Reasoning with Higher-Order Abstract Syntax in a Logical Framework

RAYMOND C. MCDOWELL
Kalamazoo College
and
DALE A. MILLER
Pennsylvania State University

Logical frameworks based on intuitionistic or linear logics with higher-type quantification have been successfully used to give high-level, modular, and formal specifications of many important judgments in the area of programming languages and inference systems. Given such specifications, it is natural to consider proving properties about the specified systems in the framework: for example, given the specification of evaluation for a functional programming language, prove that the language is deterministic or that evaluation preserves types. One challenge in developing a framework for such reasoning is that *higher-order abstract syntax* (HOAS), an elegant and declarative treatment of object-level abstraction and substitution, is difficult to treat in proofs involving induction. In this paper, we present a meta-logic that can be used to reason about judgments coded using HOAS; this meta-logic is an extension of a simple intuitionistic logic that admits higher-order quantification over simply typed λ-terms (key ingredients for HOAS) as well as 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 explore the difficulties of formal meta-theoretic analysis of HOAS encodings 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.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification Techniques*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Mechanical Theorem Proving*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: definitions, higher-order abstract syntax, induction, logical frameworks

Authors' addresses: Raymond C. McDowell, Department of Mathematics and Computer Science, Kalamazoo College, 1200 Academy Street, Kalamazoo, MI 49006-3295 USA. Dale A. Miller, Department of Computer Science and Engineering, 220 Pond Laboratory, The Pennsylvania State University, University Park, PA 16802-6106 USA.
The authors have been funded in part by the grants ONR N00014-93-1-1324, NSF CCR-92-09224, NSF CCR-94-00907, NSF CCR-98-03971, and ARO DAAH04-95-1-0092.

INTRODUCTION

Meta-logics and type systems have been used to specify the semantics of a wide range of logics and computation systems [Avron et al. 1992; Chirimar 1995; Felty 1993; Pfenning and Rohwedder 1992]. This is done by making judgments, such as "the term M denotes a program," "the program M evaluates to the value V", and "the program M has type T", into predicates that can be proved or types for which inhabitants (proofs) are needed. Since these specification languages often contain quantification at higher-order types and term structures involving λ-terms, succinct and elegant specifications can be written using *higher-order abstract syntax*, a high-level and declarative treatment of object-level bound variables and object-level substitution [Miller and Nadathur 1987; Pfenning and Elliot 1988]. In other approaches to syntactic representation where bound variables are managed directly using either names or deBruijn-style numbering, these details must be carefully addressed and dealt with at most levels of a specification.

Recently, logical specification languages have been used to not only describe how to *perform* computations but also describe *properties about* the encoded computations [Basin and Constable 1993; Magnusson and Nordström 1994; Matthews et al. 1993; VanInwegen 1996]. By proving these properties in a formal framework, we can benefit from automated proof assistance and gain greater confidence in our results. However, this work has been done in languages that do not support higher-order abstract syntax and so has not been able to benefit from this representation technique. As a result, theorems about substitution and bound variables can dominate the task [VanInwegen 1996]. But meta-theoretic reasoning about systems represented in higher-order abstract syntax has been difficult since the languages and logics that support this notion of syntax do not provide facilities for the fundamental operations of case analysis and induction. Moreover, higher-order abstract syntax leads to types and recursive definitions that do not give rise to monotone inductive operators, making inductive principles difficult to find.

These apparent difficulties can be overcome, and in this paper we present a meta-logic in which we can naturally reason about specifications in higher-order abstract syntax. This meta-logic is a higher-order intuitionistic logic with partial inductive definitions and natural number induction. Induction on natural numbers allows us to derive other induction principles via the construction of an appropriate measure. A partial inductive definition [Hallnäs 1991] is a proof-theoretic formalization that allows certain theories to be treated as "closed" or as defining fixed points. This allows us to perform case analyses on the defined judgments. We use this definition mechanism to specify a small, object-level logic which in turn is used to specify the computation systems under consideration. In this way, we can talk directly about the structure of object-logic sequents and their provability. This technique of representing a logic within a logic is not new (see, for example, Felty and Miller [1988] and Paulson [1986] for some early references) and corresponds to the structure of common informal reasoning.

The first part of this paper (Sections 1 and 2) presents the meta-logic $FO\lambda^{\Delta\mathbb{N}}$

(pronounced "fold-n"). To illustrate the use of $FO\lambda^{\Delta\mathbb{IN}}$, we derive several theorems expressing properties of natural numbers and lists. In Part II (Sections 4, 5, and 6) we consider encodings of intuitionistic and linear logics in $FO\lambda^{\Delta\mathbb{IN}}$ 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 Part III (Sections 7, 8, 9, and 10) 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 $FO\lambda^{\Delta\mathbb{IN}}$ in which we perform the reasoning; this specification logic is itself specified in $FO\lambda^{\Delta\mathbb{IN}}$. 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 $FO\lambda^{\Delta\mathbb{IN}}$ such properties as the unicity of typing, determinacy of semantics, and type preservation (subject reduction). We conclude in Section 11 with a brief discussion of our accomplishments and possible extensions of this work.

Part I: THE META-LOGIC $FO\lambda^{\Delta\mathbb{IN}}$

In this part we introduce the logic which we call $FO\lambda^{\Delta\mathbb{IN}}$, 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. We conclude the part by briefly comparing the strength of $FO\lambda^{\Delta\mathbb{IN}}$ with that of other logical systems.

1. A DESCRIPTION OF THE LOGIC

The basic logic is an intuitionistic version of a subset of Church's Simple Theory of Types [Church 1940] in which formulas have 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 $FO\lambda^{\Delta\mathbb{IN}}$ 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 I. 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.

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 are

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

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

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

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

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

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

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

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

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

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

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

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 $(\bigwedge \Gamma) \supset C$. Notice that the first two premises of the $nat\mathcal{L}$ rule involve no assumptions other than the induction hypothesis (in the second premise). This is not a restriction on induction since one can choose to do induction on, say, $\lambda w.(\bigwedge \Gamma) \supset B\, w$, which would effectively provide the first two premises with the assumptions from the multiset Γ.

A *definitional clause* is written $\forall \bar{x}[p\, \bar{t} \triangleq B]$, 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 $\triangleq$ 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 [Schroeder-Heister 1992]. 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. We then extend the notion of level to formulas and derivations. 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 inference rules for defined atoms are given relative to some fixed definition. The right-introduction rule for defined atoms is

$$\frac{\Gamma \longrightarrow B\theta}{\Gamma \longrightarrow p\,\bar{u}}\ \mathit{def}\mathcal{R}, \text{ where } p\,\bar{u} = (p\,\bar{t})\theta \text{ for some clause } \forall\bar{x}.[p\,\bar{t} \triangleq B]\ ,$$

where θ is a substitution of terms for variables. The left rule for defined concepts uses complete sets of unifiers (CSU):

$$\frac{\left\{B\theta, \Gamma\theta \longrightarrow C\theta \mid \theta \in CSU(p\,\bar{u}, p\,\bar{t}) \text{ for some clause } \forall\bar{x}.[p\,\bar{t} \triangleq B]\right\}}{p\,\bar{u}, \Gamma \longrightarrow C}\ \mathit{def}\mathcal{L}\ ,$$

where θ is a substitution of terms for variables, and the variables $\bar{x}$ are chosen to be distinct from the variables free in the lower sequent of the rule. (A set S of unifiers of t and u is *complete* if for every unifier ρ of t and u there is a unifier $\theta \in S$ such that ρ is $\theta \circ \sigma$ for some substitution σ [Huet 1975].) 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* [Schroeder-Heister 1993] (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 [Girard 1992]. This particular presentation of the rule is due to Eriksson [Eriksson 1991]. Notice that in the $\mathit{def}\mathcal{L}$ rule, the free variables of the conclusion can be instantiated in the premises.

The number of premises of the $\mathit{def}\mathcal{L}$ rule may be either infinite or finite (including zero). If the formula $p\,\bar{u}$ does not unify with the head of any definitional clause, then the number of premises will be zero. In this case $p\,\bar{u}$ is an unprovable formula logically equivalent to $\bot$, and $\mathit{def}\mathcal{L}$ corresponds to the $\bot\mathcal{L}$ rule. If the formula $p\,\bar{u}$ does unify with the head of a definitional clause, CSUs may be infinite, as is the case with unifications involving simply typed λ-terms and variables of functional type (a.k.a. higher-order unification). Clearly an inference rule with an infinite number of premises is impossible to automate directly. There are many important situations where CSUs are not only finite but are also singleton (containing a most general unifier) whenever terms are unifiable. One such case is, of course, the first-order case. Another case is when the application of functional variables are restricted to distinct bound variables in the sense of *higher-order pattern* unification [Miller 1991]. In this paper, all unification problems will fall into this latter case and,

hence, we can count on the definition left-introduction rule to have a finite (and small) number of premises.

Assuming that a definition is given and fixed, we have the following results.

PROPOSITION 1.1 CUT-ELIMINATION FOR $FO\lambda^{\Delta\mathbb{N}}$. *If a sequent is derivable in $FO\lambda^{\Delta\mathbb{N}}$, then it is derivable without using the cut rule.*

PROOF. The proofs of Schroeder-Heister [1993] regarding cut-elimination for definitions do not appear to extend to our setting where induction is included. A complete proof of this theorem appears in McDowell [1997] and McDowell and Miller [2000] and is modeled on proofs by Tait and Martin-Löf that use the technical notions of normalizability and reducibility. □

The following corollary is an immediate consequence of this cut-elimination theorem.

COROLLARY 1.2 CONSISTENCY OF $FO\lambda^{\Delta\mathbb{N}}$. *There is no derivation in $FO\lambda^{\Delta\mathbb{N}}$ of the sequent $\longrightarrow \bot$.*

Although 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 induction predicate B.

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

Table II. Definitional clauses for predicates over natural numbers

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

continue in this manner with the rule premises. Thus unproved premises represent statements of 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.

2.1 Natural Numbers

As described in Section 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:

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

The definitional clauses for these predicates are shown in Table II; 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 2.1. *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 represented 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 $FO\lambda^{\Delta\mathbb{N}}$ 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 2.2. *The formula* $\forall i(nat\ i \supset z \leq i)$ *is derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *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 2.1, 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:

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

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

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

□

It is also possible to derive in $FO\lambda^{\Delta\mathbb{IN}}$ a rule for complete induction over the natural numbers [McDowell 1997].

PROPOSITION 2.3 COMPLETE INDUCTION. *For any formula $C : o$, predicate $B : nt \to o$, term $I : nt$, multiset Γ of formulas, and eigenvariable $j : nt$ such that j is not free in B, the following rule is derivable in $FO\lambda^{\Delta\mathbb{IN}}$ using the definition $\mathcal{D}(nat)$:*

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

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

PROPOSITION 2.4. *The following formulas are derivable in $FO\lambda^{\Delta\mathbb{IN}}$ using the definition $\mathcal{D}(nat)$:*

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

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

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

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

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

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

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

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

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

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

2.2 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

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

whose definitional clauses are shown in Table III; 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 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 ($nat\mathcal{L}$) using the length of a list as our measure.

PROPOSITION 2.5. *For any formula $C : o$, predicate $B : lst \to o$, term $L : lst$, multiset Γ of formulas, and eigenvariables $x : \tau$ and $l : lst$ such that x and l are not free in B, the following rule is derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition $\mathcal{D}(list(\tau))$:*

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

PROOF. To derive this rule, we construct a partial derivation of the sequent $list\ L, \Gamma \longrightarrow C$, leaving unproved premises of the form $\longrightarrow B\ nil$, $B\ l \longrightarrow B\ (x :: l)$, 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 $(x' :: l')$;

Table III. Definitional clauses for predicates over lists

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

—$B\ L$ 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:

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

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 $\mathit{nat}\mathcal{L}$ rule with the induction predicate $\lambda i.\forall l(\mathit{length}\ l\ i \supset B\ l)$, which yields the three sequents

$$\longrightarrow \forall l(\mathit{length}\ l\ z \supset B\ l)$$

$$\forall l(\mathit{length}\ l\ i' \supset B\ l) \longrightarrow \forall l(\mathit{length}\ l\ (s\ i') \supset B\ l)$$

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

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:

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

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:

$$\dfrac{B\,L, \mathit{length}\ L\ i \longrightarrow B\,L \quad B\,L, \Gamma \longrightarrow C}{B\,L, \mathit{length}\ L\ i, \Gamma \longrightarrow C}\ \mathit{cut}\ .$$

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 $nat\mathcal{L}$ rule:

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

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 is represented in the partial derivation of the second premise of the $nat\mathcal{L}$ rule:

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

In this use of the $\mathit{def}\mathcal{L}$ rule, the complete set of unifiers for the atomic formula $\mathit{length}\ l\ (s\ i')$ and the head of the clause $\forall x', l', j[\mathit{length}\ (x' :: l')\ (s\ j) \triangleq \mathit{length}\ l'\ j]$ is the singleton set $\{[x' :: l'/l, i'/j]\}$. The unproved premise of the partial derivation above 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:

$$\dfrac{B\,l', \mathit{length}\ l'\ i' \longrightarrow B\,l' \quad B\,l' \longrightarrow B\,(x' :: l')}{B\,l', \mathit{length}\ l'\ i' \longrightarrow B\,(x' :: l')}\ \mathit{cut}\ .$$

□

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 2.6. *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 2.5 with the induction predicate $(\lambda l.\mathit{split}\ l\ \mathit{nil}\ l)$, we get the three sequents

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

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

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

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*:

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

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

PROPOSITION 2.7. *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(\mathit{list}(\tau))$*:*

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

$$\forall l_1(\mathit{list}\ l_1 \supset \forall l_2(\mathit{list}\ l_2 \supset \forall l(\mathit{split}\ l\ l_1\ l_2 \supset \mathit{list}\ l)))$$

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

$$\forall l(\mathit{list}\ l \supset \forall l_1 \forall l_2(\mathit{split}\ l\ l_1\ l_2 \supset \mathit{split}\ l\ l_2\ l_1))$$

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

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

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

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

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

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

3. THE STRENGTH OF $FO\lambda^{\Delta\mathbb{N}}$

Before proceeding to consider $FO\lambda^{\Delta\mathbb{N}}$ as a logic for meta-theoretic analysis, we comment here on how to relate $FO\lambda^{\Delta\mathbb{N}}$ to other logical systems.

First, we show that $FO\lambda^{\Delta\mathbb{N}}$ captures the theorems of an intuitionistic version of Peano's arithmetic (IPA) using a definition consisting of one clause for equality. The formulas of IPA are those of a first-order logic with equality using the same logical connectives as those in $FO\lambda^{\Delta\mathbb{N}}$ and the same symbols z for zero and s for successor. The axiom schemes for IPA can be grouped into the following collections.

(1) Axioms for first-order intuitionistic logic.

(2) Axioms for equality: reflexivity, symmetry, transitivity, and substitution.

(3) The two formulas

$$\forall x \forall y (s\, x = s\, y \supset x = y) \quad \text{and} \quad \forall x (z = s\, x \supset \bot) \ .$$

(4) The axioms of induction: all formulas of the form

$$\varphi(z) \wedge \forall j (\varphi(j) \supset \varphi(s\, j)) \supset \forall x \varphi(x) \ ,$$

where $\varphi(x)$ ranges over formulas with at most the variable x free.

There are two inference rules for IPA: *Modus Ponens* allows the formula B to be inferred from the formulas $A \supset B$ and A, while *Universal Generalization* allows the formula $\forall x B$ to be inferred from B. A list of formulas $C_1, \ldots, C_n$ $(n \geq 1)$ is an *IPA derivation* if for every $i \in \{1, \ldots, n\}$, C_i is either an axiom or is the conclusion of modus ponens or universal generalization from formulas in the list $C_1, \ldots, C_{i-1}$. We write $\vdash_{ipa} C$ if C is the last formula of an IPA derivation.

In order to map an IPA formula, say B, to a $FO\lambda^{\Delta\mathbb{N}}$ formula, say $(B)^\circ$, we must adjust for typing. The single sort used in IPA formulas will be mapped to the type nt, and all instances of quantifiers in IPA formulas must be qualified using the nat predicate: that is, $(\forall x.B)^\circ = \forall x.nat\ x \supset (B)^\circ$ and $(\exists x.B)^\circ = \exists x.nat\ x \wedge (B)^\circ$. Predicates in IPA will be mapped to the corresponding predicates in $FO\lambda^{\Delta\mathbb{N}}$ similarly adjusted for type. Let $\mathcal{D}(eq)$ be the definition consisting of the one clause:

$$I = I \triangleq \top \ .$$

We now sketch a proof that $\vdash_{ipa} C$ implies that $(C)^\circ$ has a $FO\lambda^{\Delta\mathbb{N}}$ derivation using $\mathcal{D}(eq)$. The proof is by induction on the length of IPA derivation. The axioms of intuitionistic logic are derivable in $FO\lambda^{\Delta\mathbb{N}}$ since it is complete for intuitionistic logic (the rules for definition and natural numbers are not needed). The axioms for equality are derivable using the definition rules with $\mathcal{D}(eq)$ (as noted in Girard [1992] and Schroeder-Heister [1993]). The two formulas concerning z and s are also derivable using the definition rules. The only remaining axiom that needs to be considered is that for induction in IPA. Let $\phi(x)$ be a formula with at most x free and let $\phi^\circ(x)$ be the translation of that formula into $FO\lambda^{\Delta\mathbb{N}}$. We then need to prove that the sequent

$$\longrightarrow \phi^\circ(z) \wedge \forall j (nat\ j \supset \phi^\circ(j) \supset \phi^\circ(s\, j)) \supset \forall x (nat\ x \supset \phi^\circ(x))$$

is derivable in $FO\lambda^{\Delta\mathbb{N}}$. Using the inference rules $\supset\mathcal{R}$, $\forall\mathcal{R}$, and $c\mathcal{L}$, the derivability of this sequent can be reduced to the derivability of the sequent

$$\phi^\circ(z) \wedge \forall j(nat\ j \supset \phi^\circ(j) \supset \phi^\circ(s\ j)), nat\ I, nat\ I \longrightarrow \phi^\circ(I)\ ,$$

where I is a new eigenvariable. Consider now deriving this sequent with $nat\mathcal{L}$, using the induction predicate

$$\lambda w.\ (\phi^\circ(z) \wedge \forall j(nat\ j \supset \phi^\circ(j) \supset \phi^\circ(s\ j)) \wedge nat\ w) \supset \phi^\circ(w)\ .$$

The three premises of this instance of $nat\mathcal{L}$ are now easily derived.

Second, it may be possible to base the logic $FO\lambda^{\Delta\mathbb{N}}$ on classical instead of intuitionistic logic. Since $FO\lambda^{\Delta\mathbb{N}}$ is intended to formalize informal mathematical reasoning about computation, such a choice might well be interesting and useful, although none of the many example applications we have explored require leaving intuitionistic logic. We do not explore a classical version of $FO\lambda^{\Delta\mathbb{N}}$ here and simply point out that if the classical variant satisfies a cut-elimination property, a proof of that fact does not seem to be a straightforward generalization of the proof given in McDowell [1997] and McDowell and Miller [2000].

Finally, we add a word about how $FO\lambda^{\Delta\mathbb{N}}$ can be used to reason about computation. Subsets of intuitionistic logic, such as *herditary Harrop formulas* or *Horn clauses* can be used to specify computation using goal-directed derivation search [Miller et al. 1991]. The logic $FO\lambda^{\Delta\mathbb{N}}$, which is much stronger than these subsets, can be used to reason about logic programs in the following fashion. Let $\mathcal{P}$ be, for example, a Horn clause program and let G be some goal formula (a formula composed of conjunctions, disjunctions, and existential quantifiers) such that there is goal-directed derivation of the sequent $\mathcal{P} \longrightarrow G$ in intuitionistic logic. That derivation is also a cut-free intuitionistic logic derivation [Miller et al. 1991]. Thus the sequent $\longrightarrow G$ has a cut-free derivation in $FO\lambda^{\Delta\mathbb{N}}$ using $\mathcal{P}$ as a definition (given the restrictions on G and $\mathcal{P}$, there are no occurrences of the $def\mathcal{L}$ and $nat\mathcal{L}$ inference rules in such a derivation). Now assume that we have also a derivation in $FO\lambda^{\Delta\mathbb{N}}$ using $\mathcal{P}$ as a definition of the sequent $G \longrightarrow G'$, for some goal formula G'. Using the cut-elimination theorem for $FO\lambda^{\Delta\mathbb{N}}$ (Proposition 1.1), we know that the sequent $\longrightarrow G'$ has a cut-free derivation in $FO\lambda^{\Delta\mathbb{N}}$ using $\mathcal{P}$ as a definition. Since induction is encoded as a left-introduction rule, it is easy to see that the resulting derivation does not contain occurrences of induction. Similarly, there can be no occurrences of the $def\mathcal{L}$ rule. Hence, we can conclude that $\mathcal{P} \longrightarrow G'$ will have an intuitionistic logic derivation as well as a goal-directed derivation. Thus, informally, we can conclude that if $G \supset G'$ is derivable in $FO\lambda^{\Delta\mathbb{N}}$ and there is a computation proving G, then there is computation proving G'. Hence, implications in the stronger logic can be used to show that the existence of certain computations can lead to the existence of other computations. For example, as we have mentioned in Proposition 2.7, the formula

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

can be derived in $FO\lambda^{\Delta\mathbb{N}}$ using $\mathcal{D}(list(\tau))$. If we also assume that we are given three lists L_0, L_1, L_2 such that $list\ L_0$ and $split\ L_0\ L_1\ L_2$ follow from $\mathcal{D}(list(\tau))$ (considered as a Horn clause logic program), then the above argument can be used to show that $split\ L_0\ L_2\ L_1$ must also follow from that logic program.

Part II: LOGIC REPRESENTATIONS FOR META-THEORETIC ANALYSIS

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 [1993] 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 part 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 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 Part III 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 the present part with a section discussing related work.

To keep our discussion succinct, we do not prove the adequacy of the encodings presented in Section 4. The skeptical reader is referred to the discussion of similar encodings in the literature: see Section 6 for references. The two encodings of Section 5, however, play a key role in our work, and so we do include adequacy theorems for these.

4. A SPECTRUM OF ENCODING STYLES

4.1 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. For brevity we will restrict our discussion here to a fragment of the logic containing implication and quantification. The full logic is considered in McDowell [1997], though the remaining connectives do not provide any additional insight. 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:

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

The constant $\langle\,\rangle$ coerces atoms into propositions: object-level predicates will be constants that build meta-level terms of type *atm*. The constant $\Rightarrow$ represents the implication connective 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.

Table IV. Natural deduction encoding of intuitionistic logic

$$
\begin{array}{rcl}
\mathit{prove}\,(B \Rightarrow C) & \subset & \mathit{prove}\,B \supset \mathit{prove}\,C \\
\mathit{prove}\,\bigwedge_i B & \subset & \forall_i x\ \mathit{prove}\,(B\,x) \\
\mathit{prove}\,\bigvee_i B & \subset & \exists_i x\ \mathit{prove}\,(B\,x) \\
 & & \\
\mathit{prove}\,C & \subset & \exists b(\mathit{prove}\,(b \Rightarrow C) \ \wedge\ \mathit{prove}\,b) \\
\mathit{prove}\,(B\,X) & \subset & \mathit{prove}\,\bigwedge_i B \\
\mathit{prove}\,C & \subset & \exists b(\mathit{prove}\,\bigvee_i b \ \wedge\ (\exists_i x\ \mathit{prove}\,(b\,x) \supset \mathit{prove}\,C))
\end{array}
$$

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 IV. Here we use $\subset$ for reverse implication in the meta-logic; the first clause, for example, can be rewritten as

$$(\mathit{prove}\,B \supset \mathit{prove}\,C) \supset \mathit{prove}\,(B \Rightarrow C)\ .$$

The first three clauses correspond to the introduction rules for natural deduction; the remaining three correspond to the elimination rules.

Although this encoding mirrors the rules for natural deduction, we may view it as an encoding of the sequent calculus, with the derivability of the sequent $B_1, \ldots, B_n \longrightarrow C$ represented by the $FO\lambda^{\Delta\mathbb{N}}$ formula

$$\mathit{prove}\,B_1 \supset \cdots \supset \mathit{prove}\,B_n \supset \mathit{prove}\,C\ .$$

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

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

We use double horizontal lines to represent multiple applications of an inference rule. In this case, both the $\supset\mathcal{R}$ rule and the $\forall\mathcal{R}$ rule are applied twice. The admissibility of the cut rule, encoded by the formula

$$\forall b \forall c((\mathit{prove}\,b \supset \mathit{prove}\,c) \supset \mathit{prove}\,b \supset \mathit{prove}\,c)\ ,$$

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

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

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

Table V. Sequent calculus encoding of intuitionistic logic

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

induction over the derivations. Recall that in Section 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; 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 $\triangleq$ 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.

4.2 Sequent calculus-style encoding

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

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

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 V. The first clause encodes the initial axiom, the next three correspond to the right introduction rules, and the remaining three correspond to the left introduction rules.

Since we have not changed the representation of quantification, we get α-equival-

ence 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 Pfenning [1995].

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 four sequents corresponding to the cases where the derivation of a ends with the initial rule or any of the three left rules:

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

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

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

Table VI. Explicit sequent encoding of intuitionistic logic

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

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

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 $\supset\mathcal{L}$ rule, which says that we know that the hypothesis $b \Rightarrow \langle a\rangle$ is available and so can conclude that a holds.

4.3 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, using *prp* for the 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 VI.

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 example with the encoding of Table VI, 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 four sequents, again corresponding to the cases where the derivation of a ends with the initial rule or any of the three left rules:

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

$$\begin{array}{l} \exists b' \exists c' (element\ (b' \Rightarrow c')\ ((b \Rightarrow \langle a \rangle) :: nil) \wedge \\ \qquad seq_{i'}\ (c' :: (b \Rightarrow \langle a \rangle) :: nil)\ \langle a \rangle \wedge \\ \qquad\quad seq_{i'}\ ((b \Rightarrow \langle a \rangle) :: nil)\ b') \longrightarrow \exists j\ seq_j\ ((b \Rightarrow \langle a \rangle) :: nil)\ b \end{array}$$

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

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

But this time we can easily eliminate three of the four 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:

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

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

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

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

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

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

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

Now we want to reason about the derivation of $\bigwedge_i y_1 \bigwedge_i y_2 \ldots$ 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

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

as well as three other sequents corresponding to the cases where the object logic derivation ends with the application of one of the left rules. Since these latter three 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

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

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

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

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

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

For $\langle p\, x_2\, t_3\rangle$ 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 $\langle p\, x_2\, t_3\rangle$ cannot be the other element of the list. This reasoning is represented formally by the $FO\lambda^{\Delta\mathbb{N}}$ derivation

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

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.

4.4 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 $\bigwedge$:

$$seq_{(S\ I)}\ L\ (\textstyle\bigwedge_i B) \triangleq \lambda x\ seq_I\ L\ (B\,x)\ .$$

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:

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

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:

$$seq_{(S\ I)}\ L\ (\lambda l \textstyle\bigwedge_i x(B\,l\,x)) \triangleq seq_I\ (\lambda l'\ L(rst\ l'))\ (\lambda l'\ B\,(rst\ l')\,(fst\ l'))\ .$$

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; $fst\ l'$ represents the new eigenvariable, and $rst\ l'$ represents the eigenvariables in l. The left rule for $\bigvee_i$ is similarly modified:

$$\begin{aligned} seq_{(S\ I)}\ L\ C \triangleq\ & \exists b(element\ (\lambda l \textstyle\bigvee_i x(b\,l\,x))\ L \wedge \\ & seq_I\ (\lambda l'\ (b\,(rst\ l')\,(fst\ l'))::(L\,(rst\ l')))\ (\lambda l'\ C\,(rst\ l'))\ . \end{aligned}$$

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:

$$element\ :\ (evs \to prp) \to (evs \to prplst) \to o\ .$$

Table VII 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

$$\begin{aligned} subst\ &:\ nt \to (evs \to i) \to (evs \to i) \to (evs \to i) \to o \\ subst_0\ &:\ nt \to (evs \to evs \to i) \to (evs \to evs \to i) \to (evs \to evs \to i) \to o\ . \end{aligned}$$

Table VII. Explicit eigenvariable encoding of intuitionistic logic

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

Table VIII. Encoding of substitution for eigenvariables

$$
\begin{array}{rcl}
subst\ I\ T_1\ T_2\ T_2' & \triangleq & subst_0\ I\ (\lambda l'\ T_1)\ (\lambda l'\ T_2)\ (\lambda l'\ T_2') \\
\\
subst_0\ z\ T_1\ (\lambda l' \lambda l\, T_2\, l'\,(fst\,l)\,(rst\,l))\ (\lambda l' \lambda l\, T_2\, l'\,(T_1\, l'\, l)\,(rst\,l)) & & \\
\quad \triangleq\ \top & & \\
subst_0\ (s\ I)\ (\lambda l' \lambda l\, T_1\, l'\,(fst\,l)\,(rst\,l)) & & \\
\quad (\lambda l' \lambda l\, T_2\, l'\,(fst\,l)\,(rst\,l))\ (\lambda l' \lambda l\, T_2'\, l'\,(fst\,l)\,(rst\,l)) & & \\
\quad \triangleq\ subst_0\ I\ (\lambda l' \lambda l\, T_1\,(rst\,l')\,(fst\,l')\,l) & & \\
\quad\quad (\lambda l' \lambda l\, T_2\,(rst\,l')\,(fst\,l')\,l)\ (\lambda l' \lambda l\, T_2'\,(rst\,l')\,(fst\,l')\,l)
\end{array}
$$

The judgement $subst\ i\ t_1\ t_2\ t_2'$ 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 $(fst\ (rst^i\ l))$ 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 $(rst^i\ l)$ for n applications of rst to l, *i.e.,* $(rst^0\ l)$ is l, $(rst^1\ l)$ is $(rst\,l)$, $(rst^2\ l)$ is $(rst\,(rst\,l))$, 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 VIII. 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.

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

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

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

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

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 $def\mathcal{L}$ twice, which results in the sequent

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

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 $def\mathcal{L}$ rule, yielding

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

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

$$element\ \lambda l\ \langle p\,(fst\ l)\, t_3\rangle\ \lambda l\ (\langle p\,(fst\ l)\, t_2\rangle :: \langle p\,(fst\ (rst\ l))\, t_1\rangle :: nil) \longrightarrow t_2 \equiv t_3\ .$$

If $\langle p\,(fst\ l)\, t_3\rangle$ 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 $\langle p\,(fst\ l)\, t_3\rangle$ 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

$$\dfrac{\dfrac{\dfrac{}{\top \longrightarrow \top}\ \top\mathcal{R}}{\top \longrightarrow t_2 \equiv t_2}\ def\mathcal{R} \qquad \dfrac{}{element\ \lambda l\ \langle p\,(fst\ l)\, t_3\rangle\ (\lambda l\ nil) \longrightarrow t_2 \equiv t_3}}{element\ \lambda l\ \langle p\,(fst\ l)\, t_3\rangle\ \lambda l\ (\langle p\,(fst\ l)\, t_2\rangle :: \langle p\,(fst\ (rst\ l))\, t_1\rangle :: nil) \longrightarrow t_2 \equiv t_3}\ ,$$

where the three inference rules without labels are $def\mathcal{L}$.

4.5 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 of 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 Part III 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. REPRESENTATION AND ANALYSIS OF LOGICS

In this section we illustrate the use of the some of the encoding techniques just presented. In Section 5.1 we use the explicit sequent technique of Section 4.3 to encode a fragment of intuitionistic logic; Section 5.2 discusses a fragment of linear logic encoded with the explicit eigenvariable technique of Section 4.4. 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.

5.1 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. Full intuitionistic logic could be used here instead, but its encoding is larger and that increase does not contribute to the set of examples that we wish to use here. The set of goal formulas can be encoded using the following constants:

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

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:

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

where *atmlst* is the same as the type *lst* introduced in Section 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 [Miller 1990; Miller et al. 1991]. In our setting, a derivation is uniform if every subderivation ending in a left rule is of the form

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

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 $seq_I\ L\ G'$ holds, where G' is the corresponding instantiation of G. We use the predicate

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

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

$$seq_{(S\ I)}\ L\ \langle A\rangle\ \triangleq\ \exists b (prog\ A\ b \wedge seq_I\ L\ b)\ ;$$

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 Part III. 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 Adequacy of Encoding Intuitionistic Logic. *Let* $\mathcal{D}(prog)$ *be the definition* $\{\forall \bar{x}_1 [prog\ A_1\ G_1 \triangleq \top], \ldots, \forall \bar{x}_n [prog\ A_n\ G_n \triangleq \top]\}$ *(*$n \geq 0$*) 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. In fact, we can show that a cut-free derivation of $\longrightarrow L \rhd B$ will consist only of sequents with empty antecedents [McDowell 1997]. Thus the $nat\mathcal{L}$ and $def\mathcal{L}$ rules are not used, since they both require a formula in the antecedent. □

The following theorem states that we can derive in $FO\lambda^{\Delta\mathbb{N}}$ that the specialization rule, the cut rule and the usual structural rules (exchange, weakening, and contraction) are admissible for our object logic.

Theorem 5.2 Admissibility of Rules for Intuitionistic Object Logic. *The following formulas are derivable in $FO\lambda^{\Delta\mathbb{N}}$ using the definition*

$$\mathcal{D}(nat) \cup \mathcal{D}(list(\mathrm{atm})) \cup \mathcal{D}(intuit) :$$

Specialization Rule:

$$\forall i \forall b \forall l(nat\ i\ \supset\ seq_{(S\,i)}\ l\ \bigwedge b\ \supset\ \forall x\, seq_i\ l\ (b\,x))$$

Cut Rule:

$$\forall a \forall b \forall l((a :: l) \rhd b\ \supset\ l \rhd \langle a \rangle\ \supset\ l \rhd b)$$

Structural Rules:

$$\forall i \forall b \forall l \forall l'(nat\ i\ \supset\ \forall a(element\ a\ l\ \supset\ element\ a\ l')\ \supset\ seq_i\ l\ b\ \supset\ seq_i\ l'\ b)$$

5.2 Linear logic

Now consider the fragment of second-order linear logic given by the grammar

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

where A ranges over atomic formulas and α ranges over ground types. As in Section 5.1, D and G represent definite clauses and goal formulas, respectively. The constants encoding these connectives have the same types as the corresponding constants used in Section 5.1; the new constant $\multimap$ has type $atm \to prp \to prp$.

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:

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

Table IX. Explicit eigenvariable encoding of lists

$$
\begin{array}{rcl}
\mathit{length}\ \mathit{nil}^*\ z & \triangleq & \top \\
\mathit{length}\ (A ::^* L)\ (s\ I) & \triangleq & \mathit{length}\ L\ I \\
\\
\mathit{list}\ L & \triangleq & \exists i(\mathit{nat}\ i \wedge \mathit{length}\ L\ i) \\
\\
\mathit{element}\ A\ (A ::^* L) & \triangleq & \top \\
\mathit{element}\ A\ (A' ::^* L) & \triangleq & \mathit{element}\ A\ L \\
\\
\mathit{split}\ \mathit{nil}^*\ \mathit{nil}^*\ \mathit{nil}^* & \triangleq & \top \\
\mathit{split}\ (A ::^* L_1)\ (A ::^* L_2)\ L_3 & \triangleq & \mathit{split}\ L_1\ L_2\ L_3 \\
\mathit{split}\ (A ::^* L_1)\ L_2\ (A ::^* L_3) & \triangleq & \mathit{split}\ L_1\ L_2\ L_3 \\
\\
\mathit{permute}\ \mathit{nil}^*\ \mathit{nil}^* & \triangleq & \top \\
\mathit{permute}\ (A ::^* L_1)\ L_2 & \triangleq & \exists l_{22}(\mathit{split}\ L_2\ (A ::^* \mathit{nil}^*)\ l_{22} \wedge \mathit{permute}\ L_1\ l_{22}\ )
\end{array}
$$

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 4.4 by encoding the antecedents and consequent as functions whose domain is a list of eigenvariables. We could use the explicit sequent technique to encode linear logic and still prove the adequacy and admissibility theorems of this section. However, in Part III we will use the linear logic encoding of this section as a specification logic; the proof of the unicity of typing theorem in Section 9 uses meta-theoretic analysis that is not possible if we use the explicit sequent technique here. This also gives us the opportunity to provide a detailed illustration of the explicit eigenvariable encoding style. In order to highlight both the similarities and differences between our current encoding and the encoding of Section 5.1, 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 $\mathit{evs} \to \tau$. Thus the type of *seq* above can be expressed as

$$\mathit{seq} : \mathit{nt} \to \mathit{atmlst}^* \to \mathit{atmlst}^* \to \mathit{prp}^* \to o\ .$$

We must modify the definition $\mathcal{D}(\mathit{list}(\tau))$ from Section 2.2 to work over the type lst^*. The predicates will now have the following types:

$$
\begin{array}{rlrl}
\mathit{length} & : \mathit{lst}^* \to \mathit{nt} \to o & \mathit{split} & : \mathit{lst}^* \to \mathit{lst}^* \to \mathit{lst}^* \to o \\
\mathit{list} & : \mathit{lst}^* \to o & \mathit{permute} & : \mathit{lst}^* \to \mathit{lst}^* \to o \\
\mathit{element} & : \tau^* \to \mathit{lst}^* \to o\ . & &
\end{array}
$$

The new definition $\mathcal{D}(\mathit{list}^*(\tau))$ is shown in Table IX; we use nil^* and $A ::^* L$ as abbreviations for $\lambda l\, \mathit{nil}$ 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 $\lambda l\, \mathit{tt}$, $B\ \&^*\ C$ 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 $\lambda l\, (\bigwedge x (B\, l\, x))$.

Any definite clause in our fragment of linear logic is equivalent to a formula of

Table X. Explicit eigenvariable encoding of linear logic

$$
\begin{array}{rcl}
seq_I\ IL\ (A ::^*\ nil^*)\ \langle A\rangle^* & \triangleq & \top \\
seq_I\ (A' ::^*\ IL)\ nil^*\ \langle A\rangle^* & \triangleq & element\ \langle A\rangle^*\ (A' ::^*\ IL) \\
seq_{(s\ I)}\ IL\ LL\ \langle A\rangle^* & \triangleq & \exists ll \exists il\ (list\ ll\ \wedge\ list\ il\ \wedge\ prog\ A\ ll\ il\ \wedge \\
 & & \quad split\_seq_I\ IL\ LL\ ll\ \wedge\ split\_seq_I\ IL\ nil^*\ il) \\
seq_I\ IL\ LL\ tt^* & \triangleq & \top \\
seq_{(s\ I)}\ IL\ LL\ (B\ \&^*\ C) & \triangleq & seq_I\ IL\ LL\ B\ \wedge\ seq_I\ IL\ LL\ C \\
seq_{(s\ I)}\ IL\ LL\ (A \multimap^*\ B) & \triangleq & seq_I\ IL\ (A ::^*\ LL)\ B \\
seq_{(s\ I)}\ IL\ LL\ (A \Rightarrow^*\ B) & \triangleq & seq_I\ (A ::^*\ IL)\ LL\ B \\
seq_{(s\ I)}\ IL\ LL\ (\bigwedge_i^*\ B) & \triangleq & seq_I\ (\lambda l\ IL\ (rst\ l))\ (\lambda l\ LL\ (rst\ l))\ (\lambda l\ B\ (rst\ l)\ (fst_i\ l)) \\
 & & \\
split\_seq_I\ IL\ nil^*\ nil^* & \triangleq & \top \\
split\_seq_I\ IL\ LL\ (B ::^*\ L) & \triangleq & \exists ll_1 \exists ll_2\ (split\ LL\ ll_1\ ll_2\ \wedge \\
 & & \quad seq_I\ IL\ ll_1 B\ \wedge\ split\_seq_I\ IL\ ll_2\ L)
\end{array}
$$

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 [Hodas and Miller 1994]; thus we use the predicate

$$prog\ :\ atm^* \rightarrow prplst^* \rightarrow prplst^* \rightarrow o$$

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 corresponding definitional clause for *prog*. Notice that the quantified variables of the definitional clause should be able to match terms containing object-level eigenvariables and so should have type i^* (for first-order variables) or $(i \rightarrow i)^*$ (for second-order variables). On the other hand, the definite clause itself should be closed, so the constants fst_τ and *rst* (used to encode eigenvariables) should not occur in the corresponding definitional clause. The predicate

$$split\_seq\ :\ nt \rightarrow atmlst^* \rightarrow atmlst^* \rightarrow prplst^* \rightarrow o$$

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 $\mathcal{D}(linear)$ of Table X, which defines the predicates *seq* and *split_seq*. The third clause in the definition says that an atomic formula A is derivable from intuitionistic antecedents IL and linear antecedents LL if there is a definite clause in the object-level theory whose head is A, whose linear hypotheses are derivable from the antecedents IL

and LL, and whose intuitionistic hypotheses are derivable from the antecendents IL. The other definitional clauses in Table X are similar to those in the explicit eigenvariable encoding of intuitionistic logic given in Section 4.4, but modified to reflect the linearity constraints. In the clause for $\bigwedge_i$ we subscript the constant *fst* with the type i because we also need a constant $fst_{i\to i} : evs \to i \to i$ for the representation of second-order eigenvariables in definite clauses. 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 Part III. For convenience we will abbreviate the formula $\exists i(nat\ i \wedge seq_i\ IL\ LL\ B)$ as $IL; LL \rhd B$ (or as $\rhd B$ when IL and LL are nil^*). If B is a term of type *prp* and L is a term of type *atmlst*, then let $\langle\!|B|\!\rangle$ and $\langle\!|L|\!\rangle$ be their translations into a formula of linear logic and a multiset of atomic formulas of linear logic, respectively.

Theorem 5.3 Adequacy of Encoding Linear Logic. *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*

$$\{\forall \bar{y}_1[prog\ A_1\ LL_1\ IL_1 \triangleq \top], \ldots, \forall \bar{y}_n[prog\ A_n\ LL_n\ IL_n \triangleq \top]\}$$

($n \geq 0$), where the quantified variables in the list $\bar{y}_i$ each have type i^* or $(i \to i)^*$, and the constants fst_τ and rst do not occur in A_i, LL_i, or IL_i, for all $i \in \{1, \ldots, n\}$. Let $\mathcal{P}$ be the theory in linear logic that corresponds to $\mathcal{D}(prog)$, and let $\mathcal{D}$ be a definition that extends $\mathcal{D}(nat) \cup \mathcal{D}(list^*(\mathrm{atm})) \cup \mathcal{D}(list^*(prp)) \cup \mathcal{D}(linear) \cup \mathcal{D}(prog)$ with clauses that do not define nat, length, list, element, split, split_seq, prog, or seq. Finally, let $IL\colon atmlst^*$, $LL\colon atmlst^*$, and $B\colon prp^*$ be terms that do not contain occurrences of the constant $fst_{i\to i}$. 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 $\mathcal{P}, \langle\!|IL|\!\rangle; \langle\!|LL|\!\rangle \longrightarrow \langle\!|B|\!\rangle$ 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 [Hodas and Miller 1994]. As before a cut-free derivation of $\longrightarrow IL; LL \rhd B$ 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. A more detailed proof of this theorem, including a definition of the $\langle\!|\ |\!\rangle$ translations, can be found in McDowell [1997]. □

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

$$\begin{aligned} subst &: nt \to i^* \to \tau^* \to \tau^* \to o \\ subst_0 &: nt \to i^{**} \to \tau^{**} \to \tau^{**} \to o \end{aligned}$$

will be used to represent substitution for eigenvariables; this is a simple generalization of the predicate of Section 4.4 to allow substitution in expressions of an arbitrary type τ. The type τ^{**} should be understood to mean $(\tau^*)^*$, *i.e.*, an ab-

Table XI. Encoding of eigenvariable operations

$$subst\ I\ T\ X\ X' \triangleq subst_0\ I\ (\lambda l'\ T)\ (\lambda l'\ X)\ (\lambda l'\ X')$$

$$subst_0\ z\ T\ (\lambda l'\lambda l\ X\ l'\ (fst\ l)\ (rst\ l))\ (\lambda l'\lambda l\ X\ l'\ (T\ l'\ l)\ (rst\ l)) \triangleq \top$$

$$subst_0\ (s\ I)\ (\lambda l'\lambda l\ T\ l'\ (fst\ l)\ (rst\ l))\ (\lambda l'\lambda l\ X\ l'\ (fst\ l)\ (rst\ l))\ (\lambda l'\lambda l\ X'\ l'\ (fst\ l)\ (rst\ l)) \triangleq subst_0\ I\ (\lambda l'\lambda l\ T\ (rst\ l')\ (fst\ l')\ l)\ (\lambda l'\lambda l\ X\ (rst\ l')\ (fst\ l')\ l)\ (\lambda l'\lambda l\ X'\ (rst\ l')\ (fst\ l')\ l)$$

$$extend\_evars\ I\ X\ X' \triangleq extend\_evars_0\ I\ (\lambda l'\ X)\ (\lambda l'\ X')$$

$$extend\_evars_0\ z\ (\lambda l'\lambda l\ X\ l'\ l)\ (\lambda l'\lambda l\ X\ l'\ (rst\ l)) \triangleq \top$$

$$extend\_evars_0\ (s\ I)\ (\lambda l'\lambda l\ X\ l'\ (fst\ l)\ (rst\ l))\ (\lambda l'\lambda l\ X'\ l'\ (fst\ l)\ (rst\ l)) \triangleq extend\_evars_0\ I\ (\lambda l'\lambda l\ X\ (rst\ l')\ (fst\ l')\ l)\ (\lambda l'\lambda l\ X'\ (rst\ l')\ (fst\ l')\ l)$$

brevation for $(evs \to evs \to \tau)$. We will also use the predicates

$$extend\_evars\ :\ nt \to \tau^* \to \tau^* \to o$$
$$extend\_evars_0\ :\ nt \to \tau^{**} \to \tau^{**} \to o$$

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 XI. 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, so the definition of *prog* should not contain occurrences of *fst* or *rst*. If this is the case, then P_{subst} and P_{extend} will be derivable in $FO\lambda^{\Delta\mathbb{N}}$.

The following theorem states 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 5.4 RULE ADMISSIBILITY FOR LINEAR LOGIC. *The formulas below are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *using the definition* $\mathcal{D}(nat) \cup \mathcal{D}(lists) \cup \mathcal{D}(evars) \cup \mathcal{D}(linear)$*:*
Specialization Rule:

$$\begin{array}{l} P_{subst}\ \supset \\ \forall i \forall b \forall il \forall ll (\mathrm{nat}\ i\ \supset\ \mathit{list}\ il\ \supset\ \mathit{list}\ ll\ \supset \\ \qquad \mathit{seq}_{(\mathrm{S}\ i)}\ il\ ll\ \bigwedge^*\ b\ \supset\ \forall x\, \mathit{seq}_i\ il\ ll\ (b\, x)) \end{array}$$

Cut Rule:

$$\begin{array}{l} P_{extend}\ \supset \\ \forall a \forall b \forall il \forall ll (\mathit{list}\ il\ \supset\ \mathit{list}\ ll\ \supset \\ \qquad (a ::^* il); ll \rhd b\ \supset\ il; \mathit{nil}^* \rhd \langle a \rangle^*\ \supset\ il; ll \rhd b) \end{array}$$

Structural Rules:

$$\begin{array}{l} P_{extend}\ \supset \\ \forall a \forall b \forall il \forall ll \forall ll_1 \forall ll_2 (\mathit{list}\ il\ \supset\ \mathit{list}\ ll\ \supset\ \mathit{split}\ ll\ ll_1\ ll_2\ \supset \\ \qquad il; (a ::^* ll_1) \rhd b\ \supset\ il; ll_2 \rhd \langle a \rangle^*\ \supset\ il; ll \rhd b) \end{array}$$

$$\begin{array}{l} \forall i \forall b \forall il \forall il' \forall ll \forall ll' (\mathrm{nat}\ i\ \supset\ \mathit{list}\ il\ \supset\ \mathit{list}\ il'\ \supset\ \mathit{list}\ ll\ \supset \\ \qquad \forall a (\mathit{element}\ a\ il\ \supset\ \mathit{element}\ a\ il')\ \supset\ \mathit{permute}\ ll\ ll'\ \supset \\ \qquad \mathit{seq}_i\ il\ ll\ b\ \supset\ \mathit{seq}_i\ il'\ ll'\ b) \end{array}$$

6. RELATED WORK

In this part of the paper 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 4.1 is the prototypical representation style of higher-order abstract syntax. For example, the seminal paper on the Edinburgh Logical Framework (LF) [Harper et al. 1993] 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 there.

The use of separate predicates for formulas on the left and right sides of the sequent, as was done in Section 4.2, is also common. Pfenning [1995], 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 [Pfenning and Rohwedder 1992; Rohwedder and Pfenning 1996]. Miller [1996] 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 a linear logic meta-logic.

Section 4.3 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 [Miller and Nadathur 1987]. That 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 4.4, was first used in the context of higher-order abstract syntax by Despeyroux and Hirschowitz [1994]. 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:

$$seq_{(S\ I)}\ L\ (\lambda l \bigwedge_i x(B\,(cons\,x\,l)))\ \triangleq\ seq_I\ (\lambda l'\ L(rst\ l'))\ B\ .$$

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 for variable lists. The paper Despeyroux and Hirschowitz [1994] 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 discuss their work further at the end of Part III.

Part III: OBJECT LOGICS AS SPECIFICATION LOGICS

In this part 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 part; 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 [1995], 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 8 to formalize this proof by encoding the static and dynamic semantics for untyped λ-terms in the intuitionistic object logic of Section 5.1. We also list a variety of other theorems about the language that we have derived in $FO\lambda^{\Delta\mathbb{N}}$. The remainder of the section extends the encoding to the Programming language of Computable Functions (PCF) [Scott 1969]. In Section 9 we consider an encoding of PCF with references ($\text{PCF}_{:=}$) [Gunter 1992] in the linear object logic of Section 5.2. Finally, Section 10 compares the framework of this part with other research in formal reasoning about higher-order abstract syntax encodings.

7. 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, Avron et al. [1992] and Hannan [1990].

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 7.1. *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 P is $(app\ M\ N)$, $typeof\, P\, T$ must have been derived using the formula encoding the typing rule for

app. Hence, there is a U such that $\vdash \mathit{typeof}\, M\ (\mathit{arr}\, U\ T)$ and $\vdash \mathit{typeof}\, N\ U$. Applying the inductive hypothesis to the evaluation and typing judgements for M, we have $\vdash \mathit{typeof}\, (\mathit{abs}\, R)\ (\mathit{arr}\, U\ T)$. This atomic formula must have been derived using the *typeof* formula for *abs*, and, hence, $\vdash \bigwedge x(\mathit{typeof}\, x\ U \Rightarrow \mathit{typeof}\, (R\, x)\ T)$. Since our specification logic is intuitionistic logic, we can instantiate this quantifier with N and use cut and cut-elimination to conclude that $\vdash \mathit{typeof}\, (R\, N)\ T$. Applying the inductive hypothesis to the judgements for $(R\, N)$ yields $\vdash \mathit{typeof}\, V\ T$. □

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 this second (meta-) logic would encode judgements in the first (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 $(\mathit{abs}\ R)$ and in another part of the proof to $(\mathit{app}\ M\ N)$. 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

$$\vdash \mathit{typeof}\, (\mathit{abs}\, R)\ (\mathit{arr}\, U\ T)$$

to the assumption

$$\vdash \bigwedge x(\mathit{typeof}\, x\ U \Rightarrow \mathit{typeof}\, (R\, x)\ T)\ .$$

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 Part II. The $\mathit{nat}\mathcal{L}$ rule of $FO\lambda^{\Delta\mathbb{N}}$ provides natural number induction. The last two features are accommodated by the definition facilities of $FO\lambda^{\Delta\mathbb{N}}$, in particular the $\mathit{def}\mathcal{L}$ rule. We demonstrate our approach in the remaining sections of the paper, beginning with a formalization of the example from this section.

8. REPRESENTATION AND ANALYSIS OF A FUNCTIONAL PROGRAMMING LANGUAGE

8.1 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

Table XII. 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	$(M \rightsquigarrow^* N)$	$\langle M \rightsquigarrow M'\rangle\,\&\,\langle M' \rightsquigarrow^* N\rangle$

object logic used will be the fragment of second-order intuitionistic logic encoded by the definition $\mathcal{D}(intuit)$ of Section 5.1.

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 [Miller 1991] and are those common to specifications written in, say, λProlog [Hannan and Miller 1992] and Elf [Pfenning 1989]. This object-level specification is represented in $FO\lambda^{\Delta\mathbb{N}}$ as the definition $\mathcal{D}(lambda)$ shown in Table XII. (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 theorem lists 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 8.1. *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 $\equiv\colon i \to i \to o$.*

Determinacy of semantics:

$$\forall m \forall m_1 \forall m_2 (\triangleright\langle m \Downarrow m_1\rangle \supset \triangleright\langle m \Downarrow m_2\rangle \supset m_1 \equiv m_2)$$
$$\forall m \forall m_1 \forall m_2 (\triangleright\langle m \rightsquigarrow m_1\rangle \supset \triangleright\langle m \rightsquigarrow m_2\rangle \supset m_1 \equiv m_2)$$
$$\forall m \forall r_1 \forall r_2 (\triangleright\langle m \rightsquigarrow^* (abs\,r_1)\rangle \supset \triangleright\langle m \rightsquigarrow^* (abs\,r_2)\rangle \supset (abs\,r_1) \equiv (abs\,r_2))$$

Equivalence of semantics:

$$\forall m \forall r(\rhd\langle m \Downarrow (abs\, r)\rangle \supset \rhd\langle m \rightsquigarrow^* (abs\, r)\rangle)$$
$$\forall m \forall r(\rhd\langle m \rightsquigarrow^* (abs\, r)\rangle \supset \rhd\langle m \Downarrow (abs\, r)\rangle)$$

Subject reduction:

$$\forall m \forall n(\rhd\langle m \Downarrow n\rangle \supset \forall t(\rhd\langle typeof\, m\ t\rangle \supset \rhd\langle typeof\, n\ t\rangle))$$
$$\forall m \forall n(\rhd\langle m \rightsquigarrow n\rangle \supset \forall t(\rhd\langle typeof\, m\ t\rangle \supset \rhd\langle typeof\, n\ t\rangle))$$
$$\forall m \forall n(\rhd\langle m \rightsquigarrow^* n\rangle \supset \forall t(\rhd\langle typeof\, m\ t\rangle \supset \rhd\langle typeof\, n\ t\rangle))$$

Proof. We show the derivation of the first subject reduction property, which is a formalization of Proposition 7.1.

We wish to show that evaluation preserves types:

$$\longrightarrow \forall p \forall v(\rhd\langle p \Downarrow v\rangle \supset \forall t(\rhd\langle typeof\, p\ t\rangle \supset \rhd\langle typeof\, v\ t\rangle)) \ .$$

(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

$$nat\ i, seq_i\ nil\ \langle p \Downarrow v\rangle, \rhd\langle typeof\, p\ t\rangle \longrightarrow \rhd\langle typeof\, v\ t\rangle \ .$$

(Recall that $\rhd\langle p \Downarrow v\rangle$ 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 2.3) and our induction predicate will be

$$\lambda i \forall p \forall v \forall t(seq_i\ nil\ \langle p \Downarrow v\rangle \supset \rhd\langle typeof\, p\ t\rangle \supset \rhd\langle typeof\, v\ t\rangle) \ ,$$

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

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

We use the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules to obtain

$$nat\ j, \forall k \ldots, seq_j\ nil\ \langle p \Downarrow v\rangle, \rhd\langle typeof\, p\ t\rangle \longrightarrow \rhd\langle typeof\, v\ t\rangle \ .$$

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 by applying the $def\mathcal{L}$ rule to $seq_j\ nil\ \langle p \Downarrow v\rangle$, which yields

$$nat\ (s\ j_0), \forall k \ldots, \exists b(prog\ (p \Downarrow v)\ b \wedge seq_{j_0}\ nil\ b), \rhd\langle typeof\, p\ t\rangle \longrightarrow \rhd\langle typeof\, v\ t\rangle \ .$$

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

$$\ldots, nat\ (s^2\ j_1'), seq_{j_1'}\ nil\ \langle typeof\ m\ (arr\ u\ t)\rangle, \\ seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .$$

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

$$nat\ (s^2\ j_1), \ldots \longrightarrow nat\ j_1 \qquad\qquad nat\ (s^2\ j_1), \ldots \longrightarrow j_1 < (s^2\ j_1)$$

$$\ldots, seq_{j_1}\ nil\ \langle m \Downarrow (abs\ r)\rangle, \ldots \longrightarrow seq_{j_1}\ nil\ \langle m \Downarrow (abs\ r)\rangle$$

$$\ldots, nat\ (s^2\ j_1'), seq_{j_1'}\ nil\ \langle typeof\ m\ (arr\ u\ t)\rangle, \ldots \longrightarrow \rhd\langle typeof\ m\ (arr\ u\ t)\rangle$$

$$nat\ (s^2\ j_1), \forall k \ldots, seq_{j_1}\ nil\ \langle (r\ n) \Downarrow v\rangle, \\ \rhd\langle typeof\ (abs\ r)\ (arr\ u\ t)\rangle, \\ nat\ (s^2\ j_1'), seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .$$

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 2.4 these are derivable in $FO\lambda^{\Delta\mathbb{N}}$ from $\mathcal{D}(nat)$. The third sequent is immediate, and the fourth also follows easily from Proposition 2.4.

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

$$\ldots, nat\ (s\ j_0''), seq_{j_0''}\ nil\ \bigwedge x((typeof\ x\ u) \Rightarrow \langle typeof\ (r\ x)\ t\rangle), \\ \ldots \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .$$

This is followed by applications of the $def\mathcal{L}$ and $\forall\mathcal{L}$ rules to give us

$$\ldots, nat\ (s^3\ j_1''), seq_{j_1''}\ ((typeof\ n\ u) :: nil)\ \langle typeof\ (r\ n)\ t\rangle, \ldots \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .$$

The informal proof proceeds with a use of the cut rule, and here we use the derived object-level cut rule (Theorem 5.2) with the elided assumption $seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle$ to obtain

$$\ldots, nat\ (s^3\ j_1''), seq_{j_1''}\ ((typeof\ n\ u)::nil)\ \langle typeof\ (r\ n)\ t\rangle, \ldots \longrightarrow ((typeof\ n\ u)::nil) \rhd \langle typeof\ (r\ n)\ t\rangle$$

$$\ldots, nat\ (s^2\ j_1'), seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle \longrightarrow \rhd\langle typeof\ n\ u\rangle$$

$$\ldots, \rhd\langle typeof\ (r\ n)\ t\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .$$

The first two of these follow easily from Proposition 2.4.

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

$$nat\ (s^2\ j_1) \longrightarrow nat\ j_1 \qquad\qquad nat\ (s^2\ j_1) \longrightarrow j_1 < (s^2\ j_1)$$

$$\ldots, seq_{j_1}\ nil\ \langle (r\ n) \Downarrow v\rangle, \ldots \longrightarrow seq_{j_1}\ nil\ \langle (r\ n) \Downarrow v\rangle$$

$$\ldots, \rhd\langle typeof\ (r\ n)\ t\rangle \longrightarrow \rhd\langle typeof\ (r\ n)\ t\rangle$$

$$\ldots, \rhd\langle typeof\ v\ t\rangle \longrightarrow \rhd\langle typeof\ v\ t\rangle\ .$$

The first two sequents follow from Proposition 2.4, and the last three are all immediate. □

8.2 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 [Scott 1969]. 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

$$\begin{array}{lll} zero : i_{tm} & succ : i_{tm} \to i_{tm} & if : i_{tm} \to i_{tm} \to i_{tm} \to i_{tm} \\ true : i_{tm} & pred : i_{tm} \to i_{tm} & abs : i_{ty} \to (i_{tm} \to i_{tm}) \to i_{tm} \\ false : i_{tm} & is\_zero : i_{tm} \to i_{tm} & app : i_{tm} \to i_{tm} \to i_{tm} \\ & & rec : i_{ty} \to (i_{tm} \to i_{tm}) \to i_{tm}\ . \end{array}$$

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 8.1, 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 XIII, XIV, and XV; we have again omitted the $\stackrel{\triangle}{=} \top$ body of the clauses. The following theorem lists the properties of PCF that we have derived in $FO\lambda^{\Delta\mathbb{N}}$. The type tags in PCF terms allow the unicity of

Table XIII. Object logic encoding of typing for PCF

prog	$(\mathit{typeof}\ \mathit{zero}\ \mathit{num})$	tt
prog	$(\mathit{typeof}\ \mathit{true}\ \mathit{bool})$	tt
prog	$(\mathit{typeof}\ \mathit{false}\ \mathit{bool})$	tt
prog	$(\mathit{typeof}\ (\mathit{succ}\ M)\ \mathit{num})$	$\langle \mathit{typeof}\ M\ \mathit{num}\rangle$
prog	$(\mathit{typeof}\ (\mathit{pred}\ M)\ \mathit{num})$	$\langle \mathit{typeof}\ M\ \mathit{num}\rangle$
prog	$(\mathit{typeof}\ (\mathit{is\_zero}\ M)\ \mathit{bool})$	$\langle \mathit{typeof}\ M\ \mathit{num}\rangle$
prog	$(\mathit{typeof}\ (\mathit{if}\ M\ N_1\ N_2)\ T)$	$\langle \mathit{typeof}\ M\ \mathit{bool}\rangle\ \&\ \langle \mathit{typeof}\ N_1\ T\rangle\ \&\ \langle \mathit{typeof}\ N_2\ T\rangle$
prog	$(\mathit{typeof}\ (\mathit{abs}\ T\ R)\ (\mathit{arr}\ T\ U))$	$\bigwedge n((\mathit{typeof}\ n\ T) \Rightarrow \langle \mathit{typeof}\ (R\,n)\ U\rangle)$
prog	$(\mathit{typeof}\ (\mathit{app}\ M\ N)\ T)$	$\langle \mathit{typeof}\ M\ (\mathit{arr}\ U\ T)\rangle\ \&\ \langle \mathit{typeof}\ N\ U\rangle$
prog	$(\mathit{typeof}\ (\mathit{rec}\ T\ R)\ T)$	$\bigwedge n((\mathit{typeof}\ n\ T) \Rightarrow \langle \mathit{typeof}\ (R\,n)\ T\rangle)$

Table XIV. Object logic encoding of natural semantics for PCF

prog	$(\mathit{zero} \Downarrow \mathit{zero})$	tt
prog	$(\mathit{true} \Downarrow \mathit{true})$	tt
prog	$(\mathit{false} \Downarrow \mathit{false})$	tt
prog	$((\mathit{succ}\ M) \Downarrow (\mathit{succ}\ V))$	$\langle M \Downarrow V\rangle$
prog	$((\mathit{pred}\ M) \Downarrow \mathit{zero})$	$\langle M \Downarrow \mathit{zero}\rangle$
prog	$((\mathit{pred}\ M) \Downarrow V)$	$\langle M \Downarrow (\mathit{succ}\ V)\rangle$
prog	$((\mathit{is\_zero}\ M) \Downarrow \mathit{true})$	$\langle M \Downarrow \mathit{zero}\rangle$
prog	$((\mathit{is\_zero}\ M) \Downarrow \mathit{false})$	$\langle M \Downarrow (\mathit{succ}\ V)\rangle$
prog	$((\mathit{if}\ M\ N_1\ N_2) \Downarrow V)$	$\langle M \Downarrow \mathit{true}\rangle\ \&\ \langle N_1 \Downarrow V\rangle$
prog	$((\mathit{if}\ M\ N_1\ N_2) \Downarrow V)$	$\langle M \Downarrow \mathit{false}\rangle\ \&\ \langle N_2 \Downarrow V\rangle$
prog	$((\mathit{abs}\ T\ R) \Downarrow (\mathit{abs}\ T\ R))$	tt
prog	$((\mathit{app}\ M\ N) \Downarrow V)$	$\langle M \Downarrow (\mathit{abs}\ T\ R)\rangle\ \&\ \langle (R\,N) \Downarrow V\rangle$
prog	$((\mathit{rec}\ T\ R) \Downarrow V)$	$\langle (R\,(\mathit{rec}\ T\ R)) \Downarrow V\rangle$

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 discuss below.

THEOREM 8.2. *The following formulas are derivable in* $FO\lambda^{\Delta\mathbb{N}}$ *from the definition that accumulates* $\mathcal{D}(nat)$, $\mathcal{D}(list(\mathrm{atm}))$, $\mathcal{D}(intuit)$, $\mathcal{D}(\mathrm{PCF})$ *and the clause* $X \equiv X \triangleq \top$ *defining the predicate* $\equiv\colon i \to i \to o$.

Determinacy of semantics:

$$\forall m \forall m_1 \forall m_2 (\triangleright\langle m \Downarrow m_1\rangle \supset \triangleright\langle m \Downarrow m_2\rangle \supset m_1 \equiv m_2)$$
$$\forall m \forall m_1 \forall m_2 (\triangleright\langle m \rightsquigarrow m_1\rangle \supset \triangleright\langle m \rightsquigarrow m_2\rangle \supset m_1 \equiv m_2)$$
$$\forall m \forall v_1 \forall v_2 (\triangleright\langle \mathit{value}\ v_1\rangle \supset \triangleright\langle m \rightsquigarrow^* v_1\rangle \supset \triangleright\langle \mathit{value}\ v_2\rangle \supset \triangleright\langle m \rightsquigarrow^* v_2\rangle \supset v_1 \equiv v_2)$$

Equivalence of semantics:

$$\forall m \forall v (\triangleright\langle m \Downarrow v\rangle \supset (\triangleright\langle \mathit{value}\ v\rangle \wedge \triangleright\langle m \rightsquigarrow^* v\rangle))$$
$$\forall m \forall v ((\triangleright\langle \mathit{value}\ v\rangle \wedge \triangleright\langle m \rightsquigarrow^* v\rangle) \supset \triangleright\langle m \Downarrow v\rangle)$$

Table XV. 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	$((app\ (abs\ T\,R)\ N) \rightsquigarrow (R\,N))$	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	$(value\ (abs\ T\,R))$	tt

Subject reduction:

$$\forall m \forall n (\triangleright \langle m \Downarrow n \rangle \supset \forall t (\triangleright \langle typeof\ m\ t \rangle \supset \triangleright \langle typeof\ n\ t \rangle))$$
$$\forall m \forall n (\triangleright \langle m \rightsquigarrow n \rangle \supset \forall t (\triangleright \langle typeof\ m\ t \rangle \supset \triangleright \langle typeof\ n\ t \rangle))$$
$$\forall m \forall n (\triangleright \langle m \rightsquigarrow^* n \rangle \supset \forall t (\triangleright \langle typeof\ m\ t \rangle \supset \triangleright \langle typeof\ n\ t \rangle))$$

Unicity of typing:

$$\forall m \forall t_1 \forall t_2 (\triangleright \langle typeof\ m\ t_1 \rangle \supset \triangleright \langle typeof\ m\ t_2 \rangle \supset t_1 \equiv t_2)$$

The usual informal proof of the unicity of typing relies on the requirement that the list of assumptions in the object logic sequent contains typing assignments only 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 $FO\lambda^{\Delta\mathbb{IN}}$, we cannot state the first part of this requirement in $FO\lambda^{\Delta\mathbb{IN}}$. Thus our derivation (given in McDowell [1997]) 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 does not generalize to languages without this construct. In the next section we give an encoding of an extension of PCF in the object logic of Section 5.2, which is encoded in $FO\lambda^{\Delta\mathbb{IN}}$ using the explicit eigenvariable encoding. Although this explicit eigenvariable encoding makes the syntax more cumbersome, it allows the derivations in $FO\lambda^{\Delta\mathbb{IN}}$ to be more natural. This is illustrated by the fact that we can capture in $FO\lambda^{\Delta\mathbb{IN}}$ the typical proof of the unicity of typing.

9. REPRESENTATION AND ANALYSIS OF AN IMPERATIVE PROGRAMMING LANGUAGE

In this section we consider the programming language $\text{PCF}_{:=}$, an extension of PCF with state [Gunter 1992]. 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 the value of m is a memory location, and $!m$ has type τ and evaluates to the contents of that location. 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.

To encode $\text{PCF}_{:=}$, we use the linear object logic of Section 5.2, since linear logic is well-suited as a specification logic for programming languages with state [Cervesato and Pfenning 1996; Chirimar 1995; Miller 1996]. 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. A continuation-based semantics for $\text{PCF}_{:=}$ is given in Table XVI; following Gunter [1992] we specify call-by-value evaluation. To abbreviate our presentation we omit the rules for the natural number, boolean, and conditional constructs; a presentation with the full language is given in McDowell [1997]. The semantics of Table XVI and their object logic encoding given below are a variation of those found in Cervesato and Pfenning [1996]. 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 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. In the rules of Table XVI, c is used to range over locations (reference cells). In the rule for the continuation $(\hat{x}.\mathit{ref}\ x, \kappa)$, c must be a new location, *i.e.*, a location that does not occur in the state σ. 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 constants

$$\begin{array}{lll} \mathit{refty} : i_{ty} \to i_{ty} & \mathit{ref} : i_{tm} \to i_{tm} & \mathit{assign} : i_{tm} \to i_{tm} \to i_{tm} \\ \mathit{cell} : i_{lc} \to i_{tm} & \mathit{deref} : i_{tm} \to i_{tm} & \mathit{sequence} : i_{tm} \to i_{tm} \to i_{tm} \end{array}$$

in addition to the constants of Section 8.2. Once again we have labeled the type i with subscripts to improve the readability of these declarations. The subscript *lc*

Table XVI. Continuation-based natural semantics for $\mathrm{PCF}_{:=}$

$$\frac{}{\vdash (V,\sigma)\dot{\hookrightarrow}(V,\sigma)} \qquad \frac{\hat{x}.\mathit{ref}\,x,\kappa \vdash (M,\sigma)\hookrightarrow\phi}{\kappa\vdash(\mathit{ref}\,M,\sigma)\hookrightarrow\phi} \qquad \frac{\hat{x}.!x,\kappa\vdash(M,\sigma)\hookrightarrow\phi}{\kappa\vdash(!M,\sigma)\hookrightarrow\phi}$$

$$\frac{\kappa\vdash(c,\sigma)\dot{\hookrightarrow}\phi}{\kappa\vdash(c,\sigma)\hookrightarrow\phi} \qquad \frac{\kappa\vdash(c,\sigma[c\mapsto V])\dot{\hookrightarrow}\phi}{\hat{x}.\mathit{ref}\,x,\kappa\vdash(V,\sigma)\dot{\hookrightarrow}\phi} \qquad \frac{\kappa\vdash(\sigma(c),\sigma)\dot{\hookrightarrow}\phi}{\hat{x}.!x,\kappa\vdash(c,\sigma)\dot{\hookrightarrow}\phi}$$

$$\frac{\hat{x}.x:=N,\kappa\vdash(M,\sigma)\hookrightarrow\phi}{\kappa\vdash(M:=N,\sigma)\hookrightarrow\phi} \qquad \frac{\hat{x}.V:=x,\kappa\vdash(N,\sigma)\hookrightarrow\phi}{\hat{x}.x:=N,\kappa\vdash(V,\sigma)\dot{\hookrightarrow}\phi} \qquad \frac{\kappa\vdash(V,\sigma[c\mapsto V])\dot{\hookrightarrow}\phi}{\hat{x}.c:=x,\kappa\vdash(V,\sigma)\dot{\hookrightarrow}\phi}$$

$$\frac{\hat{x}.x;N,\kappa\vdash(M,\sigma)\hookrightarrow\phi}{\kappa\vdash(M;N,\sigma)\hookrightarrow\phi} \qquad \frac{\kappa\vdash(N,\sigma)\hookrightarrow\phi}{\hat{x}.x;N,\kappa\vdash(V,\sigma)\dot{\hookrightarrow}\phi}$$

$$\frac{\hat{x}.x\,N,\kappa\vdash(M,\sigma)\hookrightarrow\phi}{\kappa\vdash(M\,N,\sigma)\hookrightarrow\phi} \qquad \frac{\hat{x}.V\,x,\kappa\vdash(N,\sigma)\hookrightarrow\phi}{\hat{x}.x\,N,\kappa\vdash(V,\sigma)\dot{\hookrightarrow}\phi} \qquad \frac{\kappa\vdash(\lambda x:\tau.M,\sigma)\dot{\hookrightarrow}\phi}{\kappa\vdash(\lambda x:\tau.M,\sigma)\hookrightarrow\phi}$$

$$\frac{\kappa\vdash(M'[V/y],\sigma)\hookrightarrow\phi}{\hat{x}.(\lambda y:\tau.M')\,x,\kappa\vdash(V,\sigma)\dot{\hookrightarrow}\phi} \qquad \frac{\kappa\vdash(M[\mathit{rec}\,x:\tau.M\,/x],\sigma)\hookrightarrow\phi}{\kappa\vdash(\mathit{rec}\,x:\tau.M\,,\sigma)\hookrightarrow\phi}$$

Table XVII. Object logic encoding of typing for $\mathrm{PCF}_{:=}$ terms

$$\begin{array}{lll}
\mathit{prog} & \mathit{typeof}^*\ (\mathit{abs}^*\ T\ R)\ (\mathrm{arr}^*\ T\ U)) & \\
 & \lambda l(\bigwedge n(\mathit{typeof}\,n\ (T\,l)\Rightarrow\langle\mathit{typeof}\,(R\,l\,n)\ (U\,l)\rangle)::\mathit{nil}) & \mathit{nil}^* \\
\mathit{prog} & (\mathit{typeof}^*\ (\mathit{app}^*\ M\ N)\ T) & \\
 & (\langle\mathit{typeof}^*\ M\ (\mathrm{arr}^*\ U\ T)\rangle^*::^*\langle\mathit{typeof}^*\ N\ U\rangle^*::^*\mathit{nil}^*) & \mathit{nil}^* \\
\mathit{prog} & (\mathit{typeof}^*\ (\mathit{rec}^*\ T\ R)\ T) & \\
 & \lambda l(\bigwedge n(\mathit{typeof}\,n\ (T\,l)\Rightarrow\langle\mathit{typeof}\,(R\,l\,n)\ (T\,l)\rangle)::\mathit{nil}) & \mathit{nil}^* \\
\mathit{prog} & (\mathit{typeof}^*\ (\mathit{ref}^*\ M)\ (\mathit{refty}^*\ T)) & \\
 & (\langle\mathit{typeof}^*\ M\ T\rangle^*::^*\mathit{nil}^*) \qquad \mathit{nil}^* & \\
\mathit{prog} & (\mathit{typeof}^*\ (\mathit{deref}^*\ M)\ T) & \\
 & (\langle\mathit{typeof}^*\ M\ (\mathit{refty}^*\ T)\rangle^*::^*\mathit{nil}^*) \qquad \mathit{nil}^* & \\
\mathit{prog} & (\mathit{typeof}^*\ (\mathit{assign}^*\ M\ N)\ T) & \\
 & (\langle\mathit{typeof}^*\ M\ (\mathit{refty}^*\ T)\rangle^*::^*\langle\mathit{typeof}^*\ N\ T\rangle^*::^*\mathit{nil}^*) & \mathit{nil}^* \\
\mathit{prog} & (\mathit{typeof}^*\ (\mathit{sequence}^*\ M\ N)\ T) & \\
 & (\langle\mathit{typeof}^*\ M\ U\rangle^*::^*\langle\mathit{typeof}^*\ N\ T\rangle^*::^*\mathit{nil}^*) \qquad \mathit{nil}^* &
\end{array}$$

indicates that the argument to *cell* represents a $\mathrm{PCF}_{:=}$ location.

The object logic predicate representing typability is denoted by the same $FO\lambda^{\Delta\mathbb{N}}$ constants as in Section 8; its object-level specification is represented in $FO\lambda^{\Delta\mathbb{N}}$ as the definition shown in Table XVII. Recall that $\mathit{prog}\ A\ (C_1::\ldots C_n::\mathit{nil})\ (B_1::\ldots B_m::\mathit{nil})$ 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 $IL; LL \rhd \langle A\rangle$, the rules of linear logic require that each assumption in LL 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 LL 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 to the constants of this section. Thus ($typeof^*$ m t) abbreviates ($\lambda l\, typeof\,(m\,l)\,(t\,l)$), ($refty^*$ t) abbreviates ($\lambda l\, refty\,(t\,l)$), etc.

The semantics for $\text{PCF}_{:=}$ is more complicated than those in the previous sections. The constant $\Downarrow$ now has type $i_{tm} \to i_{st} \to i_{ans} \to atm$. The object logic atom $(m, s) \Downarrow f$ 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 XVI. 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

$$\begin{aligned} eval &: i_{tm} \to i_{instr} & new\_ref &: i_{tm} \to i_{instr} \\ return &: i_{tm} \to i_{instr} & lookup &: i_{tm} \to i_{instr} \\ eval\_arg &: i_{tm} \to i_{tm} \to i_{instr} & eval\_rvalue &: i_{tm} \to i_{tm} \to i_{instr} \\ apply &: i_{tm} \to i_{tm} \to i_{instr} & update &: i_{tm} \to i_{tm} \to i_{instr}\ , \end{aligned}$$

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 XVIII and XIX.

This encoding differs slightly from the continuation semantics in Table XVI. The object logic judgement $nil^*; ll \rhd \langle ns\_mach\_2\, k\, (return\ v)\, f\rangle^*$ corresponds to the judgement $\kappa \vdash (v', \sigma) \dot{\hookrightarrow} \phi$, 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.

Table XVIII. Object logic encoding of natural semantics for $\mathrm{PCF}_{:=}$ (part I)

$$
\begin{array}{lll}
\mathit{prog} & ((M,S) \Downarrow^* F) & \\
 & \mathit{nil}^* \qquad (\langle \mathit{ns\_mach\_1}^*\ \mathit{init}^*\ (\mathit{eval}^*\ M)\ S\ F\rangle^* ::^* \mathit{nil}^*) & \\
 & & \\
\mathit{prog} & (\mathit{ns\_mach\_1}^*\ K\ I\ (\mathit{extend\_st}^*\ C\ V\ S)\ F) & \\
 & ((\mathit{contains}^*\ C\ V \multimap^* \langle \mathit{ns\_mach\_1}^*\ K\ I\ S\ F\rangle^*) ::^* \mathit{nil}^*) \qquad \mathit{nil}^* & \\
\mathit{prog} & (\mathit{ns\_mach\_1}^*\ K\ I\ \mathit{null\_st}^*\ F) & \\
 & (\langle \mathit{ns\_mach\_2}^*\ K\ I\ F\rangle^* ::^* \mathit{nil}^*) \qquad \mathit{nil}^* & \\
 & & \\
\mathit{prog} & (\mathit{collect\_state}^*\ (\mathit{extend\_st}^*\ C\ V\ S)) & \\
 & (\langle \mathit{contains}^*\ C\ V\rangle^* ::^* \langle \mathit{collect\_state}^*\ S\rangle^* ::^* \mathit{nil}^*) \qquad \mathit{nil}^* & \\
\mathit{prog} & (\mathit{collect\_state}^*\ \mathit{null\_st}^*) & \\
 & \mathit{nil}^* \qquad \mathit{nil}^* &
\end{array}
$$

On the other hand, the rules of Table XVI 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 XVIII, and XIX makes vital use of linear implication. Since each assumption of the form $\mathit{contains}^*\ c\ v$ 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 $(\mathit{update}^*\ (\mathit{cell}^*\ c)\ v)$; 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, $\langle \mathit{contains}^*\ c\ w\rangle^*$ and $(\mathit{contains}^*\ c\ v \multimap^* \langle \mathit{ns\_mach\_2}^*\ k\ (\mathit{return}^*\ v)\ f\rangle^*)$. 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 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 $(\mathit{return}^*\ v)$ 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 $\langle \mathit{collect\_state}^*\ s\rangle^*$ 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 XVIII has a single intuitionistic formula $\langle \mathit{ns\_mach\_1}^*\ \mathit{init}^*\ (\mathit{eval}^*\ m)\ s\ f\rangle^*$ 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:

$$
\begin{array}{lll}
\mathit{typeof}_{cntn} : i_{cntn} \to i_{ty} \to \mathit{atm} & & \mathit{typeof}_{ans} : i_{ans} \to i_{ty} \to \mathit{atm} \\
\mathit{typeof}_{instr} : i_{instr} \to i_{ty} \to \mathit{atm}\ . & &
\end{array}
$$

Table XIX. 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^* ::^*\ (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^* ::^*\ (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^* \\
\\
prog\ (ns\_mach\_2^*\ \ K\ \ (eval^*\ (app^*\ M\ N))\ \ F) \\
\quad (\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) \\
\quad (\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) \\
\quad (\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) \\
\quad (\langle ns\_mach\_2^*\ \ K\ \ (return^*\ (abs^*\ T\ R))\ \ F\rangle^* ::^*\ nil^*) \qquad nil^* \\
prog\ (ns\_mach\_2^*\ \ K\ \ (eval^*\ (rec^*\ T\ R))\ \ F) \\
\quad (\langle ns\_mach\_2^*\ \ K\ \ (eval^*\ (\lambda l\, R\, l\, (rec\ (T\, l)\ (R\, l))))\ \ F\rangle^* ::^*\ nil^*) \qquad nil^*
\end{array}
$$

The object-level specification for these predicates is represented in $FO\lambda^{\Delta\mathbb{N}}$ by the definition of Table XX. 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;

Table XX. Encoding of typing for $\text{PCF}_{:=}$ continuations, instructions, and answers

$$
\begin{array}{l}
prog\ (typeof^*_{cntn}\ \ init^*\ \ (arr^*\ T\ T)) \\
\quad nil^* \qquad nil^* \\
prog\ (typeof^*_{cntn}\ \ (I \succ^*\ K)\ \ (arr^*\ T\ U)) \\
\quad (\lambda l \bigwedge v(typeof\, v\ (T\, l) \Rightarrow \langle typeof_{instr}\ \ (I\, l\, v)\ \ (T'\, l)\rangle) ::^* \\
\quad \langle typeof^*_{cntn}\ \ K\ \ (arr^*\ T'\ U)\rangle^* ::^* nil^*) \\
\quad nil^* \\
\\
prog\ (typeof^*_{instr}\ \ (eval^*\ M)\ \ T) \\
\quad (\langle typeof^*\ M\ T\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ (typeof^*_{instr}\ \ (return^*\ V)\ \ T) \\
\quad (\langle typeof^*\ V\ T\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ (typeof^*_{instr}\ \ (eval\_arg^*\ M\ N)\ \ T) \\
\quad (\langle typeof^*\ M\ (arr^*\ U\ T)\rangle^* ::^* \langle typeof^*\ N\ U\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ (typeof^*_{instr}\ \ (apply^*\ M\ N)\ \ T) \\
\quad (\langle typeof^*\ M\ (arr^*\ U\ T)\rangle^* ::^* \langle typeof^*\ N\ U\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ (typeof^*_{instr}\ \ (new\_ref^*\ M)\ \ (refty^*\ T)) \\
\quad (\langle typeof^*\ M\ T\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ (typeof^*_{instr}\ \ (lookup^*\ M)\ \ T) \\
\quad (\langle typeof^*\ M\ (refty^*\ T)\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ (typeof^*_{instr}\ \ (eval\_rvalue^*\ M\ N)\ \ T) \\
\quad (\langle typeof^*\ M\ (refty^*\ T)\rangle^* ::^* \langle typeof^*\ N\ T\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ (typeof^*_{instr}\ \ (update^*\ M\ N)\ \ T) \\
\quad (\langle typeof^*\ M\ (refty^*\ T)\rangle^* ::^* \langle typeof^*\ N\ T\rangle^* ::^* nil^*) \qquad nil^* \\
\\
prog\ (typeof^*_{ans}\ \ (answer^*\ V\ S)\ \ T) \\
\quad (\langle typeof^*\ V\ T\rangle^* ::^* \langle well\_typed^*\ S\rangle^* ::^* nil^*) \qquad nil^* \\
prog\ (typeof^*_{ans}\ \ (new^*\ F)\ \ T) \\
\quad \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^*) \\
\quad nil^* \qquad nil^* \\
prog\ (well\_typed^*\ (extend\_st^*\ C\ V\ S)) \\
\quad (\langle typeof^*\ (cell^*\ C)\ (refty^*\ T)\rangle^* ::^* \langle typeof^*\ V\ T\rangle^* ::^* \langle well\_typed^*\ S\rangle^* ::^* nil^*) \\
\quad nil^*
\end{array}
$$

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 $\text{PCF}_{:=}$. We will refer to the collected clauses of Tables XVII, XVIII, XIX and XX as the definition $\mathcal{D}(\text{PCF}_{:=})$. To simplify the presentation of our theorems, we introduce several $FO\lambda^{\Delta\mathbb{N}}$ predicates:

$$
\begin{array}{rlrl}
store & : atmlst^* \to o & \equiv_{atml} & : atm^* \to atm^* \to o \\
store\_typing & : atmlst^* \to o & \equiv^*_i & : i^* \to i^* \to o \\
store\_typeof & : atmlst^* \to atmlst^* \to o\ .
\end{array}
$$

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 $contains^*\ c\ v$. 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

Table XXI. Meta-logic predicates for $\text{PCF}_{:=}$ stores

$$
\begin{array}{rcl}
\mathit{store}\ LL & \triangleq & \mathit{list}\ LL \wedge \forall a(\mathit{element}\ a\ LL \supset \\
& & \qquad \exists c \exists v(a \equiv_{atm^*} (\mathit{contains}^*\ c\ v))) \\
\\
\mathit{store\_typing}\ IL & \triangleq & \mathit{list}\ IL \wedge \\
& & \forall a(\mathit{element}\ a\ IL \supset \\
& & \quad \exists c \exists t(a \equiv_{atm^*} (\mathit{typeof}^*\ (\mathit{cell}^*\ c)\ (\mathit{refty}^*\ t)))) \wedge \\
& & \forall c \forall t_1 \forall t_2(\mathit{element}\ (\mathit{typeof}^*\ (\mathit{cell}^*\ c)\ (\mathit{refty}^*\ t_1))\ IL \supset \\
& & \qquad \mathit{element}\ (\mathit{typeof}^*\ (\mathit{cell}^*\ c)\ (\mathit{refty}^*\ t_2))\ IL \supset \\
& & \qquad t_1 \equiv_{i^*} t_2) \\
\\
\mathit{store\_typeof}\ LL\ IL & \triangleq & \forall c \forall v(\mathit{element}\ (\mathit{contains}^*\ c\ v)\ LL \supset \\
& & \qquad \exists t(\mathit{element}\ (\mathit{typeof}^*\ (\mathit{cell}^*\ c)\ (\mathit{refty}^*\ t))\ IL \wedge \\
& & \qquad\ IL; \mathit{nil}^* \rhd \langle \mathit{typeof}^*\ v\ t \rangle^*)) \\
\\
A \equiv_{atm^*} A & \triangleq & \top \\
\\
X \equiv_{i^*} X & \triangleq & \top
\end{array}
$$

stored in the location. Finally, $\equiv_{atml}$ and $\equiv_{itml}$ encode syntactic identity over the types atm^* and i^*. The definition $\mathcal{D}(\mathit{store})$ for these predicates is presented in Table XXI. The following theorem states that we have derived the subject reduction and unicity of typing properties for $\text{PCF}_{:=}$ 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: $\mathcal{D}(\mathit{lists})$ for

$$\mathcal{D}(\mathit{list}^*(\mathrm{atm})) \cup \mathcal{D}(\mathit{list}^*(\mathrm{prp})) \cup \mathcal{D}(\mathit{list}^{**}(\mathrm{atm})) \cup \mathcal{D}(\mathit{list}^{**}(\mathrm{prp}))\ ,$$

and $\mathcal{D}(\mathit{evars})$ for

$$\mathcal{D}(\mathit{evars}(\mathrm{atm})) \cup \mathcal{D}(\mathit{evars}(\mathrm{prp})) \cup \mathcal{D}(\mathit{evars}(\mathrm{atmlst})) \cup \mathcal{D}(\mathit{evars}(\mathrm{prplst}))\ .$$

THEOREM 9.1. *The following formulas are derivable in $FO\lambda^{\Delta\mathbb{N}}$ from the definition that accumulates $\mathcal{D}(\mathit{nat})$, $\mathcal{D}(\mathit{lists})$, $\mathcal{D}(\mathit{evars})$, $\mathcal{D}(\mathit{linear})$, $\mathcal{D}(\text{PCF}_{:=})$, and $\mathcal{D}(\mathit{store})$:*

Subject reduction:

$$
\begin{array}{l}
\forall m \forall s \forall f(\rhd(\langle (m, s) \Downarrow^*\ f \rangle^*)\ \supset \\
\qquad \forall ts \forall t(\mathit{store\_typing}\ il\ \supset\ il; \mathit{nil}^* \rhd \langle \mathit{well\_typed}^*\ s \rangle^*\ \supset \\
\qquad\quad il; \mathit{nil}^* \rhd \langle \mathit{typeof}^*\ m\ t \rangle^*\ \supset \\
\qquad\quad il; \mathit{nil}^* \rhd \langle \mathit{typeof}^*_{ans}\ f\ t \rangle^*))
\end{array}
$$

$$
\begin{array}{l}
\forall ll \forall k \forall i \forall f(\mathit{store}\ ll\ \supset\ \mathit{nil}^*; ll \rhd \langle \mathit{ns\_mach\_2}^*\ k\ i\ f \rangle^*\ \supset \\
\qquad \forall il \forall t \forall u(\mathit{store\_typing}\ il\ \supset\ \mathit{store\_typeof}\ ll\ il\ \supset \\
\qquad\quad il; \mathit{nil}^* \rhd \langle \mathit{typeof}^*_{cntn}\ k\ (\mathrm{arr}^*\ t\ u) \rangle^*\ \supset \\
\qquad\quad il; \mathit{nil}^* \rhd \langle \mathit{typeof}^*_{instr}\ i\ t \rangle^*\ \supset \\
\qquad\quad il; \mathit{nil}^* \rhd \langle \mathit{typeof}^*_{ans}\ f\ u \rangle^*))
\end{array}
$$

Unicity of typing:

$$\forall m \forall t_1 \forall t_2 (\rhd \langle typeof^* \; m \; t_1 \rangle^* \supset \rhd \langle typeof^* \; m \; t_2 \rangle^* \supset t_1 \equiv_{i^*} t_2)$$

PROOF. The derivation of the unicity of typing is by complete induction on the height of the first typing derivation $\rhd \langle typeof^* \; m \; t_1 \rangle^*$. 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

$$\begin{array}{l} \lambda j \forall il (list \; il \supset P_1 \; il \supset P_2 \; il \supset \\ \quad \forall m \forall t_1 \forall t_2 (seq_j \; il \; nil^* \; \langle typeof^* \; m \; t_1 \rangle^* \supset \\ \quad\quad il; nil^* \rhd \langle typeof^* \; m \; t_2 \rangle^* \supset t_1 \equiv_{i^*} t_2)) \; . \end{array}$$

The details of the proof are presented in McDowell [1997]. □

10. 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 in [1994; 1995] 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

$$var : vr \to tm \qquad abs : (vr \to tm) \to tm \qquad app : tm \to tm \to tm \; ,$$

and the corresponding definition of *typeof* would be

$$typeof_{vr} : vr \to ty \to o \qquad typeof : tm \to ty \to o$$

$$\begin{array}{rcl} typeof \; (var \; X) \; T & \triangleq & typeof_{vr} \; X \; T \\ typeof \; (abs \; M) \; (arr \; T \; U) & \triangleq & \forall x (typeof_{vr} \; x \; T \supset typeof \; (M \; x) \; U) \\ typeof \; (app \; M \; N) \; T & \triangleq & \exists u (typeof \; M \; (arr \; u \; T) \wedge typeof \; N \; u) \; . \end{array}$$

This is similar to our use of the two predicates *hyp* and *conc* in our encoding of intuitionistic logic in Section 4.2. 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.[1] These approaches also lessen the power

[1]Here we are comparing the object system encodings. It is true that our explicit eigenvariable encoding style requires an explicit definition of substitution *for the specification logic*. So at the

of the meta-level cut rule as a reasoning tool. Suppose that $\forall x(\mathit{typeof}_{vr}\ x\ T \supset \mathit{typeof}\ (M\,x)\ U)$ 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 $(vr \to tm)$ 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. The term language of $FO\lambda^{\Delta\mathbb{N}}$, unlike that of Coq, does not include primitive recursion, so these exotic terms do not arise in our framework.

Despeyroux, Pfenning, and Schürmann [1997] 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 $\to$ 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 and Pfenning [1998] construct a meta-logic $\mathcal{M}_2$ to reason about deductive systems represented in LF. Their 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. The meta-logic $\mathcal{M}_2$ includes a case-analysis rule comparable to our $\mathit{def}\mathcal{L}$ rule and a recursion rule that generalizes our $\mathit{nat}\mathcal{L}$ rule. Their intermediate logic, LF, includes dependent types, and so is richer than the intermediate logics we consider. On the other hand, our meta-logic is a general framework capable of supporting a variety of intermediate logics (such as intuitutionistic and linear logics), whereas $\mathcal{M}_2$ is designed for the specific, fixed intermediate logic LF.

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 [1992; 1996] investigate the design and implementation of such external validity conditions.

---

specification logic level of our framework, we too lose some of the benefits of higher-order abstract syntax. However, at the level of the object system, we use a true higher-order abstract syntax encoding with all of its benefits. Since we expect there to be only a few specification logics, but many object systems, it seems worth putting the extra effort into the specification logic to reap the benefit for the object systems.

Matthews seeks to reconcile the advantages of LF-style encodings with the facilities for meta-theoretic analysis found in theories of inductive definitions [Matthews 1997]. 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.

11. CONCLUSION

In this paper we have presented a single and simply motivated meta-logic $FO\lambda^{\Delta\mathbb{N}}$. We used this meta-logic as the basis of 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.

The meta-logic $FO\lambda^{\Delta\mathbb{N}}$ has also been used to reason about simulation and bisimulation in abstract transition systems and CCS [McDowell et al.]. These transition systems did not contain binding operators, and so both the specification and reasoning was done in the meta-logic. We have already begun using the techniques presented in the current paper to extend that work to the setting of applicative bisimulation [Abramsky 1990]. It would also be interesting to use Howe's technique [Howe 1996] to prove the congruence of bisimulation in our framework.

Additional work in analysis of programming languages along the lines of Part III 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 [1992], for example, construct abstract machines from operational semantics by applying a series of transformations and argue informally that the transformations preserve correctness. Richer languages could also be considered, including features such as concurrency, exceptions, and polymorphism. Linear logic has been used to specify such features in a manner that is suitable for use in our setting [Chirimar 1995; Miller 1996].

The formal derivations described in this paper have been checked using the Pi derivation editor of Lars-Henrik Eriksson [Eriksson 1994]; see McDowell [1997] for a discussion of the effectiveness of this editor for constructing $FO\lambda^{\Delta\mathbb{N}}$ proofs. An important next step in this line of work is to implement a theorem prover that provides semi-automated assistance in proving $FO\lambda^{\Delta\mathbb{N}}$ theorems. Miller and Wajs are building a prototype theorem prover named Iris [Wajs 2000] within λProlog.

Finally, alternatives to the explicit eigenvariable encoding of Section 4.4 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 preserved.

ACKNOWLEDGMENTS

We would like to thank Frank Pfenning for helpful feedback on early drafts of this work and Lars-Henrik Eriksson for making his Pi derivation editor [Eriksson 1994] available to help check the formal derivations described here. Two anonymous referees provided extensive comments that helped improve the presentation of this paper.

REFERENCES

Abramsky, S. 1990. The lazy lambda calculus. In *Research Topics in Functional Programming*, D. Turner, Ed. Addison Wesley, 65–117.

Avron, A., Honsell, F., Mason, I. A., and Pollack, R. 1992. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning 9*, 309–354.

Basin, D. A. and Constable, R. L. 1993. Metalogical frameworks. In *Logical Environments*, G. Huet and G. D. Plotkin, Eds. Cambridge University Press, 1–29.

Cervesato, I. and Pfenning, F. 1996. A linear logic framework. In *Proceedings, Eleventh Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, New Brunswick, New Jersey, 264–275. An extended version of this paper will appear in *Information and Computation*.

Chirimar, J. 1995. Proof theoretic approach to specification languages. Ph.D. thesis, University of Pennsylvania.

Church, A. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic 5*, 56–68.

Despeyroux, J., Felty, A., and Hirschowitz, A. 1995. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, M. Dezani-Ciancaglini and G. Plotkin, Eds. Lecture Notes in Computer Science, vol. 902. Springer-Verlag, 124–138.

Despeyroux, J. and Hirschowitz, A. 1994. Higher-order abstract syntax with induction in Coq. In *Proceedings of the Fifth International Conference on Logic Programming and Automated Reasoning*, F. Pfenning, Ed. Lecture Notes in Artificial Intelligence, vol. 822. Springer-Verlag, 159–173.

Despeyroux, J., Pfenning, F., and Schürmann, C. 1997. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications*, R. Hindley, Ed.

Eriksson, L.-H. 1991. A finitary version of the calculus of partial inductive definitions. In *Proceedings of the Second International Workshop on Extensions to Logic Programming*, L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, Eds. Lecture Notes in Artificial Intelligence, vol. 596. Springer-Verlag, 89–134.

Eriksson, L.-H. 1993. Finitary partial inductive definitions as a general logic. In *Proceedings of the Fourth International Workshop on Extensions to Logic Programming*. Lecture Notes in Artificial Intelligence, vol. 798. Springer-Verlag, 94–119.

Eriksson, L.-H. 1994. Pi: an interactive derivation editor for the calculus of partial inductive definitions. In *Proceedings of the Twelfth International Conference on Automated Deduction*, A. Bundy, Ed. Lecture Notes in Artificial Intelligence, vol. 814. Springer-Verlag, 821–825.

Felty, A. 1993. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning 11,* 1 (August), 43–81.

Felty, A. and Miller, D. 1988. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, E. Lusk and R. Overbeck, Eds. Springer-Verlag, 61–80.

Girard, J.-Y. 1992. A fixpoint theorem in linear logic. A message posted on the mailing list `linear@cs.stanford.edu`, see `http://www.csl.sri.com/linear/mailing-list-traffic/www/07/mail_3.html`.

Gunter, C. A. 1992. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press.

Hallnäs, L. 1991. Partial inductive definitions. *Theor. Comput. Sci. 87*, 115–142.

Hannan, J. and Miller, D. 1992. From operational semantics to abstract machines. *Math. Struct. Comput. Sci. 2,* 4, 415–459.

Hannan, J. J. 1990. Investigating a proof-theoretic meta-language for functional programs. Ph.D. thesis, University of Pennsylvania.

Harper, R., Honsell, F., and Plotkin, G. 1993. A framework for defining logics. *Journal of the ACM 40,* 1, 143–184.

Hodas, J. and Miller, D. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation 110,* 2, 327–365.

Howe, D. J. 1996. Proving congruence of bisimulation in functional programming languages. *Information and Computation 124,* 2, 103–112.

Huet, G. 1975. A unification algorithm for typed λ-calculus. *Theor. Comput. Sci. 1*, 27–57.

Magnusson, L. and Nordström, B. 1994. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, H. Barendregt and T. Nipkow, Eds. Number 806 in Lecture Notes in Computer Science. Springer-Verlag, 213–237.

Matthews, S. 1997. A practical implementation of simple consequence relations using inductive definitions. In *Proceedings of the 14th Conference on Automated Deduction*, W. McCune, Ed. Springer-Verlag.

Matthews, S., Smaill, A., and Basin, D. 1993. Experience with FS_0 as a framework theory. In *Logical Environments*, G. Huet and G. Plotkin, Eds. Cambridge University Press, 61–82.

McDowell, R. 1997. Reasoning in a logic with definitions and induction. Ph.D. thesis, University of Pennsylvania.

McDowell, R. and Miller, D. 2000. Cut elimination for a logic with definitions and induction. *Theor. Comput. Sci. 232*, 91–119.

McDowell, R., Miller, D., and Palamidessi, C. Encoding transition systems in sequent calculus. To appear in *Theoretical Computer Science*. Preliminary version appeared as [McDowell et al. 1996].

McDowell, R., Miller, D., and Palamidessi, C. 1996. Encoding transition systems in sequent calculus: Preliminary report. In *Proceedings of the 1996 Workshop on Linear Logic*. Electronic Notes in Theoretical Computer Science, vol. 3. Elsevier.

Miller, D. 1990. Abstractions in logic programs. In *Logic and Computer Science*, P. Odifreddi, Ed. Academic Press, 329–359.

Miller, D. 1991. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Comput. 1,* 4, 497–536.

Miller, D. 1996. Forum: A multiple-conclusion specification language. *Theor. Comput. Sci. 165*, 201–232.

Miller, D. and Nadathur, G. 1987. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming*, S. Haridi, Ed. 379–388.

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. 1991. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic 51*, 125–157.

Mitchell, J. C. 1995. *Foundations for Programming Languages*. MIT Press.

Paulson, L. C. 1986. Natural deduction as higher-order resolution. *J. Logic Program. 3*, 237–258.

Pfenning, F. 1989. Elf: A language for logic definition and verified metaprogramming. In *Proceedings,Fourth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 313–321.

Pfenning, F. 1995. Structural cut elimination. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, D. Kozen, Ed. IEEE Computer Society Press, 156–166.

Pfenning, F. and Elliot, C. 1988. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*. 199–208.

Pfenning, F. and Rohwedder, E. 1992. Implementing the meta-theory of deductive systems. In *Proceedings of the Eleventh International Conference on Automated Deduction*, D. Kapur, Ed. Lecture Notes in Artificial Intelligence, vol. 607. Springer-Verlag, 537–551.

Rohwedder, E. and Pfenning, F. 1996. Mode and termination analysis for higher-order logic programs. In *Proceedings of the European Symposium on Programming*. 296–310.

Schroeder-Heister, P. 1992. Cut-elimination in logics with definitional reflection. In *Nonclassical Logics and Information Processing*, D. Pearce and H. Wansing, Eds. Lecture Notes in Computer Science, vol. 619. Springer-Verlag, 146–171.

Schroeder-Heister, P. 1993. Rules of definitional reflection. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, M. Vardi, Ed. IEEE Computer Society Press, 222–232.

Schürmann, C. and Pfenning, F. 1998. Automated theorem proving in a simple meta logic for LF. In *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, C. Kirchner and H. Kirchner, Eds. Lecture Notes in Computer Science, vol. 1421. Springer-Verlag, 286–300.

Scott, D. S. 1969. A type theoretical alternative to CUCH, ISWIM, OWHY. Unpublished manuscript.

VanInwegen, M. 1996. The machine-assisted proof of programming language properties. Ph.D. thesis, University of Pennsylvania.

Wajs, J. D. 2000. Design and implementation of a theorem prover for operational semantics. M.S. thesis, Pennsylvania State University.

Received March 2000; revised January 2001; accepted January 2001