Correctness of Translation I)

Let $\Gamma \vdash \alpha$ be an assertion provable in C-LF. Let ρ be a variable encoding well-defined on Γ and α. Let Σ be a signature that includes the declarations of `lfsig`, `lfconv`, `lfnorm`, and the variables in $\mathrm{cod}(\rho)$. Let $\mathcal{P}$ be a set of definite clauses that includes `lfconv`, `lfnorm`, and $\mathcal{P}_{\mathrm{cod}(\rho)}$. Then

$$\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I [\![\alpha]\!]^+_\rho.$$

Proof: The proof is by induction on the height of a C-LF proof of the assertion.

Base: (A-TYPE-KIND) $[\![\mathsf{Type} \Rightarrow \mathrm{kind}]\!]^+_\rho \equiv \mathtt{true}$. Clearly the judgment $\Sigma; \mathcal{P} \vdash_I \mathtt{true}$ holds.

Case: (A-K-VAR) Clearly, the judgment $\Sigma; \mathcal{P}, [\![\Gamma, x : K]\!]^-_\rho \vdash_I \mathtt{true}$ holds.

Case: (A-T-VAR). Clearly, the judgment $\Sigma; \mathcal{P}, [\![\Gamma, x : A]\!]^-_\rho \vdash_I \mathtt{true}$ holds.

Case: (A-ABS-OBJ) We assume that the variable x does not appear free in Γ, otherwise we rename it. Let `X` be a variable of type `tm` such that $\mathtt{X} \notin \mathrm{cod}(\rho)$. By the translation:

$[\![\lambda x : A.M \Rightarrow \Pi x : A.B]\!]^+_\rho \equiv$

$$[\![A \Rightarrow \mathsf{Type}]\!]^+_\rho \texttt{, pi X\(normal\_tm1 X => } [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \texttt{ => } [\![M \Rightarrow B]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\texttt{)}$$

We must show that the following judgments hold.

$$\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I [\![A \Rightarrow \mathsf{Type}]\!]^+_\rho$$
$$\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \texttt{pi X\(normal\_tm1 X => } [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \texttt{ => } [\![M \Rightarrow B]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\texttt{)}$$

The first one holds by the induction hypothesis for the left premise. The second one holds, if by GENERIC, followed by AUGMENT twice, the following judgment holds.

$$\Sigma \cup \{\mathtt{X} : \mathtt{tm}\}; \mathcal{P}, [\![\Gamma]\!]^-_\rho, \mathtt{normal\_tm1\ X}, [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \vdash_I [\![M \Rightarrow B]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}$$

It is an easy consequence of Corollary 5.10 that since x is not free in Γ, $[\![\Gamma]\!]^-_\rho = [\![\Gamma]\!]^-_{\langle x,\mathtt{X}\rangle+\rho}$. Thus

$$[\![\Gamma]\!]^-_{\langle x,\mathtt{X}\rangle+\rho}, [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} = [\![\Gamma, x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho},$$

so the above judgment holds by the induction hypothesis for the right premise.

Cases: (A-PI-KIND), (A-PI-FAM), and (A-ABS-FAM). Similar to (A-ABS-OBJ).

Case: (C-APP-OBJ) We assume that $x_1, \ldots, x_n$ do not appear free in $N_1, \ldots, N_n$, and that for $i = 1, \ldots, n$, $x_i, \ldots, x_n$ do not appear free in A_i, otherwise we rename them in $x : \Pi x_1 : A_1 \ldots \Pi x_n : A_n.B$. We must show that

$$\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I [\![x N_1 \ldots N_n \Rightarrow ([N_1/x_1, \ldots, N_n/x_n]B)^\beta]\!]^+_\rho \tag{29}$$

Let `N` be a variable of type `tm` and `B` a variable of type `ty` that do not appear in $\mathrm{cod}(\rho)$.

By the translation:

$$\begin{aligned}&[\![xN_1 \ldots N_n \Rightarrow ([N_1/x_1, \ldots, N_n/x_n]B)^\beta]\!]^+_\rho \equiv \\ &\quad \texttt{sigma N\(sigma B\(norm\_tm } \langle\!\langle xN_1 \ldots N_n \rangle\!\rangle_\rho \texttt{ N,} \\ &\qquad\qquad\qquad \texttt{norm\_ty } \langle\!\langle ([N_1/x_1, \ldots, N_n/x_n]B)^\beta \rangle\!\rangle_\rho \texttt{ B,} \\ &\qquad\qquad\qquad \texttt{has\_type N B))}\end{aligned}$$

By Theorem 5.5 $xN_1 \ldots N_n$ is canonical. By Theorem 5.18, the following two judgments hold.

$$\begin{aligned}&\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \texttt{norm\_tm } \langle\!\langle xN_1 \ldots N_n \rangle\!\rangle_\rho \ \langle\!\langle xN_1 \ldots N_n \rangle\!\rangle_\rho \\ &\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \texttt{norm\_ty } \langle\!\langle ([N_1/x_1, \ldots, N_n/x_n]B)^\beta \rangle\!\rangle_\rho \ \langle\!\langle ([N_1/x_1, \ldots, N_n/x_n]B)^\beta \rangle\!\rangle_\rho\end{aligned}$$

Thus, (29) holds if we can show that `(has_type N B)` is provable with instances $\langle\!\langle xN_1 \ldots N_n \rangle\!\rangle_\rho$ and $\langle\!\langle ([N_1/x_1, \ldots, N_n/x_n]B)^\beta \rangle\!\rangle_\rho$ for `N` and `B`, respectively, *i.e.*, we must show the following judgment holds.

$$\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \texttt{has\_type } \langle\!\langle xN_1 \ldots N_n \rangle\!\rangle_\rho \ \langle\!\langle ([N_1/x_1, \ldots, N_n/x_n]B)^\beta \rangle\!\rangle_\rho \tag{30}$$

By the encoding:

$$\langle\!\langle xN_1 \ldots N_n \rangle\!\rangle_\rho \equiv \texttt{(app\_tm ... (app\_tm } \rho(x) \ \langle\!\langle N_1 \rangle\!\rangle_\rho\texttt{) ... } \langle\!\langle N_n \rangle\!\rangle_\rho\texttt{)}.$$

We know $x : \Pi x_1 : A_1 \ldots \Pi x_n : A_n.B \in \Gamma$. Let $\texttt{X}_1, \ldots, \texttt{X}_n, \texttt{B}$ be distinct variables of type `tm` that do not occur in $\mathrm{cod}(\rho)$. For $i = 1, \ldots, n$, let $\rho^i = \langle x_1, \texttt{X}_1 \rangle + \cdots + \langle x_i, \texttt{X}_i \rangle + \rho$. By the translation:

$[\![x : \Pi x_1 : A_1 \ldots \Pi x_n : A_n.B]\!]^-_\equiv$

$$\begin{aligned}&\texttt{pi X}_1\texttt{\((}[\![x_1 \Rightarrow A_1]\!]^+_{\rho^1} \texttt{ =>} \\ &\qquad\qquad \vdots \\ &\quad \texttt{pi X}_n\texttt{\((}[\![x_n \Rightarrow A_n]\!]^+_{\rho^n} \texttt{ =>} \\ &\qquad \texttt{pi B\(norm\_ty } \langle\!\langle B \rangle\!\rangle_{\rho^n} \texttt{ B =>} \\ &\qquad\quad \texttt{has\_type (app\_tm ... (app\_tm } \rho(x) \texttt{ X}_1\texttt{) ... X}_n\texttt{) B)...)}\end{aligned}$$

It is an easy consequence of Lemma 5.9 that since for $i = 1, \ldots, n$, $x_i, \ldots, x_n$ do not appear free in A_i, $[\![x_i \Rightarrow A_i]\!]^+_{\rho^i} = [\![x_i \Rightarrow A_i]\!]^+_{\rho^n}$. Let σ be the substitution

$$\{\langle x_1, N_1 \rangle, \ldots, \langle x_n, N_n \rangle\}.$$

Then

$$([N_1/x_1, \ldots, N_n/x_n]B)^\beta = \sigma(B)^\beta \quad \text{and} \quad \sigma_{\rho^n, \rho} = \{\langle \texttt{X}_1, \langle\!\langle N_1 \rangle\!\rangle_\rho \rangle, \ldots, \langle \texttt{X}_n, \langle\!\langle N_n \rangle\!\rangle_\rho \rangle\}.$$

To prove (30), we can BACKCHAIN on the above clause with the instances of the universally quantified variables given by the substitution $\sigma_{\rho^n,\rho} \cup \{\langle \mathtt{B}, \langle\!|\sigma(B)^\beta|\!\rangle_\rho \rangle\}$. We must then show that the following judgments hold.

$$\begin{array}{l}
\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \sigma_{\rho^n,\rho}([\![x_1 \Rightarrow A_1]\!]^+_{\rho^n}) \\
\vdots \\
\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \sigma_{\rho^n,\rho}([\![x_n \Rightarrow A_n]\!]^+_{\rho^n}) \\
\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \mathtt{norm_ty}\ \sigma_{\rho^n,\rho}(\langle\!|B|\!\rangle_{\rho^n})\ \langle\!|\sigma(B)^\beta|\!\rangle_\rho
\end{array}$$

By Lemma 5.9, $\sigma_{\rho^n,\rho}(\langle\!|B|\!\rangle_{\rho^n}) = \langle\!|\sigma(B)|\!\rangle_\rho$. Since $\sigma(B) =_\beta \sigma(B)^\beta$, by Theorem 5.18, the latter judgment holds. By Lemma 5.21 (1), the first n judgments hold if the following judgments hold.

$$\begin{array}{l}
\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I [\![N_1 \Rightarrow \sigma(A_1)^\beta]\!]^+_\rho \\
\vdots \\
\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I [\![N_n \Rightarrow \sigma(A_n)^\beta]\!]^+_\rho
\end{array}$$

For $i = 1, \ldots, n$, $\sigma(A_i)^\beta = ([N_1/x_1, \ldots, N_{i-1}/x_{i-1}]A_i)^\beta$. Thus, the above judgments hold by the induction hypothesis for the latter n premises of the (C-APP-OBJ) rule.

Case: (C-APP-FAM). Similar to (C-APP-OBJ). ■

Theorem 5.23 (Correctness of Translation II)

Let Γ be a valid canonical context and α a canonical judgment. Let ρ be a variable encoding well-defined on Γ and α. Let Σ be a signature that includes the declarations of `lfsig`, `lfconv`, `lfnorm`, and the variables in $\mathrm{cod}(\rho)$. Let $\mathcal{P}$ be a set of definite clauses that includes `lfconv`, `lfnorm`, and $\mathcal{P}_{\mathrm{cod}(\rho)}$. If $\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I [\![\alpha]\!]^+_\rho$ is provable, then $\Gamma \vdash \alpha$ in C-LF.

Proof: The judgment α has the form $K \Rightarrow \mathrm{kind}$, $A \Rightarrow K'$, or $M \Rightarrow A'$. The proof is by induction on the structure of K, A, M respectively.

Case: M is an abstraction. Then M has the form $\lambda x : A.M$ and A' has the form $\Pi x : A.B$. We assume that the variable x does not appear free in Γ, otherwise we rename it. Let `X` be a variable of type `tm` such that $\mathtt{X} \notin \mathrm{cod}(\rho)$. By the translation:

$[\![\lambda x : A.M \Rightarrow \Pi x : A.B]\!]^+_\rho \equiv$

$$[\![A \Rightarrow \mathsf{Type}]\!]^+_\rho\ \mathtt{,\ pi\ X\backslash}\Big(\mathtt{normal\_tm1\ X\ =>}\ [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho}\ \mathtt{=>}\ [\![M \Rightarrow B]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\Big)$$

Thus, the following judgments hold.

$$\begin{array}{l}
\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I [\![A \Rightarrow \mathsf{Type}]\!]^+_\rho \\
\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \mathtt{pi\ X\backslash}\Big(\mathtt{normal_tm1\ X\ =>}\ [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho}\ \mathtt{=>}\ [\![M \Rightarrow B]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}\Big)
\end{array}$$

By the induction hypothesis, the assertion $\Gamma \vdash A \Rightarrow \mathsf{Type}$ is provable. By GENERIC, followed by AUGMENT twice, the following judgment holds.

$$\Sigma \cup \{\mathtt{X} : \mathtt{tm}\}; \mathcal{P}, [\![\Gamma]\!]^-_\rho, \mathtt{normal\_tm1\ X}, [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} \vdash_I [\![M \Rightarrow B]\!]^+_{\langle x,\mathtt{X}\rangle+\rho}$$

It is an easy consequence of Corollary 5.10 that since x is not free in Γ, $[\![\Gamma]\!]^-_\rho = [\![\Gamma]\!]^-_{\langle x,\mathtt{X}\rangle+\rho}$. Thus

$$[\![\Gamma]\!]^-_{\langle x,\mathtt{X}\rangle+\rho}, [\![x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho} = [\![\Gamma, x : A]\!]^-_{\langle x,\mathtt{X}\rangle+\rho}.$$

So by the induction hypothesis, the assertion $\Gamma, x : A \vdash M \Rightarrow B$ is provable. Thus, by an application of (A-ABS-OBJ), $\Gamma \vdash \lambda x : A.M \Rightarrow \Pi x : A.B$ is provable.

Case: A is an abstraction. This case is similar to the previous one with an application of (A-ABS-FAM).

Case: A is a product and K' is Type. This case is similar to the previous one with an application of (A-PI-FAM).

Case: K is a product. This case is similar to the previous one with an application of (A-PI-KIND).

Case: M is a variable or an application. Then M has the form $xN_1 \ldots N_n$ and A' is a base type. Let `N` be a variable of type `tm` and `B` a variable of type `ty` that do not appear in $\mathrm{cod}(\rho)$. By the translation:

$[\![xN_1 \ldots N_n \Rightarrow A']\!]^+_\rho \equiv$

```
sigma N\(sigma B\(norm_tm ⟪xN1 ... Nn⟫ρ N, norm_ty ⟪A'⟫ρ B, has_type N B))
```

Then for some instances `M` and `C` of `N` and `B`, respectively, the following judgments hold.

$$\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \mathtt{norm\_tm}\ \langle\!\langle xN_1 \ldots N_n \rangle\!\rangle_\rho\ \mathtt{M}$$
$$\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \mathtt{norm\_ty}\ \langle\!\langle A' \rangle\!\rangle_\rho\ \mathtt{C}$$
$$\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \mathtt{has\_type\ M\ C}$$

Since $xN_1 \ldots N_n$ and A' are canonical, by Corollary 5.20, `M` is $\langle\!\langle xN_1 \ldots N_n \rangle\!\rangle_\rho$ and `C` is $\langle\!\langle A' \rangle\!\rangle_\rho$. By the encoding

$$\langle\!\langle xN_1 \ldots N_n \rangle\!\rangle_\rho \equiv (\mathtt{app\_tm}\ \ldots\ (\mathtt{app\_tm}\ \rho(x)\ \langle\!\langle N_1 \rangle\!\rangle_\rho)\ \ldots\ \langle\!\langle N_n \rangle\!\rangle_\rho).$$

Thus, the following judgment holds.

$$\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \mathtt{has\_type}\ (\mathtt{app\_tm}\ \ldots\ (\mathtt{app\_tm}\ \rho(x)\ \langle\!\langle N_1 \rangle\!\rangle_\rho)\ \ldots\ \langle\!\langle N_n \rangle\!\rangle_\rho)\ \langle\!\langle A' \rangle\!\rangle_\rho \quad (31)$$

The last step in a proof of the above judgment must have been a BACKCHAIN on a `has_type` clause. It is easy to see that such a clause must be the translation of an element in Γ of the form $x : \Pi x_1 : A_1 \ldots \Pi x_n : A_n.B$. We assume that $x_1, \ldots, x_n$ do not appear free in

$N_1, \ldots, N_n$, and that for $i = 1, \ldots, n$, the variables $x_i, \ldots, x_n$ do not appear free in A_i, otherwise we rename them. Let $\mathtt{X}_1, \ldots, \mathtt{X}_n$ be distinct variables of type `tm`, and `B` a variable of type `ty` that do not occur in $\mathrm{cod}(\rho)$. For $i = 1, \ldots, n$, let $\rho^i = \langle x_1, \mathtt{X}_1 \rangle + \cdots + \langle x_i, \mathtt{X}_i \rangle + \rho$. By the translation:

$[\![x : \Pi x_1 : A_1 \ldots \Pi x_n : A_n.B]\!]^-_\rho \equiv$

```
pi X1\([[x1 ⇒ A1]]+_ρ1 =>
                ⋮
    pi Xn\([[xn ⇒ An]]+_ρn =>
        pi B\(norm_ty ⟪B⟫_ρn B =>
            has_type (app_tm ... (app_tm ρ(x) X1) ... Xn) B)...)
```

It is an easy consequence of Lemma 5.9 that since for $i = 1, \ldots, n$, the variables $x_i, \ldots, x_n$ do not appear free in A_i, $[\![x_i \Rightarrow A_i]\!]^+_{\rho^i} = [\![x_i \Rightarrow A_i]\!]^+_{\rho^n}$. Let σ be the substitution

$$\{\langle x_1, N_1 \rangle, \ldots, \langle x_n, N_n \rangle\}.$$

Then, $\sigma_{\rho^n,\rho} = \{\langle \mathtt{X}_1, \langle\!\langle N_1 \rangle\!\rangle_\rho \rangle, \ldots, \langle \mathtt{X}_n, \langle\!\langle N_n \rangle\!\rangle_\rho \rangle\}$. In backchaining on the above clause to prove (31), the instances of the universally quantified variables are given by the substitution $\sigma_{\rho^n,\rho} \cup \{\langle \mathtt{B}, \langle\!\langle A' \rangle\!\rangle_\rho \rangle\}$. Thus the following judgments hold.

$$\begin{aligned}
&\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \sigma_{\rho^n,\rho}([\![x_1 \Rightarrow A_1]\!]^+_{\rho^n}) \\
&\vdots \\
&\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \sigma_{\rho^n,\rho}([\![x_n \Rightarrow A_n]\!]^+_{\rho^n}) \\
&\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I \mathtt{norm_ty}\ \sigma_{\rho^n,\rho}(\langle\!\langle B \rangle\!\rangle_{\rho^n})\ \langle\!\langle A' \rangle\!\rangle_\rho
\end{aligned} \tag{32}$$

By Lemma 5.9, $\sigma_{\rho^n,\rho}(\langle\!\langle B \rangle\!\rangle_{\rho^n}) = \langle\!\langle \sigma(B) \rangle\!\rangle_\rho$. Thus, by Corollary 5.20, $\langle\!\langle A' \rangle\!\rangle_\rho$ is $\langle\!\langle \sigma(B)^\beta \rangle\!\rangle_\rho$. Thus $A' = \sigma(B) = ([N_1/x_1, \ldots, N_n/x_n]B)^\beta$. By Lemma 5.21 (1), the following judgments also hold.

$$\begin{aligned}
&\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I [\![N_1 \Rightarrow \sigma(A_1)^\beta]\!]^+_\rho \\
&\vdots \\
&\Sigma; \mathcal{P}, [\![\Gamma]\!]^-_\rho \vdash_I [\![N_n \Rightarrow \sigma(A_n)^\beta]\!]^+_\rho
\end{aligned}$$

For $i = 1, \ldots, n$, $\sigma(A_i)^\beta = ([N_1/x_1, \ldots, N_{i-1}/x_{i-1}]A_i)^\beta$. Thus, by the induction hypothesis, the following assertions are provable in C-LF.

$$\begin{aligned}
&\Gamma \vdash N_1 \Rightarrow A_1 \\
&\Gamma \vdash N_2 \Rightarrow ([N_1/x_1]A_2)^\beta \\
&\vdots \\
&\Gamma \vdash N_n \Rightarrow ([N_1/x_1, \ldots, N_{n-1}/x_{n-1}]A_n)^\beta
\end{aligned}$$

$\Gamma \vdash \mathsf{Type} \Rightarrow \text{kind}$ holds since Γ is a valid context by assumption. Then, by an application of (C-APP-OBJ),

$$\Gamma \vdash xN_1 \ldots N_n \Rightarrow ([N_1/x_1, \ldots, N_n/x_n]B)^\beta$$

Case: A is a variable or an application. This case is similar to the previous one with an application of (C-APP-FAM).

Case: K is Type. By the assumption that Γ is a valid context, $\Gamma \vdash \mathsf{Type} \Rightarrow \text{kind}$ is provable. ■

As was the case for the results in Chapter 4 which proved the correctness of the specification of $\beta\eta$-convertibility, the theorems here illustrate a precise correspondence between proofs in LF with respect to a particular signature, and proofs in the meta-language with respect to the program obtained from the translation of this signature. In bottom-up construction of a proof in C-LF, the structure of the objects in the judgment determine which rule must be applied. For example, a judgment of the form $xN_1 \ldots N_n \Rightarrow A$ requires an application of the (C-APP-OBJ) rule, and the variable x determines exactly which signature item in Γ must be used to apply the rule. The corresponding notion in solving a logic programming goal of the form `(has_type N A)` is that the constant at the head of `N` will uniquely determine which definite clause must be used in backchaining. Since each clause corresponds to a particular signature item, it is easy to see the correspondence of this aspect of constructing proofs in the two systems. Also, consider the C-LF PI or ABS rules. In applying any of these rules in a backward fashion, a new item is added to the context in the right premise. The above proofs illustrate that this operation in LF corresponds to the logic programming operations of introducing a logic programming signature item and adding a clause which encodes information about its LF type via the GENERIC and AUGMENT search operations.

In Theorem 5.23, we assumed that the initial context was valid. We can relax this assumption, and have the program prove the context is valid before proving that a judgment holds. To do so, we introduce a third translation which takes arbitrary LF assertions as arguments and produces a goal formula. This translation again uses a variable encoding. A variable encoding is well-defined on an assertion if all of the free variables in the assertion are in its domain. We again use double brackets, but this time without a superscript. The rules for this translation are in Figure 5.8. The goal formula obtained from translating an assertion $\Gamma \vdash \alpha$ will check that each type and kind in the context Γ is valid, and then add a clause obtained by the positive translation of each context item via meta-level implication. Finally, the goal formula obtained by the negative translation of the judgment α is the innermost subgoal.

$$[\![x : K, \Gamma \vdash \alpha]\!]_\rho = [\![K \Rightarrow \text{kind}]\!]^+_\rho \texttt{, pi X\}\Big([\![x \Rightarrow K]\!]^-_{\langle x,\texttt{X}\rangle+\rho} \texttt{ => } [\![\Gamma \vdash \alpha]\!]_{\langle x,\texttt{X}\rangle+\rho}\Big)$$

$$[\![x : A, \Gamma \vdash \alpha]\!]_\rho = [\![A \Rightarrow \mathsf{Type}]\!]^+_\rho \texttt{, pi X\}\Big([\![x \Rightarrow A]\!]^-_{\langle x,\texttt{X}\rangle+\rho} \texttt{ => } [\![\Gamma \vdash \alpha]\!]_{\langle x,\texttt{X}\rangle+\rho}\Big)$$

$$[\![\vdash \alpha]\!]_\rho = [\![\alpha]\!]^+_\rho$$

Figure 5.8: Translation of Arbitrary LF Assertions

5.4 Translation of a Signature for Natural Deduction

To illustrate the translation, we present an LF signature for the fragment of N_I with the $\wedge$, $\supset$, and $\forall$ connectives only. The signature Γ_{FOL} in Figure 5.9 is taken from [HHP89]. Let

$$\begin{aligned}
i &: \mathsf{Type} \\
form &: \mathsf{Type} \\
\wedge &: form \to form \to form \\
\supset &: form \to form \to form \\
\forall &: (i \to form) \to form \\
true &: form \to \mathsf{Type} \\
\wedge\text{-I} &: \Pi A : form.\Pi B : form.true(A) \to true(B) \to true(A \wedge B) \\
\wedge\text{-E}_1 &: \Pi A : form.\Pi B : form.true(A \wedge B) \to true(A) \\
\wedge\text{-E}_2 &: \Pi A : form.\Pi B : form.true(A \wedge B) \to true(B) \\
\supset\text{-I} &: \Pi A : form.\Pi B : form.(true(A) \to true(B)) \to true(A \supset B) \\
\supset\text{-E} &: \Pi A : form.\Pi B : form.true(A) \to true(A \supset B) \to true(B) \\
\forall\text{-I} &: \Pi A : i \to form.(\Pi y : i.true(Ay)) \to true(\forall A) \\
\forall\text{-E} &: \Pi A : i \to form.\Pi t : i.true(\forall A) \to true(At)
\end{aligned}$$

Figure 5.9: LF Signature for a Fragment of N_I

ρ be the following variable encoding which will be used to translate the above signature:

$$\begin{aligned}
\{&\langle i, \texttt{i}\rangle, \langle form, \texttt{form}\rangle, \langle \wedge, \texttt{and}\rangle, \langle \supset, \texttt{imp}\rangle, \langle \forall, \texttt{forall}\rangle, \langle true, \texttt{ttrue}\rangle, \\
&\langle \wedge\text{-I}, \texttt{and_i}\rangle, \langle \wedge\text{-E}_1, \texttt{and_e1}\rangle, \langle \wedge\text{-E}_2, \texttt{and_e2}\rangle, \langle \supset\text{-I}, \texttt{imp_i}\rangle, \langle \supset\text{-E}, \texttt{imp_e}\rangle, \\
&\langle \forall\text{-I}, \texttt{forall_i}\rangle, \langle \forall\text{-E}, \texttt{forall_e}\rangle\}
\end{aligned}$$

In the previous section, we illustrated the translation with the signature item $\forall : (i \to form) \to form$, and obtained the following clause.

```
has_type (app_tm forall A) B :-
  norm_ty form B,
  sigma B1\ (norm_ty i B1, is_type B1),
  pi X\ (normal_tm1 X => (pi B2\ (norm_ty i B2 => has_type X B2)) =>
         (sigma N\ sigma B3\ (norm_tm (app_tm A X) N, norm_ty form B3,
                              has_type N B3))).
```

In general clauses obtained from the literal translation can be simplified, and this clause illustrates a few simplifications that occur quite frequently. First of all, notice that the types involved are quite simple and as a result, most of the `norm_ty` normalization subgoals are unnecessary. Removing these unnecessary subgoals, the clause becomes:

```
has_type (app_tm forall A) form :-
  is_type i,
  pi X\ (normal_tm1 X => has_type X i =>
         sigma N\ (norm_tm (app_tm A X) N, has_type N form)).
```

Clearly this clause can be further simplified by removing the `(is_type i)` subgoal since we know that this goal will always succeed since `i` is the encoding of an LF type. This subgoal is redundant, and in fact corresponds to redundant subproofs that will appear in LF proofs involving the $\forall$ constant. In general, translation of LF signature items will have both normalization and `is_type` subgoals which can be removed. The complete (and simplified) translation of the signature in Figure 5.9 is contained in the `lf_fol` and `lf_ni` modules on pages 112 and 114.

```
module lf_fol.

import lfnorm.

type    i           ty.
type    form        ty.
type    and         tm.
type    imp         tm.
type    forall      tm.
type    ttrue       ty.
type    and_i       tm.
type    and_e1      tm.
type    and_e2      tm.
type    imp_i       tm.
type    imp_e       tm.
type    forall_i    tm.
type    forall_e    tm.

is_type i.
is_type form.

has_type (app_tm (app_tm and A) B) form :- has_type A form, has_type B form.
has_type (app_tm (app_tm imp A) B) form :- has_type A form, has_type B form.
has_type (app_tm forall A) form :-
  pi X\ (normal_tm1 X => has_type X i =>
         sigma N\ (norm_tm (app_tm A X) N, has_type N form)).

is_type (app_ty ttrue A) :- has_type A form.
```

Module `lf_fol`: Translation of LF Signature for First-Order Logic

It is easy to see by inspection of the clauses in these modules that all terms and types are normalized before new `has_type` and `is_type` subgoals are formed. All terms and types in an initial goal will be in normal form if they are obtained by translating an LF judgment in canonical form. Thus, in simplifying the clauses to obtain these modules, we made the assumption that terms and types will always be in normal form before backchaining on any clause for `has_type` and `is_type`. This assumption in fact allowed us to remove several more unnecessary normalization subgoals.

The modules `lf_fol` and `lf_ni`, together with `lfsig` on page 81, `lfconv` on page 85, and `lfnorm` on page 86 make up the complete program needed to prove goals obtained from translating LF assertions with respect to the signature Γ_{FOL}. Because there is exactly one clause per signature item, it is easy to see that the `has_type` or `is_type` clause used in backchaining at each step can be uniquely determined. In the `has_type` clauses, the constant at the head of the first argument identifies the signature item from which it was translated, *e.g.*, `and`, `imp`, and `forall` in the three `has_type` clauses in the `lf_fol` module. Similarly the constant at the head of the argument in `is_type` predicates identifies its corresponding signature item. The only "non-deterministic" aspect of this program are the programs for β-convertibility, *e.g.*, because of the clause for symmetry. These programs are used by the clauses for normalization. Since normalization, rather than full convertibility is what is actually required for type checking LF assertions, we can clearly modify the program for normalization so that it operates more directly and even deterministically. By replacing `lfnorm` and `lfconv` with a deterministic program, we obtain a program that operates as a complete and deterministic type checker under the interpreter described in Section 2.4 for goals obtained by translation of LF assertions with respect to the signature Γ_{FOL}. With a deterministic normalization program, the translation of any LF signature will in fact provide a deterministic type checker.

The normalization program can in fact be simplified even further. Notice that although arbitrary signatures and judgments can be translated, the resulting clauses and goals never contain terms of type `ki` nor types containing the constants `abs_ty` and `prod_ty`. As a result, in `lfnorm`, neither the `norm_ki` nor the full `norm_ty` programs will be necessary. This simplification can, of course, be incorporated into any deterministic program for normalization.

In Sections 3.2 and 3.4, we discussed the direct specification and representation of proofs for the N_I proof system. It is interesting to compare the clauses presented there with those obtained by translating the LF signature for this proof system. Consider clauses from each specifying the $\supset$-I rule. The clauses from the `niprover` module of Section 3.4 and the `lf_ni` module are repeated below for comparison.

```
module lf_ni.

import lf_fol lfnorm.

has_type (app_tm (app_tm (app_tm (app_tm and_i A) B) P1) P2)
         (app_ty ttrue (app_tm (app_tm and A) B)) :-
  has_type A form, has_type B form,
  has_type P1 (app_ty ttrue A), has_type P2 (app_ty ttrue B).

has_type (app_tm (app_tm (app_tm and_e1 A) B) P) (app_ty ttrue A) :-
  has_type A form, has_type B form,
  has_type P (app_ty ttrue (app_tm (app_tm and A) B)).

has_type (app_tm (app_tm (app_tm and_e2 A) B) P) (app_ty ttrue B) :-
  has_type A form, has_type B form,
  has_type P (app_ty ttrue (app_tm (app_tm and A) B)).

has_type (app_tm (app_tm (app_tm imp_i A) B) P)
         (app_ty ttrue (app_tm (app_tm imp A) B)) :-
  has_type A form, has_type B form,
  pi Q\ (normal_tm1 Q => has_type Q (app_ty ttrue A) =>
         sigma N\ (norm_tm (app_tm P Q) N, has_type N (app_ty ttrue B))).

has_type (app_tm (app_tm (app_tm (app_tm imp_e A) B) P1) P2)
         (app_ty ttrue B) :-
  has_type A form, has_type B form,
  has_type P1 (app_ty ttrue A),
  has_type P2 (app_ty ttrue (app_tm (app_tm imp A) B)).

has_type (app_tm (app_tm forall_i A) P) (app_ty ttrue (app_tm forall A)) :-
  pi X\ (normal_tm1 X => has_type X i =>
         sigma N\ (norm_tm (app_tm A X) N, has_type N form)),
  pi Y\ (normal_tm1 X => has_type Y i =>
         sigma N\ (sigma B\ (norm_tm (app_tm P Y) N,
                             norm_ty (app_ty ttrue (app_tm A Y)) B,
                             has_type N B))).

has_type (app_tm (app_tm (app_tm forall_e A) T) P) B :-
  norm_ty (app_ty ttrue (app_tm A T)) B,
  pi X\ (normal_tm1 X => has_type X i =>
         sigma N\ (norm_tm (app_tm A X) N, has_type N form)),
  has_type T i,
  has_type P (app_ty ttrue (app_tm forall A)).
```

Module `lf_ni`: A Slight Simplification of the Translation of an LF Signature for Natural Deduction

```
(imp_i P) # (A imp B) :- pi PA\ ((PA # A) => ((P PA) # B)).

has_type (app_tm (app_tm (app_tm imp_i A) B) P)
         (app_ty ttrue (app_tm (app_tm imp A) B)) :-
  has_type A form, has_type B form,
  pi PA\ (normal_tm1 PA => has_type PA (app_ty ttrue A) =>
         sigma N\ (norm_tm (app_tm P PA) N, has_type N (app_ty ttrue B))).
```

Of course, the latter clause is a bit more complicated since it is obtained via a general translation algorithm on a very general specification language, while the former is simpler since it uses features of the meta-language more directly. For example, the special predicate `#` was introduced specifically for relating formulas and terms representing N_I proofs. Additionally, `i` and `form` were introduced as meta-types instead of meta-terms of type `ty` and thus the encoding of terms and type-checking subgoals for terms of type `i` and `form` are not needed. Also, β-conversion of the meta-language is used directly for substitution at the object level, while in the latter substitution it is handled via β-normalization in LF (encoded here via `norm_tm`, `norm_ty`, and `norm_ki`). Despite these differences, both make use of universal quantification and implication in the same way to handle the discharge of assumptions. Operationally, in both cases, GENERIC and AUGMENT are used to first introduce a new signature item, and then add a clause asserting that this signature item is a proof of the formula `A`. As another example, consider the two clauses from `niprover` and `lf_ni` respectively for the $\forall$-I rule.

```
(forall_i P) # (forall A) :- pi Y\ ((P Y) # (A Y)).

has_type (app_tm (app_tm forall_i A) P) (app_ty ttrue (app_tm forall A)) :-
  pi X\ (normal_tm1 X => has_type X i =>
         sigma N\ (norm_tm (app_tm A X) N, has_type N form)),
  pi Y\ (normal_tm1 X => has_type Y i =>
         sigma N\ (sigma B\ (norm_tm (app_tm P Y) N,
                             norm_ty (app_ty ttrue (app_tm A Y)) B,
                             has_type N B))).
```

Both use universal quantification at the meta-level to handle the proviso on this rule. In the latter, implication is also necessary to associate the type `i` with the quantified variable `Y`, while in the former, this is not necessary since `Y` has meta-type `i`.

In Chapter 3, we presented several choices in representing proof terms for natural deduction. The clauses above from `niprover` use the "intermediate" proof representation discussed in Section 3.4. Note that the translation of LF signatures provides yet another representation. For example, the proof term in `lf_ni` whose head is `and_i` has four arguments: two formulas and two proof terms. The proof term whose head is `forall_i` has two arguments: abstractions from terms of type `i` to a formula and to a proof term respectively. This representation in fact corresponds to the "maximal" representation of proof terms mentioned in Section 3.4. Thus proof terms obtained by translating LF judgments contain enough information to fully reconstruct the deduction trees to which they correspond.

Chapter 6

Executing Specifications Directly

In specifying theorem provers for various logics in Chapters 3 and 4, we considered not only the declarative aspects of the specifications but also their operational behavior with respect to a non-deterministic interpreter. Here, we consider executing these specifications using the deterministic depth-first interpreter described in Chapter 2. In this chapter (and the remainder of this dissertation), we assume that logic program signatures contain only constants. Thus, when we speak of "free variables," we will always mean logic variables.

As executable programs, the specifications in these chapters may serve several different purposes depending on the content of a query to the program. For example, in a query of the form `(P # A)` to the program made up of the clauses specifying N_I in Section 3.2, if initially both `P` and `A` are closed terms, the program behaves as a proof checker. In fact, for proof checking queries, variables of type `i` are allowed to appear in `P` and `A` since determining the value of these variables will require only first-order unification. On the other hand, if `P` is a logic variable, or a partially specified proof term (*i.e.*, contains logic variables with types other than `i`), the program behaves as a theorem prover, and `P` gets constructed as the interpreter proceeds in attempting to solve the query. Similarly, all of the specifications of natural deduction and sequent proof systems in Chapters 3 and 4 can be viewed as both proof checkers and theorem provers.

Another relation, the `#t` relation between terms and their types was introduced in the type-checking specifications in Sections 4.2 and 4.4. When both arguments are specified in a query of the form `(T #t S)`, the programs behave as type checkers. In queries where the first argument, `T`, is specified and the second argument, `S`, is left fully or partially unspecified, the program performs type inference. By the Curry-Howard isomorphism relating proof checking to type checking [How80], the two kinds of relations in Chapters 3 and 4, one between a formula and its proof, and the other between a term and its type can be viewed similarly. As a result, in this discussion we will consider each of the programs uniformly in three ways: as proof or type checkers (both arguments specified), as theorem provers (first argument unspecified), and as type inferencers (second argument unspecified).

6.1 A Depth-First Strategy for Proof Checking in First-Order Logic

We first consider the behavior of the specifications for first-order logic as proof checking programs with respect to the depth-first interpreter. Consider a query of the form `(Q >- Gamma --> A)` to the specification in Section 3.1 of the L_I sequent system for first-order intuitionistic logic. In an initial proof checking query, the proof term `Q` will be closed or possibly contain logic variables of type `i`. As a result, all subsequent goals will also be closed (or contain variables of type `i`). The top-level constant of a proof term completely determines the unique definite clause which can be used in backchaining. If the clause fails, there will be no other to try. As a result, under depth-first search, the program will always succeed or fail, indicating whether or not the proof term represents a proof of the sequent.

Note that at certain points, some backtracking may be necessary. For example, when backchaining on the following clause for `or_r`:

```
(or_r Q) >- (Gamma --> (A or B)) :- Q >- (Gamma --> A); Q >- (Gamma --> B).
```

the interpreter will first try to show that the subproof `Q` is a proof of the sequent `(Gamma --> A)` where `A` is the first disjunct, and if that fails, will backtrack and try the second, `(Gamma --> B)`. Also, backtracking may be necessary whenever there is more than one hypothesis with the same top-level connective. For example, the following clause for `imp_l`:

```
(imp_l Q1 Q2) >- (Gamma --> C) :- memb (A imp B) Gamma,
                                  Q1 >- (Gamma --> A),
                                  Q2 >- ((B::Gamma) --> C).
```

always operates on the first implication in the hypothesis list first. If failure is encountered, the interpreter will backtrack and try the next, and so on.

In a proof checking query, new logic variables of type `i` may be introduced during execution. For example, in the clause for `exists_r`:

```
(exists_r Q) >- (Gamma --> (exists A)) :- sigma T\ (Q >- (Gamma --> (A T))).
```

a logic variable is introduced for `T` into the subgoal (whenever the quantification on `A` is not vacuous). This term corresponds to the substitution term used in applying the rule. The clause for `forall_l` similarly introduces a logic variable. Such logic variables will always get resolved, though not necessarily instantiated to closed terms, in backchaining on the following clause for `initial`.

```
(initial A) >- (Gamma --> A) :- memb A Gamma.
```

Note that if the proof term in an initial query is closed, the term `(initial A)` will be closed. Thus any logic variables of type `i` in the terms matching the other occurrences of `A` in this clause will get instantiated to closed terms.

A similar analysis applies to all of the specifications for proof systems for first-order logic where the constant at the head of the proof term uniquely determines which definite clause must be used at each step. Thus most of the specifications for N_I including the specification of Section 3.2, the explicit context specification of Section 3.3, the `niprover` module on page 34 of Section 3.4, and the first presentation of a specification that constructs normal proofs in Section 3.5 may serve as programs for proof checking. In addition the specifications discussed in Section 4.1 for the L_C and N_C proof systems for classical logic may also serve as programs for proof checking. In each of these programs there are backtracking points, but they do not affect the completeness of the programs as proof checkers. For example, the following clause specifying the $\forall$-E rule illustrates that backtracking may be necessary when there is more than one possible unifier for logic variables.

```
(forall_e P) # (A T) :- P # (forall A).
```

In backchaining over this clause, the formula in the goal must match the pattern `(A T)`. This requires second-order matching and may have more than one solution.

In several of the specifications for natural deduction, in addition to logic variables of type `i`, logic variables of type `form` may be introduced in the subgoals of the clauses for the elimination rules. For example, in both of the following clauses specifying the $\supset$-E rule, the logic variable `A` is introduced.

```
(imp_e P1 P2) # B :- P1 # A, P2 # (A imp B).
```

```
Gamma --> (imp_e P1 P2) # B :- Gamma --> P1 # A, Gamma --> P2 # (A imp B).
```

Like logic variables of type `i`, these variables will not affect the completeness of the programs as proof checkers. Such logic variables will not occur during execution of `niprover` as a proof checker because of the presence of formulas inside proof terms. For example, in backchaining over the following clause:

```
(imp_e A P1 P2) # B :- P1 # A, P2 # (A imp B).
```

if the proof term `(imp_e A P1 P2)` does not contain logic variables of type `form`, `A` will not be a logic variable, as it is in the two clauses above.

Even when a query to any of these programs contains free variables of type `form` in the formula or sequent to be proven (a "type inference" query), the behavior is similar. There may be additional backtracking points in finding unifiers for these variables, but as long as there is a constant at the head of the proof term at each step, there will always be only one definite clause that is applicable, and thus execution will always terminate with success or failure. For a query where the formula or sequent to be proven contains logic variables of type `form`, upon successful completion, some formulas may still contain logic variables. For example, in the specifications of N_I, the natural deduction proof term `(imp_i X\X)` is a proof of `(P imp P)` for variable `P`. We may think of this formula as a "polymorphic

type," *i.e.*, the proof term is a proof of `(P imp P)` where `P` may be instantiated with any formula.

The `ninormal` module on page 39 of Section 3.5 cannot serve as a proof checker. To see why, consider the clause for `and_e` below.

```
PC # C :- P #e (A and B),
          (((and_e1 B P) #e A) => (((and_e2 A P) #e B) => (PC # C))).
```

The head of the clause will unify with any atomic goal of the form `(PC # C)`. If, for example, an atomic clause of the form `(P #e (A and B))` is added to the program, this clause will then always be applicable and could cause the interpreter to enter an infinite loop. The clauses for `imp_e` and `forall_e` are similar. Thus, this specification is too "non-deterministic" even to serve as a proof checker. On the other hand, in contrast to other specifications for N_I, logic variables of type `form` will never be introduced into proof checking subgoals during execution. To illustrate this point, we contrast the following two clauses for the $\supset$-E rule.

```
(imp_e P1 P2) #e B :- P1 # A, P2 #e (A imp B).
```

```
PC # C :- P2 #e (A imp B), P1 # A,
          (((imp_e A P1 P2) #e B) => (PC # C)).
```

The first introduces the logic variable `A` of type form into subgoals. The second is from the `ninormal` module. In executing `ninormal` as a proof checker, all clauses for the `#e` predicate will be atomic clauses added to the program dynamically, and all formulas on the right of `#e` in these clauses will be subformulas of the formula in the original query. Thus, as long as the formula to be proven in the original query is fully specified except possibly for free variables of type `i`, the instances of `(A imp B)` in the above clause, and thus also `A` in the subgoal `(P1 # A)` will not be logic variables, or contain logic variables of type `form`. These clauses illustrate the two kinds of non-determinism in proof checking queries to specifications of N_I. The first clause can introduce logic variables of type `form` into subgoals, yet is from a program which can serve as a proof checker because the constant at the head of the proof term uniquely determines which clause to use at each step. In contrast, the `ninormal` program in which the second clause appears never introduces logic variables of type `form`, but can enter an infinite loop during proof checking since it contains this clause and two others whose head contains a logic variable `PC` that can unify with any proof term.

6.2 Depth-First Theorem Proving in First-Order Logic

Theorem proving is a much more complicated task than proof checking, and simple-minded depth-first search will rarely be sufficient. First, consider the specification for L_I in Section 3.1. When proof terms in queries are variables, in general there will be multiple

definite clauses that could be applied to any one sequent. Also, if there is any non-atomic hypothesis on the left of the sequent arrow, the rule for its top level connective will always be applicable. For example, consider the clause for `and_l`.

```
(and_l Q) >- (Gamma --> C) :- memb (A and B) Gamma,
                              Q >- ((A::B::Gamma) --> C).
```

Just as for proof checking using `ninormal`, whenever there is a conjunctive hypothesis, the interpreter may enter an infinite loop repeatedly applying the rule. One possible solution is to specify each definite clause so that the principle formula is removed from the list, and does not appear in any of the subgoals of the clause. This approach may sacrifice completeness, since in any first-order sequential proof system, there are certain formulas that may need to be used more than once. Various formulations of sequent systems handle duplication of formulas that may be used more than once in different ways. In the L_I system of Section 3.1 sets are used to represent hypotheses and as a result, each rule that introduces a formula on the left of a sequent arrow allows duplication of the principle formula. In the following rule for $\forall$-L, for example:

$$\frac{[t/x]A, \Gamma \longrightarrow C}{\forall x A, \Gamma \longrightarrow C} \forall\text{-L}$$

the conclusion contains the set of hypotheses $\forall x A, \Gamma$, where the principle formula $\forall x A$ may appear in Γ. As a result, $\forall x A$ may appear in both the conclusion and premise of an application of the rule. The Gentzen LJ system [Gen69], another formulation of a sequent proof system for first-order intuitionistic logic, uses lists to represent hypotheses. In that system, the principle formula is always removed from the list when applying a rule. The $\forall$-L rule, for example, is formulated exactly as above, but $\forall x A, \Gamma$ is the list such that the principle formula $\forall x A$ is the head, and Γ is the tail. LJ includes the following contraction rule for explicitly duplicating formulas.

$$\frac{A, A, \Gamma \longrightarrow C}{A, \Gamma \longrightarrow C} \text{contract}$$

Thus, the duplication of formulas is isolated to one rule. Of course, a direct specification of such a rule will cause similar looping problems in a depth-first interpreter.

The L_C and LK systems for classical logic are similar to the L_I and LJ systems except that sets and lists, respectively, are also used to represent conclusions which appear on the right side of the sequent arrow, and in each case, duplication of conclusions is handled exactly as duplication of hypotheses. For classical logic, it is known that only universally quantified hypotheses, and existentially quantified conclusions need to be duplicated for multiple use. The sequent system in [Kle67], and the similar Gentzen system G in [Gal86] reflect this fact by restricting duplication to occur only in the $\forall$-L and $\exists$-R rules.

$$\frac{[t/x]A, \forall x A, \Gamma \longrightarrow \Delta}{\forall x A, \Gamma \longrightarrow \Delta} \forall\text{-L} \qquad \frac{\Gamma \longrightarrow [t/x]A, \exists x A, \Delta}{\Gamma \longrightarrow \exists x A, \Delta} \exists\text{-R}$$

Of course, the program given by the direct specification of these proof systems will still contain two clauses that may cause looping. We now illustrate that we can exploit the fact that only two kinds of formulas ever need to be duplicated to modify the direct specification and obtain a program that implements a complete theorem prover with respect to a depth-first interpreter.

If we restrict the proof system G to propositional rules, specification of these rules will provide an implementation of a theorem prover for propositional logic that is complete with respect to a depth-first interpreter. In any proof tree for a propositional sequent, it will always be the case that the sequents at any one level will contain one less logical connective than the sequents in the level below. Termination is guaranteed in searching for a proof using such a specification, since although the number of sequents to prove may increase as the proof branches, each sequent will eventually be reduced to a sequent containing only atoms. (See [Mil83] for an analysis of the complexity.) If the original sequent is provable, the sequents containing only atoms will all be initial sequents. Otherwise the theorem prover will fail.

A strategy for a theorem prover for full first-order logic can be described as follows: (1) Apply all rules except $\forall$-L and $\exists$-R until nothing more can be done. The result is a set of sequents with atomic and universally quantified formulas on the left, and atomic and existentially quantified formulas on the right. Iterate the following steps for each resulting sequent. (2) Apply all rules including versions of the rules for $\forall$-L and $\exists$-R that remove the quantified formula after applying the rule, and try to complete the proof. (3) Stop if a proof is successfully completed. Otherwise, add an additional copy of each quantified formula to the sequents obtained from step 1, and repeat.

A program applying this strategy is contained in the modules `lc_prove`, `lc_iter`, and `lc_auto` on pages 123-125. The complete program includes two modules of declarations: the `fol` module from Chapter 3 (page 18) introducing the connectives of first-order logic, and the `lprf` module on page 122 introducing the proof constructors for L_I and L_C proofs. (Only those for cut-free L_C are used in this program.) This program uses two relations, `>-1` and `>-2`, between proofs and sequents. All of the rules except $\forall$-L and $\exists$-R are specified using the first relation, and all of the rules including versions of $\forall$-L and $\exists$-R that remove the principle formula after applying the rule are specified using the second. The `memb_and_rest` predicate is used in this program to remove the principle formula from the list of hypotheses or conclusions. It is similar to the `memb` program but includes an extra argument for the list with the selected item removed. See Appendix A for its definition. Clauses for the `>-1` predicate are in the root module `lc_auto`. They are used to perform step 1 and then begin the iteration by calling the `nprove` predicate which appears in the `lc_iter` module imported by `lc_auto`. `nprove` performs the iteration by repeatedly calling first the `amplify` predicate to add an additional copy of each quantified formula,

```
module lprf.

import fol.

kind    lprf            type.

type    initial         form -> lprf.
type    and_l           lprf -> lprf.
type    and_r           lprf -> lprf -> lprf.
type    or_l            lprf -> lprf -> lprf.
type    or_r            lprf -> lprf.
type    or_r1           lprf -> lprf.
type    or_r2           lprf -> lprf.
type    imp_r           lprf -> lprf.
type    imp_l           lprf -> lprf -> lprf.
type    neg_r           lprf -> lprf.
type    neg_l           lprf -> lprf.
type    exists_r        i -> lprf -> lprf.
type    exists_l        (i -> lprf) -> lprf.
type    forall_r        (i -> lprf) -> lprf.
type    forall_l        i -> lprf -> lprf.
type    false_r         lprf -> lprf.
type    cut             lprf -> lprf -> lprf.
```

Module `lprf`: Proof Term Constructors for L_I and L_C

and then `>-2` to apply all the rules and attempt to complete the proof. The clauses for `>-2` are in `lc_prove` which is imported by `lc_iter`. The program stops only when a proof is successfully completed.

The order of the the clauses in `lc_auto` and `lc_prove` for the `>-1` and `>-2` predicates is not important except that the clause in `lc_auto` that calls the `nprove` program must be placed last. The others could be placed in any order without affecting the completeness of the theorem prover. As presented on pages 123 and 125, the clauses that add or remove formulas from the right hand side of a sequent are placed last. Of the remaining rules, those that do not cause branching in the proof search, *i.e.*, those that correspond to inference rules with only one premise are placed first, followed by those that cause branching.

```
module lc_prove.

import  lprf lists.

kind    seq     type.

type    '-->'   (list form) -> (list form) -> seq.
type    '>-2'   lprf -> seq -> o.

(initial A) >-2 (Gamma --> Delta) :- memb A Gamma, memb A Delta.

(and_l Q) >-2 (Gamma --> Delta) :-
   memb_and_rest (A and B) Gamma Gamma1, Q >-2 ((A::B::Gamma1) --> Delta).

(imp_r Q) >-2 (Gamma --> Delta) :-
   memb_and_rest (A imp B) Delta Delta1, Q >-2 ((A::Gamma) --> (B::Delta1)).

(exists_l Q) >-2 (Gamma --> Delta) :-
   memb_and_rest (exists A) Gamma Gamma1,
   pi T \ ((Q T) >-2 (((A T)::Gamma1) --> Delta)).

(forall_r Q) >-2 (Gamma --> Delta) :-
   memb_and_rest (forall A) Delta Delta1,
   pi T \ ((Q T) >-2 (Gamma --> ((A T)::Delta1))).

(exists_r T Q) >-2 (Gamma --> Delta) :-
   memb_and_rest (exists A) Delta Delta1, Q >-2 (Gamma --> ((A T)::Delta1)).

(forall_l T Q) >-2 (Gamma --> Delta) :-
   memb_and_rest (forall A) Gamma Gamma1, Q >-2 (((A T)::Gamma1) --> Delta).

(or_l Q1 Q2) >-2 (Gamma --> Delta) :-
   memb_and_rest (A or B) Gamma Gamma1,
   Q1 >-2 ((A::Gamma1) --> Delta), Q2 >-2 ((B::Gamma1) --> Delta).

(and_r Q1 Q2) >-2 (Gamma --> Delta) :-
   memb_and_rest (A and B) Delta Delta1,
   Q1 >-2 (Gamma --> (A::Delta1)), Q2 >-2 (Gamma --> (B::Delta1)).

(imp_l Q1 Q2) >-2 (Gamma --> Delta) :-
   memb_and_rest (A imp B) Gamma Gamma1,
   Q1 >-2 ((B::Gamma1) --> Delta), Q2 >-2 (Gamma1 --> (A::Delta)).

(neg_r Q) >-2 (Gamma --> Delta) :-
   memb_and_rest (neg A) Delta Delta1, Q >-2 ((A::Gamma) --> Delta1).

(or_r Q) >-2 (Gamma --> Delta) :-
   memb_and_rest (A or B) Delta Delta1, Q >-2 (Gamma --> (A::B::Delta1)).

(neg_l Q) >-2 (Gamma --> Delta) :-
   memb_and_rest (neg A) Gamma Gamma1, Q >-2 (Gamma1 --> (A::Delta)).
```

Module `lc_prove`: Main Search Component for Automatic Theorem Prover for L_C

```
module lc_iter.

import  lc_prove.

type    add_copies      int -> form -> (list form) -> (list form) -> o.
type    amplify_forall  int -> (list form) -> (list form) -> o.
type    amplify_exists  int -> (list form) -> (list form) -> o.
type    amplify         int -> seq -> seq -> o.
type    nprove          int -> lprf -> seq -> o.

add_copies 1 A Lis (A::Lis).
add_copies N A Lis (A::Lis1) :-
  (N > 1), M is (N - 1),
  add_copies M A Lis Lis1.

amplify_forall N nil nil.
amplify_forall N ((forall A)::Gamma) Gamma2 :-
  amplify_forall N Gamma Gamma1,
  add_copies N (forall A) Gamma1 Gamma2.
amplify_forall N (A::Gamma) (A::Gamma1) :-
  amplify_forall N Gamma Gamma1.

amplify_exists N nil nil.
amplify_exists N ((exists A)::Delta) Delta2 :-
  amplify_exists N Delta Delta1,
  add_copies N (exists A) Delta1 Delta2.
amplify_exists N (A::Delta) (A::Delta1) :-
  amplify_exists N Delta Delta1.

amplify 1 Seq Seq :- !.
amplify N (Gamma1 --> Delta1) (Gamma2 --> Delta2) :-
  amplify_forall N Gamma1 Gamma2,
  amplify_exists N Delta1 Delta2.

nprove N Q Seq1 :-
  nl, writesans "Attempting to prove the following sequent at amplification ",
  write N, nl, write Seq1, nl,
  amplify N Seq1 Seq2,
  Q >-2 Seq2,
  writesans "successful".
nprove N Q Seq :-
  M is (N + 1), nprove M Q Seq.
```

Module `lc_iter`: Iteration Loop for Automatic Theorem Prover for L_C

```
module lc_auto.

import  lc_iter lists.

type    '>-1'   lprf -> seq -> o.

(initial A) >-1 (Gamma --> Delta) :- memb A Gamma, memb A Delta.

(and_l Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (A and B) Gamma Gamma1, Q >-1 ((A::B::Gamma1) --> Delta).

(imp_r Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Delta Delta1, Q >-1 ((A::Gamma) --> (B::Delta1)).

(exists_l Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (exists A) Gamma Gamma1,
  (pi T \ ((Q T) >-1 (((A T)::Gamma1) --> Delta))).

(forall_r Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (forall A) Delta Delta1,
  (pi T \ ((Q T) >-1 (Gamma --> ((A T)::Delta1)))).

(or_l Q1 Q2) >-1 (Gamma --> Delta) :-
  memb_and_rest (A or B) Gamma Gamma1,
  Q1 >-1 ((A::Gamma1) --> Delta), Q2 >-1 ((B::Gamma1) --> Delta).

(and_r Q1 Q2) >-1 (Gamma --> Delta) :-
  memb_and_rest (A and B) Delta Delta1,
  Q1 >-1 (Gamma --> (A::Delta1)), Q2 >-1 (Gamma --> (B::Delta1)).

(imp_l Q1 Q2) >-1 (Gamma --> Delta) :-
  memb_and_rest (A imp B) Gamma Gamma1,
  Q1 >-1 ((B::Gamma1) --> Delta), Q2 >-1 (Gamma1 --> (A::Delta)).

(neg_r Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (neg A) Delta Delta1, Q >-1 ((A::Gamma) --> Delta1).

(or_r Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (A or B) Delta Delta1, Q >-1 (Gamma --> (A::B::Delta1)).

(neg_l Q) >-1 (Gamma --> Delta) :-
  memb_and_rest (neg A) Gamma Gamma1, Q >-1 (Gamma1 --> (A::Delta)).

Q >-1 Seq :- nprove 1 Q Seq.
```

Module `lc_auto`: Root Module for Automatic Theorem Prover for L_C

The `lc_prove` module contains the propositional rules plus versions of the quantifier rules that remove the quantified formula from the sequent. As in a propositional theorem prover, in executing this program it will be the case that in any proof tree, the sequents at any one level will contain one less logical connective than the sequents in the level below. Thus step (2) above will always terminate. The duplication of quantified formulas at step (3) in the above procedure is reminiscent of the amplification step in the matings procedure for automated theorem proving given in [And81]. Although the matings method is quite different from the one outlined above, the completeness of the above procedure follows from the fact proved in [And81] that duplication of outermost quantifiers is all that is necessary to obtain a complete procedure, and the fact that step (2) will always terminate. If there is a proof, the procedure will find it after some finite number of iterations, but will never terminate for sequents that are not provable.

It is interesting to compare this theorem prover to the well-known UT prover which is also an implemention of a theorem prover based on a sequent calculus [Ble77, Ble83]. Some aspects of the UT prover have been designed to handle quantifiers and substitutions in a principled fashion. In the UT prover, the IMPLY procedure is based on a set of rules for a "Gentzen type" system for first-order logic. In this procedure, formulas keep their basic propositional structure although their quantifiers are removed. In the AND-SPLIT rule of this prover (which corresponds to the $\wedge$-R rule in a sequent system), the first conjunctive subgoal returns a substitution which must then be applied to the second subgoal before it is attempted. In logic programming, such composition of substitutions obtained from separate subgoals is handled automatically via shared logical variables. An issue that arises as a result of the AND-SPLIT rule is the occurrence of "conflicting bindings" due to the need to instantiate a quantified formula more than once. The UT prover uses *generalized substitutions* [TB79] to handle such multiple instances. A substitution is the final result returned when a complete proof is found. In contrast, in the logic programming setting, by representing quantification using λ-abstraction, we do not need to remove quantifiers before attempting a proof. Instead, the duplication of quantified formulas allows multiple instantiations by introducing a new logical variable for each one. As the result of a successful proof we obtain a proof term rather than a substitution. These proof terms are defined to include the substitution information. Such proof terms could in fact be simplified to only return substitution information. Such simplified proof terms would be very similar to the generalized substitutions of the UT prover.

Like the specifications for L_I and L_C, specifications for N_I and N_C cannot serve directly as theorem provers for natural deduction. In fact, much of the non-determinism comes from the same source as that for sequent systems: a rule can be applied to a non-atomic hypothesis infinitely many times. In specifications that use meta-level implication to add object level hypotheses as program clauses, this problem cannot be remedied. For example,

in proving a formula of the form `((p and q) imp r)`, in backchaining over the following clause for `imp_i` from Section 3.2:

```
(imp_i P) # (A imp B) :- pi PA\ ((PA # A) => ((P PA) # B)).
```

followed by applying GENERIC with constant `pa` for `PA`, and then AUGMENT, a clause of the form `(pa # (p and q))` will be added to the program. From this point on, in attempting to prove `r` the following clause for `and_e` will always be applicable.

```
(and_e P) # A :- P # (A and B); P # (B and A).
```

Once a hypothesis is added it cannot be removed (except after the goal that introduced it has succeeded or failed). As a result, although meta-level implication is very useful in specification, it will not be practical for implementing theorem provers. The alternate way of specifying natural deduction, using lists to handle assumptions, provides more control. This control is necessary for regulating the duplication of hypotheses, an essential aspect of developing complete implementations under depth-first search, and as we will see in Chapter 7, a useful mechanism in interactive proof search.

6.3 Depth-First Search With a Higher-Order Object Logic

We next consider various queries to the programs of Sections 4.2 and 4.4 specifying type-checking and proof rules for a higher-order logic. We first consider the specification for the simply-typed λ-calculus and queries of the form `(M #t S)` to the two-line program given by the definite clauses for the `#t` predicate in Section 4.2.

```
(abs M) #t (R --> S) :- pi X\ ((X #t R) => ((M X) #t S)).
(app M N) #t S :- M #t (R --> S), N #t R.
```

As in first-order logic, the constant at the head of a closed term uniquely determines the definite clause that must be used at each backchaining step. If the clauses in Section 4.4 for type-checking formulas and terms built up from the non-logical constants are included, the same analysis applies. Thus depth-first interpretation of this specification is complete for type checking.

Type inference for the `#t` program is also complete for the same reason: the constant at the head of a closed term uniquely determines the definite clause that must be used at each backchaining step. In a successful query of the form `(M #t S)` where `S` starts out unspecified, the resulting instantiation for `S` may contain logic variables. For example, the term

```
(abs F\ (abs N\ (app F (app F N))))
```

will have inferred "polymorphic" type `(R --> S) --> R --> S`. If types of bound variables in abstractions are included as an argument to `abs`, and the program is modified accordingly, closed terms will get closed types. For example, the term

```
(abs F\ (abs N\ (app F (app F N)) i) (i -->i))
```

will get type `(i --> i) --> i --> i`.

Theorem proving, or asking if a type is inhabited by a term, is not feasible in the program for `#t` as it is specified. The clause for `app` will always be applicable and could take the interpreter down an infinite path.

In proof checking goals of higher-order logic of the form `(P #p A)`, when both the proof `P` and the formula `A` are closed, in all subgoals of the form `(Q #p B)`, `Q` will be closed, and in all subgoals of the form `(M #t S)`, `M` will be closed. Thus, for all subgoals of either form, there will be only one definite clause to use at each step. The λ-convertibility program, on the other hand, is very non-deterministic. The clause for symmetry alone can cause a depth-first interpreter to go into an infinite loop. There are many ways to specify the equality relation between λ-terms. We could choose a "more deterministic" way to check equality between typed terms such as always reducing them to normal form before comparing them. The `lfnorm` module in Chapter 5 for normalization in LF illustrated one approach to term normalization, yet the programs given there were not deterministic since the normalization programs made use of the non-deterministic convertibility programs in the `lfconv` module. The following clauses illustrate a deterministic approach to normalization for simply typed terms.

```
norm M N :- red1 M P, !, norm P N.
norm M M.
```

The cut (`!`) and order of clauses are essential to the correct behavior of this program. The second clause must not be used until all possible reductions are done by the first clause. In the specification in Section 4.4, the `conv` program could be replaced with this `norm` program, and the convertibility clause modified as follows.

```
(convert A B P) #p A :- B #t form, P #p B, norm A N, norm B N.
```

With this modification, we obtain a complete deterministic proof checker for our higher-order logic. Since termination depends only on the fact that the proof term is closed, the program will still be complete for queries of the form `(P #p A)` where the formula `A` is not fully specified.

Note that the top-level predicate `#` given by the following single definite clause:

```
P # A :- A #t form, P #p A.
```

cannot be used when `A` is not specified since it will first try to check that variable `A` is a formula. As just discussed, queries of the form `(A #t S)` where `A` is not specified cannot be executed using depth-first control. We may simply omit the first subgoal with the understanding that when `(P # A)` is provable, `P` is a proof of all instances of `A` where the free variables of `A` get instantiated to terms of the appropriate object-level type as long as these types are inhabited.

Not surprisingly, depth-first theorem proving in higher-order logic is even more problematic than depth-first theorem proving for first-order natural deduction. There are several more sources of non-determinism. For example, consider the type-checking subgoal in the clause for `exists_i`.

```
(exists_i S T P) #p (exists S A) :- T #t S, P #p (A T).
```

When the proof term is unspecified, the term `T` will also be unspecified. In solving the subgoal `(T #t S)`, the interpreter is asked to find a term `T` that inhabits the type `S` before finding a proof `P` of the formula `(A T)`. As discussed, such a query cannot be executed deterministically under depth-first control. The two subgoals may be reversed, but only in the case when `T` gets instantiated to a closed term does the second goal becomes a type checking goal which can be handled by the deterministic interpreter. A similar problem occurs in the above clause for convertibility between λ-terms. If `B` is unspecified in the proof term `(convert A B P)`, the subgoal `(B #t form)` cannot be feasibly solved under depth-first search. Non-determinism will also be a problem in any program specifying convertibility between object level terms. We have already seen that the `conv` may loop indefinitely, even when both terms are closed. Although the implementation of a `norm` program was suggested to solve this problem, it too will fail when logic variables in terms are permitted. For example, in the above clause for convertibility, if `A` is unspecified, normalization will loop indefinitely on the subgoal `(norm A N)`.

Chapter 7

Implementing Interpreters for Theorem Provers

As we have seen, it is sometimes possible to write theorem provers that are complete with respect to a depth-first interpreter. In general, though, it is desirable to have more control over which definite clauses are applied at different points during the search for proofs. One way to provide such control is to write interpreters in our logic programming language that define some other form of control. We argue that logic programming is well-suited for the task of implementing such interpreters. We focus largely on the implementation of a tactic style interpreter. We retain depth-first control for the meta-language, so the interpreters themselves must function well under depth-first search.

7.1 Defining and Interpreting Goal Structures

First, we need to introduce new goal structures that will be manipulated by the programs that implement interpreters. In the next subsection, we define one goal constructor corresponding to each of the search operations of the logic programming interpreter. These constructors are used in building *compound* goals. In subsection 7.1.2, we discuss the form of *atomic* goals in tactic theorem provers. Then in subsection 7.1.3 we define a primitive operation of all of the interpreters: interpreting compound goals.

7.1.1 Goal Structures

We first introduce a new base type `goal` for the new goal structures. We will call goals of type `o` *meta-goals* to distinguish them from the new goals. Meta-goals have a specific meaning given to them by the depth-first interpreter. Goals of type `goal`, on the other hand, will be given meaning by the new programs we write to implement various interpreters. We want to have each of the search operations of the meta-language available

to our interpreters, so we introduce one goal constructor corresponding to each and give them types as in the `goals` module below. In this module, `tt` corresponds to the trivially

```
module goals.

kind    goal    type.

type    tt      goal.
type    ff      goal.
type    '&&'    goal -> goal -> goal.
type    'vv'    goal -> goal -> goal.
type    all     (A -> goal) -> goal.
type    some    (A -> goal) -> goal.
type    '==>>'  A -> goal -> goal.
```

Module `goals`: Goal Constructors for Meta-Goals

satisfied goal, `ff` corresponds to failure, `&&` corresponds to the AND search operation, `vv` to OR, `all` to GENERIC, `some` to INSTANCE, and `==>>` to AUGMENT. Note that all of the goals except `==>>` have types similar to their corresponding meta-goals with the type `o` replaced by `goal` everywhere. The reason why the type for `==>>` is not `goal -> goal -> goal` will become apparent later. Note that the goal quantifiers `all` and `some` have polymorphic type. In general, for each theorem prover, quantification in goals will be limited to a small number of base types. An alternative approach to defining quantified goals is to introduce a quantifier for each such type. For example, if `tm` is one of the base types, we would have the following declarations.

```
type    all_tm      (tm -> goal) -> goal.
type    some_tm     (tm -> goal) -> goal.
```

Using this approach, the definition of an interpreter would depend on the particular application and its base types. We choose to take the more general approach, so that one interpreter can be adopted without modification by different application programs. In practice, the instances of `A` will be restricted to a subset of the base types of a particular application program.

The specifications of proof systems in Chapters 3 and 4, on several occasions, made use of disjunctive, existential, and implicational goals in the bodies of clauses, which operationally correspond to the use of the OR, INSTANCE, and AUGMENT search operations, respectively. While useful for specification, in general these three search connectives are not essential. Disjunctive goals in the bodies of clauses, can be eliminated by introducing a clause for each disjunct. For example, in the `liprover` specification of L_I in Section 3.1, we saw that the clause for the $\vee$-R rule with a disjunctive subgoal could be replaced by two clauses, one for each subgoal. In fact it will usually be desirable to have multiple clauses, so that our interpreters have control in choosing which one to use. Existential quantification

in subgoals can be eliminated by replacing it with universal quantification over the whole clause. Meta-level implication was used in the natural deduction specifications of Chapter 3 to specify the discharge of assumptions. As noted there, lists can be used instead to provide more flexibility in the manipulation of assumptions. In general, the three goal structures `vv`, `some`, and `==>>` will not be used in implementations of theorem provers in this chapter, but we will include them to be complete.

7.1.2 Inference Rules as a Relation on Goals

Note that we can view definite clauses as the specification of a binary relation on meta-goals. In each clause, the first goal is given by the head of the clause which can be viewed operationally as an "input goal." It may unify with any goal that the interpreter is trying to solve. The second goal is the body of the clause which can be viewed as the "output goal." Its instances provide new subgoals which must subsequently be proved. In the specification of inference rules of various proof systems in Chapters 3 and 4, this input/output relation was between a conclusion of a rule and its premises. In the specification of inference rules using the new goal structures, this relation will be made explicit. Inference rules will be specified as clauses of a special form; they will be named facts where the name is a predicate of type `goal -> goal -> o` where the first argument is the input goal specifying the conclusion, and the second is the output goal specifying the premises. Basic goals will again encode the relation between a formula and its proof, a sequent and its proof, a term and its type, etc. as they did in Chapters 3 and 4. For example, to implement a theorem prover for the natural deduction system N_I, we may again use the infix constant `#` to encode the relation between a formula and its proof, but in this case it will have type `nprf -> form -> goal`. Goals of the form `(P # A)` would be the atomic goals of such a prover, in contrast to compound goals which will be built using the goal constructors from the last section. As an example, the inference rule for the $\wedge$-I rule of N_I can be specified by the following clause, where the input goal is an atomic goal and the output goal is a conjunctive compound goal.

```
and_i_tac  ((and_i P1 P2) # (A and B))  ((P1 # A) && (P2 # B)).
```

While this clause differs in syntax from the corresponding clause in Section 3.2, its meaning is similar. The declarative reading is exactly the same: if `P1` is a proof of `A` and `P2` is a proof of `B`, then `(and_i P1 P2)` is a proof of `(A and B)`. In Chapters 3 and 4, the use of search connectives in clauses provided an operational reading with respect to a non-deterministic interpreter. Here, the operational reading is similar but indirect since it depends on the fact that the goal structures `&&`, `vv`, etc. will be implemented in terms of their corresponding search connectives. Clauses of the above form which state facts about the relation between goals will be called *tactics*. They will be the primitive operations of our theorem provers.

7.1.3 Interpreting Compound Goal Structures

Any interpreter we write must have the ability to break down compound goal structures, so that primitive operations can be applied to atomic goals. The `maptac` program on page 133 is a general program that will handle this task in all of the interpreters we specify. It takes a tactic as an argument and applies it to the input goal in a manner consistent with the meaning of the goal structure. The type of `maptac` is:

```
(goal -> goal -> o) -> goal -> goal -> o.
```

That is, the metalevel predicate `maptac` takes as its first argument a metalevel predicate which represents a tactic. Its second and third arguments are an input and output goal. On an `&&` structure, `maptac` will apply the tactic to each subgoal separately, forming a new

```
module maptac.

import goals.

type    maptac  (goal -> goal -> o) -> goal -> goal -> o.
type    memo    A -> o.

maptac Tac tt tt.

maptac Tac (InGoal1 && InGoal2) (OutGoal1 && OutGoal2) :-
  maptac Tac InGoal1 OutGoal1, maptac Tac InGoal2 OutGoal2.

maptac Tac (all InGoal) (all OutGoal) :-
  pi T\ (maptac Tac (InGoal T) (OutGoal T)).

maptac Tac (InGoal1 vv InGoal2) OutGoal :-
  maptac Tac InGoal1 OutGoal; maptac Tac InGoal2 OutGoal.

maptac Tac (some InGoal) OutGoal :-
  sigma T\ (maptac Tac (InGoal T) OutGoal).

maptac Tac (D ==>> InGoal) (D ==>> OutGoal) :-
  (memo D) => (maptac Tac InGoal OutGoal).

maptac Tac InGoal OutGoal :- Tac InGoal OutGoal.
```

Module `maptac`: Interpreting Compound Goal Structures

`&&` structure to combine the results. The clause for `all` is implemented using the universal quantifier `pi` of the meta-language to introduce a new constant for `T` which gets substituted for the bound variable in `InGoal`. `OutGoal` is the result of abstracting over this constant. The next three clauses for `vv`, `some`, and `==>>` are not needed as illustrated earlier, but to be complete, we include them here. The clause for `vv` attempts to apply the tactic to the first disjunct, and if that fails tries the second. The clause for `some` introduces a new logic

variable into the input goal via the existential quantifier `sigma` of the meta-language. In the clause implementing `==>>`, note that an auxiliary predicate `memo` (of polymorphic type `A -> o`) was introduced. This allows the introduction of new clauses into the program. Its polymorphic type allows arbitrary information to be added as clauses. The particular information and its use will, of course, depend on the particular application. The use of `memo` to introduce clauses avoids the possibility of introducing clauses with predicate variables as disallowed in the meta-theory. The last clause above is used once the goal is reduced to an atomic form. It simply applies the tactic directly.

7.2 N_I Inference Rules as Tactics

In this section, we present in full a specification of the N_I inference rules as tactics. This specification will provide an example to be used in later sections for illustrating the behavior of various interpreters. In addition, in this section, we address the issue of how best to specify the rules of N_I as tactics. As we saw in Chapter 3, there are many ways to specify N_I. We choose one that is well-suited for interactive proof search.

The clauses we present here are based on the clauses in the `ninormal` module on page 39. This core set of tactics will only build normal proofs. Later we will add tactics that allow more flexibility in user-guided proof search. The clauses presented here are obtained from the clauses of `ninormal` by two mechanical changes of syntax. First, we modify them to use judgment sequents with explicit context lists as described in Section 3.3. Then, we further modify them so that they are named clauses expressing a relation between two goals, as discussed in Section 7.1.2. In this setting `-->` will be the atomic goal constructor. The declarations needed for the primitive goals of this prover are in the `ndgoal` module below. It imports the `goals` module presented in this chapter, and the `nprf` module on page 33 in Chapter 3. Again, the only difference in these declarations and the ones for the

```
module ndgoal.

import  goals nprf.

kind    judg    type.

type    '#'     nprf -> form -> judg.
type    '-->'   (list judg) -> judg -> goal.
```

Module `ndgoal`: Primitive Goal Constructors for N_I

direct explicit context specifications of N_I is the occurrence of `goal` in place of `o`.

A complete set of tactics for N_I appears in the `ndtac` module on pages 137 and 138. The change of syntax needed to obtain the tactics for the introductions rules and $\perp_I$ from

the clauses of ninormal is very straightforward. For example, the tactic for the ∧-I rule is as follows.

```
and_i_tac  (Gamma --> (and_i P1 P2) # (A and B))
           ((Gamma --> P1 # A) && (Gamma --> P2 # B)).
```

In Section 3.5, we showed that operationally, the clauses for the E-rules of the ninormal module apply inference rules in a forward direction from the current set of assumptions. Using explicit contexts, this behavior corresponds to looking for a formula of a particular form in the assumption list, applying an E-rule to it, and adding the new partial proof to the assumption lists of the subgoals. In the tactic specification, we provide the additional capability to choose a specific formula within the list to which the rule will be applied. We add an integer argument to each tactic which specifies the position in the list of a particular assumption. For example, the clause for and_e is specified as follows.

```
and_e_tac N (Gamma --> PC # C)
            ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> PC # C) :-
   nth_item N (P # (A and B)) Gamma.
```

The nth_item predicate is given in Appendix A. In the above clause, the call to this predicate will succeed if there is a conjunction in position N in Gamma. When N is 0, nth_item calls the memb program, which in this case will find the first occurrence of a conjunction in Gamma. Notice that and_e_tac itself is not a proper tactic, although (and_e_tac N) is for any integer N.

It is also possible to specify programs that manipulate lists, such as nth_item, as tactics, and include such subgoals in the output goal structure so that they must also be solved by the tactic interpreter instead of the logic programming interpreter. Then the above tactic could be modified as follows.

```
and_e_tac N (Gamma --> PC # C)
            ((nth_item N (P # (A and B)) Gamma) &&
             ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> PC # C)).
```

Since all of the list manipulations we use behave well under depth-first search, we continue to specify such subgoals as meta-goals. The remaining tactics for the elimination rules are similar. (See pages 137 and 138.)

As stated in Section 3.3, it is also possible and often useful to specify rules that apply to assumptions so that they remove the assumption from the hypothesis list after applying the rule. Here, we can include these new tactics by adding them to the existing set of tactics and giving them different names. For example, the tactic that removes a universally quantified hypothesis can be defined as follows.

```
forall_e_tacr N (Gamma --> PC # C)
                ((((forall_e T A P) # (A T))::Gamma1) --> PC # C) :-
   nth_and_rest N (P # (forall A)) Gamma Gamma1.
```

The nth_and_rest predicate also is given in Appendix A. In the above clause, Gamma1 is the list Gamma with the pair (P # (forall A)) removed. This tactic and similar tactics for the other elimination rules are also in the ndtac module on page 138.

Finally, the following tactic is needed to complete proofs.

```
close_tac N (Gamma --> P # A) tt :- nth_item N (P # A) Gamma.
```

Again, an integer argument allows the user to choose a particular assumption from Gamma during interactive proof. Notice that the only compound goal constructors that appear in the tactics of ndtac are && and all.

There are several reasons for choosing ninormal with explicit contexts as the starting point for specifying the basic tactics of a natural deduction theorem prover. First of all, since any information contained inside atomic goals will be readily accessible to the interpreter, including explicit assumption lists inside goals allows operations on them to be easily accomplished. For example, we were able to specify versions of the inference rule tactics that remove the principle formula from the context at the time the rule is applied. Other useful operations such as printing out all of the assumptions during an interactive session are also straightforward to implement. In contrast, consider the following tactic for imp_i which uses implicational goals to specify the discharge of assumptions.

```
imp_i_tac  ((imp_i P) # (A imp B))
           (all PA\ ((PA # A) ==>> (P PA) # B)).
```

Using this clause, object level hypotheses get added as memo clauses via meta-level implication. With this formulation, certain tasks become much more difficult, and others, such as removing hypotheses become impossible.

Specifying the elimination rules so that they look in the assumption list for a formula of a particular form also has several advantages. First of all, as discussed in Section 6.1, such a specification prevents new logic variables of type form from appearing in goals. In contrast, consider the following tactic for and_e1.

```
and_e1_tac (Gamma --> (and_e1 B P) # A) (Gamma --> P # (A and B)).
```

This tactic introduces a logic variable for formula B in the output goal whenever the proof in the input goal is unspecified at the time the tactic is applied. By always applying elimination rules to assumptions, on the other hand, as long as Gamma in the original query contains no variables of type form, the output goal will contain no variables of type form. All logic variables inside formulas, if any, will have type i. Thus, in interactive theorem proving, the user need not keep track of formula variables and their potential unifiers. In addition the capability to reason forward from the hypotheses is useful in interactive theorem proving.

```
module ndtac.

import  ndgoal lists.

type    close_tac       int -> goal -> goal -> o.
type    and_i_tac       goal -> goal -> o.
type    or_i1_tac       goal -> goal -> o.
type    or_i2_tac       goal -> goal -> o.
type    imp_i_tac       goal -> goal -> o.
type    neg_i_tac       goal -> goal -> o.
type    forall_i_tac    goal -> goal -> o.
type    exists_i_tac    goal -> goal -> o.
type    false_i_tac     goal -> goal -> o.
type    and_e_tac       int -> goal -> goal -> o.
type    imp_e_tac       int -> goal -> goal -> o.
type    neg_e_tac       int -> goal -> goal -> o.
type    forall_e_tac    int -> goal -> goal -> o.
type    or_e_tac        int -> goal -> goal -> o.
type    exists_e_tac    int -> goal -> goal -> o.
type    and_e_tacr      int -> goal -> goal -> o.
type    imp_e_tacr      int -> goal -> goal -> o.
type    neg_e_tacr      int -> goal -> goal -> o.
type    forall_e_tacr   int -> goal -> goal -> o.
type    or_e_tacr       int -> goal -> goal -> o.
type    exists_e_tacr   int -> goal -> goal -> o.

close_tac N (Gamma --> P # A) tt :- nth_item N (P # A) Gamma.

and_i_tac  (Gamma --> (and_i P1 P2) # (A and B))
           ((Gamma --> P1 # A) && (Gamma --> P2 # B)).

or_i1_tac  (Gamma --> (or_i1 P) # (A or B))  (Gamma --> P # A).

or_i2_tac  (Gamma --> (or_i2 P) # (A or B))  (Gamma --> P # B).

imp_i_tac  (Gamma --> (imp_i P) # (A imp B))
           (all PA\ (((PA # A)::Gamma) --> (P PA) # B)).

neg_i_tac  (Gamma --> (neg_i P) # (neg A))
           (all PA\ (((PA # A)::Gamma) --> (P PA) # false)).

forall_i_tac  (Gamma --> (forall_i P) # (forall A))
              (all Y\ (Gamma --> (P Y) # (A Y))).

exists_i_tac  (Gamma --> (exists_i T P) # (exists A)) (Gamma --> P # (A T)).

false_i_tac  (Gamma --> (false_i P) # A)  (Gamma --> P # false).

and_e_tac N (Gamma --> PC # C)
            ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> PC # C) :-
   nth_item N (P # (A and B)) Gamma.
```

Module `ndtac` Part I: Tactics for N_I

```
imp_e_tac N (Gamma --> PC # C)
            ((Gamma --> P1 # A) &&
             ((((imp_e A P1 P2) # B)::Gamma) --> PC # C)) :-
   nth_item N (P2 # (A imp B)) Gamma.

neg_e_tac N (Gamma --> PC # C)
            ((Gamma --> P1 # A) &&
             ((((neg_e A P1 P2) # false)::Gamma) --> PC # C)) :-
   nth_item N (P2 # (neg A)) Gamma.

forall_e_tac N (Gamma --> PC # C)
               ((((forall_e T A P) # (A T))::Gamma) --> PC # C) :-
   nth_item N (P # (forall A)) Gamma.

or_e_tac N (Gamma --> (or_e A B P P1 P2) # C)
           ((all PA\ (((PA # A)::Gamma) --> (P1 PA) # C)) &&
            (all PB\ (((PB # B)::Gamma) --> (P2 PB) # C))) :-
   nth_item N (P # (A or B)) Gamma.

exists_e_tac N (Gamma --> (exists_e A P1 P2) # B)
               (all Y\ (all P\ (((P # (A Y))::Gamma) --> (P2 Y P) # B))) :-
   nth_item N (P1 # (exists A)) Gamma.

and_e_tacr N (Gamma --> PC # C)
             ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma1) --> PC # C) :-
   nth_and_rest N (P # (A and B)) Gamma Gamma1.

imp_e_tacr N (Gamma --> PC # C)
             ((Gamma --> P1 # A) &&
              ((((imp_e A P1 P2) # B)::Gamma1) --> PC # C)) :-
   nth_and_rest N (P2 # (A imp B)) Gamma Gamma1.

neg_e_tacr N (Gamma --> PC # C)
             ((Gamma --> P1 # A) &&
              ((((neg_e A P1 P2) # false)::Gamma1) --> PC # C)) :-
   nth_and_rest N (P2 # (neg A)) Gamma Gamma1.

forall_e_tacr N (Gamma --> PC # C)
               ((((forall_e T A P) # (A T))::Gamma1) --> PC # C) :-
   nth_and_rest N (P # (forall A)) Gamma Gamma1.

or_e_tacr N (Gamma --> (or_e A B P P1 P2) # C)
            ((all PA\ (((PA # A)::Gamma1) --> (P1 PA) # C)) &&
             (all PB\ (((PB # B)::Gamma1) --> (P2 PB) # C))) :-
   nth_and_rest N (P # (A or B)) Gamma Gamma1.

exists_e_tacr N (Gamma --> (exists_e A P1 P2) # B)
               (all Y\ (all P\ (((P # (A Y))::Gamma1) --> (P2 Y P) # B))) :-
   nth_and_rest N (P1 # (exists A)) Gamma Gamma1.
```

Module `ndtac` Part II: Tactics for N_I

7.3 Some Simple Interpreters

We begin the implementation of interpreters in the next subsection by illustrating a simple depth-first interpreter. Such an interpreter will behave the same as the depth-first logic programming interpreter, although it will be less efficient because of the extra level of interpretation. Its purpose here will be to serve as a starting point for the implementation of interpreters for other more complex strategies.

Breadth-first search, another common search strategy, operates by building a search tree one level at a time. Its main advantage is that it is a complete search strategy. If there is some path that terminates with success, this success node will eventually be reached. An interpreter implementing breadth-first control, together with a set of tactics for any proof system or type system would implement a complete theorem prover, proof checker, and type inferencer. The main problem with this strategy is that it is very space-inefficient since it must keep track of all of the search paths. Although it could be implemented using the goal structures and `maptac` program illustrated so far, it is not practical in building efficient implementations, and we do not implement it here.

Depth-first iterative deepening [Kor85, ST85], like breadth-first is a complete search strategy, but like depth-first does not require huge amounts of memory. Iterative deepening operates by first performing a depth-first search to depth one, then discarding the results and performing a depth-first search to depth two, and so on. Discarding the results at each iteration does not greatly affect efficiency because most of the work is done at the deepest level. It was argued in [Sti86] that this strategy is very suitable for automated deduction in first-order logic. We will see that this strategy can be implemented by slightly modifying the depth-first interpreter.

In the sections following this one, we implement a tactic interpreter which can be used to create an environment for theorem proving that can integrate any or all of the search strategies discussed in this section with interactive theorem proving and other kinds of search strategies and proof search heuristics.

7.3.1 A Depth-First Interpreter

The program for the depth-first interpreter is contained in the `dfs` module below. It uses the three predicates `app_tac`, `dfs`, and `dfs_interp`. The `app_tac` predicate takes a list of tactics to use in the search, an input goal, and an output goal. Its role is to perform one step of the search at a time. It traverses the list of tactics and attempts them one by one until one succeeds (or they all fail). The `dfs` predicate is the main predicate for depth-first search. The input goal to `dfs` is assumed to be atomic. First, the `app_tac` procedure is called to apply a tactic, resulting in an intermediate goal structure. Then `maptac` is called to break down any possible compound structure of the intermediate goal, and call

```
module dfs.

import lists maptac.

type app_tac            list (goal -> goal -> o) -> goal -> goal -> o.
type dfs                list (goal -> goal -> o) -> goal -> goal -> o.
type dfs_interp         list (goal -> goal -> o) -> goal -> o.


app_tac (Tac::Rest) InGoal OutGoal :- Tac InGoal OutGoal.
app_tac (Tac::Rest) InGoal OutGoal :- app_tac Rest InGoal OutGoal.

dfs Tacs InGoal OutGoal :- app_tac Tacs InGoal MidGoal,
                           maptac (dfs Tacs) MidGoal OutGoal.

dfs_interp Tacs InGoal :- maptac (dfs Tacs) InGoal OutGoal.
```

Module `dfs`: A Depth-First Interpreter

`dfs` again on each of the atomic subgoals. The recursion between `dfs` and `maptac` forms the main loop for this search strategy. When a subgoal is reduced to `tt`, the first clause of the `maptac` program succeeds without calling `dfs` again, terminating a branch of the search. If all subgoals are reduced to `tt`, the top-level call to `dfs` terminates with success. `OutGoal` will then be a compound goal structure containing only `tt` as its atomic subgoals. Otherwise depth-first search fails. `dfs_interp` is the top-level predicate. It simply calls `maptac` with the tactic `(dfs Tacs)` to decompose the initial compound goal structure before beginning the depth-first loop. This predicate does not include an argument for the output goal since the program will either completely solve the input goal or loop indefinitely. Thus, the output goal will contain no useful information.

For theorem proving in natural deduction, we can attempt depth-first search over the list of tactics implementing the inference rules given in Section 7.2. For some formula `A` and some proof term `P`, a query of the form:

```
dfs_interp ((close_tac 0)::and_i_tac::or_i1_tac::or_i2_tac::imp_i_tac::
            neg_i_tac::forall_i_tac::exists_i_tac::false_i_tac::
            (and_e_tac 0)::(imp_e_tac 0)::(forall_e_tac 0)::(neg_e_tac 0)::
            (or_e_tac 0)::(exists_e_tac 0)::nil)
           (nil --> P # A).
```

will behave the same as a query of the form `(P # A)` to `ninormal`, or a query of the form `(nil --> P # A)` to the explicit context version of `ninormal`.

7.3.2 A Depth-First Iterative Deepening Interpreter

To implement iterative deepening, the main modification to ordinary depth-first search is that at each step, we must compare the current depth of search to the current bound

on search. Search must fail on a particular subgoal if the bound has been reached and the subgoal is not completed. The `idfs` module below implements this strategy. Like the

```
module idfs.

import lists maptac.

type app_tac            list (goal -> goal -> o) -> goal -> goal -> o.
type idfs               int -> int -> list (goal -> goal -> o)
                          -> goal -> goal -> o.
type idfs_interpb       int -> int -> int -> list (goal -> goal -> o)
                          -> goal -> o.
type idfs_interp        int -> int -> list (goal -> goal -> o) -> goal -> o.

app_tac (Tac::T) InGoal OutGoal :- Tac InGoal OutGoal.
app_tac (Tac::T) InGoal OutGoal :- app_tac T InGoal OutGoal.

idfs Bnd N Tacs InGoal OutGoal :-
  N =< Bnd,
  app_tac Tacs InGoal MidGoal,
  M is (N + 1),
  maptac (idfs Bnd M Tacs) MidGoal OutGoal.

idfs_interpb Bnd Incr UpBnd Tacs InGoal :-
  maptac (idfs Bnd 1 Tacs) InGoal OutGoal.
idfs_interpb Bnd Incr UpBnd Tacs InGoal :-
  Bnd1 is (Bnd + Incr), Bnd1 =< UpBnd,
  idfs_interpb Bnd1 Incr UpBnd Tacs InGoal.

idfs_interp Bnd Incr Tacs InGoal :- maptac (idfs Bnd 1 Tacs) InGoal OutGoal.
idfs_interp Bnd Incr Tacs InGoal :- Bnd1 is (Bnd + Incr),
                                    idfs_interp Bnd1 Incr Tacs InGoal.
```

Module `idfs`: An Iterative Deepening Interpreter

depth-first interpreter of the previous section, this interpreter uses `app_tac` to perform each single step of the search. The `idfs` predicate is a modified form of the `dfs` predicate of the depth-first interpreter. Here it must also perform a check on the search bound. Thus it has two additional integer arguments for the current depth of search and the bound. The first subgoal will fail if the bound has been exceeded. Otherwise `app_tac` is called to perform one step in the search. The current bound is then incremented before calling `maptac` and re-entering the search loop.

The `idfs` interpreter contains two top-level predicates, `idfs_interpb` for bounded search, and `idfs_interp` for unbounded search. They are both modified forms of `dfs_interp`. The `idfs_interpb` predicate has three integer arguments for the current bound, an increment for increasing the bound, and an upper bound on the depth of search. The first clause begins search using the bound initially specified by `Bnd`, and setting the

current depth to 1. If it fails, the second clause increments the bound by the amount specified, and checks the new bound against the upper bound. If the bound has not been exceeded search begins again with the new bound. Otherwise, if the upper bound has been exceeded, search fails. The search bound is useful for controlling the amount of effort spent on a particular goal. Note however, that because of this bound, this search strategy is not complete.

The `idfs_interp` predicate is the same as `dfs_interpb` except that the check of the bound against an upper bound is omitted. Thus if the input goal is not provable, search will continue indefinitely. Without an upper bound on the depth of search, iterative deepening, like breadth-first search, is a complete strategy. Thus `idfs_interp` will act as a complete theorem prover, proof checker, and type inferencer for any proof system or type system that can be specified as a set of tactics. In fact, such execution can be more efficient than breadth-first search since iterative deepening avoids the memory overhead (see [Kor85]).

Note that neither top-level predicate distinguishes failure caused by exhausting the search space from failure caused by some branch exceeding the depth bound. Such a test could be incorporated into either implementation. For the `idfs_interp` predicate, search may then terminate rather than loop indefinitely on some unprovable subgoals.

Consider the execution of this interpreter on a set of tactics specifying the inference rules of the L_C proof system introduced in Section 4.1. There is a correspondence between its behavior under an iterative deepening interpreter, and the behavior of `lc_auto`, the automatic theorem prover for L_C given in Section 6.2 that operates under depth-first search. In the execution of the former, the bound on the depth of search induces a limit on the number of times an inference rule can be applied to the same formula. In the latter, the number of rule applications is limited by the number of copies of the formula. Under iterative deepening, the interpreter handles the constraints on applying inference rules at each iteration. The tactics need not be modified. In contrast, in `lc_auto`, the constraints are explicitly a part of the program. Note that `lc_auto` behaves like the unbounded interpreter `idfs_interp` since it will never terminate for sequents that are not provable. On the other hand, a bound on the number of allowable copies of quantified formulas could easily be incorporated into `lc_auto`.

7.4 A Tactic Interpreter

Tactic style theorem provers were first built in the early LCF systems and, as mentioned earlier, have been adopted as a central mechanism in such notable theorem proving systems as Edinburgh LCF [GMW79], HOL [Gor85], Nuprl [C$^+$86], and Isabelle [Pau88]. In these systems, as is the case here, primitive tactics generally implement inference rules while compound tactics are built from these using a compact but powerful set of *tacticals*.

Tacticals provide the basic control over search. Tactics and tacticals have proved valuable for several reasons. They promote modular design and provide flexibility in controlling the search for proofs. They also allow for blending automatic and interactive theorem proving techniques in one environment. This environment can also be grown incrementally.

We have already discussed the implementation of tactics. In addition, we have discussed the implementation of the `maptac` program which will be just one of the basic control primitives or tacticals of this interpreter. Here, we also implement several other tacticals. They are some of the familiar ones found in many tactic style theorem provers.

Generally tactics and tacticals have been implemented in the functional programming language ML. The λProlog implementation is very natural and extends the usual meaning of tacticals by permitting them to have access to logic variables and all six search operations. A comparison between the ML and λProlog implementations is contained in Section 7.9.

Six common tacticals are implemented in the `tacticals` module on page 143. The

```
module tacticals.

import  maptac goalred.

type    then            (goal -> goal -> o)
                          -> (goal -> goal -> o) -> goal -> goal -> o.
type    orelse          (goal -> goal -> o)
                          -> (goal -> goal -> o) -> goal -> goal -> o.
type    idtac           goal -> goal -> o.
type    repeat          (goal -> goal -> o) -> goal -> goal -> o.
type    try             (goal -> goal -> o) -> goal -> goal -> o.
type    complete        (goal -> goal -> o) -> goal -> goal -> o.

then Tac1 Tac2 InGoal OutGoal :- Tac1 InGoal MidGoal,
                                 maptac Tac2 MidGoal OutGoal.

orelse Tac1 Tac2 InGoal OutGoal :- Tac1 InGoal OutGoal; Tac2 InGoal OutGoal.

idtac Goal Goal.

repeat Tac InGoal OutGoal :-
  orelse (then Tac (repeat Tac)) idtac InGoal OutGoal.

try Tac InGoal OutGoal :- orelse Tac idtac InGoal OutGoal.

complete Tac InGoal tt :- Tac InGoal OutGoal, goalreduce OutGoal tt.
```

Module `tacticals`: Some Common Tacticals

`then` tactical performs the composition of tactics. `Tac1` is applied to the input goal, and then `Tac2` is applied to the resulting goal. In this tactical and all others, we assume

that the input goal is atomic. `maptac` is used in the second case since the application of `Tac1` may result in an output goal (`MidGoal`) with compound structure. This tactical plays a fundamental role in combining the results of step-by-step proof construction. The substitutions resulting from applying these separate tactics get combined correctly since `MidGoal` provides the necessary sharing of logic variables between these two calls to tactics. The `orelse` tactical simply uses the OR search operation so that `Tac1` is attempted, and if it fails (in the sense that the logic programming interpreter cannot satisfy the corresponding meta-level goal), then `Tac2` is tried. The third tactical, `idtac`, returns the input goal unchanged. This tactical is useful in constructing compound tactic expressions such as the one found in the `repeat` tactical. `repeat` is recursively defined using the three tacticals, `then`, `orelse`, and `idtac`. It repeatedly applies a tactic until it is no longer applicable. This tactical could also be defined directly by the following clauses.

```
repeat Tac InGoal OutGoal :- Tac InGoal MidGoal,
                             maptac (repeat Tac) MidGoal OutGoal.
repeat Tac Goal Goal.
```

The `try` tactical prevents failure of the given tactic by using `idtac` when `Tac` fails. It might be used, for example, in the second argument of an application of the `then` tactical. It prevents failure when the first argument tactic succeeds and the second does not. Finally the `complete` tactical tries to completely solve the given goal. It will fail if there is a non-trivial goal remaining after `Tac` is applied. It requires an auxiliary procedure `goalreduce` of type `goal -> goal -> o` which simplifies compound goal expressions by removing occurrences of `tt` from them. The `complete` tactical succeeds only if the output goal is simplified to `tt`. The code for `goalreduce` and its auxiliary procedure `remove_tt` is in the `goalred` module on page 145. Instances of cut (`!`) are used here for efficiency only. The program would behave the same without them.

Note that we could build eager reduction of goals directly into the `maptac` program so that it simplifies goals and removes occurrences of `tt` as it progresses. This requires redefining the three `maptac` clauses for the `&&`, `all`, and `==>>` goal structures as follows.

```
maptac Tac (InGoal1 && InGoal2) OutGoal :-
  maptac Tac InGoal1 OutGoal1, maptac Tac InGoal2 OutGoal2.
  remove_tt (OutGoal1 && OutGoal2) OutGoal.

maptac Tac (all InGoal) OutGoal :-
  pi T\ (maptac Tac (InGoal T) (OutGoal1 T)), remove_tt (all OutGoal1) OutGoal.

maptac Tac (D ==>> InGoal) OutGoal :-
  (memo D) => (maptac Tac InGoal OutGoal1),
   remove_tt (D ==>> OutGoal1) OutGoal.
```

Using this implementation of `maptac`, occurrences of `tt` will never appear nested inside compound structures since they are always removed at the earliest point possible. Thus

```
module goalred.

import  goals.

type    goalreduce      goal -> goal -> o.
type    remove_tt       goal -> goal -> o.

goalreduce (Goal1 && Goal2) RGoal :- !,
  goalreduce Goal1 RGoal1, goalreduce Goal2 RGoal2,
  remove_tt (RGoal1 && RGoal2) RGoal.
goalreduce (all Goal) RGoal :- !,
  pi T\ (goalreduce (Goal T) (RGoal1 T)), remove_tt (all RGoal1) RGoal.
goalreduce (D ==>> Goal) RGoal :- !,
  goalreduce Goal RGoal1, remove_tt (D ==>> RGoal1) RGoal.
goalreduce Goal Goal.

remove_tt (Goal && tt) Goal :- !.
remove_tt (tt && Goal) Goal :- !.
remove_tt (all T\tt) tt :- !.
remove_tt (D ==>> tt) tt :- !.
remove_tt Goal Goal.
```

Module `goalred`: Simplifying Compound Goals

the simpler `remove_tt` procedure is all that is needed. The `complete` tactical could then by defined as follows.

```
complete Tac InGoal tt :- Tac InGoal tt.
```

The definite clauses in the `maptac` (page 133) and `tacticals` (page 143) modules with the auxiliary modules that they import provide a complete implementation of tacticals, the primitive control operations of the interpreter. Together with the primitive tactics which specify a basic set of operations for a particular application, they provide a simple programming language for writing search strategies. In the next sections, we illustrate the use of this language, and how it can be modularly extended to include other capabilities. We begin, in the next section, by adding an interactive component to the basic interpreter.

7.5 Interactive Component for the Tactic Interpreter

In this section we define the clauses that make up an interactive component for the tactic interpreter. The complete set of clauses is in the `inter_tacs` module on page 150 at the end of this section.

We first present an alternative implementation of the `orelse` tactical that will be useful here. Note that in the version from the `maptac` module:

```
orelse Tac1 Tac2 InGoal OutGoal :- Tac1 InGoal OutGoal; Tac2 InGoal OutGoal.
```

if Tac1 succeeds, a backtracking point will be set up so that if there is a subsequent failure, control may return to this clause, and Tac2 will be attempted. We may define another version of this tactical which eliminates this backtracking point as follows.

```
orelse! Tac1 Tac2 InGoal OutGoal :- Tac1 InGoal OutGoal, !; Tac2 InGoal OutGoal.
```

The clause without the cut is more general in the sense that there may be more ways for it to succeed. This is the first clause that relies on a non-logical feature (!) of the meta-language to obtain the desired operational behavior. There will be a few other occasions where the use of cut will be crucial in defining operational behavior in clauses in this section since more fine-tuned control is important for good user interaction.

The orelse! tactical will be used by the query tactic, which will be the primitive operation of the interactive component. It will query the user for the next action to take. Its task can be divided into three steps. The first step is to output some information about the state of the interpreter. The second step is to get input from the user about what action to take, and the third step is to perform the action specified by the input. In the first step, the form of the output will depend on the particular theorem prover or other program being interpreted. For example, in a natural deduction theorem prover we will want to display the formula to be proven, and possibly the current list of assumptions. For each application we must write a specialized procedure to perform the output. Such a procedure will extract information from the input goal to the query tactic. In implementing the general query tactic, we make this function a parameter of type (goal -> o). For the second step, the task of asking for input may be as simple as asking the user for the next tactic to attempt. The following simple clause will perform this operation.

```
readtac Tac :- writesans "Enter tactic: ", read X\ (Tac = X).
```

Such a clause will be included in the interactive component, but for generality, in implementing query, we will parameterize the input function also so that the user may specify a different one if desired. In general, the input function will have type (goal -> goal -> o) -> o, *i.e.*, it takes one argument which must be a tactic.

For slightly more generality, instead of two separate parameters for output functions, there will be one parameter to the query tactic for both input and output, so that both of these tasks may be specified in one procedure if desired. To handle the case when they are separate, we define the clause below which takes separate output and input predicates as arguments.

```
basic_io PrintPred ReadPred Goal Tac :- PrintPred Goal, ReadPred Tac.
```

The query tactic is defined by the following clause.

```
query IO InGoal OutGoal :-
   IO InGoal Tac,
   ((Tac = backup), !, fail;
    orelse! Tac report_fail InGoal OutGoal;
    query IO InGoal OutGoal).
```

Its first argument has type `(goal -> (goal -> goal -> o) -> o)`. Thus `IO` is a predicate with two arguments: a goal and a tactic. The subgoal `(IO InGoal Tac)` first performs the output using the information in the input goal `InGoal` and then obtains input from the user causing `Tac` to be bound to the tactic to be applied to the input goal. For example, if `ndprint` is the name of a procedure to print out the state of a natural deduction theorem prover, the `IO` argument may be, for example, `(basic_io ndprint readtac)`. There are then three options in applying the tactic `Tac` input by the user, given by the three disjuncts above. In the first disjunct, notice the use of cut (`!`) and `fail`. One requirement of a good interactive system is to provide the user with some capability to back up the search to previous points. In this implementation, the user will be allowed to incrementally backup the search one step at a time, by invoking the `backup` tactic. Here `backup` is implemented by causing the logic programming interpreter to fail to a previous point. The use of cut here insures that the other two disjuncts will not be attempted. Cuts must be strategically placed in all of the clauses that make up the interactive component so that invoking the `backup` tactic takes the interpreter to the desired backtracking point, the one corresponding to the last invocation of the `query` tactic. The second disjunct attempts to apply the requested tactic. It either applies the tactic successfully, or reports failure when the tactic fails. The `report_fail` tactic has the following simple definition.

```
report_fail Goal Goal :- writesans "Tactic failed.", nl.
```

The `orelse!` tactical is used inside `query` since if the tactic succeeds, we do not want the interpreter to be able to later report failure. The third disjunct in the `query` tactic handles the case when the `backup` tactic is used in subsequent calls to `query`. To return the search to the state it was in upon entering this particular invocation of `query`, it simply calls itself recursively.

With `query` as the primitive operation for interaction, we can define an interactive loop by repeatedly applying this tactic. The `inter` tactic below implements this loop.

```
inter IO InGoal OutGoal :- repeat (query IO) InGoal OutGoal.
```

Since the `repeat` tactical loops until the tactic fails, and since the `query` tactic only fails when the `backup` tactic is invoked, the `inter` tactic will terminate in one of two ways. It will fail if backed up all the way to the beginning, or will succeed when the input goal is completely solved by the user. As in the `dfs` and `idfs` interpreters, the first clause in the `maptac` module terminates each branch of the search as the input goals are reduced to `tt`. A good interactive interpreter must also provide the user with the capability to stop the search without losing the work done so far, or to stop particular branches of search in favor of pursuing others. For this task we define a `quit` tactic, and a new looping tactical, called `inter_repeat` which terminates search when the `quit` tactic is invoked and returns the current input goal as the output goal. This tactic and tactical are defined by the following clauses.

```
quit InGoal ff.

inter_repeat Tac InGoal OutGoal :-
   Tac InGoal MidGoal,
   ((MidGoal = ff), !, (OutGoal = InGoal);
   maptac (inter_repeat Tac) MidGoal OutGoal).
```

The quit tactic stops search by causing the tactic interpreter to fail (returning `ff` as the output goal). The logic programming interpreter, on the other hand, does not fail. The `inter_repeat` clause checks for the `ff` token, and terminates successfully assigning the value of `InGoal` to `OutGoal`. Note that `quit` is defined in terms of failure of the tactic interpreter, while `backup` is defined in terms of failure of the logic programming interpreter. Using `inter_repeat`, we can redefine the `inter` tactic as follows.

```
inter IO InGoal OutGoal :- inter_repeat (query IO) InGoal OutGoal.
```

This tactic provides a top-level interactive loop that allows both backing up and stopping search branches.

In implementing a particular theorem prover, it may be useful to include tactics which ask the user for input at various stages of proof search. For example, consider the tactic for the $\exists$-I rule in N_I.

```
exists_i_tac  (Gamma --> (exists_i T P) # (exists A)) (Gamma --> P # (A T)).
```

In proving `(exists A)`, the logic variable `T` is introduced, and in completing the proof, may later get instantiated. It may also be desirable to provide a version of this tactic that allows the user to input a substitution term at the time the rule is invoked. Such a tactic could be specified as follows.

```
exists_i_query  (Gamma --> (exists_i T P) # (exists A)) (Gamma --> P # (A T)) :-
   writesans "Enter substitution term: ", read T.
```

Another tactic which may be useful is the following `modus_ponens` tactic which allows the user to add a hypothesis as a lemma, prove it, and then use it in proving the original theorem.

```
modus_ponens  (Gamma --> P # A)
              ((Gamma --> Q # B) && (((Q # B)::Gamma) --> P # A)) :-
   writesans "Enter lemma: ", read B.
```

As stated in Section 7.2, `ndtac` is the set of tactics corresponding to the explicit context specification of the clauses in `ninormal` which only build normal proofs. One invariant in using these tactics alone is that only E-part proofs ever appear in any hypothesis list `Gamma`. With the addition of the `modus_ponens` tactic, since `Q` is not restricted to being an E-part proof, this invariant may no longer hold. As a result, proofs will not necessarily be normal. In fact, the addition of this tactic allows arbitrary N_I proofs to be built.

Clearly the set of tactics for a natural deduction theorem prover can be extended by simply adding tactics such as those above to `ndtac`. It general, though, it may be desirable

to organize tactics into modules containing sets of related tactics, and give the user the flexibility to access only those that are needed at different points during proof construction. The following `with_tacs` tactical allows the user to dynamically add a set of tactics, and temporarily extend the current theorem proving environment.

```
with_tacs M Tac InGoal OutGoal :- M ==> (Tac InGoal OutGoal).
```

The type of the first argument `M` is `modul`, the meta-level base type for λProlog module names. The `==>` symbol is the meta-level connective that instructs the interpreter to load the module `M` into memory and add all of the clauses in `M` to the current program. The tactic `Tac` is applied in the new environment. After successful completion or failure of this tactic, the clauses of `M` will no longer be available unless explicitly added again.

In more sophisticated domains, modules could be used to organize libraries and theories. Such modules could contain any number of items specific to a particular theory such as definitions, theorems and their proofs, tactics implementing proof search heuristics, etc. Such a modular organization of the search space provides a way to integrate many different elements into one theorem proving environment, while at the same time via tacticals such as `with_tacs`, provides a mechanism to constrain search to a manageable subspace at any one time.

```
module inter_tacs.

import tacticals maptac.

type    orelse!         (goal -> goal -> o)
                          -> (goal -> goal -> o) -> goal -> goal -> o.
type    with_tacs       modul -> (goal -> goal -> o) -> goal -> goal -> o.
type    inter_repeat    (goal -> goal -> o) -> goal -> goal -> o.
type    query           (goal -> (goal -> goal -> o) -> o) ->
                         goal -> goal -> o.
type    backup          goal -> goal -> o.
type    inter           (goal -> (goal -> goal -> o) -> o) ->
                         goal -> goal -> o.
type    report_fail     goal -> goal -> o.
type    quit            goal -> goal -> o.
type    basic_io        (goal -> o) -> ((goal -> goal -> o) -> o) ->
                         goal -> (goal -> goal -> o) -> o.
type    readtac         (goal -> goal -> o) -> o.

orelse! Tac1 Tac2 InGoal OutGoal :- Tac1 InGoal OutGoal,!; Tac2 InGoal OutGoal.

with_tacs M Tac InGoal OutGoal :- M ==> (Tac InGoal OutGoal).

inter_repeat Tac InGoal OutGoal :-
   Tac InGoal MidGoal,
   ((MidGoal = ff), !, (OutGoal = InGoal);
   maptac (inter_repeat Tac) MidGoal OutGoal).

query IO InGoal OutGoal :-
   IO InGoal Tac,
   ((Tac = backup), !, fail;
    orelse! Tac report_fail InGoal OutGoal;
    query IO InGoal OutGoal).

inter IO InGoal OutGoal :- inter_repeat (query IO) InGoal OutGoal.

report_fail Goal Goal :- writesans "Tactic failed.", nl.

quit InGoal ff.

basic_io PrintPred ReadPred Goal Tac :- PrintPred Goal, ReadPred Tac.

readtac Tac :- writesans "Enter tactic: ", read X\ (Tac = X).
```

Module `inter_tacs`: Interactive Component for Tactic Interpreter

7.6 A Tactic Theorem Prover for Natural Deduction

We have presented a complete tactic interpreter with an interactive component in addition to a set of tactics for N_I. The only remaining component in the implementation of a tactic theorem prover for natural deduction is a program for printing information about the current state of the search during interactive theorem proving. The `ndprint` module

```
module ndprint.

import ndgoal lists.

type    ndoutput            goal -> o.
type    print_form_list     list judg -> int -> o.

ndoutput (Gamma --> P # A) :-
  nl, writesans "Assumptions: ",
  nl, print_form_list Gamma 1,
  nl, writesans "Conclusion: ",
  nl, write A, nl.

print_form_list nil N.
print_form_list ((P # A)::Tail) N :-
  write N, writesans " ", write A, nl,
  M is (N + 1),
  print_form_list Tail M.
```

Module `ndprint`: Output Program for Interactive Proof Search for N_I

shown here contains a simple program which can be used for this task. The `ndprint` predicate prints out all of the assumptions using the `print_form_list` program, and then prints the conclusion. This program is meant to serve as a simple example. Certainly, other programs could be written, for example, to allow the user more flexibility in choosing which assumptions to print out at each step of the proof.

Finally, we include some top-level clauses in the `nd` module below which can be used to begin an interactive session with the theorem prover. The first argument to the top-level predicate `inter_top` is an input formula, and its second argument is the proof which generally will start out as a variable and will get bound to the proof term as far as it gets constructed during an interactive proof session. At the end of a session, it may contain variables which will also appear in the third argument, the goal structure containing the goals that must still be completed in order to finish the proof. The clause for `inter_top` simply calls the `inter` tactic with `ndprint` as the output predicate, `readtac` as the input predicate, an atomic goal containing an empty assumption list, the proof, and the formula, and finally the output goal. `load_tacs` is a tactic which can be called at any point during an interactive session to load a set of clauses and enter a new top-level session.

```
module nd.

import  ndtac ndprint inter_tacs.

type    inter_top   form -> nprf -> goal -> o.
type    load_tacs   modul -> goal -> goal -> o.

inter_top A P OutGoal :-
  inter (basic_io ndoutput readtac) (nil --> P # A) OutGoal.

load_tacs M InGoal OutGoal :-
  with_tacs M (inter (basic_io ndoutput readtac)) InGoal OutGoal.
```

Module `nd`: Root Module for N_I Tactic Theorem Prover

The import structure that puts all of these components together to form a tactic prover for natural deduction is given in Figure 7.1. The `goals`, `ndgoal`, `nprf`, and `fol` modules contain only declarations. The `fol` and `nprf` modules are on pages 18 and 33, respectively, in Chapter 3. All other modules appear in this chapter. The links from `ndtac` and `ndprint` to the `lists` module (see Appendix A) are not shown here. As discussed in Chapter 2, the import structure satisfies the criterion that whenever a clause calls a predicate as a subgoal, the predicate either appears in the same module, or in a module directly imported by the module in which the clause appears. Note that the tactic interpreter is a separate component (comprised of `inter_tacs`, `tacticals`, `maptac`, `goalred`, and `goals`). The module `nd` containing the top-level predicate imports the tactic interpreter in addition to the programs specific to the natural deduction theorem prover, namely `ndtac` and `ndprint`. To develop a tactic theorem prover for any other logic, we must at least specify the basic tactics and a print routine. Then, at the top-level, these two modules, in addition to the tactic interpreter must be imported.

In the import structure in Figure 7.1, both the depth-first interpreter `dfs`, and the iterative deepening interpreter `idfs` would appear immediatedly below `maptac`. In fact, they can both be imported dynamically and used as tactics within the tactic interpreter. Thus all of the interpreters can be integrated in one setting. Note that a tactic for depth-first search can also be defined in terms of the `orelse` and `repeat` tacticals. Given a formula `A` and a proof term `P`, the following query will have the same operational behavior as the query to the `dfs_interp` program given at the end of Section 7.3.1.

```
repeat (orelse (close_tac 0) (orelse and_i_tac (orelse or_i1_tac
        (orelse or_i2_tac (orelse imp_i_tac (orelse neg_i_tac
         (orelse forall_i_tac (orelse exists_i_tac (orelse false_i_tac
          (orelse (and_e_tac 0) (orelse (imp_e_tac 0)
           (orelse (forall_e_tac 0) (orelse (neg_e_tac 0)
            (orelse (or_e_tac 0) (exists_e_tac 0)))))))))))))
       (nil --> P # A) OutGoal.
```

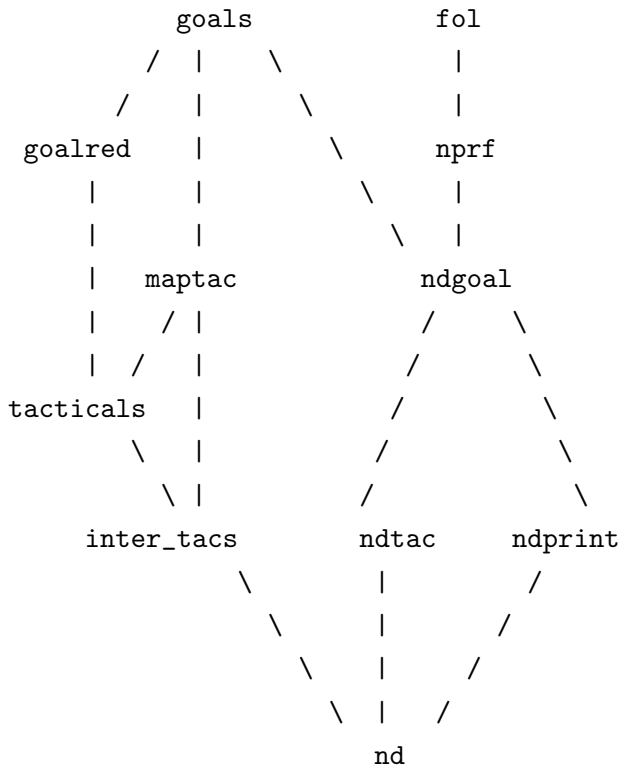

Figure 7.1: The Import Structure of a Tactic Theorem Prover for N_I

7.7 A Session With a Natural Deduction Tactic Prover

The following is a simple example session with the tactic theorem prover for N_I where the formula $q(a) \vee q(b) \supset \exists x q(x)$ is proved. Notice that a bad attempt to prove this formula is backed out of before the right solution is found.

```
?- inter_top (((q a) or (q b)) imp (exists X\(q X))) Proof OutGoal.

Assumptions:

Conclusion:
q a or q b imp exists X\(q X)
Enter tactic:  ?- imp_i_tac.

Assumptions:
1 q a or q b

Conclusion:
exists X\(q X)
Enter tactic:  ?- exists_i_tac.

Assumptions:
```

```
1 q a or q b

Conclusion:
q T
Enter tactic:  ?- or_e_tac 1.

Assumptions:
1 q a
2 q a or q b

Conclusion:
q T
Enter tactic:  ?- close_tac 0.

Assumptions:
1 q b
2 q a or q b

Conclusion:
q a
Enter tactic:  ?- backup.

Assumptions:
1 q a
2 q a or q b

Conclusion:
q T
Enter tactic:  ?- backup.

Assumptions:
1 q a or q b

Conclusion:
q T
Enter tactic:  ?- backup.

Assumptions:
1 q a or q b

Conclusion:
exists X\(q X)
Enter tactic:  ?- then (or_e_tac 1) (then exists_i_tac (close_tac 0)).

Proof = (imp_i P\(or_e (q a) (q b) P (P1\(exists_i a P1)) (P2\(exists_i b P2)))
OutGoal = (all P\((all P1\tt) && (all P2\tt))).
```

The application of the `exists_i_tac` tactic introduces a logic variable `T` for the substitution term. Then, when `(close_tac 0)` is applied, `T` is instantiated to `a`, the proof branch is completed, and this unifier is carried over to the second branch of the proof. Since this branch cannot be completed, the user backs the proof up to the point where it can be

corrected. The final compound expression that completes the proof first applies `or_e_tac` causing the search to branch, and then applies `exists_i_tac` followed by `(close_tac 0)` to each of the branches. Thus a new logic variable is introduced in each branch separately. The first is intantiated to `a` and the second to `b`, allowing the proof to be completed. Thus `Proof` is instantiated to a complete proof, and `OutGoal` is a compound goal expression whose atomic subgoals are all instances of `tt`.

7.8 Use of Meta-Language Features in Tactic Provers

As in the programs of Chapters 3 and 4, the use of higher-order features of the meta-language in this chapter is quite limited. Predicate variables are one feature which we had not used previously in the specification of theorem provers but have used extensively here, particularly in the implementation of tacticals. The use of these variables is not essential and can be eliminated. Its main advantage is that it enhances readability. To eliminate predicate variables, instead of giving tactics type `goal -> goal -> o`, we introduce a new base type `tac` for tactics, and a predicate `interp`, declared as follows, for interpreting tactics.

```
type    interp      tac -> goal -> goal -> o.
```

All tactics would be specified using this clause. For example the clause for `and_i_tac` would be as follows.

```
interp and_i_tac  (Gamma --> (and_i P1 P2) # (A and B))
                  ((Gamma --> P1 # A) && (Gamma --> P2 # B)).
```

`maptac` would then have the same type as `interp`, and its last clause would be the following.

```
maptac Tac InGoal OutGoal :- interp Tac InGoal OutGoal.
```

All other tactics and tacticals can be similarly modified. The same technique can be used to eliminate the only other predicate variables, `PrintPred` and `ReadPred` in the clause implementing `basic_io`, and `IO` in the `query` tactic.

We also made use of λProlog's polymorphism in defining the `all` and `some` goal constructors. As stated in Section 7.1.1, the instances of these type variables will generally be base types. In the tactics for N_I for example, `all` is used to quantify over objects of type `i` and `nprf`.

As in Chapters 3 and 4, quantification over functions in the examples in this chapter has been at most second-order. Here again, unification problems that arise in these programs will be at most second-order, and will often require only second-order matching.

7.9 A Contrast to ML Tactic Theorem Provers

The programming language ML is the metalanguage used in all of the other tactic theorem provers mentioned at the beginning of this chapter. There are several differences in the implementations of both tactics and tacticals in these two languages. First, the λProlog implementation of the `then` tactical is different from its ML counterpart. The λProlog implementation of `then` reveals its very simple nature: `then` is very similar to the natural join of two relations. In ML, the `then` tactical applies the first tactic to the input goal and then maps the application of the second tactic over the list of intermediate subgoals. The full list of subgoals must be built as well as the compound validation function from the results. These tasks can be quite complicated, requiring some auxiliary list processing functions. In λProlog, the analogue of a list of subgoals is a nested `andgoal` structure. These are processed by the `andgoal` clause of `maptac`. The behavior of `then` (in conjunction with `maptac`) in λProlog is actually richer than its ML counterpart since the `maptac` procedure is richer than the usual notion of a mapping function in that, in addition to nested `andgoal` structures, it handles all of the other goal structures corresponding to the λProlog search operations.

In the λProlog implementation of `then` that we presented in Section 7.4, if the first tactic succeeds and the second fails, the logic programming interpreter will backtrack and try to find a new way to successfully apply the first tactic, exhausting all possibilities before completely failing. It is also possible to implement `then` so that if the second tactic fails after a successful call to the first tactic, the full tactic still fails. To do so requires the use of cut (`!`) to restrict its backtracking behavior. We also saw that there were two implementations of the `orelse` tactical, one in the `tactical` module on page 143 that attempts all possible ways to apply either tactic before failing, and the other in the `inter_tacs` module on page 150 whose backtracking behavior was restricted by the use of cut. While both versions of both tacticals can and have been implemented in ML, in λProlog we were able to make direct use of the search and backtracking mechanisms of the logic programming interpreter to obtain the desired behavior.

Another difference between the two implementations is that in ML, tactics are functions that take a goal as input and return a pair consisting of a list of subgoals and a validation. In contrast, tactics in λProlog are relational, which is very natural when the relation being modeled is "is a proof of." The fact that input and output distinctions can be blurred makes it possible, as described in Chapter 6, for tactics to be used in both a theorem proving and proof checking context. The functional aspects of ML do not permit this dual use of tactics. The ML notion of validations is replaced in our system by (potentially much larger and more complex) proof objects. Validations like those in ML could easily be supported.

A third difference is in the manipulation of quantified formulas. In ML, manipulating quantified formulas requires that the binding be separated from its body. In logic programming, we identify a term as a universal quantification if it can be unified with the term `(forall A)`. However, since terms in λProlog represent $\beta\eta$-equivalence classes of λ-terms, the programmer does not have access to bound variable names. Although such a restriction may appear to limit access to the structure of λ-terms, we have seen that sophisticated analysis of λ-terms is still possible to perform using higher-order unification and the universal quantifier `pi`. In addition, there are certain advantages to such a restriction. For example, in the case of applying substitutions, all the renaming of bound variables is handled by the metalanguage, freeing the programmer from such concerns.

Another difference is the use of logic variables in λProlog for lazy determination of substitution instances. The example in Section 7.7 illustrated how such variables can be used so that substitution instances do not have to be given at the point where the substitution takes place.

As mentioned in Chapter 4, the Isabelle theorem prover [Pau88] contains a specification language which is essentially a subset of the higher-order hereditary Harrop formulas. Hence, it seems very likely that Isabelle could be rather directly implemented inside λProlog. Although such an implementation might achieve the same functionality as is currently available in Isabelle, it is not likely to be nearly as efficient. This is due partly to the fact that a λProlog implementation implements a general purpose programming language. An alternative to implementing interpreters as programs on top of the depth-first interpreter is to modify the interpreter to use a different control strategy. For example, tacticals and interactive tactics could be implemented at the level of the meta-language. The result would be an interactive logic programming language with depth-first search possibly available as one of the tactics. This approach to control is more like that found in Isabelle. There, the language used to specify inference rules is distinct from that used to implement tacticals and specify tactics, namely ML.

Chapter 8

Operations on Proof Terms

Each of the specifications and implementations of theorem provers in this thesis provided an illustration of how to construct proof terms that correspond to proofs or deductions in a particular proof system. In this chapter we present several operations in which such proof terms are the central data object.

First, in Section 8.1, we present a program that translates L_I proofs to N_I deductions. We obtain such a program by merging a specification of a theorem prover for L_I with one for N_I. Then in Section 8.2, we present a program for proof normalization in N_I. The procedure we present is based on the proof normalization result for first-order intuitionistic logic given in [Pra71]. Finally, in Section 8.3, we discuss how proof terms can play a role in reasoning by analogy. We will demonstrate how, within the framework of the tactic interpreter presented in Chapter 7, existing proofs can be used to guide the construction of proofs for new theorems.

8.1 Transforming L_I Proofs to N_I Proofs

We first present separately specifications for L_I and N_I that can then be merged to obtain a specification for a theorem prover for both systems. With respect to the nondeterministic interpreter, given any formula and set of assumptions, this program will simultaneously construct both an L_I and an N_I proof term. Alternatively, if also given a proof term in either system, this specification can be viewed as a translator from proofs in one system to proofs in the other. As we will see, with respect to the deterministic depth-first interpreter, the program will be complete only for translating L_I proof terms to N_I proof terms.

A specification of a theorem prover for L_I was presented in Section 3.1. In that section, there were often several alternatives in specifying the inference rules. Here, we use the specification that includes separate clauses for `or_r1` and `or_r2` to correspond to the two clauses for `or_i1` and `or_i2`, and we choose the clauses that include substitution terms in

proofs. The complete module is given on page 162.

We have also seen that there are many options in specifying the inference rules of N_I, and have noted that in particular the `ninormal` module of Section 3.5 that builds normal natural deduction proofs was motivated in part by the desire to obtain a set of clauses for N_I that corresponds to a specification of L_I. Here we further modify `ninormal` to obtain a set of clauses that will match exactly those of `liprover` on page 162. To obtain such a set of clauses, we simply need to perform the mechanical transformation described in Section 3.3 to obtain a specification that uses explicit context lists rather than meta-level implication for the discharge of assumptions. (In Chapter 7, the set of tactics for N_I was described as being obtained by two transformations on the clauses of `ninormal`. The first of these two transformations was the explicit context modification, the same transformation required here.) Performing this modification, we obtain the clauses in the `ninormal1` module on page 163.

We are now ready to put these clauses together with the clauses of `liprover`. If proof terms in each module are ignored, note that the two sets of clauses are identical, *i.e.*, each `ninormal1` clause specifying an I-rule corresponds to the clause in `liprover` specifying the sequent rule that introduces the same connective on the right of a sequent; each E-rule clause corresponds to the sequent rule clause that introduces the same connective on the left; the clause for $\bot_I$ corresponds to the clause for $\bot$-R. We now illustrate how to combine these two modules to obtain a specification of a theorem prover that constructs both kind of proof terms simultaneously. First, we combine the data structures for sequents and judgments using the same constants as before, redeclared with the following types.

```
type    '#'     nprf -> form -> judg.
type    '-->'   (list judg) -> judg -> seq.
type    '>-'    lprf -> seq -> o.
```

As in `liprover`, `>-` is the top-level predicate, but here its second argument is a judgment sequent again specified using the arrow `-->`. Natural deduction proofs are paired with formulas (using `#`) on both sides of the sequent arrow. An atomic goal now has the form `(Q >- (Gamma --> P # A))`. If such a goal succeeds then `Q` represents an L_I proof of the sequent `(Gamma --> A)`, and `P` represents an N_I deduction of `A` from `Gamma`. The following four clauses illustrate how the clauses of `liprover` and `ninormal1` are combined. The complete module, named `lniprover`, is given on page 164.

```
(and_r Q1 Q2) >- (Gamma --> (and_i P1 P2) # (A and B)) :-
   Q1 >- (Gamma --> P1 # A), Q2 >- (Gamma --> P2 # B).

(and_l Q) >- (Gamma --> PC # C) :- memb (P # (A and B)) Gamma,
   Q >- ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> PC # C).

(forall_r Q) >- (Gamma --> (forall_i P) # (forall A)) :-
   pi Y\ ((Q Y) >- (Gamma --> (P Y) # (A Y))).
```

```
(imp_r Q) >- (Gamma --> (imp_i P) # (A imp B)) :-
   pi PA\ (Q >- (((PA # A)::Gamma) --> (P PA) # B)).
```

The first two clauses illustrate how N_I proof terms are associated with formulas within a sequent on the right and left, respectively, while the L_I proof terms are associated with the entire sequent in the top-level relation. The third clause illustrates how universal quantification at the meta-level is used to introduce a new variable `Y` to instantiate the quantified formula `(forall A)`, and to handle simultaneously the provisos on both $\forall$-R and $\forall$-I. (See Sections 3.1 and 3.2.) Both `Q` and `P` are abstractions over this variable since the variable may appear inside either proof term in the subgoal. The fourth clause illustrates the discharge of assumptions in N_I. Universal quantification is used to introduce the variable `PA` to represent a proof for hypothesis `A`. `PA` may appear in the proof for `B`, so `P` is an abstraction over this variable. As in the `liprover` clause, `Q` represents an L_I proof of the premise sequent of the $\supset$-R rule. Of course, a natural deduction proof will not appear inside a sequent proof term, so `Q` does not need to be an abstraction over `PA`.

As mentioned earlier, with respect to the non-deterministic interpreter, the `lniprover` module can take on several roles. If only `A` is specified in a query of the form `(Q >- (nil --> P # A))`, the program behaves as a theorem prover, and simultaneously constructs proofs in both proof systems. If `P` is also specified, then the program acts as a proof transformer, transforming an N_I proof `P` to an L_I proof `Q`. Conversely, an L_I proof `Q` can be used to guide the construction of an N_I proof `P`. In fact, the program is complete with respect to the deterministic depth-first interpreter for this latter transformation, for the same reason that `liprover` is complete as a proof checker. The constant at the head of the proof term `Q` uniquely determines which definite clause must be used at each step. Some backtracking may be necessary, but as long as `Q` is a proof of `(nil --> A)`, the interpreter will succeed in constructing the N_I proof term `P`. The reverse transformation, on the other hand, is not always possible for the same reason that `ninormal` could not serve as a proof checker. Any of the clauses for `and_e`, `imp_e`, or `forall_e` could cause the interpreter to enter an infinite loop when the proof term `Q` is not specified. Of course the program can also serve as a simultaneous proof checker for both kinds of proof terms.

The fact that the L_I proof to N_I deduction transformation is deterministic illustrates that this operation is "functional," *i.e.*, there is exactly one N_I deduction corresponding to each cut-free L_I proof. On the other hand, there may be many L_I proofs that correspond to one N_I deduction. For example, the N_I deduction of $\forall x q(x) \supset \exists x q(x)$ (in a language containing a constant c) in Figure 8.1 (a) can be seen to correspond to the two sequent proofs in Figure 8.1 (b) and (c). In a sense, sequent proofs contain extra information corresponding to the order in which rules are applied.

As shown in Section 3.3, any specification of a natural deduction system that uses meta-level implication for the discharge of assumptions can be converted to a specification

$$\dfrac{\dfrac{\dfrac{\forall xq(x)}{q(c)}\ \forall\text{-E}(c)}{\exists xq(x)}\ \exists\text{-I}(c)}{\forall xq(x) \supset \exists xq(x)}\ \supset\text{-I}$$

(a)

$$\dfrac{\dfrac{\dfrac{q(c) \longrightarrow q(c)}{q(c) \longrightarrow \exists xq(x)}\ \exists\text{-R}}{\forall xq(x) \longrightarrow \exists xq(x)}\ \forall\text{-L}}{\longrightarrow \forall xq(x) \supset \exists xq(x)}\ \supset\text{-R}$$

(b)

$$\dfrac{\dfrac{\dfrac{q(c) \longrightarrow q(c)}{\forall xq(x) \longrightarrow q(c)}\ \forall\text{-L}}{\forall xq(x) \longrightarrow \exists xq(x)}\ \exists\text{-R}}{\longrightarrow \forall xq(x) \supset \exists xq(x)}\ \supset\text{-R}$$

(c)

Figure 8.1: L_I and N_I Proofs of $\forall xq(x) \supset \exists xq(x)$

using explicit assumption lists. In fact, any such explicit context specification can be viewed as a specification of a sequent style proof system for the same logic. In the case of N_I, as we have seen, the inference rules of N_I could be specified in such a way that the corresponding sequent system was exactly L_I without the cut rule. The corresponding systems for classical logic, N_C and L_C, on the other hand, cannot be so easily related in this way.

The correspondence between sequential and natural deduction systems for first-order intuitionistic logic is well-known and has been formalized in [Zuc74] and [Pot77]. There, the relation between cut-elimination and proof normalization is also explored. Here, by merging specifications of L_I and N_I, we were able to obtain both a declarative illustration and an operational description of the correspondence between these two systems.

```
module liprover.

import  lprf lists.

kind    seq             type.

type    '-->'           (list  form) -> form -> seq.
type    '>-'            lprf -> seq -> o.

(initial A) >- (Gamma --> A) :- memb A Gamma.

(and_r Q1 Q2) >- (Gamma --> (A and B)) :- Q1 >- (Gamma --> A),
                                         Q2 >- (Gamma --> B).

(or_r1 Q) >- (Gamma --> (A or B)) :- Q >- (Gamma --> A).

(or_r2 Q) >- (Gamma --> (A or B)) :- Q >- (Gamma --> B).

(imp_r Q) >- (Gamma --> (A imp B)) :- Q >- ((A::Gamma) --> B).

(neg_r Q) >- (Gamma --> (neg A)) :- Q >- ((A::Gamma) --> false).

(forall_r Q) >- (Gamma --> (forall A)) :- pi Y\ ((Q Y) >- (Gamma --> (A Y))).

(exists_r T Q) >- (Gamma --> (exists A)) :- Q >- (Gamma --> (A T)).

(false_r Q) >- (Gamma --> A) :- Q >- (Gamma --> false).

(and_l Q) >- (Gamma --> C) :- memb (A and B) Gamma,
                              Q >- ((A::B::Gamma) --> C).

(imp_l Q1 Q2) >- (Gamma --> C) :- memb (A imp B) Gamma,
                                  Q1 >- (Gamma --> A),
                                  Q2 >- ((B::Gamma) --> C).

(forall_l T Q) >- (Gamma --> C) :- memb (forall A) Gamma,
                                   Q >- (((A T)::Gamma) --> C).

(neg_l Q) >- (Gamma --> false) :- memb (neg A) Gamma,
                                  Q >- (Gamma --> A).

(or_l Q1 Q2) >- (Gamma --> C) :- memb (A or B) Gamma,
                                 Q1 >- ((A::Gamma) --> C),
                                 Q2 >- ((B::Gamma) --> C).

(exists_l Q) >- (Gamma --> C) :- memb (exists A) Gamma,
                                 pi Y\ ((Q Y) >- (((A Y)::Gamma) --> C)).
```

Module `liprover`: Specification of the L_I Inference Rules Without Cut

```
module ninormal1.

import  nprf lists.

kind    judg            type.

type    '#'     nprf -> form -> judg.
type    '-->'   (list judg) -> judg -> o.

Gamma --> P # A :- memb (P # A) Gamma.

Gamma --> (and_i P1 P2) # (A and B) :- Gamma --> P1 # A,
                                       Gamma --> P2 # B.

Gamma --> (or_i1 P) # (A or B) :- Gamma --> P # A.

Gamma --> (or_i2 P) # (A or B) :- Gamma --> P # B.

Gamma --> (imp_i P) # (A imp B) :-
   pi PA\ (((PA # A)::Gamma) --> (P PA) # B).

Gamma --> (neg_i P) # (neg A) :-
   pi PA\ (((PA # A)::Gamma) --> (P PA) # false).

Gamma --> (forall_i P) # (forall A) :-
   pi Y\ (Gamma --> (P Y) # (A Y)).

Gamma --> (exists_i T P) # (exists A) :- Gamma --> P # (A T).

Gamma --> (false_i P) # A :- Gamma --> P # false.

Gamma --> PC # C :- memb (P # (A and B)) Gamma,
   (((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> PC # C.

Gamma --> PC # C :- memb (P2 # (A imp B)) Gamma,
   Gamma --> P1 # A,
   (((imp_e A P1 P2) # B)::Gamma) --> PC # C.

Gamma --> PC # C :- memb (P # (forall A)) Gamma,
   (((forall_e T A P) # (A T))::Gamma) --> PC # C.

Gamma --> (neg_e A P1 P2) # false :- memb (P2 # (neg A)) Gamma,
                                    Gamma --> P1 # A.

Gamma --> (or_e A B P P1 P2) # C :- memb (P # (A or B)) Gamma,
   pi PA\ (((PA # A)::Gamma) --> (P1 PA) # C),
   pi PB\ (((PB # B)::Gamma) --> (P2 PB) # C).

Gamma --> (exists_e A P1 P2) # B :- memb (P1 # (exists A)) Gamma,
   pi Y\ (pi P\ (((P # (A Y))::Gamma) --> (P2 Y P) # B)).
```

Module `ninormal`: Explicit Context Specification of N_I that Constructs Normal Deductions

```
module lniprover.

import  nprf lprf lists.

kind    judg            type.

type    '#'     nprf -> form -> judg.
type    '-->'   (list judg) -> judg -> seq.
type    '>-'    lprf -> seq -> o.

(initial A) >- (Gamma --> P # A) :- memb (P # A) Gamma.

(and_r Q1 Q2) >- (Gamma --> (and_i P1 P2) # (A and B)) :-
   Q1 >- (Gamma --> P1 # A), Q2 >- (Gamma --> P2 # B).

(or_r1 Q) >- (Gamma --> (or_i1 P) # (A or B)) :- Q >- (Gamma --> P # A).

(or_r2 Q) >- (Gamma --> (or_i2 P) # (A or B)) :- Q >- (Gamma --> P # B).

(imp_r Q) >- (Gamma --> (imp_i P) # (A imp B)) :-
   pi PA\ (Q >- (((PA # A)::Gamma) --> (P PA) # B)).

(neg_r Q) >- (Gamma --> (neg_i P) # (neg A)) :-
   pi PA\ (Q >- (((PA # A)::Gamma) --> (P PA) # false)).

(forall_r Q) >- (Gamma --> (forall_i P) # (forall A)) :-
   pi Y\ ((Q Y) >- (Gamma --> (P Y) # (A Y))).

(exists_r T Q) >- (Gamma --> (exists_i T P) # (exists A)) :-
   Q >- (Gamma --> P # (A T)).

(false_r Q) >- (Gamma --> (false_i P) # A) :- Q >- (Gamma --> P # false).

(and_l Q) >- (Gamma --> PC # C) :- memb (P # (A and B)) Gamma,
   Q >- ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> PC # C).

(imp_l Q1 Q2) >- (Gamma --> PC # C) :- memb (P2 # (A imp B)) Gamma,
   Q1 >- (Gamma --> P1 # A),
   Q2 >- ((((imp_e A P1 P2) # B)::Gamma) --> PC # C).

(forall_l T Q) >- (Gamma --> PC # C) :- memb (P # (forall A)) Gamma,
   Q >- ((((forall_e T A P) # (A T))::Gamma) --> PC # C).

(neg_l Q) >- (Gamma --> (neg_e A P1 P2) # false) :-
   memb (P2 # (neg A)) Gamma, Q >- (Gamma --> P1 # A).

(or_l Q1 Q2) >- (Gamma --> (or_e A B P P1 P2) # C) :-
   memb (P # (A or B)) Gamma,
   pi PA\ (Q1 >- (((PA # A)::Gamma) --> (P1 PA) # C)),
   pi PB\ (Q2 >- (((PB # B)::Gamma) --> (P2 PB) # C)).

(exists_l Q) >- (Gamma --> (exists_e A P1 P2) # B) :-
   memb (P1 # (exists A)) Gamma,
   pi Y\ (pi P\ ((Q Y) >- (((P # (A Y))::Gamma) --> (P2 Y P) # B))).
```

Module `lniprover`: Proof Transformer from L_I Proofs to N_I Deductions

8.2 Proof Normalization in N_I

Proof normalization, as presented in [Pra71], is based on proof reductions that remove maximal formulas and reduce the length of maximal segments. (See Section 3.5 for definitions related to normal deductions.) These reductions are given in Figure 8.2. For simplification, we exclude the rules for negation from this presentation. Negation can be defined in terms of implication, *e.g.*, $\neg A$ is defined to be $A \supset \bot$. The rules for negation are then special cases of the rules for implication.

There are two reductions for each connective, one for the case when the maximal formula is the conclusion of an I-rule, and one for the case when it is the conclusion of $\bot_I$. The last two reductions reduce the length of maximal segments. In order for either to be applied, the last occurrence of C in the segments ending with an application of $\vee$-E or $\exists$-E must be the major premise of an E-rule or $\bot_I$, and the first occurrence of C in Π_2 and/or Π_3 must be the conclusion of an I-rule. In either of these two reductions, when the E-rule at the root is $\wedge$-E or $\forall$-E, Π_4 and Π_5 are empty. When it is $\supset$-E or $\exists$-E, Π_5 is empty. In the case of $\supset$-E, Π_4 should be written to the left. These two reductions can also be used to reduce the length of E-segments. For this task, they can be applied whenever the last occurrence of C is the major premise of an E-rule. The first occurrence need not be the conclusion of an I-rule or $\bot_I$. Note that these last two reductions involve an implicit change to the discharge functions. For example, consider $\vee$-reduction. In the deduction on the left, any occurrence of A in Π_2 that is discharged at the end-formula C in Π_2 is discharged at D, the middle premise of the application of $\vee$-E, in the reduced deduction. In all reductions except the last two, the discharge functions remain unchanged. In the last reduction, in order to insure that the proviso on the reduced deduction holds, we assume that every parameter to an application of $\exists$-E occurs only in the subtree above the minor premise. By the lemma on parameters in [Pra65], the parameters in any deduction can be renamed to satisfy this criteria. Thus, in the last reduction in Figure 8.2, y will not occur in D or anywhere in Π_4 or Π_5, and the proviso will be satisfied.

We first present a program that will use these reductions to remove E-segments, and thus transform arbitrary deductions to E-normal deductions. First, we define a predicate used to specify reductions. We call this predicate `redex` since proof reductions for N_I share some similarities with reductions in the λ-calculus. A predicate of the same name was used in the `convert` module of Section 4.2 to specify β and η redexes for untyped terms, and in the `lfconv` module in Section 5.2 to specify β-redexes for LF terms and types. Here `redex` is declared with the following type.

```
type    redex   nprf -> nprf -> o.
```

$$\dfrac{\dfrac{\Pi_1 \quad \Pi_2}{A_1 \quad A_2}}{\dfrac{A_1 \wedge A_2}{A_i}\,\wedge\text{-E}_i}\,\wedge\text{-I} \quad \Rightarrow \quad \Pi_i \atop A_i \qquad\qquad \dfrac{\dfrac{\Pi_1}{\bot}}{\dfrac{A_1 \wedge A_2}{A_i}\,\wedge\text{-E}_i}\,\bot_I \quad \Rightarrow \quad \dfrac{\Pi_i}{\bot}{A_i}$$

$i = 1$ or 2

$$\dfrac{\dfrac{\Pi}{A_i}}{\dfrac{A_1 \vee A_2}{C}\,\vee\text{-E}}\,\vee\text{-I} \quad \Rightarrow \quad \Pi \atop (A_i) \atop \Pi_i \atop C \qquad\qquad \dfrac{\dfrac{\Pi}{\bot}}{\dfrac{A_1 \vee A_2}{C}\,\vee\text{-E}}\,\bot_I \quad \Rightarrow \quad \dfrac{\Pi}{\dfrac{\bot}{C}}$$

$$\dfrac{\Pi_1 \quad \dfrac{(A)\ \Pi_2}{B}}{\dfrac{A}{B}\,\supset\text{-E}}\,\supset\text{-I} \quad \Rightarrow \quad \Pi_1 \atop (A) \atop \Pi_2 \atop B \qquad\qquad \dfrac{\Pi_1 \quad \dfrac{\Pi_2}{\bot}}{\dfrac{A}{B}\,\supset\text{-E}}\,\bot_I \quad \Rightarrow \quad \dfrac{\Pi_2}{\dfrac{\bot}{B}}$$

$$\dfrac{\dfrac{\Pi}{[y/x]A}}{\dfrac{\forall x A}{[t/x]A}\,\forall\text{-E}(t)}\,\forall\text{-I}(y) \quad \Rightarrow \quad [t/y]\Pi \atop [t/x]A \qquad\qquad \dfrac{\dfrac{\Pi}{\bot}}{\dfrac{\forall x A}{[t/x]A}\,\forall\text{-E}(t)}\,\bot_I \quad \Rightarrow \quad \dfrac{\Pi}{\dfrac{\bot}{[t/x]A}}$$

$$\dfrac{\Pi_1 \quad \dfrac{([y/x]A)\ \Pi_2}{B}}{\dfrac{[t/x]A}{\exists x A}\,\exists\text{-I}(t)}\,\exists\text{-E}(y) \quad \Rightarrow \quad \Pi_1 \atop ([t/x]A) \atop [t/y]\Pi_2 \atop B \qquad\qquad \dfrac{\Pi_1 \quad \dfrac{([y/x]A)\ \Pi_2}{B}}{\dfrac{\bot}{\exists x A}\,\bot_I}\,\exists\text{-E}(y) \quad \Rightarrow \quad \dfrac{\Pi_1}{\dfrac{\bot}{B}}$$

$$\dfrac{\Pi_1 \quad \dfrac{\Pi_2 \quad \Pi_3}{C \quad C}\,\vee\text{-E} \quad \dfrac{\Pi_4 \quad \Pi_5}{F \quad F}}{D} \quad \Rightarrow \quad \dfrac{\Pi_1 \quad \dfrac{\dfrac{\Pi_2 \quad \Pi_4 \quad \Pi_5}{C \quad F \quad F}}{D} \quad \dfrac{\dfrac{\Pi_3 \quad \Pi_4 \quad \Pi_5}{C \quad F \quad F}}{D}\,\vee\text{-E}}{D}$$

$$\dfrac{\Pi_1 \quad \dfrac{\Pi_2}{C}\,\exists\text{-E}(y) \quad \dfrac{\Pi_4 \quad \Pi_5}{F \quad F}}{D} \quad \Rightarrow \quad \dfrac{\Pi_1 \quad \dfrac{\dfrac{\Pi_2 \quad \Pi_4 \quad \Pi_5}{C \quad F \quad F}}{D}\,\exists\text{-E}(y)}{D}$$

Figure 8.2: Reductions for Proof Normalization in N_I

The first argument is any N_I deduction that can be reduced by one of the above transformations, and the second is the proof term representing the reduced deduction. Operationally, for proof normalization, the first proof term will be the input proof, and the second the output proof. We will continue to use the proof representation given by the `nprf` module on page 33 that includes substitution information and some formulas inside proof terms. For reference, the `ndredex` module on page 171 contains a specification of all of the reductions in Figure 8.2. First, consider the reductions for removing maximal formulas that are the conclusion of an I-rule. They are specified by the following 7 clauses.

```
redex (and_e1 B (and_i P1 P2)) P1.
redex (and_e2 A (and_i P1 P2)) P2.
redex (imp_e A P1 (imp_i P2)) (P2 P1).
redex (or_e A B (or_i1 P1) P2 P3) (P2 P1).
redex (or_e A B (or_i2 P1) P2 P3) (P3 P1).
redex (forall_e T A (forall_i P)) (P T).
redex (exists_e A (exists_i T P1) P2) (P2 T P1).
```

Operationally, the reductions for $\wedge$-E_1 and $\wedge$-E_2 involve pattern matching that is essentially first-order. One of these clauses can be used whenever the input proof term matches one of the two patterns for a deduction with a conjunctive maximal formula. The second argument is then the appropriate subproof. The clause for implication also involves pattern matching, but in this case `P2` is functional: it is a function from proofs of `A` to proofs of some formula `B`. In specifying and implementing theorem provers, we have seen many times how the application of λ-terms can be used to specify the substitution of terms for variables in first-order formulas. Here, the application of `P2` to `P1` very naturally specifies $\supset$-reduction, the operation of substituting a proof into another proof at certain leaf nodes. Operationally, β-conversion replaces the bound variable in `P2`, which serves as a "placeholder" for proofs of `A`, by `P1`, an actual proof of `A`. Note that although this operation is quite natural to specify, it can be expensive operationally since the proof `P1` (which may be large) can get substituted in many different places in `P2`. The reductions for disjunction similarly apply proof functions to subproofs. In the clause for $\forall$-reduction, the term `P` is a function from first-order terms to proofs. Again the application of λ-terms is used to specify the desired operation, in this case for substitution of first-order terms for variables in deductions. Finally, the clause for $\exists$-reduction involves the application of the proof function `P2` to both a term and a proof.

The clauses for the removal of maximal formulas that are the conclusion of $\bot_I$ are below. They also operate by simple pattern matching, in this case without the need for any β-reduction.

```
redex (and_e1 B (false_i P)) (false_i P).
redex (and_e2 A (false_i P)) (false_i P).
redex (imp_e A (false_i P1) P2) (false_i P1).
redex (or_e A B (false_i P1) P2 P3) (false_i P).
```

```
redex (forall_e T A (false_i P)) (false_i P).
redex (exists_e A (false_i P1) P2) (false_i P1).
```

The last two reduction rules in Figure 8.2 that reduce the length of E-segments are applicable when the last inference is any E-rule. Thus, there will be one clause for each E-rule. The clauses for the case when the last rule is $\supset$-E are as follows. The other clauses are similar. (See page 171).

```
redex (imp_e C P4 (or_e A B P1 Q\(P2 Q) Q\(P3 Q)))
      (or_e A B P1 (Q\ (imp_e C P4 (P2 Q))) (Q\ (imp_e C P4 (P3 Q)))).
redex (imp_e C P4 (exists_e A P1 Y\Q\(P2 Y Q)))
      (exists_e A P1 (Y\Q\ (imp_e C P4 (P2 Y Q)))).
```

Second-order matching is required to match a proof term to the pattern given by the first argument in each clause. The $\beta\eta$-long forms `Q\(P2 Q)`, `Q\(P3 Q)`, and `Y\Q\(P2 Y Q)` are used here to make the abstractions in these terms explicit. In the second argument, the scope of the bound variable `Q` (and also `Y` in the second clause) is modified. This change in scope of `Q` corresponds to the modification of the discharge function that occurs when the corresponding reduction in Figure 8.2 is applied. Note that in the expression `(Y\Q\(imp_e C P4 (P2 Y Q)))` in the second clause, `Y` will not occur free in `P4`. Operationally, when this clause is used for proof normalization, renaming of variables at the meta-level may be required to avoid a clash between the bound variable `Y` and any free variables in `P4`. This renaming corresponds to the renaming of parameters that may need to occur in a deduction to insure that every parameter to an application of $\exists$-E occurs only in the subtree above the minor premise.

In [Pra71], a strong normalization result is established for natural deduction with respect to the reductions of Figure 8.2. Thus any sequence of reductions will eventually terminate in a normal deduction. A very simple strategy for reducing a deduction to normal form, which we adopt here, is to traverse a tree from the root upwards, apply a reduction to the first maximal formula or E-segment encountered, and then start over at the root of the new tree. Although this strategy is not necessarily the most efficient, we will see that it is quite straightforward to implement. The complete program is given by the `ndredex` module just described and the `ndnormalize` module on page 172, which imports `ndredex`. In addition to the `redex` predicate, we will need the following two predicates.

```
type    red1    nprf -> nprf -> o.
type    reduce  nprf -> nprf -> o.
```

The predicate `red1` relates two proof terms if the second is obtained from the first by one reduction. The `redex` clauses are a special case of this relation when the maximal formula or last occurrence in an E-segment occur just above the root. Thus we include the following clause.

```
red1 P1 P2 :- redex P1 P2.
```

We must also specify clauses for reductions that occur further up in the tree. Those that occur above the root of a tree whose last rule is an application of $\wedge$-I and $\supset$-I are as follows. (See page 172 for the complete list.)

```
red1 (and_i P1 P2) (and_i POut P2) :- red1 P1 POut.
red1 (and_i P1 P2) (and_i P1 POut) :- red1 P2 POut.
red1 (imp_i P) (imp_i POut) :- pi PA\ (red1 (P PA) (POut PA)).
```

Operationally, the `red1` clauses descend into a tree until a maximal formula or E-segment is encountered, and then perform a reduction. In the clauses for `and_i` the second proof term is a reduced form of the first if a reduction is applied in either subtree. In the clause for `imp_i` universal quantification is used to descend through the abstraction in `P`. The GENERIC search operation introduces a constant to replace the bound variable in `P`. This constant serves as the bound variable name in the resulting subgoal. If the subgoal succeeds, the proof function `POut` is the abstraction over that constant in the reduced deduction.

The following clauses for the `reduce` predicate complete the implementation.

```
reduce PIn POut :- red1 PIn PMid, reduce PMid POut.
reduce P P.
```

Nondeterministically, a goal of the form `(reduce P1 P2)` succeeds if `P2` is obtained from `P1` by zero or more reductions. With respect to the depth-first interpreter, the order of the above two clauses is very important. The above order implements proof normalization, as desired. The first clause repeatedly applies reductions until no more can be done, at which point the second clause terminates the execution with E-normal deduction `P`. Thus, `(reduce P1 P2)` succeeds only when `P2` is in E-normal form.

There are several alternatives in implementing a normalization algorithm for N_I. First, if normal rather than E-normal deductions are desired, an extra check would be necessary to see if the first formula in a segment is the conclusion of an I-rule or $\perp_I$ before applying either of the last two reductions. Additionally, in [Pra71], there are several other definitions of normal. We define and implement one other here. A *redundant* application of $\vee$-E or $\exists$-E is an application such that no assumption is discharged at the end-formula in one of the minor premises. A *fully* normal deduction is a deduction that contains no redundant applications of $\vee$-E or $\exists$-E. The reductions in Figure 8.3 illustrate how to remove redundant applications. These reductions are specified by the following clauses.

```
redex (or_e A B P1 Q\P2 P3) P2.
redex (or_e A B P1 P2 Q\P3) P3.
redex (exists_e A P1 Y\Q\P2) P2.
```

In the first clause, the term `Q\P2` represents an abstraction where the bound variable `Q` does not appear in `P2`. The other clauses are similar. Such vacuous quantification indicates a redundant application of the corresponding rule. By adding these clauses to the `ndredex`

$$\dfrac{\begin{array}{ccc}\Pi_1 & \Pi_2 & \Pi_3\\ A \vee B & C & C\end{array}}{C} \quad \Rightarrow \quad \begin{array}{c}\Pi_i\\ C\end{array} \qquad\qquad \dfrac{\begin{array}{cc}\Pi_1 & \Pi_2\\ \exists x A & C\end{array}}{C} \quad \Rightarrow \quad \begin{array}{c}\Pi_2\\ C\end{array}$$

In $\vee$-reduction, $i = 2$ or 3 and no assumption is discharged at the end-formula in Π_i.

In $\exists$-reduction, no assumption is discharged at the end-formula in Π_2.

Figure 8.3: Reductions for Removing Redundant Applications of $\vee$-E or $\exists$-E

module on pages 171, the `ndnormalize` module on 172 becomes a program that reduces deductions to fully E-normal form.

```
module ndredex.

import nprf.

type    redex   nprf -> nprf -> o.

redex (and_e1 B (and_i P1 P2)) P1.
redex (and_e2 A (and_i P1 P2)) P2.
redex (imp_e A P1 (imp_i P2)) (P2 P1).
redex (or_e A B (or_i1 P1) P2 P3) (P2 P1).
redex (or_e A B (or_i2 P1) P2 P3) (P3 P1).
redex (forall_e T A (forall_i P)) (P T).
redex (exists_e A (exists_i T P1) P2) (P2 T P1).

redex (and_e1 B (false_i P)) (false_i P).
redex (and_e2 A (false_i P)) (false_i P).
redex (imp_e A (false_i P1) P2) (false_i P1).
redex (or_e A B (false_i P1) P2 P3) (false_i P).
redex (forall_e T A (false_i P)) (false_i P).
redex (exists_e A (false_i P1) P2) (false_i P1).

redex (and_e1 C (or_e A B P1 P2 P3))
      (or_e A B P1 (Q\ (and_e1 C (P2 Q))) (Q\ (and_e1 C (P3 Q)))).
redex (and_e2 C (or_e A B P1 P2 P3))
      (or_e A B P1 (Q\ (and_e2 C (P2 Q))) (Q\ (and_e2 C (P3 Q)))).
redex (or_e C D (or_e A B P1 P2 P3) P4 P5)
      (or_e A B P1 (Q\ (or_e C D (P2 Q) P4 P5))
                   (Q\ (or_e C D (P3 Q) P4 P5))).
redex (imp_e C P4 (or_e A B P1 P2 P3))
      (or_e A B P1 (Q\ (imp_e C P4 (P2 Q))) (Q\ (imp_e C P4 (P3 Q)))).
redex (forall_e T C (or_e A B P1 P2 P3))
      (or_e A B P1 (Q\ (forall_e T C (P2 Q))) (Q\ (forall_e T C (P3 Q)))).
redex (exists_e C (or_e A B P1 P2 P3) P4)
      (or_e A B P1 (Q\ (exists_e C (P2 Q) P4)) (Q\ (exists_e C (P3 Q) P4))).

redex (and_e1 C (exists_e A P1 P2))
      (exists_e A P1 (Y\Q\ (and_e1 C (P2 Y Q)))).
redex (and_e2 C (exists_e A P1 P2))
      (exists_e A P1 (Y\Q\ (and_e2 C (P2 Y Q)))).
redex (or_e C D (exists_e A P1 P2) P4 P5)
      (exists_e A P1 (Y\Q\ (or_e C D (P2 Y Q) P4 P5))).
redex (imp_e C P4 (exists_e A P1 P2))
      (exists_e A P1 (Y\Q\ (imp_e C P4 (P2 Y Q)))).
redex (forall_e T C (exists_e A P1 P2))
      (exists_e A P1 (Y\Q\ (forall_e T C (P2 Y Q)))).
redex (exists_e C (exists_e A P1 P2) P4)
      (exists_e A P1 (Y\Q\ (exists_e C (P2 Y Q) P4))).
```

Module `ndredex`: Reductions for Proof Normalization in N_I

```
module ndnormalize.

import ndredex.

type    red1        nprf -> nprf -> o.
type    reduce      nprf -> nprf -> o.

red1 P1 P2 :- redex P1 P2.
red1 (and_i P1 P2) (and_i POut P2) :- red1 P1 POut.
red1 (and_i P1 P2) (and_i P1 POut) :- red1 P2 POut.
red1 (or_i1 P) (or_i1 POut) :- red1 P POut.
red1 (or_i2 P) (or_i2 POut) :- red1 P POut.
red1 (imp_i P) (imp_i POut) :- pi PA\ (red1 (P PA) (POut PA)).
red1 (forall_i P) (forall_i POut) :- pi Y\ (red1 (P Y) (POut T)).
red1 (exists_i T P) (exists_i T POut) :- red1 P POut.
red1 (and_e1 B P) (and_e1 B POut) :- red1 P POut.
red1 (and_e2 A P) (and_e2 A POut) :- red1 P POut.
red1 (imp_e A P1 P2) (imp_e A POut P2) :- red1 P1 POut.
red1 (imp_e A P1 P2) (imp_e A P1 POut) :- red1 P2 POut.
red1 (or_e A B P1 P2 P3) (or_e A B POut P2 P3) :- red1 P1 POut.
red1 (or_e A B P1 P2 P3) (or_e A B P1 POut P3) :-
   pi PA\ (red1 (P2 PA) (POut PA)).
red1 (or_e A B P1 P2 P3) (or_e A B P1 P2 POut) :-
   pi PB\ (red1 (P3 PB) (POut PB)).
red1 (forall_e T A P) (forall_e T A POut) :- red1 P POut.
red1 (exists_e A P1 P2) (exists_e A POut P2) :- red1 P1 POut.
red1 (exists_e A P1 P2) (exists_e A P1 POut) :-
   pi Y\ (pi P\ (red1 (P2 Y P) (POut Y P))).

reduce PIn POut :- red1 PIn PMid, reduce PMid POut.
reduce P P.
```

Module `ndnormalize`: Proof Normalization for N_I

8.3 Some Tactics for Proof by Analogy

Proof by analogy has been recognized as a powerful tool used in human mathematical reasoning, one that is important yet difficult to incorporate in machine theorem provers [Ble86, Ble77]. Work in the area has largely centered on constructing analogous proofs based on structural similarities [BCP86, C$^+$86, dlTC87]. In this section, we illustrate how this kind of proof by analogy can be naturally incorporated into the tactic theorem proving environment discussed in the previous chapter. The tactics for analogy that we present here can be incorporated into the interpreter component (see Figure 7.1), and thus can be made available to any tactic theorem prover or other program adopting the tactic interpreter as its basic control mechanism.

The proof terms that are constructed by any of the theorem provers presented in this dissertation can be viewed as a device for keeping a record of which inference rules were applied at each step in the process of proving a particular formula. In proof by analogy, we use such records to determine the sequence of steps to follow in attempting a new proof for a different formula. In Chapter 7, we saw that basic judgments of objects logics such as "is a proof of" were encoded as atomic goals of the tactic interpreter. For example, `(nil --> P # A)` is an example of an atomic goal for the tactic theorem prover for N_I (see Section 7.1.2) encoding the fact that `P` is a proof of `A`. The programs we describe for proof by analogy will take two such goals as input arguments. One of the two goals (generally the first) will be the *guiding goal* containing a fully specified formula and proof. The other, which we call the *target goal*, contains the formula for which we want to find an analogous proof, and its proof term which generally starts out unspecified. For example, if `P` and `A` in the above goal are fully specified and `B` is a formula for which we want to build an analogous proof, then `(nil --> Q # B)` may serve as the target goal where `Q` is a logic variable. It is important to note that the programs we present here will work for arbitrary atomic goals, although we use natural deduction as an example throughout this section.

The tactics and tacticals that we define for proof by analogy will generally have at least four arguments in the following order: the guiding goal, an output goal for the guiding goal, a target goal, and an output goal for the target goal. We first present the `mapcopy` program, which is analogous to the `maptac` program presented in Section 7.1.3. Its role is to break down the conjunctive goal structure of the guiding and target goals in parallel. The clauses for this program are contained in the `mapcopy` module below. We only include clauses for the `tt`, `&&`, and `all` goal constructors. The others can be defined similarly. The first two clauses illustrate that if either input goal is completed (is equal to `tt`), `maptac` terminates with success. The other output goal will contain the subgoals that must still be completed (if any) in order to finish the incomplete proof. If the target goal is unfinished, some other means must then be used to complete it. In the clause for conjunctive goals, the first conjunct of the guiding goal becomes the guiding goal for the first conjunct of the target goal, and similarly for the second conjuncts of each goal. Note that in order for this clause to be used at all, both the guiding and target goals must be conjuncts. Thus the conjunctive branching structure of two goals must be exactly the same in order for the copying procedure to proceed. There are two clauses for the `all` goal constructor, one each for a universally quantified goal in the guiding and target goals. Operationally, the GENERIC search operation is used in each clause separately to introduce a new constant into the corresponding goal. As in the `maptac` program, the last clause above applies the tactic `Tac` directly to atomic goals.

```
module mapcopy.

import  goals.

type    mapcopy         (goal -> goal -> goal -> goal -> o) ->
                          goal -> goal -> goal -> goal -> o.

mapcopy Tac GGoal GGoal tt tt.
mapcopy Tac tt tt TGoal TGoal.

mapcopy Tac (GInGoal1 && GInGoal2) (GOutGoal1 && GOutGoal2)
            (TInGoal1 && TInGoal2) (TOutGoal1 && TOutGoal2) :-
   mapcopy Tac GInGoal1 GOutGoal1 TInGoal1 TOutGoal1,
   mapcopy Tac GInGoal2 GOutGoal2 TInGoal2 TOutGoal2.

mapcopy Tac (all GInGoal) (all GOutGoal) TInGoal TOutGoal :-
   pi X\ (mapcopy Tac (GInGoal X) (GOutGoal X) TInGoal TOutGoal).

mapcopy Tac GInGoal GOutGoal (all TInGoal) (all TOutGoal) :-
   pi X\ (mapcopy Tac GInGoal GOutGoal (TInGoal X) (TOutGoal X)).

mapcopy Tac GInGoal GOutGoal TInGoal TOutGoal :-
   Tac GInGoal GOutGoal TInGoal TOutGoal.
```

Module `mapcopy`: Processing Compound Goals in Proof By Analogy

```
module copy.

import  mapcopy lists.

type    copy1   list (goal -> goal -> o) ->
                  goal -> goal -> goal -> goal -> o.
type    copy    list (goal -> goal -> o) ->
                  goal -> goal -> goal -> goal -> o.

copy1 TacLis GInGoal GOutGoal TInGoal TOutGoal :-
  memb Tac TacLis,
  Tac GInGoal GOutGoal,
  Tac TInGoal TOutGoal.

copy TacLis GInGoal GOutGoal TInGoal TOutGoal :-
  copy1 TacLis GInGoal GMidGoal TInGoal TMidGoal,
  mapcopy (copy TacLis) GMidGoal GOutGoal TMidGoal TOutGoal.
copy TacLis GGoal GGoal TGoal TGoal.
```

Module `copy`: Some Basic Tactics for Proof by Analogy

We now define a general tactic that performs one step of the proof-by-analogy process. This tactic, called `copy1` takes as arguments, a list of tactics and four goals: an atomic guiding goal, an output goal for the guiding goal, an atomic target goal, and an output goal for the target goal. It is defined in module `copy`. It traverses the list of tactics until it finds one that succeeds on the guiding goal, then attempts to apply the same tactic to the second goal. The `copy` tactic in the same module simply loops calling `copy1` repeatedly, and proceeding as far as it can in using the first proof to guide the construction of the second. Like `copy1`, the `copy` tactical assumes that the guiding and target goals are atomic. It first calls `copy1` with the input arguments, and then calls `mapcopy` with `(copy TacLis)` as its argument tactic to break down compound goal structure in the intermediate goals, and then call `copy1` again with the same list of tactics on each of the atomic subgoals. Once the copying has proceeded as far as possible, the second clasue is used to halt the loop. At that point, the target goal will contain the partial proof as far as it was constructed and its output goal will contain the subgoals which must still be completed (if any) in order to complete the proof. If there are incomplete subgoals, some other means must then be used to solve them and finish the proof.

This completes the definition of the copying capabilities that will be added to the general tactic interpreter. We now present some examples of copying tactics for natural deduction. The first tactic we define, the `exactcopy` tactic in the `ndcopy` module on page 176 is a heuristic which operates by applying the same primitive tactic to both the guiding and target goals at every step. It calls `copy` with a list containing one tactic corresponding to each inference rule (except $\perp_I$). Note that it uses `and_e_tacr`, `imp_e_tacr`, `neg_e_tacr`, and `forall_e_tacr`, tactics which remove the hypothesis from the list. Thus it will not complete any proof, for example, in which a universally quantified hypothesis must be instantiated more than once. This tactic will proceed in constructing the target proof as far as it can as long as the exact rules can be applied in the target proof that were applied in the guiding proof.

The `orcopy` tactic is slightly less trivial. Its first argument is a list containing two nested `orelse` tactics. The first contains primitive nonbranching tactics, *i.e.*, tactics for rules of N_I whose output goal does not contain a conjunctive goal. The second contains branching tactics. Together, they contain all of the same rules as `exactcopy`. With this list as the argument to the `copy` tactic, as long as any rule within one nested `orelse` succeeds on the guiding proof, any rule in the same nested `orelse` structure, not necessarily the same as that applied to the guiding proof, may succeed on the target goal. The branching and nonbranching tactics are separated because, as stated above, the `copy` tactic requires that the branching structure of the guiding and target goals be the same.

```
module ndcopy.

import  copy ndtac tacticals.

type    exactcopy       goal -> goal -> goal -> goal -> o.
type    orcopy          goal -> goal -> goal -> goal -> o.

exactcopy GInGoal GOutGoal TInGoal TOutGoal :-
   copy ((close_tac 0)::and_i_tac::or_i1_tac::or_i2_tac::imp_i_tac::
         neg_i_tac::forall_i_tac::exists_i_tac::(and_e_tacr 0)::
         (imp_e_tacr 0)::(neg_e_tacr 0)::(forall_e_tacr 0)::(or_e_tac 0)::
         (exists_e_tac 0)::nil)
        GInGoal GOutGoal TInGoal TOutGoal.

orcopy GInGoal GOutGoal TInGoal TOutGoal :-
   copy ((orelse (close_tac 0) (orelse or_i1_tac (orelse or_i2_tac
          (orelse imp_i_tac (orelse neg_i_tac (orelse forall_i_tac
           (orelse exists_i_tac (orelse (and_e_tacr 0)
             (orelse (forall_e_tacr 0) (exists_e_tac 0))))))))))::
         (orelse and_i_tac (orelse (imp_e_tacr 0) (orelse (neg_e_tacr 0)
          (or_e_tac 0))))::nil)
        GInGoal GOutGoal TInGoal TOutGoal.
```

Module `ndcopy`: Tactics for Proof By Analogy in N_I

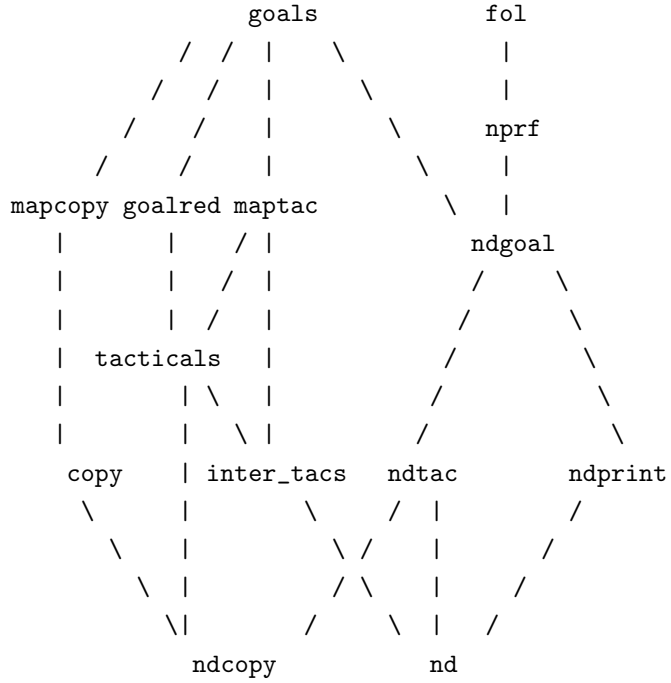

Figure 8.4: A Tactic Theorem Prover for N_I With Proof By Analogy Tactics

Figure 8.4 illustrates how the three modules we have presented can be incorporated into the tactic theorem prover for N_I. The `mapcopy` and `copy` modules can be added to the general interpreter (from `goals` to `inter_tacs` and all intermediate modules in the diagram), and thus be made available to any tactic theorem prover. The `ndcopy` module containing tactics for proof by analogy specific to N_I imports the `copy` module and `ndtac`, the tactics for natural deduction. The link from `copy` to the `lists` module is not shown here.

$$\dfrac{\forall xq(x) \vee p \qquad \dfrac{\dfrac{\dfrac{\dfrac{\forall xq(x)}{q(c)}\,\forall\text{-E}}{q(c) \vee p}\,\vee\text{-I}_1}{\forall x(q(x) \vee p)}\,\forall\text{-I} \qquad \dfrac{\dfrac{p}{q(c) \vee p}\,\vee\text{-I}_2}{\forall x(q(x) \vee p)}\,\forall\text{-I}}{\forall x(q(x) \vee p)}\,\vee\text{-E}}{(\forall xq(x) \vee p) \supset \forall x(q(x) \vee p)}\,\supset\text{-I}$$

(a)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x(q(x) \wedge p)}{q(a) \wedge p}\,\forall\text{-E}}{q(a)}\,\wedge\text{-E}_1}{\forall xq(x)}\,\forall\text{-I} \qquad \dfrac{\dfrac{\forall x(q(x) \wedge p)}{q(c) \wedge p}\,\forall\text{-E}}{p}\,\wedge\text{-E}_2}{\forall xq(x) \wedge p}\,\wedge\text{-I}}{\forall x(q(x) \wedge p) \supset (\forall xq(x) \wedge p)}\,\supset\text{-I}$$

(b)

Figure 8.5: N_I Proofs of $(\forall xq(x) \vee p) \supset \forall x(q(x) \vee p)$ and $\forall x(q(x) \wedge p) \supset (\forall xq(x) \wedge p)$

The following is an example of a query using the `orcopy` tactic.

```
orcopy (nil --> (imp_i P\ (or_e (forall X\ (q X)) p P
                          P1\ (forall_i Y\ (or_i1 (forall_e Y X\ (q X) P1)))
                          P2\ (forall_i Y\ (or_i2 P2)))) #
                (((forall X\ (q X)) or p) imp (forall X\ ((q X) or p))))
       GOutGoal
       (nil --> P2 #
                ((forall X\ ((q X) and z)) imp ((forall X\ (q X)) and z)))
       TOutGoal.
```

In this query, the proof in Figure 8.5 (a) of $(\forall xq(x) \vee p) \supset \forall x(q(x) \vee p)$ is used to guide the construction of the proof of $\forall x(q(x) \wedge p) \supset (\forall xq(x) \wedge p)$ in Figure 8.5 (b). Since the guiding and target goals have the same branching structure throughout the execution, the query will in fact successfully complete the target proof.

It is interesting to note that in a tactic theorem prover for the corresponding sequent system L_I, the minimum criterion for two sequent proofs to be considered analogous in this framework is that their tree structure be exactly the same. This is because

the branching structure of goals in such a prover corresponds exactly to the branching structure of the proofs. Thus in a query similar to the one above for natural deduction, if given a guiding proof for $\longrightarrow (\forall x q(x) \vee p) \supset \forall x(q(x) \vee p)$ a target proof for $\longrightarrow \forall x(q(x) \wedge p) \supset (\forall x q(x) \wedge p)$ could be successfully completed and would have exactly the same tree structure as the guiding proof.

Chapter 9

Conclusion and Future Work

In this dissertation, we have demonstrated the use of a higher-order logic programming language for both specifying and implementing theorem provers and other programs that manipulate formulas and proofs. We have seen that this language is quite suitable for the direct specification of various object logics and their inference systems. The data structures of this language, simply typed λ-terms, proved to be useful for expressing the higher-order abstract syntax of various object logics. Various binding operators in object logics were directly expressible as λ-abstraction, and object level substitution was naturally specified as meta-level β-conversion.

The connectives of the meta-language played an important role in the specification of inference rules. Universal quantification was crucial to the correct specification of provisos on various rules, while implication was used to naturally specify the discharge of assumptions in natural deduction style inference systems. For implementation purposes, universal quantification was exploited for rather sophisticated manipulation of λ-terms. One example that appeared repeatedly was the use of the GENERIC search operation to introduce a constant to replace a bound variable making it possible to "descend" through an abstraction in order to manipulate the body of a λ-term. Although quite useful for specification, implication on the other hand is quite limited for implementing theorem provers. In implementing a tactic theorem prover for natural deduction for first-order logic in Chapter 7, we argued that more programmer control over the manipulation of assumptions than could be provided by implication was desirable. To solve this problem, explicit context lists were adopted to store assumptions in the implementation of tactics for natural deduction.

The fact that an interpreter could be described in terms of basic search operations corresponding to the logical connectives of the language gave a direct operational reading to specifications. In general, this operational reading provided a description of goal directed search for proofs in the object logic. Unification played a central role in the description of goal directed search, in particular for instantiating inference rule schemas. We saw that

although the meta-language provided full higher-order unification, only restricted subcases were necessary in the example theorem provers that were presented. In general, for theorem proving and proof checking, second-order matching, a decidable subcase of higher-order unification, is all that is needed.

In terms of specification, one of our main claims is that our meta-language can be used to naturally specify a variety of object logics. Many of the examples provided were for first-order logics for which we were able to make use of many of the features of the meta-language directly. In more complex logics such as the higher-order object logic presented in Section 4.4, the specification was not so direct, and required an encoding of object terms and auxiliary specifications for type checking and λ-convertibility. On the other hand, the operational behavior of specifications for more complex logics is no more complex than that of programs specifying theorem provers for first-order logics: the search behavior can be described similarly, and unification problems are generally no more complex that second-order matching.

In this dissertation the same meta-language was used for both specification and implementation, although in terms of implementation, many choices had to be made in order to fully describe a deterministic interpreter for the language. Our purpose here was not to examine how to make such choices, and for the purpose of discussion we made choices similar to those in standard logic programming languages, such as depth-first control. With respect to this interpreter, for the task of proof checking, the distinction between specification and implementation could be blurred since the same programs were often able to serve as both.

For the more complicated task of theorem proving, control of execution becomes an important issue. Although we demonstrated that it is sometimes possible to modify specifications so that they are complete implementations of automatic theorem provers, our main concern has been the organization of a comprehensive proof system in which theorem proving and other manipulations on formulas and proofs could be performed. For this task, we have argued that tactic style theorem provers provide a good environment for building such a system. Several of the operational aspects of the higher-order features of our language proved to be useful in implementing an interpreter for tactics and tacticals. In particular, quantification over predicates was exploited to provide a simple implementation of the basic control mechanisms used in proof search. In addition, meta-level implication provided a notion of modules in logic programming that we were able to employ in building tactic theorem provers. We illustrated how to incorporate various capabilities such as basic search operations for a particular object logic, interactive user-guided proof search, and proof by analogy. These examples demonstrated that many object logics and formula and proof manipulation programs can be integrated in a unified framework.

One consequence of the perspicuity of the specifications for inference systems is that

they enabled us to obtain an operational description of various proof theoretic results. For example, based on a characterization of normal natural deduction proofs, we were able to specify the inference rules of natural deduction so that only normal proofs get constructed. In addition, based on the proof of normalization for natural deduction, we were able to implement a program to transform arbitrary proofs to normal ones. Also, by recognizing the similarity between specifications of a sequent system and natural deduction for first-order intuitionistic logic, we were able to combine the two to obtain a specification that both illustrates the correspondence between cut-free sequent proofs and normal natural deduction proofs, and serves as a program to transform a proof in one system to a proof in the other.

9.1 Future Work

A Specialized Meta-Language The higher-order features of our meta-language that we have used extensively, arguing that they are valuable for specification and implementation of theorem provers, are actually only a subset of the features present in the full logic programming language based on hohh. In fact with only minor modification, all of the programs in Chapters 3-6 fall within the restricted sublanguage L_λ mentioned in Chapter 4. The remaining programs are only mildly outside the scope of this language. There are a couple reasons for isolating a sublanguage such as L_λ specialized to the task of theorem proving. First, it allows us to isolate exactly which features are important for the specification and implementation of theorem provers. Second, since unification problems are greatly simplified, it may be possible to implement a specialized unification algorithm much more efficiently than full higher-order unification.

It is also possible to take the opposite approach and enrich the meta-language to increase its capabilities for specifying logics and theorem provers. The Elf language [Pfe89], for example, is an extension over hohh without predicate quantification. It is based on the LF type theory, which as we have seen, is a relatively rich language developed for the purpose of specifying a large class of logics and capturing the uniformities among them. A non-deterministic interpreter can be described for Elf in much the same way as for hohh by providing a small set of search operations which, in this case, give types an operational interpretation. A more complex unification procedure is required to implement this language [Ell89], but it appears feasible that a sublanguage similar to L_λ could be isolated for Elf. Such a language should have relatively strong specification power, but simple operational behavior and, ideally, a decidable unification procedure.

Efficient Implementation of Theorem Provers In implementing theorem provers, efficiency has not been a specific concern thus far. In fact, execution of the programs

we have presented generally is less efficient than the implementation of comparable tasks in theorem provers which use the functional programming language ML as their meta-language (*e.g.* Isabelle [Pau88]). There are other ways to address this concern besides modifying the meta-language as suggested above. For example, although depth-first search was adopted as the control mechanism for the deterministic interpreter, there may be other forms of control that are more suited to implementing theorem provers. As mentioned in Chapter 7, any of the interpreters implemented there may be implemented as the control strategy for the meta-language. Such modifications to the meta-interpreter are worth investigating for the development of proof systems with practical import. One avenue currently being explored, which would of course have impact on theorem proving applications, is the efficient implementation of λProlog [EP89, NJ89].

Adding to the Capabilities of Practical Theorem Provers There are many directions in which future work in implementing theorem provers could go, and only practical experience will tell which are the most fruitful. We have provided a basic organization for building proof systems based on tactic style theorem proving. There are many other general and specialized theorem proving techniques to be investigated and incorporated into such a system. Some examples include induction, rewrite systems, and equality reasoning. The ability to quantify over function variables suggests techniques for replacing equals by equals in an equational logic for example. For instance, if `s` and `t` are two equal terms of an object logic, a template of the form `(F s)` could be used to obtain instances of `F`, which are abstractions over instances of `s` in an object level term. The term `(F t)` is then a modification of the original term with zero or more occurrences of `s` replaced by occurrences of `t`. In such an operation, there are potentially many unifiers. Control of the generation and presentation of unifiers must be examined to evaluate the usefulness of higher-order unification for this task.

For theorem proving in a mathematical domain, it will be essential to organize libraries of definitions, theorems, and perhaps specialized strategies for proof search. The notion of modules in our logic programming language suggests a basic organization of such libraries which must be explored further. Mechanisms for accessing the information in libraries, expanding definitions, etc. must be developed and incorporated into the system. The separation of different strategies for different logics and domains into modules allows the search space to be traversed in a systematic way: it provides a means to choose which modules to access and to limit the number of available operations at any one time.

Proof By Analogy We have only scratched the surface of potential investigations of proof by analogy by demonstrating how techniques for building structurally analogous proofs can be incorporated into the tactic theorem proving setting. Although the value

of proof by analogy in human theorem proving is well-known, it is a difficult problem to incorporate into computer systems. We feel that the basic mechanisms we have presented and their incorporation into our implementation of a tactic interpreter shows promise as a starting point for investigating this strategy.

Operational Descriptions of Results in Proof Theory As mentioned, some of the specifications we have presented provide an operational description of certain well-known proof-theoretic results. It seems likely that other results might similarly be given operational reading by specifying them as logic programs. For example, in [Pot77], the correspondence between cut-elimination in sequent systems and normalization in natural deduction is described. A program that simultaneously performs both operations might provide additional insight into this correspondence. As another example, the construction of an interpolation formula from a sequent proof is defined in [Smu68]. It should be straightforward to construct such a formula from sequent proof structures described in this dissertation. In addition, such programs may suggest simpler or alternative ways of establishing the corresponding proof-theoretic results.

Operations on Proof Terms We have illustrated that it is straightforward to write programs that manipulate proof terms of various object logics. Other operations which have proven useful in other systems such as extracting programs from proofs or extracting English explanations from proofs should also be straightforward to implement in this setting. The main challenge that remains is to determine how to best integrate such operations into a theorem proving environment so that proofs, once discovered, serve as useful objects in accomplishing a variety of other tasks. Much practical experience will be needed to gain insight into this area. The examples we have shown provide evidence that the logic programming setting is a good environment for further investigation.

A λProlog Programs for Manipulating Lists

```
module lists.

type    '::'            A -> (list A) -> (list A).

type    memb            A -> (list A) -> o.
type    member          A -> (list A) -> o.
type    append          (list A) -> (list A) -> (list A) -> o.
type    nth_item        int -> A -> (list A) -> o.
type    memb_and_rest   A -> (list A) -> (list A) -> o.
type    nth_and_rest    int -> A -> (list A) -> (list A) -> o.

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

member X (X::L) :- !.
member X (Y::L) :- member X L.

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

memb_and_rest A (A::Rest) Rest.
memb_and_rest A (B::Tail) (B::Rest) :- memb_and_rest A Tail Rest.

nth_item 0 A List :- !, memb A List.
nth_item 1 A (A::Rest) :- !.
nth_item N A (B::Tail) :- M is (N - 1), nth_item M A Tail.

nth_and_rest 0 A List Rest :- !, memb_and_rest A List Rest.
nth_and_rest 1 A (A::Rest) Rest :- !.
nth_and_rest N A (B::Tail) (B::Rest) :-
  M is (N - 1), nth_and_rest M A Tail Rest.
```

Module `lists`: Some Simple Operations on Lists

B Infix Operators Used in λProlog Modules

Figure B.1 contains all of the infix symbols introduced in the modules in this dissertation. They are listed according to their order of precedence: the further down in the list, the stronger the binding power of the operator.

`=>>`			implicational goal for tactic theorem provers
`&&`	`vv`		conjunctive and disjunctive goals for tactic theorem provers
`>-`	`>-1`	`>-2`	relations between sequents and their proofs
`-->`			sequent arrow/type arrow for constructing simple types
`#`	`#e`	`#p`	relations between formulas and natural deduction proofs
`#t`			relation between a term and its type
`imp`			implication for object-level formulas
`and`	`or`		conjunction and disjunction for object-level formulas
`::`			list constructor

Figure B.1: Infix Operators Used in λProlog Modules

References

[AHM87] Arnon Avron, Furio A. Honsell, and Ian A. Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, Laboratory for the Foundations of Computer Science, University of Edinburgh, June 1987.

[And81] Peter B. Andrews. Theorem proving via general matings. *Journal of the Association for Computing Machinery*, 28:193–214, 1981.

[Bar84] Hank Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, revised edition, 1984.

[BCP86] Bishop Brock, Shaun Cooper, and William Pierce. Some experiments with analogy in proof discovery (preliminary report). Technical Report AI-347-86, MCC, Austin, Texas, October 1986.

[Ble77] W. W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9:1–35, 1977.

[Ble83] W. W. Bledsoe. The ut prover. Technical Report ATP-17B, University of Texas at Austin, April 1983.

[Ble86] W. W. Bledsoe. Some thoughts on proof discovery. In *Third Annual IEEE Symposium on Logic Programming*, pages 2–10, Salt Lake City, Utah, September 1986. MCC Tech Report AI-208-86 June 1986.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[dlTC87] Thierry Boy de la Tour and Ricardo Caferra. Proof analogy in interactive theorem proving: A method to express and use it via second order pattern matching. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 95–99, Seattle, WA, July 1987. AAAI, Morgan Kaufmann.

[Dum77] Michael Dummett. *Elements of Intuitionism*. Clarendon Press, Oxford, 1977.

[Ell89] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, pages 121–136. Springer-Verlag Lecture Notes in Computer Science, April 1989.

[EP89] Conal Elliott and Frank Pfenning. elp, a common lisp implementation of λprolog. May 1989.

[Fel87] Amy Felty. Implementing theorem provers in logic programming. Dissertation Proposal, University of Pennsylvania, Technical Report MS-CIS-87-109, November 1987.

[FM88] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pages 61–80. Springer-Verlag Lecture Notes in Computer Science, May 1988.

[FM89] Amy Felty and Dale Miller. A meta language for type checking and inference. Presented at the 1989 Workshop on Programming Logic, Bålstad, Sweden, May 1989.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.

[Gen69] Gerhard Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Goa80] Christopher Alan Goad. *Computational Uses of the Manipulation of Formal Proofs*. PhD thesis, Stanford University, August 1980.

[Gor85] Mike Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, July 1985.

[HHP89] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. Submitted to the J.ACM, 1989.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HM88] John Hannan and Dale Miller. Enriching a meta-language with higher-order features. In *Workshop on Meta-Programming in Logic Programming*, Bristol, June 1988.

[How80] William A. Howard. The formulae-as-type notion of construction, 1969. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Kle67] Stephen Cole Kleene. *Mathematical Logic*. John Wiley & Sons, Inc., 1967.

[Kor85] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Mey81] Albert Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, 1981.

[Mil83] Dale A. Miller. *Proofs in Higher-order Logic*. PhD thesis, Carnegie-Mellon University, August 1983.

[Mil88] Dale Miller. Unification under a mixed prefix. Unpublished draft, September 1988.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory Lecture Notes. BIBLIOPOLIS, Napoli, 1984.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming*, pages 379–388, September 1987.

[MN88] Dale Miller and Gopalan Nadathur. λProlog Version 2.7. Distribution in C-Prolog and Quintus sources, July 1988.

[MNPS] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. To appear in the Annals of Pure and Applied Logic.

[MNS87] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Second Annual Symposium on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.

[Nad87] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming*. PhD thesis, University of Pennsylvania, Technical Report MS-CIS-87-48, June 1987.

[NJ89] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM model for λProlog. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1180–1198, October 1989.

[NM88] Gopalan Nadathur and Dale Miller. Higher-order horn clauses. To appear in the J.ACM, April 1988.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau87] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[Pau88] Lawrence C. Paulson. The foundation of a generic theorem prover. To appear in the Journal of Automated Reasoning, March 1988.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.

[Pot77] Garrel Pottinger. Normalization as a homomorphic image of cut-elimination. *Annals of Mathematical Logic*, 12(3):223–357, 1977.

[Pra65] Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965.

[Pra71] Dag Prawitz. Ideas and results in proof theory. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 235–307. North-Holland, 1971.

[SG88] Wayne Snyder and Jean H. Gallier. Higher order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 1988. To Appear.

[Smu68] Raymond M. Smullyan. *First-Order Logic*. Springer-Verlag New York Inc., 1968.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge MA, 1986.

[ST85] Mark E. Stickel and W. Mabry Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the International Joint Conference on Artificial Intelligence 1985*, pages 1073–1075, Los Angeles, August 1985.

[Sti86] Mark E. Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler. In Jörg H. Siekmann, editor, *Eighth International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 573–587. Springer-Verlag, July 1986.

[TB79] Mabry Tyson and W. W. Bledsoe. Conflicting bindings and generalized substitutions. In *Fourth International Conference on Automated Deduction*, volume 103 of *Lecture Notes in Computer Science*, pages 14–18. Springer-Verlag, February 1979.

[Zuc74] J. I. Zucker. Cut-elimination and normalization. *Annals of Mathematical Logic*, 1(1):1–112, 1974.