REASONING IN A LOGIC WITH DEFINITIONS AND INDUCTION

RAYMOND CHARLES MCDOWELL

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.

1997

Dale Miller

Supervisor of Dissertation

Mark Steedman

Graduate Group Chairperson

In loving memory of my father and grandfathers:

James E. McDowell	*June 6, 1933 – March 11, 1992*
Gilford G. Quarles	*December 24, 1909 – April 27, 1994*
Theodore E. McDowell	*June 9, 1907 – November 21, 1996*

Acknowledgements

I appreciate the help that I have received from a number of people in the preparation of this dissertation and in the conduct of the research that it presents. First, I thank Dale Miller, my supervisor, for his encouragement and guidance on both technical and professional matters through all the twists and turns that have been taken by my graduate studies in general and this research project in particular. I have also benefited from the stimulating discussions, probing questions, and enthusiastic interest of Frank Pfenning on a number of occasions. I also appreciate Frank's invitation to visit Carnegie Mellon University in March, and for the hospitality that he, Carsten Schürmann, and Iliano Cervesato provided during that time. Catuscia Palamidessi has contributed to this work, not only in our joint work presented in Chapter 4, but also through other interactions which led, for example, to the introduction of levels to allow limited use of implication in definitions. Lars-Henrik Eriksson generously made his derivation editor Pi available to me, which has been a great help in this work. The remaining members of my dissertation committee have also been very supportive of me and my work. I especially thank Val Tannen for his encouragement and for his example to me as a scholar-teacher, Carl Gunter for the personal interest he has taken in me, and Jean Gallier for his enthusiasm and sense of humor, as well as his suggestions and pointers to the literature. I also thank Jesus Christ for the assurance he has given me, and particularly for his assistance when I faced several perplexing technical problems. Finally, I am grateful for the patience and supportiveness of Sharon McDowell, my wife, especially during the past several months when so much of my time has been committed to work on this dissertation.

This work has been funded in part by the grants ONR N00014-93-1-1324, NSF CCR-92-09224, NSF CCR-94-00907, and ARO DAAH04-95-1-0092.

Abstract

Reasoning in a Logic with Definitions and Induction

Raymond Charles McDowell
Supervisor: Dale Miller

We present a logic for the specification and analysis of deductive systems. This logic is an extension of a simple intuitionistic logic that admits higher-order quantification over simply typed λ-terms. These are key ingredients for *higher-order abstract syntax*, an elegant and declarative treatment of object-level abstraction and substitution. The logic also supports induction and a notion of *definition*. The latter concept of definition is a proof-theoretic device that allows certain theories to be treated as "closed" or as defining fixed points. We prove that cut-elimination and consistency results hold for this logic, extending a technique due to Tait and Martin-Löf. We also demonstrate the effectiveness of the logic for encoding meta-level predicates such as bisimulation and for reasoning about judgements encoded using higher-order abstract syntax. The sense of closure in definitions allows us to cleanly express the notions of simulation and bisimulation, and we derive in our logic some high-level properties about these notions in the context of abstract transition systems. Formal meta-theoretic analysis of higher-order abstract syntax encodings has been inadequately addressed in previous research. We explore the difficulties of this task by considering encodings of intuitionistic and linear logics, and formally derive the admissibility of cut for important subsets of these logics. We then propose an approach to avoid the apparent tradeoff between the benefits of higher-order abstract syntax and the ability to analyze the resulting encodings. We illustrate this approach through examples involving the simple functional and imperative programming languages PCF and $\text{PCF}_{:=}$. We formally derive such properties as unicity of typing, subject reduction, determinacy of evaluation, and the equivalence of transition semantics and natural semantics presentations of evaluation.

Contents

List of Tables

Chapter 1

Introduction

A logical framework is a formal meta-language for specifying deductive systems such as logics, operational semantics, and type systems. Typically the deduction rules for the object system are specified as an axiom system, or *theory*, of the specification logic. To give a very simple example, membership in the natural numbers might be encoded by the two formulas

$$nat\ z \qquad\qquad \forall x(nat\ x \supset nat\ (s\ x))\ .$$

We can then use this theory to derive formulas in the specification logic that indicate that various expressions denote natural numbers. Analogously, we can construct theories to specify operational semantics and type systems so that formulas denoting judgements such as "the term M denotes a program", "the program M evaluates to the value V", and "the program M has type T" can be derived.

One of the advantages of such formal specifications is that they allow logical and mathematical analyses to be used to prove properties about the specified systems. Given the specification of evaluation for a functional programming language, for example, we may wish to prove that the language is deterministic or that evaluation preserves types. As larger and more complex languages and systems are considered, it becomes desirable to provide automated support for proving these properties. To do this we must make these proofs formal as well, and it thus becomes natural to consider proving such properties within the logical framework.

In this dissertation we introduce a new logic $FO\lambda^{\Delta\mathbb{IN}}$ (pronounced "fold-n") and investigate its use for this purpose. One of the distinctive features of this logic is its notion of *definition*. Like a theory, a $FO\lambda^{\Delta\mathbb{IN}}$ definition provides a collection of formulas that are available for constructing derivations in the logic. In addition, a definition carries with it a sense of closure or completeness: if a defined proposition holds, it necessarily holds by one of the clauses of the definition. Viewing the sample theory given above as a definition, for example, adds the idea that these two formulas are the only way to establish that the predicate *nat* holds. The logic then provides a means to perform case analyses on defined concepts; thus given *nat* y as a hypothesis, we can consider the two cases $y = z$ and $y = (s\ y')$ for some y' such that *nat* y'. This notion of definition has been investigated in various proof systems in recent years [17, 19, 20, 50]. Since it provides the basis for case analysis of defined concepts, the idea has obvious benefit for a logic used to encode object systems and reason about them.

There is also a more subtle benefit for logical frameworks provided by definitions. To establish a proposition by definition, we need only show that one clause of the definition applies. To show that *nat* y holds, for example, it is sufficient to show *either* that $y = z$ *or* that $y = (s\ y')$ for some y' such that *nat* y'. This is similar to the idea of *may* behavior of a concurrent system: "there exists a definitional clause such that ..." can be used to encode "there exists a computation such that ...". On the other hand, to show something from the hypothesis *nat* y by case analysis, we need to prove that it holds *both* for $y = z$ *and* for $y = (s\ y')$, given that *nat* y'. Thus the sense of closure that definitions provide allows certain forms of *must* behavior ("all computations...") to be captured.

Several existing logical frameworks are equipped with a similar notion of definition, but are not suitable for our purposes. The Calculus of Inductive Constructions (implemented in Coq) [43] and FS_0 [30], for example, contain inductive definition facilities. However, the function type of both of these systems is too strong to naturally support the key representation technique of higher-order abstract syntax. Higher-order abstract syntax is an elegant and declarative encoding of abstraction and substitution. With most approaches to syntactic representation, the details of variable binding and substitution must be carefully addressed throughout a specification, and theorems about substitution and bound variables

can dominate the system analyses. With higher-order abstract syntax, on the other hand, these features are specified concisely and their basic properties follow immediately from the specification logic.

In contrast to Coq and FS_0, the finitary calculus of partial inductive definitions [11] does support the use of higher-order abstract syntax. However, the cut rule is not admissible for this calculus; the class of supported definitions is rich enough to allow inconsistency in the presence of cut. To see how this is possible, consider the formula $(a \supset \bot) \supset a$ as a definition for the propositional constant a. It is not hard to derive the sequent $a \longrightarrow \bot$ with this definition: by the hypothesis a holds, and a case analysis on the definition shows that if it holds, it must be because $a \supset \bot$ holds; the consequent $\bot$ then easily follows from the two hypotheses a and $a \supset \bot$. But from this derivation of $a \longrightarrow \bot$, we can derive $\longrightarrow a \supset \bot$, and thus by definition, $\longrightarrow a$. We have shown that there are derivations of $a \longrightarrow \bot$ and $\longrightarrow a$, so the cut rule would give us a derivation of $\longrightarrow \bot$. The admissibility of cut is important beyond the (fundamental) question of the consistency of the calculus; otherwise we could just use the calculus without a cut rule. Our intent is to reason within the calculus about encoded systems. Using the natural number example again, we would like the derivation of a sequent $nat\ x \longrightarrow P\,x$, where x is a variable, to represent the idea that the property encoded by P holds for all natural numbers. However, without the cut rule, we do not have any guarantee that for all i such that $\longrightarrow nat\ i$ is provable, $\longrightarrow P\,i$ is also provable. Thus we need the admissibility of the cut rule.

The logic $FO\lambda^{\Delta\mathbb{N}}$ can be viewed as a variation of the finitary calculus of partial inductive definitions in which the cut rule is admissible: the class of definitions is restricted and the form of the induction rule is modified. We include only natural number induction in $FO\lambda^{\Delta\mathbb{N}}$. This keeps the meta-theory of the logic from becoming overly complex, but is still powerful enough to allow us to derive the other induction principles we need. Because of $FO\lambda^{\Delta\mathbb{N}}$'s close relationship with the finitary calculus of partial inductive definitions, we are able to use Eriksson's derivation editor Pi [13] to construct $FO\lambda^{\Delta\mathbb{N}}$ derivations. All $FO\lambda^{\Delta\mathbb{N}}$ derivations mentioned in this dissertation have been created using Pi, which provides us with a much greater degree of confidence in their correctness than would be the case if they were constructed by hand.

1.1 Outline of the Dissertation

The dissertation is organized in the following manner. The remainder of this chapter gives an overview of higher-order abstract syntax. As has been mentioned, support for this representation technique was an important criterion in the design of $FO\lambda^{\Delta\mathbb{N}}$, and its use will pervade this dissertation.

Chapter 2 presents the logic $FO\lambda^{\Delta\mathbb{N}}$; as suggested by the title of this dissertation, its two key features are the notion of definition sketched above, and induction over natural numbers. To illustrate the use of $FO\lambda^{\Delta\mathbb{N}}$, we derive several theorems expressing properties of natural numbers and lists. In Chapter 3 we prove cut-elimination and consistency theorems for the logic. The cut-elimination proof extends a technique due to Tait and Martin-Löf to a sequent calculus setting, and uses the technical notions of normalizability and reducibility.

We proceed in the remaining chapters to explore the use of $FO\lambda^{\Delta\mathbb{N}}$ for reasoning about various deductive systems. In Chapter 4 we use the correlation between the different quantificational aspects of definitions and the computational notions of *may* and *must* behavior to express the rich notions of simulation and bisimulation for abstract transition systems. We capture the largest bisimulation relation — bisimulation equivalence — and explore its meta-theory, proving, for example, that the relation is indeed an equivalence.

The next two chapters develop an approach for formal reasoning about higher-order abstract syntax encodings. In Chapter 5 we consider encodings of intuitionistic and linear logics in $FO\lambda^{\Delta\mathbb{N}}$ to illustrate some difficulties with reasoning in the specification logic about higher-order abstract syntax and to also demonstrate some strategies to deal with these difficulties. Unfortunately these strategies involve sacrificing some benefits of higher-order abstract syntax in order to gain the ability to perform some meta-theoretic analyses. We avoid this tradeoff in Chapter 6 by taking a different approach to formal reasoning. The key to this approach is to encode the object system in a specification logic that is separate from the logic $FO\lambda^{\Delta\mathbb{N}}$ in which we perform the reasoning; this specification logic is itself specified in $FO\lambda^{\Delta\mathbb{N}}$. This separation of the specification logic and the meta-logic allows us to reason formally about specification logic sequents and their derivability, and also reflects the structure of informal reasoning about higher-order abstract syntax

encodings. We illustrate this approach by considering the static and dynamic semantics of small functional and imperative programming languages; we are able to derive in $FO\lambda^{\Delta\mathbb{N}}$ such properties as the unicity of typing, determinacy of semantics, and type preservation (subject reduction).

We conclude in Chapter 7 with a brief discussion of our accomplishments and possible extensions of this work.

1.2 Higher-Order Abstract Syntax

To set the stage for our work, we provide a brief introduction to higher-order abstract syntax; we refer the reader to [23, 46] for a more comprehensive discussion.

The key idea of higher-order abstract syntax is that variable binding in the object language is represented by λ-abstraction of the meta-language. For example, consider the simple functional language whose terms are described by the following grammar:

$$M \quad ::= \quad x \mid \lambda x.M \mid M\,M \mid \mathtt{let}\, x = M\, \mathtt{in}\, M \mid \mu x.M \ .$$

To encode these terms, we introduce a type tm and the constants

$$\begin{array}{rcllrcl} abs & : & (tm \to tm) \to tm & & let & : & tm \to (tm \to tm) \to tm \\ app & : & tm \to tm \to tm & & rec & : & (tm \to tm) \to tm \ . \end{array}$$

The following function $(\![\ ]\!)$ maps an object language term to its representation:

$$\begin{array}{rcll} (\!|x|\!) & = & x^* : tm & \text{for } x \text{ a variable} \\ (\!|\lambda x.M|\!) & = & (abs\ \lambda x^* : tm.(\!|M|\!)) & \\ (\!|M\,N|\!) & = & (app\ (\!|M|\!)\ (\!|N|\!)) & \\ (\!|\mathtt{let}\, x = M\, \mathtt{in}\, N|\!) & = & (let\ (\!|M|\!)\ \lambda x^* : tm.(\!|N|\!)) & \\ (\!|\mu x.M|\!) & = & (rec\ \lambda x^* : tm.(\!|M|\!)\ ) \ , & \end{array}$$

where $x \mapsto x^*$ defines a bijective mapping of object language variables to meta-language variables of type tm. Variables in the object language are encoded by meta-language variables, and the scope of the object language variable binding is encoded by the scope of the meta-level λ-binding. This representation takes advantage of meta-level α-equivalence to provide α-equivalence for object language terms, i.e. $(\!|M|\!) =_\alpha (\!|N|\!)$ if and only if M

and N are α-equivalent. Another benefit is that meta-level β-reduction provides capture-avoiding substitution, i.e. $(\lambda x^*.(\!|M|\!))(\!|N|\!) =_\beta (\!|M|\!)[(\!|N|\!)/x^*] = (\!|M[N/x]|\!)$.

The type inference rules for our object language will include

$$\frac{}{x : T, \Gamma \rhd x : T} \qquad \frac{y : T, \Gamma \rhd M[y/x] : U}{\Gamma \rhd \lambda x.M : T \to U} \ ,$$

where in the second rule y is a variable that does not occur in Γ or M. If we let ty be a new type used to represent object language types, $arr : ty \to ty \to ty$ a constant representing object language function types, and $typeof : tm \to ty \to o$ a predicate representing object language typing judgements, then these two rules are encoded by the following formula B:

$$\forall n(typeof\ n\ T \supset typeof\ (R\, n)\ U) \supset typeof\ (abs\ R)\ (arr\ T\ U) \ .$$

To understand this encoding, let us first examine the structure of typing derivations for abstractions. The premise of the typing rule for abstraction renames the variable bound by the abstraction to avoid name conflict. The body of the abstraction is then typed in the typing environment Γ extended with the type assignment $y : T$. In the typing derivation for the body, the typing rule for variables will be used to infer the type T for y. This can be represented schematically as

$$\frac{\begin{array}{c}\overline{y : T, \Gamma' \rhd y : T} \\ \vdots \\ y : T, \Gamma \rhd M[y/x] : U\end{array}}{\Gamma \rhd \lambda x.M : T \to U} \ .$$

In our encoding, we use assumptions in place of a typing environment. The variable bound by the abstraction is replaced by a new eigenvariable, and the result is typed under the assumption that the eigenvariable has type T. Thus the typing rule for variables is replaced by uses of typing assumptions:

$$\frac{\begin{array}{c}\overline{B, typeof\ n\ T, \Gamma' \longrightarrow typeof\ n\ T} \\ \vdots \\ B, typeof\ n\ T, \Gamma \longrightarrow typeof\ (R\, n)\ U\end{array}}{B, \Gamma \longrightarrow typeof\ (abs\ R)\ (arr\ T\ U)} \ .$$

(To simplify the presentation, we will sometimes combine the application of several rules and display it as the use of a single derived rule. We indicate this by a double line between

the premises and conclusion of the derived rule, as illustrated above.) This reading of the encoding highlights the similarities between typing derivations constructed from the inference rules and derivations of typing judgements from the formula B. However, there are also significant differences. In the derivations constructed from the encodings, the name of the variable bound by the abstraction does not appear, and in fact is irrelevant: the encoding respects the α-equivalence of terms. Also, any derivation of the sequent $\longrightarrow \mathit{typeof}\ (\mathit{abs}\ R)\ (\mathit{arr}\ T\ U)$ must have a subderivation of $\mathit{typeof}\ n\ T \longrightarrow \mathit{typeof}\ (R\, n)\ U$. Thus the cut rule allows us to conclude that for any N such that $\mathit{typeof}\ N\ T$ is derivable, $\mathit{typeof}\ (R\, N)\ U$ is derivable. This is a non-trivial property of the type inference system for the object language. This pattern of reasoning using the cut rule will prove useful as we proceed to formally derive meta-theoretic properties of inference systems.

In this brief introduction we have seen three benefits to higher-order abstract syntax: α-equivalence of object language terms is achieved cleanly via meta-language α-equivalence; capture-avoiding substitution for the object language is automatically provided by β-reduction in the meta-language; and the meta-language cut rule becomes a significant tool for meta-theoretic reasoning.

Chapter 2

The Logic $FO\lambda^{\Delta\mathbb{N}}$

In this chapter we introduce the logic which we call $FO\lambda^{\Delta\mathbb{N}}$, an acronym for "first-order logic for λ with definitions and natural numbers." We present the logic in the first section, and then proceed in the next with some sample definitions and propositions. The chapter concludes with a discussion of related work on logics with definitions.

2.1 A Description of the Logic

The basic logic is an intuitionistic version of a subset of Church's Simple Theory of Types [7] in which meta-level formulas will be given the type o. The logical connectives are $\bot$, $\top$, $\wedge$, $\vee$, $\supset$, $\forall_\tau$, and $\exists_\tau$. The quantification types τ (and thus the types of variables) are restricted to not contain o. Thus $FO\lambda^{\Delta\mathbb{N}}$ supports quantification over higher-order (non-predicate) types, a crucial feature for higher-order abstract syntax, but has a first-order proof theory, since there is no quantification over predicate types. We will use sequents of the form $\Gamma \longrightarrow B$, where Γ is a finite multiset of formulas and B is a single formula. The basic inference rules for the logic are shown in Table 2.1. In the $\forall\mathcal{R}$ and $\exists\mathcal{L}$ rules, y is an eigenvariable that is not free in the lower sequent of the rule. The multicut (mc) rule is a generalization of cut due to Slaney [53], and is used to simplify the presentation of the cut-elimination proof of Chapter 3.

We introduce the natural numbers via the constants $z : nt$ for zero and $s : nt \to nt$ for successor and the predicate $nat : nt \to o$. The right and left rules for this new predicate

Table 2.1: Inference rules for the core of $FO\lambda^{\Delta\mathbb{N}}$

$$\frac{}{\bot, \Gamma \longrightarrow B}\ \bot\mathcal{L} \qquad \frac{}{\Gamma \longrightarrow \top}\ \top\mathcal{R}$$

$$\frac{B, \Gamma \longrightarrow D}{B \wedge C, \Gamma \longrightarrow D}\ \wedge\mathcal{L} \qquad \frac{C, \Gamma \longrightarrow D}{B \wedge C, \Gamma \longrightarrow D}\ \wedge\mathcal{L} \qquad \frac{B[t/x], \Gamma \longrightarrow C}{\forall x.B, \Gamma \longrightarrow C}\ \forall\mathcal{L}$$

$$\frac{\Gamma \longrightarrow B \quad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\ \wedge\mathcal{R} \qquad \frac{\Gamma \longrightarrow B[y/x]}{\Gamma \longrightarrow \forall x.B}\ \forall\mathcal{R}$$

$$\frac{B, \Gamma \longrightarrow D \quad C, \Gamma \longrightarrow D}{B \vee C, \Gamma \longrightarrow D}\ \vee\mathcal{L} \qquad \frac{B[y/x], \Gamma \longrightarrow C}{\exists x.B, \Gamma \longrightarrow C}\ \exists\mathcal{L}$$

$$\frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow B \vee C}\ \vee\mathcal{R} \qquad \frac{\Gamma \longrightarrow C}{\Gamma \longrightarrow B \vee C}\ \vee\mathcal{R} \qquad \frac{\Gamma \longrightarrow B[t/x]}{\Gamma \longrightarrow \exists x.B}\ \exists\mathcal{R}$$

$$\frac{\Gamma \longrightarrow B \quad C, \Gamma \longrightarrow D}{B \supset C, \Gamma \longrightarrow D}\ \supset\mathcal{L} \qquad \frac{B, \Gamma \longrightarrow C}{\Gamma \longrightarrow B \supset C}\ \supset\mathcal{R}$$

$$\frac{}{A, \Gamma \longrightarrow A}\ \textit{init}, \text{ where } A \text{ is atomic} \qquad \frac{B, B, \Gamma \longrightarrow C}{B, \Gamma \longrightarrow C}\ c\mathcal{L}$$

$$\frac{\Delta_1 \longrightarrow B_1 \quad \cdots \quad \Delta_n \longrightarrow B_n \quad B_1, \ldots, B_n, \Gamma \longrightarrow C}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc, \text{ where } n \geq 0$$

are

$$\frac{}{\Gamma \longrightarrow nat\ z}\ nat\mathcal{R} \qquad \frac{\Gamma \longrightarrow nat\ I}{\Gamma \longrightarrow nat\ (s\ I)}\ nat\mathcal{R}$$

$$\frac{\longrightarrow B\,z \quad B\,j \longrightarrow B\,(s\ j) \quad B\,I, \Gamma \longrightarrow C}{nat\ I, \Gamma \longrightarrow C}\ nat\mathcal{L}\ .$$

In the left rule, the predicate $B : nt \to o$ represents the property that is proved by induction, and j is an eigenvariable that is not free in B. The third premise of that inference rule witnesses the fact that, in general, B will express a property stronger than $(\bigwedge \Gamma) \supset C$.

A *definitional clause* is written $\forall \bar{x}[p\,\bar{t} \stackrel{\triangle}{=} B]$, where p is a predicate constant, every free variable of the formula B is also free in at least one term in the list $\bar{t}$ of terms, and all variables free in $\bar{t}$ are contained in the list $\bar{x}$ of variables. Since all free variables in $p\,\bar{t}$ and B are universally quantified, we often leave these quantifiers implicit when displaying definitional clauses. The atomic formula $p\,\bar{t}$ is called the *head* of the clause, and the formula B is called the *body*. The symbol $\stackrel{\triangle}{=}$ is used simply to indicate a definitional clause: it is not a logical connective. A *definition* is a (perhaps infinite) set of definitional clauses. The same predicate may occur in the head of multiple clauses of a definition: it is best to think of a definition as a mutually recursive definition of the predicates in the heads of the clauses.

We must also restrict the use of implication in the bodies of definitional clauses; otherwise cut-elimination does not hold [49]. Toward that end we assume that each predicate symbol p in the language has associated with it a natural number $\mathrm{lvl}(p)$, the *level* of the predicate. The following definition extends the notion of level to formulas and derivations.

Definition 2.1 Given a formula B, its *level* $\mathrm{lvl}(B)$ is defined as follows:

1. $\mathrm{lvl}(p\,\bar{t}) = \mathrm{lvl}(p)$
2. $\mathrm{lvl}(\bot) = \mathrm{lvl}(\top) = 0$
3. $\mathrm{lvl}(B \wedge C) = \mathrm{lvl}(B \vee C) = \max(\mathrm{lvl}(B), \mathrm{lvl}(C))$
4. $\mathrm{lvl}(B \supset C) = \max(\mathrm{lvl}(B) + 1, \mathrm{lvl}(C))$
5. $\mathrm{lvl}(\forall x.B) = \mathrm{lvl}(\exists x.B) = \mathrm{lvl}(B)$.

Given a derivation Π of $\Gamma \longrightarrow B$, $\mathrm{lvl}(\Pi) = \mathrm{lvl}(B)$.

We now require that for every definitional clause $\forall\bar{x}[p\,\bar{t} \triangleq B]$, $\mathrm{lvl}(B) \leq \mathrm{lvl}(p\,\bar{t})$.

The logic has inference rules for defined atoms; the following relation will be useful for describing these rules.

Definition 2.2 Let the four-place relation $\mathrm{dfn}(\rho, A, \sigma, B)$ be defined to hold for the formulas A and B and the substitutions ρ and σ if there is a clause $\forall\bar{x}[A' \triangleq B]$ in the given definition such that $A\rho = A'\sigma$.

The right and left rules for defined atoms are

$$\frac{\Gamma \longrightarrow B\theta}{\Gamma \longrightarrow A}\ def\mathcal{R},\ \text{where } \mathrm{dfn}(\epsilon, A, \theta, B)$$

$$\frac{\{B\sigma, \Gamma\rho \longrightarrow C\rho \mid \mathrm{dfn}(\rho, A, \sigma, B)\}}{A, \Gamma \longrightarrow C}\ def\mathcal{L}\ ,$$

where ϵ is the empty substitution and the bound variables $\bar{x}$ in the definitional clauses are chosen to be distinct from the variables free in the lower sequent of the rule. Specifying a set of sequents as the premise should be understood to mean that each sequent in the set is a premise of the rule. The right rule corresponds to the logic programming notion of *backchaining* if we think of $\triangleq$ in definitional clauses as reverse implication. The left rule is similar to *definitional reflection* [50] (not to be confused with another notion of reflection often considered between a meta-logic and object-logic) and to an inference rule used by Girard in his note on fixed points [17]. Notice that in the $def\mathcal{L}$ rule, the free variables of the conclusion can be instantiated in the premises.

The number of premises of the $def\mathcal{L}$ rule may be zero or infinite. If the formula A does not unify with the head of any definitional clause, then the number of premises will be zero. In this case, A is an unprovable formula logically equivalent to $\bot$, and $def\mathcal{L}$ corresponds to the $\bot\mathcal{L}$ rule. If the formula A does unify with the head of a definitional clause, the number of premises will be infinite, since the domains of the substitutions ρ and σ may include variables that are not free in A and B. Any implementation of the logic will necessarily be finitary. In practice we construct only definitions with a finite number of clauses and restrict our uses of the $def\mathcal{L}$ rule to those formulas A such that for every definitional clause there is a finite, complete set of unifiers (CSU) [27] of A and the head of the clause. In this

case we can implement Eriksson's rule [11]

$$\frac{\{B\theta, \Gamma\theta \longrightarrow C\theta \mid \theta \in CSU(A, A') \text{ for some clause } \forall\bar{x}[A' \triangleq B]\}}{A, \Gamma \longrightarrow C}\ def\mathcal{L}_{CSU} \quad ,$$

where the variables $\bar{x}$ are chosen to be distinct from the variables free in the lower sequent of the rule. When the CSUs and definition are finite, this rule will have a finite number of premises.

Proposition 2.3 *The rules $def\mathcal{L}$ and $def\mathcal{L}_{CSU}$ are interadmissible, i.e. if either $def\mathcal{L}$ or $def\mathcal{L}_{CSU}$ is admissible in a logic, then the other is as well.*

Proof Given the set of derivations

$$\left\{ \begin{matrix} \Pi^{\theta,B} \\ B\theta, \Gamma\theta \longrightarrow C\theta \end{matrix} \right\}_{\theta \in CSU(A,A') \text{ for some clause } \forall\bar{x}[A' \triangleq B]} \quad ,$$

we can construct a derivation of $A, \Gamma \longrightarrow C$ using $def\mathcal{L}$ as follows. For any definitional clause $\forall\bar{x}[A' \triangleq B]$ and substitutions ρ and σ such that $A\rho = A'\sigma$, the substitution

$$\rho_\sigma(y) = \begin{cases} \sigma(y) & \text{if } y \in FV(A') \\ \rho(y) & \text{otherwise} \end{cases}$$

will be a unifier of A and A'. Thus for some $\theta \in CSU(A, A')$ there is a substitution θ' such that ρ_σ is $\theta \circ \theta'$. We can thus use $\Pi^{\theta,B}\theta'$ as the premise derivation of $B\sigma, \Gamma\rho \longrightarrow C\rho$ for $def\mathcal{L}$. (We will formally define what it means to apply a substitution to a derivation in Definition 2.5. For now it is enough to know that it yields a derivation whose endsequent is obtained by applying the substitution to the endsequent of the original derivation.)

Given the set of derivations

$$\left\{ \begin{matrix} \Pi^{\rho,\sigma,B} \\ B\sigma, \Gamma\rho \longrightarrow C\rho \end{matrix} \right\}_{\text{dfn}(\rho,A,\sigma,B)} \quad ,$$

we can construct a derivation of $A, \Gamma \longrightarrow C$ using $def\mathcal{L}_{CSU}$ as follows. For any definitional clause $\forall\bar{x}[A' \triangleq B]$ and substitution $\theta \in CSU(A, A')$, $\text{dfn}(\theta, A, \theta, B)$ holds. We can thus use $\Pi^{\theta,\theta,B}$ as the premise derivation of $B\theta, \Gamma\theta \longrightarrow C\theta$ for $def\mathcal{L}$. ∎

Observe that several of the rules of $FO\lambda^{\Delta\mathbb{N}}$ may have variables that are free in the premise but not in the conclusion: this results from the eigenvariable y of $\forall\mathcal{R}$ and $\exists\mathcal{L}$,

the term t of $\forall\mathcal{L}$ and $\exists\mathcal{R}$, the cut formulas $B_1, \ldots, B_n$ of mc, the induction predicate B of $nat\mathcal{L}$, and the substitutions ρ and σ of $def\mathcal{L}$. We view the choice of such variables as arbitrary and identify all derivations that differ only in the choice of variables that are not free in end-sequent.

Although we will show in Chapter 3 that cut-elimination holds for this logic, we do not have the subformula property since the induction predicate B used in the $nat\mathcal{L}$ rule is not necessarily a subformula of the conclusion of that inference rule. In fact, the following inference rule is derivable from the induction rule:

$$\frac{\longrightarrow B \quad B, \Gamma \longrightarrow C}{nat\ I, \Gamma \longrightarrow C} \quad .$$

This inference rule resembles the cut rule except that it requires a *nat* assumption. Although we fail to have the subformula property, the cut-elimination theorem still provides a strong basis for reasoning about proofs in $FO\lambda^{\Delta\mathbb{N}}$. Also this formulation of the induction principle is natural and close to the one used in actual mathematical practice: that is, invariants must be, at times, clever inventions that are not simply rearrangements of subformulas. Any automation of $FO\lambda^{\Delta\mathbb{N}}$ will almost certainly need to be interactive, at least for retrieving instantiations for the invariant B.

We define an ordinal measure which corresponds to the height of a derivation:

Definition 2.4 Given a derivation Π with premise derivations $\{\Pi_i\}$, the measure $\mathrm{ht}(\Pi)$ is the least upper bound of $\{\mathrm{ht}(\Pi_i) + 1\}$.

Substitutions are finite maps from variables to terms. It is common to view substitutions as maps from terms to terms by applying the substitution to all free variables of a term. We can then extend the mapping in turn to formulas and multisets by applying it to every term in a formula and every formula in a multiset. The following definition extends substitutions yet again to apply to derivations. Since we identify derivations that differ only in the choice of variables that are not free in the end-sequent, we will assume that such variables are chosen to be distinct from the variables in the domain of the substitution and from the free variables of the range of the substitution. Thus applying a substitution to a derivation will only affect the variables free in the end-sequent.

Definition 2.5 If Π is a derivation of $\Gamma \longrightarrow C$ and θ is a substitution, then we define the derivation $\Pi\theta$ of $\Gamma\theta \longrightarrow C\theta$ as follows:

1. Suppose Π ends with the $def\mathcal{L}$ rule

$$\frac{\left\{\begin{array}{c}\Pi^{\rho,\sigma,B} \\ B\sigma, \Gamma'\rho \longrightarrow C\rho\end{array}\right\}_{\mathrm{dfn}(\rho,A,\sigma,B)}}{A, \Gamma' \longrightarrow C}\ def\mathcal{L}\ .$$

   Observe that if $\mathrm{dfn}(\rho', A\theta, \sigma', B)$ then $\mathrm{dfn}(\theta \circ \rho', A, \sigma', B)$. Thus $\Pi\theta$ is

$$\frac{\left\{\begin{array}{c}\Pi^{\theta\circ\rho',\sigma',B} \\ B\sigma', \Gamma'\theta\rho' \longrightarrow C\theta\rho'\end{array}\right\}_{\mathrm{dfn}(\rho',A\theta,\sigma',B)}}{A\theta, \Gamma'\theta \longrightarrow C\theta}\ def\mathcal{L}\ .$$

2. If Π ends with any other rule and has premise derivations $\Pi_1, \ldots, \Pi_n$, then $\Pi\theta$ also ends with the same rule and has premise derivations $\Pi_1\theta, \ldots, \Pi_n\theta$.

Lemma 2.6 *For any substitution θ and derivation Π of $\Gamma \longrightarrow C$, $\Pi\theta$ is a derivation of $\Gamma\theta \longrightarrow C\theta$.*

Proof This lemma states that Definition 2.5 is well-constructed, and follows by induction on $\mu(\Pi)$. Observe that if Π ends with the $def\mathcal{R}$ rule

$$\frac{\begin{array}{c}\Pi' \\ \Gamma \longrightarrow B\sigma\end{array}}{\Gamma \longrightarrow A}\ def\mathcal{R}\ ,$$

then $\mathrm{dfn}(\epsilon, A, \sigma, B)$, and so it is also true that $\mathrm{dfn}(\epsilon, A\theta, \sigma \circ \theta, B)$. Therefore

$$\frac{\begin{array}{c}\Pi'\theta \\ \Gamma\theta \longrightarrow B\sigma\theta\end{array}}{\Gamma\theta \longrightarrow A\theta}\ def\mathcal{R}$$

is a valid derivation. ∎

Lemma 2.7 *For any derivation Π and substitution θ, $\mathrm{ht}(\Pi) \geq \mathrm{ht}(\Pi\theta)$.*

Proof The proof of this lemma is a simple induction on $\mathrm{ht}(\Pi)$. The measures may not be equal because when the derivations end with the $def\mathcal{L}$ rule, some of the premise derivations of Π may not be needed to construct the premise derivations of $\Pi\theta$. ∎

Our logic does not contain a weakening rule; instead we allow extra assumptions in the axioms. The following definition provides meta-level weakening on derivations. Since we identify derivations that differ only in the choice of variables that are not free in the end-sequent, we will assume that such variables are chosen to be distinct from the free variables of the weakening formulas.

Definition 2.8 If Π is a derivation of $\Gamma \longrightarrow C$ and Δ is a multiset of formulas, then we define the derivation $w(\Delta, \Pi)$ of $\Gamma, \Delta \longrightarrow C$ as follows:

1. If Π ends with the $def\mathcal{L}$ rule

$$\frac{\left\{\begin{array}{c}\Pi^{\rho,\sigma,B} \\ B\sigma, \Gamma'\rho \longrightarrow C\rho\end{array}\right\}}{A, \Gamma' \longrightarrow C}\ def\mathcal{L}\ ,$$

   then $w(\Delta, \Pi)$ is

$$\frac{\left\{\begin{array}{c}w(\Delta\rho, \Pi^{\rho,\sigma,B}) \\ B\sigma, \Gamma'\rho, \Delta\rho \longrightarrow C\rho\end{array}\right\}}{A, \Gamma', \Delta \longrightarrow C}\ def\mathcal{L}\ .$$

2. If Π ends with the $nat\mathcal{L}$ rule

$$\frac{\begin{array}{ccc}\Pi_1 & \Pi_2 & \Pi_3 \\ \longrightarrow B\,z & B\,j \longrightarrow B\,(s\,j) & B\,I, \Gamma \longrightarrow C\end{array}}{nat\ I, \Gamma \longrightarrow C}\ nat\mathcal{L}\ ,$$

   then $w(\Delta, \Pi)$ is

$$\frac{\begin{array}{ccc}\Pi_1 & \Pi_2 & w(\Delta, \Pi_3) \\ \longrightarrow B\,z & B\,j \longrightarrow B\,(s\,j) & B\,I, \Gamma, \Delta \longrightarrow C\end{array}}{nat\ I, \Gamma, \Delta \longrightarrow C}\ nat\mathcal{L}\ .$$

3. If Π ends with the mc rule

$$\frac{\begin{array}{ccccc}\Pi_1 & & \Pi_n & & \Pi' \\ \Delta_1 \longrightarrow B_1 & \cdots & \Delta_n \longrightarrow B_n & & B_1, \ldots, B_n, \Gamma \longrightarrow C\end{array}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc\ ,$$

   then $w(\Delta, \Pi)$ is

$$\frac{\begin{array}{ccccc}\Pi_1 & & \Pi_n & & w(\Delta, \Pi') \\ \Delta_1 \longrightarrow B_1 & \cdots & \Delta_n \longrightarrow B_n & & B_1, \ldots, B_n, \Gamma, \Delta \longrightarrow C\end{array}}{\Delta_1, \ldots, \Delta_n, \Gamma, \Delta \longrightarrow C}\ mc\ .$$

4. If Π ends with any other rule and has premise derivations $\Pi_1, \ldots, \Pi_n$, then $w(\Delta, \Pi)$ also ends with the same rule and has premise derivations $w(\Delta, \Pi_1), \ldots, w(\Delta, \Pi_n)$.

Lemma 2.9 *For any multiset Δ of formulas and derivation Π of $\Gamma \longrightarrow C$, $w(\Delta, \Pi)$ is a derivation of $\Gamma, \Delta \longrightarrow C$.*

Proof This lemma states that Definition 2.8 is well-constructed, and follows by a simple induction on $\mu(\Pi)$. ∎

Lemma 2.10 *For any derivation Π and multiset Δ of formulas,*

$$\mathrm{ht}(\Pi) = \mathrm{ht}(w(\Delta, \Pi)) \ .$$

Proof The proof of this lemma goes by induction on $\mu(\Pi)$. All cases follow immediately from the induction hypothesis. ∎

Lemma 2.11 *For any derivation Π, multiset Δ of formulas, and substitution θ, $w(\Delta, \Pi)\theta$ and $w(\Delta\theta, \Pi\theta)$ are the same derivation.*

Proof We prove this lemma by induction on $\mathrm{ht}(\Pi)$. The only case that is not straightforward is the one in which Π ends with $def\,\mathcal{L}$:

If Π is

$$\frac{\left\{\begin{array}{c}\Pi^{\rho,\sigma,B} \\ B\sigma, \Gamma\rho \longrightarrow C\rho\end{array}\right\}}{A, \Gamma \longrightarrow C}\ def\,\mathcal{L}\ ,$$

then $w(\Delta, \Pi)$ is

$$\frac{\left\{\begin{array}{c}w(\Delta\rho, \Pi^{\rho,\sigma,B}) \\ B\sigma, \Gamma\rho, \Delta\rho \longrightarrow C\rho\end{array}\right\}_{\mathrm{dfn}(\rho,A,\sigma,B)}}{A, \Gamma, \Delta \longrightarrow C}\ def\,\mathcal{L}$$

and $w(\Delta, \Pi)\theta$ is

$$\frac{\left\{\begin{array}{c}w(\Delta\theta\rho', \Pi^{\theta\circ\rho',\sigma',B}) \\ B\sigma', \Gamma\theta\rho', \Delta\theta\rho' \longrightarrow C\theta\rho'\end{array}\right\}_{\mathrm{dfn}(\rho',A\theta,\sigma',B)}}{A\theta, \Gamma\theta, \Delta\theta \longrightarrow C\theta}\ def\,\mathcal{L}\ .$$

On the other hand, $\Pi\theta$ is

$$\frac{\left\{\begin{array}{c}\Pi^{\theta\circ\rho'',\sigma'',B} \\ B\sigma'', \Gamma\theta\rho'' \longrightarrow C\theta\rho''\end{array}\right\}_{\mathrm{dfn}(\rho'',A\theta,\sigma'',B)}}{A\theta, \Gamma\theta \longrightarrow C\theta}\ def\,\mathcal{L}$$

and $w(\Delta\theta, \Pi\theta)$ is

$$\frac{\left\{\begin{array}{c} w(\Delta\theta\rho'', \Pi^{\theta\circ\rho'',\sigma'',B}) \\ B\sigma'', \Gamma\theta\rho'', \Delta\theta\rho'' \longrightarrow C\theta\rho'' \end{array}\right\}_{\mathrm{dfn}(\rho'', A\theta, \sigma'', B)}}{A\theta, \Gamma\theta, \Delta\theta \longrightarrow C\theta}\ def\mathcal{L} \quad .$$

Both $w(\Delta, \Pi)\theta$ and $w(\Delta\theta, \Pi\theta)$ have a premise derivation for each combination of formula B and substitutions ρ and σ such that $\mathrm{dfn}(\rho, A\theta, \sigma, B)$, and in both cases that premise will be $w(\Delta\theta\rho, \Pi^{\theta\circ\rho,\sigma,B})$. Thus these two derivations are the same. ∎

Lemma 2.12 *For any derivation* Π *and multisets* Δ *and* Δ' *of formulas,*

$$w(\Delta, w(\Delta', \Pi)) \quad \textit{and} \quad w(\Delta \cup \Delta', \Pi)$$

are the same derivation.

Proof The proof of this lemma goes by induction on $\mu(\Pi)$. All cases follow immediately from the induction hypothesis. ∎

2.2 Some Simple Definitions and Propositions

In this section we illustrate the use of the logic $FO\lambda^{\Delta\mathbb{N}}$ with some examples. We first define some predicates over the natural numbers and reason about them. Then we introduce a list type and consider predicates for it. As we prove properties about these types and predicates, we will interleave informal descriptions of the proofs with their realization as derivations in $FO\lambda^{\Delta\mathbb{N}}$. The formal derivations are by nature detailed and low-level, breaking down proof principles into small pieces. As a result, what can seem obvious or be described informally in a small number of words may take a number of steps to accomplish in the formal derivation. But it is exactly this nature that makes formal derivations amenable to automation; tools such as proof editors and theorem provers can make the construction of formal derivations more natural as well as more robust.

We will describe derivations in a "bottom-up" manner – that is, we will start with the sequent we wish to derive, apply a rule with that sequent as the conclusion, and continue in this manner with the rule premises. Thus unproved premises represent statements of

Table 2.2: Definitional clauses for predicates over natural numbers

$I = I$	$\triangleq$	$\top$	$sum\ z\ J\ J$	$\triangleq$	$nat\ J$
			$sum\ (s\ I)\ J\ (s\ K)$	$\triangleq$	$sum\ I\ J\ K$
$z < (s\ J)$	$\triangleq$	$nat\ J$	$I \leq I$	$\triangleq$	$\top$
$(s\ I) < (s\ J)$	$\triangleq$	$I < J$	$I \leq J$	$\triangleq$	$I < J$

what remains to be proved to establish the original sequent. Since the formal ($FO\lambda^{\Delta\mathbb{N}}$) derivation is presented in pieces, intermixed with descriptive text, pieces that occur later in the text will generally be (partial) derivations of unproved premises from earlier pieces.

2.2.1 Natural Numbers

As described in Section 2.1, $FO\lambda^{\Delta\mathbb{N}}$ includes a type nt encoding natural numbers and a membership predicate nat. We now introduce predicates representing equality, the less-than relation, the less-than-or-equal-to relation, and the addition function. The types for these predicates are as follows:

$$= \ : \ nt \to nt \to o \qquad sum \ : \ nt \to nt \to nt \to o$$
$$< \ : \ nt \to nt \to o \qquad \leq \ : \ nt \to nt \to o \quad .$$

The definitional clauses for these predicates are shown in Table 2.2; we shall refer to this set of clauses as $\mathcal{D}(nat)$. We define two numbers to be equal if they are unifiable. The clauses for sum indicate that the sum of zero and any other number J is J, and the sum of $(s\ I)$ and J is the successor of the sum of I and J. Zero is less than the successor of any number, and $(s\ I)$ is less than $(s\ J)$ whenever I is less than J. Finally, $I \leq J$ if I is equal to J or if I is less than J.

We now proceed to reason in the logic $FO\lambda^{\Delta\mathbb{N}}$ about natural numbers and these predicates over them. As our first example, we derive a case analysis rule for natural numbers. In general the $def\mathcal{L}$ rule is used to formalize case analysis, but the predicate nat is not a defined predicate, and so the $def\mathcal{L}$ rule does not apply in the case of natural numbers.

However, a case analysis may be viewed as an induction in which we do not use the induction hypothesis in the induction step. Thus we can derive a case analysis rule for natural numbers from the induction ($nat\mathcal{L}$) rule.

Proposition 2.13 *For any formula $C : o$, predicate $B : nt \to o$, term $I : nt$, multiset Γ of formulas, and eigenvariable $i : nt$ such that i is not free in B, the following rule is derivable in $FO\lambda^{\Delta\mathbb{N}}$:*

$$\frac{\longrightarrow B\,z \quad nat\;i \longrightarrow B\,(s\;i) \quad B\,I, \Gamma \longrightarrow C}{nat\;I, \Gamma \longrightarrow C}\ .$$

Proof This rule expresses the following idea: we want to show that C follows from Γ and the fact that I is a natural number. Since I is a natural number, it must be either zero or the successor of another natural number. Thus if we can show that B holds for zero and for the successor of any natural number (the first two premises), then we know that B holds for I. It then remains to show that C follows from $B\,I$ and Γ (the third premise).

To derive this rule, we assume that we have derivations of the premises and proceed to prove the conclusion. That is, we construct in $FO\lambda^{\Delta\mathbb{N}}$ a partial derivation of the sequent $nat\;I, \Gamma \longrightarrow C$, leaving unproved premises of the form $\longrightarrow B\,z$, $nat\;i \longrightarrow B\,(s\;i)$, and $B\,I, \Gamma \longrightarrow C$. This corresponds to working under the assumption that B holds both for zero and for the successor of any number and that $B\,I$ and Γ imply C. We proceed by induction on I, using $(\lambda i.nat\;i \wedge B\,i)$ as our induction predicate. As a result, we must establish three things:

1. the base case: zero is a natural number and B holds for it;

2. the induction step: if i is a natural number and B holds for it, then the same is true for $(s\;i)$;

3. the relevance of the induction predicate: if I is a natural number and B holds for it, then Γ implies C.

This staging of the problem is representing in $FO\lambda^{\Delta\mathbb{N}}$ by applying the $nat\mathcal{L}$ rule:

$$\frac{\longrightarrow nat\;z \wedge B\,z \quad nat\;i \wedge B\,i \longrightarrow nat\;(s\;i) \wedge B\,(s\;i) \quad nat\;I \wedge B\,I, \Gamma \longrightarrow C}{nat\;I, \Gamma \longrightarrow C}\;nat\mathcal{L}\ .$$

The three premises to the $nat\mathcal{L}$ rule correspond to the three proof obligations enumerated above.

Let us first consider the relevance of the induction predicate. This is clear, since we are working under the assumption that C follows from $B\,I$ and Γ. This is formally represented by the partial derivation

$$\frac{B\,I, \Gamma \longrightarrow C}{nat\;I \wedge B\,I, \Gamma \longrightarrow C}\;\wedge\mathcal{L}\;.$$

The base case is also simple: zero is obviously a natural number, and we are working under the assumption that B holds for zero. This is expressed in $FO\lambda^{\Delta\mathbb{IN}}$ by the partial derivation

$$\frac{\dfrac{}{\longrightarrow nat\;z}\;nat\mathcal{R} \qquad \longrightarrow B\,z}{\longrightarrow nat\;z \wedge B\,z}\;\wedge\mathcal{R}\;.$$

It remains to prove the induction step. Since i is a natural number, $(s\;i)$ is as well. In addition, B holds for $(s\;i)$ by our working assumption. The formal representation of this reasoning is

$$\frac{\dfrac{\dfrac{}{nat\;i \longrightarrow nat\;i}\;init}{nat\;i \longrightarrow nat\;(s\;i)}\;nat\mathcal{R} \qquad nat\;i \longrightarrow B\,(s\;i)}{\dfrac{nat\;i \longrightarrow nat\;(s\;i) \wedge B\,(s\;i)}{nat\;i \wedge B\,i \longrightarrow nat\;(s\;i) \wedge B\,(s\;i)}\;\wedge\mathcal{L}}\;\wedge\mathcal{R}\;.$$

■

We now use this derived case analysis rule to prove that zero is the smallest natural number.

Proposition 2.14 *The formula $\forall i(nat\;i \supset z \leq i)$ is derivable in $FO\lambda^{\Delta\mathbb{IN}}$ using the definition $\mathcal{D}(nat)$.*

Proof The proof is a simple case analysis on i. To represent this in $FO\lambda^{\Delta\mathbb{IN}}$, we apply the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules to get

$$nat\;i \longrightarrow z \leq i\;,$$

and then use the derived rule of Proposition 2.13, which yields the three sequents

$$\longrightarrow z \leq z \qquad nat\;i' \longrightarrow z \leq (s\;i') \qquad z \leq i \longrightarrow z \leq i\;.$$

In this case, the third premise is immediate:

$$\frac{}{z \leq i \longrightarrow z \leq i}\;init\;.$$

If i is zero, then it is immediate that zero is equal to itself and thus less than or equal to itself:

$$\dfrac{\dfrac{}{\longrightarrow \top}\,\top\mathcal{R}}{\longrightarrow z \leq z}\,def\mathcal{R}\ .$$

If i is the successor of some number i', then $z < (s\ i')$ by definition, and so $z \leq (s\ i')$ also by definition. This is represented formally by the derivation

$$\dfrac{\dfrac{\dfrac{}{nat\ i' \longrightarrow nat\ i'}\,init}{nat\ i' \longrightarrow z < (s\ i')}\,def\mathcal{R}}{nat\ i' \longrightarrow z \leq (s\ i')}\,def\mathcal{R}\ . \qquad \blacksquare$$

We will now prove a more complicated property, namely that if I is less than $(s\ J)$, then $I \leq J$.

Proposition 2.15 *The formula*

$$\forall i(nat\ i \supset \forall j(i < (s\ j) \supset i \leq j))$$

is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition $\mathcal{D}(nat)$.

Proof To prove this we prove the stronger assertion that for any k greater than i, k is the successor of a number greater than or equal to i, i.e.

$$\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k'))\ .$$

We will prove this by induction on i, so we must prove

1. the base case: the induction predicate holds for zero;
2. the induction step: if the induction predicate is true of i', then it is true of $(s\ i')$;
3. the relevance of the induction predicate: the induction predicate applied to i implies the original proposition.

This staging of the problem is represented in $FO\lambda^{\Delta\mathbb{N}}$ by applying the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules to obtain the sequent

$$nat\ i \longrightarrow \forall j(i < (s\ j) \supset i \leq j)$$

and then the $nat\mathcal{L}$ rule to get the three sequents

$$\longrightarrow \forall k(z < k \supset \exists k'(k = (s\ k') \wedge z \leq k'))$$

$$\forall k(i' < k \supset \exists k'(k = (s\ k') \wedge i' \leq k')) \longrightarrow \forall k((s\ i') < k \supset \exists k'(k = (s\ k') \wedge (s\ i') \leq k'))$$

$$\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')) \longrightarrow \forall j(i < (s\ j) \supset i \leq j)\ .$$

First consider the relevance of the induction predicate. By assumption $(s\ j)$ is greater than i, so our induction predicate implies that $(s\ j)$ is the successor of a number greater than or equal to i. This is represented in $FO\lambda^{\Delta\mathbb{N}}$ by the partial derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{i < (s\ j) \longrightarrow i < (s\ j)}\ init \quad \exists k'((s\ j) = (s\ k') \wedge i \leq k'), i < (s\ j) \longrightarrow i \leq j}{i < (s\ j) \supset \exists k'((s\ j) = (s\ k') \wedge i \leq k'), i < (s\ j) \longrightarrow i \leq j}\ \supset \mathcal{L}}{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')), i < (s\ j) \longrightarrow i \leq j}\ \forall \mathcal{L}}{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')) \longrightarrow i < (s\ j) \supset i \leq j}\ \supset \mathcal{R}}{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')) \longrightarrow \forall j(i < (s\ j) \supset i \leq j)}\ \forall \mathcal{R}\ .$$

Of course, $(s\ j)$ is only the successor of j, so j must be greater than or equal to i:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\top, i \leq j, i < (s\ j) \longrightarrow i \leq j}\ init}{(s\ j) = (s\ k'), i \leq k', i < (s\ j) \longrightarrow i \leq j}\ def\mathcal{L}}{(s\ j) = (s\ k'), (s\ j) = (s\ k') \wedge i \leq k', i < (s\ j) \longrightarrow i \leq j}\ \wedge \mathcal{L}}{(s\ j) = (s\ k') \wedge i \leq k', (s\ j) = (s\ k') \wedge i \leq k', i < (s\ j) \longrightarrow i \leq j}\ \wedge \mathcal{L}}{(s\ j) = (s\ k') \wedge i \leq k', i < (s\ j) \longrightarrow i \leq j}\ c\mathcal{L}}{\exists k'((s\ j) = (s\ k') \wedge i \leq k'), i < (s\ j) \longrightarrow i \leq j}\ \exists \mathcal{L}\ .$$

To prove the base case of the induction, we must realize that if k is greater than zero, it must be of the form $(s\ k_0)$ for some number k_0. This realization is accomplished formally as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{nat\ k_0 \longrightarrow \exists k'((s\ k_0) = (s\ k') \wedge z \leq k')}{z < k \longrightarrow \exists k'(k = (s\ k') \wedge z \leq k')}\ def\mathcal{L}}{\longrightarrow z < k \supset \exists k'(k = (s\ k') \wedge z \leq k')}\ \supset \mathcal{R}}{\longrightarrow \forall k(z < k \supset \exists k'(k = (s\ k') \wedge z \leq k'))}\ \forall \mathcal{R}\ .$$

It is now clear that k' should be k_0 and it only remains to prove that k_0 is greater than or equal to zero:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{nat\ k_0 \longrightarrow \top}\ \top \mathcal{R}}{nat\ k_0 \longrightarrow (s\ k_0) = (s\ k_0)}\ def\mathcal{R} \quad nat\ k_0 \longrightarrow z \leq k_0}{nat\ k_0 \longrightarrow (s\ k_0) = (s\ k_0) \wedge z \leq k_0}\ \wedge \mathcal{R}}{nat\ k_0 \longrightarrow \exists k'((s\ k_0) = (s\ k') \wedge z \leq k')}\ \exists \mathcal{R}\ .$$

But any natural number is greater than or equal to zero, as was shown formally above in Proposition 2.14.

It now remains to prove the induction step, represented in $FO\lambda^{\Delta\mathbb{N}}$ by the second premise of the $def\mathcal{L}$ rule:

$$\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')) \longrightarrow \forall k((s\ i) < k \supset \exists k'(k = (s\ k') \wedge (s\ i) \leq k'))\ .$$

Since $(s\ i) < k$, by definition k must be of the form $(s\ k_0)$ for some k_0 such that $i < k_0$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')), i < k_0 \longrightarrow \exists k'((s\ k_0) = (s\ k') \wedge (s\ i) \leq k')}{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')), (s\ i) < k \longrightarrow \exists k'(k = (s\ k') \wedge (s\ i) \leq k')}\ def\mathcal{L}}{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')) \longrightarrow (s\ i) < k \supset \exists k'(k = (s\ k') \wedge (s\ i) \leq k')}\ \supset\mathcal{R}}{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')) \longrightarrow \forall k((s\ i) < k \supset \exists k'(k = (s\ k') \wedge (s\ i) \leq k'))}\ \forall\mathcal{R}\ .$$

It is now clear that k' should be k_0, so we proceed as in the base case:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall k \ldots, i < k_0 \longrightarrow \top}\ \top\mathcal{R}}{\forall k \ldots, i < k_0 \longrightarrow (s\ k_0) = (s\ k_0)}\ def\mathcal{R} \quad \forall k \ldots, i < k_0 \longrightarrow (s\ i) \leq k_0}{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')), i < k_0 \longrightarrow (s\ k_0) = (s\ k_0) \wedge (s\ i) \leq k_0}\ \wedge\mathcal{R}}{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')), i < k_0 \longrightarrow \exists k'((s\ k_0) = (s\ k') \wedge (s\ i) \leq k')}\ \exists\mathcal{R}\ .$$

Since $i < k_0$, by the induction hypothesis k_0 is the successor of a number k_0' that is greater than or equal to i:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{i < k_0 \longrightarrow i < k_0}\ init \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\top, i \leq k_0', i < (s\ k_0') \longrightarrow (s\ i) \leq (s\ k_0')}{k_0 = (s\ k_0'), i \leq k_0', i < k_0 \longrightarrow (s\ i) \leq k_0}\ def\mathcal{L}}{\ldots \wedge \ldots, \ldots \wedge \ldots, i < k_0 \longrightarrow (s\ i) \leq k_0}\ \wedge\mathcal{L}}{k_0 = (s\ k_0') \wedge i \leq k_0', i < k_0 \longrightarrow (s\ i) \leq k_0}\ c\mathcal{L}}{\exists k'(k_0 = (s\ k') \wedge i \leq k'), i < k_0 \longrightarrow (s\ i) \leq k_0}\ \exists\mathcal{L}}{i < k_0 \supset \exists k'(k_0 = (s\ k') \wedge i \leq k'), i < k_0 \longrightarrow (s\ i) \leq k_0}\ \supset\mathcal{L}}{\forall k(i < k \supset \exists k'(k = (s\ k') \wedge i \leq k')), i < k_0 \longrightarrow (s\ i) \leq k_0}\ \forall\mathcal{L}\ .$$

Since $i \leq k_0'$, either $i = k_0'$ or $i < k_0'$. If $i = k_0'$, then $(s\ i) = (s\ k_0')$. If $i < k_0'$, then $(s\ i) < (s\ k_0')$. In either case, $(s\ i) \leq (s\ k_0')$ by definition. This reasoning is represented formally by the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\top, \top, i < (s\ i) \longrightarrow \top}\ \top\mathcal{R}}{\top, \top, i < (s\ i) \longrightarrow (s\ i) \leq (s\ i)}\ def\mathcal{R} \quad \dfrac{\dfrac{\dfrac{}{\top, i < k_0', i < (s\ k_0') \longrightarrow i < k_0'}\ init}{\top, i < k_0', i < (s\ k_0') \longrightarrow (s\ i) < (s\ k_0')}\ def\mathcal{R} \quad \dfrac{\top, i < k_0', i < (s\ k_0') \longrightarrow (s\ i) \leq (s\ k_0')}{\top, i < k_0', i < (s\ k_0') \longrightarrow (s\ i) \leq (s\ k_0')}\ def\mathcal{R}}{\top, i \leq k_0', i < (s\ k_0') \longrightarrow (s\ i) \leq (s\ k_0')}\ def\mathcal{L}\ .$$ ∎

As a final example for the predicates over natural numbers, we will derive a rule for complete induction.

Proposition 2.16 *For any formula $C : o$, predicate $B : nt \to o$, term $I : nt$, and multiset Γ of formulas, the following rule is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition $\mathcal{D}(nat)$:*

$$\frac{nat\ j, \forall k(nat\ k \supset k < j \supset B\ k) \longrightarrow B\ j \quad B\ I, \Gamma \longrightarrow C}{nat\ I, \Gamma \longrightarrow C}\ .$$

Proof To derive this rule, we construct a partial derivation of the sequent $nat\ I, \Gamma \longrightarrow C$, leaving unproved premises of the form $nat\ j, \forall k(nat\ k \supset k < j \supset B\ k) \longrightarrow B\ j$ and $B\ I, \Gamma \longrightarrow C$. This corresponds to proving that C follows from Γ and the fact that I is a natural number under the assumptions

- for any natural number j, if B holds for all numbers less than k, then B holds for j; and
- $B\ I$ and Γ imply C.

The idea behind the proof is as follows. We want to show that the first working assumption is sufficient to imply that $B\ I$ holds, and thus (by the second working assumption) that C holds. We do this by induction on I, of course, but instead of using B itself for the induction predicate, we use $(\lambda i.nat\ i \wedge \forall k(nat\ k \supset k < i \supset B\ k))$. If this is true of I, then we can combine it with the first working assumption to infer that $B\ I$ holds. This is all represented formally as follows. We proceed to derive the sequent $nat\ I, \Gamma \longrightarrow C$ by induction on I (via the $nat\mathcal{L}$ rule), using $(\lambda i.nat\ i \wedge \forall k(nat\ k \supset k < i \supset B\ k))$ as the induction predicate. This yields the three sequents

$$\longrightarrow nat\ z \wedge \forall k(nat\ k \supset k < z \supset B\ k)$$

$$nat\ j \wedge \forall k(nat\ k \supset k < j \supset B\ k) \longrightarrow nat\ (s\ j) \wedge \forall k(nat\ k \supset k < (s\ j) \supset B\ k)$$

$$nat\ I \wedge \forall k(nat\ k \supset k < I \supset B\ k), \Gamma \longrightarrow C\ .$$

The third premise of the $nat\mathcal{L}$ rule indicates that the induction predicate applied to I implies the conclusion. As described above, this follows from the two premises of the complete induction rule we are deriving. The formalization of this is a cut between the partial derivation

$$\dfrac{\dfrac{\dfrac{nat\ I, \forall k(nat\ k \supset k < I \supset B\ k) \longrightarrow B\ I}{nat\ I, nat\ I \wedge \forall k(nat\ k \supset k < I \supset B\ k) \longrightarrow B\ I}\ \wedge\mathcal{L}}{nat\ I \wedge \forall k(nat\ k \supset k < I \supset B\ k), nat\ I \wedge \forall k(nat\ k \supset k < I \supset B\ k) \longrightarrow B\ I}\ \wedge\mathcal{L}}{nat\ I \wedge \forall k(nat\ k \supset k < I \supset B\ k) \longrightarrow B\ I}\ c\mathcal{L}$$

and the second premise of the rule we are deriving.

The base case of the induction is represented by the first premise of the $nat\mathcal{L}$ rule

$$\longrightarrow nat\ z \wedge \forall k(nat\ k \supset k < z \supset B\,k)\ .$$

The first part, that zero is a natural number, is obvious, and the second part holds vacuously, since there are no natural numbers less than zero. The formal representation of this case is the derivation

$$\dfrac{\dfrac{}{\longrightarrow nat\ z}\ nat\mathcal{R} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{nat\ k, k < z \longrightarrow B\,k}\ def\mathcal{L}}{nat\ k \longrightarrow k < z \supset B\,k}\supset\mathcal{R}}{\longrightarrow nat\ k \supset k < z \supset B\,k}\supset\mathcal{R}}{\longrightarrow \forall k(nat\ k \supset k < z \supset B\,k)}\forall\mathcal{R}}{\longrightarrow nat\ z \wedge \forall k(nat\ k \supset k < z \supset B\,k)}\wedge\mathcal{R}\ .$$

It remains to derive the second premise to the $nat\mathcal{L}$ rule,

$$nat\ j \wedge \forall k(nat\ k \supset k < j \supset B\,k) \longrightarrow nat\ (s\ j) \wedge \forall k(nat\ k \supset k < (s\ j) \supset B\,k)\ ,$$

which corresponds to the induction step. Thus given that j is a natural number and that B holds for all numbers less than j, we must show that the same are true for $(s\ j)$. Formally, we apply the $\wedge\mathcal{R}$ rule to obtain the two sequents

$$nat\ j \wedge \forall k(nat\ k \supset k < j \supset B\,k) \longrightarrow nat\ (s\ j)$$

$$nat\ j \wedge \forall k(nat\ k \supset k < j \supset B\,k) \longrightarrow \forall k(nat\ k \supset k < (s\ j) \supset B\,k)\ .$$

The first of these, that $(s\ j)$ is a natural number, follows immediately from the assumption that j is a natural number:

$$\dfrac{\dfrac{\dfrac{}{nat\ j \longrightarrow nat\ j}\ init}{nat\ j \longrightarrow nat\ (s\ j)}\ def\mathcal{R}}{nat\ j \wedge \forall k(nat\ k \supset k < j \supset B\,k) \longrightarrow nat\ (s\ j)}\wedge\mathcal{L}\ .$$

To derive the second sequent, we must show that $B\,k$ holds for all numbers $k < (s\ j)$. To

do this, we first deduce $k \leq j$ from $k < (s\ j)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{nat\ k, k < (s\ j) \longrightarrow k \leq j \quad \forall k(nat\ k \supset k < j \supset B\ k), nat\ k, k \leq j \longrightarrow B\ k}{\forall k(nat\ k \supset k < j \supset B\ k), nat\ k, nat\ k, k < (s\ j) \longrightarrow B\ k}\ mc}{\forall k(nat\ k \supset k < j \supset B\ k), nat\ k, k < (s\ j) \longrightarrow B\ k}\ c\mathcal{L}}{\forall k(nat\ k \supset k < j \supset B\ k), nat\ k \longrightarrow k < (s\ j) \supset B\ k}\ \supset \mathcal{R}}{\forall k(nat\ k \supset k < j \supset B\ k) \longrightarrow nat\ k \supset k < (s\ j) \supset B\ k}\ \supset \mathcal{R}}{\forall k(nat\ k \supset k < j \supset B\ k) \longrightarrow \forall k(nat\ k \supset k < (s\ j) \supset B\ k)}\ \forall \mathcal{R}}{nat\ j \wedge \forall k(nat\ k \supset k < j \supset B\ k) \longrightarrow \forall k(nat\ k \supset k < (s\ j) \supset B\ k)}\ \wedge \mathcal{L}\ .$$

The first premise to the multicut rule is derivable according to Proposition 2.15, so we proceed with the second. Since $k \leq j$, either $k = j$ or $k < j$, and we consider these cases separately. We accomplish this formally by applying the $def\mathcal{L}$ rule to $k \leq j$, yielding the two sequents

$$\forall k(nat\ k \supset k < j \supset B\ k), nat\ j, \top \longrightarrow B\ j$$

$$\forall k(nat\ k \supset k < j \supset B\ k), nat\ k, k < j \longrightarrow B\ k\ .$$

The first sequent above, corresponding to the case when $k = j$, is a weakening of the first premise of the induction rule we are deriving. We do not have an explicit weakening rule in $FO\lambda^{\Delta\mathbb{N}}$, but it suffices here to use the cut rule:

$$\dfrac{\dfrac{}{nat\ j, \top \longrightarrow nat\ j}\ init \quad \forall k(nat\ k \supset k < j \supset B\ k), nat\ j \longrightarrow B\ j}{\forall k(nat\ k \supset k < j \supset B\ k), nat\ j, \top \longrightarrow B\ j}\ mc\ .$$

When $k < j$, $B\ k$ follows from the induction hypothesis. This is formalized as

$$\dfrac{\dfrac{\dfrac{}{nat\ k, k < j \longrightarrow nat\ k}\ init \quad \dfrac{\dfrac{}{nat\ k, k < j \longrightarrow k < j}\ init \quad B\ k, nat\ k, k < j \longrightarrow B\ k}{k < j \supset B\ k, nat\ k, k < j \longrightarrow B\ k}\ \supset \mathcal{L}}{nat\ k \supset k < j \supset B\ k, nat\ k, k < j \longrightarrow B\ k}\ \supset \mathcal{L}}{\forall k(nat\ k \supset k < j \supset B\ k), nat\ k, k < j \longrightarrow B\ k}\ \forall \mathcal{L}\ .$$

The premise $B\ k, \ldots \longrightarrow B\ k$ is derivable for any predicate B, since the consequent $B\ k$ follows from the antecedent $B\ k$. ∎

The following proposition presents additional properties of natural numbers that we have derived in $FO\lambda^{\Delta\mathbb{N}}$, although we do not show the derivations here.

Proposition 2.17 *The following formulas are derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition $\mathcal{D}(nat)$:*

$$\forall i(nat\ (s\ i) \supset nat\ i)$$

$$\forall i(nat\ i \supset \forall j(i < j \supset nat\ j))$$

$$\forall i(nat\ i \supset i < (s\ i))$$

$$\forall i(nat\ i \supset \forall j \forall k(i < j \supset j < k \supset i < k))$$

$$\forall i(nat\ i \supset \forall j(nat\ j \supset \exists k(nat\ k \wedge i < k \wedge j < k)))$$

$$\forall i(nat\ i \supset \forall j \forall k(sum\ i\ (s\ j)\ k \supset sum\ (s\ i)\ j\ k))$$

$$\forall i(nat\ i \supset \forall j(nat\ j \supset \exists k(nat\ k \wedge sum\ i\ j\ k)))$$

$$\forall i(nat\ i \supset \forall j \forall k(nat\ j \supset sum\ i\ j\ k \supset i \leq k))$$

$$\forall i(nat\ i \supset \forall j \forall k(nat\ j \supset sum\ (s\ i)\ j\ k \supset j < k))\ .$$

2.2.2 Lists

In this section we introduce a type *lst* for lists over an arbitrary but fixed type τ. The type has two constructors, $nil : lst$ representing the empty list and the infix operator :: of type $\tau \to lst \to lst$ that adds an element to the front of a list. Consider the list predicates

$$\begin{array}{rcl@{\qquad}rcl} length & : & lst \to nt \to o & split & : & lst \to lst \to lst \to o \\ list & : & lst \to o & permute & : & lst \to lst \to o \\ element & : & \tau \to lst \to o & & & \end{array} ,$$

whose definitional clauses are shown in Table 2.3; we shall refer to this set of clauses as $\mathcal{D}(list(\tau))$. The predicate *length* represents the function that returns the length of its list argument. The length of the empty list is zero, and the length of $(X :: L)$ is one more than the length of L. The predicate *list* indicates that its argument has a finite (natural number) length. We shall find this predicate useful for constructing induction principles over lists. The predicate *element* indicates that its first argument is a member of its second argument. X is an element of $(Y :: L)$ if X and Y are the same or if X is an element of L. The predicate *split* holds if its first argument represents a merging of the second and third in which the order of elements in second and third lists is preserved in the first. The empty list can only be split into two empty lists. To split $(X :: L)$, we split L and add X to the front of either of the resulting lists. The predicate *permute* holds if its two arguments

Table 2.3: Definitional clauses for predicates over lists

$\mathit{length}\ \mathit{nil}\ z$	$\triangleq$	$\top$
$\mathit{length}\ (X :: L)\ (s\ I)$	$\triangleq$	$\mathit{length}\ L\ I$
$\mathit{list}\ L$	$\triangleq$	$\exists i(\mathit{nat}\ i \wedge \mathit{length}\ L\ i)$
$\mathit{element}\ X\ (X :: L)$	$\triangleq$	$\top$
$\mathit{element}\ X\ (Y :: L)$	$\triangleq$	$\mathit{element}\ X\ L$
$\mathit{split}\ \mathit{nil}\ \mathit{nil}\ \mathit{nil}$	$\triangleq$	$\top$
$\mathit{split}\ (X :: L_1)\ (X :: L_2)\ L_3$	$\triangleq$	$\mathit{split}\ L_1\ L_2\ L_3$
$\mathit{split}\ (X :: L_1)\ L_2\ (X :: L_3)$	$\triangleq$	$\mathit{split}\ L_1\ L_2\ L_3$
$\mathit{permute}\ \mathit{nil}\ \mathit{nil}$	$\triangleq$	$\top$
$\mathit{permute}\ (X :: L_1)\ L_2$	$\triangleq$	$\exists l_{22}(\mathit{split}\ L_2\ (X :: \mathit{nil})\ l_{22} \wedge \mathit{permute}\ L_1\ l_{22}\ )$

contain the same elements (including repetitions), though not necessarily in the same order. The empty list only permutes to itself. A list $(X :: L_1)$ permutes to L_2 if removing X from L_2 yields a permutation of L_1.

We now derive an induction rule for lists from the induction rule for natural numbers ($\mathit{nat}\mathcal{L}$) using the length of a list as our measure.

Proposition 2.18 *For any formula $C : o$, predicate $B : \mathit{lst} \to o$, term $L : \mathit{lst}$, multiset Γ of formulas, and eigenvariable $l : \mathit{lst}$ such that l is not free in B, the following rule is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition $\mathcal{D}(\mathit{list}(\tau))$:*

$$\frac{\longrightarrow B\ \mathit{nil} \quad B\ l \longrightarrow B\ (x :: l) \quad B\ L, \Gamma \longrightarrow C}{\mathit{list}\ L, \Gamma \longrightarrow C}\ .$$

Proof To derive this rule, we construct a partial derivation of the sequent $\mathit{list}\ L, \Gamma \longrightarrow C$, leaving unproved premises of the form $\longrightarrow B\ \mathit{nil}$, $B\ l \longrightarrow B\ (x :: l)$, and $B\ L, \Gamma \longrightarrow C$. This corresponds to proving that C follows from Γ and the fact that L is a list under the assumptions

- B holds for *nil*;

- for any x' and l', if B holds for l', then it also holds for $(x' :: l')$;
- $B\,L$ and Γ imply C.

The proof is by induction on the length of the list L. Since *list* L holds, by definition L has a length which is a natural number:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{nat\ i, length\ L\ i, \Gamma \longrightarrow C}{nat\ i, nat\ i \wedge length\ L\ i, \Gamma \longrightarrow C}\ \wedge\mathcal{L}}{nat\ i \wedge length\ L\ i, nat\ i \wedge length\ L\ i, \Gamma \longrightarrow C}\ \wedge\mathcal{L}}{nat\ i \wedge length\ L\ i, \Gamma \longrightarrow C}\ c\mathcal{L}}{\exists i(nat\ i \wedge length\ L\ i), \Gamma \longrightarrow C}\ \exists\mathcal{L}}{list\ L, \Gamma \longrightarrow C}\ def\mathcal{L}\quad .$$

We now claim that B holds for lists of any length, and wish to prove this claim by induction on the length of the list. Thus we must prove

1. the base case: B holds for lists of length zero;
2. the induction step: if B holds for lists of length i', it holds for lists of length $(s\ i')$;
3. the relevance of the claim: C follows from Γ, the fact that L has length i, and the fact that B holds for lists of length i.

This is represented in $FO\lambda^{\Delta\mathbb{N}}$ by applying the $nat\mathcal{L}$ rule with the induction predicate $\lambda i.\forall l(length\ l\ i \supset B\ l)$, which yields the three sequents

$$\longrightarrow \forall l(length\ l\ z \supset B\,l)$$

$$\forall l(length\ l\ i' \supset B\,l) \longrightarrow \forall l(length\ l\ (s\ i') \supset B\,l)$$

$$\forall l(length\ l\ i \supset B\,l), length\ L\ i, \Gamma \longrightarrow C\ .$$

Once we have proved that B holds for lists of length i, then we know it holds for L. Thus we know that C follows from Γ, since our third working assumption says that C follows from $B\,L$ and Γ. This is represented formally by the partial derivation of the third premise of the $nat\mathcal{L}$ rule:

$$\dfrac{\dfrac{\dfrac{}{length\ L\ i, \Gamma \longrightarrow length\ L\ i}\ init \qquad B\,L, length\ L\ i, \Gamma \longrightarrow C}{length\ L\ i \supset B\,L, length\ L\ i, \Gamma \longrightarrow C}\ \supset\mathcal{L}}{\forall l(length\ l\ i \supset B\,l), length\ L\ i, \Gamma \longrightarrow C}\ \forall\mathcal{L}\quad .$$

The unproved premise of this partial derivation is actually a weakening of the third premise of the induction rule we are deriving. We do not have an explicit weakening rule in $FO\lambda^{\Delta\mathbb{N}}$, but it suffices here to use the cut rule:

$$\frac{B\,L, \mathit{length}\;L\;i \longrightarrow B\,L \quad B\,L, \Gamma \longrightarrow C}{B\,L, \mathit{length}\;L\;i, \Gamma \longrightarrow C}\;\mathit{mc}\quad .$$

The first premise of the cut rule is derivable for any B and L, since the consequent $B\,L$ also occurs as an antecedent. The second premise is the desired premise of the rule we are deriving.

In the base case of the induction, we must show that B holds for lists of length zero. Since the only list of length zero is *nil*, this follows from the first working assumption, which says that $B\,\mathit{nil}$ holds. This case is formalized in the following partial derivation of the first premise of the $\mathit{nat}\mathcal{L}$ rule:

$$\dfrac{\dfrac{\dfrac{\top \longrightarrow B\,\mathit{nil}}{\mathit{length}\;l\;z \longrightarrow B\,l}\;\mathit{def}\mathcal{L}}{\longrightarrow \mathit{length}\;l\;z \supset B\,l}\;\supset\mathcal{R}}{\longrightarrow \forall l(\mathit{length}\;l\;z \supset B\,l)}\;\forall\mathcal{R}\quad .$$

The induction step requires us to prove that B holds for all lists of length $(s\;i')$, given that it holds for all lists of length i'. Since a list of length $(s\;i')$ is constructed by adding an element to the front of a list of length i', this step follows from the second working assumption, which says that if B holds for a list l, then for any $x : \tau$, B holds for $x :: l$. This reasoning represented in the partial derivation of the second premise of the $\mathit{nat}\mathcal{L}$ rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathit{length}\;l'\;i' \longrightarrow \mathit{length}\;l'\;i'}\;\mathit{init} \quad B\,l', \mathit{length}\;l'\;i' \longrightarrow B\,(x' :: l')}{\mathit{length}\;l'\;i' \supset B\,l', \mathit{length}\;l'\;i' \longrightarrow B\,(x' :: l')}\;\supset\mathcal{L}}{\forall l(\mathit{length}\;l\;i' \supset B\,l), \mathit{length}\;l'\;i' \longrightarrow B\,(x' :: l')}\;\forall\mathcal{L}}{\forall l(\mathit{length}\;l\;i' \supset B\,l), \mathit{length}\;l\;(s\;i') \longrightarrow B\,l}\;\mathit{def}\mathcal{L}}{\forall l(\mathit{length}\;l\;i' \supset B\,l) \longrightarrow \mathit{length}\;l\;(s\;i') \supset B\,l}\;\supset\mathcal{R}}{\forall l(\mathit{length}\;l\;i' \supset B\,l) \longrightarrow \forall l(\mathit{length}\;l\;(s\;i') \supset B\,l)}\;\forall\mathcal{R}\quad .$$

The unproved premise of this partial derivation is a weakening of the second premise of the induction rule we are deriving. We can achieve this weakening using the cut rule in the same manner as we did for the third premise:

$$\frac{B\,l', \mathit{length}\;l'\;i' \longrightarrow B\,l' \quad B\,l' \longrightarrow B\,(x' :: l')}{B\,l', \mathit{length}\;l'\;i' \longrightarrow B\,(x' :: l')}\;\mathit{mc}\quad . \qquad \blacksquare$$

We will now use this derived induction rule for lists to prove a very simple property, namely that we can split any list L into *nil* and L.

Proposition 2.19 *The formula* $\forall l(\mathit{list}\ l \supset \mathit{split}\ l\ \mathit{nil}\ l)$ *is derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(\mathit{list}(\tau))$.

Proof We prove this by induction on l; using the right rules for $\forall$ and $\supset$ and the derived rule of Proposition 2.18 with the induction predicate $(\lambda l.\mathit{split}\ l\ \mathit{nil}\ l)$, we get the three sequents

$$\longrightarrow \mathit{split}\ \mathit{nil}\ \mathit{nil}\ \mathit{nil}$$

$$\mathit{split}\ l'\ \mathit{nil}\ l' \longrightarrow \mathit{split}\ (x'::l')\ \mathit{nil}\ (x'::l')$$

$$\mathit{split}\ l\ \mathit{nil}\ l \longrightarrow \mathit{split}\ l\ \mathit{nil}\ l \quad .$$

Since the induction predicate applied to l is the same as the consequent, the relevance of the induction predicate is immediate. Thus the third sequent follows from the *init* rule.

The base case follows immediately from the definition of *split*, and so the first sequent is derivable using the $def\mathcal{R}$ and $\top\mathcal{R}$ rules.

The induction step also follows easily from the definition of *split*:

$$\dfrac{\dfrac{}{\mathit{split}\ l'\ \mathit{nil}\ l' \longrightarrow \mathit{split}\ l'\ \mathit{nil}\ l'}\ \mathit{init}}{\mathit{split}\ l'\ \mathit{nil}\ l' \longrightarrow \mathit{split}\ (x'::l')\ \mathit{nil}\ (x'::l')}\ def\mathcal{R} \quad . \qquad \blacksquare$$

Next we prove that any list is a permutation of itself.

Proposition 2.20 *The formula* $\forall l(\mathit{list}\ l \supset \mathit{permute}\ l\ l\ )$ *is derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(\mathit{list}(\tau))$.

Proof We prove this by induction on l; applying the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules and the derived rule of Proposition 2.18 with the induction predicate $(\lambda l.\mathit{list}\ l \wedge \mathit{permute}\ l\ l\ )$ yields the three sequents

$$\longrightarrow \mathit{list}\ \mathit{nil} \wedge \mathit{permute}\ \mathit{nil}\ \mathit{nil}$$

$$\mathit{list}\ l' \wedge \mathit{permute}\ l'\ l' \longrightarrow \mathit{list}\ (x'::l') \wedge \mathit{permute}\ (x'::l')\ (x'::l')$$

$$\mathit{list}\ l \wedge \mathit{permute}\ l\ l \longrightarrow \mathit{permute}\ l\ l \quad .$$

The third premise, representing the relevance of the induction predicate, is easily derived:

$$\dfrac{\dfrac{}{\mathit{permute}\ l\ l\ \longrightarrow \mathit{permute}\ l\ l}\ \mathit{init}}{\mathit{list}\ l \wedge \mathit{permute}\ l\ l\ \longrightarrow \mathit{permute}\ l\ l}\ \wedge\mathcal{L}\ .$$

The base case, that the empty list is a list and permutes to itself, follows simply from the definitions:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\longrightarrow \mathit{nat}\ z}\ \mathit{nat}\mathcal{R} \quad \dfrac{\dfrac{}{\longrightarrow \top}\ \top\mathcal{R}}{\longrightarrow \mathit{length}\ \mathit{nil}\ z}\ \mathit{def}\mathcal{R}}{\longrightarrow \mathit{nat}\ z \wedge \mathit{length}\ \mathit{nil}\ z}\ \wedge\mathcal{R}}{\longrightarrow \exists i(\mathit{nat}\ i \wedge \mathit{length}\ \mathit{nil}\ i)}\ \exists\mathcal{R}}{\longrightarrow \mathit{list}\ \mathit{nil}}\ \mathit{def}\mathcal{R} \qquad \dfrac{\dfrac{}{\longrightarrow \top}\ \top\mathcal{R}}{\longrightarrow \mathit{permute}\ \mathit{nil}\ \mathit{nil}}\ \mathit{def}\mathcal{R}}{\longrightarrow \mathit{list}\ \mathit{nil} \wedge \mathit{permute}\ \mathit{nil}\ \mathit{nil}}\ \wedge\mathcal{R}\ .$$

For the induction step, we must prove that $(x' :: l')$ is a list and permutes to itself, given that these are true of l':

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathit{list}\ l', \ldots \longrightarrow \mathit{list}\ (x' :: l') \quad \mathit{list}\ l', \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{permute}\ (x' :: l')\ (x' :: l')}{\mathit{list}\ l', \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{list}\ (x' :: l') \wedge \mathit{permute}\ (x' :: l')\ (x' :: l')}\ \wedge\mathcal{R}}{\mathit{list}\ l', \mathit{list}\ l' \wedge \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{list}\ (x' :: l') \wedge \mathit{permute}\ (x' :: l')\ (x' :: l')}\ \wedge\mathcal{L}}{\mathit{list}\ l' \wedge \mathit{permute}\ l'\ l'\ , \mathit{list}\ l' \wedge \mathit{permute}\ l'\ l'\ \longrightarrow \ldots}\ \wedge\mathcal{L}}{\mathit{list}\ l' \wedge \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{list}\ (x' :: l') \wedge \mathit{permute}\ (x' :: l')\ (x' :: l')}\ c\mathcal{L}\ .$$

To show that $(x' :: l')$ is a list, we must provide its length. This is just one more than the length of l':

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathit{nat}\ i \wedge \mathit{length}\ l'\ i, \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{nat}\ (s\ i) \wedge \mathit{length}\ (x' :: l')\ (s\ i)}{\mathit{nat}\ i \wedge \mathit{length}\ l'\ i, \mathit{permute}\ l'\ l'\ \longrightarrow \exists i(\mathit{nat}\ i \wedge \mathit{length}\ (x' :: l')\ i)}\ \exists\mathcal{R}}{\mathit{nat}\ i \wedge \mathit{length}\ l'\ i, \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{list}\ (x' :: l')}\ \mathit{def}\mathcal{R}}{\exists i(\mathit{nat}\ i \wedge \mathit{length}\ l'\ i), \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{list}\ (x' :: l')}\ \exists\mathcal{L}}{\mathit{list}\ l', \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{list}\ (x' :: l')}\ \mathit{def}\mathcal{L}\ .$$

Applying the $\wedge\mathcal{R}$ rule to the unproved sequent of this partial derivation yields the two sequents:

$$\mathit{nat}\ i \wedge \mathit{length}\ l'\ i, \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{nat}\ (s\ i)$$

$$\mathit{nat}\ i \wedge \mathit{length}\ l'\ i, \mathit{permute}\ l'\ l'\ \longrightarrow \mathit{length}\ (x' :: l')\ (s\ i)\ .$$

That $(s\ i)$ is a natural number follows from the fact that the length i of l' is a natural number:

$$\dfrac{\dfrac{\dfrac{}{nat\ i, permute\ l'\ l'\ \longrightarrow nat\ i}\ init}{nat\ i \wedge length\ l'\ i, permute\ l'\ l'\ \longrightarrow nat\ i}\ \wedge\mathcal{L}}{nat\ i \wedge length\ l'\ i, permute\ l'\ l'\ \longrightarrow nat\ (s\ i)}\ nat\mathcal{R}\ .$$

That $(s\ i)$ is the length of $(x'::l')$ follows by definition from the fact that i is the length of l':

$$\dfrac{\dfrac{\dfrac{}{length\ l'\ i, permute\ l'\ l'\ \longrightarrow length\ l'\ i}\ init}{nat\ i \wedge length\ l'\ i, permute\ l'\ l'\ \longrightarrow length\ l'\ i}\ \wedge\mathcal{L}}{nat\ i \wedge length\ l'\ i, permute\ l'\ l'\ \longrightarrow length\ (x'::l')\ (s\ i)}\ def\mathcal{R}\ .$$

It still remains to prove that $(x'::l')$ permutes to itself. By the definition of *permute*, this is true if $(x'::l')$ splits into $(x'::nil)$ and l':

$$\dfrac{\dfrac{\dfrac{\dfrac{list\ l', \ldots \longrightarrow split\ (x'::l')\ (x'::nil)\ l' \quad \dfrac{}{\ldots, permute\ l'\ l'\ \longrightarrow permute\ l'\ l'}\ init}{list\ l', permute\ l'\ l'\ \longrightarrow split\ (x'::l')\ (x'::nil)\ l' \wedge permute\ l'\ l'}\ \wedge\mathcal{R}}{list\ l', permute\ l'\ l'\ \longrightarrow \exists l_{22}(split\ (x'::l')\ (x'::nil)\ l_{22} \wedge permute\ l'\ l_{22}\ )}\ \exists\mathcal{R}}{list\ l', permute\ l'\ l'\ \longrightarrow permute\ (x'::l')\ (x'::l')}\ def\mathcal{R}\ .$$

By the definition of *split*, $(x'::l')$ splits into $(x'::nil)$ and l' if l' splits into *nil* and itself:

$$\dfrac{list\ l', permute\ l'\ l'\ \longrightarrow split\ l'\ nil\ l'}{list\ l', permute\ l'\ l'\ \longrightarrow split\ (x'::l')\ (x'::nil)\ l'}\ def\mathcal{R}\ .$$

We proved that any list splits into *nil* and itself in Proposition 2.19, so we are done. ∎

We conclude this section with a proposition that presents additional properties of lists that we have derived in $FO\lambda^{\Delta\mathbb{IN}}$, though we omit the derivations here.

Proposition 2.21 *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{IN}}$ *using the definition* $\mathcal{D}(list(\tau))$*:*

$$\forall l(list\ l \supset \forall l_1 \forall l_2(split\ l\ l_1\ l_2 \supset (list\ l_1 \wedge list\ l_2)))$$

$$\forall l_1(list\ l_1 \supset \forall l_2(list\ l_2 \supset \forall l(split\ l\ l_1\ l_2 \supset list\ l)))$$

$$\forall l(list\ l \supset \forall l_1 \forall l_2(split\ l\ l_1\ l_2 \supset (\forall x(element\ x\ l_1 \supset element\ x\ l) \wedge \forall x(element\ x\ l_2 \supset element\ x\ l))))$$

$$\forall l(list\ l \supset \forall l_1 \forall l_2(split\ l\ l_1\ l_2 \supset split\ l\ l_2\ l_1))$$

$$\forall l(list\ l \supset \forall l_{23} \forall l_1 \forall l_2 \forall l_3(split\ l\ l_1\ l_{23} \supset split\ l_{23}\ l_2\ l_3 \supset \exists l_{12}(split\ l\ l_{12}\ l_3 \wedge split\ l_{12}\ l_1\ l_2)))$$

$$\forall l(\mathit{list}\ l \supset \forall l_{12}\forall l_1\forall l_2\forall l_3(\mathit{split}\ l\ l_{12}\ l_3 \supset \mathit{split}\ l_{12}\ l_1\ l_2 \supset \exists l_{23}(\mathit{split}\ l\ l_1\ l_{23} \wedge \mathit{split}\ l_{23}\ l_2\ l_3)))$$

$$\forall l(\mathit{list}\ l \supset \forall l'(\mathit{permute}\ l\ l' \supset \mathit{list}\ l'))$$

$$\forall l(\mathit{list}\ l \supset \forall l'\forall l_1\forall l_2(\mathit{list}\ l' \supset \mathit{permute}\ l\ l' \supset \mathit{split}\ l\ l_1\ l_2 \supset \exists l'_1\exists l'_2(\mathit{permute}\ l_1\ l'_1 \wedge \mathit{permute}\ l_2\ l'_2 \wedge \mathit{split}\ l'\ l'_1\ l'_2)))$$

$$\forall l(\mathit{list}\ l \supset \forall l'\forall l_1\forall l'_1\forall l_2\forall l'_2(\mathit{list}\ l' \supset \mathit{split}\ l\ l_1\ l_2 \supset \mathit{split}\ l'\ l'_1\ l'_2 \supset \mathit{permute}\ l_1\ l'_1 \supset \mathit{permute}\ l_2\ l'_2 \supset \mathit{permute}\ l\ l'))\ .$$

2.3 Related Work

The logic $FO\lambda^{\Delta\mathbb{N}}$ is related to the various logics with "definitional reflection" [50]. The key idea behind these logics is that they allow a larger class of definitions than the class of monotone inductive definitions. In particular, implications are allowed to occur in the bodies of definitional clauses. As we have seen in Section 2.2, a rule of definitional reflection such as the $\mathit{def}\mathcal{L}$ rule of $FO\lambda^{\Delta\mathbb{N}}$ allows one to reason about all ways in which a defined atom might hold. Hallnäs introduced these ideas in his calculus of partial inductive definitions [19], a propositional calculus that is infinitary in that it allows conjunctions over infinite sets and definitions with an infinite number of clauses. Hallnäs also worked with Schroeder-Heister to generalize the system to a predicate calculus setting [20]. Eriksson formulated a finitary version of the calculus, enriched the definitional reflection rule, and added an induction principle valid for defined concepts whose definitions fall within the class of monotone inductive definitions [11]. Girard independently constructed a rule similar to Eriksson's in a linear logic setting [17].

In all of these systems except Girard's, the cut rule is not admissible and definitions may introduce inconsistencies. Hallnäs and Schroeder-Heister were not concerned with the inadmissibility of the cut rule, since they were using the systems to specify computations in a logic programming sense. Our intent, however, is to use the logic for reasoning about systems. As a result, it is important to us that the cut rule be admissible, and consistency is essential for our purposes. To achieve these we limit the use of implication in definitions by means of the level restriction discussed in Section 2.1. Note that this is a restriction and not a prohibition of the use implication in definitions, and so our class of definitions is

still richer than monotone inductive definitions. This is important for our use of $FO\lambda^{\Delta\mathbb{N}}$ in Chapter 4. We also formulate our induction rule differently than Eriksson (and we also limit it to natural numbers for simplicity). As a result of the restriction on definitions and the formulation of the $nat\mathcal{L}$ rule, cut-elimination and consistency hold for $FO\lambda^{\Delta\mathbb{N}}$, as we prove in Chapter 3.

Chapter 3

Cut-Elimination for $FO\lambda^{\Delta\mathbb{IN}}$

The purpose of this chapter is to present a proof of cut-elimination for $FO\lambda^{\Delta\mathbb{IN}}$; the consistency of the logic will be a simple corollary of this. Gentzen's classic proof of cut-elimination for first-order logic [15] uses an induction on the number of logical connectives in the cut formula. A cut on a compound formula is replaced by cuts on its subformulas, which necessarily contain a lower number of connectives. For example, the derivation

$$\dfrac{\dfrac{\begin{matrix}\Pi_1 & \Pi_2\\ \Delta \longrightarrow B_1 & \Delta \longrightarrow B_2\end{matrix}}{\Delta \longrightarrow B_1 \wedge B_2}\wedge\mathcal{R} \quad \dfrac{\begin{matrix}\Pi_3\\ B_1, \Gamma \longrightarrow C\end{matrix}}{B_1 \wedge B_2, \Gamma \longrightarrow C}\wedge\mathcal{L}}{\Delta, \Gamma \longrightarrow C}\, mc$$

is reduced to

$$\dfrac{\begin{matrix}\Pi_1 & \Pi_3\\ \Delta \longrightarrow B_1 & B_1, \Gamma \longrightarrow C\end{matrix}}{\Delta, \Gamma \longrightarrow C}\, mc \quad .$$

By the induction hypothesis, this cut on B_1 is eliminable, hence the original cut on $B_1 \wedge B_2$ is also eliminable. In first-order logic, when the cut formula is atomic, the cut can easily be removed by permuting the cut up toward the leaves of the proof; eventually an initial rule is reached, at which point the removal of the cut is trivial.

In $FO\lambda^{\Delta\mathbb{IN}}$, however, the rules for natural numbers and defined concepts act on atoms, so the atomic case is not simple. Consider, for example, the derivation

$$\dfrac{\dfrac{\begin{matrix}\Pi_1\\ \Delta \longrightarrow B\theta\end{matrix}}{\Delta \longrightarrow A}\, def\mathcal{R} \quad \dfrac{\left\{\begin{matrix}\Pi^{\rho,\sigma,D}\\ D\sigma, \Gamma\rho \longrightarrow C\rho\end{matrix}\right\}}{A, \Gamma \longrightarrow C}\, def\mathcal{L}}{\Delta, \Gamma \longrightarrow C}\, mc \quad .$$

The obvious reduction for this is a cut between Π_1 and the appropriate premise of the $def\mathcal{L}$ rule; however, $B\theta$ is a formula of arbitrary complexity, and so will in general have a greater number of connectives than the atom A, which has zero. Thus a different induction measure is needed.

Schroeder-Heister proves cut-elimination for several logics with definitions [49, 50] by including the number of uses of the $def\mathcal{L}$ rule in the derivation as part of the induction measure. However, the logics he considers do not contain induction, which complicates things further. The derivation

$$\dfrac{\dfrac{\dfrac{\overset{\Pi_1}{\Delta \longrightarrow nat\ I}}{\Delta \longrightarrow nat\ (s\ I)}\ nat\mathcal{R} \quad \dfrac{\overset{\Pi_2}{\longrightarrow B\,z} \quad \overset{\Pi_3}{B\,j \longrightarrow B\,(s\ j)} \quad \overset{\Pi_4}{B\,(s\ I), \Gamma \longrightarrow C}}{nat\ (s\ I), \Gamma \longrightarrow C}\ nat\mathcal{L}}{\Delta, \Gamma \longrightarrow C}\ mc$$

can be reduced in a number of ways, but the reductions are all variations of the derivation

$$\dfrac{\dfrac{\overset{\Pi_1}{\Delta \longrightarrow nat\ I} \quad \dfrac{\overset{\Pi_2}{\longrightarrow B\,z} \quad \overset{\Pi_3}{B\,j \longrightarrow B\,(s\ j)} \quad \overset{\Pi_3[I/j]}{B\,I \longrightarrow B\,(s\ I)}}{nat\ I \longrightarrow B\,(s\ I)}\ nat\mathcal{L}}{\Delta \longrightarrow B\,(s\ I)}\ mc \quad \dfrac{\overset{\Pi_4}{B\,(s\ I), \Gamma \longrightarrow C}}{\Delta, \Gamma \longrightarrow C}\ mc}{}\ .$$

Here, the cut on the atomic formula $nat\ (s\ I)$ is replaced by two cuts, one on the atom $nat\ I$ and the other on the formula $B\,(s\ I)$. It is not clear what induction measure can be used here. For the first cut, the atom $nat\ I$ contains no logical connectives, but this is true of the original cut formula $nat\ (s\ I)$ as well. The number of $nat\mathcal{R}$ rules in the right subderivation of the cut has gone down by one, but the duplication of Π_3 might offset this. For the second cut, the cut formula $B\,(s\ I)$ is not related at all to the original cut formula; it certainly can have no fewer connectives than the atom $nat\ (s\ I)$. And though its left premise is shorter than the left premise of the original cut, it is unclear how the heights of the right premises compare.

Our proof uses a technique introduced by Tait [54] to prove normal form theorems. Martin-Löf extended the method to apply beyond terms to natural deduction proofs [28], and we use it here in a sequent calculus setting. Rather than associate an induction measure with derivations, we use the derivations themselves as a measure by defining well-founded orderings on derivations, and performing the induction relative to those orderings. The

basis for the orderings is a set of reduction rules, such as those suggested above, that will be used to eliminate applications of the cut rule. We give these reduction rules in Section 3.1. This is followed by a section which discusses two orderings on derivations, a *normalizability* ordering and a *reducibility* ordering. The well-foundedness of the normalizability ordering for a derivation implies that the reduction rules can be used to reduce the derivation to a cut-free derivation of the same end-sequent. The reducibility ordering is a superset of the normalizability ordering; thus its well-foundedness implies the well-foundedness of the normalizability ordering. (This notion of reducibility was called convertibility by Tait and computability by Martin-Löf. We prefer to avoid these terms, since they carry other meanings in theoretical computer science, and instead use reducibility after Girard [16].) In Section 3.3 we prove the key lemma: for every derivation, the tree of its successive predecessors in the reducibility relation is well-founded. From this we conclude that the corresponding tree in the normalizability relation is also well-founded, and hence the cut rule can be eliminated from that derivation. Since this holds for every derivation, the consistency of $FO\lambda^{\Delta\mathbb{N}}$ follows. We conclude with a brief comparison of our work with the work of Martin-Löf and Schroeder-Heister mentioned above.

3.1 Reduction Rules for Derivations

Here we define a reduction relation between derivations, which is an adaptation of the reduction rules used in Gentzen's original Hauptsatz [15].

Definition 3.1 We define a *reduction* relation between derivations. The redex is always a derivation Ξ ending with the multicut rule

$$\frac{\overset{\Pi_1}{\Delta_1 \longrightarrow B_1} \quad \cdots \quad \overset{\Pi_n}{\Delta_n \longrightarrow B_n} \quad \overset{\Pi}{B_1, \ldots, B_n, \Gamma \longrightarrow C}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc \quad .$$

If $n = 0$, Ξ reduces to the premise derivation Π.

For $n > 0$ we specify the reduction relation based on the last rule of the premise derivations. If the rightmost premise derivation Π ends with a left rule acting on a cut formula B_i, then the last rule of Π_i and the last rule of Π together determine the reduction rules that apply. We classify these rules according to the following criteria: we call the rule

an *essential* case when Π_i ends with a right rule; if it ends with a left rule, it is a *right-commutative* case; if Π_i ends with the *init* rule, then we have an *axiom* case; a *multicut* case arises when it ends with the *mc* rule. When Π does not end with a left rule acting on a cut formula, then its last rule is alone sufficient to determine the reduction rules that apply. If Π ends in a rule acting on a formula other than a cut formula, then we call this a *left-commutative* case. A *structural* case results when Π ends with a contraction on a cut formula. If Π ends with the *init* rule, this is also an axiom case; similarly a multicut case arises if Π ends in the *mc* rule. For simplicity of presentation, we always show $i = 1$.

Essential cases:

$\wedge\mathcal{R}/\wedge\mathcal{L}$**:** If Π_1 and Π are

$$\frac{\overset{\Pi_1'}{\Delta_1 \longrightarrow B_1'} \quad \overset{\Pi_1''}{\Delta_1 \longrightarrow B_1''}}{\Delta_1 \longrightarrow B_1' \wedge B_1''}\ \wedge\mathcal{R} \qquad\qquad \frac{\overset{\Pi'}{B_1', B_2, \ldots, B_n, \Gamma \longrightarrow C}}{B_1' \wedge B_1'', B_2, \ldots, B_n, \Gamma \longrightarrow C}\ \wedge\mathcal{L}\ ,$$

then Ξ reduces to

$$\frac{\overset{\Pi_1'}{\Delta_1 \longrightarrow B_1'} \quad \overset{\Pi_2}{\Delta_2 \longrightarrow B_2} \quad \cdots \quad \overset{\Pi_n}{\Delta_n \longrightarrow B_n} \quad \overset{\Pi'}{B_1', B_2, \ldots, B_n, \Gamma \longrightarrow C}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc\ .$$

The case for the other $\wedge\mathcal{L}$ rule is symmetric.

$\vee\mathcal{R}/\vee\mathcal{L}$**:** If Π_1 and Π are

$$\frac{\overset{\Pi_1'}{\Delta_1 \longrightarrow B_1'}}{\Delta_1 \longrightarrow B_1' \vee B_1''}\ \vee\mathcal{R} \qquad\qquad \frac{\overset{\Pi'}{B_1', B_2, \ldots, B_n, \Gamma \longrightarrow C} \quad \overset{\Pi''}{B_1'', B_2, \ldots, B_n, \Gamma \longrightarrow C}}{B_1' \vee B_1'', B_2, \ldots, B_n, \Gamma \longrightarrow C}\ \vee\mathcal{L}\ ,$$

then Ξ reduces to

$$\frac{\overset{\Pi_1'}{\Delta_1 \longrightarrow B_1'} \quad \overset{\Pi_2}{\Delta_2 \longrightarrow B_2} \quad \cdots \quad \overset{\Pi_n}{\Delta_n \longrightarrow B_n} \quad \overset{\Pi'}{B_1', B_2, \ldots, B_n, \Gamma \longrightarrow C}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc\ .$$

The case for the other $\vee\mathcal{R}$ rule is symmetric.

$\supset\mathcal{R}/\supset\mathcal{L}$**:** Suppose Π_1 and Π are

$$\frac{\overset{\Pi_1'}{B_1', \Delta_1 \longrightarrow B_1''}}{\Delta_1 \longrightarrow B_1' \supset B_1''}\ \supset\mathcal{R} \qquad\qquad \frac{\overset{\Pi'}{B_2, \ldots, B_n, \Gamma \longrightarrow B_1'} \quad \overset{\Pi''}{B_1'', B_2, \ldots, B_n, \Gamma \longrightarrow C}}{B_1' \supset B_1'', B_2, \ldots, B_n, \Gamma \longrightarrow C}\ \supset\mathcal{L}\ .$$

Let Ξ_1 be

$$\dfrac{\dfrac{\left\{\begin{array}{c}\Pi_i\\ \Delta_i \longrightarrow B_i\end{array}\right\}_{i\in\{2..n\}} \quad \begin{array}{c}\Pi'\\ B_2,\ldots,B_n,\Gamma\longrightarrow B_1'\end{array}}{\Delta_2,\ldots,\Delta_n,\Gamma\longrightarrow B_1'}\,mc \qquad \begin{array}{c}\Pi_1'\\ B_1',\Delta_1\longrightarrow B_1''\end{array}}{\Delta_1,\ldots,\Delta_n,\Gamma\longrightarrow B_1''}\,mc \quad .$$

Then Ξ reduces to

$$\dfrac{\begin{array}{c}\Xi_1\\ \ldots\longrightarrow B_1''\end{array} \quad \left\{\begin{array}{c}\Pi_i\\ \Delta_i \longrightarrow B_i\end{array}\right\}_{i\in\{2..n\}} \quad \begin{array}{c}\Pi''\\ B_1'',\{B_i\}_{i\in\{2..n\}},\Gamma\longrightarrow C\end{array}}{\dfrac{\Delta_1,\ldots,\Delta_n,\Gamma,\Delta_2,\ldots,\Delta_n,\Gamma\longrightarrow C}{\Delta_1,\ldots,\Delta_n,\Gamma\longrightarrow C}\,c\mathcal{L}}\,mc \quad .$$

We use the double horizontal lines to indicate that the relevant inference rule (in this case, $c\mathcal{L}$) may need to be applied zero or more times.

$\forall\mathcal{R}/\forall\mathcal{L}$**:** If Π_1 and Π are

$$\dfrac{\begin{array}{c}\Pi_1'\\ \Delta_1\longrightarrow B_1'[y/x]\end{array}}{\Delta_1\longrightarrow\forall x.B_1'}\,\forall\mathcal{R} \qquad\qquad \dfrac{\begin{array}{c}\Pi'\\ B_1'[t/x],B_2,\ldots,B_n,\Gamma\longrightarrow C\end{array}}{\forall x.B_1',B_2,\ldots,B_n,\Gamma\longrightarrow C}\,\forall\mathcal{L} \quad ,$$

then Ξ reduces to

$$\dfrac{\begin{array}{c}\Pi_1'[t/y]\\ \Delta_1\longrightarrow B_1'[t/x]\end{array} \quad \left\{\begin{array}{c}\Pi_i\\ \Delta_i \longrightarrow B_i\end{array}\right\}_{i\in\{2..n\}} \quad \begin{array}{c}\Pi'\\ \ldots\longrightarrow C\end{array}}{\Delta_1,\ldots,\Delta_n,\Gamma\longrightarrow C}\,mc \quad .$$

$\exists\mathcal{R}/\exists\mathcal{L}$**:** If Π_1 and Π are

$$\dfrac{\begin{array}{c}\Pi_1'\\ \Delta_1\longrightarrow B_1'[t/x]\end{array}}{\Delta_1\longrightarrow\exists x.B_1'}\,\exists\mathcal{R} \qquad\qquad \dfrac{\begin{array}{c}\Pi'\\ B_1'[y/x],B_2,\ldots,B_n,\Gamma\longrightarrow C\end{array}}{\exists x.B_1',B_2,\ldots,B_n,\Gamma\longrightarrow C}\,\exists\mathcal{L} \quad ,$$

then Ξ reduces to

$$\dfrac{\begin{array}{c}\Pi_1'\\ \Delta_1\longrightarrow B_1'[t/x]\end{array} \quad \left\{\begin{array}{c}\Pi_i\\ \Delta_i \longrightarrow B_i\end{array}\right\}_{i\in\{2..n\}} \quad \begin{array}{c}\Pi'[t/y]\\ \ldots\longrightarrow C\end{array}}{\Delta_1,\ldots,\Delta_n,\Gamma\longrightarrow C}\,mc \quad .$$

natR/*natL*: Suppose Π_1 is $\dfrac{}{\Delta_1 \longrightarrow nat\ z}\ natR$ and Π is

$$\dfrac{\overset{\Pi'}{\longrightarrow D\,z} \quad \overset{\Pi''}{D\,j \longrightarrow D\,(s\ j)} \quad \overset{\Pi'''}{D\,z, B_2, \ldots, B_n, \Gamma \longrightarrow C}}{nat\ z, B_2, \ldots, B_n, \Gamma \longrightarrow C}\ natL\ .$$

Then Ξ reduces to

$$\dfrac{\overset{w(\Delta_1,\Pi')}{\Delta_1 \longrightarrow D\,z} \quad \left\{\overset{\Pi_i}{\Delta_i \longrightarrow B_i}\right\}_{i\in\{2..n\}} \quad \overset{\Pi'''}{D\,z, B_2, \ldots, B_n, \Gamma \longrightarrow C}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc\ .$$

natR/*natL*: Suppose Π_1 is

$$\dfrac{\overset{\Pi'_1}{\Delta_1 \longrightarrow nat\ I}}{\Delta_1 \longrightarrow nat\ (s\ I)}\ natR$$

and Π is

$$\dfrac{\overset{\Pi'}{\longrightarrow D\,z} \quad \overset{\Pi''}{D\,j \longrightarrow D\,(s\ j)} \quad \overset{\Pi'''}{D\,(s\ I), B_2, \ldots, B_n, \Gamma \longrightarrow C}}{nat\ (s\ I), B_2, \ldots, B_n, \Gamma \longrightarrow C}\ natL\ .$$

Let Ξ_1 be

$$\dfrac{\overset{\Pi'_1}{\Delta_1 \longrightarrow nat\ I} \quad \dfrac{\overset{\Pi'}{\longrightarrow D\,z} \quad \overset{\Pi''}{D\,j \longrightarrow D\,(s\ j)} \quad \dfrac{}{D\,I \longrightarrow D\,I}\ init}{nat\ I \longrightarrow D\,I}\ natL}{\Delta_1 \longrightarrow D\,I}\ mc\ ,$$

and Ξ_2 be

$$\dfrac{\overset{\Xi_1}{\Delta_1 \longrightarrow D\,I} \quad \overset{\Pi''[I/j]}{D\,I \longrightarrow D\,(s\ I)}}{\Delta_1 \longrightarrow D\,(s\ I)}\ mc\ .$$

Then Ξ reduces to

$$\dfrac{\overset{\Xi_2}{\Delta_1 \longrightarrow D\,(s\ I)} \quad \left\{\overset{\Pi_i}{\Delta_i \longrightarrow B_i}\right\}_{i\in\{2..n\}} \quad \overset{\Pi'''}{D\,(s\ I), B_2, \ldots, B_n, \Gamma \longrightarrow C}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc\ .$$

defR/*defL*: Suppose Π_1 and Π are

$$\dfrac{\overset{\Pi'_1}{\Delta_1 \longrightarrow B'_1\theta}}{\Delta_1 \longrightarrow B_1}\ defR \qquad \dfrac{\left\{\overset{\Pi^{\rho,\sigma,D}}{D\sigma, B_2\rho, \ldots, B_n\rho, \Gamma\rho \longrightarrow C\rho}\right\}}{B_1, B_2, \ldots, B_n, \Gamma \longrightarrow C}\ defL\ .$$

Then by the $def\mathcal{R}$ rule in Π_1 $\mathrm{dfn}(\epsilon, B_1, \theta, B_1')$ holds. Let θ' be the restriction of θ to the variables $\bar{x}$ of the relevant definitional clause. Since B_1' is the body of this clause, its free variables are included in $\bar{x}$, and so $B_1'\theta' = B_1'\theta$. Then Ξ reduces to

$$\dfrac{\begin{array}{cccc}\Pi_1' & \left\{\begin{array}{c}\Pi_i\\ \Delta_i \longrightarrow B_i\end{array}\right\}_{i\in\{2..n\}} & & \Pi^{\epsilon,\theta',B_1'}\\ \Delta_1 \longrightarrow B_1'\theta & & B_1'\theta, B_2, \ldots, B_n, \Gamma \longrightarrow C\end{array}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc \quad .$$

Left-commutative cases:

$\bullet\mathcal{L}/\circ\mathcal{L}$: Suppose Π ends with a left rule other than $c\mathcal{L}$ acting on B_1, and Π_1 is

$$\dfrac{\left\{\begin{array}{c}\Pi_1^i\\ \Delta_1^i \longrightarrow B_1\end{array}\right\}}{\Delta_1 \longrightarrow B_1}\ \bullet\mathcal{L} \quad ,$$

where $\bullet\mathcal{L}$ is any left rule except $\supset\mathcal{L}$, $def\mathcal{L}$, or $nat\mathcal{L}$ (but including $c\mathcal{L}$). Then Ξ reduces to

$$\dfrac{\left\{\dfrac{\begin{array}{cccc}\Pi_1^i & \left\{\begin{array}{c}\Pi_j\\ \Delta_j \longrightarrow B_j\end{array}\right\}_{j\in\{2..n\}} & & \Pi\\ \Delta_1^i \longrightarrow B_1 & & B_1, \ldots, B_n, \Gamma \longrightarrow C\end{array}}{\Delta_1^i, \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc\right\}}{\Delta_1, \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C}\ \bullet\mathcal{L} \quad .$$

$\supset\mathcal{L}/\circ\mathcal{L}$: Suppose Π ends with a left rule other than $c\mathcal{L}$ acting on B_1 and Π_1 is

$$\dfrac{\begin{array}{cc}\Pi_1' & \Pi_1''\\ \Delta_1' \longrightarrow D_1' & D_1'', \Delta_1' \longrightarrow B_1\end{array}}{D_1' \supset D_1'', \Delta_1' \longrightarrow B_1}\ \supset\mathcal{L} \quad .$$

Let Ξ_1 be

$$\dfrac{\begin{array}{ccccccc}\Pi_1'' & & \Pi_2 & & \Pi_n & & \Pi\\ D_1'', \Delta_1' \longrightarrow B_1 & \Delta_2 \longrightarrow B_2 & \cdots & \Delta_n \longrightarrow B_n & B_1, \ldots, B_n, \Gamma \longrightarrow C\end{array}}{D_1'', \Delta_1', \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc \quad .$$

Then Ξ reduces to

$$\dfrac{\begin{array}{cc}w(\Delta_2 \cup \ldots \cup \Delta_n \cup \Gamma, \Pi_1') & \Xi_1\\ \Delta_1', \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow D_1' & D_1'', \Delta_1', \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C\end{array}}{D_1' \supset D_1'', \Delta_1', \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C}\ \supset\mathcal{L} \quad .$$

nat$\mathcal{L}/\circ\mathcal{L}$**:** Suppose Π ends with a left rule other than $c\mathcal{L}$ acting on B_1, and Π_1 is

$$\dfrac{\begin{array}{ccc}\Pi_1^1 & \Pi_1^2 & \Pi_1^3\\ \longrightarrow D_1\, z & D_1\, j \longrightarrow D_1\,(s\; j) & D_1\, I, \Delta_1' \longrightarrow B_1\end{array}}{nat\; I, \Delta_1' \longrightarrow B_1}\; nat\mathcal{L}\ .$$

Let Ξ_1 be

$$\dfrac{\begin{array}{ccc}\Pi_1^3 & \left\{\begin{array}{c}\Pi_i\\ \Delta_i \longrightarrow B_i\end{array}\right\}_{i\in\{2..n\}} & \Pi\\ D_1\, I, \Delta_1' \longrightarrow B_1 & & B_1, \ldots, B_n, \Gamma \longrightarrow C\end{array}}{D_1\, I, \Delta_1', \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C}\; mc\ .$$

Then Ξ reduces to

$$\dfrac{\begin{array}{ccc}\Pi_1^1 & \Pi_1^2 & \Xi_1\\ \longrightarrow D_1\, z & D_1\, j \longrightarrow D_1\,(s\; j) & D_1\, I, \Delta_1', \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C\end{array}}{nat\; I, \Delta_1', \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C}\; nat\mathcal{L}\ .$$

def$\mathcal{L}/\circ\mathcal{L}$**:** If Π ends with a left rule other than $c\mathcal{L}$ acting on B_1 and Π_1 is

$$\dfrac{\left\{\begin{array}{c}\Pi_1^{\rho,\sigma,D}\\ D\sigma, \Delta_1'\rho \longrightarrow B_1\rho\end{array}\right\}}{A, \Delta_1' \longrightarrow B_1}\; def\mathcal{L}\ ,$$

then Ξ reduces to

$$\dfrac{\left\{\dfrac{\begin{array}{ccc}\Pi_1^{\rho,\sigma,D} & \left\{\begin{array}{c}\Pi_i\rho\\ \Delta_i\rho \longrightarrow B_i\rho\end{array}\right\}_{i\in\{2..n\}} & \Pi\rho\\ D\sigma, \Delta_1'\rho \longrightarrow B_1\rho & & \ldots \longrightarrow C\rho\end{array}}{D\sigma, \Delta_1'\rho, \Delta_2\rho, \ldots, \Delta_n\rho, \Gamma\rho \longrightarrow C\rho}\; mc\right\}}{A, \Delta_1', \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C}\; def\mathcal{L}\ .$$

Right-commutative cases:

$-/\circ\mathcal{L}$**:** Suppose Π is

$$\dfrac{\left\{\begin{array}{c}\Pi^i\\ B_1, \ldots, B_n, \Gamma^i \longrightarrow C\end{array}\right\}}{B_1, \ldots, B_n, \Gamma \longrightarrow C}\; \circ\mathcal{L}\ ,$$

where $\circ\mathcal{L}$ is any left rule other than $\supset\mathcal{L}$, *def*$\mathcal{L}$, or *nat*$\mathcal{L}$ (but including $c\mathcal{L}$) acting on a formula other than $B_1, \ldots, B_n$. Then Ξ reduces to

$$\dfrac{\left\{\dfrac{\begin{array}{ccccc}\Pi_1 & & \Pi_n & & \Pi^i\\ \Delta_1 \longrightarrow B_1 & \cdots & \Delta_n \longrightarrow B_n & B_1, \ldots, B_n, \Gamma^i \longrightarrow C & \end{array}}{\Delta_1, \ldots, \Delta_n, \Gamma^i \longrightarrow C}\; mc\right\}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\; \circ\mathcal{L}\ .$$

$-/\supset\mathcal{L}$**:** Suppose Π is

$$\dfrac{\overset{\Pi'}{B_1,\ldots,B_n,\Gamma'\longrightarrow D'}\quad\overset{\Pi''}{B_1,\ldots,B_n,D'',\Gamma'\longrightarrow C}}{B_1,\ldots,B_n,D'\supset D'',\Gamma'\longrightarrow C}\supset\mathcal{L}\ .$$

Let Ξ_1 be

$$\dfrac{\overset{\Pi_1}{\Delta_1\longrightarrow B_1}\quad\cdots\quad\overset{\Pi_n}{\Delta_n\longrightarrow B_n}\quad\overset{\Pi'}{B_1,\ldots,B_n,\Gamma'\longrightarrow D'}}{\Delta_1,\ldots,\Delta_n,\Gamma'\longrightarrow D'}\,mc$$

and Ξ_2 be

$$\dfrac{\overset{\Pi_1}{\Delta_1\longrightarrow B_1}\quad\cdots\quad\overset{\Pi_n}{\Delta_n\longrightarrow B_n}\quad\overset{\Pi''}{B_1,\ldots,B_n,D'',\Gamma'\longrightarrow C}}{\Delta_1,\ldots,\Delta_n,D'',\Gamma'\longrightarrow C}\,mc\ .$$

Then Ξ reduces to

$$\dfrac{\overset{\Xi_1}{\Delta_1,\ldots,\Delta_n,\Gamma'\longrightarrow D'}\quad\overset{\Xi_2}{\Delta_1,\ldots,\Delta_n,D'',\Gamma'\longrightarrow C}}{\Delta_1,\ldots,\Delta_n,D'\supset D'',\Gamma'\longrightarrow C}\supset\mathcal{L}\ .$$

$-/nat\mathcal{L}$**:** Suppose Π is

$$\dfrac{\overset{\Pi'}{\longrightarrow D\,z}\quad\overset{\Pi''}{D\,j\longrightarrow D\,(s\,j)}\quad\overset{\Pi'''}{B_1,\ldots,B_n,D\,I,\Gamma'\longrightarrow C}}{B_1,\ldots,B_n,nat\;I,\Gamma'\longrightarrow C}\,nat\mathcal{L}\ .$$

Let Ξ_1 be

$$\dfrac{\overset{\Pi_1}{\Delta_1\longrightarrow B_1}\quad\cdots\quad\overset{\Pi_n}{\Delta_n\longrightarrow B_n}\quad\overset{\Pi'''}{B_1,\ldots,B_n,D\,I,\Gamma'\longrightarrow C}}{\Delta_1,\ldots,\Delta_n,D\,I,\Gamma'\longrightarrow C}\,mc\ .$$

then Ξ reduces to

$$\dfrac{\overset{\Pi'}{\longrightarrow D\,z}\quad\overset{\Pi''}{D\,j\longrightarrow D\,(s\,j)}\quad\overset{\Xi_1}{\Delta_1,\ldots,\Delta_n,D\,I,\Gamma'\longrightarrow C}}{\Delta_1,\ldots,\Delta_n,nat\;I,\Gamma'\longrightarrow C}\,nat\mathcal{L}\ .$$

$-/def\mathcal{L}$**:** If Π is

$$\dfrac{\left\{\begin{array}{c}\overset{\Pi^{\rho,\sigma,D}}{B_1\rho,\ldots,B_n\rho,D\sigma,\Gamma'\rho\longrightarrow C\rho}\end{array}\right\}}{B_1,\ldots,B_n,A,\Gamma'\longrightarrow C}\,def\mathcal{L}\ ,$$

then Ξ reduces to

$$\dfrac{\left\{\dfrac{\left\{\begin{array}{c}\Pi_i\rho\\ \Delta_i\rho \longrightarrow B_i\rho\end{array}\right\}_{i\in\{1..n\}} \quad \begin{array}{c}\Pi^{\rho,\sigma,D}\\ \{B_i\rho\}_{i\in\{1..n\}}, D\sigma, \Gamma'\rho \longrightarrow C\rho\end{array}}{\Delta_1\rho,\ldots,\Delta_n\rho, D\sigma, \Gamma'\rho \longrightarrow C\rho}\ mc\right\}}{\Delta_1,\ldots,\Delta_n, A, \Gamma' \longrightarrow C}\ def\mathcal{L}\ .$$

$-/\circ\mathcal{R}$**:** If Π is

$$\dfrac{\left\{\begin{array}{c}\Pi^i\\ B_1,\ldots,B_n,\Gamma^i \longrightarrow C^i\end{array}\right\}}{B_1,\ldots,B_n,\Gamma \longrightarrow C}\ \circ\mathcal{R}\ ,$$

where $\circ\mathcal{R}$ is any right rule, then Ξ reduces to

$$\dfrac{\left\{\dfrac{\begin{array}{ccccc}\Pi_1 & & \Pi_n & & \Pi^i\\ \Delta_1 \longrightarrow B_1 & \cdots & \Delta_n \longrightarrow B_n & B_1,\ldots,B_n,\Gamma^i \longrightarrow C^i\end{array}}{\Delta_1,\ldots,\Delta_n,\Gamma^i \longrightarrow C^i}\ mc\right\}}{\Delta_1,\ldots,\Delta_n,\Gamma \longrightarrow C}\ \circ\mathcal{R}\ .$$

Multicut cases:

$mc/\circ\mathcal{L}$**:** If Π ends with a left rule other than $c\mathcal{L}$ acting on B_1 and Π_1 ends with a multicut and reduces to Π_1', then Ξ reduces to

$$\dfrac{\begin{array}{ccccccc}\Pi_1' & & \Pi_2 & & \Pi_n & & \Pi\\ \Delta_1 \longrightarrow B_1 & \Delta_2 \longrightarrow B_2 & \cdots & \Delta_n \longrightarrow B_n & B_1,\ldots,B_n,\Gamma \longrightarrow C\end{array}}{\Delta_1,\ldots,\Delta_n,\Gamma \longrightarrow C}\ mc\ .$$

$-/mc$**:** Suppose Π is

$$\dfrac{\left\{\begin{array}{c}\Pi^j\\ \{B_i\}_{i\in I^j},\Gamma^j \longrightarrow D^j\end{array}\right\}_{j\in\{1..m\}} \quad \begin{array}{c}\Pi'\\ \{D^j\}_{j\in\{1..m\}}, \{B_i\}_{i\in I'}, \Gamma' \longrightarrow C\end{array}}{B_1,\ldots,B_n,\Gamma^1,\ldots,\Gamma^m,\Gamma' \longrightarrow C}\ mc\ ,$$

where $I^1,\ldots,I^m,I'$ partition the formulas $\{B_i\}_{i\in\{1..n\}}$ among the premise derivations $\Pi_1,\ldots,\Pi_m,\Pi'$. For $1\leq j\leq m$ let Ξ^j be

$$\dfrac{\left\{\begin{array}{c}\Pi_i\\ \Delta_i \longrightarrow B_i\end{array}\right\}_{i\in I^j} \quad \begin{array}{c}\Pi^j\\ \{B_i\}_{i\in I^j},\Gamma^j \longrightarrow D^j\end{array}}{\{\Delta_i\}_{i\in I^j},\Gamma^j \longrightarrow D^j}\ mc\ .$$

Then Ξ reduces to

$$\dfrac{\left\{\begin{array}{c}\Xi^j\\ \ldots \longrightarrow D^j\end{array}\right\}_{j\in\{1..m\}} \quad \left\{\begin{array}{c}\Pi_i\\ \Delta_i \longrightarrow B_i\end{array}\right\}_{i\in I'} \quad \begin{array}{c}\Pi'\\ \ldots \longrightarrow C\end{array}}{\Delta_1,\ldots,\Delta_n,\Gamma^1,\ldots\Gamma^m,\Gamma' \longrightarrow C}\ mc\ .$$

Structural case:

$-/c\mathcal{L}$**:** If Π is

$$\dfrac{\begin{matrix}\Pi'\\ B_1, B_1, B_2, \ldots, B_n, \Gamma \longrightarrow C\end{matrix}}{B_1, \ldots, B_n, \Gamma \longrightarrow C}\ c\mathcal{L}\ ,$$

then Ξ reduces to

$$\dfrac{\dfrac{\begin{matrix}\Pi_1\\ \Delta_1 \longrightarrow B_1\end{matrix}\quad \left\{\begin{matrix}\Pi_i\\ \Delta_i \longrightarrow B_i\end{matrix}\right\}_{i\in\{1..n\}}\quad \begin{matrix}\Pi'\\ B_1, B_1, B_2, \ldots, B_n, \Gamma \longrightarrow C\end{matrix}}{\Delta_1, \Delta_1, \Delta_2, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ c\mathcal{L}\ .$$

Axiom cases:

init$/\circ\mathcal{L}$**:** If Π ends with either $nat\mathcal{L}$ or $def\mathcal{L}$ acting on B_1 and Π_1 ends with the *init* rule, then Ξ reduces to

$$\dfrac{\begin{matrix}\Pi_2\\ \Delta_2 \longrightarrow B_2\end{matrix}\quad \cdots\quad \begin{matrix}\Pi_n\\ \Delta_n \longrightarrow B_n\end{matrix}\quad \begin{matrix}w(\Delta_1 \setminus B_1, \Pi)\\ \Delta_1, B_2, \ldots, B_n, \Gamma \longrightarrow C\end{matrix}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc\ .$$

$-/$*init*: If Π ends with the *init* rule and C is a formula in Γ, then Ξ reduces to

$$\dfrac{}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ init\ .$$

If Π ends with the *init* rule, but C is not a formula in Γ, then C must be one of the cut formulas, say B_1. In this case Ξ reduces to $w(\Delta_2 \cup \ldots \cup \Delta_n \cup \Gamma, \Pi_1)$.

An inspection of the rules of the logic and this definition will reveal that every derivation ending with a multicut has a reduct. Because we use a multiset as the left side of the sequent, there may be ambiguity as to whether a formula occurring on the left side of the rightmost premise to a multicut rule is in fact a cut formula, and if so, which of the left premises corresponds to it. As a result, several of the reduction rules may apply, and so a derivation may have multiple reducts.

We now prove that the reduction relation is preserved by weakening.

Lemma 3.2 *If Ξ reduces to Ξ', then, for any multiset Δ of formulas, $w(\Delta, \Xi)$ reduces to $w(\Delta, \Xi')$.*

Proof We prove this lemma by a simple case analysis on the reduction rule used to reduce Ξ to Ξ'. Since Ξ is a redex, it is of the form

$$\dfrac{\begin{array}{cccc}\overset{\Pi_1}{\Delta_1 \longrightarrow B_1} & \cdots & \overset{\Pi_n}{\Delta_n \longrightarrow B_n} & \overset{\Pi}{B_1, \ldots, B_n, \Gamma \longrightarrow C}\end{array}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc\ ,$$

and $w(\Delta, \Xi)$ is of the form

$$\dfrac{\begin{array}{cccc}\overset{\Pi_1}{\Delta_1 \longrightarrow B_1} & \cdots & \overset{\Pi_n}{\Delta_n \longrightarrow B_n} & \overset{w(\Delta,\Pi)}{B_1, \ldots, B_n, \Gamma, \Delta \longrightarrow C}\end{array}}{\Delta_1, \ldots, \Delta_n, \Gamma, \Delta \longrightarrow C}\ mc\ .$$

As in the definition of the reduction relation, we shall arbitrarily distinguish Π_1 without loss of generality.

The only interesting case is that of $\exists\mathcal{R}/\exists\mathcal{L}$ shown below; the remaining cases follow easily from Definition 2.8 and Lemma 2.12.

Suppose Π_1 and Π are

$$\dfrac{\overset{\Pi'_1}{\Delta_1 \longrightarrow B'_1[t/x]}}{\Delta_1 \longrightarrow \exists x.B'_1}\ \exists\mathcal{R} \qquad \dfrac{\overset{\Pi'}{B'_1[y/x], B_2, \ldots, B_n, \Gamma \longrightarrow C}}{\exists x.B'_1, B_2, \ldots, B_n, \Gamma \longrightarrow C}\ \exists\mathcal{L}\ .$$

Then $w(\Delta, \Pi)$ is

$$\dfrac{\overset{w(\Delta,\Pi')}{B'_1[y/x], B_2, \ldots, B_n, \Gamma, \Delta \longrightarrow C}}{\exists x.B'_1, B_2, \ldots, B_n, \Gamma, \Delta \longrightarrow C}\ \exists\mathcal{L}\ ,$$

and $w(\Delta, \Xi)$ reduces to

$$\dfrac{\begin{array}{ccc}\overset{\Pi'_1}{\Delta_1 \longrightarrow B'_1[t/x]} & \left\{\overset{\Pi_i}{\Delta_i \longrightarrow B_i}\right\}_{i\in\{2..n\}} & \overset{w(\Delta,\Pi')[t/y]}{B'_1[t/x], B_2, \ldots, B_n, \Gamma, \Delta \longrightarrow C}\end{array}}{\Delta_1, \ldots, \Delta_n, \Gamma, \Delta \longrightarrow C}\ mc\ ,$$

where we choose y so that it is not free in Δ. On the other hand, Ξ reduces to Ξ'

$$\dfrac{\begin{array}{ccc}\overset{\Pi'_1}{\Delta_1 \longrightarrow B'_1[t/x]} & \left\{\overset{\Pi_i}{\Delta_i \longrightarrow B_i}\right\}_{i\in\{2..n\}} & \overset{\Pi'[t/y]}{B'_1[t/x], B_2, \ldots, B_n, \Gamma \longrightarrow C}\end{array}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc\ ,$$

and $w(\Delta, \Xi')$ is

$$\dfrac{\begin{array}{ccc}\overset{\Pi'_1}{\Delta_1 \longrightarrow B'_1[t/x]} & \left\{\overset{\Pi_i}{\Delta_i \longrightarrow B_i}\right\}_{i\in\{2..n\}} & \overset{w(\Delta,\Pi'[t/y])}{B'_1[t/x], B_2, \ldots, B_n, \Gamma, \Delta \longrightarrow C}\end{array}}{\Delta_1, \ldots, \Delta_n, \Gamma, \Delta \longrightarrow C}\ mc\ .$$

By Lemma 2.11, $w(\Delta, \Pi')[t/y]$ is the same derivation as $w(\Delta[t/y], \Pi'[t/y])$. Since y is not free in Δ, $\Delta[t/y] = \Delta$, so the reduct of $w(\Delta, \Xi)$ is $w(\Delta, \Xi')$. ∎

3.2 Normalizability and Reducibility

We now define two properties of derivations: normalizability and reducibility. Each of these properties implies that the derivation can be reduced to a cut-free derivation of the same end-sequent.

Definition 3.3 We define the set of *normalizable* derivations to be the smallest set that satisfies the following conditions:

1. If a derivation Π ends with a multicut, then it is normalizable if for every substitution θ there is a normalizable reduct of $\Pi\theta$.

2. If a derivation ends with any rule other than a multicut, then it is normalizable if the premise derivations are normalizable.

These clauses assert that a given derivation is normalizable provided certain (perhaps infinitely many) other derivations are normalizable. If we call these other derivations the predecessors of the given derivation, then a derivation is normalizable if and only if the tree of the derivation and its successive predecessors is well-founded. In this case, the well-founded tree is call the *normalization* of the derivation.

Since a normalization is well-founded, it has an associated induction principle: for any property P of derivations, if for every derivation Π in the normalization, P holds for every predecessor of Π implies that P holds for Π, then P holds for every derivation in the normalization.

Lemma 3.4 *If there is a normalizable derivation of a sequent, then there is a cut-free derivation of the sequent.*

Proof Let Π be a normalizable derivation of the sequent $\Gamma \longrightarrow B$. We show by induction on the normalization of Π that there is a cut-free derivation of $\Gamma \longrightarrow B$.

1. If Π ends with a multicut, then one of its reducts is one of its predecessors (by way of the empty substitution) and so is normalizable. But the reduct is also a derivation of $\Gamma \longrightarrow B$, so by the induction hypothesis this sequent has a cut-free derivation.

2. Suppose Π ends with a rule other than multicut. Since we are given that Π is normalizable, by definition the premise derivations are normalizable. These premise derivations are the predecessors of Π, so by the induction hypothesis there are cut-free derivations of the premises. Thus there is a cut-free derivation of $\Gamma \longrightarrow B$. ∎

Lemma 3.5 *If Π is a normalizable derivation, then for any substitution θ, $\Pi\theta$ is normalizable.*

Proof We prove this lemma by induction on the normalization of Π.

1. If Π ends with a multicut, then $\Pi\theta$ also ends with a multicut. For any substitution θ', some reduct of $\Pi\theta\theta'$ is a predecessor of Π and is thus normalizable. Therefore $\Pi\theta$ is normalizable.

2. Suppose Π ends with a rule other than multicut and has premise derivations $\{\Pi_i\}$. By Definition 2.5 each premise derivation in $\Pi\theta$ is either Π_i or $\Pi_i\theta$. Since Π is normalizable, Π_i is normalizable, and so by the induction hypothesis $\Pi_i\theta$ is also normalizable. Thus $\Pi\theta$ is normalizable. ∎

Lemma 3.6 *If Π is a normalizable derivation, then for any multiset Δ of formulas, $w(\Delta, \Pi)$ is normalizable.*

Proof We prove this lemma by induction on the normalization of Π.

1. Suppose Π ends with a multicut. To show that $w(\Delta, \Pi)$ is normalizable, we must show that for any substitution θ some reduct of $w(\Delta, \Pi)\theta$ is normalizable. Since Π is normalizable, it has as one of its predecessors a reduct Π' of $\Pi\theta$. By Lemmas 2.11 and 3.2, $w(\Delta, \Pi)\theta$ reduces to $w(\Delta\theta, \Pi')$. Π' is normalizable since it is a predecessor of Π, so by the induction hypothesis $w(\Delta\theta, \Pi')$ is normalizable.

2. Suppose Π ends with a rule other than multicut and has premise derivations $\{\Pi_i\}$. By Definition 2.8 each premise derivation of $w(\Delta, \Pi)$ is either Π_i or $w(\Delta', \Pi_i)$ for some multiset Δ' of formulas. Since Π is normalizable, Π_i is normalizable, and so by the induction hypothesis $w(\Delta', \Pi_i)$ is also normalizable. Thus all the premise derivations of $w(\Delta, \Pi)$ are normalizable, and so $w(\Delta, \Pi)$ is normalizable. ∎

We now define the property of reducibility for derivations. We do this by induction on the level of the derivation: in the definition of reducibility for derivations of level i we assume that reducibility is already defined for all levels $j < i$. (Recall from Definition 2.1 that the level of a derivation is defined to be the level of the consequent of its end-sequent.)

Definition 3.7 For any i, we define the set of *reducible* i-level derivations to be the smallest set of i-level derivations that satisfies the following conditions:

1. If a derivation Π ends with a multicut, then it is reducible if for every substitution θ there is a reducible reduct of $\Pi\theta$.

2. Suppose the derivation ends with the implication right rule

$$\frac{\begin{matrix}\Pi\\ B, \Gamma \longrightarrow C\end{matrix}}{\Gamma \longrightarrow B \supset C} \supset \mathcal{R} \ .$$

   Then the derivation is reducible if the premise derivation Π is reducible and, for every substitution θ, multiset Δ of formulas, and reducible derivation Π' of $\Delta \longrightarrow B\theta$, the derivation

$$\frac{\begin{matrix}\Pi' & \Pi\theta\\ \Delta \longrightarrow B\theta & B\theta, \Gamma\theta \longrightarrow C\theta\end{matrix}}{\Delta, \Gamma\theta \longrightarrow C\theta} \ mc$$

   is reducible.

3. If the derivation ends with the implication left rule or the *nat* left rule, then it is reducible if the right premise derivation is reducible and the other premise derivations are normalizable.

4. If the derivation ends with any other rule, then it is reducible if the premise derivations are reducible.

These clauses assert that a given derivation is reducible provided certain (perhaps infinitely many) other derivations are reducible. If we call these other derivations the predecessors of the given derivation, then a derivation is reducible only if the tree of the derivation and its successive predecessors is well-founded. In this case, the well-founded tree is call the *reduction* of the derivation.

In defining reducibility for a derivation of $\Gamma \longrightarrow B \supset C$ ending with $\supset \mathcal{R}$ we quantify over reducible derivations of $\Delta \longrightarrow B\theta$. This is legitimate since we are defining reducibility for a derivation having level $\max(\mathrm{lvl}(B)+1, \mathrm{lvl}(C))$, so the set of reducible derivations having level $\mathrm{lvl}(B\theta) = \mathrm{lvl}(B)$ is already defined. For a derivation ending with $\supset \mathcal{L}$ or $nat\mathcal{L}$, some premise derivations may have consequents with a higher level than that of the consequent of the conclusion. As a result, we cannot use the reducibility of those premise derivations to define the reducibility of the derivation as a whole, since the reducibility of the premise derivations may not yet be defined. Thus we use the weaker notion of normalizability for those premise derivations. Also observe that the consequent of the premise to the rule $def\mathcal{R}$ cannot have a higher level than the consequent of the conclusion because of the level restriction on definitional clauses. Finally, as with normalizations, reductions have associated induction principles.

Lemma 3.8 *If a derivation is reducible, then it is normalizable.*

Proof The proof of this lemma is a straightforward induction on the reduction of the given derivation. ∎

Lemma 3.9 *If* Π *is a reducible derivation, then for any substitution* θ, $\Pi\theta$ *is reducible.*

Proof We prove this lemma by induction on the reduction of Π.

1. If Π ends with a multicut, then $\Pi\theta$ also ends with a multicut. For any substitution θ', some reduct of $\Pi\theta\theta'$ is a predecessor of Π and is thus reducible. Therefore $\Pi\theta$ is reducible.

2. If Π ends with the implication right rule then Π and $\Pi\theta$ are

$$\dfrac{\begin{array}{c}\Pi' \\ B, \Gamma \longrightarrow C\end{array}}{\Gamma \longrightarrow B \supset C}\supset \mathcal{R} \qquad \dfrac{\begin{array}{c}\Pi'\theta \\ B\theta, \Gamma\theta \longrightarrow C\theta\end{array}}{\Gamma\theta \longrightarrow B\theta \supset C\theta}\supset \mathcal{R} \ .$$

$\Pi\theta$ is reducible if $\Pi'\theta$ is reducible and, for every substitution θ', multiset Δ, and reducible derivation Π'', the derivation Ξ

$$\dfrac{\begin{array}{cc}\Pi'' & \Pi'\theta\theta' \\ \Delta \longrightarrow B\theta\theta' & B\theta\theta', \Gamma\theta\theta' \longrightarrow C\theta\theta'\end{array}}{\Delta, \Gamma\theta\theta' \longrightarrow C\theta\theta'}\ mc$$

is also reducible. $\Pi'\theta$ is reducible by the induction hypothesis, and Ξ is reducible since it is a predecessor of Π. Therefore $\Pi\theta$ is reducible.

3. If Π ends with $\supset \mathcal{L}$ or $nat\mathcal{L}$, then the right premise derivation is reducible and the other premise derivations are normalizable. By Definition 2.5 each premise derivation in $\Pi\theta$ is obtained by applying θ to a premise derivation in Π. By the induction hypothesis the right premise derivation of $\Pi\theta$ is reducible, and by Lemma 3.5 the other premise derivations are normalizable. Thus $\Pi\theta$ is reducible.

4. Suppose Π ends with any other rule and has premise derivations $\{\Pi_i\}$. By Definition 2.5 each premise derivation in $\Pi\theta$ is either Π_i or $\Pi_i\theta$. Since Π is reducible, Π_i is reducible, and so by the induction hypothesis $\Pi_i\theta$ is also reducible. Thus $\Pi\theta$ is reducible. ∎

Lemma 3.10 *If Π is a reducible derivation, then for any multiset Δ of formulas, $w(\Delta, \Pi)$ is reducible.*

Proof We prove this lemma by induction on the reduction of Π.

1. Suppose Π ends with a multicut. To show that $w(\Delta, \Pi)$ is reducible, we must show that for any substitution θ some reduct of $w(\Delta, \Pi)\theta$ is reducible. Since Π is reducible, it has as one of its predecessors a reduct Π' of $\Pi\theta$. By Lemmas 2.11 and 3.2, $w(\Delta, \Pi)\theta$ reduces to $w(\Delta\theta, \Pi')$. Π' is reducible since it is a predecessor of Π, so by the induction hypothesis $w(\Delta\theta, \Pi')$ is reducible.

2. If Π ends with the implication right rule then Π and $w(\Delta, \Pi)$ are

$$\dfrac{\begin{array}{c}\Pi' \\ B, \Gamma \longrightarrow C\end{array}}{\Gamma \longrightarrow B \supset C} \supset \mathcal{R} \qquad \dfrac{\begin{array}{c}w(\Delta, \Pi') \\ B, \Gamma, \Delta \longrightarrow C\end{array}}{\Gamma, \Delta \longrightarrow B \supset C} \supset \mathcal{R}\ .$$

$w(\Delta, \Pi)$ is reducible if $w(\Delta, \Pi')$ is reducible and, for every substitution θ, multiset Δ', and reducible derivation Π'', the derivation Ξ

$$\dfrac{\begin{array}{cc}\Pi'' & w(\Delta,\Pi')\theta \\ \Delta' \longrightarrow B\theta & B\theta, \Gamma\theta, \Delta\theta \longrightarrow C\theta\end{array}}{\Delta', \Gamma\theta, \Delta\theta \longrightarrow C\theta}\ mc$$

is reducible. By Lemma 2.11 $w(\Delta, \Pi')\theta$ is the same derivation as $w(\Delta\theta, \Pi'\theta)$. Thus Ξ is the same as $w(\Delta\theta, \Xi')$, where Ξ' is

$$\dfrac{\begin{array}{cc}\Pi'' & \Pi'\theta \\ \Delta' \longrightarrow B\theta & B\theta, \Gamma\theta \longrightarrow C\theta\end{array}}{\Delta', \Gamma\theta \longrightarrow C\theta}\ mc \quad .$$

Π' and Ξ' are predecessors of Π, so they are reducible. Thus by the induction hypothesis $w(\Delta, \Pi')$ and $w(\Delta\theta, \Xi')$ are reducible, so $w(\Delta, \Pi)$ is reducible.

3. (a) If Π ends with the implication left rule

$$\dfrac{\begin{array}{cc}\Pi_1 & \Pi_2 \\ \Gamma \longrightarrow B & C, \Gamma \longrightarrow D\end{array}}{B \supset C, \Gamma \longrightarrow D}\ \supset \mathcal{L} \quad ,$$

then the derivation Π_1 is normalizable and the derivation Π_2 is reducible. By Lemma 3.6 $w(\Delta, \Pi_1)$ is normalizable, and by the induction hypothesis $w(\Delta, \Pi_2)$ is reducible. Thus $w(\Delta, \Pi)$ is reducible.

(b) If Π ends with the $nat\mathcal{L}$ rule

$$\dfrac{\begin{array}{ccc}\Pi_1 & \Pi_2 & \Pi_3 \\ \longrightarrow C\,z & C\,j \longrightarrow C\,(s\,j) & C\,I, \Gamma \longrightarrow B\end{array}}{nat\ I, \Gamma \longrightarrow B}\ nat\mathcal{L} \quad ,$$

then the derivations Π_1 and Π_2 are normalizable and the derivation Π_3 is reducible. $w(\Delta, \Pi)$ is

$$\dfrac{\begin{array}{ccc}\Pi_1 & \Pi_2 & w(\Delta,\Pi_3) \\ \longrightarrow C\,z & C\,j \longrightarrow C\,(s\,j) & C\,I, \Gamma, \Delta \longrightarrow B\end{array}}{nat\ I, \Gamma, \Delta \longrightarrow B}\ nat\mathcal{L} \quad .$$

By the induction hypothesis $w(\Delta, \Pi_3)$ is reducible, so $w(\Delta, \Pi)$ is reducible.

4. If Π ends with any other rule, then the premise derivations are reducible. By Definition 2.8 each premise derivation of $w(\Delta, \Pi)$ is $w(\Delta', \Pi_i)$ for some premise derivation Π_i in Π and multiset Δ' of formulas. $w(\Delta', \Pi_i)$ is reducible by the induction hypothesis, so $w(\Delta, \Pi)$ is reducible. ∎

3.3 Cut-Elimination

In the previous section we proved that every reducible derivation is normalizable and that every normalizable derivation can be reduced to a cut-free derivation of the same end-sequent. In this section we show that every $FO\lambda^{\Delta\mathbb{N}}$ derivation is reducible, and thus every derivable sequent can be derived without the cut rule. The consistency of $FO\lambda^{\Delta\mathbb{N}}$ is then a simple corollary of the cut-elimination theorem.

Lemma 3.11 *For any derivation* Π *of* $B_1, \ldots, B_n, \Gamma \longrightarrow C$ *and reducible derivations* $\Pi_1, \ldots, \Pi_n$ *of* $\Delta_1 \longrightarrow B_1, \ldots, \Delta_n \longrightarrow B_n$ *($n \geq 0$), the derivation* Ξ

$$\dfrac{\overset{\Pi_1}{\Delta_1 \longrightarrow B_1} \quad \cdots \quad \overset{\Pi_n}{\Delta_n \longrightarrow B_n} \quad \overset{\Pi}{B_1, \ldots, B_n, \Gamma \longrightarrow C}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C}\ mc$$

is reducible.

Proof The proof is by induction on $\mathrm{ht}(\Pi)$, with subordinate inductions on n and on the reductions of $\Pi_1, \ldots, \Pi_n$. The proof does not rely on the order of the inductions on reductions. Thus when we need to distinguish one of the Π_i, we shall refer to it as Π_1 without loss of generality.

The derivation Ξ is reducible if for every substitution θ some reduct of $\Xi\theta$ is reducible. If $n = 0$, then $\Xi\theta$ reduces to $\Pi\theta$. By Lemma 3.9 it suffices to show that Π is reducible. This is proved by a case analysis of the last rule in Π. For each case, the result follows easily from the outer induction hypothesis and Definition 3.7. The $\supset \mathcal{R}$ case requires that substitution for variables doesn't increase the measure of a derivation (Lemma 2.7). In the cases for $\supset \mathcal{L}$ and $nat\mathcal{L}$ we need the additional information that reducibility implies normalizability (Lemma 3.8).

For $n > 0$ we proceed with a case analysis of the reduction rules that apply to Ξ (and thus to $\Xi\theta$) to show that in fact every reduct of $\Xi\theta$ is reducible. Most cases follow easily from the induction hypothesis, Definition 3.7, and Lemmas 2.7, 2.10, 3.5, 3.6, 3.8, 3.9, and 3.10. We show the interesting cases below.

$\supset\mathcal{R}/\supset\mathcal{L}$: Π_1 and Π are

$$\dfrac{\begin{matrix}\Pi_1'\\ B_1', \Delta_1 \longrightarrow B_1''\end{matrix}}{\Delta_1 \longrightarrow B_1' \supset B_1''}\supset\mathcal{R} \qquad \dfrac{\begin{matrix}\Pi'\\ B_2, \ldots, B_n, \Gamma \longrightarrow B_1'\end{matrix} \quad \begin{matrix}\Pi''\\ B_1'', B_2, \ldots, B_n, \Gamma \longrightarrow C\end{matrix}}{B_1' \supset B_1'', B_2, \ldots, B_n, \Gamma \longrightarrow C}\supset\mathcal{L} \ .$$

Recall that substitution for variables preserves reducibility (Lemma 3.9) and does not increase the measure of a derivation (Lemma 2.7). Thus the derivation Ξ_1

$$\dfrac{\begin{matrix}\Pi_2\theta\\ \Delta_2\theta \longrightarrow B_2\theta\end{matrix} \quad \cdots \quad \begin{matrix}\Pi_n\theta\\ \Delta_n\theta \longrightarrow B_n\theta\end{matrix} \quad \begin{matrix}\Pi'\theta\\ B_2\theta, \ldots, B_n\theta, \Gamma\theta \longrightarrow B_1'\theta\end{matrix}}{\Delta_2\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow B_1'\theta}\, mc$$

is reducible by the outer induction hypothesis. Since we are given that Π_1 is reducible, by Definition 3.7 the derivation Ξ_2

$$\dfrac{\begin{matrix}\Xi_1\\ \Delta_2\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow B_1'\theta\end{matrix} \quad \begin{matrix}\Pi_1'\theta\\ B_1'\theta, \Delta_1\theta \longrightarrow B_1''\theta\end{matrix}}{\Delta_1\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow B_1''\theta}\, mc$$

is reducible. Therefore the derivation

$$\dfrac{\dfrac{\begin{matrix}\Xi_2\\ \ldots \longrightarrow B_1''\theta\end{matrix} \quad \left\{\begin{matrix}\Pi_i\theta\\ \Delta_i\theta \longrightarrow B_i\theta\end{matrix}\right\}_{i\in\{2..n\}} \quad \begin{matrix}\Pi''\theta\\ B_1''\theta, \{B_i\theta\}_{i\in\{2..n\}}, \Gamma\theta \longrightarrow C\theta\end{matrix}}{\Delta_1\theta, \ldots, \Delta_n\theta, \Gamma\theta, \Delta_2\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow C\theta}\, mc}{\Delta_1\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow C\theta}\, c\mathcal{L} \ ,$$

which is the reduct of $\Xi\theta$, is reducible by the outer induction hypothesis and Definition 3.7.

$nat\mathcal{R}/nat\mathcal{L}$: Π_1 is

$$\dfrac{\begin{matrix}\Pi_1'\\ \Delta_1 \longrightarrow nat\ I\end{matrix}}{\Delta_1 \longrightarrow nat\ (s\ I)}\, nat\mathcal{R}$$

and Π is

$$\dfrac{\begin{matrix}\Pi'\\ \longrightarrow D\, z\end{matrix} \quad \begin{matrix}\Pi''\\ D\, j \longrightarrow D\, (s\ j)\end{matrix} \quad \begin{matrix}\Pi'''\\ D\, (s\ I), B_2, \ldots, B_n, \Gamma \longrightarrow C\end{matrix}}{nat\ (s\ I), B_2, \ldots, B_n, \Gamma \longrightarrow C}\, nat\mathcal{L} \ .$$

Consider the derivation Ξ_1

$$\dfrac{\dfrac{\Pi_1'}{\Delta_1 \longrightarrow nat\ I} \quad \dfrac{\dfrac{\Pi'}{\longrightarrow D\,z} \quad \dfrac{\Pi''}{D\,j \longrightarrow D\,(s\ j)} \quad \dfrac{}{D\,I \longrightarrow D\,I}\ init}{nat\ I \longrightarrow D\,I}\ nat\mathcal{L}}{\Delta_1 \longrightarrow D\,I}\ mc \quad .$$

Since the measure of the right premise derivation is no larger than $\mathrm{ht}(\Pi)$, Ξ_1 is reducible by induction on the reduction of Π_1 (Π_1' is a predecessor of Π_1). Again recall that substitution for variables preserves reducibility (Lemma 3.9) and does not increase the measure of a derivation (Lemma 2.7). The derivation Ξ_2

$$\dfrac{\dfrac{\Xi_1\theta}{\Delta_1\theta \longrightarrow D\theta\,I\theta} \quad \dfrac{\Pi''\theta[I\theta/j]}{D\theta\,I\theta \longrightarrow D\theta\,(s\ I\theta)}}{\Delta_1\theta \longrightarrow D\theta\,(s\ I\theta)}\ mc$$

is then reducible by the outer induction hypothesis. Therefore the derivation

$$\dfrac{\dfrac{\Xi_2}{\Delta_1\theta \longrightarrow D\theta\,(s\ I\theta)} \quad \left\{\dfrac{\Pi_i\theta}{\Delta_i\theta \longrightarrow B_i\theta}\right\}_{i\in\{2..n\}} \quad \dfrac{\Pi'''\theta}{\ldots \longrightarrow C\theta}}{\Delta_1\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow C\theta}\ mc \quad ,$$

which is the reduct of $\Xi\theta$, is reducible by the outer induction hypothesis.

def$\mathcal{L}$/ $\circ\,\mathcal{L}$**:** Π_1 and $\Pi_1\theta$ are

$$\dfrac{\left\{\dfrac{\Pi_1^{\rho,\sigma,D}}{D\sigma, \Delta_1'\rho \longrightarrow B_1\rho}\right\}}{A, \Delta_1' \longrightarrow B_1}\ def\mathcal{L} \qquad \dfrac{\left\{\dfrac{\Pi_1^{\theta\circ\rho',\sigma',D}}{D\sigma', \Delta_1'\theta\rho' \longrightarrow B_1\theta\rho'}\right\}}{A\theta, \Delta_1'\theta \longrightarrow B_1\theta}\ def\mathcal{L} \quad .$$

The derivation $\Xi^{\rho',\sigma',D}$

$$\dfrac{\dfrac{\Pi_1^{\theta\circ\rho',\sigma',D}}{D\sigma', \Delta_1'\theta\rho' \longrightarrow B_1\theta\rho'} \quad \left\{\dfrac{\Pi_i\theta\rho'}{\Delta_i\theta\rho' \longrightarrow B_i\theta\rho'}\right\}_{i\in\{2..n\}} \quad \dfrac{\Pi\theta\rho'}{\ldots \longrightarrow C\theta\rho'}}{D\sigma', \Delta_1'\theta\rho', \Delta_2\theta\rho', \ldots, \Delta_n\theta\rho', \Gamma\theta\rho' \longrightarrow C\theta\rho'}\ mc \quad .$$

is reducible by Lemmas 2.7 and 3.9 and induction on the reduction of Π_1 ($\Pi_1^{\theta\circ\rho',\sigma',D}$ is a predecessor of Π_1). Therefore the derivation

$$\dfrac{\left\{\dfrac{\Xi^{\rho',\sigma',D}}{D\sigma', \Delta_1'\theta\rho', \Delta_2\theta\rho', \ldots, \Delta_n\theta\rho', \Gamma\theta\rho' \longrightarrow C\theta\rho'}\right\}}{A\theta, \Delta_1'\theta, \Delta_2\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow C\theta}\ def\mathcal{L} \quad ,$$

which is the reduct of $\Xi\theta$, is reducible by Definition 3.7.

$-/\supset\mathcal{R}$**:** Ξ has the form

$$\dfrac{\overset{\Pi_1}{\Delta_1 \longrightarrow B_1} \quad \cdots \quad \overset{\Pi_n}{\Delta_n \longrightarrow B_n} \quad \dfrac{\overset{\Pi'}{C', B_1, \ldots, B_n, \Gamma \longrightarrow C''}}{B_1, \ldots, B_n, \Gamma \longrightarrow C' \supset C''} \supset \mathcal{R}}{\Delta_1, \ldots, \Delta_n, \Gamma \longrightarrow C' \supset C''} \; mc \quad .$$

Once again recall that substitution for variables preserves reducibility (Lemma 3.9) and does not increase the measure of a derivation (Lemma 2.7). The derivation Ξ_1

$$\dfrac{\overset{\Pi_1\theta}{\Delta_1\theta \longrightarrow B_1\theta} \quad \cdots \quad \overset{\Pi_n\theta}{\Delta_n\theta \longrightarrow B_n\theta} \quad \overset{\Pi'\theta}{C'\theta, B_1\theta, \ldots, B_n\theta, \Gamma\theta \longrightarrow C''\theta}}{C'\theta, \Delta_1\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow C''\theta} \; mc$$

is reducible by the outer induction hypothesis. For any substitutions θ' and θ'' and reducible derivation Ξ', the derivation

$$\dfrac{\overset{\Xi'\theta''}{(\Delta' \longrightarrow C'\theta\theta')\theta''} \quad \left\{\overset{\Pi_i\theta\theta'\theta''}{(\Delta_i \longrightarrow B_i)\theta\theta'\theta''}\right\}_{i\in\{1..n\}} \quad \overset{\Pi'\theta\theta'\theta''}{(\ldots \longrightarrow C'')\theta\theta'\theta''}}{\Delta'\theta'', \Delta_1\theta\theta'\theta'', \ldots, \Delta_n\theta\theta'\theta'', \Gamma\theta\theta'\theta'' \longrightarrow C''\theta\theta'\theta''} \; mc$$

is reducible by the outer induction hypothesis. This is a reduct of $\Xi_2\theta''$, where Ξ_2 is

$$\dfrac{\overset{\Xi'}{\Delta' \longrightarrow C'\theta\theta'} \quad \overset{\Xi_1\theta'}{C'\theta\theta', \Delta_1\theta\theta', \ldots, \Delta_n\theta\theta', \Gamma\theta\theta' \longrightarrow C''\theta\theta'}}{\Delta', \Delta_1\theta\theta', \ldots, \Delta_n\theta\theta', \Gamma\theta\theta' \longrightarrow C''\theta\theta'} \; mc \quad .$$

Since a reduct of $\Xi_2\theta''$ is reducible for every θ'', by Definition 3.7 Ξ_2 is reducible. Since Ξ_1 is reducible and Ξ_2 is reducible for every substitution θ' and reducible derivation Ξ', by Definition 3.7

$$\dfrac{\overset{\Xi_1}{C'\theta, \Delta_1\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow C''\theta}}{\Delta_1\theta, \ldots, \Delta_n\theta, \Gamma\theta \longrightarrow C'\theta \supset C''\theta} \supset \mathcal{R}$$

is reducible. This last derivation is the reduct of $\Xi\theta$ by the current reduction rule. ∎

Corollary 3.12 *Every derivation is reducible.*

Proof This result follows immediately from Lemma 3.11 with $n = 0$. ∎

Theorem 3.13 *If a sequent is derivable, then there is a cut-free derivation of the sequent.*

Proof This result follows immediately from Corollary 3.12, Lemma 3.8, and Lemma 3.4. ∎

Since there is no right rule for $\bot$, there is no cut-free derivation of $\longrightarrow \bot$. Thus consistency is a simple corollary of cut-elimination.

Corollary 3.14 *There is no $FO\lambda^{\Delta\mathbb{N}}$ derivation of the sequent $\longrightarrow \bot$.*

3.4 Related Work

The logic $FO\lambda^{\Delta\mathbb{N}}$ is related to Schroeder-Heister's "logics with definitional reflection" [49]. He proved cut-elimination for two logics: the first without contraction but allowing arbitrary implications in definitions, the second with contraction but only implication-free definitions. He also showed a counter-example to cut-elimination for the logic with both contraction and definitions with arbitrary implications, but conjectured that cut-elimination should hold if the definitions were stratified (as we accomplish in $FO\lambda^{\Delta\mathbb{N}}$ through the level restriction). The proof presented in this chapter clearly establishes that Schroeder-Heister's conjecture is true.

However, there are significant differences between Schroeder-Heister's logics and ours. The first is that $FO\lambda^{\Delta\mathbb{N}}$ uses a stronger version of the left rule for definitions; Schroeder-Heister has extended his cut-elimination results to logics with this stronger rule [50]. More significantly, Schroeder-Heister has no induction rules in his logics. Because of the presence of the $nat\mathcal{L}$ rule in $FO\lambda^{\Delta\mathbb{N}}$, Schroeder-Heister's cut-elimination proofs do not extend to our setting.

The proof of cut-elimination presented in this chapter is patterned after Martin-Löf's normalization proof for a natural deduction system with iterated inductive definitions [28]. Our work can be viewed as an extension of his to the sequent calculus setting: our rules for definitions and natural numbers roughly correspond to his introduction and elimination rules for inductively defined predicates.

Chapter 4

Reasoning about Transition Systems in $FO\lambda^{\Delta\mathbb{N}}$

Cut-free sequent calculus proofs have been successfully used to encode the operational semantics of a wide range of computational systems. For example, the evaluation of functional programming languages and of their abstract machines have been specified in intuitionistic logic [4, 21, 22]; imperative and concurrency features have been modeled using linear logic programming languages [5, 6, 14, 36, 37]; and the sequential and concurrent (pipe-line) semantics of a RISC processor have also been specified in linear logic [6]. But we would like to extend our use of logic beyond specifying computation to reasoning about it. One of the interesting and important properties of a computational system is the set of equivalences among terms that the system suggests. Bisimulation [42] is a natural and widely-used equivalence relation that has grown out of the study of concurrency. Informally, two terms are bisimilar if every computational step that applies to either of the terms also applies to the other, and applying the step to both terms will result in two new terms that are also bisimilar. This requirement that every possible step in the computation of a term be matched in the computation of another term cannot be expressed with the natural representation of the computation system as a logical theory. When the system is encoded as a definition, however, such a requirement is easily captured. In this chapter, we show that in this way the definition facility of $FO\lambda^{\Delta\mathbb{N}}$ makes it possible to go beyond operational

semantics and both encode and reason about concepts such as simulation and bisimulation. This chapter presents work previously reported in [32], coauthored with Miller and Palamidessi.

4.1 Background

The triple $\mathcal{T} = (\Lambda, S, \delta)$ is an *abstract transition system* (*ats*) if Λ is a non-empty set of *actions*, S is a non-empty set of *states*, and $\delta \subseteq S \times \Lambda \times S$ (Λ and S are assumed to be disjoint). We write $p \xrightarrow{a} q$ if $(p, a, q) \in \delta$. For $w \in \Lambda^*$ we write $p \stackrel{w}{\Longrightarrow} q$ to mean that p makes a transition to q along a path of actions given by w. More formally, this relation is defined by induction on the length of w: thus $p \stackrel{\epsilon}{\Longrightarrow} p$ (where ϵ is the empty string) and if $p \xrightarrow{a} r$ and $r \stackrel{w}{\Longrightarrow} q$ then $p \stackrel{aw}{\Longrightarrow} q$. For a state p, define $\langle\langle p \rangle\rangle = \{(a, q) \mid (p, a, q) \in \delta\}$. The ats $\mathcal{T}$ is *finitely branching* if, for each p, the set $\langle\langle p \rangle\rangle$ is finite. $\mathcal{T}$ is *noetherian* if it contains no infinite paths. In a noetherian ats we can define the *measure* of a state p, denoted by $\mathrm{meas}(p)$, as the ordinal number given by

$$\mathrm{meas}(p) = \mathrm{lub}(\{\mathrm{meas}(q) + 1 \mid p \xrightarrow{a} q \text{ for some } a\}) \ ,$$

where we assume $\mathrm{lub}(\emptyset) = 0$. If the ats is finitely branching then all its states have finite measure.

The notions of simulation and bisimulation provide important judgments on pairs of states in an abstract transition system. A relation $\mathcal{R}$ is a *simulation* between p and q if and only if for every transition $p \xrightarrow{a} p'$, there exists a transition $q \xrightarrow{a} q'$, such that $p' \mathcal{R} q'$. The largest such relation is written $\sqsubseteq$; that is, $p \sqsubseteq q$ (read "q simulates p") if and only if there exists a simulation $\mathcal{R}$ such that $p \mathcal{R} q$. If $p \sqsubseteq q$ and $q \sqsubseteq p$ both hold, then p and q are *similar*.

A relation $\mathcal{R}$ is a *bisimulation* between p and q if and only if for every transition $p \xrightarrow{a} p'$, there exists a transition $q \xrightarrow{a} q'$ such that $p' \mathcal{R} q'$, and for every transition $q \xrightarrow{a} q'$, there exists a transition $p \xrightarrow{a} p'$ such that $q' \mathcal{R} p'$. The largest such relation is called the *bisimulation equivalence* and is denoted by $\equiv$; that is, $p \equiv q$ (read "p is *bisimilar* to q") if and only if there exists a bisimulation $\mathcal{R}$ such that $p \mathcal{R} q$. It is well-known that bisimilarity implies similarity but not vice-versa.

Table 4.1: CCS transition rules

$$\frac{}{a.p \xrightarrow{a} p} \qquad \frac{p \xrightarrow{a} q}{p \mid r \xrightarrow{a} q \mid r} \qquad \frac{p \xrightarrow{a} q}{r \mid p \xrightarrow{a} r \mid q} \qquad \frac{p \xrightarrow{a} r \quad q \xrightarrow{\bar{a}} s}{p \mid q \xrightarrow{\tau} r \mid s}$$

$$\frac{p \xrightarrow{a} q}{p + r \xrightarrow{a} q} \qquad \frac{r \xrightarrow{a} q}{p + r \xrightarrow{a} q} \qquad \frac{p[\mu_x p/x] \xrightarrow{a} q}{\mu_x p \xrightarrow{a} q}$$

To illustrate our results, we will consider throughout the chapter a more concrete example of an abstract transition system: the concurrent language CCS [40]. For convenience, we ignore the renaming and hiding combinators, and concentrate on the sublanguage described by the grammar

$$p \ ::= \ 0 \ \mid \ a.p \ \mid \ p + p \ \mid \ p \mid p \ \mid \ \mu_x p \ ,$$

where a ranges over an arbitrary set of actions $\mathcal{A}$, the set of the complementary actions $\bar{\mathcal{A}}$, and $\{\tau\}$. The intended meaning of these symbols is as follows: 0 represents the inactive process, $a.p$ represents a process prefixed by the action a, $+$ and $\mid$ are choice and parallel composition, respectively, and μ_x is the least fixed point operator, providing recursion. The operational semantics of CCS is specified by the transition rules in Table 4.1.

CCS can be seen as an abstract transition system where $\Lambda = \mathcal{A} \cup \bar{\mathcal{A}} \cup \{\tau\}$, S is the set of all expressions denoting CCS expressions, and δ is the set of transitions which are derivable by the rules above. A *finite CCS process* is a CCS process that does not contain μ. If S is restricted to the set of all finite CCS processes, then the resulting ats is noetherian.

4.2 Encoding Abstract Transition Systems

In this section we give an encoding of abstract transition systems in the logic $FO\lambda^{\Delta\mathbb{IN}}$. Let (Λ, S, δ) be an ats, and let the primitive types σ and α denote elements of S and Λ, respectively. Let $one : \sigma \to \alpha \to \sigma \to o$ be a predicate of three arguments denoting the one step transition relation and let the definition $\mathcal{D}(ats(\delta))$ contain the clause $one\ p\ a\ q \stackrel{\triangle}{=} \top$ for every $(p, a, q) \in \delta$. This definition is essentially a table. We also need the following

definition $\mathcal{D}(path)$:

$$\begin{array}{rcl} multi\ P\ nil\ P & \triangleq & \top \\ multi\ P\ (A :: W)\ Q & \triangleq & \exists r(one\ P\ A\ r \wedge multi\ r\ W\ Q) \quad . \end{array}$$

The predicates *one* and *multi* are assumed to have level 0. Here, members of Λ^* are represented as terms of type $lst(\alpha)$ using $nil : lst(\alpha)$ for the empty list and :: of type $\alpha \to lst(\alpha) \to lst(\alpha)$ for the list constructor. We now prove the adequacy of this encoding.

Proposition 4.1 *Let (Λ, S, δ) be an ats. Then $p \stackrel{w}{\Longrightarrow} q$ if and only if $(multi\ p\ w\ q)$ is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using definition $\mathcal{D}(ats(\delta)) \cup \mathcal{D}(path)$.*

Proof We prove this proposition by induction with respect to the length l of the path w. The key idea is to show that the transition along the path matches faithfully a certain sequence of inference rules in the derivation.

If $l = 0$, then $p \stackrel{\epsilon}{\Longrightarrow} p$ holds by definition, and the proposition $multi\ p\ nil\ p$ is derivable:

$$\dfrac{\dfrac{}{\longrightarrow \top}\ \top\mathcal{R}}{\longrightarrow multi\ p\ nil\ p}\ def\mathcal{R} \quad .$$

If $l > 0$, consider the path $p \stackrel{a}{\longrightarrow} r \stackrel{w'}{\Longrightarrow} q$. Observe that a derivation of the sequent $\longrightarrow multi\ p\ (a :: w')\ q$ must end in the following way, since at each point no other rule applies:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\longrightarrow \top}\ \top\mathcal{R}}{\longrightarrow one\ p\ a\ r}\ def\mathcal{R} \quad \longrightarrow multi\ r\ w'\ q}{\longrightarrow one\ p\ a\ r \wedge multi\ r\ w'\ q}\ \wedge\mathcal{R}}{\longrightarrow \exists r'(one\ p\ a\ r' \wedge multi\ r\ w'\ q)}\ \exists\mathcal{R}}{\longrightarrow multi\ p\ (a :: w')\ q}\ def\mathcal{R} \quad .$$

By the construction of $\mathcal{D}(ats(\delta))$, the sequent $\longrightarrow one\ p\ a\ r$ is derivable if and only if $p \stackrel{a}{\Longrightarrow} r$, and by the inductive hypothesis, $\longrightarrow multi\ r\ w'\ q$ is derivable if and only if $r \stackrel{w'}{\Longrightarrow} q$. ∎

The encoding $\mathcal{D}(ats(\delta))$ is based on an extensional description of δ, hence the definition will be infinite if δ is infinite. In specific transition systems the transition relation might be described intentionally. This is the case for CCS, whose transitions can be encoded as

the following definition $\mathcal{D}(ccs(\mathcal{A}))$ for any non-empty set $\mathcal{A}$ of actions:

$$\begin{array}{rcl}
one\ A.P\ A\ P & \triangleq & \top \\
one\ (P \mid R)\ A\ (Q \mid R) & \triangleq & one\ P\ A\ Q \\
one\ (R \mid P)\ A\ (R \mid Q) & \triangleq & one\ P\ A\ Q \\
one\ (P + R)\ A\ Q & \triangleq & one\ P\ A\ Q \\
one\ (P + R)\ A\ Q & \triangleq & one\ R\ A\ Q \\
one\ \mu M\ A\ Q & \triangleq & one\ (M\ \mu M)\ A\ Q \\
one\ (P \mid Q)\ \tau\ (R \mid S) & \triangleq & \exists a \exists b (comp\ a\ b \wedge one\ P\ a\ R \wedge one\ Q\ b\ S)] \quad ,
\end{array}$$

plus the clauses $comp\ a\ \bar{a} \triangleq \top$ and $comp\ \bar{a}\ a \triangleq \top$ for every $a \in \mathcal{A}$. We assume $\tau : \alpha \notin \mathcal{A}$.

Observe that we are using meta-level λ-abstraction to encode $\mu_x P$: such a term is represented as μM, where M is meant to be the abstraction $\lambda x.P$. Thus the term $P[\mu_x P/x]$ can be represented simply by $M(\mu M)$ without introducing an explicit notion of substitution (β-conversion in the meta-logic can perform substitution for us).

The following result shows that CCS transitions are completely described by logical derivability in $\mathcal{D}(ccs(\mathcal{A}))$.

Proposition 4.2 *The CCS transition $p \xrightarrow{a} q$ holds if and only if the formula $(one\ p\ a\ q)$ is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using definition $\mathcal{D}(ccs(\mathcal{A}))$. The transition $p \xRightarrow{w} q$ holds if and only if $(multi\ p\ w\ q)$ is derivable using definition $\mathcal{D}(ccs(\mathcal{A})) \cup \mathcal{D}(path)$.*

This theorem can be proved by simple structural induction by showing that derivations using the inference rules in Table 4.1 for CCS are essentially identical to $FO\lambda^{\Delta\mathbb{N}}$ derivations over the corresponding clauses in the definition $ccs(\mathcal{A})$. As in the proof of Proposition 4.1, the $FO\lambda^{\Delta\mathbb{N}}$ derivations involve only right introduction rules.

4.3 Encoding Simulation and Bisimulation

4.3.1 Finite Behavior

We now consider encodings of simulation and bisimulation relations between states in a transition system. If we represent the transition step by the predicate *one* (as in the

definitions $\mathcal{D}(ats(\delta))$ and $\mathcal{D}(ccs(\mathcal{A}))$), then it is possible to characterize simulation and bisimulation as predicates *sim* and *bisim* given by the following definition $\mathcal{D}(sims)$:

$$\begin{aligned} sim\ P\ Q \ &\triangleq\ \forall a \forall p' (one\ P\ a\ p' \supset \\ &\qquad \exists q' (one\ Q\ a\ q' \wedge sim\ p'\ q')) \end{aligned}$$

$$\begin{aligned} bisim\ P\ Q \ &\triangleq\ \forall a \forall p' (one\ P\ a\ p' \supset \\ &\qquad \exists q' (one\ Q\ a\ q' \wedge bisim\ p'\ q')) \wedge \\ &\quad \forall a \forall q' (one\ Q\ a\ q' \supset \\ &\qquad \exists p' (one\ P\ a\ p' \wedge bisim\ q'\ p')) \quad . \end{aligned}$$

Since the level of *one* is 0, we need to assign to both *sim* and *bisim* the level 1 (or higher).

We proceed now to prove the correctness of these encodings. To do so, we introduce two new items. First, given a fixed ats (Λ, S, δ) and a pair $(p, q) \in S \times S$, a *premise set for* (p, q) is a set $\mathcal{P} \subseteq S \times S$ such that for every $a \in \Lambda$ and $p' \in S$ such that $(a, p') \in \langle\langle p \rangle\rangle$ there exists a $q' \in S$ such that $(a, q') \in \langle\langle q \rangle\rangle$ and $(p', q') \in \mathcal{P}$. Premise sets need not exist, but if there is a simulation $\mathcal{R}$ that contains (p, q) then there is a premise set $\mathcal{P}$ for that pair such that $\mathcal{P} \subset \mathcal{R}$. We can restrict premise sets to be minimal, although this is not strictly necessary. Second, we introduce the following inference rules:

$$\frac{\{\longrightarrow sim\ p'\ q' \mid (p', q') \in \mathcal{P}\}}{\longrightarrow sim\ p\ q}\ SIM$$

$$\frac{\{\longrightarrow bisim\ p'\ q' \mid (p', q') \in \mathcal{P}\} \cup \{\longrightarrow bisim\ q'\ p' \mid (q', p') \in \mathcal{Q}\}}{\longrightarrow bisim\ p\ q}\ BISIM\ ,$$

where $\mathcal{P}$ is a premise set for (p, q) and $\mathcal{Q}$ is a premise set for (q, p). Notice that this rule is finitary if the ats is finitely branching. In the case of CCS, one condition which guarantees this property is that recursion variables in bodies of μ-terms only occur prefixed.

Let $\vdash_{SIM} \Delta \longrightarrow C$ (respectively, $\vdash_{BISIM} \Delta \longrightarrow C$) denote the proposition that the sequent $\Delta \longrightarrow C$ can be derived using only the SIM (respectively, $BISIM$) inference rule. The following lemma shows that these proof systems are a correct and complete representation of our encodings.

Lemma 4.3 *Let (Λ, S, δ) be an ats, let p and q be members of S, and let $\vdash$ denote derivability in $FO\lambda^{\Delta\mathbb{N}}$ using definition $\mathcal{D}(ats(\delta)) \cup \mathcal{D}(sims)$. Then*

- $\vdash sim\ p\ q$ *if and only if* $\vdash_{SIM} sim\ p\ q$, *and*
- $\vdash bisim\ p\ q$ *if and only if* $\vdash_{BISIM} bisim\ p\ q$.

The same holds if we replace $\mathcal{D}(ats(\delta))$ *by* $\mathcal{D}(ccs(\mathcal{A}))$.

Proof We outline the proof of the first case; the second can be done similarly. Consider a derivation of the sequent $\longrightarrow sim\ p\ q$. This is derivable only by a $def\mathcal{R}$ rule using $\mathcal{D}(sims)$, and thus the sequents

$$\longrightarrow \forall a \forall p'(one\ p\ a\ p' \supset \exists q'(one\ q\ a\ q' \wedge sim\ p'\ q'))$$

and

$$one\ p\ a\ p' \longrightarrow \exists q'(one\ q\ a\ q' \wedge sim\ p'\ q')$$

must be derivable. If this latter sequent is derivable, there is a derivation of it ending with $def\mathcal{L}$, and thus the sequent

$$\longrightarrow \exists q'(one\ q\ a_0\ q' \wedge sim\ p'_0\ q')$$

must be derivable, where the pair (a_0, p'_0) ranges over $\langle\langle p \rangle\rangle$. This sequent is derivable only if the quantifier $\exists q'$ is instantiated with q'_0 where $(a_0, q'_0) \in \langle\langle q \rangle\rangle$. Let $\mathcal{P}$ be the premise set arising from collecting together all pairs (p'_0, q'_0) for such values p'_0 and q'_0. Thus, our original sequent is derivable if and only if for every $(p'_0, q'_0) \in \mathcal{P}$ the sequent $\longrightarrow sim\ p'_0\ q'_0$ is derivable. The other direction follows by reversing these reasoning steps. ∎

We can now use this lemma to prove the correctness of our encodings of simulation and bisimulation.

Theorem 4.4 *Let* (Λ, S, δ) *be a noetherian ats, let* p *and* q *be members of* S, *and let* $\vdash$ *denote derivability in* $FO\lambda^{\Delta\mathbb{N}}$ *using definition* $\mathcal{D}(ats(\delta)) \cup \mathcal{D}(sims)$. *Then*

- $\vdash sim\ p\ q$ *if and only if* $p \sqsubseteq q$, *and*
- $\vdash bisim\ p\ q$ *if and only if* $p \equiv q$.

Proof Again we only show the proof for simulation; the proof for bisimulation is similar.

Given Lemma 4.3, we need only show that $\vdash_{SIM} p \longrightarrow q$ if and only if q simulates p. First, assume that the sequent $p \longrightarrow q$ has a derivation that contains only the SIM inference rule. Let $\mathcal{R}$ be the set of all pairs (r, s) such that the sequent $r \longrightarrow s$ has an occurrence in that derivation. It is an easy matter to verify that $\mathcal{R}$ is a simulation.

Conversely, assume that q simulates p. Thus there is a simulation $\mathcal{R}$ such that $p\mathcal{R}q$. The proof is by induction on the measure of p, meas(p). Since $p\mathcal{R}q$, there is a premise set $\mathcal{P} \subseteq \mathcal{R}$ for (p, q). If $(p', q') \in \mathcal{P}$, then $p'\mathcal{R}q'$ and meas(p') < meas(p), so we have by the induction hypothesis $\vdash_{SIM} p' \longrightarrow q'$. Thus, we have derived $\vdash_{SIM} p \longrightarrow q$. ∎

Concerning CCS, the full language is not noetherian because of the presence of the recursion operator. If we consider only terms without μ, i.e. finite processes, then the same property holds, as is witnessed by the following theorem. We omit the proof of this theorem since it is essentially like the preceding proof: the main difference is that the definition of one-step transitions in CCS is given by recursion.

Theorem 4.5 *Let p and q be finite CCS processes, and let $\vdash$ denote derivability in $FO\lambda^{\Delta\mathbb{N}}$ using definition $\mathcal{D}(ccs(\mathcal{A})) \cup \mathcal{D}(sims)$. Then*

- $\vdash$ *sim p q if and only if* $p \sqsubseteq q$, *and*
- $\vdash$ *bisim p q if and only if* $p \equiv q$.

4.3.2 Definitions and Fixed Points

As we mentioned in Section 2.1, a definition can be seen as a mutually recursive definition for the predicates that are present in the heads of clauses. To make the notion of recursive definition more explicit, we consider in more detail the definitions used in the previous section. For a general discussion of using stratified specifications to provide for mutually recursive predicate definitions, see, for example, [2].

The clauses for defining the level 0 predicate *one* (both for abstract transition systems and for CCS) can be seen as simple Horn clauses (when $\triangleq$ is read as a reverse implication). The usual means for providing meaning to Horn clauses by seeing them as a monotone mapping on the Herbrand universe [1] can be exploited here as well. In particular, we can observe that if we remove the one clause of the CCS definition pertaining to μ (that is,

if we do not allow processes to contain occurrences of the fixed point operator) then the resulting definition for *one* would yield an operator on the Herbrand universe with exactly one fixed point. We shall take this fixed point as the meaning of the definition for *one*.

Simulation and bisimulation are predicates of level 1 using recursive definitional clauses of the form $\forall P \forall Q[r(P,Q) \stackrel{\triangle}{=} \Phi]$, where the formula Φ contains free occurrences of the variables P, Q, strictly positive occurrences of the predicate r, and both positive and negative occurrences of the predicate *one*. With such a clause we associate a function ϕ from binary relations to binary relations. Given the structure of Φ it is easy to see that ϕ will be monotone and thus have fixed points: in general, however, there will be more than one such fixed point.

Notice that both $def\,\mathcal{L}$ and $def\,\mathcal{R}$ are sound for all the relations which are fixed points of ϕ. To see this, assume that for any relation r there is only one such definitional clause (in case there are more, we group them in one clause which has as body the disjunction of the bodies). Then observe that the use of $def\,\mathcal{L}$ corresponds to replacing $\stackrel{\triangle}{=}$ by $\supset$ in the clause, that is, to assuming the formula $\forall P \forall Q[r(P,Q) \supset \Phi]$. The case for $def\,\mathcal{R}$ corresponds to the converse: that is, of replacing $\stackrel{\triangle}{=}$ with $\subset$.

Let Φ_s and Φ_b be the bodies of the clauses given in $\mathcal{D}(sims)$ for *sim* and *bisim*, respectively, and consider the corresponding functions ϕ_s and ϕ_b on binary relations associated with these formulas. More explicitly, we may define these two functions on binary relations as follows:

$$\phi_s(\mathcal{R}) := \{(p,q) \mid \text{ for all } a \in \Lambda \text{ and } p' \in S, \text{ if } p \stackrel{a}{\longrightarrow} p' \text{ then there is a } q' \text{ such that } q \stackrel{a}{\longrightarrow} q' \text{ and } (p',q') \in \mathcal{R}\}$$

$$\phi_b(\mathcal{R}) := \{(p,q) \mid \text{ for all } a \in \Lambda \text{ and } p' \in S, \text{ if } p \stackrel{a}{\longrightarrow} p' \text{ then there is a } q' \text{ such that } q \stackrel{a}{\longrightarrow} q' \text{ and } (p',q') \in \mathcal{R}, \text{ and for all } a \in \Lambda \text{ and } q' \in S, \text{ if } q \stackrel{a}{\longrightarrow} q' \text{ then there is a } p' \text{ such that } p \stackrel{a}{\longrightarrow} p' \text{ and } (q',p') \in \mathcal{R}\} \quad .$$

We can see from their definitions that $\sqsubseteq$ and $\equiv$ are the greatest fixed points of ϕ_s and ϕ_b, respectively. Notice that in derivations of $\longrightarrow$ *sim* p q and $\longrightarrow$ *bisim* p q using the definition $\mathcal{D}(ats(\delta)) \cup \mathcal{D}(sims)$, $def\,\mathcal{L}$ is used on clauses in $\mathcal{D}(ats(\delta))$ but not with those in $\mathcal{D}(sims)$.

As the following example shows, when the transition system is not noetherian, the "if" parts of Theorem 4.4 may not hold.

Example 4.6 Consider a transition system with two states only, p and q, and two transitions $p \xrightarrow{a} p$ and $q \xrightarrow{a} q$. Then $p \sqsubseteq q$ holds, but *sim p q* cannot be derived. Notice that an attempt to derive it would end up in a circularity.

Since $def\mathcal{L}$ and $def\mathcal{R}$ are sound in all fixed points, if *sim p q* is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using $\mathcal{D}(ats(\delta)) \cup \mathcal{D}(sims)$, then (p, q) must be contained in every fixed point of ϕ_s. In a noetherian abstract transition system, ϕ_s and ϕ_b have unique fixed points, and it is for this reason that $\sqsubseteq$ and $\equiv$ can be completely characterized in a noetherian ats by derivability (Theorem 4.4).

Before proceeding to consider an encoding that captures the greatest fixed point of ϕ_s and ϕ_b, we briefly explore the kinds of properties about simulation and bisimulation that we can derive in $FO\lambda^{\Delta\mathbb{N}}$ with $\mathcal{D}(sims)$.

In any transition system, bisimulation is symmetric; the encoding of this property is derivable in $FO\lambda^{\Delta\mathbb{N}}$:

Proposition 4.7 *The formula* $\forall p \forall q(bisim\ p\ q \supset bisim\ q\ p)$ *is derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(ats(\delta)) \cup \mathcal{D}(sims)$.

In CCS, bisimulation is preserved by the prefix operator; the encoding of this property is also derivable in $FO\lambda^{\Delta\mathbb{N}}$:

Proposition 4.8 *The formula* $\forall a \forall p \forall q(bisim\ p\ q \supset bisim\ a.p\ a.q)$ *is derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(ccs(\mathcal{A})) \cup \mathcal{D}(sims)$.

However, as the following examples illustrate, there are plenty of true properties of $\equiv$ and $\sqsubseteq$ that cannot be derived within the logic. One reason for this lack is, intuitively, we can prove properties of *sim* and *bisim* from $\mathcal{D}(sims)$ only if they are true for every fixed point of ϕ_s and ϕ_b, but in the non-noetherian case, there is in general more than one fixed point.

Example 4.9 Bisimulation equivalence implies the largest simulation (or more formally: $\equiv$ is a subset of $\sqsubseteq$) in any transition system. This property can be expressed by the formula

$\forall p \forall q(\mathit{bisim}\ p\ q \supset \mathit{sim}\ p\ q)$ but, in general, if δ is a non-noetherian transition relation, this formula cannot be derived using the definitions $\mathcal{D}(\mathit{ats}(\delta))$ and $\mathcal{D}(\mathit{sims})$. For example, if we take the transition system $(\{a\}, \{p\}, \{(p, a, p)\})$ it is immediate to see that $\{(p, p)\}$ is a bisimulation (the greatest fixed point of ϕ_b, namely bisimulation equivalence) and $\emptyset$ is a simulation (the least fixed point of ϕ_s). Hence, this formula cannot be derived for this transition system.

Example 4.10 The bisimulation equivalence relation is reflexive in any transition system, but the formula $\forall p(\mathit{bisim}\ p\ p)$ cannot be derived using the definitions $\mathcal{D}(\mathit{ats}(\delta))$ and $\mathcal{D}(\mathit{sims})$. Consider, for instance, the same transition system as in Example 4.9: the empty set $\emptyset$ is a bisimulation (the least fixed point of ϕ_b), and it is, of course, not reflexive.

Example 4.11 In CCS, bisimulation equivalence is preserved by the + operator. This property can be expressed as the formula $\forall p \forall q \forall r(\mathit{bisim}\ p\ q \supset \mathit{bisim}\ (p + r)\ (q + r))$. This sequent, however, cannot be derived using $\mathcal{D}(\mathit{ccs}(\mathcal{A}))$ and $\mathcal{D}(\mathit{sims})$. If we let $p = a.0$, $q = a.0 + a.0$ and $r = \mu_x a.x$, the least fixed point of ϕ_b contains the pair $(a.0, a.0 + a.0)$ but not the pair $(a.0 + \mu_x a.x, a.0 + a.0 + \mu_x a.x)$.

4.3.3 Non-Finite Behavior

We now provide an encoding of simulation and bisimulation that uses induction to capture the greatest fixed point. In particular, define the binary relations $\sqsubseteq_i$ and $\equiv_i$ for each natural number i as follows. Both $\sqsubseteq_0$ and $\equiv_0$ are defined to be $S \times S$, and $\sqsubseteq_{i+1} := \phi_s(\sqsubseteq_i)$ and $\equiv_{i+1} := \phi_b(\equiv_i)$. Now set $\sqsubseteq_\omega := \bigcap_i \sqsubseteq_i$ and $\equiv_\omega := \bigcap_i \equiv_i$. It is well known that, for finitely branching transition systems, ϕ_s and ϕ_b are downward-continuous and, hence, $\sqsubseteq$ equals $\sqsubseteq_\omega$ and $\equiv$ equals $\equiv_\omega$. In CCS, for instance, finite branching is guaranteed whenever all the recursion variables in μ-expressions are prefixed.

Thus one approach to showing that two states are bisimilar is to show that for all natural numbers i, those two states are related by $\equiv_i$. Such statements can often be proved by induction on natural numbers. If n is a natural number, then we write $\bar{n}$ to denote the corresponding "numeral" for n, that is, $\bar{n}$ is the term containing n occurrences of s and one occurrence of z.

Table 4.2: Indexed definition for simulation and bisimulation

$$
\begin{array}{rcl}
\mathit{sim}\ P\ Q & \triangleq & \forall k(\mathit{nat}\ k \supset \mathit{isim}\ k\ P\ Q) \\
\mathit{isim}\ z\ P\ Q & \triangleq & \top \\
\mathit{isim}\ (s\ K)\ P\ Q & \triangleq & \forall a \forall p'(\mathit{one}\ P\ a\ p' \supset \\
 & & \quad \exists q'(\mathit{one}\ Q\ a\ q' \wedge \mathit{isim}\ K\ p'\ q')) \\
 & & \\
\mathit{bisim}\ P\ Q & \triangleq & \forall k(\mathit{nat}\ k \supset \mathit{ibisim}\ k\ P\ Q) \\
\mathit{ibisim}\ z\ P\ Q & \triangleq & \top \\
\mathit{ibisim}\ (s\ K)\ P\ Q & \triangleq & \forall a \forall p'(\mathit{one}\ P\ a\ p' \supset \\
 & & \quad \exists q'(\mathit{one}\ Q\ a\ q' \wedge \mathit{ibisim}\ K\ p'\ q')) \wedge \\
 & & \forall a \forall q'(\mathit{one}\ Q\ a\ q' \supset \\
 & & \quad \exists p'(\mathit{one}\ P\ a\ p' \wedge \mathit{ibisim}\ K\ q'\ p'))
\end{array}
$$

We can now encode $\sqsubseteq_i$ and $\equiv_i$ by using the indexed versions of *sim* and *bisim* found in the definition $\mathcal{D}(\mathit{isims})$ shown in Table 4.2.

Lemma 4.12 *Let (Λ, S, δ) be an ats, let p and q be members of S, and let $\vdash$ denote derivability in $FO\lambda^{\Delta\mathbb{N}}$ using definition $\mathcal{D}(\mathit{ats}(\delta)) \cup \mathcal{D}(\mathit{isims})$.*

- *If $\vdash \mathit{sim}\ p\ q$ then for every natural number n, $\vdash \mathit{isim}\ \bar{n}\ p\ q$.*
- *If $\vdash \mathit{bisim}\ p\ q$ then for every natural number n, $\vdash \mathit{ibisim}\ \bar{n}\ p\ q$.*

Proof We prove the first result about simulation: the result about bisimulation is similar. A cut-free derivation of the sequent $\longrightarrow \mathit{sim}\ p\ q$ must end in a $\mathit{def}\mathcal{R}$ rule, which (using $\forall\mathcal{R}$ and $\supset\mathcal{R}$ also) means that the sequent $\mathit{nat}\ k \longrightarrow \mathit{isim}\ k\ p\ q$ is derivable, where k is a variable. Call this derivation Ξ. Now let n be a natural number. It is possible to substitute $\bar{n}$ for the variable k into the derivation Ξ to obtain the derivation $\Xi[\bar{n}/k]$ of the sequent $\mathit{nat}\ \bar{n} \longrightarrow \mathit{isim}\ \bar{n}\ p\ q$. Given that n is a natural number, it is easy to construct a cut-free derivation of $\longrightarrow \mathit{nat}\ \bar{n}$, one using only the right rules for *nat*. Now placing these two derivations together with a cut rule yields

$$\frac{\longrightarrow \mathit{nat}\ \bar{n} \qquad \mathit{nat}\ \bar{n} \longrightarrow \mathit{isim}\ \bar{n}\ p\ q}{\longrightarrow \mathit{isim}\ \bar{n}\ p\ q}\ \mathit{cut}\ .$$

Given the cut-elimination result for $FO\lambda^{\Delta\mathbb{N}}$ (Theorem 3.13), we can conclude that $\longrightarrow$

isim $\bar{n}$ p q has a cut-free derivation (a conclusion that is needed in the proof of Proposition 4.13 below). ∎

Proposition 4.13 *Let* (Λ, S, δ) *be an ats, let* p *and* q *be members of* S*, let* n *be a natural number, and let* $\vdash$ *denote derivability in* $FO\lambda^{\Delta\mathbb{N}}$ *using definition* $\mathcal{D}(ats(\delta)) \cup \mathcal{D}(isims)$*.*

- *If* $\vdash$ *isim* $\bar{n}$ p q *then* $p \sqsubseteq_n q$*.*
- *If* $\vdash$ *ibisim* $\bar{n}$ p q *then* $p \equiv_n q$*.*

Proof We prove the first result about simulation: the result about bisimulation is similar. Assume that n is 0. Then $p \sqsubseteq_0 q$ holds immediately. Otherwise, let n be $m+1$. Assume that *isim* s $\bar{m}$ p q has a cut-free derivation. An analysis of the inference rules used to derive this sequent (as we argued similarly in Section 4.3.1) shows that that for some premise set $\mathcal{P}$, there is a subderivation of the sequent $\longrightarrow$ *isim* $\bar{m}$ p' q' for every $(p', q') \in \mathcal{P}$. Using the inductive assumption, $p' \sqsubseteq_m q'$ for all $(p', q') \in \mathcal{P}$. Hence, by the definition of ϕ_s, we have $p \sqsubseteq_{m+1} q$. ∎

Putting these results together with the one mentioned earlier regarding when ϕ_s and ϕ_b are downward continuous, we can prove the following.

Theorem 4.14 *Let* (Λ, S, δ) *be an ats, and let* p *and* q *be members of* S*.*

- *If* $(sim\ p\ q)$ *is derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *using definition* $\mathcal{D}(ats(\delta)) \cup \mathcal{D}(isims)$*, then* $p \sqsubseteq_\omega q$*.*
- *If* $(bisim\ p\ q)$ *is derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *using definition* $\mathcal{D}(ats(\delta)) \cup \mathcal{D}(isims)$*, then* $p \equiv_\omega q$*.*

If the abstract transition system is finitely branching, then we can conclude the stronger fact that $p \sqsubseteq q$ *or* $p \equiv q$*.*

With this indexed definition it is now possible to derive many properties of simulation and bisimulation.

Theorem 4.15 *The following formulas can be derived in* $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(ccs(\mathcal{A})) \cup \mathcal{D}(isims)$*:*

$$\forall p \forall q (bisim\ p\ q \supset sim\ p\ q) \qquad \forall p \forall q \forall r (bisim\ p\ q \supset bisim\ (p + r)\ (q + r))$$

$$\forall p \forall q (\mathit{bisim}\ p\ q \supset \mathit{bisim}\ q\ p) \qquad \forall a \forall p \forall q (\mathit{bisim}\ p\ q \supset \mathit{bisim}\ a.p\ a.q)$$

$$\forall p \forall q \forall r (\mathit{bisim}\ p\ q \supset \mathit{bisim}\ q\ r \supset \mathit{bisim}\ p\ r) \qquad \forall a\ \mathit{bisim}\ \mu_x a.x\ \mu_x (a.x + a.x)$$

$$\forall p\ \mathit{bisim}\ p\ p \qquad \forall p\ \mathit{bisim}\ (p + 0)\ p \qquad \forall p\ \mathit{bisim}\ (p + p)\ p\ .$$

4.4 Conclusion

We have shown that the definitional facilities of $FO\lambda^{\Delta\mathbb{N}}$ allow us to naturally capture certain properties about elements of transition systems, namely simulation and bisimulation. Furthermore, with induction over natural numbers we can establish more high-level facts about these properties, such as the fact that bisimulation is an equivalence.

From a high-level point-of-view, we can characterize the experiments we have reported here in two ways. From a (traditional) logic programming point of view, a definition D is generally either a set of (positive) Horn clauses or an extension of them that allows negated atoms in the body of clauses. In that case, sequents in a derivation of an atomic formula are either of the form $\longrightarrow B$ or $B \longrightarrow$. In the first case, $def\mathcal{R}$ is used to establish B and, in the second case, $def\mathcal{L}$ is used to build a finite refutation of B. Here we consider richer definitions so that the search for deriations must consider sequents of the form $B \longrightarrow C$; with such sequents, both left and right introduction of definitions are used together. From a computational or concurrency point-of-view, derivations using just $def\mathcal{R}$ only capture the *may* behavior of a system: "there exists a computation such that ..." is easily translated to "there exists a derivation of ...". The addition of the $def\mathcal{L}$ inference rule allows certain forms of *must* behavior to be captured.

Chapter 5

Reasoning about Logics in $FO\lambda^{\Delta\mathbb{N}}$

In Chapter 4 we showed how the logic $FO\lambda^{\Delta\mathbb{N}}$ can be used to reason about transition systems. The only variable-binding constructor that we considered was the CCS μ constructor allowing the recursive description of processes; in that setting, higher-order abstract syntax played only a minor role. We now turn our attention to object systems where bound variables and substitution are more prominent and thus higher-order abstract syntax becomes more important. In this chapter we consider intuitionistic and linear logics; we proceed to reason about programming languages in Chapter 6.

Since $FO\lambda^{\Delta\mathbb{N}}$ contains quantification at higher-order types and term structures involving λ-terms, it easily supports higher-order abstract syntax. Eriksson [12] demonstrated the use of his finitary calculus of partial inductive definitions (which is similar to $FO\lambda^{\Delta\mathbb{N}}$) for the specification of various logics and type systems using higher-order abstract syntax. Our goal is to go a step beyond that and also reason within $FO\lambda^{\Delta\mathbb{N}}$ about the object systems. As we set about to do so, we encounter some difficulties in reasoning about higher-order abstract syntax specifications within the specification logic and develop strategies for surmounting those difficulties.

We begin the first section of this chapter by presenting the usual higher-order abstract syntax representation of intuitionistic logic and illustrating the problems alluded to above. We then proceed through several modifications of this encoding which improve our ability to perform meta-theoretic analyses, although at some loss of the benefits of higher-order abstract syntax. In Section 5.2 we further illustrate these encoding techniques through two

examples involving fragments of intuitionistic and linear logic. The specifications of these two logics will also be used in Chapter 6 as part of an alternative strategy for formal reasoning with higher-order abstract syntax that retains the full benefits of this representation style. We conclude this chapter with a section discussing related work.

5.1 Logic Representations for Meta-Theoretic Analysis

5.1.1 Natural Deduction-Style Encoding

In order to examine our ability to reason about higher-order abstract syntax encodings in $FO\lambda^{\Delta\mathbb{N}}$, we present a definition of first-order intuitionistic logic. We use the type i for terms of the object logic, the type atm for atoms (atomic propositions) and the type prp for general propositions; we also introduce the following constants:

$$\begin{array}{lcllcllcl} \langle\,\rangle & : & atm \to prp & \& & : & prp \to prp \to prp & \bigwedge_i & : & (i \to prp) \to prp \\ tt & : & prp & \oplus & : & prp \to prp \to prp & \bigvee_i & : & (i \to prp) \to prp \\ ff & : & prp & \Rightarrow & : & prp \to prp \to prp & & & \end{array} .$$

The constant $\langle\,\rangle$ coerces atoms into propositions: object-level predicates will be constants that build meta-level terms of type atm. The constants tt and ff are the representations of true and false, the constants $\&$, $\oplus$, and $\Rightarrow$ represent the conjunction, disjunction, and implication connectives, and $\bigwedge_i$ and $\bigvee_i$ encode universal and existential quantification at type i. Notice that we are using the λ-abstraction of $FO\lambda^{\Delta\mathbb{N}}$'s term language to represent the variable binding of the two object logic quantifiers. As a result, α-equivalence of quantified object logic formulas follows from the α-equivalence of λ-bound terms in $FO\lambda^{\Delta\mathbb{N}}$, and substitution for object logic variables can be accomplished by β-reduction at the level of $FO\lambda^{\Delta\mathbb{N}}$ terms.

Derivability in the object logic is encoded via the predicate *prove* of type $prp \to o$; the usual higher-order abstract syntax encoding of this predicate is the theory shown in Table 5.1. The first seven clauses correspond to the introduction rules for natural deduction; the remaining seven correspond to the elimination rules.

Although this encoding mirrors the rules for natural deduction, it may be viewed as an encoding of the sequent calculus, with the derivability of the sequent $B_1, \ldots, B_n \longrightarrow C$

Table 5.1: Natural deduction encoding of intuitionistic logic

$prove\ tt$	$\subset$	$\top$
$prove\ (B\ \&\ C)$	$\subset$	$prove\ B\ \wedge\ prove\ C$
$prove\ (B \oplus C)$	$\subset$	$prove\ B$
$prove\ (B \oplus C)$	$\subset$	$prove\ C$
$prove\ (B \Rightarrow C)$	$\subset$	$prove\ B \supset prove\ C$
$prove\ \bigwedge_i B$	$\subset$	$\forall_i x\ prove\ (B\,x)$
$prove\ \bigvee_i B$	$\subset$	$\exists_i x\ prove\ (B\,x)$
$prove\ B$	$\subset$	$prove\ ff$
$prove\ B$	$\subset$	$\exists c\ prove\ (B\ \&\ c)$
$prove\ C$	$\subset$	$\exists b\ prove\ (b\ \&\ C)$
$prove\ D$	$\subset$	$\exists b \exists c (prove\ (b \oplus c)\ \wedge\ (prove\ b \supset prove\ D)\ \wedge\ (prove\ c \supset prove\ D))$
$prove\ C$	$\subset$	$\exists b (prove\ (b \Rightarrow C)\ \wedge\ prove\ b)$
$prove\ (B\,X)$	$\subset$	$prove\ \bigwedge_i B$
$prove\ C$	$\subset$	$\exists b (prove\ \bigvee_i b\ \wedge\ (\exists_i x\ prove\ (b\,x) \supset prove\ C))$

represented by the $FO\lambda^{\Delta\mathbb{N}}$ formula

$$prove\ B_1 \supset \cdots \supset prove\ B_n \supset prove\ C\ .$$

This is in keeping with the higher-order abstract syntax principle of using specification logic hypotheses to represent contexts (in this case, the left side of the sequent). The structural rules (exchange, weakening, and contraction) follow immediately from this representation; for example, the derivation for weakening is

$$\dfrac{\dfrac{\dfrac{\dfrac{}{prove\ c, prove\ b \longrightarrow prove\ c}\ init}{\longrightarrow prove\ c \supset (prove\ b \supset prove\ c)}\ \supset\mathcal{R}}{\longrightarrow \forall b \forall c (prove\ c \supset (prove\ b \supset prove\ c))}\ \forall\mathcal{R}}\ .$$

The admissibility of the cut rule, encoded by the formula

$$\forall b \forall c ((prove\ b \supset prove\ c) \supset prove\ b \supset prove\ c)\ ,$$

also follows easily from the $\supset\mathcal{L}$ rule. The right rules are the same as the corresponding introduction rules, and the left rules are easily derived from the clauses for the corresponding elimination rules. The left rule for $\bigwedge_i$, for instance, is encoded by the $FO\lambda^{\Delta\mathbb{N}}$ formula

$$\forall b \forall c (\exists x (prove\ (b\,x) \supset prove\ c) \supset (prove\ \bigwedge_i b \supset prove\ c))\ ,$$

whose derivation is evident from the clause for the elimination rule for $\bigwedge_i$.

However, this encoding is not appropriate for meta-theoretic analysis of object logic derivations. To do such analysis in $FO\lambda^{\Delta\mathbb{N}}$, we need to be able to perform induction over the derivations. Recall that in Section 2.2.2 we used the natural number measure in the *length* predicate to derive an induction principle for lists. But there is no apparent way to add a natural number induction measure to the *prove* predicate because of the clause for the $\Rightarrow$ introduction rule. This reflects the fact that this clause gives rise to a non-monotone operator (cf. Section 4.3.2); this is generally true of the types and theories in higher-order abstract syntax encodings, and makes inductive principles difficult to find. We would also like to change the specification into a definition so that we can use the $def\mathcal{L}$ rule for the analysis of derivations. Simply replacing the $\subset$ in each clause by $\stackrel{\triangle}{=}$ is problematic for two reasons. First, the clause resulting from the introduction rule for $\Rightarrow$ would not satisfy the level restriction for any level we might assign to *prove*. Second, the clause resulting from the elimination rule for $\bigwedge_i$ would have a problematic head. There are too many ways that $(B\,X)$ can match and unify with other terms; this makes the practical application of the $def\mathcal{R}$ and $def\mathcal{L}$ rules difficult, and would result in many cases that are not productive.

5.1.2 Sequent-Style Encoding

We can solve the problems with the encoding of the introduction rule for $\Rightarrow$ by introducing separate predicates

$$hyp \;:\; prp \to o \qquad\qquad conc \;:\; nt \to prp \to o$$

for the left and right sides of the sequent, respectively. The predicate *hyp* will not be a defined predicate, and so can have level zero. The negative occurrence of *prove* in the introduction clause for $\Rightarrow$ becomes an occurence of *hyp*, so the predicate *conc* can then have level one. This also makes possible the assignment of a measure to *conc*, as suggested by its type. To emphasize that the first argument to *conc* is a measure, we will write it as a subscript. The problem introduced by the elimination clause for $\bigwedge_i$ is avoided by patterning the encoding after the sequent calculus rules rather than natural deduction rules. The resulting definition is shown in Table 5.2. The first clause encodes the initial

Table 5.2: Sequent calculus encoding of intuitionistic logic

$$
\begin{array}{rcl}
conc_I\ \langle A\rangle & \triangleq & hyp\ \langle A\rangle \\
 & & \\
conc_I\ tt & \triangleq & \top \\
conc_{(s\ I)}\ (B\ \&\ C) & \triangleq & conc_I\ B\ \wedge\ conc_I\ C \\
conc_{(s\ I)}\ (B \oplus C) & \triangleq & conc_I\ B \\
conc_{(s\ I)}\ (B \oplus C) & \triangleq & conc_I\ C \\
conc_{(s\ I)}\ (B \Rightarrow C) & \triangleq & hyp\ B \supset conc_I\ C \\
conc_{(s\ I)}\ \bigwedge_i B & \triangleq & \forall_i x\ conc_I\ (B\ x) \\
conc_{(s\ I)}\ \bigvee_i B & \triangleq & \exists_i x\ conc_I\ (B\ x) \\
 & & \\
conc_I\ B & \triangleq & hyp\ ff \\
conc_{(s\ I)}\ D & \triangleq & \exists b \exists c (hyp\ (b\ \&\ c)\ \wedge\ (hyp\ b \supset conc_I\ D)) \\
conc_{(s\ I)}\ D & \triangleq & \exists b \exists c (hyp\ (b\ \&\ c)\ \wedge\ (hyp\ c \supset conc_I\ D)) \\
conc_{(s\ I)}\ D & \triangleq & \exists b \exists c (hyp\ (b \oplus c)\ \wedge\ (hyp\ b \supset conc_I\ D)\ \wedge\ (hyp\ c \supset conc_I\ D)) \\
conc_{(s\ I)}\ D & \triangleq & \exists b \exists c (hyp\ (b \Rightarrow c)\ \wedge\ (hyp\ c \supset conc_I\ D)\ \wedge\ conc_I\ b) \\
conc_{(s\ I)}\ C & \triangleq & \exists b (hyp\ \bigwedge_i b\ \wedge\ (\forall_i x\ hyp\ (b\ x) \supset conc_I\ C)) \\
conc_{(s\ I)}\ C & \triangleq & \exists b (hyp\ \bigvee_i b\ \wedge\ (\exists_i x\ hyp\ (b\ x) \supset conc_I\ C))
\end{array}
$$

axiom, the next seven correspond to the right introduction rules, and the remaining seven correspond to the left introduction rules.

Since we have not changed the representation of quantification, we get α-equivalence of quantified object logic formulas and substitution for object logic variables from the relevant features of $FO\lambda^{\Delta\mathbb{N}}$ as before. We are still using $FO\lambda^{\Delta\mathbb{N}}$ hypotheses to represent contexts, so the structural rules also follow as before. However, the admissibility of the cut rule, now encoded as

$$\forall b \forall c (\exists i (hyp\ b \supset conc_i\ c) \supset \exists i\ conc_i\ b \supset \exists i\ conc_i\ c)\ ,$$

is no longer immediate: there is no simple proof of $\exists i\ conc_i\ b \longrightarrow hyp\ b$. We expect, though, that the admissibility of cut is still derivable in $FO\lambda^{\Delta\mathbb{N}}$ following the method of [45].

This encoding has another limitation; to see it, consider the following example. Suppose we know that the sequent $b \Rightarrow a \longrightarrow a$ is derivable in intuitionistic logic for some atom a and proposition b. Since a is atomic, the derivation must end with a left rule, and since

the only formula on the left is $b \Rightarrow a$, it must be the left implication rule. Thus there are derivations of $b \Rightarrow a \longrightarrow b$ and $a, b \Rightarrow a \longrightarrow a$. This second sequent is not so interesting, since it is an initial sequent. So we have shown that if $b \Rightarrow a \longrightarrow a$ is derivable then $b \Rightarrow a \longrightarrow b$ is as well.

Now let us try to capture this reasoning in $FO\lambda^{\Delta\mathbb{N}}$ using our current encoding of intuitionistic logic. We want to derive the sequent

$$\longrightarrow \forall a \forall b (\exists i (hyp\ (b \Rightarrow \langle a \rangle) \supset conc_i\ \langle a \rangle) \supset \exists j (hyp\ (b \Rightarrow \langle a \rangle) \supset conc_j\ b)) \ .$$

After the obvious uses of $\forall\mathcal{R}$ and $\supset \mathcal{R}$, we get

$$\exists i (hyp\ (b \Rightarrow \langle a \rangle) \supset conc_i\ \langle a \rangle) \longrightarrow \exists j (hyp\ (b \Rightarrow \langle a \rangle) \supset conc_j\ b) \ .$$

From our informal reasoning, we know that the derivation of b will have a smaller measure than the derivation of a; thus in applying the $\exists\mathcal{L}$ and $\exists\mathcal{R}$ rules it is conservative to substitute i for j:

$$hyp\ (b \Rightarrow \langle a \rangle) \supset conc_i\ \langle a \rangle \longrightarrow hyp\ (b \Rightarrow \langle a \rangle) \supset conc_i\ b \ .$$

To follow the informal proof, we now want to indicate that $hyp\ (b \Rightarrow \langle a \rangle) \supset conc_i\ \langle a \rangle$ must be true by the definitional clause encoding the left $\Rightarrow$ rule. However, we cannot apply the $def\mathcal{L}$ rule to this formula, since it is not an atom. The closest thing to this that we can do is to eliminate the $\supset$ and then apply $def\mathcal{L}$ to $conc_i\ \langle a \rangle$. We can eliminate the $\supset$ by using $\supset \mathcal{R}$ and then $\supset \mathcal{L}$, yielding the two sequents

$$hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow hyp\ (b \Rightarrow \langle a \rangle)$$

$$conc_i\ \langle a \rangle, hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow conc_i\ b \ .$$

The first is immediate by the *init* rule. Applying the $def\mathcal{L}$ rule to $conc_i\ \langle a \rangle$ in the second sequent yields eight sequents corresponding to the cases where the derivation of a ends with the initial rule or any of the seven left rules:

$$hyp\ \langle a \rangle, hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow conc_i\ b$$

$$hyp\ ff, hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow conc_i\ b$$

$$\exists b' \exists c' (hyp\ (b'\ \&\ c')\ \wedge\ (hyp\ b' \supset conc_{i'}\ \langle a \rangle)), hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow conc_{(S\ i')}\ b$$

$$\exists b' \exists c' (hyp\ (b'\ \&\ c')\ \wedge\ (hyp\ c' \supset conc_{i'}\ \langle a \rangle)), hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow conc_{(S\ i')}\ b$$

$$\exists b' \exists c' (hyp\ (b' \oplus c')\ \wedge\ (hyp\ b' \supset conc_{i'}\ \langle a \rangle)\ \wedge\ (hyp\ c' \supset conc_{i'}\ \langle a \rangle)),$$
$$hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow conc_{(S\ i')}\ b$$

$$\exists b' \exists c' (hyp\ (b' \Rightarrow c')\ \wedge\ (hyp\ c' \supset conc_{i'}\ \langle a \rangle)\ \wedge\ conc_{i'}\ b'), hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow conc_{(S\ i')}\ b$$

$$\exists b' (hyp\ \textstyle\bigwedge_i b'\ \wedge\ (\forall_i x\ hyp\ (b'\ x) \supset conc_{i'}\ \langle a \rangle)), hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow conc_{(S\ i')}\ b$$

$$\exists b' (hyp\ \textstyle\bigvee_i b'\ \wedge\ (\exists_i x\ hyp\ (b'\ x) \supset conc_{i'}\ \langle a \rangle)), hyp\ (b \Rightarrow \langle a \rangle) \longrightarrow conc_{(S\ i')}\ b\ .$$

This is clearly not what we want. Even in the case corresponding to the left $\Rightarrow$ rule we do not know that the rule was applied to the implication $b \Rightarrow \langle a \rangle$. There are really two problems here. The first is that $hyp\ (b \Rightarrow \langle a \rangle) \supset conc_i\ \langle a \rangle$ expresses the idea that $b \Rightarrow \langle a \rangle$ is a hypothesis available in the derivation of $conc_i\ \langle a \rangle$, but it does not capture the idea that it is the only hypothesis available. Thus the $def\mathcal{L}$ rule forces us to consider derivations ending with the initial rule or any of the left rules, since the appropriate formula may be available as a hypothesis. The second problem is that we do not have any way to examine the different ways of deriving something from a specific set of hypotheses. Although the formula $hyp\ (b \Rightarrow \langle a \rangle) \supset conc_i\ \langle a \rangle$ indicates that the atom a is derivable from the hypothesis $b \Rightarrow \langle a \rangle$, we cannot examine how that derivation might take place. All we can do is use the $def\mathcal{L}$ rule, which says that we know that the hypothesis $b \Rightarrow \langle a \rangle$ is available and so can conclude that a holds.

5.1.3 Explicit Sequent Encoding

To remedy this situation, we explicitly represent the entire sequent in a single atomic judgement. As a result, the relevant object logic hypotheses are known to be exactly those listed in the judgement, and the $def\mathcal{L}$ rule can be applied to the judgement to examine how the corresponding sequent might be derived. Thus derivability is encoded via the predicate

$$seq\ \ :\ \ nt \rightarrow prplst \rightarrow prp \rightarrow o\ .$$

The first argument is an induction measure and will be displayed as a subscript. The second argument is a list of terms of type *prp* and represents the left side of the sequent. We will assume that *prplst* is the same as the type *lst* introduced in Section 2.2.2, using *prp* for the

Table 5.3: Explicit sequent encoding of intuitionistic logic

$$
\begin{array}{rcl}
seq_I\ L\ \langle A\rangle & \triangleq & element\ \langle A\rangle\ L \\
 & & \\
seq_I\ L\ tt & \triangleq & \top \\
seq_{(s\ I)}\ L\ (B\,\&\,C) & \triangleq & seq_I\ L\ B\ \wedge\ seq_I\ L\ C \\
seq_{(s\ I)}\ L\ (B\oplus C) & \triangleq & seq_I\ L\ B \\
seq_{(s\ I)}\ L\ (B\oplus C) & \triangleq & seq_I\ L\ C \\
seq_{(s\ I)}\ L\ (B\Rightarrow C) & \triangleq & seq_I\ (B::L)\ C \\
seq_{(s\ I)}\ L\ (\bigwedge_i B) & \triangleq & \forall_i x\ seq_I\ L\ (B\,x) \\
seq_{(s\ I)}\ L\ (\bigvee_i B) & \triangleq & \exists_i x\ seq_I\ L\ (B\,x) \\
 & & \\
seq_I\ L\ B & \triangleq & element\ ff\ L \\
seq_{(s\ I)}\ L\ D & \triangleq & \exists b \exists c(element\ (b\,\&\,c)\ L\ \wedge\ seq_I\ (b::L)\ D) \\
seq_{(s\ I)}\ L\ D & \triangleq & \exists b \exists c(element\ (b\,\&\,c)\ L\ \wedge\ seq_I\ (c::L)\ D) \\
seq_{(s\ I)}\ L\ D & \triangleq & \exists b \exists c(element\ (b\oplus c)\ L\ \wedge\ seq_I\ (b::L)\ D\ \wedge\ seq_I\ (c::L)\ D) \\
seq_{(s\ I)}\ L\ D & \triangleq & \exists b \exists c(element\ (b\Rightarrow c)\ L\ \wedge\ seq_I\ (c::L)\ D\ \wedge\ seq_I\ L\ b) \\
seq_{(s\ I)}\ L\ C & \triangleq & \exists b(element\ \bigwedge_i b\ L\ \wedge\ \exists_i x\ seq_I\ ((b\,x)::L)\ C) \\
seq_{(s\ I)}\ L\ C & \triangleq & \exists b(element\ \bigvee_i b\ L\ \wedge\ \forall_i x\ seq_I\ ((b\,x)::L)\ C)
\end{array}
$$

type of elements. In particular we will assume that we have constructors *nil* and ::, and a predicate *element* as defined in $\mathcal{D}(list(prp))$. The third argument to *seq* corresponds to the right side of the sequent. The definition for this predicate is shown in Table 5.3.

Since we have not changed the representation of quantification, we get α-equivalence of quantified object logic formulas and substitution for object logic variables from the relevant features of $FO\lambda^{\Delta\mathbb{N}}$ as before. We are no longer using $FO\lambda^{\Delta\mathbb{N}}$ hypotheses to represent contexts, however, so the structural rules must now be derived by induction. The admissibility of the cut rule must also be derived by induction, as was the case with the previous encoding. With the atomic encoding of sequents, we now can analyze derivations of propositions from hypotheses. To see this, we revisit the example from above. To formalize this with the encoding of Table 5.3, we derive the sequent

$$\longrightarrow \forall a \forall b(\exists i\ seq_i\ ((b\Rightarrow\langle a\rangle)::nil)\ \langle a\rangle \supset \exists j\ seq_j\ ((b\Rightarrow\langle a\rangle)::nil)\ b)\ .$$

Applying the $\forall\mathcal{R}$, $\supset\mathcal{R}$, and $\exists\mathcal{L}$ rules yields the sequent

$$seq_i\ ((b \Rightarrow \langle a\rangle) :: nil)\ \langle a\rangle \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b\ .$$

Now we apply the $def\mathcal{L}$ rule to the judgement on the left, which yields eight sequents, again corresponding to the cases where the derivation of a ends with the initial rule or any of the seven left rules:

$$element\ \langle a\rangle\ ((b \Rightarrow \langle a\rangle) :: nil) \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b$$

$$element\ ff\ ((b \Rightarrow \langle a\rangle) :: nil) \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b$$

$$\exists b' \exists c' (element\ (b'\ \&\ c')\ ((b \Rightarrow \langle a\rangle) :: nil) \wedge seq_{i'}\ (b' :: (b \Rightarrow \langle a\rangle) :: nil)\ \langle a\rangle) \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b$$

$$\exists b' \exists c' (element\ (b'\ \&\ c')\ ((b \Rightarrow \langle a\rangle) :: nil) \wedge seq_{i'}\ (c' :: (b \Rightarrow \langle a\rangle) :: nil)\ \langle a\rangle) \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b$$

$$\exists b' \exists c' (element\ (b' \oplus c')\ ((b \Rightarrow \langle a\rangle) :: nil) \wedge seq_{i'}\ (b' :: (b \Rightarrow \langle a\rangle) :: nil)\ \langle a\rangle \wedge seq_{i'}\ (c' :: (b \Rightarrow \langle a\rangle) :: nil)\ \langle a\rangle) \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b$$

$$\exists b' \exists c' (element\ (b' \Rightarrow c')\ ((b \Rightarrow \langle a\rangle) :: nil) \wedge seq_{i'}\ (c' :: (b \Rightarrow \langle a\rangle) :: nil)\ \langle a\rangle \wedge seq_{i'}\ ((b \Rightarrow \langle a\rangle) :: nil)\ b') \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b$$

$$\exists b' (element\ \textstyle\bigwedge_i b'\ ((b \Rightarrow \langle a\rangle) :: nil) \wedge \exists_i x\ seq_{i'}\ ((b' x) :: (b \Rightarrow \langle a\rangle) :: nil)\ \langle a\rangle) \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b$$

$$\exists b' (element\ \textstyle\bigvee_i b'\ ((b \Rightarrow \langle a\rangle) :: nil) \wedge \forall_i x\ seq_{i'}\ ((b' x) :: (b \Rightarrow \langle a\rangle) :: nil)\ \langle a\rangle) \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b\ .$$

But this time we can easily eliminate seven of the eight possibilities, since the *element* assumption is obviously false. In the first sequent, for example, we have the assumption $element\ \langle a\rangle\ ((b \Rightarrow \langle a\rangle) :: nil)$. Since $\langle a\rangle$ cannot unify with $(b \Rightarrow \langle a\rangle)$, $\langle a\rangle$ cannot be the first element of the list; therefore it must be an element of the remainder. But the remainder is

the empty list, so $\langle a \rangle$ cannot be an element of it either. This is accomplished formally by applying the $def\,\mathcal{L}$ rule twice:

$$\dfrac{\dfrac{}{element\ \langle a\rangle\ nil \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b}\ def\,\mathcal{L}}{element\ \langle a\rangle\ ((b \Rightarrow \langle a\rangle) :: nil) \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b}\ def\,\mathcal{L}\ .$$

The remaining cases are done similarly, except for the one valid case, which corresponds to a use of the left $\Rightarrow$ rule:

$$\begin{aligned}\exists b' \exists c' (&element\ (b' \Rightarrow c')\ ((b \Rightarrow \langle a\rangle) :: nil) \wedge \\ &seq_{i'}\ (c' :: (b \Rightarrow \langle a\rangle) :: nil)\ \langle a\rangle \wedge \\ &seq_{i'}\ ((b \Rightarrow \langle a\rangle) :: nil)\ b') \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b\ .\end{aligned}$$

In this case, $b' \Rightarrow c'$ does match the first element of the list, so we must consider the case where the left $\Rightarrow$ rule was applied to $(b \Rightarrow \langle a\rangle)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\top, seq_{i'}\ ((b \Rightarrow \langle a\rangle) :: nil)\ b \longrightarrow \exists j \ldots \quad \dfrac{}{element\ (b' \Rightarrow c')\ nil, \ldots \longrightarrow \exists j \ldots}\ def\,\mathcal{L}}{element\ (b' \Rightarrow c')\ ((b \Rightarrow \langle a\rangle) :: nil), seq_{i'}\ ((b \Rightarrow \langle a\rangle) :: nil)\ b' \longrightarrow \exists j \ldots}\ def\,\mathcal{L}}{element\ (b' \Rightarrow c')\ \ldots \wedge \ldots, element\ (b' \Rightarrow c')\ \ldots \wedge \ldots \longrightarrow \exists j \ldots}\ \wedge\mathcal{L}}{element\ (b' \Rightarrow c')\ ((b \Rightarrow \langle a\rangle) :: nil) \wedge \ldots \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b}\ c\mathcal{L}}{\ldots \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b}\ \exists\mathcal{L}\ .$$

But the unproved sequent is easily derived by choosing j to be i':

$$\dfrac{\dfrac{}{\top, seq_{i'}\ ((b \Rightarrow \langle a\rangle) :: nil)\ b \longrightarrow seq_{i'}\ ((b \Rightarrow \langle a\rangle) :: nil)\ b}\ init}{\top, seq_{i'}\ ((b \Rightarrow \langle a\rangle) :: nil)\ b \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a\rangle) :: nil)\ b}\ \exists\mathcal{R}\ .$$

Now let us consider another example. Suppose we know that the sequent

$$\longrightarrow \bigwedge y_1 \bigwedge y_2 (p\, y_1\, t_1 \Rightarrow p\, y_2\, t_2 \Rightarrow p\, y_2\, t_3)$$

is derivable in intuitionistic logic for some predicate constant p and some terms t_1, t_2, and t_3. The derivation must end with applications of the right rules for $\bigwedge$ and $\Rightarrow$, since these are the only rules that apply. Thus we know that the sequent $p\, y_1\, t_1, p\, y_2\, t_2 \longrightarrow p\, y_2\, t_3$ is derivable. Since p is a predicate constant, these formulas are all atomic, so the only rule that applies is the initial rule. The eigenvariable condition for the application of the right

rule for $\bigwedge$ guarantees that y_1 and y_2 are distinct, so the initial rule must apply to the second hypothesis. Therefore, it must be the case that t_2 and t_3 are the same term.

Now let us try to capture this reasoning in $FO\lambda^{\Delta\mathbb{N}}$ using our current encoding of intuitionistic logic. To do this, we will need some way to indicate term identity, and so we introduce the predicate $\equiv$ of type $i \to i \to o$ defined by the clause $X \equiv X \stackrel{\triangle}{=} \top$. We then want to derive the sequent

$$\longrightarrow \forall p \forall t_1 \forall t_2 \forall t_3 (\exists i\ seq_i\ nil\ \textstyle\bigwedge_i y_1 \bigwedge_i y_2 (p\, y_1\, t_1 \Rightarrow p\, y_2\, t_2 \Rightarrow \langle p\, y_2\, t_3 \rangle) \supset t_2 \equiv t_3) \ .$$

The only way to proceed is by applying $\forall\mathcal{R}$ and $\supset \mathcal{R}$, yielding

$$\exists i\ seq_i\ nil\ \textstyle\bigwedge_i y_1 \bigwedge_i y_2 (p\, y_1\, t_1 \Rightarrow p\, y_2\, t_2 \Rightarrow \langle p\, y_2\, t_3 \rangle) \longrightarrow t_2 \equiv t_3 \ .$$

There is nothing more that we can do on the right, since the definitional clause for $\equiv$ does not apply. Applying $\exists\mathcal{L}$ gives us the sequent

$$seq_i\ nil\ \textstyle\bigwedge_i y_1 \bigwedge_i y_2 (p\, y_1\, t_1 \Rightarrow p\, y_2\, t_2 \Rightarrow \langle p\, y_2\, t_3 \rangle) \longrightarrow t_2 \equiv t_3 \ .$$

Now we want to reason about the derivation of $\bigwedge y_1 \bigwedge y_2 \ldots$ to conclude that $t_2 \equiv t_3$. In the informal proof, we reasoned that this derivation must end with the right rule for $\bigwedge$; we do the same thing here using $def\,\mathcal{L}$, which yields the sequent

$$\forall y_1\ seq_{i_1}\ nil\ \textstyle\bigwedge_i y_2 v (p\, y_1\, t_1 \Rightarrow p\, y_2\, t_2 \Rightarrow \langle p\, y_2\, t_3 \rangle) \longrightarrow t_2 \equiv t_3 \ ,$$

as well as seven other sequents corresponding to the cases where the object logic derivation ends with the application of one of the left rules. Since these latter seven sequents represent cases that are not applicable, they are easily derivable as shown in the previous example; we thus focus on the sequent shown above. Before we can proceed to apply $def\,\mathcal{L}$ again for the second use of the right rule for $\bigwedge$, we must first apply $\forall\mathcal{L}$, which requires supplying a substitution term for y_1. For this proof, it doesn't matter what term we use for y_1, as long as it is something that does not unify with the term we supply for y_2. So let x_1 and x_2 be two distinct, non-unifiable terms of type i. If we use x_1 for y_1, and then apply $def\,\mathcal{L}$ and $\forall\mathcal{L}$ again using x_2 for y_2, we get

$$seq_{i_2}\ nil\ (p\, x_1\, t_1 \Rightarrow p\, x_2\, t_2 \Rightarrow \langle p\, x_2\, t_3 \rangle) \longrightarrow t_2 \equiv t_3 \ .$$

We now apply $def\mathcal{L}$ two more times, each of which corresponds to reasoning that the object logic derivation must proceed with a use of the right rule for $\Rightarrow$. This yields the sequent

$$seq_{i_3}\ ((p\,x_2\,t_2)::(p\,x_1\,t_1)::nil)\ \langle p\,x_2\,t_3\rangle \longrightarrow t_2 \equiv t_3\ .$$

Another application of $def\mathcal{L}$ reflects the fact that in the object logic derivation only the initial rule now applies:

$$element\ (p\,x_2\,t_3)\ ((p\,x_2\,t_2)::(p\,x_1\,t_1)::nil) \longrightarrow t_2 \equiv t_3\ .$$

For $p\,x_2\,t_3$ to be the first element of the list, t_2 and t_3 must be the same, and this is what we want to prove. We have chosen x_1 and x_2 to be terms that do not unify, so $p\,x_2\,t_3$ cannot be the other element of the list. This reasoning is represented formally by the $FO\lambda^{\Delta\mathbb{N}}$ derivation

$$\dfrac{\dfrac{\dfrac{}{\top \longrightarrow \top}\ \top\mathcal{R}}{\top \longrightarrow t_2 \equiv t_2}\ def\mathcal{R} \quad \dfrac{\dfrac{}{element\ (p\,x_2\,t_3)\ nil \longrightarrow t_2 \equiv t_3}\ def\mathcal{L}}{element\ (p\,x_2\,t_3)\ ((p\,x_1\,t_1)::nil) \longrightarrow t_2 \equiv t_3}\ def\mathcal{L}}{element\ (p\,x_2\,t_3)\ ((p\,x_2\,t_2)::(p\,x_1\,t_1)::nil) \longrightarrow t_2 \equiv t_3}\ def\mathcal{L}\ .$$

If we are able to construct the two non-unifiable terms x_1 and x_2, we are able to conduct this analysis in $FO\lambda^{\Delta\mathbb{N}}$. But the need for these two terms is rather disturbing. The informal proof is independent of the type of y_1 and y_2 and the term structure of this type. In fact, the informal proof is valid even for a type that is uninhabited; this is obviously not the case for our representation in $FO\lambda^{\Delta\mathbb{N}}$. The problem is that our representation of object-level quantification in terms of $FO\lambda^{\Delta\mathbb{N}}$ quantification doesn't allow us to examine a derivation that is generic over certain terms. Although the formula $\forall y\ seq_i\ L\ (B\,y)$ indicates that the proposition $B\,y$ is derivable from the hypotheses in L for any y, it does not indicate that the derivation is the same for all y, and we cannot examine that derivation generically. All we can do is use the $\forall\mathcal{L}$ rule, which requires us to substitute a specific term for y, and then examine the derivation for that specific term. This is analagous to the problem we encountered before related to the encoding of object logic implication in terms of $FO\lambda^{\Delta\mathbb{N}}$ implication.

5.1.4 Explicit Eigenvariable Encoding

To solve this problem we must explicitly keep track of the eigenvariables introduced by the quantifier rules. We do not wish to abandon, however, our higher-order abstract syntax representation of quantification. In the earlier encodings of this section, we encoded the rules for object logic quantification using $FO\lambda^{\Delta\mathbb{N}}$ quantification; the key idea of our solution is to replace that use of $FO\lambda^{\Delta\mathbb{N}}$ quantification with the use of $FO\lambda^{\Delta\mathbb{N}}$ λ-abstraction. If we follow this idea naively and simply replace the quantification by λ-abstraction, we get the following encoding of the right rule for $\bigwedge$:

$$seq_{(s\ I)}\ L\ (\bigwedge_i B) \quad \triangleq \quad \lambda x\ seq_I\ L\ (B\ x)\ .$$

This does not work, of course, since the body of this clause now has type $i \to o$ instead of type o. To address this problem, it is important to first realize that as more eigenvariables are added and propositions are moved between the left and right sides of the sequent, we must deal more generally with "judgements" of the form

$$\lambda x_1 \ldots \lambda x_n\ seq_I\ (L\ x_1 \ldots x_n)\ (B\ x_1 \ldots x_n)$$

for arbitrary $n \geq 0$. First consider "uncurrying" this expression by replacing the λ-abstractions over $x_1, \ldots, x_n$ by a single λ-abstraction over the n-tuple $(x_1, \ldots, x_n)$:

$$\lambda x.seq_I\ (L\ (\pi_1\ x) \ldots (\pi_n\ x))\ (B\ (\pi_1\ x) \ldots (\pi_n\ x))\ .$$

Now we can deal with the arbitrary n by replacing the n-tuple with a list, and using $fst\ x$ in place of $\pi_1\ x$, $fst\ (rst\ x)$ in place of $\pi_2\ x$, $fst\ (rst\ (rst\ x))$ in place of $\pi_3\ x$, etc. Finally, we push the λ-abstraction into the seq predicate by changing its type:

$$seq \quad : \quad nt \to (evs \to prplst) \to (evs \to prp) \to o\ .$$

Here evs is a new type representing a list of eigenvariables. We have already seen the two operators on this type, $fst\colon evs \to i$ and $rst\colon evs \to evs$; $fst\ l$ represents the first eigenvariable in the list l, and $rst\ l$ represents the remainder of the list. The right rule for $\bigwedge$ is now encoded as follows:

$$seq_{(s\ I)}\ L\ (\lambda l\ \bigwedge_i (B\ l)) \quad \triangleq \quad seq_I\ (\lambda l'\ L(rst\ l'))\ (\lambda l'\ B\ (rst\ l')\ (fst\ l'))\ .$$

The bound variable l' in the body of the clause should be thought of as a list whose length is one longer than the length of the bound variable l in the head of the clause; $\mathit{fst}\ l'$ represents the new eigenvariable, and $\mathit{rst}\ l'$ represents the eigenvariables in l. The left rule for $\bigvee_i$ is similarly modified:

$$\begin{aligned} \mathit{seq}_{(s\ I)}\ L\ C \quad &\triangleq \quad \exists b(\mathit{element}\ (\lambda l \bigvee_i (b\, l))\ L \wedge \\ &\qquad \mathit{seq}_I\ (\lambda l'\ (b\, (\mathit{rst}\ l')\, (\mathit{fst}\ l'))::(L\, (\mathit{rst}\ l')))\ (\lambda l'\ C\, (\mathit{rst}\ l'))\ . \end{aligned}$$

The remainder of the clauses are only modified to reflect the change in the type of *seq*. Note in particular that $FO\lambda^{\Delta\mathbb{N}}$ quantification can still be used in the encodings of the left rule for $\bigwedge$ and the right rule for $\bigvee$; since these rules do not introduce eigenvariables, this use of $FO\lambda^{\Delta\mathbb{N}}$ quantification is not problematic. The type of the predicate *element* also changes:

$$\mathit{element} \quad : \quad (\mathit{evs} \to \mathit{prp}) \to (\mathit{evs} \to \mathit{prplst}) \to o\ .$$

Table 5.4 presents the definition for the entire logic.

Since we have not changed the representation of quantification, we get α-equivalence of quantified object logic formulas and substitution for object logic bound variables from the relevant features of $FO\lambda^{\Delta\mathbb{N}}$ as before. Substitution for eigenvariables is a little more involved, as shown by its encoding via the predicates

$$\begin{aligned} \mathit{subst} \quad &: \quad \mathit{nt} \to (\mathit{evs} \to i) \to (\mathit{evs} \to i) \to (\mathit{evs} \to i) \to o \\ \mathit{subst}_0 \quad &: \quad \mathit{nt} \to (\mathit{evs} \to \mathit{evs} \to i) \to (\mathit{evs} \to \mathit{evs} \to i) \to (\mathit{evs} \to \mathit{evs} \to i) \to o\ . \end{aligned}$$

The judgement $\mathit{subst}\ i\ t_1\ t_2\ t_2'$ indicates that t_2' is the result of substituting t_1 in t_2 for the $(i+1)^{\text{th}}$ eigenvariable. We could just as easily use the actual encoding $(\mathit{fst}\ (\mathit{rst}^i\ l))$ of the $(i+1)^{\text{th}}$ eigenvariable in place of its index, but we find it more convenient to use the index so that we can perform induction on it. (Here we use $(\mathit{rst}^i\ l)$ for n applications of *rst* to l, i.e. $(\mathit{rst}^0\ l)$ is l, $(\mathit{rst}^1\ l)$ is $(\mathit{rst}\ l)$, $(\mathit{rst}^2\ l)$ is $(\mathit{rst}\ (\mathit{rst}\ l))$, etc.) The subst_0 predicate is used in the definition of *subst*; the extra *evs* argument is used to keep track of eigenvariables at the beginning of the list as we search down the list for the substitution variable. The encoding of these predicates is shown in Table 5.5. Substitution for the first eigenvariable can be done directly; to substitute for the $(i+2)^{\text{th}}$ eigenvariable we move the first eigenvariable from the list l to the list l' and substitute for the $(i+1)^{\text{th}}$ eigenvariable of l.

Table 5.4: Explicit eigenvariable encoding of intuitionistic logic

$$
\begin{array}{rcl}
seq_I\ L\ \lambda l\ \langle (A\,l)\rangle & \triangleq & element\ \lambda l\ \langle (A\,l)\rangle\ L \\
 & & \\
seq_I\ L\ (\lambda l\ tt) & \triangleq & \top \\
seq_{(s\ I)}\ L\ \lambda l\ ((B\,l)\,\&\,(C\,l)) & \triangleq & seq_I\ L\ B\ \wedge\ seq_I\ L\ C \\
seq_{(s\ I)}\ L\ \lambda l\ ((B\,l)\oplus(C\,l)) & \triangleq & seq_I\ L\ B \\
seq_{(s\ I)}\ L\ \lambda l\ ((B\,l)\oplus(C\,l)) & \triangleq & seq_I\ L\ C \\
seq_{(s\ I)}\ L\ \lambda l\ ((B\,l)\Rightarrow(C\,l)) & \triangleq & seq_I\ \lambda l\ ((B\,l)::(L\,l))\ C \\
seq_{(s\ I)}\ L\ (\lambda l \bigwedge_i (B\,l)) & \triangleq & seq_I\ (\lambda l'\ L\,(rst\ l'))\ (\lambda l'\ B\,(rst\ l')\,(fst\ l')) \\
seq_{(s\ I)}\ L\ (\lambda l \bigvee_i (B\,l)) & \triangleq & \exists x\ seq_I\ L\ (\lambda l\ B\,l\,(x\,l)) \\
 & & \\
seq_I\ L\ B & \triangleq & element\ (\lambda l\ ff)\ L \\
seq_{(s\ I)}\ L\ D & \triangleq & \exists b \exists c (element\ \lambda l\ ((b\,l)\,\&\,(c\,l))\ L \wedge \\
 & & \quad seq_I\ \lambda l\ ((b\,l)::(L\,l))\ D) \\
seq_{(s\ I)}\ L\ D & \triangleq & \exists b \exists c (element\ \lambda l\ ((b\,l)\,\&\,(c\,l))\ L \wedge \\
 & & \quad seq_I\ \lambda l\ ((c\,l)::(L\,l))\ D) \\
seq_{(s\ I)}\ L\ D & \triangleq & \exists b \exists c (element\ \lambda l\ ((b\,l)\oplus(c\,l))\ L \wedge \\
 & & \quad seq_I\ \lambda l\ ((b\,l)::(L\,l))\ D\ \wedge\ seq_I\ \lambda l\ ((c\,l)::(L\,l))\ D) \\
seq_{(s\ I)}\ L\ D & \triangleq & \exists b \exists c (element\ \lambda l\ ((b\,l)\Rightarrow(c\,l))\ L \wedge \\
 & & \quad seq_I\ \lambda l\ ((c\,l)::(L\,l))\ D \wedge \\
 & & \quad seq_I\ L\ b) \\
seq_{(s\ I)}\ L\ C & \triangleq & \exists b (element\ (\lambda l \bigwedge_i (b\,l))\ L \wedge \\
 & & \quad \exists x\ seq_I\ \lambda l\ ((b\,l\,(x\,l))::(L\,l))\ C) \\
seq_{(s\ I)}\ L\ C & \triangleq & \exists b (element\ (\lambda l \bigvee_i (b\,l))\ L \wedge \\
 & & \quad seq_I\ \lambda l'\ ((b\,(rst\ l')\,(fst\ l'))::(L\,(rst\ l')))\ (\lambda l'\ C\,(rst\ l'))) \\
 & & \\
element\ X\ \lambda l\ ((X\,l)::(L\,l)) & \triangleq & \top \\
element\ X\ \lambda l\ ((Y\,l)::(L\,l)) & \triangleq & element\ X\ L
\end{array}
$$

Table 5.5: Encoding of substitution for eigenvariables

$$
\begin{array}{lcl}
\mathit{subst}\ I\ T_1\ T_2\ T_2' & \triangleq & \mathit{subst}_0\ I\ (\lambda l'\, T_1)\ (\lambda l'\, T_2)\ (\lambda l'\, T_2') \\
\\
\mathit{subst}_0\ z\ T_1\ (\lambda l' \lambda l\, T_2\, l'\, (\mathit{fst}\ l)\, (\mathit{rst}\ l))\ (\lambda l' \lambda l\, T_2\, l'\, (T_1\, l'\, l)\, (\mathit{rst}\ l)) & & \\
\qquad \triangleq\ \top & & \\
\mathit{subst}_0\ (s\ I)\ (\lambda l' \lambda l\, T_1\, l'\, (\mathit{fst}\ l)\, (\mathit{rst}\ l)) & & \\
\qquad (\lambda l' \lambda l\, T_2\, l'\, (\mathit{fst}\ l)\, (\mathit{rst}\ l))\ (\lambda l' \lambda l\, T_2'\, l'\, (\mathit{fst}\ l)\, (\mathit{rst}\ l)) & & \\
\qquad \triangleq\ \mathit{subst}_0\ I\ (\lambda l' \lambda l\, T_1\, (\mathit{rst}\ l')\, (\mathit{fst}\ l')\, l) & & \\
\qquad\qquad (\lambda l' \lambda l\, T_2\, (\mathit{rst}\ l')\, (\mathit{fst}\ l')\, l)\ (\lambda l' \lambda l\, T_2'\, (\mathit{rst}\ l')\, (\mathit{fst}\ l')\, l) &
\end{array}
$$

As with the previous encoding of intuitionistic logic, we must derive the admissibility of the structural rules and the cut rule by induction. We have retained the atomic encoding of sequents, so we can still analyze derivations of propositions from hypotheses. In addition, the explicit encoding of eigenvariables allows us to better analyze derivations of generic propositions. To see this, we revisit the example from before; the sequent we wish to derive is

$$\longrightarrow \forall p \forall t_1 \forall t_2 \forall t_3 (\exists i\ \mathit{seq}_i\ \lambda l\ \mathit{nil}\ (\lambda l \bigwedge_i y_1 \bigwedge_i y_2 (p\, y_1\, t_1 \Rightarrow p\, y_2\, t_2 \Rightarrow \langle p\, y_2\, t_3 \rangle))) \supset t_2 \equiv t_3) \ .$$

As before, we begin by applying the $\forall\mathcal{R}$, $\supset\mathcal{R}$, and $\exists\mathcal{L}$ rules to obtain the sequent

$$\mathit{seq}_i\ \lambda l\ \mathit{nil}\ (\lambda l \bigwedge_i y_1 \bigwedge_i y_2 (p\, y_1\, t_1 \Rightarrow p\, y_2\, t_2 \Rightarrow \langle p\, y_2\, t_3 \rangle)) \longrightarrow t_2 \equiv t_3 \ .$$

The derivation of the object logic formula $\bigwedge y_1 \bigwedge y_2 \ldots$ must end with two applications of the right rule for $\bigwedge$; we formalize this by applying $\mathit{def}\mathcal{L}$ twice, which results in the sequent

$$\mathit{seq}_{i_1}\ \lambda l\ \mathit{nil}\ \lambda l\ (p\, (\mathit{fst}\ (\mathit{rst}\ l))\, t_1 \Rightarrow p\, (\mathit{fst}\ l)\, t_2 \Rightarrow \langle p\, (\mathit{fst}\ l)\, t_3 \rangle) \longrightarrow t_2 \equiv t_3 \ .$$

The object logic derivation must proceed with two applications of the right rule for $\Rightarrow$; we deduce this formally by two more applications of the $\mathit{def}\mathcal{L}$ rule, yielding

$$\mathit{seq}_{i_2}\ \lambda l\ ((p\, (\mathit{fst}\ l)\, t_2) :: (p\, (\mathit{fst}\ (\mathit{rst}\ l))\, t_1) :: \mathit{nil})\ \lambda l\ \langle p\, (\mathit{fst}\ l)\, t_3 \rangle \longrightarrow t_2 \equiv t_3 \ .$$

An additional use of the $\mathit{def}\mathcal{L}$ rule corresponds to the realization that the initial rule must complete the object logic derivation, giving us the sequent

$$\mathit{element}\ (\lambda l\ p\, (\mathit{fst}\ l)\, t_3)\ \lambda l\ ((p\, (\mathit{fst}\ l)\, t_2) :: (p\, (\mathit{fst}\ (\mathit{rst}\ l))\, t_1) :: \mathit{nil}) \longrightarrow t_2 \equiv t_3 \ .$$

If $p\,(\mathit{fst}\ l)\,t_3$ is the first element of the list, then t_2 and t_3 are the same, which is the result we are trying to establish. The formula $p\,(\mathit{fst}\ l)\,t_3$ cannot be the other element of the list, because the first argument to p differs; thus we are done. This is all formally encoded by the derivation

$$\dfrac{\dfrac{\dfrac{}{\top \longrightarrow \top}\ \top\mathcal{R}}{\top \longrightarrow t_2 \equiv t_2}\ def\mathcal{R} \quad \dfrac{\dfrac{}{\mathit{element}\ (\lambda l\ p\,(\mathit{fst}\ l)\,t_3)\ (\lambda l\ \mathit{nil}) \longrightarrow t_2 \equiv t_3}\ def\mathcal{L}}{\mathit{element}\ (\lambda l\ p\,(\mathit{fst}\ l)\,t_3)\ \lambda l\ ((p\,(\mathit{fst}\ (\mathit{rst}\ l))\,t_1) :: \mathit{nil}) \longrightarrow t_2 \equiv t_3}\ def\mathcal{L}}{\mathit{element}\ (\lambda l\ p\,(\mathit{fst}\ l)\,t_3)\ \lambda l\ ((p\,(\mathit{fst}\ l)\,t_2) :: (p\,(\mathit{fst}\ (\mathit{rst}\ l))\,t_1) :: \mathit{nil}) \longrightarrow t_2 \equiv t_3}\ def\mathcal{L}\ .$$

5.1.5 Discussion

Before going on to formally derive theorems about encodings of logics, let us reflect on the encoding styles we have discussed. What we have is a spectrum of styles, all of which share the same higher-order abstract syntax encoding of formulas, but which vary in the degree to which they use the higher-order abstract syntax encoding of inference rules. The first encoding used the typical higher-order abstract syntax techniques, which made a number of significant properties of the object logic fall out easily from the properties of $FO\lambda^{\Delta\mathbb{N}}$. Unfortunately this encoding did not lend itself to formal analysis within $FO\lambda^{\Delta\mathbb{N}}$, since it could not be expressed as a definition nor given an induction measure. We then progressed through three other encodings, each of which compromised the use higher-order abstract syntax a bit more. The cost of each compromise was a decrease in the elegance and an increase in the complexity of the encoding, and a reduction in the extent to which fundamental properties of the object logic followed from corresponding properties of $FO\lambda^{\Delta\mathbb{N}}$. The benefit, of course, was a greater ability to perform formal meta-theoretic analysis.

In Chapter 6 we will discuss an approach which lets us use the typical higher-order abstract syntax encodings and also perform meta-theoretic analyses on these encodings. The key to this approach is the use of a specification logic that is separate from $FO\lambda^{\Delta\mathbb{N}}$, and in fact is itself specified in $FO\lambda^{\Delta\mathbb{N}}$. In the next section we present two logics which will be used for this purpose, and which also serve as examples of the last two encoding techniques discussed in this section.

5.2 Examples

In this section we illustrate the use of the some of the encoding techniques just presented. In Section 5.2.1 we use the explicit sequent technique of Section 5.1.3 to encode a fragment of intuitionistic logic; Section 5.2.2 discusses a fragment of linear logic encoded with the explicit eigenvariable technique of Section 5.1.4. In each case we prove the adequacy of the encoding and also derive in $FO\lambda^{\Delta\mathbb{N}}$ some properties of the object logic. The results of Section 5.2.1 were presented in [31], coauthored with Miller.

5.2.1 Intuitionistic Logic

Consider the fragment of second-order intuitionistic logic given by the grammar

$$\begin{array}{rcl} D & ::= & A \mid G \Rightarrow A \mid \bigwedge_\alpha x.D \mid \bigwedge_{\alpha\to\alpha} x.D \\ G & ::= & A \mid tt \mid G \,\&\, G \mid A \Rightarrow G \mid \bigwedge_\alpha x.G \ , \end{array}$$

where A ranges over atomic formulas and α ranges over ground types. D and G represent definite clauses and goal formulas, respectively. Although this seems like a rather simple fragment, higher-order abstract syntax encodings generally fall within the set of definite clauses given by this grammar. The set of goal formulas can be encoded by the following constants:

$$\begin{array}{llllll} \langle\,\rangle & : & atm \to prp & \& & : & prp \to prp \to prp \qquad \bigwedge_i \; : \; (i \to prp) \to prp \\ tt & : & prp & \Rightarrow & : & atm \to prp \to prp \qquad . \end{array}$$

Notice that the antecendent of the implication is restricted to be atomic.

If we take any sequent calculus inference rule and restrict the conclusion to be a sequent whose antecedents are definite clauses and whose consequent is a goal formula, then the premises will also be sequents of this form. In fact, any antecedent in the premises will either be an antecedent of the conclusion or an atomic formula. Thus in a derivation in this fragment of intuitionistic logic, all non-atomic antecedents in any sequent of the derivation appear as antecedents in the end-sequent. So we can divide the antecedents into the original theory, which remains constant throughout the derivation, and some atomic antecedents, which vary throughout the derivation. Leaving the fixed theory aside for the moment, we

can restrict our sequents to have only atomic antecedents:

$$seq \quad : \quad nt \to atmlst \to prp \to o \ ,$$

where *atmlst* is the same as the type *lst* introduced in Section 2.2.2, using *atm* for the type of elements. Since the antecedents are atomic, only the initial and right rules are necessary:

$$\begin{aligned}
seq_I\ (A' :: L)\ \langle A \rangle &\triangleq element\ A\ (A' :: L) \\
seq_I\ L\ tt &\triangleq \top \\
seq_{(s\ I)}\ L\ (B\ \&\ C) &\triangleq seq_I\ L\ B\ \wedge\ seq_I\ L\ C \\
seq_{(s\ I)}\ L\ (A \Rightarrow B) &\triangleq seq_I\ (A :: L)\ B \\
seq_{(s\ I)}\ L\ (\textstyle\bigwedge_i B) &\triangleq \forall_i x\ seq_I\ L\ (B\ x)\ .
\end{aligned}$$

We now turn to consider the set of definite clauses that make up the theory for the derivation. Notice that the atomic formula A is equivalent to the formula $tt \Rightarrow A$, so every definite clause can be written in the form $\bigwedge x_1 \cdots \bigwedge x_n (G \Rightarrow A)$. In addition, the logic under consideration is a subset of the logic of hereditary Harrop formulas. As a result, for any derivable sequent there is a uniform derivation of that sequent [34, 39]. In our setting, a derivation is uniform if every subderivation ending in a left rule is of the form

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \longrightarrow G[t_1, \ldots, t_n/x_1, \ldots, x_n]} \quad \dfrac{}{A', \Gamma \longrightarrow A'}\ init}{(G \Rightarrow A)[t_1, \ldots, t_n/x_1, \ldots, x_n], \Gamma \longrightarrow A'}\ \Rightarrow \mathcal{L}}{\bigwedge x_1 \cdots \bigwedge x_n (G \Rightarrow A), \Gamma \longrightarrow A'}\ \bigwedge \mathcal{L} \quad ,$$

where A' and $A[t_1, \ldots, t_n/x_1, \ldots, x_n]$ are the same. If we group these steps together, our aggregate left rule encoding needs to say that $seq_{(s\ I)}\ L\ \langle A' \rangle$ holds if and only if there is a clause $\bigwedge x_1 \cdots \bigwedge x_n (G \Rightarrow A)$ in the theory such that A can be instantiated to match A', and $seq_{(s\ I)}\ L\ \langle G' \rangle$ holds, where G' is the corresponding instantiation of G. We use the predicate

$$prog \quad : \quad atm \to prp \to o$$

to encode the theory. The fact that the definite clause $\bigwedge x_1 \cdots \bigwedge x_n (G \Rightarrow A)$ is in the theory is represented by the definitional clause $prog\ A\ G \triangleq \top$; the quantification of the definite clause is encoded by the (elided) quantification of the definitional clause. The encoding for

the aggregate left rule is

$$seq_{(s\ I)}\ L\ \langle A\rangle \ \stackrel{\triangle}{=}\ \exists b(prog\ A\ b \wedge seq_I\ L\ b)\ ;$$

notice that the matching between A and the head of the definite clause is accomplished by the definition rules. Different object-level theories can be considered by varying the definition of *prog*, as illustrated in Chapter 6. The object-level formulas encoded using *prog* are treated by the object logic as a theory and not as a definition: there is no rule corresponding to $FO\lambda^{\Delta\mathbb{N}}$'s $def\mathcal{L}$ in the object logic.

We will refer to the six clauses for *seq* given in this section as $\mathcal{D}(intuit)$. For convenience we will abbreviate the formula $\exists i(nat\ i \wedge seq_i\ L\ B)$ as $L \rhd B$ (or as $\rhd B$ when L is *nil*). We now state the following properties about this presentation of the object logic. If B is a term of type *prp*, then let $\langle\!|B|\!\rangle$ be its (obvious) translation into a formula of intuitionistic logic. If L is a term of type *atmlst*, let $\langle\!|L|\!\rangle$ be its (obvious) translation to a multiset of atomic formulas of intuitionistic logic.

Theorem 5.1 *Let $\mathcal{D}(prog)$ be the definition $\{\forall\bar{x}_1[prog\ A_1\ G_1 \stackrel{\triangle}{=} \top], \ldots, \forall\bar{x}_n[prog\ A_n\ G_n \stackrel{\triangle}{=} \top]\}$ ($n \geq 0$) which represents an object-level theory, and let $\mathcal{P}$ be the corresponding theory in intuitionistic logic (i.e. the set of formulas $\bigwedge \bar{x}_i(\langle\!|G_i|\!\rangle \Rightarrow \langle\!|A_i|\!\rangle)$, for all $i \in \{1, \ldots, n\}$). Let $\mathcal{D}$ be a definition that extends $\mathcal{D}(nat) \cup \mathcal{D}(list(\mathrm{atm})) \cup \mathcal{D}(intuit) \cup \mathcal{D}(prog)$ with clauses that do not define nat, seq, element, or prog. Then the sequent $\longrightarrow L \rhd B$ is derivable in $FO\lambda^{\Delta\mathbb{N}}$ with definition $\mathcal{D}$ if and only if $\langle\!|B|\!\rangle$ is an intuitionistic consequence of $\langle\!|L|\!\rangle \cup \mathcal{P}$.*

Proof The reverse direction follows easily from the definition $\mathcal{D}(intuit)$. For the forward direction, the use of the $def\mathcal{R}$ rule with $\mathcal{D}(intuit)$ will cause the structure of the $FO\lambda^{\Delta\mathbb{N}}$ derivation to closely follow that of the corresponding derivation in intuitionistic logic. However, we need to be sure that the $nat\mathcal{L}$ and $def\mathcal{L}$ rules don't allow us to derive anything that we can't derive in intuitionistic logic. By Theorem 3.13, we only need to consider cut-free $FO\lambda^{\Delta\mathbb{N}}$ derivations. By Lemma 5.2 a cut-free derivation of $\longrightarrow L \rhd B$ will consist only of sequents with empty antecedents. Thus the $nat\mathcal{L}$ and $def\mathcal{L}$ rules are not used, since they both require a formula in the antecedent. ∎

Lemma 5.2 *If $\mathcal{D}$ is an implication-free definition, C is an implication-free formula, and Π is a cut-free derivation of the sequent $\longrightarrow C$ in $FO\lambda^{\Delta\mathbb{N}}$ using $\mathcal{D}$, then every sequent in Π has an empty antecedent and an implication-free consequent.*

Proof Consider the following two observations about the inference rules in $FO\lambda^{\Delta\mathbb{N}}$:

1. The only rule that can have a non-empty antecedent in a premise and an empty antecedent in the conclusion is $\supset \mathcal{R}$.

2. The only rules that allow an implication in the consequent of a premise without one in the consequent of the conclusion are $\supset \mathcal{L}$ and $nat\mathcal{L}$. (The $def\mathcal{R}$ rule does not, since the clauses in $\mathcal{D}$ do not contain implications. The $def\mathcal{L}$ rule does not, since the types of variables cannot contain o.)

But the $\supset \mathcal{L}$ and $nat\mathcal{L}$ rules have conclusions with a non-empty antecedent, and the $\supset \mathcal{R}$ rule has a conclusion with an implication in the consequent. Since the end-sequent has an empty antecedent and an implication-free consequent, a simple induction on the height of the proof shows that all sequents in the proof have this property. ∎

The following theorems state that we can derive in $FO\lambda^{\Delta\mathbb{N}}$ that the specialization rule, the cut rule and the usual structural rules are admissible for our object logic.

Theorem 5.3 *The formula*

$$\forall i \forall b \forall l (nat\ i\ \supset\ seq_{(s\ i)}\ l\ \bigwedge b\ \supset\ \forall x\, seq_i\ l\ (b\, x))$$

is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition $\mathcal{D}(nat) \cup \mathcal{D}(list(\mathrm{atm})) \cup \mathcal{D}(intuit)$.

Theorem 5.4 *The formula $\forall a \forall b \forall l((a :: l) \rhd b\ \supset\ l \rhd \langle a \rangle\ \supset\ l \rhd b)$ is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition $\mathcal{D}(nat) \cup \mathcal{D}(list(\mathrm{atm})) \cup \mathcal{D}(intuit)$.*

Theorem 5.5 *The formula*

$$\forall i \forall b \forall l \forall l' (nat\ i\ \supset\ \forall a (element\ a\ l\ \supset\ element\ a\ l')\ \supset\ seq_i\ l\ b\ \supset\ seq_i\ l'\ b)$$

is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition $\mathcal{D}(nat) \cup \mathcal{D}(list(\mathrm{atm})) \cup \mathcal{D}(intuit)$.

Table 5.6: Inference rules for a fragment of intuitionistic linear logic

$$\frac{}{\Gamma; A \longrightarrow A}\ \mathit{initial} \qquad \frac{B, \Gamma; B, \Delta \longrightarrow C}{B, \Gamma; \Delta \longrightarrow C}\ \mathit{absorb} \qquad \frac{}{\Gamma; \Delta \longrightarrow tt}\ tt\mathcal{R}$$

$$\frac{\Gamma; B, \Delta \longrightarrow E}{\Gamma; B \,\&\, C, \Delta \longrightarrow E}\ \&\mathcal{L} \qquad \frac{\Gamma; C, \Delta \longrightarrow E}{\Gamma; B \,\&\, C, \Delta \longrightarrow E}\ \&\mathcal{L} \qquad \frac{\Gamma; B[t/x], \Delta \longrightarrow C}{\Gamma; \bigwedge x.B, \Delta \longrightarrow C}\ \bigwedge\mathcal{L}$$

$$\frac{\Gamma; \Delta \longrightarrow B \quad \Gamma; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \,\&\, C}\ \&\mathcal{R} \qquad \frac{\Gamma; \Delta \longrightarrow B[y/x]}{\Gamma; \Delta \longrightarrow \bigwedge x.B}\ \bigwedge\mathcal{R}$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \quad \Gamma; A, \Delta_2 \longrightarrow C}{\Gamma; B \multimap A, \Delta_1, \Delta_2 \longrightarrow C}\ \multimap\mathcal{L} \qquad \frac{\Gamma; A, \Delta \longrightarrow B}{\Gamma; \Delta \longrightarrow A \multimap B}\ \multimap\mathcal{R}$$

$$\frac{\Gamma; \longrightarrow B \quad \Gamma; A, \Delta \longrightarrow C}{\Gamma; B \Rightarrow A, \Delta \longrightarrow C}\ \Rightarrow\mathcal{L} \qquad \frac{A, \Gamma; \Delta \longrightarrow B}{\Gamma; \Delta \longrightarrow A \Rightarrow B}\ \Rightarrow\mathcal{R}$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \quad \Gamma; B, \Delta_2 \longrightarrow C}{\Gamma; \Delta_1, \Delta_2 \longrightarrow C}\ \mathit{cut} \qquad \frac{\Gamma; \longrightarrow B \quad B, \Gamma; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow C}\ \mathit{cut}!$$

5.2.2 Linear Logic

Now consider the fragment of second-order linear logic given by the grammar

$$\begin{aligned} D &::= A \mid G \multimap A \mid G \Rightarrow A \mid \textstyle\bigwedge_\alpha x.D \mid \bigwedge_{\alpha\to\alpha} x.D \\ G &::= A \mid tt \mid G \,\&\, G \mid A \multimap G \mid A \Rightarrow G \mid \textstyle\bigwedge_\alpha x.G\ , \end{aligned}$$

where A ranges over atomic formulas and α ranges over ground types. As in Section 5.2.1, D and G represent definite clauses and goal formulas, respectively. A derivation system for this logic is shown in Table 5.6; the left side of the sequent is divided into a set Γ of intuitionistic (non-linear) antecedents and a multiset Δ of linear antecedents. In the $\bigwedge\mathcal{R}$ rule, y is an eigenvariable that is not free in the lower sequent of the rule. We encode the set of goal formulas using the following constants:

$$\begin{array}{llllllllll} \langle\,\rangle & : & atm \to prp & \quad \& & : & prp \to prp \to prp & \quad \Rightarrow & : & atm \to prp \to prp \\ tt & : & prp & \quad \multimap & : & atm \to prp \to prp & \quad \bigwedge_i & : & (i \to prp) \to prp \quad . \end{array}$$

We again separate the antecedents of sequents in a derivation into a theory, which remains constant throughout the derivation and is encoded via a predicate *prog*, and some

atomic antecedents, which vary from sequent to sequent in the derivation and are shown explicitly in the sequent. The atomic antecedents are further divided into linear and intuitionistic antecedents:

$$seq \quad : \quad nt \to (evs \to atmlst) \to (evs \to atmlst) \to (evs \to prp) \to o \ .$$

The second and third arguments to *seq* represent multisets of intuitionistic and linear antecedents, respectively. Notice that we follow the explicit eigenvariable encoding style of Section 5.1.4 by encoding the antecedents and consequent as funtions whose domain is a list of eigenvariables. In order to highlight both the similarities and differences between this encoding and the encoding of Section 5.2.1, we will use a number of abbreviations; we introduce the first of these now. For any type τ, we will use τ^* as an abbreviation for $evs \to \tau$. Thus the type of *seq* above can be expressed as

$$seq \quad : \quad nt \to atmlst^* \to atmlst^* \to prp^* \to o \ .$$

We must modify the definition $\mathcal{D}(list(\tau))$ from Section 2.2.2 to work over the type lst^*. The predicates will now have the following types:

$$\begin{array}{rclcrcl} length & : & lst^* \to nt \to o & \qquad & split & : & lst^* \to lst^* \to lst^* \to o \\ list & : & lst^* \to o & & permute & : & lst^* \to lst^* \to o \\ element & : & \tau^* \to lst^* \to o & & & & \end{array} \ .$$

The new definition $\mathcal{D}(list^*(\tau))$ is shown in Table 5.7; we use nil^* and $A ::^* L$ as abbreviations for $\lambda l\, nil$ and $\lambda l\, ((A\, l) :: (L\, l))$.

We similarly introduce abbreviations corresponding to constructors of prp^*: $\langle A \rangle^*$ abbreviates $\lambda l\, \langle A\, l \rangle$, tt^* abbreviates $\lambda l\, tt$, $B \,\&^*\, C$ abbreviates $\lambda l\, ((B\, l) \& (C\, l))$, $A \multimap^* B$ abbreviates $\lambda l\, ((A\, l) \multimap (B\, l))$, $A \Rightarrow^* B$ abbreviates $\lambda l\, ((A\, l) \Rightarrow (B\, l))$, and $\bigwedge^* B$ abbreviates $\lambda l\, (\bigwedge (B\, l))$.

Any definite clause in our fragment of linear logic is equivalent to a formula of the form

$$\bigwedge x_1 \cdots \bigwedge x_k (B_1 \Rightarrow \cdots B_m \Rightarrow C_1 \multimap \cdots C_n \multimap A) \ ,$$

for some $k, m, n \geq 0$ and goal formulas $B_1, \ldots, B_m, C_1, \ldots, C_n$. Uniform derivations have also been shown to be complete for this logic [25]; thus we use the predicate

$$prog \quad : \quad atm^* \to prplst^* \to prplst^* \to o$$

Table 5.7: Explicit eigenvariable encoding of lists

$$
\begin{array}{rcl}
\mathit{length}\ \mathit{nil}^*\ z & \triangleq & \top \\
\mathit{length}\ (A ::^* L)\ (s\ I) & \triangleq & \mathit{length}\ L\ I \\
\\
\mathit{list}\ L & \triangleq & \exists i(\mathit{nat}\ i \wedge \mathit{length}\ L\ i) \\
\\
\mathit{element}\ A\ (A ::^* L) & \triangleq & \top \\
\mathit{element}\ A\ (A' ::^* L) & \triangleq & \mathit{element}\ A\ L \\
\\
\mathit{split}\ \mathit{nil}^*\ \mathit{nil}^*\ \mathit{nil}^* & \triangleq & \top \\
\mathit{split}\ (A ::^* L_1)\ (A ::^* L_2)\ L_3 & \triangleq & \mathit{split}\ L_1\ L_2\ L_3 \\
\mathit{split}\ (A ::^* L_1)\ L_2\ (A ::^* L_3) & \triangleq & \mathit{split}\ L_1\ L_2\ L_3 \\
\\
\mathit{permute}\ \mathit{nil}^*\ \mathit{nil}^* & \triangleq & \top \\
\mathit{permute}\ (A ::^* L_1)\ L_2 & \triangleq & \exists l_{22}(\mathit{split}\ L_2\ (A ::^* \mathit{nil}^*)\ l_{22} \wedge \mathit{permute}\ L_1\ l_{22}\ )
\end{array}
$$

to encode the set of definite clauses that make up the theory. The first argument represents the atomic head of the definite clause; the second and third arguments represent the lists $C_1, \ldots, C_n$ of linear hypotheses and $B_1, \ldots, B_m$ of intuitionistic hypotheses, respectively. The quantification of the definite clause is again encoded by the (elided) quantification of the definitional clause. The predicate

$$\mathit{split\_seq} \quad : \quad \mathit{nt} \to \mathit{atmlst}^* \to \mathit{atmlst}^* \to \mathit{prplst}^* \to o$$

will be used to express the idea that the propositions in the last argument are derivable from the intuitionistic and linear antecedents in the second and third arguments. Each linear antecedent must be used exactly once in the derivation of all propositions in the last list. The inference rules for this logic are encoded in the definition $\mathcal{D}(\mathit{linear})$ of Table 5.8, which defines the predicates *seq* and *split_seq*. In the clause for $\bigwedge_i$ we subscript the constant *fst* with the type i because we will also be introducing a constant $\mathit{fst}_{i \to i}$ for the representation of second-order eigenvariables. As in the previous section, different object-level theories can be considered by varying the definition of *prog*; an example theory will be given in Chapter 6. For convenience we will abbreviate the formula $\exists i(\mathit{nat}\ i \wedge \mathit{seq}_i\ \mathit{IL}\ \mathit{LL}\ B)$ as $\mathit{IL}; \mathit{LL} \rhd B$ (or as $\rhd B$ when IL and LL are nil^*).

Table 5.8: Explicit eigenvariable encoding of linear logic

$$
\begin{array}{rcl}
seq_I\ IL\ (A' ::^*\ nil^*)\ \langle A\rangle^* & \triangleq & \top \\
seq_I\ (A' ::^*\ IL)\ nil^*\ \langle A\rangle^* & \triangleq & element\ \langle A\rangle^*\ (A' ::^*\ IL) \\
seq_{(S\ I)}\ IL\ LL\ \langle A\rangle^* & \triangleq & \exists ll \exists il\ (list\ ll\ \wedge\ list\ il\ \wedge\ prog\ A\ ll\ il\ \wedge \\
 & & \qquad split\_seq_I\ IL\ LL\ ll\ \wedge\ split\_seq_I\ IL\ nil^*\ il) \\
seq_I\ IL\ LL\ tt^* & \triangleq & \top \\
seq_{(S\ I)}\ IL\ LL\ (B\ \&^*\ C) & \triangleq & seq_I\ IL\ LL\ B\ \wedge\ seq_I\ IL\ LL\ C \\
seq_{(S\ I)}\ IL\ LL\ (A \multimap^*\ B) & \triangleq & seq_I\ IL\ (A ::^*\ LL)\ B \\
seq_{(S\ I)}\ IL\ LL\ (A \Rightarrow^*\ B) & \triangleq & seq_I\ (A ::^*\ IL)\ LL\ B \\
seq_{(S\ I)}\ IL\ LL\ (\bigwedge_i B) & \triangleq & seq_I\ (\lambda l\ IL\,(rst\ l))\ (\lambda l\ LL\,(rst\ l))\ (\lambda l\ B\,(rst\ l)\,(fst_i\ l)) \\
 & & \\
split\_seq_I\ IL\ nil^*\ nil^* & \triangleq & \top \\
split\_seq_I\ IL\ LL\ (B ::^*\ L) & \triangleq & \exists ll_1 \exists ll_2\ (split\ LL\ ll_1\ ll_2\ \wedge \\
 & & seq_I\ IL\ ll_1 B\ \wedge\ split\_seq_I\ IL\ ll_2\ L)
\end{array}
$$

We now proceed to prove the adequacy of this encoding. Since the representation is more complex and varies more from the typical higher-order abstract syntax encoding than the one presented in Section 5.2.1, we show its adequacy proof in more detail. We assume that all constants with return type i have at most second-order types built from only i and $\rightarrow$. The restriction to types built from i is reasonable, since we expect the terms of our object logic to only be applied to other terms in the object logic. The second-order restriction reflects the second-order nature of the object logic. We also assume that *rst* is the only constant with return type *evs*, and that *rst*, fst_i, and $fst_{i\rightarrow i}$ are the only constants taking arguments of type *evs*. As a result, all terms of type *evs* will be of the form $(rst^n\ l)$; furthermore, if such a term occurs as a subterm within a term of a different type, it must occur as an argument to either fst_i or $fst_{i\rightarrow i}$.

As a first step toward proving the adequacy of our encoding, we formally define the mappings from closed terms (in $\beta\eta$ long normal form) of type i^*, atm^*, and prp^* to the terms, atoms, and formulas they represent. We assume that we have a canonical list of variables $x_1, x_2, x_3, \ldots$ that do not occur in the term under consideration; we will use these both for the free variables that are represented by the parameter of type *evs* and for bound variables. An upper bound on the number of free variables in a term can be determined

Table 5.9: Decoding function for terms, atoms and goal formulas

$$
\begin{array}{rcl}
\langle\!|\lambda l\, fst_{\upsilon}\ (rst^{m}\, l)|\!\rangle_{n}^{\upsilon} & = & x_{n-m} \\
\langle\!|\lambda l\, c|\!\rangle_{n}^{\tau} & = & c \quad \text{where } c \text{ is a constant symbol} \\
\langle\!|\lambda l \lambda y : \upsilon\ t|\!\rangle_{n}^{\upsilon\to\tau} & = & \lambda x_n.\langle\!|\lambda l'\, t[rst\ l'/l][fst_{\tau}\ l'/y]|\!\rangle_{n+1}^{\tau} \\
\langle\!|\lambda l\, (f\ t)|\!\rangle_{n}^{\tau} & = & \langle\!|\lambda l\, f|\!\rangle_{n}^{\upsilon\to\tau}\ (\langle\!|\lambda l\, t|\!\rangle_{n}^{\upsilon}) \\
 & & \\
\langle\!|\lambda l\, (p\, t_1 \ldots t_m)|\!\rangle_{n}^{\mathrm{atm}} & = & p\ \langle\!|\lambda l\, t_1|\!\rangle_{n}^{\upsilon_1} \ldots \langle\!|\lambda l\, t_m|\!\rangle_{n}^{\upsilon_m} \\
 & & \\
\langle\!|\lambda l\, \langle A\rangle|\!\rangle_{n}^{\mathrm{prp}} & = & \langle\!|\lambda l\, A|\!\rangle_{n}^{\mathrm{atm}} \\
\langle\!|\lambda l\ tt|\!\rangle_{n}^{\mathrm{prp}} & = & tt \\
\langle\!|\lambda l\ (B\ \&\ C)|\!\rangle_{n}^{\mathrm{prp}} & = & \langle\!|\lambda l\ B|\!\rangle_{n}^{\mathrm{prp}}\ \&\ \langle\!|\lambda l\ C|\!\rangle_{n}^{\mathrm{prp}} \\
\langle\!|\lambda l\ (A \multimap B)|\!\rangle_{n}^{\mathrm{prp}} & = & \langle\!|\lambda l\ A|\!\rangle_{n}^{\mathrm{atm}} \multimap \langle\!|\lambda l\ B|\!\rangle_{n}^{\mathrm{prp}} \\
\langle\!|\lambda l\ (A \Rightarrow B)|\!\rangle_{n}^{\mathrm{prp}} & = & \langle\!|\lambda l\ A|\!\rangle_{n}^{\mathrm{atm}} \Rightarrow \langle\!|\lambda l\ B|\!\rangle_{n}^{\mathrm{prp}} \\
\langle\!|\lambda l\ \bigwedge_i(\lambda y\ B)|\!\rangle_{n}^{\mathrm{prp}} & = & \bigwedge x_n.\langle\!|\lambda l'\ B[rst\ l'/l][fst_i\ l'/y]|\!\rangle_{n+1}^{\mathrm{prp}}
\end{array}
$$

as follows: given the term $\lambda l.X$, if X contains occurrences of $(rst^n\, l)$, but does not contain occurrences of $(rst^m\, l)$ for $n < m$, then at most $n+1$ free variables will be needed for the mapping of $\lambda l.X$. We parameterize our mappings by a number n that we assume is no less than this number of free variables in the term under consideration. This n will determine the specific variables from the list $x_1, x_2, \ldots$ that will be used as free variables. The mappings for terms, atoms, and goal formulas are defined in Table 5.9. In this table we use τ and υ to range over types built only from i and $\to$; τ is used when the order of the type may be at most 2, and υ is used when the order may be at most 1. We will generally omit the type superscript when it is evident from the context. The mappings from lists to sets and multisets are obvious extensions of these. The definitional clause

$$\forall y_1 \ldots \forall y_k.[prog\ A\ (C_1 ::^* \cdots C_n ::^* nil^*)\ (B_1 ::^* \cdots B_m ::^* nil^*) \triangleq \top]$$

represents the definite clause

$$\bigwedge x_1 \ldots \bigwedge x_k\ (\langle\!|B_1\theta|\!\rangle_k \Rightarrow \cdots \langle\!|B_m\theta|\!\rangle_k \Rightarrow \langle\!|C_1\theta|\!\rangle_k \multimap \cdots \langle\!|C_n\theta|\!\rangle_k \multimap \langle\!|A\theta|\!\rangle_k)\ ,$$

where θ is the substitution that replaces each variable $y_i : \tau$ by $\lambda l\ fst_{\tau}\ (rst^{k-i}\, l)$. Since we allow first and second order variables in definite clauses, τ is either i or $i \to i$. The constant $fst_{i\to i}$: $evs \to i \to i$ is only used in this translation for definite clauses. Notice that

the quantified variables $y_1, \ldots, y_k$ should be able to match terms containing object-level eigenvariables, so should be of the type $evs \to \tau$. On the other hand, the definite clause itself should be closed, so the constants fst_τ and *rst* should not occur in the definitional clause and only bound variables should be needed by the mapping.

The adequacy of the logic encoding depends on the adequacy of the encoding of lists. The following proposition is provable by induction on L.

Proposition 5.6 *Let $\mathcal{D}$ be a definition that extends $\mathcal{D}(list^*(\tau))$ with clauses that do not define length, element, split, or permute. Then for any $X{:}\,\tau^*$, $L, L'{:}\,lst^*$, multisets L'_1 and L'_2, and natural number n such that $\langle\!|X|\!\rangle_n$, $\langle\!|L|\!\rangle_n$, and $\langle\!|L'|\!\rangle_n$ are well-defined,*

- *$\langle\!|L|\!\rangle_n$ is a multiset with cardinality m if and only if the sequent $\longrightarrow$ length L $(s^m\ z)$ is derivable in $FO\lambda^{\Delta\mathbb{N}}$ with definition $\mathcal{D}$;*
- *$\langle\!|X|\!\rangle_n \in \langle\!|L|\!\rangle_n$ if and only if the sequent $\longrightarrow$ element X L is derivable in $FO\lambda^{\Delta\mathbb{N}}$ with definition $\mathcal{D}$;*
- *$\langle\!|L|\!\rangle_n$ is the multiset union of L'_1 and L'_2 if and only if there exist L_1 and L_2 such that $\langle\!|L_1|\!\rangle_n = L'_1$, $\langle\!|L_2|\!\rangle_n = L'_2$, and the sequent $\longrightarrow$ split L L_1 L_2 is derivable in $FO\lambda^{\Delta\mathbb{N}}$ with definition $\mathcal{D}$;*
- *$\langle\!|L|\!\rangle_n$ and $\langle\!|L'|\!\rangle_n$ are the same multiset if and only if the sequent $\longrightarrow$ permute L L' is derivable in $FO\lambda^{\Delta\mathbb{N}}$ with definition $\mathcal{D}$.*

Theorem 5.7 *Fix a $FO\lambda^{\Delta\mathbb{N}}$ signature whose only constants with types involving* evs *are fst_i, $fst_{i\to i}$, and* rst*. Let $\mathcal{D}(prog)$ be the definition*

$$\{\forall \bar{y}_1[prog\ A_1\ LL_1\ IL_1 \stackrel{\triangle}{=} \top], \ldots, \forall \bar{y}_m[prog\ A_m\ LL_m\ IL_m \stackrel{\triangle}{=} \top]\}$$

$(m \geq 0)$, where the quantified variables in the list $\bar{y}_i$ each have type evs $\to i$ *or* evs $\to i \to i$*, and the constants fst_τ and* rst *do not occur in A_i, LL_i, or IL_i, for all $i \in \{1, \ldots, m\}$. Let $\mathcal{P}$ be the theory in linear logic that corresponds to $\mathcal{D}(prog)$, and let $\mathcal{D}$ be a definition that extends $\mathcal{D}(nat) \cup \mathcal{D}(list^*(\mathrm{atm})) \cup \mathcal{D}(list^*(\mathrm{prp})) \cup \mathcal{D}(linear) \cup \mathcal{D}(prog)$ with clauses that do not define* nat*, length, list, element, split, split_seq, prog, or* seq*. Finally, let $IL{:}\,\mathrm{atmlst}^*$, $LL{:}\,\mathrm{atmlst}^*$, and $B{:}\,\mathrm{prp}^*$ be terms that do not contain occurrences of the constant $fst_{i\to i}$*

and let n be a natural number such that $(\!|IL|\!)_n$,$(\!|LL|\!)_n$, and $(\!|B|\!)_n$ are well-defined. Then the sequent $\longrightarrow IL; LL \rhd B$ is derivable in $FO\lambda^{\Delta\mathbb{N}}$ with definition $\mathcal{D}$ if and only if the sequent $\mathcal{P}, (\!|IL|\!)_n; (\!|LL|\!)_n \longrightarrow (\!|B|\!)_n$ is derivable in linear logic.

Proof: We can restrict our attention to uniform derivations in linear logic, since they are complete for this fragment of linear logic [25]. By Theorem 3.13 we can focus on cut-free $FO\lambda^{\Delta\mathbb{N}}$ derivations, and by Lemma 5.2 these derivations will consist only of sequents with empty antecedents. Thus the definition of *seq* will ensure that the structure of the $FO\lambda^{\Delta\mathbb{N}}$ derivation will closely follow that of the corresponding derivation in linear logic. The proof of the forward direction goes by induction on the structure of the $FO\lambda^{\Delta\mathbb{N}}$ derivation, and the reverse direction by induction on the structure of the linear logic derivation. In general each case follows easily from the induction hypothesis. When the derivation ends with the *initial* rule using an intuitionistic antecedent, Proposition 5.6 is needed. When the derivation ends with the use of a definite clause in the theory, we need to know that there is an i such that the sequents $\longrightarrow nat\ i$ and $\longrightarrow split\_seq_i\ IL\ LL\ (B_1 ::^* \cdots B_m ::^* nil^*)$ are derivable in $FO\lambda^{\Delta\mathbb{N}}$ if and only if there are multisets $LL'_1, \ldots, LL'_m$ such that $(\!|LL|\!)_n$ is the multiset union of $LL'_1, \ldots, LL'_m$ and the sequents $\mathcal{P}, (\!|IL|\!)_n; LL'_1 \longrightarrow (\!|B_1|\!)_n,\ \ldots,\ \mathcal{P}, (\!|IL|\!)_n; LL'_m \longrightarrow (\!|B_m|\!)_n$ are derivable in linear logic. This follows from the induction hypothesis by Proposition 5.6 and an additional induction on m. For the reverse direction of this case and the $\&\mathcal{R}$ case, we also need to know that the formulas

$$\forall i(nat\ i \supset \forall j(nat\ j \supset \exists k(nat\ k\ \wedge\ i < k\ \wedge\ j < k)))$$

$$\forall i \forall j \forall il \forall ll \forall b(nat\ i \supset i < j \supset seq_i\ il\ ll\ b \supset seq_j\ il\ ll\ b)$$

$$\forall i \forall j \forall l \forall il \forall ll(nat\ i \supset i < j \supset split\_seq_i\ il\ ll\ l \supset split\_seq_j\ il\ ll\ l)$$

are derivable in $FO\lambda^{\Delta\mathbb{N}}$; this follows from Propositions 2.17 and 5.11. ∎

We now present the theorems that we have derived in $FO\lambda^{\Delta\mathbb{N}}$ about our object logic. In order to express and prove these theorems, we need additional predicates for operations related to the *evs* parameter. The predicates

$$\begin{aligned} subst &: nt \to i^* \to \tau^* \to \tau^* \to o \\ subst_0 &: nt \to i^{**} \to \tau^{**} \to \tau^{**} \to o\ , \end{aligned}$$

Table 5.10: Encoding of eigenvariable operations

$$
\begin{array}{l}
\hline
\quad subst\ I\ T\ X\ X' \ \triangleq\ subst_0\ I\ (\lambda l'\, T)\ (\lambda l'\, X)\ (\lambda l'\, X') \\
\\
subst_0\ \ z\ \ T\ \ (\lambda l' \lambda l\, X\, l'\, (fst\ l)\, (rst\ l))\ \ (\lambda l' \lambda l\, X\, l'\, (T\, l'\, l)\, (rst\ l)) \\
\qquad\qquad \triangleq\ \top \\
subst_0\ \ (s\ I)\ \ (\lambda l' \lambda l\, T\, l'\, (fst\ l)\, (rst\ l)) \\
\qquad (\lambda l' \lambda l\, X\, l'\, (fst\ l)\, (rst\ l))\ \ (\lambda l' \lambda l\, X'\, l'\, (fst\ l)\, (rst\ l)) \\
\qquad\qquad \triangleq\ subst_0\ \ I\ \ (\lambda l' \lambda l\, T\, (rst\ l')\, (fst\ l')\, l) \\
\qquad\qquad\qquad\qquad (\lambda l' \lambda l\, X\, (rst\ l')\, (fst\ l')\, l)\ \ (\lambda l' \lambda l\, X'\, (rst\ l')\, (fst\ l')\, l) \\
\\
extend\_evars\ I\ X\ X' \ \triangleq\ extend\_evars_0\ I\ (\lambda l'\, X)\ (\lambda l'\, X') \\
\\
extend\_evars_0\ \ z\ \ (\lambda l' \lambda l\, X\, l'\, l)\ \ (\lambda l' \lambda l\, X\, l'\, (rst\ l)) \\
\qquad\qquad \triangleq\ \top \\
extend\_evars_0\ \ (s\ I)\ \ (\lambda l' \lambda l\, X\, l'\, (fst\ l)\, (rst\ l))\ \ (\lambda l' \lambda l\, X'\, l'\, (fst\ l)\, (rst\ l)) \\
\qquad\qquad \triangleq\ extend\_evars_0\ \ I\ \ (\lambda l' \lambda l\, X\, (rst\ l')\, (fst\ l')\, l) \\
\qquad\qquad\qquad\qquad (\lambda l' \lambda l\, X'\, (rst\ l')\, (fst\ l')\, l) \\
\hline
\end{array}
$$

will be use to represent substitution for eigenvariables; this is a simple generalization of the predicate of Section 5.1.4 to allow substitution in expressions of an arbitrary type τ. The type τ^{**} should be understood to mean $(\tau^*)^*$, i.e. an abbrevation for $(evs \to evs \to \tau)$. We will also use the predicates

$$
\begin{array}{rcl}
extend\_evars & : & nt \to \tau^* \to \tau^* \to o \\
extend\_evars_0 & : & nt \to \tau^{**} \to \tau^{**} \to o\ \ ,
\end{array}
$$

to add a new eigenvariable to the list at an offset. Thus $extend\_evars\ i\ x\ x'$ indicates that x' is the result of adding a new eigenvariable in x at the $(i+1)^{\text{th}}$ position in the list; the eigenvariables that previously occupied positions $(i+1)$ or greater are shifted to one position later in the list. These predicates are defined in the definition $\mathcal{D}(evars(\tau))$ of Table 5.10. We will also need an version of $\mathcal{D}(list(\tau))$ to work over the type lst^{**}; it is similar to $\mathcal{D}(list^*(\tau))$ and we will refer it as $\mathcal{D}(list^{**}(\tau))$.

Since we want our theorems about the object logic to be independent of any particular object logic theory, we need to include some assumptions about the predicate *prog*. Specifically, we will need to know that if an atom matches the head of a clause in the theory, then

if we substitute for an eigenvariable in the atom or extend the list of eigenvariables, then the resulting atom will still match the head of the clause. We encode these assumptions as the following two formulas:

$$\forall i \forall t \forall a \forall a' \forall ll \forall il (nat\ i \supset prog\ a\ ll\ il \supset subst\ i\ t\ a\ a' \supset \exists ll' \exists il' (prog\ a'\ ll'\ il' \wedge subst\ i\ t\ ll\ ll' \wedge subst\ i\ t\ il\ il'))\ ,$$

which we will refer to as P_{subst}, and

$$\forall i \forall a \forall a' \forall ll \forall il (nat\ i \supset prog\ a\ ll\ il \supset extend\_evars\ i\ a\ a'\ \supset \exists ll' \exists il' (prog\ a'\ ll'\ il' \wedge extend\_evars\ i\ ll\ ll'\ \wedge extend\_evars\ i\ il\ il'\ ))\ ,$$

which we will refer to as P_{extend}. The theory should not contain occurrences of eigenvariables, and if the definition of *prog* does not contain occurrences of *fst* or *rst*, then P_{subst} and P_{extend} will be derivable in $FO\lambda^{\Delta\mathbb{N}}$.

The following theorems state that we can derive in $FO\lambda^{\Delta\mathbb{N}}$ that the specialization rule, the cut rule, and the usual linear logic structural rules are admissible for our object logic. We refer to the definition

$$\mathcal{D}(list^*(atm)) \cup \mathcal{D}(list^*(prp)) \cup \mathcal{D}(list^{**}(atm)) \cup \mathcal{D}(list^{**}(prp))$$

as $\mathcal{D}(lists)$ and the definition

$$\mathcal{D}(evars(\mathrm{atm})) \cup \mathcal{D}(evars(\mathrm{prp})) \cup \mathcal{D}(evars(\mathrm{atmlst})) \cup \mathcal{D}(evars(\mathrm{prplst}))$$

as $\mathcal{D}(evars)$.

Theorem 5.8 *The formula*

$$P_{subst}\ \supset\ \forall i \forall b \forall il \forall ll (\mathrm{nat}\ i\ \supset\ list\ il\ \supset\ list\ ll\ \supset\ seq_{(S\ i)}\ il\ ll\ \bigwedge{}^*\ b\ \supset\ \forall x\, seq_i\ il\ ll\ (b\, x))$$

is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition $\mathcal{D}(nat) \cup \mathcal{D}(lists) \cup \mathcal{D}(evars) \cup \mathcal{D}(linear)$.

Theorem 5.9 *The formulas*

$$P_{extend}\ \supset\ \forall a \forall b \forall il \forall ll (list\ il\ \supset\ list\ ll\ \supset\ (a ::^* il); ll \rhd b\ \supset\ il; nil^* \rhd \langle a \rangle^*\ \supset\ il; ll \rhd b)$$

$$P_{extend} \supset$$
$$\forall a \forall b \forall il \forall ll \forall ll_1 \forall ll_2 (\mathit{list}\ il \supset \mathit{list}\ ll \supset \mathit{split}\ ll\ ll_1\ ll_2 \supset$$
$$il; (a ::^* ll_1) \rhd b \supset il; ll_2 \rhd \langle a \rangle^* \supset il; ll \rhd b)$$

are derivable in $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(nat) \cup \mathcal{D}(lists) \cup \mathcal{D}(evars) \cup \mathcal{D}(linear)$.

Theorem 5.10 *The formula*

$$\forall i \forall b \forall il \forall il' \forall ll \forall ll' (\mathit{nat}\ i \supset \mathit{list}\ il \supset \mathit{list}\ il' \supset \mathit{list}\ ll \supset$$
$$\forall a (\mathit{element}\ a\ il \supset \mathit{element}\ a\ il') \supset \mathit{permute}\ ll\ ll' \supset$$
$$\mathit{seq}_i\ il\ ll\ b \supset \mathit{seq}_i\ il'\ ll'\ b)$$

is derivable in $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(nat) \cup \mathcal{D}(lists) \cup \mathcal{D}(evars) \cup \mathcal{D}(linear)$.

The last proposition of this section states two additional properties of our object logic that are derivable in $FO\lambda^{\Delta\mathbb{N}}$.

Proposition 5.11 *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(nat) \cup \mathcal{D}(lists) \cup \mathcal{D}(evars) \cup \mathcal{D}(linear)$*:*

$$\forall i \forall j \forall il \forall ll \forall b (\mathit{nat}\ i \supset i < j \supset \mathit{seq}_i\ il\ ll\ b \supset \mathit{seq}_j\ il\ ll\ b)$$

$$\forall i \forall j \forall l \forall il \forall ll (\mathit{nat}\ i \supset i < j \supset \mathit{split\_seq}_i\ il\ ll\ l \supset \mathit{split\_seq}_j\ il\ ll\ l)\ .$$

5.3 Related Work

In this chapter we have presented several different encodings of logics; for each we discussed the extent to which reasoning about the encoded logic can take place within the meta-logic $FO\lambda^{\Delta\mathbb{N}}$. None of the encoding techniques is completely original, but their ability to support formal meta-theoretic analysis is a relatively new concern.

The natural deduction-style encoding of Section 5.1.1 is the prototypical representation style of higher-order abstract syntax. For example, the seminal paper on the Edinburgh Logical Framework (LF) [24] encodes first-order and higher-order logic in this manner and proves the adequacy of these encodings. The issue of meta-theoretic analysis of the encodings within the meta-logic is not addressed.

The use of separate predicates for formulas on the left and right sides of the sequent, as was done in Section 5.1.2, is also common. Pfenning [45], for example, uses this representation style to encode structural cut-elimination proofs for intuitionistic, classical, and linear logics. The induction cases of these proofs are represented in Elf, so some amount of reasoning about the encoded logics is done in the meta-logic. However Elf does not itself contain any support for induction, so the completeness of the cases must be checked outside of the formal framework using techniques such as schema checking [47, 48]. Miller [37] uses both this sequent style of encoding and the natural deduction style. The two encodings are used to show that natural deduction and sequent calculus presentations of minimal logic have the same theorems. The proof of this result combines informal reasoning with formal reasoning in the meta-logic Forum.

Section 5.1.3 presented an encoding of logic which encoded the derivability of a sequent in a single predicate. This style of encoding was used in an early paper on the use of higher-order abstract syntax [38]. The paper focuses on an operational interpretation of such a specification, however, and does not discuss the potential for reasoning about the encoded logic in the meta-logic.

The idea of representing free variables as a list, discussed in Section 5.1.4, was first used in the context of higher-order abstract syntax by Despeyroux and Hirschowitz [9]. Their intent was to develop a way to use higher-order abstract syntax within the setting of the inductive definition facility of Coq. A key difference between their technique and ours is that they use both constructor and deconstructor operators for lists in the context of an equality theory. The encoding of the right rule for universal quantification in that setting might look like the following:

$$seq_{(s\ I)}\ L\ (\lambda l \bigwedge_i (\lambda x\, B\, (cons\, x\, l))) \quad \stackrel{\triangle}{=} \quad seq_I\ (\lambda l'\ L(rst\ l'))\ B\ .$$

Within terms, bound and free variables are accessed by selecting the appropriate element from the list. In our simpler setting (without an equality theory) we use unification to get by with only deconstructors. The paper [9] was the first attempt to fully support formal reasoning about higher-order abstract syntax encodings within a meta-logic. Their examples involved encodings of simply-typed λ-terms, so we will further discuss their work at the end of the next chapter.

Chapter 6

Reasoning about Programming Languages in $FO\lambda^{\Delta\mathbb{N}}$

In this chapter we consider reasoning about higher-order abstract syntax encodings of programming languages. We could choose one of the representation strategies used for logics in the previous chapter; instead we adopt a different strategy that allows us to use the traditional higher-order abstract syntax representation to its full advantage and still reason formally about the encoded system. The key to accomplishing this is to not specify the programming language directly in $FO\lambda^{\Delta\mathbb{N}}$, but in a small object logic that is itself specified in $FO\lambda^{\Delta\mathbb{N}}$. In this way we can reason in $FO\lambda^{\Delta\mathbb{N}}$ about the structure of object logic sequents and their derivability.

The use of object-level sequents may seem at first a rather drastic step to take to embed the kind of hypothetical judgements common with higher-order abstract syntax into a meta-logic. Such a representation is, however, used in various areas of programming language semantics. For example, Mitchell, in his textbook [41], uses typing judgements of the form $\Gamma \rhd M : \sigma$ and performs induction over their (sequent-style) derivation. This separation of the (object) specification logic from the meta-logic ($FO\lambda^{\Delta\mathbb{N}}$) in which reasoning is performed also reflects the usual structure of informal reasoning about higher-order abstract syntax encodings.

In the next section we motivate this approach through an informal proof of subject

reduction for the untyped λ-calculus. We proceed in Section 6.2 to formalize this proof by encoding the static and dynamic semantics for untyped λ-terms in the intuitionistic object logic of Section 5.2.1. We also list a variety of other theorems about the language that we have derived in $FO\lambda^{\Delta\mathbb{N}}$. Section 6.3 extends the encoding of Section 6.2 to the Programming language of Computable Functions (PCF) [52]. We examine a derivation of the unicity of typing for PCF to show that the encoding of quantification in the object logic does not support the kinds of analysis used in the typical informal proof of this theorem. In Section 6.4 we consider an encoding of PCF with references ($\text{PCF}_{:=}$) [18] in the linear object logic of Section 5.2.2. Since the encoding of this object logic uses the explicit eigenvariable style of Section 5.1.4, we are able to encode the standard proof of the unicity of typing in $FO\lambda^{\Delta\mathbb{N}}$. Finally, Section 6.5 compares the framework of this chapter with other research in formal reasoning about higher-order abstract syntax encodings. Most of the research discussed in this chapter was presented in [31], coauthored with Miller; the notable exception is Section 6.4, which is new.

6.1 Motivation from Informal Reasoning

In order to motivate our framework for reasoning about higher-order abstract syntax encodings, we consider a specification in intuitionistic logic of call-by-name evaluation and simple typing for the untyped λ-calculus. We introduce two types, tm and ty, to denote object-level terms and types. To represent the untyped λ-terms we introduce the two constants abs of type $(tm \to tm) \to tm$ and app of type $tm \to tm \to tm$ to denote object-level abstraction and application, respectively. Object-level types will be built up from a single primitive type using the arrow type constructor; these are denoted in the specification logic by the constants gnd of type ty and arr of type $ty \to ty \to ty$.

To specify call-by-name evaluation, we use an infix predicate $\Downarrow$ of type $tm \to tm \to o$ and the two formulas

$$\bigwedge r((abs\ r) \Downarrow (abs\ r))$$
$$\bigwedge m \bigwedge n \bigwedge v \bigwedge r((m \Downarrow (abs\ r)\ \ \&\ \ (r\ n) \Downarrow v)\ \Rightarrow\ (app\ m\ n) \Downarrow v)\ .$$

To specify simple typing at the object-level, we use the binary predicate $typeof$ of type

$tm \to ty \to o$ and the two formulas

$$\bigwedge m \bigwedge n \bigwedge t \bigwedge u((typeof\ m\ (arr\ u\ t)\ \&\ typeof\ n\ u)\ \Rightarrow\ typeof\ (app\ m\ n)\ t)$$
$$\bigwedge r \bigwedge t \bigwedge u(\bigwedge x(typeof\ x\ t\ \Rightarrow\ typeof\ (r\ x)\ u)\ \Rightarrow\ typeof\ (abs\ r)\ (arr\ t\ u))\ .$$

Proofs that these two predicates correctly capture the notions of call-by-name evaluation and of simple typing can be found in various places in the literature: see, for example, [3, 23].

Now consider the following subject reduction theorem and its proof. We use $\vdash$ here to represent derivability in intuitionistic logic from the above formulas encoding evaluation and typing; we omit displaying these formulas on the left of the turnstile to simplify the presentation.

Proposition 6.1 *If* $\vdash P \Downarrow V$ *and* $\vdash typeof\ P\ T$, *then* $\vdash typeof\ V\ T$.

Proof We prove this theorem by induction on the height of the derivation of $P \Downarrow V$. Since $P \Downarrow V$ is atomic, its derivation must end with the use of one of the formulas encoding evaluation. If the $\Downarrow$ formula for *abs* is used, then P and V are both equal to $abs\ R$, for some R, and the consequent is immediate. If $P \Downarrow V$ was derived using the $\Downarrow$ formula for *app*, then P is of the form $(app\ M\ N)$, and for some R there are shorter derivations of $M \Downarrow (abs\ R)$ and $(R\,N) \Downarrow V$. Since $\vdash typeof\ (app\ M\ N)\ T$, this typing judgement must have been derived using one of the formulas encoding the typing rules and, hence, there is a U such that $\vdash typeof\ M\ (arr\ U\ T)$ and $\vdash typeof\ N\ U$. Using the inductive hypothesis, we have $\vdash typeof\ (abs\ R)\ (arr\ U\ T)$. This atomic formula must have been derived using the *typeof* formula for *abs*, and, hence, $\vdash \bigwedge x(typeof\ x\ U \Rightarrow typeof\ (R\,x)\ T)$. Since our specification logic is intuitionistic logic, we can instantiate this quantifier with N and use cut and cut-elimination to conclude that $\vdash typeof\ (R\,N)\ T$. Using the inductive hypothesis a second time yields $\vdash typeof\ V\ T$. ∎

This proof is clear and natural, and we would like to be able to formally capture proofs quite similar to this in structure. This suggests that the following features would be valuable in our framework:

1. *Two distinct logics.* One of the logics would correspond to the one written with logical syntax above and would capture judgements, e.g. about typability and evaluation.

The second logic would represent a formalization of the English text in the proof above. Atomic formulas of that logic would encode judgements in the object logic.

2. *Induction* over at least natural numbers.

3. *Instantiation of meta-level eigenvariables.* In the proof above, for example, the meta-level variable P was instantiated in one part of the proof to $(abs\ R)$ and in another part of the proof to $(app\ M\ N)$. Notice that this instantiation of eigenvariables within a proof does not happen in a strictly intuitionistic sequent calculus.

4. *Analysis of the derivation of an assumed judgement.* In the proof above this was done a few times, leading, for example, from the assumption

$$\vdash typeof\ (abs\ R)\ (arr\ U\ T)$$

to the assumption

$$\vdash \bigwedge x(typeof\ x\ U \Rightarrow typeof\ (R\,x)\ T)\ .$$

The specification of *typeof* allows the implication to go in the other direction, but given the structure of the specification of *typeof*, this direction can also be justified at the meta-level.

In our framework, we accommodate the first feature by specifying an object logic within the meta-logic $FO\lambda^{\Delta\mathbb{N}}$, as illustrated in Chapter 5. The $nat\mathcal{L}$ rule of $FO\lambda^{\Delta\mathbb{N}}$ provides natural number induction. The last two features are accommodated by the definition facilities of $FO\lambda^{\Delta\mathbb{N}}$.

6.2 The Language of Untyped λ-Terms

We first demonstrate our approach to formal reasoning about higher-order abstract syntax encodings using the example of untyped λ-terms. This encoding will be similar to the one used to motivate the framework in the preceding section. The object logic used will be the fragment of second-order intuitionistic logic encoded by the definition $\mathcal{D}(intuit)$ of Section 5.2.1.

Table 6.1: Object logic encoding of typing and evaluation of untyped λ-terms

prog	$(typeof\ (abs\ R)\ (arr\ T\ U))$	$\bigwedge n((typeof\ n\ T) \Rightarrow \langle typeof\ (R\,n)\ U\rangle)$
prog	$(typeof\ (app\ M\ N)\ T)$	$\langle typeof\ M\ (arr\ U\ T)\rangle\ \&\ \langle typeof\ N\ U\rangle$
prog	$((abs\ R) \Downarrow (abs\ R))$	tt
prog	$((app\ M\ N) \Downarrow V)$	$\langle M \Downarrow (abs\ R)\rangle\ \&\ \langle (R\,N) \Downarrow V\rangle$
prog	$((app\ (abs\ R)\ M) \rightsquigarrow (R\,M))$	tt
prog	$((app\ M\ N) \rightsquigarrow (app\ M'\ N))$	$\langle M \rightsquigarrow M'\rangle$
prog	$(M \rightsquigarrow^* M)$	tt
prog	$(M \rightsquigarrow^* N)$	$\langle M \rightsquigarrow M'\rangle\ \&\ \langle M' \rightsquigarrow^* N\rangle$

The required constants to represent λ-terms are $abs : (i_{tm} \to i_{tm}) \to i_{tm}$ and $app : i_{tm} \to i_{tm} \to i_{tm}$; for simple types (over one primitive type) we need $gnd\colon i_{ty}$ and $arr\colon i_{ty} \to i_{ty} \to i_{ty}$. Since both types and terms in the language are represented by the object logic type i, we have added subscripts tm and ty. These subscripts should not be considered part of the encoding, but are added to improve the readability of these declarations.

Our object logic predicate representing typability is denoted by the $FO\lambda^{\Delta\mathbb{N}}$ constant *typeof* of type $i_{tm} \to i_{ty} \to atm$. The predicates for natural semantics and transition semantics are denoted by the constants $\Downarrow$, $\rightsquigarrow$, and $\rightsquigarrow^*$, all of type $i_{tm} \to i_{tm} \to atm$. The object logic specifications for these are the usual ones, written in the L_λ subset of higher-order logic [35] and are those common to specifications written in, say, λProlog [22] and Elf [44]. This object-level specification is represented in $FO\lambda^{\Delta\mathbb{N}}$ as the definition $\mathcal{D}(lambda)$ shown in Table 6.1. (We have dropped the $\stackrel{\triangle}{=} \top$ body of these clauses.) This definition can be interpreted in a logic programming fashion to compute object-level simple type checking and call-by-name evaluation in both structural operational semantic and natural semantic styles. Call-by-value is just as easily represented and used.

The following theorems list the properties of the untyped λ-calculus that we have derived in $FO\lambda^{\Delta\mathbb{N}}$: determinacy of semantics, equivalence of semantics, and subject reduction. The $FO\lambda^{\Delta\mathbb{N}}$ derivations closely follow the informal proofs of these properties.

Theorem 6.2 *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *from the definition that*

accumulates $\mathcal{D}(nat)$, $\mathcal{D}(list(\mathrm{atm}))$, $\mathcal{D}(intuit)$, $\mathcal{D}(lambda)$ *and the clause* $X \equiv X \stackrel{\triangle}{=} \top$ *defining the predicate* $\equiv\colon i \to i \to o$*:*

$$\forall m \forall m_1 \forall m_2 (\triangleright\langle m \Downarrow m_1\rangle \supset \triangleright\langle m \Downarrow m_2\rangle \supset m_1 \equiv m_2)$$
$$\forall m \forall m_1 \forall m_2 (\triangleright\langle m \rightsquigarrow m_1\rangle \supset \triangleright\langle m \rightsquigarrow m_2\rangle \supset m_1 \equiv m_2)$$
$$\forall m \forall r_1 \forall r_2 (\triangleright\langle m \rightsquigarrow^* (\mathrm{abs}\ r_1)\rangle \supset \triangleright\langle m \rightsquigarrow^* (\mathrm{abs}\ r_2)\rangle \supset (\mathrm{abs}\ r_1) \equiv (\mathrm{abs}\ r_2)) \ .$$

Theorem 6.3 *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *from the definition that accumulates* $\mathcal{D}(nat)$, $\mathcal{D}(list(\mathrm{atm}))$, $\mathcal{D}(intuit)$, $\mathcal{D}(lambda)$ *and the clause* $X \equiv X \stackrel{\triangle}{=} \top$ *defining the predicate* $\equiv\colon i \to i \to o$*:*

$$\forall m \forall r (\triangleright\langle m \Downarrow (\mathrm{abs}\ r)\rangle \supset \triangleright\langle m \rightsquigarrow^* (\mathrm{abs}\ r)\rangle)$$
$$\forall m \forall r (\triangleright\langle m \rightsquigarrow^* (\mathrm{abs}\ r)\rangle \supset \triangleright\langle m \Downarrow (\mathrm{abs}\ r)\rangle) \ .$$

Theorem 6.4 *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *from the definition that accumulates* $\mathcal{D}(nat)$, $\mathcal{D}(list(\mathrm{atm}))$, $\mathcal{D}(intuit)$, $\mathcal{D}(lambda)$ *and the clause* $X \equiv X \stackrel{\triangle}{=} \top$ *defining the predicate* $\equiv\colon i \to i \to o$*:*

$$\forall m \forall n (\triangleright\langle m \Downarrow n\rangle \supset \forall t (\triangleright\langle typeof\ m\ t\rangle \supset \triangleright\langle typeof\ n\ t\rangle))$$
$$\forall m \forall n (\triangleright\langle m \rightsquigarrow n\rangle \supset \forall t (\triangleright\langle typeof\ m\ t\rangle \supset \triangleright\langle typeof\ n\ t\rangle))$$
$$\forall m \forall n (\triangleright\langle m \rightsquigarrow^* n\rangle \supset \forall t (\triangleright\langle typeof\ m\ t\rangle \supset \triangleright\langle typeof\ n\ t\rangle)) \ .$$

Proof We show the derivation of the first subject reduction property, which is a formalization of Proposition 6.1.

We wish to show that evaluation preserves types:

$$\longrightarrow \forall p \forall v (\triangleright\langle p \Downarrow v\rangle \supset \forall t (\triangleright\langle typeof\ p\ t\rangle \supset \triangleright\langle typeof\ v\ t\rangle)) \ .$$

(We have changed the names of the quantified variables to agree with those in the informal proof.) Applying the $\forall\mathcal{R}$, $\supset\mathcal{R}$, $\exists\mathcal{L}$, $c\mathcal{L}$, and $\wedge\mathcal{L}$ rules to the above sequent yields

$$nat\ i, seq_i\ nil\ \langle p \Downarrow v\rangle, \triangleright\langle typeof\ p\ t\rangle \longrightarrow \triangleright\langle typeof\ v\ t\rangle \ .$$

(Recall that $\triangleright\langle p \Downarrow v\rangle$ is an abbreviation for $\exists i(nat\ i \wedge seq_i\ nil\ \langle p \Downarrow v\rangle)$.)

As in the informal proof, we proceed with an induction on the height of the derivation of $p \Downarrow v$, which is represented here by i. We will use the derived rule for complete induction (Proposition 2.16) and our induction predicate will be

$$\lambda i \forall p \forall v \forall t (seq_i\ nil\ \langle p \Downarrow v\rangle \supset \triangleright\langle typeof\ p\ t\rangle \supset \triangleright\langle typeof\ v\ t\rangle) \ ,$$

which we will denote by *IP*. The derivation of the conclusion from the induction predicate applied to i is trivial, so it only remains to derive the induction step

$$nat\ j, \forall k(nat\ k \supset k < j \supset (IP\ k)) \longrightarrow (IP\ j)\ .$$

In the informal proof we use the fact that the derivation of the atomic formula $p \Downarrow v$ must end with the use of a clause from the specification of evaluation. We deduce this formally using the $def\mathcal{L}$ rule as follows:

$$\dfrac{\dfrac{\dfrac{nat\ (s\ j_0), \forall k \ldots, \exists b(prog\ (p \Downarrow v)\ b \wedge seq_{j_0}\ nil\ b), \rhd\langle typeof\ p\ t\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle}{nat\ j, \forall k \ldots, seq_j\ nil\ \langle p \Downarrow v\rangle, \rhd\langle typeof\ p\ t\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle}\ def\mathcal{L}}{nat\ j, \forall k \ldots \longrightarrow seq_j\ nil\ \langle p \Downarrow v\rangle \supset \rhd\langle typeof\ p\ t\rangle \supset \rhd\langle typeof\ v\ t\rangle}\ \supset\mathcal{R}}{nat\ j, \forall k(nat\ k \supset k < j \supset (IP\ k)) \longrightarrow (IP\ j)}\ \forall\mathcal{R}\ .$$

We next apply the $\exists\mathcal{L}$, $c\mathcal{L}$, and $\wedge\mathcal{L}$ rules, and then apply the $def\mathcal{L}$ rule to $prog\ (p \Downarrow v)\ b$ which yields the two sequents

$$nat\ (s\ j_0), \forall k \ldots, seq_{j_0}\ nil\ tt, \rhd\langle typeof\ (abs\ r)\ t\rangle \longrightarrow \rhd\langle typeof\ (abs\ r)\ t\rangle$$

$$nat\ (s\ j_0), \forall k \ldots, seq_{j_0}\ nil\ \langle m \Downarrow (abs\ r)\rangle\ \&\ \langle (r\ n) \Downarrow v\rangle, \rhd\langle typeof\ (app\ m\ n)\ t\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .$$

This use of the $def\mathcal{L}$ rule corresponds to the case analysis of the formula used to derive $p \Downarrow v$. As in the informal case, the *abs* case (represented here by the first sequent) is immediate. The derivation of the second sequent, representing the *app* case, begins with the use of the $def\mathcal{L}$, $c\mathcal{L}$, and $\wedge\mathcal{L}$, bringing us to the sequent

$$nat\ (s^2\ j_1), \forall k \ldots, seq_{j_1}\ nil\ \langle m \Downarrow (abs\ r)\rangle, seq_{j_1}\ nil\ \langle (r\ n) \Downarrow v\rangle, \rhd\langle typeof\ (app\ m\ n)\ t\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .$$

(We use the term $s^2\ j_1$ as an abbreviation for $s\ (s\ j_1)$.)

The informal proof continues with an analysis of the derivation of $typeof\ (app\ m\ n)\ t$. Again we accomplish this through two uses of the $def\mathcal{L}$ rule, the first to indicate that the derivation must end with the use of a specification clause, and the second to determine the applicable clauses. In this case there is only one applicable clause, so we are left to derive the sequent

$$\ldots, nat\ (s\ j_0'), seq_{j_0'}\ nil\ \langle typeof\ m\ (arr\ u\ t)\rangle\ \&\ \langle typeof\ n\ u\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .$$

Additional uses of the $def\mathcal{L}$, $c\mathcal{L}$ and $\wedge\mathcal{L}$ rules bring us to the sequent

$$\ldots, nat\ (s^2\ j_1'), seq_{j_1'}\ nil\ \langle typeof\ m\ (arr\ u\ t)\rangle, seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle \longrightarrow \triangleright\langle typeof\ v\ t\rangle\ .$$

In the informal proof we now apply the induction hypothesis to the evaluation and typing judgments for m. We accomplish this here by applying the appropriate left rules to the elided induction hypothesis $\forall k \ldots$. This requires the derivation of the five sequents

$$nat\ (s^2\ j_1), \ldots \longrightarrow nat\ j_1 \qquad\qquad nat\ (s^2\ j_1), \ldots \longrightarrow j_1 < (s^2\ j_1)$$

$$\ldots, seq_{j_1}\ nil\ \langle m \Downarrow (abs\ r)\rangle, \ldots \longrightarrow seq_{j_1}\ nil\ \langle m \Downarrow (abs\ r)\rangle$$

$$\ldots, nat\ (s^2\ j_1'), seq_{j_1'}\ nil\ \langle typeof\ m\ (arr\ u\ t)\rangle, \ldots \longrightarrow \triangleright\langle typeof\ m\ (arr\ u\ t)\rangle$$

$$\begin{array}{c} nat\ (s^2\ j_1), \forall k \ldots, seq_{j_1}\ nil\ \langle (r\ n) \Downarrow v\rangle, \\ \triangleright\langle typeof\ (abs\ r)\ (arr\ u\ t)\rangle, \\ nat\ (s^2\ j_1'), seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle \longrightarrow \triangleright\langle typeof\ v\ t\rangle\ . \end{array}$$

The first two of these represent the fact that the measure of the evaluation derivation for m is a natural number that is smaller than the measure of the original evaluation derivation for p. By Proposition 2.17 these are derivable in $FO\lambda^{\Delta\mathbb{N}}$ from $\mathcal{D}(nat)$. The third sequent is immediate, and the fourth also follows from Proposition 2.17:

$$\dfrac{\dfrac{\ldots, nat\ (s^2\ j_1'), \ldots \longrightarrow nat\ j_1' \quad \dfrac{}{\ldots \longrightarrow seq_{j_1'}\ \langle typeof\ m\ (arr\ u\ t)\rangle}\ init}{\ldots \longrightarrow nat\ j_1' \wedge seq_{j_1'}\ \langle typeof\ m\ (arr\ u\ t)\rangle}\ \wedge\mathcal{R}}{\ldots, nat\ (s^2\ j_1'), seq_{j_1'}\ nil\ \langle typeof\ m\ (arr\ u\ t)\rangle, \ldots \longrightarrow \triangleright\langle typeof\ m\ (arr\ u\ t)\rangle}\ \exists\mathcal{R}\ .$$

The derivation of the fifth sequent proceeds with another two applications of the $def\mathcal{L}$ rule, corresponding to the analysis of the proof of $typeof\ (abs\ r)\ (arr\ u\ t)$ in the informal proof. This yields the sequent

$$\ldots, nat\ (s\ j_0''), seq_{j_0''}\ nil\ \bigwedge x((typeof\ x\ u) \Rightarrow \langle typeof\ (r\ x)\ t\rangle), \ldots \longrightarrow \triangleright\langle typeof\ v\ t\rangle\ .$$

This is followed by applications of the $def\mathcal{L}$ and $\forall\mathcal{L}$ rules to give us

$$\ldots, nat\ (s^3\ j_1''), seq_{j_1''}\ ((typeof\ n\ u)::nil)\ \langle typeof\ (r\ n)\ t\rangle, \ldots \longrightarrow \triangleright\langle typeof\ v\ t\rangle\ .$$

The informal proof proceeds with a use of the cut rule, and here we use the derived object-level cut rule (Theorem 5.4) with the elided assumption $seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle$ to obtain

$$\begin{gathered}\ldots, nat\ (s^3\ j_1''), seq_{j_1''}\ ((typeof\ n\ u)::nil)\ \langle typeof\ (r\,n)\ t\rangle,\\ \ldots \longrightarrow ((typeof\ n\ u)::nil) \rhd \langle typeof\ (r\,n)\ t\rangle\\ \ldots, nat\ (s^2\ j_1'), seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle \longrightarrow \rhd\langle typeof\ n\ u\rangle\\ \ldots, \rhd\langle typeof\ (r\,n)\ t\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .\end{gathered}$$

The first two of these follow easily from Proposition 2.17.

The informal proof concludes by applying the induction hypothesis to the evaluation and typing judgments for $(r\,n)$. Again we accomplish this by applying the appropriate left rules to the induction hypothesis $\forall k \ldots$, which requires the derivation of the five sequents

$$\begin{gathered}nat\ (s^2\ j_1) \longrightarrow nat\ j_1 \qquad\qquad nat\ (s^2\ j_1) \longrightarrow j_1 < (s^2\ j_1)\\ \ldots, seq_{j_1}\ nil\ \langle (r\,n) \Downarrow v\rangle, \ldots \longrightarrow seq_{j_1}\ nil\ \langle (r\,n) \Downarrow v\rangle\\ \ldots, \rhd\langle typeof\ (r\,n)\ t\rangle \longrightarrow \rhd\langle typeof\ (r\,n)\ t\rangle\\ \ldots, \rhd\langle typeof\ v\ t\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .\end{gathered}$$

The first two sequents follow from Proposition 2.17, and the last three are all immediate. ∎

6.3 A Language for Computable Functions

We now extend the encoding of the static and dynamic semantics for untyped λ-terms from the previous section to the programming language PCF [52]. The necessary $FO\lambda^{\Delta\mathbb{N}}$ constants for PCF types are

$$num : i_{ty} \qquad bool : i_{ty} \qquad arr : i_{ty} \to i_{ty} \to i_{ty}\ .$$

Those for PCF terms are

$$\begin{array}{llll} zero : i_{tm} & succ : i_{tm} \to i_{tm} & if : i_{tm} \to i_{tm} \to i_{tm} \to i_{tm}\\ true : i_{tm} & pred : i_{tm} \to i_{tm} & abs : i_{ty} \to (i_{tm} \to i_{tm}) \to i_{tm}\\ false : i_{tm} & is\_zero : i_{tm} \to i_{tm} & app : i_{tm} \to i_{tm} \to i_{tm}\\ & & rec : i_{ty} \to (i_{tm} \to i_{tm}) \to i_{tm}\ .\end{array}$$

Table 6.2: Object logic encoding of typing for PCF

prog	$(typeof\ zero\ num)$	tt
prog	$(typeof\ true\ bool)$	tt
prog	$(typeof\ false\ bool)$	tt
prog	$(typeof\ (succ\ M)\ num)$	$\langle typeof\ M\ num\rangle$
prog	$(typeof\ (pred\ M)\ num)$	$\langle typeof\ M\ num\rangle$
prog	$(typeof\ (is\_zero\ M)\ bool)$	$\langle typeof\ M\ num\rangle$
prog	$(typeof\ (if\ M\ N_1\ N_2)\ T)$	$\langle typeof\ M\ bool\rangle\ \&\ \langle typeof\ N_1\ T\rangle\ \&\ \langle typeof\ N_2\ T\rangle$
prog	$(typeof\ (abs\ T\ R)\ (arr\ T\ U))$	$\bigwedge n((typeof\ n\ T) \Rightarrow \langle typeof\ (R\,n)\ U\rangle)$
prog	$(typeof\ (app\ M\ N)\ T)$	$\langle typeof\ M\ (arr\ U\ T)\rangle\ \&\ \langle typeof\ N\ U\rangle$
prog	$(typeof\ (rec\ T\ R)\ T)$	$\bigwedge n((typeof\ n\ T) \Rightarrow \langle typeof\ (R\,n)\ T\rangle)$

We have again labeled the type i with subscripts to improve the readability of these declarations. The first argument to *abs* and *rec* represent the PCF type tag for the variable bound by the abstraction and recursion constructs.

The object logic predicates representing typability and evaluation are denoted by the same $FO\lambda^{\Delta\mathbb{N}}$ constants as in Section 6.2, plus the additional constant $value : i_{tm} \to atm$. The object-level specification is represented in $FO\lambda^{\Delta\mathbb{N}}$ as the definition $\mathcal{D}$(PCF) shown in Tables 6.2, 6.3, and 6.4; we have again omitted the $\stackrel{\triangle}{=} \top$ body of the clauses. The following theorems list the properties of PCF that we have derived in $FO\lambda^{\Delta\mathbb{N}}$. The type tags in PCF terms allow the unicity of typing to hold in addition to the determinacy of semantics, equivalence of semantics and subject reduction. The $FO\lambda^{\Delta\mathbb{N}}$ derivations again closely follow the informal proofs of these properties; the only exception is the derivation of the unicity of typing property, which we sketch below.

Theorem 6.5 *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *from the definition that accumulates* $\mathcal{D}(nat)$, $\mathcal{D}(list(\mathrm{atm}))$, $\mathcal{D}(intuit)$, $\mathcal{D}$(PCF) *and the clause* $X \equiv X \stackrel{\triangle}{=} \top$ *defining the predicate* $\equiv\colon i \to i \to o$*:*

$$\forall m \forall m_1 \forall m_2 (\triangleright\langle m \Downarrow m_1\rangle \supset \triangleright\langle m \Downarrow m_2\rangle \supset m_1 \equiv m_2)$$

$$\forall m \forall m_1 \forall m_2 (\triangleright\langle m \rightsquigarrow m_1\rangle \supset \triangleright\langle m \rightsquigarrow m_2\rangle \supset m_1 \equiv m_2)$$

$$\forall m \forall v_1 \forall v_2 (\triangleright\langle value\ v_1\rangle \supset \triangleright\langle m \rightsquigarrow^* v_1\rangle \supset \triangleright\langle value\ v_2\rangle \supset \triangleright\langle m \rightsquigarrow^* v_2\rangle \supset v_1 \equiv v_2)\ .$$

Table 6.3: Object logic encoding of natural semantics for PCF

prog	$(zero \Downarrow zero)$	tt
prog	$(true \Downarrow true)$	tt
prog	$(false \Downarrow false)$	tt
prog	$((succ\ M) \Downarrow (succ\ V))$	$\langle M \Downarrow V \rangle$
prog	$((pred\ M) \Downarrow zero)$	$\langle M \Downarrow zero \rangle$
prog	$((pred\ M) \Downarrow V)$	$\langle M \Downarrow (succ\ V) \rangle$
prog	$((is\_zero\ M) \Downarrow true)$	$\langle M \Downarrow zero \rangle$
prog	$((is\_zero\ M) \Downarrow false)$	$\langle M \Downarrow (succ\ V) \rangle$
prog	$((if\ M\ N_1\ N_2) \Downarrow V)$	$\langle M \Downarrow true \rangle\ \&\ \langle N_1 \Downarrow V \rangle$
prog	$((if\ M\ N_1\ N_2) \Downarrow V)$	$\langle M \Downarrow false \rangle\ \&\ \langle N_2 \Downarrow V \rangle$
prog	$((abs\ T\ R) \Downarrow (abs\ T\ R))$	tt
prog	$((app\ M\ N) \Downarrow V)$	$\langle M \Downarrow (abs\ T\ R) \rangle\ \&\ \langle (R\,N) \Downarrow V \rangle$
prog	$((rec\ T\ R) \Downarrow V)$	$\langle (R\,(rec\ T\ R)) \Downarrow V \rangle$

Table 6.4: Object logic encoding of transition semantics for PCF

prog	$((succ\ M) \rightsquigarrow (succ\ M'))$	$\langle M \rightsquigarrow M' \rangle$
prog	$((pred\ zero) \rightsquigarrow zero)$	tt
prog	$((pred\ (succ\ V)) \rightsquigarrow V)$	$\langle value\ V \rangle$
prog	$((pred\ M) \rightsquigarrow (pred\ M'))$	$\langle M \rightsquigarrow M' \rangle$
prog	$((is\_zero\ zero) \rightsquigarrow true)$	tt
prog	$((is\_zero\ (succ\ V)) \rightsquigarrow false)$	$\langle value\ V \rangle$
prog	$((is\_zero\ M) \rightsquigarrow (is\_zero\ M'))$	$\langle M \rightsquigarrow M' \rangle$
prog	$((if\ true\ M\ N) \rightsquigarrow M)$	tt
prog	$((if\ false\ M\ N) \rightsquigarrow N)$	tt
prog	$((if\ M\ N_1\ N_2) \rightsquigarrow (if\ M'\ N_1\ N_2))$	$\langle M \rightsquigarrow M' \rangle$
prog	$((app\ (abs\ TR)\ N) \rightsquigarrow (R\,N))$	tt
prog	$((app\ M\ N) \rightsquigarrow (app\ M'\ N))$	$\langle M \rightsquigarrow M' \rangle$
prog	$((rec\ T\ R) \rightsquigarrow (R\,(rec\ T\ R)))$	tt
prog	$(M \rightsquigarrow^* M)$	tt
prog	$(M \rightsquigarrow^* N)$	$(\langle M \rightsquigarrow M' \rangle\ \&\ \langle M' \rightsquigarrow^* N \rangle)$
prog	$(value\ zero)$	tt
prog	$(value\ true)$	tt
prog	$(value\ false)$	tt
prog	$(value\ (succ\ V))$	$\langle value\ V \rangle$
prog	$(value\ (abs\ TR))$	tt

Theorem 6.6 *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *from the definition that accumulates* $\mathcal{D}(nat)$, $\mathcal{D}(list(\mathrm{atm}))$, $\mathcal{D}(intuit)$, $\mathcal{D}(\mathrm{PCF})$ *and the clause* $X \equiv X \stackrel{\triangle}{=} \top$ *defining the predicate* $\equiv\colon i \to i \to o$*:*

$$\forall m \forall v(\triangleright\langle m \Downarrow v\rangle \supset (\triangleright\langle \mathit{value}\ v\rangle \wedge \triangleright\langle m \rightsquigarrow^* v\rangle))$$
$$\forall m \forall v((\triangleright\langle \mathit{value}\ v\rangle \wedge \triangleright\langle m \rightsquigarrow^* v\rangle) \supset \triangleright\langle m \Downarrow v\rangle)\ .$$

Theorem 6.7 *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *from the definition that accumulates* $\mathcal{D}(nat)$, $\mathcal{D}(list(\mathrm{atm}))$, $\mathcal{D}(intuit)$, $\mathcal{D}(\mathrm{PCF})$ *and the clause* $X \equiv X \stackrel{\triangle}{=} \top$ *defining the predicate* $\equiv\colon i \to i \to o$*:*

$$\forall m \forall n(\triangleright\langle m \Downarrow n\rangle \supset \forall t(\triangleright\langle \mathit{typeof}\ m\ t\rangle \supset \triangleright\langle \mathit{typeof}\ n\ t\rangle))$$
$$\forall m \forall n(\triangleright\langle m \rightsquigarrow n\rangle \supset \forall t(\triangleright\langle \mathit{typeof}\ m\ t\rangle \supset \triangleright\langle \mathit{typeof}\ n\ t\rangle))$$
$$\forall m \forall n(\triangleright\langle m \rightsquigarrow^* n\rangle \supset \forall t(\triangleright\langle \mathit{typeof}\ m\ t\rangle \supset \triangleright\langle \mathit{typeof}\ n\ t\rangle))\ .$$

Theorem 6.8 *The following formula is derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *from the definition that accumulates* $\mathcal{D}(nat)$, $\mathcal{D}(list(\mathrm{atm}))$, $\mathcal{D}(intuit)$, $\mathcal{D}(\mathrm{PCF})$ *and the clause* $X \equiv X \stackrel{\triangle}{=} \top$ *defining the predicate* $\equiv\colon i \to i \to o$*:*

$$\forall m \forall t_1 \forall t_2(\triangleright\langle \mathit{typeof}\ m\ t_1\rangle \supset \triangleright\langle \mathit{typeof}\ m\ t_2\rangle \supset t_1 \equiv t_2)\ .$$

Proof We wish to show that every term has at most one type:

$$\longrightarrow \forall m \forall t_1 \forall t_2(\triangleright\langle \mathit{typeof}\ m\ t_1\rangle \supset \triangleright\langle \mathit{typeof}\ m\ t_2\rangle \supset t_1 \equiv t_2)\ .$$

Applying the $\forall\mathcal{R}$, $\supset\mathcal{R}$, $\exists\mathcal{L}$, $c\mathcal{L}$, and $\wedge\mathcal{L}$ rules to the above sequent yields

$$\mathit{nat}\ i, \mathit{seq}_i\ \mathit{nil}\ \langle \mathit{typeof}\ m\ t_1\rangle, \triangleright\langle \mathit{typeof}\ m\ t_2\rangle \longrightarrow t_1 \equiv t_2\ .$$

(Recall that $\triangleright\langle \mathit{typeof}\ m\ t_1\rangle$ is an abbreviation for $\exists i(\mathit{nat}\ i \wedge \mathit{seq}_i\ \mathit{nil}\ \langle \mathit{typeof}\ m\ t_1\rangle)$.)

We proceed with a complete induction on the height i of the first typing derivation, using the derived rule of Proposition 2.16. If we let P be the binary predicate

$$\lambda m \lambda t_1 \forall t_2(\triangleright\langle \mathit{typeof}\ m\ t_2\rangle \supset \triangleright t_1 \equiv t_2)\ ,$$

then our induction predicate IP will be

$$\lambda j \forall l \forall m \forall t(\forall n \forall u(\mathit{element}\ (\mathit{typeof}\ n\ u)\ l \supset (P\, n\, u)) \supset \mathit{seq}_j\ l\ \langle \mathit{typeof}\ m\ t\rangle \supset (P\, m\, t))\ .$$

If $(P\,m\,t)$ holds, then t is the only type we can derive for m from an empty list of typing assumptions. The induction predicate then states that if this is true for every type assignment in the list l of hypotheses, then it is also true for every type assignment we can derive from l. We omit the derivation of the conclusion from $IP\,i$, which is straightforward if we choose *nil* for l.

It remains to derive the induction step

$$nat\ j, \forall k(nat\ k \supset k < j \supset (IP\,k)) \longrightarrow (IP\,j)\ .$$

We proceed with applications of the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules, which yield the sequent

$$nat\ j, \forall k(nat\ k \supset k < j \supset (IP\,k)), \forall n \forall u \ldots, seq_j\ l\ \langle typeof\ m\ t\rangle \longrightarrow (P\,m\,t)\ .$$

Since $\langle typeof\ m\ t\rangle$ is atomic, its object logic derivation must end with the use of either a typing assumption or a clause from the specification of typing. We obtain this formally by the application of the $def\mathcal{L}$ rule to the hypothesis $seq_j\ l\ \langle typeof\ m\ t\rangle$, which gives us the two sequents

$$nat\ j, \forall k \ldots, \forall n \forall u \ldots, element\ (typeof\ m\ t)\ l \longrightarrow (P\,m\,t)$$

$$nat\ (s\ j_0), \forall k \ldots, \forall n \forall u \ldots, \exists b(prog\ (typeof\ m\ t)\ b \wedge seq_{j_0}\ l\ b) \longrightarrow (P\,m\,t)\ .$$

To derive the first sequent, we can use the $\forall n \forall u \ldots$ hypothesis (via $\forall\mathcal{L}$ and $\supset\mathcal{L}$) to conclude that P holds of m and t. For the second sequent, we can apply the $\exists\mathcal{L}$, $c\mathcal{L}$ and $\wedge\mathcal{L}$ rules, and then apply the $def\mathcal{L}$ rule to the hypothesis $prog\ (typeof\ m\ t)\ b$. This corresponds to a case analysis of the last typing rule in the typing derivation for m.

If the typing derivation for m ends with a typing rule for *zero*, then m is *zero* and t is *num*:

$$nat\ (s\ j_0), \forall k \ldots, \forall n \forall u \ldots \longrightarrow (P\,zero\,num)\ .$$

If we recall that $(P\,zero\,num)$ is $\forall t_2(\triangleright\langle typeof\ zero\ t_2\rangle \supset \triangleright num \equiv t_2)$, then it is easy to derive this formula. The typing derivation for *zero* and t_2 must also end with the typing rule for *zero*, since no other typing rules apply and there are no typing assumptions in that derivation. Thus t_2 must also be *num*. This is accomplished by first applying the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules, then applying the $\exists\mathcal{L}$, $\wedge\mathcal{L}$, and $def\mathcal{L}$ rules to $\triangleright\langle typeof\ zero\ t_2\rangle$ twice,

and finally using the $def\mathcal{R}$ rule. The cases for *true*, *false*, *succ*, *pred*, *is_zero*, and *rec* are similar.

If the typing derivation for m ends with a typing rule for *if*, then we know that m is of the form $(if\ m'\ n_1\ n_2)$, m' has type *bool*, and both n_1 and n_2 have type t:

$$nat\ (s\ j_1), \forall k \ldots, \forall n \forall u \ldots,$$
$$seq_{j_1}\ l\ (\langle typeof\ m'\ bool\rangle\ \&\ \langle typeof\ n_1\ t\rangle\ \&\ \langle typeof\ n_2\ t\rangle) \longrightarrow (P\,(if\ m'\ n_1\ n_2)\,t)\ .$$

Applying the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules brings us to the sequent

$$nat\ (s\ j_1), \forall k \ldots, \forall n \forall u \ldots, seq_{j_1}\ l\ \ldots, \triangleright\langle typeof\ (if\ m'\ n_1\ n_2)\ t_2\rangle \longrightarrow num \equiv t_2\ .$$

This second typing derivation for $(if\ m'\ n_1\ n_2)$ must also end with the typing rule for *if*; applying the $\exists\mathcal{L}$, $c\mathcal{L}$, $\wedge\mathcal{L}$, and $def\mathcal{L}$ rules twice yields

$$nat\ (s\ j_1), \forall k \ldots, \forall n \forall u \ldots, seq_{j_1}\ l\ \ldots, nat\ (s\ k_1),$$
$$seq_{k_1}\ nil\ (\langle typeof\ m'\ bool\rangle\ \&\ \langle typeof\ n_1\ t_2\rangle\ \&\ \langle typeof\ n_2\ t_2\rangle) \longrightarrow t_1 \equiv t_2\ .$$

If we apply the $def\mathcal{L}$ and $\wedge\mathcal{L}$ rules to both $seq_{j_1}\ l\ \ldots$ and $seq_{k_1}\ nil\ \ldots$ we can obtain hypotheses $seq_{j_2}\ l\ \langle typeof\ n_1\ t_1\rangle$ and $seq_{k_2}\ nil\ \langle typeof\ n_1\ t_2\rangle$. We can then apply the induction hypothesis $\forall k \ldots$ to these to conclude that $t_1 \equiv t_2$. The case for *app* is similar.

It remains to derive the case for *abs*:

$$nat\ (s\ j_1), \forall k \ldots, \forall n \forall u \ldots,$$
$$seq_{j_1}\ l\ \bigwedge x((typeof\ x\ u_1) \Rightarrow \langle typeof\ (r\,x)\ u_2\rangle) \longrightarrow (P\,(abs\ u_1\ r)\,(arr\ u_1\ u_2))\ .$$

As in the other cases, we apply the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules to obtain the sequent

$$nat\ (s\ j_1), \forall k \ldots, \forall n \forall u \ldots, seq_{j_1}\ l\ \ldots, \triangleright\langle typeof\ (abs\ u_1\ r)\ t_2\rangle \longrightarrow (arr\ u_1\ u_2) \equiv t_2\ ,$$

and then the $\exists\mathcal{L}$, $c\mathcal{L}$, $\wedge\mathcal{L}$,and $def\mathcal{L}$ rules, yielding

$$nat\ (s\ j_1), \forall k \ldots, \forall n \forall u \ldots, seq_{j_1}\ l\ \ldots, nat\ (s\ k_1),$$
$$seq_{k_1}\ nil\ \bigwedge x((typeof\ x\ u_1) \Rightarrow \langle typeof\ (r\,x)\ u_2'\rangle) \longrightarrow (arr\ u_1\ u_2) \equiv (arr\ u_1\ u_2')\ .$$

We would like to proceed as before to obtain formulas

$$seq_{j_2}\ (typeof\ x\ u_1 :: l)\ \langle typeof\ (r\,x)\ u_2\rangle$$

and

$$seq_{k_2}\ (typeof\ x\ u_1 :: nil)\ \langle typeof\ (r\,x)\ u_2' \rangle\ ,$$

and then apply the induction hypothesis to conclude that $u_2 \equiv u_2'$. This requires the use of the $\forall\mathcal{L}$ rule, so we must actually choose a term for x. In order for the induction hypothesis to apply, we must also remove the $(typeof\ x\ u_1)$ assumption from the second formula to obtain an empty list of typing assumptions. We choose the term $(rec\ u_1\ (\lambda y\, y))$ for x. It is a simple matter to construct a derivation of $\longrightarrow \triangleright\langle typeof\ (rec\ u_1\ (\lambda y\, y))\ u_1\rangle$, so we can apply the derived object logic cut rule (Theorem 5.4) to get rid of the unwanted typing assumption:

$$\ldots, \triangleright\langle typeof\ (r\,(rec\ u_1\ (\lambda y\, y)))\ u_2'\rangle \longrightarrow (arr\ u_1\ u_2) \equiv arr\ u_1\ u_2')\ .$$

To apply the induction hypothesis to

$$seq_{j_2}\ (typeof\ (rec\ u_1\ (\lambda y\, y))\ u_1 :: l)\ \langle typeof\ (r\,(rec\ u_1\ (\lambda y\, y))\ u_2\rangle$$

and $\triangleright\langle typeof\ \ (r\,(rec\ \ u_1\ \ (\lambda y\, y)))\ \ u_2'\rangle$, we must show that P holds for the new typing assumption $typeof\ (rec\ u_1\ (\lambda y\, y))\ u_1$. But we have already shown how to do this in the *rec* case of our derivation. Therefore we conclude by the induction hypothesis that $u_2 \equiv u_2'$, and hence $(arr\ u_1\ u_2) \equiv (arr\ u_1\ u_2')$. ∎

The usual informal proof of the unicity of typing relies on the requirement that the list of assumptions contains only typing assignments for variables and no more than one assignment for any particular variable. Since we have encoded the variables of PCF as variables of our object logic, which in turn are encoded as variables of $FO\lambda^{\Delta\mathbb{N}}$, we cannot state the first part of this requirement in $FO\lambda^{\Delta\mathbb{N}}$. Thus our derivation must differ from the informal proof. In fact, we make essential use of the PCF recursion construct in the *abs* case of the derivation; for an arbitrary type u, the term $(rec\ u\ (\lambda y\, y))$ has the type u and no other type. As a result, our derivation so does not generalize to languages without this construct. In the next section we give an encoding of an extension of PCF in the linear object logic of Section 5.2.2, which is encoded in $FO\lambda^{\Delta\mathbb{N}}$ using the explicit eigenvariable encoding. That encoding will allow us to capture in $FO\lambda^{\Delta\mathbb{N}}$ the typical proof of the unicity of typing.

6.4 A Language with References

In this section we consider the programming language $\text{PCF}_{:=}$, an extension of PCF with state [18]. This language extends PCF with reference types and constructs for referencing, dereferencing, assignment, and sequential evaluation. The type (*refty* τ) is the type of references to values of type τ. If m is a term of type τ, then (*ref* m) has type (*refty* τ) and evaluates to a new memory location containing the value of m. If m is a term of type (*refty* τ), then $!m$ has type τ and evaluates to the contents of the value of m. If m has type (*refty* τ) and n has type τ, then $(m := n)$ has type τ. The evaluation of $(m := n)$ changes the contents of the value of m to be the value of n; its value is the same as the value of n. If m_1 and m_2 have types τ_1 and τ_2, respectively, then $(m_1; m_2)$ has type τ_2. To evaluate $(m_1; m_2)$, we first evaluate m_1, then evaluate m_2, and finally return the value of m_2. Clearly the value of a $\text{PCF}_{:=}$ term will depend on the state in which it is evaluated, and the state may be modified in the evaluation process; thus evaluation becomes a mapping from a term-state pair to a value-state pair. The natural semantics for $\text{PCF}_{:=}$ is given in Table 6.5; following [18] we specify call-by-value evaluation. In these evaluation rules, c is used to range over locations (reference cells). In the evaluation rule for (*ref* M), c must be a new location, i.e. a location that does not occur in σ_1. The expression $\sigma[c \mapsto V]$ represents the state that is the same as σ except that location c contains the value V.

To encode $\text{PCF}_{:=}$, we use the linear object logic of Section 5.2.2, since linear logic is well-suited as a specification logic for programming languages with state [5, 6, 37]. For such languages, the order of evaluation becomes important, and so a continuation-based operational semantics is often used for the encoding. In a continuation-based semantics, each rule has at most one premise, and any additional evaluation steps are encoded in the continuation. This encoding of the evaluation steps into the continuation makes the order of evaluation explicit. Table 6.6 shows a continuation-based variation of the semantics for $\text{PCF}_{:=}$ given in Table 6.5. These semantics and their object logic encoding given below are a variation of those found in [5]. The judgement $\kappa \vdash (M, \sigma) \hookrightarrow \phi$ represents the idea that the evaluation of the term M in state σ with continuation κ results in the final answer ϕ. A continuation is a list whose elements are of the form $\hat{x}.M$, where M is a term containing the variable x. (We use $\hat{x}$ instead of λx to avoid confusion with λ-abstraction in $\text{PCF}_{:=}$.) The

Table 6.5: Natural semantics for $\mathrm{PCF}_{:=}$

$$\frac{}{(c, \sigma_0) \Downarrow (c, \sigma_0)} \qquad \frac{(M, \sigma_0) \Downarrow (V, \sigma_1)}{(\mathit{ref}\ M, \sigma_0) \Downarrow (c, \sigma_1[c \mapsto V])} \qquad \frac{(M, \sigma_0) \Downarrow (c, \sigma_1)}{(!M, \sigma_0) \Downarrow (\sigma_1(c), \sigma_1)}$$

$$\frac{(M, \sigma_0) \Downarrow (c, \sigma_1) \quad (N, \sigma_1) \Downarrow (V, \sigma_2)}{(M := N, \sigma_0) \Downarrow (V, \sigma_2[c \mapsto V])} \qquad \frac{(M, \sigma_0) \Downarrow (W, \sigma_1) \quad (N, \sigma_1) \Downarrow (V, \sigma_2)}{(M; N, \sigma_0) \Downarrow (V, \sigma_2)}$$

$$\frac{}{(\mathit{zero}, \sigma_0) \Downarrow (\mathit{zero}, \sigma_0)} \qquad \frac{}{(\mathit{true}, \sigma_0) \Downarrow (\mathit{true}, \sigma_0)} \qquad \frac{}{(\mathit{false}, \sigma_0) \Downarrow (\mathit{false}, \sigma_0)}$$

$$\frac{(M, \sigma_0) \Downarrow (V, \sigma_1)}{(\mathit{succ}\ M, \sigma_0) \Downarrow (\mathit{succ}\ V, \sigma_1)} \qquad \frac{(M, \sigma_0) \Downarrow (\mathit{zero}, \sigma_1)}{(\mathit{pred}\ M, \sigma_0) \Downarrow (\mathit{zero}, \sigma_1)} \qquad \frac{(M, \sigma_0) \Downarrow (\mathit{succ}\ V, \sigma_1)}{(\mathit{pred}\ M, \sigma_0) \Downarrow (V, \sigma_1)}$$

$$\frac{(M, \sigma_0) \Downarrow (\mathit{zero}, \sigma_1)}{(\mathit{is\_zero}\ M, \sigma_0) \Downarrow (\mathit{true}, \sigma_1)} \qquad \frac{(M, \sigma_0) \Downarrow (\mathit{succ}\ V, \sigma_1)}{(\mathit{is\_zero}\ M, \sigma_0) \Downarrow (\mathit{false}, \sigma_1)}$$

$$\frac{(M, \sigma_0) \Downarrow (\mathit{true}, \sigma_1) \quad (N_1, \sigma_1) \Downarrow (V, \sigma_2)}{(\mathit{if}\ M\ N_1\ N_2, \sigma_0) \Downarrow (V, \sigma_2)} \qquad \frac{(M, \sigma_0) \Downarrow (\mathit{false}, \sigma_1) \quad (N_2, \sigma_1) \Downarrow (V, \sigma_2)}{(\mathit{if}\ M\ N_1\ N_2, \sigma_0) \Downarrow (V, \sigma_2)}$$

$$\frac{(M, \sigma_0) \Downarrow (\lambda x : \tau.M', \sigma_1) \quad (N, \sigma_1) \Downarrow (W, \sigma_2) \quad (M'[W/x], \sigma_2) \Downarrow (V, \sigma_3)}{(M\,N, \sigma_0) \Downarrow (V, \sigma_3)}$$

$$\frac{}{(\lambda x : \tau.M, \sigma_0) \Downarrow (\lambda x : \tau.M, \sigma_0)} \qquad \frac{(M[\mathrm{rec}\ x : \tau.M\ /x], \sigma_0) \Downarrow (V, \sigma_1)}{(\mathrm{rec}\ x : \tau.M\ , \sigma_0) \Downarrow (V, \sigma_1)}$$

answer ϕ is a pair including the final value and the final state. The judgement $\kappa \vdash (V, \sigma) \dot{\hookrightarrow} \phi$ indicates that passing the value V with state σ to the continuation κ results in the final answer ϕ.

To encode $\text{PCF}_{:=}$, we use the constants

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathit{refty} & : & i_{ty} \to i_{ty} & \mathit{deref} & : & i_{tm} \to i_{tm} \\ \mathit{cell} & : & i_{lc} \to i_{tm} & \mathit{assign} & : & i_{tm} \to i_{tm} \to i_{tm} \\ \mathit{ref} & : & i_{tm} \to i_{tm} & \mathit{sequence} & : & i_{tm} \to i_{tm} \to i_{tm} \end{array}$$

in addition to the constants of Section 6.3. Once again we have labeled the type i with subscripts to improve the readability of these declarations. The subscript *lc* indicates that the argument to *cell* represents a $\text{PCF}_{:=}$ location.

The object logic predicate representing typability is denoted by the same $FO\lambda^{\Delta\mathbb{N}}$ constants as in Sections 6.2 and 6.3; its object-level specification is represented in $FO\lambda^{\Delta\mathbb{N}}$ as the definition shown in Table 6.7. Recall that *prog* A $(C_1 :: \ldots C_n :: \mathit{nil})$ $(B_1 :: \ldots B_m :: \mathit{nil})$ represents the definite clause

$$\bigwedge \bar{x} (B_1 \Rightarrow \cdots B_m \Rightarrow C_1 \multimap \cdots C_n \multimap A) \ ,$$

where the free variables of A, B_1, ..., B_m, C_1, ..., C_n are included in the list $\bar{x}$. This means that to derive an instance of A, we can instead derive the corresponding instances of B_1, ..., B_m, C_1, ..., C_n. To establish $LL; IL \rhd \langle A \rangle$, the rules of linear logic require that each assumption in IL be used exactly once in the derivation of one of the C_i's; it cannot be used in the derivation of any of the B_i's, or in the derivation of more than one C_i. In the specification of typing, no linear assumptions are introduced, so IL will be empty. In general, we will use linear formulas (C_1, ..., C_n) in the bodies of specification clauses; we use intuitionistic formulas (B_1, ..., B_n) only where we specifically wish to preclude the use of linear assumptions. This is only done in one clause in the encoding of the operational semantics, and will be discussed when it is introduced. We extend the abbreviation convention of Section 5.2.2 to the constants of this section. Thus $(\mathit{typeof}^*\ m\ t)$ abbreviates $(\lambda l\, \mathit{typeof}\ (m\, l)\ (t\, l))$, $(\mathit{refty}^*\ t)$ abbreviates $(\lambda l\, \mathit{refty}\ (t\, l))$, etc.

The semantics for $\text{PCF}_{:=}$ is more complicated than those in the previous sections. The constant $\Downarrow$ now has type $i_{tm} \to i_{st} \to i_{ans} \to \mathit{atm}$. The object logic atom $(m, s) \Downarrow f$

Table 6.6: Continuation-based natural semantics for $\mathrm{PCF}_{:=}$

$$\frac{}{\vdash (V,\sigma)\dot{\hookleftarrow}(V,\sigma)} \qquad \frac{\hat{x}.ref\ x, \kappa \vdash (M,\sigma) \hookrightarrow \phi}{\kappa \vdash (ref\ M,\sigma) \hookrightarrow \phi} \qquad \frac{\hat{x}.!x, \kappa \vdash (M,\sigma) \hookrightarrow \phi}{\kappa \vdash (!M,\sigma) \hookrightarrow \phi}$$

$$\frac{\kappa \vdash (c,\sigma)\dot{\hookleftarrow}\phi}{\kappa \vdash (c,\sigma) \hookrightarrow \phi} \qquad \frac{\kappa \vdash (c,\sigma[c \mapsto V])\dot{\hookleftarrow}\phi}{\hat{x}.ref\ x, \kappa \vdash (V,\sigma)\dot{\hookleftarrow}\phi} \qquad \frac{\kappa \vdash (\sigma(c),\sigma)\dot{\hookleftarrow}\phi}{\hat{x}.!x, \kappa \vdash (c,\sigma)\dot{\hookleftarrow}\phi}$$

$$\frac{\hat{x}.x := N, \kappa \vdash (M,\sigma) \hookrightarrow \phi}{\kappa \vdash (M := N,\sigma) \hookrightarrow \phi} \qquad \frac{\hat{x}.V := x, \kappa \vdash (N,\sigma) \hookrightarrow \phi}{\hat{x}.x := N, \kappa \vdash (V,\sigma)\dot{\hookleftarrow}\phi} \qquad \frac{\kappa \vdash (V,\sigma[c \mapsto V])\dot{\hookleftarrow}\phi}{\hat{x}.c := x, \kappa \vdash (V,\sigma)\dot{\hookleftarrow}\phi}$$

$$\frac{\hat{x}.x; N, \kappa \vdash (M,\sigma) \hookrightarrow \phi}{\kappa \vdash (M; N,\sigma) \hookrightarrow \phi} \qquad \frac{\kappa \vdash (N,\sigma) \hookrightarrow \phi}{\hat{x}.x; N, \kappa \vdash (V,\sigma)\dot{\hookleftarrow}\phi}$$

$$\frac{\kappa \vdash (zero,\sigma)\dot{\hookleftarrow}\phi}{\kappa \vdash (zero,\sigma) \hookrightarrow \phi} \qquad \frac{\kappa \vdash (true,\sigma)\dot{\hookleftarrow}\phi}{\kappa \vdash (true,\sigma) \hookrightarrow \phi} \qquad \frac{\kappa \vdash (false,\sigma)\dot{\hookleftarrow}\phi}{\kappa \vdash (false,\sigma) \hookrightarrow \phi}$$

$$\frac{\hat{x}.succ\ x, \kappa \vdash (M,\sigma) \hookrightarrow \phi}{\kappa \vdash (succ\ M,\sigma) \hookrightarrow \phi} \qquad \frac{\kappa \vdash (succ\ V,\sigma)\dot{\hookleftarrow}\phi}{\hat{x}.succ\ x, \kappa \vdash (V,\sigma)\dot{\hookleftarrow}\phi}$$

$$\frac{\hat{x}.pred\ x, \kappa \vdash (M,\sigma) \hookrightarrow \phi}{\kappa \vdash (pred\ M,\sigma) \hookrightarrow \phi} \qquad \frac{\kappa \vdash (zero,\sigma)\dot{\hookleftarrow}\phi}{\hat{x}.pred\ x, \kappa \vdash (zero,\sigma)\dot{\hookleftarrow}\phi} \qquad \frac{\kappa \vdash (V,\sigma)\dot{\hookleftarrow}\phi}{\hat{x}.pred\ x, \kappa \vdash (succ\ V,\sigma)\dot{\hookleftarrow}\phi}$$

$$\frac{\hat{x}.is\_zero\ x, \kappa \vdash (M,\sigma) \hookrightarrow \phi}{\kappa \vdash (is\_zero\ M,\sigma) \hookrightarrow \phi} \qquad \frac{\hat{x}.if\ x\ N_1\ N_2, \kappa \vdash (M,\sigma) \hookrightarrow \phi}{\kappa \vdash (if\ M\ N_1\ N_2,\sigma) \hookrightarrow \phi}$$

$$\frac{\kappa \vdash (true,\sigma)\dot{\hookleftarrow}\phi}{\hat{x}.is\_zero\ x, \kappa \vdash (zero,\sigma)\dot{\hookleftarrow}\phi} \qquad \frac{\kappa \vdash (N_1,\sigma) \hookrightarrow \phi}{\hat{x}.if\ x\ N_1\ N_2, \kappa \vdash (true,\sigma)\dot{\hookleftarrow}\phi}$$

$$\frac{\kappa \vdash (false,\sigma)\dot{\hookleftarrow}\phi}{\hat{x}.is\_zero\ x, \kappa \vdash (succ\ V,\sigma)\dot{\hookleftarrow}\phi} \qquad \frac{\kappa \vdash (N_2,\sigma) \hookrightarrow \phi}{\hat{x}.if\ x\ N_1\ N_2, \kappa \vdash (false,\sigma)\dot{\hookleftarrow}\phi}$$

$$\frac{\hat{x}.x\ N, \kappa \vdash (M,\sigma) \hookrightarrow \phi}{\kappa \vdash (M\ N,\sigma) \hookrightarrow \phi} \qquad \frac{\hat{x}.V\ x, \kappa \vdash (N,\sigma) \hookrightarrow \phi}{\hat{x}.x\ N, \kappa \vdash (V,\sigma)\dot{\hookleftarrow}\phi} \qquad \frac{\kappa \vdash (\lambda x : \tau.M,\sigma)\dot{\hookleftarrow}\phi}{\kappa \vdash (\lambda x : \tau.M,\sigma) \hookrightarrow \phi}$$

$$\frac{\kappa \vdash (M'[V/y],\sigma) \hookrightarrow \phi}{\hat{x}.(\lambda y : \tau.M')\ x, \kappa \vdash (V,\sigma)\dot{\hookleftarrow}\phi} \qquad \frac{\kappa \vdash (M[rec\ x : \tau.M\ /x],\sigma) \hookrightarrow \phi}{\kappa \vdash (rec\ x : \tau.M\ ,\sigma) \hookrightarrow \phi}$$

Table 6.7: Object logic encoding of typing for $\mathrm{PCF}_{:=}$ terms

$prog$	$(typeof^*\ zero^*\ num^*)$	
	$nil^*\quad nil^*$	
$prog$	$(typeof^*\ true^*\ bool^*)$	
	$nil^*\quad nil^*$	
$prog$	$(typeof^*\ false^*\ bool^*)$	
	$nil^*\quad nil^*$	
$prog$	$(typeof^*\ (succ^*\ M)\ num^*)$	
	$(\langle typeof^*\ M\ num^*\rangle^* ::^* nil^*)$	nil^*
$prog$	$(typeof^*\ (pred^*\ M)\ num^*)$	
	$(\langle typeof^*\ M\ num^*\rangle^* ::^* nil^*)$	nil^*
$prog$	$(typeof^*\ (is\_zero\ M)\ bool^*)$	
	$(\langle typeof^*\ M\ num^*\rangle^* ::^* nil^*)$	nil^*
$prog$	$(typeof^*\ (if^*\ M\ N_1\ N_2)\ T)$	
	$(\langle typeof^*\ M\ bool^*\rangle^* ::^* \langle typeof^*\ N_1\ T\rangle^* ::^* \langle typeof^*\ N_2\ T\rangle^* ::^* nil^*)$	nil^*
$prog$	$typeof^*\ (abs^*\ T\ R)\ (arr^*\ T\ U))$	
	$\lambda l(\bigwedge n(typeof\ n\ (T\,l) \Rightarrow \langle typeof\ (R\,l\,n)\ (U\,l)\rangle) :: nil)$	nil^*
$prog$	$(typeof^*\ (app^*\ M\ N)\ T)$	
	$(\langle typeof^*\ M\ (arr^*\ U\ T)\rangle^* ::^* \langle typeof^*\ N\ U\rangle^* ::^* nil^*)$	nil^*
$prog$	$(typeof^*\ (rec^*\ T\ R)\ T)$	
	$\lambda l(\bigwedge n(typeof\ n\ (T\,l) \Rightarrow \langle typeof\ (R\,l\,n)\ (T\,l)\rangle) :: nil)$	nil^*
$prog$	$(typeof^*\ (ref^*\ M)\ (refty^*\ T))$	
	$(\langle typeof^*\ M\ T\rangle^* ::^* nil^*)$	nil^*
$prog$	$(typeof^*\ (deref^*\ M)\ T)$	
	$(\langle typeof^*\ M\ (refty^*\ T)\rangle^* ::^* nil^*)$	nil^*
$prog$	$(typeof^*\ (assign^*\ M\ N)\ T)$	
	$(\langle typeof^*\ M\ (refty^*\ T)\rangle^* ::^* \langle typeof^*\ N\ T\rangle^* ::^* nil^*)$	nil^*
$prog$	$(typeof^*\ (sequence^*\ M\ N)\ T)$	
	$(\langle typeof^*\ M\ U\rangle^* ::^* \langle typeof^*\ N\ T\rangle^* ::^* nil^*)$	nil^*

represents the evaluation of the term m in the state s yielding the final answer f. State is encoded using the constants $null\_st\colon i_{st}$ and $extend\_st\colon i_{lc} \to i_{tm} \to i_{st} \to i_{st}$; $null\_st$ represents the state with no locations, and $(extend\_st\ c\ v\ s)$ represents the state obtained by adding the location c containing value v to the state s. A value and a state are combined into an answer using the constant $answer\colon i_{tm} \to i_{st} \to i_{ans}$; variables representing new locations are bound using $new\colon (i_{lc} \to i_{ans}) \to i_{ans}$. Our specification of evaluation will also use the predicates

$$\begin{aligned} ns\_mach\_1 &: i_{cntn} \to i_{instr} \to i_{st} \to i_{ans} \to atm \\ ns\_mach\_2 &: i_{cntn} \to i_{instr} \to i_{ans} \to atm \\ contains &: i_{lc} \to i_{tm} \to atm \\ collect\_state &: i_{st} \to atm\ . \end{aligned}$$

The object logic atom $ns\_mach\_1\ k\ i\ s\ f$ corresponds to the two judgements of Table 6.6. Continuations are constructed using $init\colon i_{cntn}$ to represent the initial continuation and $\succ\colon (i_{tm} \to i_{instr}) \to i_{cntn} \to i_{cntn}$ to extend a continuation. Instructions, constructed from the constants

$$\begin{aligned} eval &: i_{tm} \to i_{instr} & \qquad eval\_arg &: i_{tm} \to i_{tm} \to i_{instr} \\ return &: i_{tm} \to i_{instr} & \qquad apply &: i_{tm} \to i_{tm} \to i_{instr} \\ monus &: i_{tm} \to i_{instr} & \qquad new\_ref &: i_{tm} \to i_{instr} \\ zero\_test &: i_{tm} \to i_{instr} & \qquad lookup &: i_{tm} \to i_{instr} \\ switch &: i_{tm} \to i_{tm} \to i_{tm} \to i_{instr} & \qquad eval\_rvalue &: i_{tm} \to i_{tm} \to i_{instr} \\ & & \qquad update &: i_{tm} \to i_{tm} \to i_{instr}\ , \end{aligned}$$

are used to indicate the current task in the evaluation of a term. The object logic atom $ns\_mach\_2\ k\ i\ f$ is a variation of $ns\_mach\_1\ k\ i\ s\ f$ which does not contain the state; instead the contents of each location is recorded using the object logic predicate denoted by the constant *contains*. The evaluation of terms is specified using this distributed representation of state; the state portion of the final answer is constructed again using the predicate *collect_state*. The specifications for all of these predicates are represented by the $FO\lambda^{\Delta\mathbb{N}}$ definition in Tables 6.8, 6.9, and 6.10.

This encoding differs slightly from the continuation semantics in Table 6.6. The object logic judgement $nil^*; ll \rhd \langle ns\_mach\_2\ \ k\ \ (return\ \ v)\ \ f\rangle^*$ corresponds to the judgement

Table 6.8: Object logic encoding of natural semantics for $\text{PCF}_{:=}$ (part I)

prog	$((M, S) \Downarrow^* F)$
	$nil^* \quad (\langle ns\_mach\_1^* \ init^* \ (eval^* \ M) \ S \ F \rangle^* ::^* nil^*)$
prog	$(ns\_mach\_1^* \ K \ I \ (extend\_st^* \ C \ V \ S) \ F)$
	$((contains^* \ C \ V \multimap^* \langle ns\_mach\_1^* \ K \ I \ S \ F \rangle^*) ::^* nil^*) \quad nil^*$
prog	$(ns\_mach\_1^* \ K \ I \ null\_st^* \ F)$
	$(\langle ns\_mach\_2^* \ K \ I \ F \rangle^* ::^* nil^*) \quad nil^*$
prog	$(collect\_state^* \ (extend\_st^* \ C \ V \ S))$
	$(\langle contains^* \ C \ V \rangle^* ::^* \langle collect\_state^* \ S \rangle^* ::^* nil^*) \quad nil^*$
prog	$(collect\_state^* \ null\_st^*)$
	$nil^* \quad nil^*$

$\kappa \vdash (v', \sigma) \dot{\hookrightarrow} \phi$, where κ is the continuation encoded by k, v' is the value encode by v, σ is the state encoded by the list ll of *contains* assumptions, and ϕ is the answer encoded by f. However, the specification for $ns\_mach\_2^* \ k \ (return^* \ v) \ f$ takes the first instruction from k and substitutes in the value v to obtain the new instruction. This new instruction then determines the next step in the evaluation. On the other hand, the rules of Table 6.6 examine the return value and the first term of the continuation to determine the next evaluation step. Other than this small difference, the encoding mirrors the continuation semantics very closely.

The distributed encoding of state in Tables 6.8, 6.9, and 6.10 makes vital use of linear implication. Since each assumption of the form $contains^* \ c \ v$ is a linear assumption, it can only be used once. This linearity is used, for example, in the clause for *ns_mach_2* with the instruction $(update^* \ (cell^* \ c) \ v)$; the desired behavior is that the contents of location c be replaced by the value v. This clause has two linear formulas in its body, $\langle contains^* \ c \ w \rangle^*$ and $(contains^* \ c \ v \multimap^* \langle ns\_mach\_2^* \ k \ (return^* \ v) \ f \rangle^*)$. Each *contains* assumption must be used exactly once in the derivation of these two formulas. Since there is no clause for *contains* in the object logic theory, the first formula must be derived by the initial rule, and so will use the one assumption representing the contents of location c. The remainder of the state is then available for the other formula, which adds a new assumption

Table 6.9: Object logic encoding of natural semantics for $\mathrm{PCF}_{:=}$ (part II)

$$
\begin{array}{l}
prog\ \ (ns\_mach\_2^*\ \ init^*\ \ (return^*\ V)\ \ (answer^*\ V\ S)) \\
\quad (\langle collect\_state^*\ S\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ \ (ns\_mach\_2^*\ \ (I \succ^* K)\ \ (return^*\ V)\ \ F) \\
\quad (\langle ns\_mach\_2^*\ \ K\ \ (\lambda l\, I\, l\, (V\, l))\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
\\
prog\ \ (ns\_mach\_2^*\ \ K\ \ (eval^*\ (cell^*\ C))\ \ F) \\
\quad (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ (cell^*\ C))\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
\\
prog\ \ (ns\_mach\_2^*\ \ K\ \ (eval^*\ (ref^*\ M))\ \ F) \\
\quad (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\, new\_ref\ v) \succ^* K)\ \ (eval^*\ M)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ \ (ns\_mach\_2^*\ \ K\ \ (new\_ref^*\ V)\ \ (new^*\ F)) \\
\quad \lambda l(\bigwedge c(contains\ c\ (V\, l) \multimap \langle ns\_mach\_2\ \ (K\, l)\ \ (return\ (cell\ c))\ \ (F\, l\, c)\rangle) :: nil^*) \\
\quad nil^* \\
\\
prog\ \ (ns\_mach\_2^*\ \ K\ \ (eval^*\ (deref^*\ M))\ \ F) \\
\quad (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\, lookup\ v) \succ^* K)\ \ (eval^*\ M)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ \ (ns\_mach\_2^*\ \ K\ \ (lookup^*\ (cell^*\ C))\ \ F) \\
\quad (\langle contains^*\ C\ V\rangle^* ::^* \\
\quad\ (contains^*\ C\ V \multimap^* \langle ns\_mach\_2^*\ \ K\ \ (return^*\ V)\ \ F\rangle^*) ::^* nil^*) \\
\quad nil^* \\
\\
prog\ \ (ns\_mach\_2^*\ \ K\ \ (eval^*\ (assign^*\ M\ N))\ \ F) \\
\quad (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\, eval\_rvalue\ v\ (N\, l)) \succ^* K)\ \ (eval^*\ M)\ \ F\rangle^* ::^* nil^*) \\
\quad nil^* \\
prog\ \ (ns\_mach\_2^*\ \ K\ \ (eval\_rvalue^*\ V\ N)\ \ F) \\
\quad (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\, update\ (V\, l)\ v) \succ^* K)\ \ (eval^*\ N)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ \ (ns\_mach\_2^*\ \ K\ \ (update^*\ (cell^*\ C)\ V)\ \ F) \\
\quad (\langle contains^*\ C\ W\rangle^* ::^* \\
\quad\ (contains^*\ C\ V \multimap^* \langle ns\_mach\_2^*\ \ K\ \ (return^*\ V)\ \ F\rangle^*) ::^* nil^*) \\
\quad nil^* \\
\\
prog\ \ (ns\_mach\_2^*\ \ K\ \ (eval^*\ (sequence^*\ M\ N))\ \ F) \\
\quad (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\, eval\ (N\, l)) \succ^* K)\ \ (eval^*\ M)\ \ F\rangle^* ::^* nil^*) \qquad nil^*
\end{array}
$$

Table 6.10: Object logic encoding of natural semantics for $\mathrm{PCF}_{:=}$ (part III)

$$
\begin{array}{ll}
prog & (ns\_mach\_2^*\ \ (K\,l)\ \ (eval^*\ zero^*)\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ zero^*)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval^*\ true^*)\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ true^*)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval^*\ false^*)\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ false^*)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval^*\ (succ^*\ M))\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\ return\ (succ\ v)) \succ^* K)\ \ (eval^*\ M)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
 & \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval^*\ (pred^*\ M))\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\ monus\ v) \succ^* K)\ \ (eval^*\ M)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (monus^*\ zero^*)\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ zero^*)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (monus^*\ (succ^*\ V))\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ V)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
 & \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval^*\ (is\_zero^*\ M))\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ (\lambda l \lambda v\, zero\_test\ v) \succ^* K\ \ (eval^*\ M)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (zero\_test^*\ zero^*)\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ true^*)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (zero\_test^*\ (succ^*\ V))\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ false^*)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
 & \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval^*\ (if^*\ M\ N_1\ N_2))\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\, switch\ v\ (N_1\,l)\ (N_2\,l)) \succ^* K)\ \ (eval^*\ M)\ \ F\rangle^* ::^* nil^*) \\
 & nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (switch^*\ true^*\ N_1\ N_2)\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (eval^*\ N_1)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (switch^*\ false^*\ N_1\ N_2)\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (eval^*\ N_2)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
 & \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval^*\ (app^*\ M\ N))\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\, eval\_arg\ v\ (N\,l)) \succ^* K)\ \ (eval^*\ M)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval\_arg^*\ V\ N)\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ ((\lambda l \lambda v\, apply\ (V\,l)\ v) \succ^* K)\ \ (eval^*\ N)\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (apply^*\ (abs^*\ T\ R)\ V)\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (eval^*\ (\lambda l\, R\, l\, (V\,l)))\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
 & \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval^*\ (abs^*\ T\ R))\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ (abs^*\ T\ R))\ \ F\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (ns\_mach\_2^*\ \ K\ \ (eval^*\ (rec^*\ T\ R))\ \ F) \\
 & (\langle ns\_mach\_2^*\ \ K\ \ (eval^*\ (\lambda l\, R\, l\, (rec\ (T\,l)\ (R\,l))))\ \ F\rangle^* ::^* nil^*) \qquad nil^*
\end{array}
$$

about the contents of c and then continues the evaluation encoded in the continuation k. The linearity of the *contains* assumptions is also used in the clause for *ns_mach_2* with the instruction $(return^*\ v)$ and the continuation $init^*$. This clause represents the situation where the evaluation is complete and we wish to construct the final answer from the value v and the state encoded in the assumptions. The clause has the single linear formula $\langle collect\_state^*\ s\rangle^*$ as its body. Thus the derivation of this formula must use all of the *contains* assumptions; this ensures that the constructed state includes all of the locations represented in the assumptions. Dually, the clause for $\Downarrow$ in Table 6.8 has a single intuitionistic formula $\langle ns\_mach\_1^*\ init^*\ (eval^*\ m)\ s\ f\rangle^*$ as its body. This clause represents the situation where we wish to evaluate the term m in the state s. Since the formula in the body is intuitionistic, it must be derived from an empty set of linear assumptions. Since there are no linear formulas in the body, this means that $(m, s) \Downarrow^* f$ is only derivable from an empty set of linear assumptions, i.e. the state is entirely represented in s.

We also introduce typing predicates for continuations, instructions, and answers:

$$\begin{aligned} typeof_{cntn} &: i_{cntn} \to i_{ty} \to atm \\ typeof_{instr} &: i_{instr} \to i_{ty} \to atm \\ typeof_{ans} &: i_{ans} \to i_{ty} \to atm\ . \end{aligned}$$

The object-level specification for these predicates is represented in $FO\lambda^{\Delta\mathbb{N}}$ by the definition of Table 6.11. A continuation has type $(arr^*\ t\ u)$ if it expects a value of type t in order to produce a value of type u. Instructions are typed in the same way as the corresponding terms. The type of an answer is the same as the type of its value component under some typing assumptions for any new memory locations. These assumptions must be consistent with the values stored in those locations; this consistency is expressed by the predicate $well\_typed: i_{st} \to atm$.

We now present the theorems we have derived in $FO\lambda^{\Delta\mathbb{N}}$ about this object logic encoding of $\mathrm{PCF}_{:=}$. We will refer to the collected clauses of Tables 6.7, 6.8, 6.9, 6.10, and 6.11 as the definition $\mathcal{D}(\mathrm{PCF}_{:=})$. To simplify the presentation of our theorems, we introduce

Table 6.11: Encoding of typing for $\mathrm{PCF}_{:=}$ continuations, instructions, and answers

$$
\begin{array}{ll}
prog & (typeof^*_{cntn}\ \ init^*\ \ (arr^*\ T\ T)) \\
 & nil^* \qquad nil^* \\
prog & (typeof^*_{cntn}\ \ (I \succ^* K)\ \ (\mathrm{arr}^*\ T\ U)) \\
 & (\lambda l\ \bigwedge v(typeof\ v\ (T\ l) \Rightarrow \langle typeof_{instr}\ \ (I\ l\ v)\ \ (T'\ l)\rangle) ::^* \\
 & \ \langle typeof^*_{cntn}\ \ K\ \ (arr^*\ T'\ U)\rangle^* ::^* nil^*) \\
 & nil^* \\
 & \\
prog & (typeof^*_{instr}\ \ (eval^*\ M)\ \ T) \\
 & (\langle typeof^*\ M\ T\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (return^*\ V)\ \ T) \\
 & (\langle typeof^*\ V\ T\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (monus^*\ M)\ \ num^*) \\
 & (\langle typeof^*\ M\ num^*\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (zero\_test^*\ M)\ \ bool^*) \\
 & (\langle typeof^*\ M\ num^*\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (switch^*\ M\ N_1\ N_2)\ \ T) \\
 & (\langle typeof^*\ M\ bool\rangle^* ::^* \langle typeof^*\ N_1\ T\rangle^* ::^* \langle typeof^*\ N_2\ T\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (eval\_arg^*\ M\ N)\ \ T) \\
 & (\langle typeof^*\ M\ (arr^*\ U\ T)\rangle^* ::^* \langle typeof^*\ N\ U\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (apply^*\ M\ N)\ \ T) \\
 & (\langle typeof^*\ M\ (arr^*\ U\ T)\rangle^* ::^* \langle typeof^*\ N\ U\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (new\_ref^*\ M)\ \ (refty^*\ T)) \\
 & (\langle typeof^*\ M\ T\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (lookup^*\ M)\ \ T) \\
 & (\langle typeof^*\ M\ (refty^*\ T)\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (eval\_rvalue^*\ M\ N)\ \ T) \\
 & (\langle typeof^*\ M\ (refty^*\ T)\rangle^* ::^* \langle typeof^*\ N\ T\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{instr}\ \ (update^*\ M\ N)\ \ T) \\
 & (\langle typeof^*\ M\ (refty^*\ T)\rangle^* ::^* \langle typeof^*\ N\ T\rangle^* ::^* nil^*) \qquad nil^* \\
 & \\
prog & (typeof^*_{ans}\ \ (answer^*\ V\ S)\ \ T) \\
 & (\langle typeof^*\ V\ T\rangle^* ::^* \langle well\_typed^*\ S\rangle^* ::^* nil^*) \qquad nil^* \\
prog & (typeof^*_{ans}\ \ (new^*\ F)\ \ T) \\
 & \lambda l\,(\bigwedge c(typeof\ \ (cell\ c)\ \ (refty\ \ (U\ l)) \Rightarrow \langle typeof_{ans}\ \ (F\ l\ c)\ \ (T\ l)\rangle) :: nil) \qquad nil^* \\
 & \\
prog & (well\_typed^*\ null\_st^*) \\
 & nil^* \qquad nil^* \\
prog & (well\_typed^*\ (extend\_st^*\ C\ V\ S)) \\
 & (\langle typeof^*\ (cell^*\ C)\ (refty^*\ T)\rangle^* ::^* \langle typeof^*\ V\ T\rangle^* ::^* \langle well\_typed^*\ S\rangle^* ::^* nil^*) \\
 & nil^*
\end{array}
$$

Table 6.12: Meta-logic predicates for $\mathrm{PCF}_{:=}$ stores

$$
\begin{array}{rcl}
\mathit{store}\ LL & \triangleq & \mathit{list}\ LL \wedge \forall a(\mathit{element}\ a\ LL \supset \\
& & \qquad \exists c \exists v(a \equiv_{atm^*} (\mathit{contains}^*\ c\ v))) \\
\\
\mathit{store\_typing}\ IL & \triangleq & \mathit{list}\ IL \wedge \\
& & \forall a(\mathit{element}\ a\ IL \supset \\
& & \quad \exists c \exists t(a \equiv_{atm^*} (\mathit{typeof}^*\ (\mathit{cell}^*\ c)\ (\mathit{refty}^*\ t)))) \wedge \\
& & \forall c \forall t_1 \forall t_2(\mathit{element}\ (\mathit{typeof}^*\ (\mathit{cell}^*\ c)\ (\mathit{refty}^*\ t_1))\ IL \supset \\
& & \qquad \mathit{element}\ (\mathit{typeof}^*\ (\mathit{cell}^*\ c)\ (\mathit{refty}^*\ t_2))\ IL \supset \\
& & \qquad t_1 \equiv_{i^*} t_2) \\
\\
\mathit{store\_typeof}\ LL\ IL & \triangleq & \forall c \forall v(\mathit{element}\ (\mathit{contains}^*\ c\ v)\ LL \supset \\
& & \qquad \exists t(\mathit{element}\ (\mathit{typeof}^*\ (\mathit{cell}^*\ c)\ (\mathit{refty}^*\ t))\ IL \wedge \\
& & \qquad IL; \mathit{nil}^* \rhd \langle \mathit{typeof}^*\ v\ t \rangle^*)) \\
\\
A \equiv_{atm^*} A & \triangleq & \top \\
\\
X \equiv_{i^*} X & \triangleq & \top
\end{array}
$$

several $FO\lambda^{\Delta\mathbb{N}}$ predicates:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{store} & : & \mathit{atmlst}^* \to o & \equiv^*_{atm} & : & \mathit{atm}^* \to \mathit{atm}^* \to o \\
\mathit{store\_typing} & : & \mathit{atmlst}^* \to o & \equiv^*_{i} & : & i^* \to i^* \to o \\
\mathit{store\_typeof} & : & \mathit{atmlst}^* \to \mathit{atmlst}^* \to o & & & .
\end{array}
$$

The *store* predicate indicates that a list of object logic atoms is a valid distributed encoding of state, that is, its elements are of the form $\mathit{contains}^*\ c\ v$. The predicate *store_typing* holds if its argument is a valid list of typing assumptions for locations. The *store_typeof* predicate holds for a store and store typing if every location in the store is assigned a type by the store typing that agrees with a type of the value stored in the location. Finally, $\equiv^*_{atm}$ and $\equiv^*_i$ encode syntactic identity over the types atm^* and i^*. The definition $\mathcal{D}(\mathit{store})$ for these predicates is presented in Table 6.12. The following theorems state that we have derived the subject reduction and unicity of typing properties for $\mathrm{PCF}_{:=}$ in $FO\lambda^{\Delta\mathbb{N}}$. The $FO\lambda^{\Delta\mathbb{N}}$ derivations again closely follow the informal proofs of these properties. We expect that the determinacy of semantics is also derivable, but have not yet shown this. We use the

following abbreviations from Section 5.2.2: $\mathcal{D}(lists)$ for

$$\mathcal{D}(list^*(\mathrm{atm})) \cup \mathcal{D}(list^*(prp)) \cup \mathcal{D}(list^{**}(\mathrm{atm})) \cup \mathcal{D}(list^{**}(prp)) \ ,$$

and $\mathcal{D}(evars)$ for

$$\mathcal{D}(evars(\mathrm{atm})) \cup \mathcal{D}(evars(prp)) \cup \mathcal{D}(evars(\mathrm{atmlst})) \cup \mathcal{D}(evars(prplst)) \ .$$

Theorem 6.9 *The following formulas are derivable in $FO\lambda^{\Delta\mathbb{N}}$ from the definition that accumulates $\mathcal{D}(nat)$, $\mathcal{D}(lists)$, $\mathcal{D}(evars)$, $\mathcal{D}(linear)$, $\mathcal{D}(\mathrm{PCF}_{:=})$, and $\mathcal{D}(store)$:*

$$\begin{array}{l}
\forall m \forall s \forall f (\rhd(\langle (m,s) \Downarrow^* f \rangle^*) \supset \\
\qquad \forall ts \forall t (store_typing\ il \supset il; nil^* \rhd \langle well_typed^*\ s \rangle^* \supset \\
\qquad\qquad il; nil^* \rhd \langle typeof^*\ m\ t \rangle^* \supset \\
\qquad\qquad il; nil^* \rhd \langle typeof^*_{\mathrm{ans}}\ f\ t \rangle^*))
\end{array}$$

$$\begin{array}{l}
\forall ll \forall k \forall i \forall f (store\ ll \supset nil^*; ll \rhd \langle ns_mach_2^*\ k\ i\ f \rangle^* \supset \\
\qquad \forall il \forall t \forall u (store_typing\ il \supset store_typeof\ ll\ il \supset \\
\qquad\qquad il; nil^* \rhd \langle typeof^*_{cntn}\ k\ (\mathrm{arr}^*\ t\ u) \rangle^* \supset \\
\qquad\qquad il; nil^* \rhd \langle typeof^*_{instr}\ i\ t \rangle^* \supset \\
\qquad\qquad il; nil^* \rhd \langle typeof^*_{\mathrm{ans}}\ f\ u \rangle^*)) \ .
\end{array}$$

Theorem 6.10 *The formula*

$$\forall m \forall t_1 \forall t_2 (\rhd \langle typeof^*\ m\ t_1 \rangle^* \supset \rhd \langle typeof^*\ m\ t_2 \rangle^* \supset t_1 \equiv_{i^*} t_2)$$

is derivable in $FO\lambda^{\Delta\mathbb{N}}$ from the definition that accumulates $\mathcal{D}(nat)$, $\mathcal{D}(lists)$, $\mathcal{D}(evars)$, $\mathcal{D}(linear)$, $\mathcal{D}(\mathrm{PCF}_{:=})$, and $\mathcal{D}(store)$.

Proof To derive the unicity of typing

$$\longrightarrow \forall m \forall t_1 \forall t_2 (\rhd \langle typeof^*\ m\ t_1 \rangle^* \supset \rhd \langle typeof^*\ m\ t_2 \rangle^* \supset t_1 \equiv_{i^*} t_2) \ ,$$

we begin by applying the $\forall\mathcal{R}$, $\supset\mathcal{R}$, $\exists\mathcal{L}$, $c\mathcal{L}$, and $\wedge\mathcal{L}$ rules, obtaining

$$nat\ i, seq_i\ nil^*\ nil^*\ \langle typeof^*\ m\ t_1 \rangle^*, \rhd \langle typeof^*\ m\ t_2 \rangle^* \longrightarrow t_1 \equiv_{i^*} t_2 \ .$$

We proceed with a complete induction on the height i of the first typing derivation. Let P_1 be the predicate

$$\lambda il \forall a(element\ a\ il\ \supset\ \exists x \exists t(a \equiv_{atm^*} (typeof^*\ (fst_i^*\ x)\ t)))$$

and P_2 the predicate

$$\lambda il \forall x \forall t_1 \forall t_2(element\ (typeof^*\ x\ t_1)\ il\ \supset\ element\ (typeof^*\ x\ t_2)\ il\ \supset\ t_1 \equiv_{i^*} t_2)\ .$$

These predicates encode the requirements that the list of assumptions contains only typing assignments for variables and assigns only one type to any one variable. Our induction predicate IP is then

$$\begin{aligned}&\lambda j \forall il(list\ il \supset P_1\ il \supset P_2\ il\ \supset\\ &\quad \forall m \forall t_1 \forall t_2(seq_j\ il\ nil^*\ \langle typeof^*\ m\ t_1\rangle^*\ \supset\\ &\qquad il; nil^* \rhd \langle typeof^*\ m\ t_2\rangle^*\ \supset\ t_1 \equiv_{i^*} t_2))\ .\end{aligned}$$

The desired conclusion follows easily from this induction predicate, since P_1 and P_2 hold vacuously for nil^*.

For the induction step, we must derive the sequent

$$nat\ j, \forall k(nat\ k \supset k < j \supset (IP\ k)) \longrightarrow (IP\ j)\ .$$

We proceed with applications of the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules, which yield the sequent

$$\begin{aligned}&nat\ j, \forall k \ldots, list\ il, P_1\ il, P_2\ il,\\ &\quad seq_j\ il\ nil^*\ \langle typeof^*\ m\ t_1\rangle^*,\\ &\qquad il; nil^* \rhd \langle typeof^*\ m\ t_2\rangle^* \longrightarrow t_1 \equiv_{i^*} t_2\ .\end{aligned}$$

Since $\langle typeof^*\ m\ t_1\rangle^*$ represents an atomic proposition, its object logic derivation must end with the use of either a typing assumption or a clause from the specification of typing. We obtain this formally by applying the $def\mathcal{L}$ rule to the hypothesis seq_j ..., which gives us the two sequents

$$\ldots,\ element\ (typeof^*\ m\ t_1)\ il,\ il; nil^* \rhd \langle typeof^*\ m\ t_2\rangle^* \longrightarrow t_1 \equiv_{i^*} t_2$$

$$\ldots, \exists ll_0 \exists il_0\ (\mathit{list}\ ll_0\ \wedge\ \mathit{list}\ il_0\ \wedge\ \mathit{prog}\ (\mathit{typeof}^*\ m\ t_1)\ ll_0\ il_0\ \wedge \\ \mathit{split\_seq}_{j_1}\ \mathit{il}\ \mathit{nil}^*\ ll_0\ \wedge\ \mathit{split\_seq}_{j_1}\ \mathit{il}\ \mathit{nil}^*\ il_0), \\ \mathit{il}; \mathit{nil}^* \rhd \langle \mathit{typeof}^*\ m\ t_2 \rangle^* \longrightarrow t_1 \equiv_{i^*} t_2\ .$$

We proceed to derive the first sequent, representing the case where the first typing derivation for m ends with the use of a typing assumption. First we can use the elided hypothesis $(P_1\ il)$ to conclude that m is an eigenvariable:

$$\ldots,\ \mathit{element}\ (\mathit{typeof}^*\ (\mathit{fst}_i^*\ m')\ t_1)\ \mathit{il},\ \mathit{il}; \mathit{nil}^* \rhd \langle \mathit{typeof}^*\ (\mathit{fst}_i^*\ m')\ t_2 \rangle^* \longrightarrow t_1 \equiv_{i^*} t_2\ .$$

Given that, the second typing derivation for m must also end with the use of a typing assumption, since none of the specification clauses apply. We obtain this formally by applying the $\exists\mathcal{L}$, $\wedge\mathcal{L}$, and $\mathit{def}\mathcal{L}$ rule to the hypothesis $\mathit{il}; \mathit{nil}^* \rhd \ldots$, which gives us the two sequents

$$\ldots,\ \mathit{element}\ (\mathit{typeof}^*\ (\mathit{fst}_i^*\ m')\ t_1)\ \mathit{il},\ \mathit{element}\ (\mathit{typeof}^*\ (\mathit{fst}_i^*\ m')\ t_2)\ \mathit{il} \longrightarrow t_1 \equiv_{i^*} t_2$$

$$\ldots, \mathit{element}\ (\mathit{typeof}^*\ (\mathit{fst}_i^*\ m')\ t_1)\ \mathit{il}, \\ \exists ll_0 \exists il_0 (\mathit{list}\ ll_0\ \wedge\ \mathit{list}\ il_0\ \wedge\ \mathit{prog}\ (\mathit{typeof}^*\ (\mathit{fst}_i^*\ m')\ t_2)\ ll_0\ il_0\ \wedge \\ \mathit{split\_seq}_{k_1}\ \mathit{il}\ \mathit{nil}^*\ ll_0\ \wedge\ \mathit{split\_seq}_{k_1}\ \mathit{il}\ \mathit{nil}^*\ il_0) \longrightarrow t_1 \equiv_{i^*} t_2\ .$$

The second sequent corresponds to the possibility that the typing derivation ends with the use of a specification clause. We show this is not applicable by applying the $\exists\mathcal{L}$ and $\wedge\mathcal{L}$ rules, and then apply the $\mathit{def}\mathcal{L}$ rule to the hypothesis $\mathit{prog}\ (\mathit{typeof}^*\ (\mathit{fst}_i^*\ m')\ t_2)\ ll_0\ il_0$. The first sequent indicates that the typing derivation ends with a use of a typing assumption. As a result, the elided hypothesis $(P_2\ il)$ allows us to conclude $t_1 \equiv_{i^*} t_2$.

We now return to the sequent that representing the case where the first typing derivation of m ends with the use of a clause specifying a typing rule:

$$\ldots, \exists ll_0 \exists il_0\ (\mathit{list}\ ll_0\ \wedge\ \mathit{list}\ il_0\ \wedge\ \mathit{prog}\ (\mathit{typeof}^*\ m\ t_1)\ ll_0\ il_0\ \wedge \\ \mathit{split\_seq}_{j_1}\ \mathit{il}\ \mathit{nil}^*\ ll_0\ \wedge\ \mathit{split\_seq}_{j_1}\ \mathit{il}\ \mathit{nil}^*\ il_0), \\ \mathit{il}; \mathit{nil}^* \rhd \langle \mathit{typeof}^*\ m\ t_2 \rangle^* \longrightarrow t_1 \equiv_{i^*} t_2\ .$$

We apply the $\exists\mathcal{L}$, $c\mathcal{L}$ and $\wedge\mathcal{L}$ rules, and then apply the $\mathit{def}\mathcal{L}$ rule to the hypothesis $\mathit{prog}\ (\mathit{typeof}^*\ m\ t_1)\ ll_0\ il_0$. This corresponds to a case analysis of the last typing rule used in the first typing derivation for m.

If the first typing derivation for m ends with a typing rule for *zero*, then m is $zero^*$ and t_1 is num^*:

$$\ldots,\ split\_seq_{j_1}\ il\ nil^*\ nil^*,\ il; nil^* \rhd \langle typeof^*\ zero^*\ t_2\rangle^* \longrightarrow num^* \equiv_{i^*} t_2\ .$$

The second typing derivation for m must also end with the typing rule for *zero*, since no other typing rules apply and by $(P_1\ il)$ the typing assumptions only apply to variables. We deduce this in $FO\lambda^{\Delta\mathbb{N}}$ by first applying the $\exists\mathcal{L}$, $\wedge\mathcal{L}$, and $def\mathcal{L}$ rules to $il; nil^* \rhd \ldots$. This gives us two sequents corresponding to the possibilities that the typing derivation ends with the use of a typing assumption or the use of a typing rule:

$$\ldots,\ split\_seq_{j_1}\ il\ nil^*\ nil^*,\ element\ (typeof^*\ zero^*\ t_2)\ il \longrightarrow num^* \equiv_{i^*} t_2$$

$$\begin{aligned}&\ldots, split\_seq_{j_1}\ il\ nil^*\ nil^*,\\ \exists ll_0 \exists il_0\ (&list\ ll_0\ \wedge\ list\ il_0\ \wedge\ prog\ (typeof^*\ zero^*\ t_2)\ ll_0\ il_0\ \wedge\\ &split\_seq_{k_1}\ il\ nil^*\ ll_0\ \wedge\ split\_seq_{k_1}\ il\ nil^*\ il_0) \longrightarrow num^* \equiv_{i^*} t_2\ .\end{aligned}$$

We show that the first case is not relevant by using the elided hypothesis $(P_1\ il)$ to get the hypothesis

$$(typeof^*\ zero^*\ num^*) \equiv_{atm^*} (typeof^*\ (fst_i^*\ x)\ t)\ ,$$

and then applying $def\mathcal{L}$ to it. For the second sequent we apply the $\exists\mathcal{L}$ and $\wedge\mathcal{L}$ rules and then apply the $def\mathcal{L}$ to $prog\ (typeof^*\ zero^*\ t_2)\ ll_0\ il_0$, yielding

$$\ldots, split\_seq_{j_1}\ il\ nil^*\ nil^* \longrightarrow num^* \equiv_{i^*} num^*\ .$$

This shows that since the second typing derivation ends with the typing rule for *zero*, t_2 must also be num^*. We complete the derivation with an application of the $def\mathcal{R}$ rule. The cases for *true*, *false*, *succ*, *pred*, *is_zero*, and *rec* are similar.

If the first typing derivation for m ends with a typing rule for *if*, then we know that m is of the form $(if^*\ m'\ n_1\ n_2)$, m' has type *bool*, and both n_1 and n_2 have type t_1:

$$\begin{aligned}\ldots, split\_seq_{j_1}\ il\ nil^*\ (&\langle typeof^*\ m'\ bool^*\rangle^* ::^*\\ &\langle typeof^*\ n_1\ t_1\rangle^* ::^*\\ &\langle typeof^*\ n_2\ t_1\rangle^* ::^*\ nil^*),\\ il; nil^* \rhd \langle typeof^*\ (if^*\ m'\ n_1\ n_2)\ t_2\rangle^* &\longrightarrow t_1 \equiv_{i^*} t_2\ .\end{aligned}$$

The second typing derivation for $(if^*\ m'\ n_1\ n_2)$ must also end with the typing rule for *if*; this is deduced in $FO\lambda^{\Delta\mathbb{N}}$ in the same way as in the *zero* case. This brings us to the sequent

$$\begin{array}{l} \ldots, split\_seq_{j_1}\ il\ nil^*\ \ldots, nat\ (s\ k_1), \\ split\_seq_{k_1}\ il\ nil^*\ (\langle typeof^*\ m'\ bool^*\rangle^* ::^* \\ \qquad\qquad \langle typeof^*\ n_1\ t_2\rangle^* ::^* \\ \qquad\qquad \langle typeof^*\ n_2\ t_2\rangle^* ::^* nil^*) \longrightarrow t_1 \equiv_{i^*} t_2\ . \end{array}$$

By applying the $def\mathcal{L}$, $\exists\mathcal{L}$, $c\mathcal{L}$, and $\wedge\mathcal{L}$ rules to both $split\_seq_{j_1}\ \ldots$ and $split\_seq_{k_1}\ \ldots$ we can obtain hypotheses $seq_{j_2}\ il\ nil^*\ \langle typeof\ n_1\ t_1\rangle$ and $seq_{k_2}\ il\ nil^*\ \langle typeof\ n_1\ t_2\rangle$. We can then apply the induction hypothesis $\forall k \ldots$ to these to conclude that $t_1 \equiv t_2$. The cases for *app*, *ref*, *deref*, *assign*, and *sequence* are similar.

It remains to derive the case for *abs*:

$$\begin{array}{l} \ldots, split\_seq_{j_1}\ il\ nil^*\ \lambda l(\bigwedge x((typeof\ x\ (u\, l)) \Rightarrow \\ \qquad\qquad \langle typeof\ (r\, l\, x)\ (t_1'\, l)\rangle) :: nil), \\ \qquad il; nil^* \rhd \langle typeof^*\ (abs^*\ u\ r)\ t_2\rangle^* \longrightarrow (arr^*\ u\ t_1') \equiv_{i^*} t_2\ . \end{array}$$

As in the other cases, we deduce that the second typing derivation must end with the same typing rule:

$$\begin{array}{l} \ldots, split\_seq_{j_1}\ il\ nil^*\ \ldots, nat\ (s\ k_1), \\ split\_seq_{k_1}\ il\ nil^*\ \lambda l(\bigwedge x((typeof\ x\ (u\, l)) \Rightarrow \\ \qquad\qquad \langle typeof\ (r\, l\, x)\ (t_2'\, l)\rangle) :: nil) \longrightarrow (arr^*\ u\ t_1') \equiv_{i^*} (arr^*\ u\ t_2')\ . \end{array}$$

By applying the $def\mathcal{L}$, $\exists\mathcal{L}$, $c\mathcal{L}$, and $\wedge\mathcal{L}$ rules to both $split\_seq_{j_1}\ \ldots$ and $split\_seq_{k_1}\ \ldots$ we can obtain hypotheses

$$seq_{j_2}\ \lambda l((typeof\ (fst_i\ l)\ (u\,(rst\ l))) :: (il\,(rst\ l)))\ nil^*\ \lambda l\langle typeof\ (r\,(rst\ l)\,(fst_i\ l))\ (t_1'\, l)\rangle$$

and

$$seq_{k_2}\ \lambda l((typeof\ (fst_i\ l)\ (u\,(rst\ l))) :: (il\,(rst\ l)))\ nil^*\ \lambda l\langle typeof\ (r\,(rst\ l)\,(fst_i\ l))\ (t_2'\, l)\rangle\ .$$

To apply the elided induction hypothesis $\forall k \ldots$ to these, we must be able to show that the extended list of typing assumptions satisfies the predicates *list*, P_1 and P_2. To show these,

we first derive the following formulas by simple induction:

$$\forall il(\mathit{list}\ il \supset \mathit{list}\ (\lambda l\, il\, (\mathit{rst}\ l)))$$

$$\forall a \forall il(\mathit{list}\ (\lambda l\, il\, (\mathit{rst}\ l)) \supset \mathit{element}\ a\ (\lambda l\, il\, (\mathit{rst}\ l)) \supset \exists a'(a \equiv_{atm^*} (\lambda l\, a'\, (\mathit{rst}\ l)) \wedge \mathit{element}\ a'\ il))\ .$$

It follows easily from the first of these that the extended list of assumptions satisfies the *list* predicate, since it has length one greater than $\lambda l\, il\, (\mathit{rst}\ l))$. Since il satisfies P_1, to show that the extended list of assumptions satisfies P_1, it suffices by the second formula above to show that the new typing assumption $\lambda l\, (\mathit{typeof}\ (\mathit{fst}_i\ l)\ (u\, (\mathit{rst}\ l)))$ gives a type for a variable; this is immediate. Similarly, to show that the extended list of assumptions satisfies P_2, it suffices to show that the new typing assumption is consistent with any other typing assumption for $(\mathit{fst}_i\ l)$ in $\lambda l\, il\, (\mathit{rst}\ l)$:

$$\ldots, \mathit{element}\ (\lambda l\, \mathit{typeof}\ (\mathit{fst}_i\ l)\ (u'\, l))\ (\lambda l\, il(\mathit{rst}\ l)) \longrightarrow (\lambda l\, u(\mathit{rst}\ l)) \equiv_{i^*} u'\ .$$

But $\lambda l\, il\, (\mathit{rst}\ l)$ cannot contain any typing assumptions for $(\mathit{fst}_i\ l)$, since it cannot contain occurrences of $(\mathit{fst}_i\ l)$. We derive this in $FO\lambda^{\Delta\mathbb{N}}$ by applying the second formula above to the hypothesis $\mathit{element}\ (\lambda l\, \mathit{typeof}\ (\mathit{fst}_i\ l)\ (u'\, l))\ (\lambda l\, il\, (\mathit{rst}\ l))$, yielding

$$\ldots, \exists a'((\lambda l\, \mathit{typeof}\ (\mathit{fst}_i\ l)\ (u'\, l)) \equiv_{atm^*} (\lambda l\, a'\, (\mathit{rst}\ l)) \wedge \mathit{element}\ a'\ il) \longrightarrow (\lambda l\, u(\mathit{rst}\ l)) \equiv_{i^*} u'\ .$$

Applying $\exists\mathcal{L}$ and $\wedge\mathcal{L}$ to this, and then applying $\mathit{def}\mathcal{L}$ to $\lambda l\, (\mathit{typeof}\ (\mathit{fst}_i\ l)\ (u'\, l)) \equiv_{atm^*} (\lambda l\, a'\, (\mathit{rst}\ l))$ will complete the derivation. Having shown that the induction hypothesis applies, we can now conclude that $t'_1 \equiv_{i^*} t'_2$, and hence $(\mathit{arr}^*\ u\ t'_1) \equiv_{i^*} (\mathit{arr}^*\ u\ t'_2)$. ∎

6.5 Related Work

There are several approaches others have taken to reason about higher-order abstract syntax encodings directly in a formalized meta-language. Despeyroux, Felty, and Hirschowitz [9, 8] show that induction principles for a restricted form of second-order abstract syntax can be derived in the Coq proof development system. To keep the definitions monotone, they

introduce a separate type for variables and explicit coercions from variables to other types. For example, their constructors for λ-terms would be

$$\mathit{var} \;:\; \mathit{vr} \to \mathit{tm} \qquad \mathit{abs} \;:\; (\mathit{vr} \to \mathit{tm}) \to \mathit{tm} \qquad \mathit{app} \;:\; \mathit{tm} \to \mathit{tm} \to \mathit{tm} \;,$$

and the corresponding definition of *typeof* would be

$$\mathit{typeof}_{\mathit{vr}} \;:\; \mathit{vr} \to \mathit{ty} \to o \qquad \mathit{typeof} \;:\; \mathit{tm} \to \mathit{ty} \to o$$

$$\begin{aligned}
\mathit{typeof}\ (\mathit{var}\ X)\ T &\triangleq \mathit{typeof}_{\mathit{vr}}\ X\ T \\
\mathit{typeof}\ (\mathit{abs}\ M)\ (\mathit{arr}\ T\ U) &\triangleq \forall x(\mathit{typeof}_{\mathit{vr}}\ x\ T \supset \mathit{typeof}\ (M\ x)\ U) \\
\mathit{typeof}\ (\mathit{app}\ M\ N)\ T &\triangleq \exists u(\mathit{typeof}\ M\ (\mathit{arr}\ u\ T) \wedge \mathit{typeof}\ N\ u)\ .
\end{aligned}$$

This is similar to our use of the two predicates *hyp* and *conc* in our encoding of intuitionistic logic in Section 5.1.2. Notice that the type *tm* does not occur negatively in the type of any of its constructors, nor does the predicate *typeof* occur negatively in its definition. This allows Coq to automatically construct induction principles for *tm* and *typeof*. Since object-level variable binding is still represented by meta-level λ-abstraction, the object language still inherits α-equivalence from the meta-language. Because the abstraction is over the type *vr*, however, meta-level β-reduction cannot be used for substitution. These approaches also lessen the power of the meta-level cut as a reasoning tool. Suppose that $\forall x(\mathit{typeof}_{\mathit{vr}}\ x\ T \supset \mathit{typeof}\ (M\,x)\ U)$ and $\mathit{typeof}\ N\ T$ are derivable. In contrast to our encoding, it is not immediate that substituting N for $(\mathit{var}\,x)$ in $(M\,x)$ yields a term M' such that $\mathit{typeof}\ M'\ U$ is derivable. Thus of the three key benefits to higher-order abstract syntax, they only retain α-conversion. In addition, the Coq type $(\mathit{vr} \to \mathit{tm})$ includes functions besides those expressible as λ-terms, so the type *tm* includes expressions that do not encode terms of the object language. They avoid these *exotic* terms through the definition and use of a validation predicate.

Despeyroux, Pfenning, and Schürmann [10] address the problem of exotic terms by using a modal operator to distinguish the types of parametric functions (expressible as λ-terms) from the types of arbitrary functions. As a result, their calculus allows primitive recursive functionals while preserving the adequacy of higher-order abstract syntax encodings. This represents a start toward a logical framework supporting meta-theoretic

reasoning, higher-order abstract syntax, and the judgments-as-types principle. In such a framework a derivation would be represented as a function whose type is the derived property. Thus the $\rightarrow$ type constructor must be rich enough to include the mappings from derivations to derivations such as the realizations of case analysis and induction. Their work is orthogonal to our work presented in this paper. We are not attempting to support the judgments-as-types principle, so the types of our meta-logic are only used to encode syntactic structure. Thus we can restrict these types to include only λ-terms, ensuring the adequacy of encodings in higher-order abstract syntax. They, on the other hand, do not address the issue of induction principles for higher-order abstract syntax, or more generally, the issue of formal reasoning about higher-order abstract syntax encodings.

Schürmann [51] offers another framework supporting higher-order abstract syntax and meta-theoretic analysis. He constructs a meta-logic MLF to reason about deductive systems represented in the Horn fragment of LF. This meta-logic includes a recursion rule that is used for induction and case analysis. This approach is similar in spirit to ours in that there are three levels: the deductive system(s) under consideration, the logic in which the deductive systems are encoded, and the logic in which meta-theoretic analysis takes place. His meta-logic MLF, however, is designed for a specific, fixed intermediate logic, the Horn fragment of LF. In our case, the meta-logic is a general framework capable of representing and reasoning about a variety of logics. In addition, the validity of Schürmann's work depends on cut-elimination for MLF, which is still an open question.

Still another strategy for meta-theoretic reasoning about higher-order abstract syntax encodings is to perform each case of a proof in the meta-logic, but verify the completeness of the proof outside the logical framework. Rohwedder and Pfenning [47, 48] investigate the design and implementation of such external validity conditions.

Matthews seeks to reconcile the advantages of LF-style encodings with the facilities for meta-theoretic analysis found in theories of inductive definitions [29]. His approach has some similarity to our own, in that he creates a three-level hierarchy, with each level being encoded in the previous. As in our approach, his top level contains a definition facility and induction principles for reasoning about encodings at the next level. However, his logic at the intermediate level contains only an implication connective and no quantifiers.

Thus he does not address the treatment of object-level bound variables, a major feature of higher-order abstract syntax and, consequently, of our work.

Chapter 7

Conclusion and Future Work

7.1 Summary of Accomplishments

In this dissertation we presented a logic $FO\lambda^{\Delta\mathbb{N}}$ with definitions and natural number induction. Induction is a fundamental tool for reasoning about formal systems. The notion of definition supported by $FO\lambda^{\Delta\mathbb{N}}$ extends the concept of a theory to include the sense that there are no other ways that defined concepts can be established. This sense of closure increases the logic's capabilities for both expression and reasoning. In addition, the term language of $FO\lambda^{\Delta\mathbb{N}}$ is the simply-typed λ-calculus, which is appropriate for the use of higher-order abstract syntax. We have proved cut-elimination and consistency theorems for $FO\lambda^{\Delta\mathbb{N}}$. The usual cut-elimination proofs for logics with definitions do not apply to $FO\lambda^{\Delta\mathbb{N}}$ because of the presence of the induction rule; we instead used an extension of a technique due to Tait and Martin-Löf.

We demonstrated the expressive power of our logic in the realm of abstract transition systems. The sense of closure in $FO\lambda^{\Delta\mathbb{N}}$'s definitions made it possible to encode both may and must behavior of systems, and thus to express the notions of simulation and bisimulation. By also using induction over natural numbers, we were able to capture the greatest fixed-point of these notions, and thus the largest simulation and bisimulation relations, and to prove some high-level properties about them.

Finally, we developed a framework for formal reasoning about systems expressed in higher-order abstract syntax, avoiding the apparent tradeoff between the benefits of this

representation technique and the ability to perform meta-theoretic analyses of encodings. We demonstrated this framework on encodings of three programming languages encompassing both functional and imperative paradigms. A number of significant theorems about these languages were derived in this framework, including unicity of typing and subject reduction. The flexibility of the framework was also shown through the use of intuitionistic and linear specification logics.

7.2 Future Work

There are many ways in which this work could be continued, both in extensions to the logic and in additional applications of it.

An obvious extension of $FO\lambda^{\Delta\mathbb{N}}$ would be to replace the induction rule for natural numbers with induction over arbitrary definitions. Coq and FS_0, for example, both support induction for definitions in this manner. Care must be taken here to ensure that the cut-elimination and consistency results extend to the extended logic.

Another interesting extension to consider is enriching the class of definitions to include the form of typical higher-order abstract syntax encodings. This would involve relaxing the level restriction, and it is not clear if this can be done without losing the cut-elimination and consistency properties. This also would cause the logic to support definitions outside the class of monotone inductive definitions, so the existence of induction principles for these definitions would be in question. With such an extension, however, it might become possible to both use higher-order abstract syntax and reason effectively about the encodings in the same logic. The presence of case analysis and induction in the logic make its implication connective stronger than that of intuitionistic logic, so a modal operator along the lines of [10] might become necessary to maintain the adequacy of encodings.

The implementation of the logic could also be improved. For this dissertation, the Pi derivation editor for the finitary calculus of partial inductive definitions was used. To ensure that the constructed derivations were valid for $FO\lambda^{\Delta\mathbb{N}}$, the induction rule was only used in a restricted manner, definitions for additional connectives were constructed, and all definitions were checked by hand to ensure that they satisfied the $FO\lambda^{\Delta\mathbb{N}}$ level restrictions. An implementation of $FO\lambda^{\Delta\mathbb{N}}$ would remove this burden from the user and

further increase confidence in the derivations. Improvements to Pi itself are also possible; our use showed the need for increased robustness, more powerful unification, and the ability to import other derivations as lemmas or derived rules. In addition, a graphical display showing the shape of the entire derivation and allowing movement within the derivation would be helpful, and some automatic theorem proving capabilities would also be useful.

In terms of applications of the logic, an interesting extension of the work presented here would be to explore the encoding of simulation and bisimulation for languages with significant variable-binding constructs. For example, Howe [26] explores a bisimulation relation for functional programming languages and uses a syntactic technique to prove that it is a congruence. It would be interesting to see if this could be done formally within the framework presented in this dissertation. The logic could also be used to formally establish equivalences among specific programs; Chirimar [6] used linear logic specifications to informally prove various equivalences suggested by Meyer and Sieber [33].

Additional work in analysis of programming languages along the lines of Chapter 6 could also be done. Time precluded us from proving the determinacy of evaluation for $\text{PCF}_{:=}$, for example, and a transition semantics for the language could be constructed and shown to be equivalent to the natural semantics we constructed. It would also be interesting to formalize other analyses; Hannan and Miller, for example, construct abstract machines from operational semantics by applying of a series of transformations and argue informally that the transformations preserve correctness [22]. Richer languages could also be considered, including features such as concurrency, exceptions, polymorphism, etc.

Finally, alternatives to the explicit eigenvariable encoding of Section 5.1.4 could be explored. Although this encoding supports the higher-order abstract syntax representation of bound variables and allows substantial meta-theoretic analysis, it does have some drawbacks. The pervasive presence of the *evs* parameter representing the free variable list is somewhat cumbersome, and numerous lemmas must be proved to show that various properties are preserved by extensions of this list or substitution for free variables. The obvious alternative, a de Bruin-style encoding of free variables, would require a similar amount of work and would not support the higher-order abstract syntax representation for bound variables. It is important to point out that this issue relates to the encoding of the

specification logic, not the object systems, of our framework. Thus these lemmas need to be proved only once for any specification logic, *not* for every object system, and so the representational advantage of higher-order abstract syntax for the object systems is not lost.

Bibliography

[1] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[2] K.R. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[3] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309 – 354, 1992.

[4] R. Burstall and F. Honsell. A natural deduction treatment of operational semantics. In *Proceedings of the 8th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 338 of *Lecture Notes in Computer Science*, pages 250–269. Springer-Verlag, 1988.

[5] Iliano Cervesato and Frank Pfenning. A linear logic framework. In *Proceedings, Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275. IEEE Computer Society Press, July 1996.

[6] Jawahar Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, February 1995.

[7] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[8] Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, April 1995.

[9] Joëlle Despeyroux and Andre Hirschowitz. Higher-order abstract syntax with induction in Coq. In F. Pfenning, editor, *Proceedings of the Fifth International Conference on Logic Programming and Automated Reasoning*, volume 822 of *Lecture Notes in Artificial Intelligence*, pages 159–173. Springer-Verlag, June 1994.

[10] Joelle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Third International Conference on Typed Lambda Calculi and Applications*, April 1997.

[11] Lars-Henrik Eriksson. A finitary version of the calculus of partial inductive definitions. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions to Logic Programming*, volume 596 of *Lecture Notes in Artificial Intelligence*, pages 89–134. Springer-Verlag, 1991.

[12] Lars-Henrik Eriksson. Finitary partial inductive definitions as a general logic. In *Proceedings of the Fourth International Workshop on Extensions to Logic Programming*, volume 798 of *Lecture Notes in Artificial Intelligence*, pages 94–119. Springer-Verlag, 1993.

[13] Lars-Henrik Eriksson. Pi: an interactive derivation editor for the calculus of partial inductive definitions. In A. Bundy, editor, *Proceedings of the Twelfth International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 821–825. Springer-Verlag, June 1994.

[14] Vijay Gehlot and Carl Gunter. Normal process representatives. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 200–207. IEEE Computer Society Press, June 1990.

[15] 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., 1969.

[16] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'émination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92, Amsterdam, 1971. North-Holland.

[17] Jean-Yves Girard. A fixpoint theorem in linear logic. A message posted on the `linear@cs.stanford.edu` mailing list, `http://www.csl.sri.com/linear/mailing-list-traffic/www/07/mail_3.html`, February 1992.

[18] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[19] Lars Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87:115–142, 1991.

[20] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. II. Programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, 1991.

[21] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[22] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[23] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. PhD thesis, University of Pennsylvania, August 1990.

[24] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[25] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[26] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[27] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[28] Per Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*,

volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 179–216. North-Holland, 1971.

[29] Seán Matthews. A practical implementation of simple consequence relations using inductive definitions. In W. McCune, editor, *Proceedings of the 14th Conference on Automated Deduction*. Springer-Verlag, July 1997.

[30] Sean Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In G. Huet and G.D. Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993.

[31] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1997.

[32] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus: Preliminary report. In *Proceedings of the 1996 Workshop on Linear Logic*, volume 3 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[33] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Proceedings of the* 15^{th} *Annual ACM Symposium on the Principles of Programming Languages*, pages 191–203, 1988.

[34] Dale Miller. Abstractions in logic programs. In P. Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[35] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[36] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, volume 660 of *Lecture Notes in Computer Science*, pages 242–265. Springer-Verlag, 1993.

[37] Dale Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165:201–232, 1996.

[38] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In S. Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, September 1987.

[39] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[40] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[41] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1995.

[42] D. M. Park. Concurrency on automata and infinite sequences. In P. Deussen, editor, *Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[43] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, 1993.

[44] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings,Fourth Annual Symposium on Logic in Computer Science*, pages 313–321. IEEE Computer Society Press, June 1989.

[45] Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 156–166. IEEE Computer Society Press, June 1995.

[46] 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, June 1988.

[47] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 537–551. Springer-Verlag, June 1992.

[48] Ekkehard Rohwedder and Frank Pfenning. Mode and termination analysis for higher-order logic programs. In *Proceedings of the European Symposium on Programming*, pages 296–310, April 1996.

[49] Peter Schroeder-Heister. Cut-elimination in logics with definitional reflection. In D. Pearce and H. Wansing, editors, *Nonclassical Logics and Information Processing*, volume 619 of *Lecture Notes in Computer Science*, pages 146–171. Springer-Verlag, 1992.

[50] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, June 1993.

[51] Carsten Schürmann. A computational meta logic for the Horn fragment of LF. Master's thesis, Carnegie Mellon University, December 1995.

[52] Dana S. Scott. A type theoretical alternative to CUCH,ISWIM,OWHY. Unpublished manuscript, 1969.

[53] John Slaney. Solution to a problem of Ono and Komori. *Journal of Philosophic Logic*, 18:103–111, 1989.

[54] W.W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.