A Logic for Reasoning with Higher-Order Abstract Syntax

Raymond McDowell and Dale Miller
Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19104-6389 USA
mcdowell@saul.cis.upenn.edu, dale@saul.cis.upenn.edu

Abstract

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 the subject-reduction theorem holds. 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 a definition is a proof-theoretic device that allows certain theories to be treated as "closed" or as defining fixed points. The resulting meta-logic can specify various logical frameworks and a large range of judgments regarding programming languages and inference systems. We illustrate this point through examples, including the admissibility of cut for a simple logic and subject reduction, determinacy of evaluation, and the equivalence of SOS and natural semantics presentations of evaluation for a simple functional programming language.*

1. Introduction

Meta-logics and type systems have been used to specify the semantics of a wide range of logics and computation systems [2, 4, 11, 34]. 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 [28, 33]. 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 [3, 19, 21, 38]. 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 [38]. 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 [14] 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, [12, 31] for some early references) and corresponds to the structure of common informal reasoning.

In the next section we present our meta-logic and motivate its design through an informal proof of subject reduction for the untyped λ-calculus. We proceed in Section 3 to use this meta-logic to define an object-logic and describe some of its meta-theory. Section 4 contains a specification in the object-logic of the dynamic and static semantics for a simple functional programming language. We also list a variety of theorems about the language that we have proved in our meta-logic. Finally, Section 5 discusses some other research with similar goals to our own, and Section 6 summarizes our accomplishments and plans for continuing the work.

2. Designing the meta-logic

In this paper we use an intuitionistic logic for our meta-logic; in particular, we start with an intuitionistic version of a subset of Church's Simple Theory of Types [5] (assuming $\beta\eta$-conversion for the equality of terms). Formulas will have the type o, the logical constants for true and false are $\top$ and $\bot$, for conjunction and implication are $\wedge$ and $\supset$, and universal and existential quantification at type τ are $\forall_\tau$ and $\exists_\tau$. In this paper, τ will not contain o and will be either of primitive type or of order 1.

We use the following facts about cut-free intuitionistic provability of sequents involving just these connectives [29]. Let $\mathcal{P}$ be a finite set of formulas and let B, B_1, B_2 be formulas. The sequent $\mathcal{P} \longrightarrow B_1 \wedge B_2$ is provable if and only if the sequents $\mathcal{P} \longrightarrow B_1$ and $\mathcal{P} \longrightarrow B_2$ are provable, and the sequent $\mathcal{P} \longrightarrow B_1 \supset B_2$ is provable if and only if the sequent $\mathcal{P}, B_1 \longrightarrow B_2$ is provable. Furthermore, if $\mathcal{P}$ does not contain any positive occurrences of an existential quantifier, then the sequent $\mathcal{P} \longrightarrow \forall_\tau x.B$ is provable if and only if the sequent $\mathcal{P} \longrightarrow B[y/x]$ is provable, where y is some (eigen)variable that does not free occur in $\mathcal{P}$ or in $\forall_\tau x.B$; the sequent $\mathcal{P} \longrightarrow \exists_\tau x.B$ is provable if and only if the sequent $\mathcal{P} \longrightarrow B[t/x]$ is provable for some term t of type τ. Finally, if A is an atomic formula, then the sequent $\mathcal{P} \longrightarrow A$ is provable if and only if it is possible to backchain on a formula in $\mathcal{P}$: backchaining can be described as the process of repeatedly applying left-introduction rules to a given formula in $\mathcal{P}$ and its positive subformulas until the atom A is exposed. In the particular case of formulas of the form $\forall\bar{x}(G_1 \wedge \cdots \wedge G_n \supset A')$ ($n \geq 0$), backchaining involves finding a substitution θ for the variables $\bar{x}$ such that $A'\theta$ equals A and the sequents $\mathcal{P} \longrightarrow G_i\theta$ are provable for all $i = 1, \ldots, n$. We write $\mathcal{P} \vdash B$ whenever the sequent $\mathcal{P} \longrightarrow B$ has a cut-free proof.

2.1. Motivation from informal reasoning

In order to motivate the extensions to the core of the meta-logic presented above, we consider a specification of call-by-name evaluation and simple typing for the untyped λ-calculus. To do this, we will find it useful to distinguish between meta-level and object-level structures. For example, at the meta-level we introduce two types, *tm* and *ty*, to denote object-level terms and types. To represent the untyped λ-terms we introduce the two meta-level constants abs of type $(\mathit{tm} \rightarrow \mathit{tm}) \rightarrow \mathit{tm}$ and app of type $\mathit{tm} \rightarrow \mathit{tm} \rightarrow \mathit{tm}$ to denote object-level abstraction and application, respectively. Using such a coding places α-equivalence classes of object-level terms in one-to-one correspondence with $\beta\eta$-equivalence classes in the meta-level. Object-level types will be built up from a single primitive type using the arrow type constructor; these are denoted at the meta-level by the constants gnd of type *ty* and arr of type $\mathit{ty} \rightarrow \mathit{ty} \rightarrow \mathit{ty}$.

To specify call-by-name evaluation, we use an infix predicate $\Downarrow$ of type $\mathit{tm} \rightarrow \mathit{tm} \rightarrow o$ and the two formulas

$$\forall r.[(abs\ r) \Downarrow (abs\ r)]$$
$$\forall m, n, v, r.[m \Downarrow (abs\ r)\ \wedge\ (r\ n) \Downarrow v\ \supset\ (app\ m\ n) \Downarrow v]$$

(Here we took the liberty of abbreviating a list of universal quantifiers as a universal quantifier of a list of variables. We also dropped the type subscript on quantifiers since the context makes their type clear: here, all variables are of type *tm* except for r which is of type $\mathit{tm} \rightarrow \mathit{tm}$.) Meta-level β-reduction on instances of $(r\ n)$ will perform the substitution of the term n into the abstraction r.

To specify simple typing at the object-level, we use the binary predicate $typeof$ of meta-level type $\mathit{tm} \rightarrow \mathit{ty} \rightarrow o$ and the two formulas

$$\forall m, n, t, u.[typeof\ m\ (arr\ u\ t)\ \wedge\ typeof\ n\ u$$
$$\supset\ typeof\ (app\ m\ n)\ t]$$
$$\forall r, t, u.[\forall x.[typeof\ x\ t\ \supset\ typeof\ (r\ x)\ u]$$
$$\supset\ typeof\ (abs\ r)\ (arr\ t\ u)]$$

Here, notice that the meta-level use of implication and universal quantification with the typing rule for *abs* provides an elegant management of the typing of object-level bound variables. 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, [2, 34].

Now consider the following theorem and its proof. To simplify the presentation we omit displaying on the left of the turnstile the above formulas encoding evaluation and typing.

Theorem 2.1 *If P evaluates to V and P has type T then V has type T.*

Proof Given our specifications above, we prove this theorem by proving by induction on the height of the proof of $P \Downarrow V$ that for all T, if $\vdash \mathit{typeof}\ P\ T$ then $\vdash \mathit{typeof}\ V\ T$. Since $P \Downarrow V$ is atomic, its proof must end by backchaining on one of the formulas encoding evaluation. If the backchaining is on the *eval* formula for *abs*, then P and V are both equal to $abs\ R$, for some R, and the consequent is immediate. If $P \Downarrow V$ was proved using the *eval* formula for *app*, then P is of the form $app\ M\ N$ and for some R, there are shorter proofs of $M \Downarrow (abs\ R)$ and $(R\,N) \Downarrow V$. Since $\vdash \mathit{typeof}\ (app\ M\ N)\ T$, this typing judgment must have been proved using backchaining and, hence, there is a U such that $\vdash \mathit{typeof}\ M\ (arr\ U\ T)$ and $\vdash \mathit{typeof}\ N\ U$. Using the inductive hypothesis, we have $\vdash \mathit{typeof}\ (abs\ R)\ (arr\ U\ T)$. This atomic formula must have been proved by backchaining on the *typeof* formula for *abs*, and, hence, $\vdash \forall x.[\mathit{typeof}\ x\ U \supset \mathit{typeof}\ (R\,x)\ T]$. Since our logic of judgments is intuitionistic logic, we can instantiate this quantifier with N and use cut and cut-elimination to conclude that $\vdash \mathit{typeof}\ (R\,N)\ T$. Using the inductive hypothesis a second time yields $\vdash \mathit{typeof}\ V\ T$. ▮

This proof is clear and natural, and we would like our meta-logic to allow proofs quite similar to this in structure. This suggests that the following features would be valuable in the meta-logic.

1. *Two distinct logics.* One of the logics would correspond to the one written with logical syntax above and would capture judgments, e.g. about typability and evaluation. The second logic would represent a formalization of the English text in the proof above. Atomic formulas of that logic would be judgments in the object-logic.

2. *Induction* over at least natural numbers.

3. *Instantiation of meta-level eigenvariables.* In the proof above, for example, the meta-level variable P was instantiated in one part of the proof to $abs\ R$ and in another part of the proof to $app\ M\ N$. Notice that this instantiation of eigenvariables within a proof does not happen in a strictly intuitionistic sequent calculus.

4. *Analysis of the proof of an assumed judgment.* In the proof above this was done a few times, leading, for example, from the assumption

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

to the assumption

$$\vdash \forall x.[\mathit{typeof}\ x\ U \supset \mathit{typeof}\ (R\,x)\ T].$$

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 meta-logic, we accommodate the first two features by specifying an object-logic within the meta-logic and by introducing natural numbers and induction. The last two features are accommodated by introducing a notion of *definition* and two sequent calculus rules for the left and right introduction of defined concepts. We address this latter notion first.

2.2. Definitions

Definitions will be written in the following style.

$$\forall \bar{x}_1.[p_1(\bar{t}_1) \triangleq H_1] \ \cdots\ \forall \bar{x}_n.[p_n(\bar{t}_n) \triangleq H_n] \quad (n \geq 0)$$

For $i = 1, \ldots, n$, p_i is a predicate constant, every free variable of the formula H_i is also free in at least one term in the list $\bar{t}_i$ of terms, and all variables free in $\bar{t}_i$ are contained in the list $\bar{x}_i$ of variables. The expression $\forall \bar{x}_i.[p_i(\bar{t}_i) \triangleq H_i]$ is a *clause* of the definition and H_i is the *body* and $p_i(\bar{t}_i)$ is the *head* of that clause. The symbol $\triangleq$ is used simply to indicate definitions: it is not a logical connective. We do not assume that the predicates $p_1, \ldots, p_n$ are distinct: it is best to think of a definition as a mutual recursive definition of predicates that are in the set $\{p_1, \ldots, p_n\}$. In this paper we only consider definitions containing a finite number of clauses; in other work, however, infinite definitions play an important role [25].

For the cut rule to be eliminable from our meta-logic (Theorem 2.2), it is necessary to place further restrictions on the form of definitions [36]. Schroeder-Heister shows that it is sufficient to prohibit the use of implication in clause bodies [36], and we adopt this solution here. It is possible to loosen this restriction by either restricting occurrences of the modal operators ! and ? in a linear logic setting [13] or stratifying the defined predicates [23], but we shall not require such flexibility in this paper.

The right-introduction rule for defined atoms is

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

where θ is a substitution of terms for variables and Γ is a multiset of formulas. This rule corresponds to the logic programming notion of *backchaining* if we think of $\triangleq$ in definitions as reverse implication.

The left-introduction rule for defined concepts uses complete sets of unifiers (CSU) [18]:

$$\frac{\left\{ H\theta, \Gamma\theta \longrightarrow B\theta \;\middle|\; \begin{array}{l}\theta \in CSU(p\,\bar{u}, p\,\bar{t}) \text{ for}\\ \text{some clause } \forall \bar{x}.[p\,\bar{t} \triangleq H]\end{array} \right\}}{p\,\bar{u}, \Gamma \longrightarrow B}\ \mathit{def}\mathcal{L}$$

where θ is a substitution of terms for variables, Γ is a multiset of formulas, B is a formula, and the variables $\bar{x}$ are chosen to be distinct from the variables free in the lower sequent of the rule. Specifying a set of sequents as the premise means that each sequent in the set is a premise of the rule.

Notice that the number of premises of the $def\mathcal{L}$ 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 and $p\,\bar{u}$, which is unprovable, is treated as false by this 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 [26]. 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.

This left-introduction rule is similar to *definitional reflection* [36] (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 [13]. This particular presentation of the rule is due to Eriksson [9]. Notice that in the $def\mathcal{L}$ rule, the free variables of the conclusion can be instantiated in the premises (see item 3 in the list of desired meta-logic features).

2.3. Natural number induction

We incorporate induction by introducing natural numbers using $z : nt$ for zero and $s : nt \rightarrow nt$ for successor and using the predicate $nat : nt \rightarrow o$. The rules for this new predicate are

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

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

Here, I, B, and C are schematic variables of these inference rules, and j is a variable not free in B. The first two rules can be seen as right-introduction rules for nat while the third rule, encoding induction over natural numbers, can be seen as a left-introduction rule. In the left-introduction rule, B ranges over formulas with one variable extracted (say, using λ-abstraction) and represents the property that is proved by induction; the third premise of that inference rule witnesses the fact that, in general, B will express a property stronger than C.

2.4. $FO\lambda^{\Delta\mathbb{N}}$

The extension of intuitionistic logic that results from adding the rules for definitions and natural numbers we call $FO\lambda^{\Delta\mathbb{N}}$, an acronym for "first-order logic for λ with definitions and natural numbers". Assuming that a definition is given and fixed, we have the following results.

Theorem 2.2 (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 in [36] 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 [22, 23] and is modeled on proofs by Tait and Martin-Löf that use the technical notions of normalizability and computability. ▮

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

Corollary 2.3 (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 invariant formula 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.

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

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

As our first example of a theorem in our meta-logic, we derive a complete induction principle.

Theorem 2.4 (Complete induction) *For any formula $C : o$ and predicate $B : nt \rightarrow o$, the formula*

$$\forall j.[nat\ j \supset \forall k.[nat\ k \supset k < j \supset B\,k] \supset B\,j]$$
$$\supset \forall i.[(B\,i \supset C) \supset (nat\ i \supset C)]$$

is provable in $FO\lambda^{\Delta\mathbb{N}}$, where $<$ is defined by the clauses

$$z < (s\ J) \triangleq nat\ J \qquad (s\ I) < (s\ J) \triangleq I < J$$

(In the definition of $<$, we have not shown the quantification of the variables I and J around the clauses. Throughout this paper we will implicitly assume the universal closure of all definitional clauses.)

We now take the meta-logic $FO\lambda^{\Delta\mathbb{N}}$ as our logical framework. One can imagine adding stronger induction principles, such as transfinite induction, but we will not, in fact, need such a principle for a great many of the theorems that we wish to prove in the area of programming languages and deductive systems. Many forms of induction, such as structural induction and induction on the height of object-level proofs (as used in the proof of Theorem 2.1 above) are simple derived rules of $FO\lambda^{\Delta\mathbb{N}}$ and do not need to be considered as extensions to this logic.

We may, however, wish to have many different object-logics to reason about. We now discuss how an object-level logic can be accommodated inside $FO\lambda^{\Delta\mathbb{N}}$.

3. Representing an object-logic

Looking back to the informal proof of subject reduction (Theorem 2.1), the first observation stated that we needed to have two logics, which, in fact, means that we need to have three "languages": the meta-logic for reasoning and inductive proofs ($FO\lambda^{\Delta\mathbb{N}}$), the object-logic of judgments, and finally the language of untyped λ-terms and types for them. We shall now define a simple object-level language that is capable of representing a large number of judgments regarding programming systems and deductive systems. This logic, a second-order fragment of minimal logic, is encoded using the two meta-logic types *atm* for atoms (atomic propositions) and *prp* for general propositions and the following constants.

$$\begin{array}{rcl}
\langle\ \rangle & : & atm \rightarrow prp \\
1 & : & prp \\
\& & : & prp \rightarrow prp \rightarrow prp \\
\Rightarrow & : & atm \rightarrow prp \rightarrow prp \\
\bigwedge_\tau & : & (\tau \rightarrow prp) \rightarrow prp \\
\bigvee_\tau & : & (\tau \rightarrow prp) \rightarrow prp
\end{array}$$

We shall use the type *i* to denote the ground type for terms in our object-logic (e.g. the types *tm* and *ty* of our example will both be mapped to *i*). The syntactic variable τ above, representing the object-logic quantification types, will be restricted to range over types built from *i* and $\rightarrow$. The constant $\langle\ \rangle$ coerces atoms into propositions: object-level predicates (atomic judgments) will be constants that build meta-level terms of type *atm*. There are few meta-level predicates that we need to deal with provability at the object-logic. These are given below. Since object-level sequents require lists of atomic propositions, we also introduce the type *atm_lst* and two constructors *nil* and :: for building lists.

$$\begin{array}{rcl}
nil & : & atm\_lst \\
:: & : & atm \rightarrow atm\_lst \rightarrow atm\_lst \\
 & & \\
prog & : & atm \rightarrow prp \rightarrow o \\
seq & : & nt \rightarrow atm\_lst \rightarrow prp \rightarrow o \\
element & : & atm \rightarrow atm\_lst \rightarrow o
\end{array}$$

The meta-level atomic formula $prog\ A\ B$ will encode the fact that the universal closure of $B \Rightarrow A$ is part of the object-level theory. The predicate *seq* represents object-level derivability of a sequent with respect to the theory stored in the *prog* clauses. The first argument is used as an induction measure and is written as a subscript for convenience. Finally, the predicate *element* represents list membership. The definition $\mathcal{D}(seq)$ for *seq* and *element* is

$$\begin{array}{rcl}
seq_{(s\ I)}\ L\ \langle A\rangle & \triangleq & \exists b.[prog\ A\ b \wedge seq_I\ L\ b] \\
seq_I\ (A' :: L)\ \langle A\rangle & \triangleq & element\ A\ (A' :: L) \\
seq_I\ L\ 1 & \triangleq & \top \\
seq_{(s\ I)}\ L\ (B \,\&\, C) & \triangleq & seq_I\ L\ B \wedge seq_I\ L\ C \\
seq_{(s\ I)}\ L\ (A \Rightarrow B) & \triangleq & seq_I\ (A :: L)\ B \\
seq_{(s\ I)}\ L\ (\bigwedge_\tau B) & \triangleq & \forall_\tau x.[seq_I\ L\ (B\, x)] \\
seq_{(s\ I)}\ L\ (\bigvee_\tau B) & \triangleq & \exists_\tau x.[seq_I\ L\ (B\, x)] \\
 & & \\
element\ A\ (A :: L) & \triangleq & \top \\
element\ A\ (A' :: L) & \triangleq & element\ A\ L
\end{array}$$

The object-level theory declared by *prog* will vary according to the logic specification under consideration, as illustrated in the next section. At the object-level, a specification is used as a theory and not as a definition: there are no definitions involved at the object-level. The clauses shown for $\bigwedge$ and $\bigvee$ are actually schemas giving the form of the clauses for any type τ. Including instances of the schemas for all types would result in an infinite number of clauses. For any application, however, we will only need a finite number of instances; for the examples in this paper we need only consider the types i and $i \rightarrow i$. 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 $\hat{B}$ be its (obvious) translation into a formula of intuitionistic logic. If L is a term of type atm_lst, let $\hat{L}$ be its (obvious) translation to a multiset of atomic formulas of intuitionistic logic. The following adequacy result follows from the cut-elimination theorem for intuitionistic logic and the restriction to clausal second-order clauses.

Theorem 3.1 (Adequacy) *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 logic specification, and let $\mathcal{P}$ be the corresponding logic specification in intuitionistic logic (i.e. the set of formulas $\forall \bar{x_i}.[\hat{G}_i \supset \hat{A}_i]$, for all $i \in \{1, \ldots, n\}$). Let $\mathcal{D}$ be a definition that extends $\mathcal{D}(seq) \cup \mathcal{D}(prog)$ with clauses that do not define nat, seq, element, or prog. Then the sequent $\longrightarrow L \rhd B$ is provable in $FO\lambda^{\Delta\mathbb{N}}$ with definition $\mathcal{D}$ if and only if $\hat{B}$ is an intuitionistic consequence of $\hat{L} \cup \mathcal{P}$.

A complete proof can be found in [22]. The following theorem states that we can prove in the meta-logic that the usual structural rules and the cut rule are admissible for our object-logic.

Theorem 3.2 (Object-level cut, exchange, weakening, contraction) *The following formulas are provable in $FO\lambda^{\Delta\mathbb{N}}$ with respect to the definition $\mathcal{D}(seq)$.*

$$\begin{array}{c}
\forall a, b, l.[(a :: l) \rhd b \;\supset\; l \rhd \langle a \rangle \;\supset\; l \rhd b] \\
\forall i, b, l, l'.[\mathit{nat}\ i \;\supset\; \forall a.[\mathit{element}\ a\ l \;\supset\; \mathit{element}\ a\ l'] \\
\supset\; \mathit{seq}_i\ l\ b \;\supset\; \mathit{seq}_i\ l'\ b]
\end{array}$$

Since our object-logic is restricted to second-order, it is sufficient to show that cuts on atomic formulas are eliminable, and this only requires natural number induction in the meta-logic. If we consider higher-order object-logics, we would need richer induction schemes in our meta-logic. Fortunately, second-order object-logics are appropriate for the vast majority of specifications using higher-order abstract syntax.

4. Representing static and dynamic semantics

We now develop in this object-logic the specification of judgments regarding the typing and evaluation of λ-terms. We have chosen the language of λ-terms to correspond to our example in Section 2.1 and to keep the initial presentation brief and simple. We then show how to extend this specification to the programming language PCF. The required meta-logic constants for λ-terms are $abs : (i \rightarrow i) \rightarrow i$ and $app : i \rightarrow i \rightarrow i$, and for simple types (over one primitive type) we need $gnd : i$ and $arr : i \rightarrow i \rightarrow i$. Our object-logic predicates representing typability, natural semantics, and transition semantics are denoted by the meta-logic constants $typeof$, $\Downarrow$, $\leadsto$, and $\leadsto^*$, all of type $i \rightarrow i \rightarrow atm$. The object-logic specifications for these are the usual ones, written in the L_λ subset of higher-order logic [26] and are those common to specifications written in, say, λProlog [15] and Elf [32]. This object-level specification is represented at the meta-level as the definition $\mathcal{D}(lambda)$ shown in Table 1. (We have dropped the $\triangleq \top$ body of these clauses.) This definition can be interpreted in a logic programming fashion to compute object-level substitutions, simple type checking, and call-by-name evaluation in both SOS and natural semantic styles. (Call-by-value is just as easily represented and used.) We now show that this same definition can be integrated into a framework in which properties about these judgments can be proved.

We list several formulas that can be proved in this meta-logic.

Theorem 4.1 *The following formulas are provable in $FO\lambda^{\Delta\mathbb{N}}$ from the definition that accumulates $\mathcal{D}(seq)$, $\mathcal{D}(lambda)$ and the clause $X \equiv X \triangleq \top$ defining the predicate $\equiv : i \rightarrow i \rightarrow o$.*

Determinacy of semantics:

$$\begin{array}{c}
\forall m, m_1, m_2.[\,\rhd\langle m \Downarrow m_1 \rangle \supset\; \rhd\langle m \Downarrow m_2 \rangle \\
\supset m_1 \equiv m_2] \\
\forall m, m_1, m_2.[\,\rhd\langle m \leadsto m_1 \rangle \supset\; \rhd\langle m \leadsto m_2 \rangle \\
\supset m_1 \equiv m_2] \\
\forall m, r_1, r_2.[\,\rhd\langle m \leadsto^* (\mathit{abs}\ r_1) \rangle \supset\; \rhd\langle m \leadsto^* (\mathit{abs}\ r_2) \rangle \\
\supset (\mathit{abs}\ r_1) \equiv (\mathit{abs}\ r_2)]
\end{array}$$

Equivalence of semantics:

$$\begin{array}{c}
\forall m, r.[\,\rhd\langle m \Downarrow (\mathit{abs}\ r) \rangle \supset\; \rhd\langle m \leadsto^* (\mathit{abs}\ r) \rangle] \\
\forall m, r.[\,\rhd\langle m \leadsto^* (\mathit{abs}\ r) \rangle \supset\; \rhd\langle m \Downarrow (\mathit{abs}\ r) \rangle]
\end{array}$$

Subject reduction:

$$\begin{array}{l}
\forall m, n.[\,\rhd\langle m \Downarrow n \rangle \\
\qquad \supset \forall t(\;\rhd\langle \mathit{typeof}\ m\ t \rangle \supset\; \rhd\langle \mathit{typeof}\ n\ t \rangle)] \\
\forall m, n.[\,\rhd\langle m \leadsto n \rangle \\
\qquad \supset \forall t(\;\rhd\langle \mathit{typeof}\ m\ t \rangle \supset\; \rhd\langle \mathit{typeof}\ n\ t \rangle)] \\
\forall m, n.[\,\rhd\langle m \leadsto^* n \rangle \\
\qquad \supset \forall t(\;\rhd\langle \mathit{typeof}\ m\ t \rangle \supset\; \rhd\langle \mathit{typeof}\ n\ t \rangle)]
\end{array}$$

Although the meta-level proofs are not difficult and generally follow closely an informal proof, we do not include them here since they take at least a couple pages to present in detail. The first subject reduction theorem is a formalization of Theorem 2.1; its $FO\lambda^{\Delta\mathbb{N}}$ proof is given in Appendix A. All these theorems and the corresponding ones for PCF mentioned below have been constructed formally using the Pi proof editor of Eriksson [10].

We now extend this encoding of the static and dynamic semantics for untyped λ-terms to PCF. The necessary meta-logic constants for PCF types are

$$num : \underline{i} \qquad bool : \underline{i} \qquad arr : \underline{i} \rightarrow \underline{i} \rightarrow \underline{i}$$

Those for PCF terms are

$zero$	:	i	is_zero	:	$i \rightarrow i$
$true$	:	i	if	:	$i \rightarrow i \rightarrow i \rightarrow i$
$false$	:	i	abs	:	$\underline{i} \rightarrow (i \rightarrow i) \rightarrow i$
$succ$	:	$i \rightarrow i$	app	:	$i \rightarrow i \rightarrow i$
$pred$	:	$i \rightarrow i$	rec	:	$\underline{i} \rightarrow (i \rightarrow i) \rightarrow i$

prog	$(\mathit{typeof}\ (\mathit{abs}\ R)\ (\mathit{arr}\ T\ U))$	$\bigwedge n.[(\mathit{typeof}\ n\ T) \Rightarrow \langle \mathit{typeof}\ (R\,n)\ U\rangle]$
prog	$(\mathit{typeof}\ (\mathit{app}\ M\ N)\ T)$	$\bigvee u.[\langle \mathit{typeof}\ M\ (\mathit{arr}\ u\ T)\rangle\ \&\ \langle \mathit{typeof}\ N\ u\rangle]$
prog	$((\mathit{abs}\ R) \Downarrow (\mathit{abs}\ R))$	1
prog	$((\mathit{app}\ M\ N) \Downarrow V)$	$\bigvee r.[\langle M \Downarrow (\mathit{abs}\ r)\rangle\ \&\ \langle (r\ N) \Downarrow V\rangle]$
prog	$((\mathit{app}\ (\mathit{abs}\ R)\ M) \leadsto (R\,M))$	1
prog	$((\mathit{app}\ M\ N) \leadsto (\mathit{app}\ M'\ N))$	$\langle M \leadsto M'\rangle$
prog	$(M \leadsto^* M)$	1
prog	$(M \leadsto^* N)$	$\bigvee m'.[\langle M \leadsto m'\rangle\ \&\ \langle m' \leadsto^* N\rangle]$

Table 1. $\mathcal{D}(lambda)$**: Object-logic encoding of typing and evaluation of** λ**-terms.**

prog	$(\mathit{typeof}\ \mathit{zero}\ \mathit{num})$	1
prog	$(\mathit{typeof}\ \mathit{true}\ \mathit{bool})$	1
prog	$(\mathit{typeof}\ \mathit{false}\ \mathit{bool})$	1
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	$(\mathit{typeof}\ (\mathit{app}\ M\ N)\ T)$	$\bigvee u.[\langle \mathit{typeof}\ M\ (\mathit{arr}\ u\ T)\rangle\ \&\ \langle \mathit{typeof}\ N\ u\rangle]$
prog	$(\mathit{typeof}\ (\mathit{rec}\ T\ R)\ T)$	$\bigwedge n.[(\mathit{typeof}\ n\ T) \Rightarrow \langle \mathit{typeof}\ (R\,n)\ T\rangle]$
prog	$(\mathit{zero} \Downarrow \mathit{zero})$	1
prog	$(\mathit{true} \Downarrow \mathit{true})$	1
prog	$(\mathit{false} \Downarrow \mathit{false})$	1
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	$((\mathit{is\_zero}\ M) \Downarrow \mathit{false})$	$\bigvee v.[\langle M \Downarrow (\mathit{succ}\ v)\rangle]$
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	$((\mathit{abs}\ T\ R) \Downarrow (\mathit{abs}\ T\ R))$	1
prog	$((\mathit{app}\ M\ N) \Downarrow V)$	$\bigvee r.\bigvee t.[\langle M \Downarrow (\mathit{abs}\ t\ r)\rangle\ \&\ \langle (r\ N) \Downarrow V\rangle]$
prog	$((\mathit{rec}\ T\ R) \Downarrow V)$	$\langle (R\,(\mathit{rec}\ T\ R)) \Downarrow V\rangle$

Table 2. $\mathcal{D}(pcf)$**: Object-logic encoding of typing and evaluation for PCF.**

Since both types and terms of PCF are represented by the object logic type i, we have underlined the occurrences of i that correspond to PCF types 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 natural semantics are denoted by the same meta-logic constants as above, $typeof : i \rightarrow \underline{i} \rightarrow atm$ and $\Downarrow : i \rightarrow i \rightarrow atm$, plus the additional constant $value : i \rightarrow atm$. The object-level specification is represented at the meta-level as the definition $\mathcal{D}(pcf)$ shown in Table 2. The transition semantics for PCF can be represented by a similar extension of the corresponding specification for λ-terms given in Table 1. The type tags in PCF terms allow the unicity of typing

$$\forall m, t_1, t_2. [\, \rhd \langle typeof\ m\ t_1 \rangle \supset\ \rhd \langle typeof\ m\ t_2 \rangle \\ \supset t_1 \equiv t_2]$$

to hold in addition to formulas corresponding to those of Theorem 4.1.

The use of object-level sequents may seem at first a rather drastic step to take to embed the kind of hypothetical judgments 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 [30], uses typing judgments of the form $\Gamma\ \rhd M : \sigma$ and performs induction over their (sequent-style) derivation.

5. Related work

There are several other approaches to dealing with higher-order abstract syntax directly in a formalized meta-language. Despeyroux, Felty, and Hirschowitz [7, 6] 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 constructor for λ-abstraction would have type $(var \rightarrow tm) \rightarrow tm$. 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 *var*, they lose several key benefits of higher-order abstract syntax: meta-level β-reduction cannot be used for object-level substitution and the power of meta-level cut-elimination is reduced significantly (both of these features were key aspects of the proof of Theorem 2.1). In addition, the Coq type $(var \rightarrow tm)$ includes functions besides those expressible as λ-terms, so the type *tm* includes expressions that do not encode terms of the object-language. They avoid these *exotic* terms through the definition and use of a validation predicate.

Despeyroux, Pfenning, and Schürmann [8] address the problem of exotic terms by using a modal operator to distinguish the types of parametric functions (expressible as λ-terms) from the types of arbitrary functions. As a result, their calculus allows primitive recursive functionals while preserving the adequacy of higher-order abstract syntax encodings. This represents a start toward a logical framework supporting meta-theoretic reasoning, higher-order abstract syntax, and the judgments-as-types principle. In such a framework a derivation would be represented as a function whose type is the derived property. Thus the $\rightarrow$ type constructor must be rich enough to include the mappings from derivations to derivations such as the realizations of case analysis and induction. Their work is orthogonal to our work presented in this paper. We are not attempting to support the judgments-as-types principle, so the types of our meta-logic are only used to encode syntactic structure. Thus we can restrict these types to include only λ-terms, ensuring the adequacy of encodings in higher-order abstract syntax. They, on the other hand, do not address the issue of induction principles for higher-order abstract syntax.

Schürmann [37] offers another framework supporting higher-order abstract syntax and meta-theoretic analysis. He constructs a meta-logic MLF to reason about deductive systems represented in the Horn fragment of LF. This meta-logic includes a recursion rule that is used for induction and case analysis. This approach is similar in spirit to ours in that there are three levels: the deductive system(s) under consideration, the logic in which the deductive systems are encoded, and the logic in which meta-theoretic analysis takes place. His meta-logic MLF, however, is designed for a specific, fixed intermediate logic, the Horn fragment of LF. In our case, the meta-logic is a general framework capable of representing and reasoning about a variety of logics. In addition, the validity of Schürmann's work depends on cut-elimination for MLF, which is still an open question.

Still another strategy for meta-theoretic reasoning about higher-order abstract syntax encodings is to perform each case of a proof in the meta-logic, but verify the completeness of the proof outside the logical framework. Rohwedder and Pfenning [34, 35] investigate the design and implementation of such external validity conditions.

Matthews seeks to reconcile the advantages of LF-style encodings with the facilities for meta-theoretic analysis found in theories of inductive definitions [20]. 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.

6. Conclusion

In this paper we have presented a single and simply motivated meta-logic $FO\lambda^{\Delta\mathbb{N}}$. Within this logic we have shown how to encode a simple second-order intuitionistic logic and in that logic we have encoded and reasoned with typing and evaluation judgments for a simple functional programming language. The main contribution of this research is that the encodings at both levels can be done using higher-order abstract syntax, and we are able to reason naturally in our framework about these encodings.

The meta-logic $FO\lambda^{\Delta\mathbb{N}}$ has also been used to reason about simulation and bisimulation in abstract transition systems and CCS [25]. 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 [1]. It would also be interesting to use Howe's technique [17] to prove the congruence of bisimulation in our framework.

In $FO\lambda^{\Delta\mathbb{N}}$ we can easily represent object-logics other than the intuitionistic one used here. Encoding fragments of second-order linear logic, along the lines of Lolli [16] and Forum [27], can be done simply by changing the definition of *seq* given in Section 3. These various intuitionistic and linear logics are known to be able to capture a wide range of judgments in the areas of functional, imperative, and concurrent programming languages. Our meta-logic $FO\lambda^{\Delta\mathbb{N}}$ should be able to formalize many proofs about judgments made within those logics, and we plan to demonstrate this in our future work.

Acknowledgments

We would like to thank Frank Pfenning for helpful feedback on early presentations of this work and Lars-Henrik Eriksson for making his Pi derivation editor [10] available to help check the formal proofs described here. The authors have been funded in part by the grants ONR N00014-93-1-1324, NSF CCR-92-09224, NSF CCR-94-00907, and ARO DAAH04-95-1-0092.

A. Subject reduction

We describe here a formal proof of subject reduction, i.e. a proof of the sequent

$$\longrightarrow \forall p.\forall v.[\,\rhd\langle p \Downarrow v\rangle \supset \forall t.(\,\rhd\langle \mathit{typeof}\ p\ t\rangle \supset\ \rhd\langle \mathit{typeof}\ v\ t\rangle)]$$

using the encoding of typability and natural semantics in Section 4. As we step through the proof we will correlate it with the informal proof of Theorem 2.1. Applying the $\forall\mathcal{R}$, $\supset\mathcal{R}$, $\exists\mathcal{L}$, and $\wedge\mathcal{L}$ rules to the above sequent yields

$$\mathit{nat}\ i, \mathit{seq}_i\ \mathit{nil}\ \langle p \Downarrow v\rangle, \rhd\langle \mathit{typeof}\ p\ t\rangle \longrightarrow\ \rhd\langle \mathit{typeof}\ v\ t\rangle$$

(Recall that $\rhd\langle p \Downarrow v\rangle$ is an abbreviation for $\exists i.[\mathit{nat}\ i \wedge \mathit{seq}_i\ \mathit{nil}\ \langle p \Downarrow v\rangle]$.)

As in the informal proof, we proceed with an induction on the height of the proof of $p \Downarrow v$, which is represented here by i. We will use the derived rule for complete induction (Theorem 2.4) and our invariant will be

$$\lambda i.\forall p.\forall v.[\mathit{seq}_i\ \mathit{nil}\ \langle p \Downarrow v\rangle \supset \forall t.(\,\rhd\langle \mathit{typeof}\ p\ t\rangle \supset\ \rhd\langle \mathit{typeof}\ v\ t\rangle)]$$

which we will denote by *INV*. The proof of the conclusion from the invariant applied to i is trivial, so it only remains to prove the induction step

$$\mathit{nat}\ j, \forall k.[\mathit{nat}\ k \supset k < j \supset (\mathit{INV}\ k)] \longrightarrow (\mathit{INV}\ j)$$

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

$$\mathit{nat}\ j, \forall k \ldots, \mathit{seq}_j\ \mathit{nil}\ \langle p \Downarrow v\rangle,\ \rhd\langle \mathit{typeof}\ p\ t\rangle \longrightarrow\ \rhd\langle \mathit{typeof}\ v\ t\rangle$$

In the informal proof we use the fact that the proof of the atomic formula $p \Downarrow v$ must end with a backchain. We deduce this here by applying the $\mathit{def}\mathcal{L}$ rule to $\mathit{seq}_j\ \mathit{nil}\ \langle p \Downarrow v\rangle$, which yields

$$\mathit{nat}\ (s\ j_0), \forall k \ldots, \exists d.[\mathit{prog}\ (p \Downarrow v)\ d \wedge \mathit{seq}_{j_0}\ \mathit{nil}\ d], \rhd\langle \mathit{typeof}\ p\ t\rangle \longrightarrow\ \rhd\langle \mathit{typeof}\ v\ t\rangle$$

We then apply the $\exists\mathcal{L}$ and $\wedge\mathcal{L}$ rules, and then the $\mathit{def}\mathcal{L}$ rule to $\mathit{prog}\ (p \Downarrow v)\ d$ which yields the two sequents

$$\mathit{nat}\ (s\ j_0), \forall k \ldots, \mathit{seq}_{j_0}\ \mathit{nil}\ 1, \rhd\langle \mathit{typeof}\ (\mathit{abs}\ r)\ t\rangle \longrightarrow\ \rhd\langle \mathit{typeof}\ (\mathit{abs}\ r)\ t\rangle$$

$$\mathit{nat}\ (s\ j_0), \forall k \ldots, \mathit{seq}_{j_0}\ \mathit{nil}\ \bigvee r.[\langle m \Downarrow (\mathit{abs}\ r)\rangle \& \langle (r\ n) \Downarrow v\rangle], \rhd\langle \mathit{typeof}\ (\mathit{app}\ m\ n)\ t\rangle \longrightarrow\ \rhd\langle \mathit{typeof}\ v\ t\rangle$$

This use of the $\mathit{def}\mathcal{L}$ rule corresponds to the case analysis of the formula used to prove $p \Downarrow v$. As in the informal case, the *abs* case (represented here by the first sequent) is immediate. The proof of the second sequent, representing the *app* case, begins with the use of the $\mathit{def}\mathcal{L}$, $\exists\mathcal{L}$, and $\wedge\mathcal{L}$, bringing us to the sequent

$$\mathit{nat}\ (s^3\ j_1), \forall k \ldots, \mathit{seq}_{j_1}\ \mathit{nil}\ \langle m \Downarrow (\mathit{abs}\ r)\rangle, \mathit{seq}_{j_1}\ \mathit{nil}\ \langle (r\ n) \Downarrow v\rangle, \rhd\langle \mathit{typeof}\ (\mathit{app}\ m\ n)\ t\rangle \longrightarrow\ \rhd\langle \mathit{typeof}\ v\ t\rangle$$

(We use the term $s^3\, j_1$ as an abbreviation for $s\ (s\ (s\ j_1))$.)

The informal proof continues with an analysis of the proof 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 proof must end with a backchain, and the second to determine the applicable backchain formulas. In this case there is only one applicable formula, so we are left to prove the sequent

$$\begin{array}{l} \qquad\qquad \ldots, nat\ (s\ j_0'), \\ seq_{j_0'}\ nil\ \bigvee u.[\langle typeof\ m\ (arr\ u\ t)\rangle \\ \qquad\qquad \&\langle typeof\ n\ u\rangle] \qquad \longrightarrow\ \rhd\langle typeof\ v\ t\rangle \end{array}$$

Additional uses of the $def\mathcal{L}$, $\exists\mathcal{L}$ and $\wedge\mathcal{L}$ rules brings us to the sequent

$$\begin{array}{l} \qquad\qquad \ldots, nat\ (s^3\, j_1'), \\ seq_{j_1'}\ nil\ \langle typeof\ m\ (arr\ u\ t)\rangle, \\ \qquad seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle \longrightarrow\ \rhd\langle typeof\ v\ t\rangle \end{array}$$

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 induction hypothesis $\forall k \ldots$. This requires the proof of the three sequents

$$nat\ (s^3\, j_1) \longrightarrow nat\ j_1 \qquad nat\ (s^3\, j_1) \longrightarrow j_1 < (s^3\, j_1)$$

$$\begin{array}{l} \qquad\qquad nat\ (s^3\, j_1), \forall k \ldots, \\ \qquad\quad seq_{j_1}\ nil\ \langle (r\ n) \Downarrow v\rangle, \\ \quad \rhd\langle typeof\ (abs\ r)\ (arr\ u\ t)\rangle, \\ nat\ (s^3\, j_1'), seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle \longrightarrow\ \rhd\langle typeof\ v\ t\rangle \end{array}$$

The first two of these represent the fact that the measure of the evaluation proof for m is a natural number that is smaller than the measure of the original evaluation proof for p. These can be proved by simple inductions.

The proof of the third sequent proceeds with 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

$$\begin{array}{l} \qquad\qquad \ldots, nat\ (s\ j_0''), \\ seq_{j_0''}\ nil\ \bigwedge x.[(typeof\ x\ u) \qquad , \\ \qquad\qquad \Rightarrow \langle typeof\ (r\ x)\ t\rangle] \\ \qquad\qquad\qquad\qquad \ldots \longrightarrow\ \rhd\langle typeof\ v\ t\rangle \end{array}$$

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

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

The informal proof proceeds with a use of the cut rule, and here we use the derived object-level cut rule (Theorem 3.2) with the elided assumption $seq_{j_1'}\ nil\ \langle typeof\ n\ u\rangle$ to obtain

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

The informal proof concludes by applying the induction hypothesis to the evaluation and typing judgments for m'. Again we accomplish this by applying the appropriate left rules to the induction hypothesis $\forall k \ldots$, which requires the proof of the three sequents

$$nat\ (s^3\, j_1) \longrightarrow nat\ j_1 \qquad nat\ (s^3\, j_1) \longrightarrow j_1 < (s^3\, j_1)$$

$$\rhd\langle typeof\ v\ t\rangle \longrightarrow\ \rhd\langle typeof\ v\ t\rangle$$

The first two sequents can be proved by simple inductions, and the third sequent is immediate.

References

[1] Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.

[2] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

[3] David A. Basin and Robert L. Constable. Metalogical frameworks. In G. Huet and G. D. Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.

[4] Jawahar Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, February 1995.

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

[6] Joelle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, April 1995.

[7] Joelle Despeyroux and Andre Hirschowitz. Higher-order abstract syntax with induction in Coq. In *Fifth International Conference on Logic Programming and Automated Reasoning*, pages 159–173, June 1994.

[8] Joelle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications*, April 1997.

[9] Lars-Henrik Eriksson. A finitary version of the calculus of partial inductive definitions. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the January 1991 Workshop on Extensions to Logic Programming*, volume 596 of *LNAI*, pages 89–134. Springer-Verlag, 1992.

[10] Lars-Henrik Eriksson. Pi: an interactive derivation editor for the calculus of partial inductive definitions. In A. Bundy, editor, *Proceedings of the Twelfth International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 821–825. Springer-Verlag, June 1994.

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

[12] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In E. Lusk and R. Overbeck, editors, *Ninth International Conference on Automated Deduction*, pages 61–80. Springer-Verlag, May 1988.

[13] Jean-Yves Girard. A fixpoint theorem in linear logic. A message posted on the linear@cs.stanford.edu mailing list, http://www.csl.sri.com/linear/mailing-list-traffic/www/07/mail_3.html, February 1992.

[14] Lars Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87:115–142, 1991.

[15] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[16] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[17] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[18] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[19] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, number 806 in LNCS, pages 213–237. Springer-Verlag, 1994.

[20] Seán Matthews. A practical implementation of simple consequence relations using inductive definitions. In *Proceedings of the 14th Conference on Automated Deduction*. Springer-Verlag, July 1997.

[21] Sean Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In G. Huet and G. D. Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993.

[22] Raymond McDowell. Proving meta-theorems in a logical framework. Dissertation proposal, University of Pennsylvania, November 1996.

[23] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. Draft manuscript submitted to the proceedings of the TYPES '96 workshop, March 1997.

[24] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus: Preliminary report. In *Proceedings of the 1996 Workshop on Linear Logic*, volume 3 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[25] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. Draft manuscript submitted to *Theoretical Computer Science*. Preliminary version appeared as [24], April 1997.

[26] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[27] Dale Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165:201–232, 1996.

[28] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In S. Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, September 1987.

[29] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[30] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1995.

[31] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[32] Frank Pfenning. Elf: a language for logic definition and verified meta-programming. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 313–321, June 1989.

[33] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, June 1988.

[34] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction*, volume 607 of *LNAI*, pages 537–551. Springer-Verlag, June 1992.

[35] Ekkehard Rohwedder and Frank Pfenning. Mode and termination analysis for higher-order logic programs. In *Proceedings of the European Symposium on Programming*, pages 296–310, April 1996.

[36] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, June 1993.

[37] Carsten Schürmann. A computational meta logic for the Horn fragment of LF. Master's thesis, Carnegie Mellon University, December 1995.

[38] Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, May 1996.

Papers by McDowell are available via anonymous ftp from ftp.cis.upenn.edu in pub/papers/mcdowell or via the WWW at http://www.cis.upenn.edu/˜mcdowell. Papers by Miller are available via anonymous ftp from ftp.cis.upenn.edu in pub/papers/miller or via the WWW at http://www.cis.upenn.edu/˜dale.