Two-Level Meta-Reasoning in Coq *

Amy P. Felty

School of Information Technology and Engineering
University of Ottawa, Ottawa, Ontario K1N 6N5, Canada
afelty@site.uottawa.ca

Abstract. The use of *higher-order abstract syntax* is central to the direct, concise, and modular specification of languages and deductive systems in a logical framework. Developing a framework in which it is also possible to reason about such deductive systems is particularly challenging. One difficulty is that the use of higher-order abstract syntax complicates reasoning by induction because it leads to definitions for which there are no monotone inductive operators. In this paper, we present a methodology which allows Coq to be used as a framework for such meta-reasoning. This methodology is directly inspired by the two-level approach to reasoning used in the $FO\lambda^{\Delta\mathbb{N}}$ (pronounced *fold-n*) logic. In our setting, the Calculus of Inductive Constructions (CIC) implemented by Coq represents the highest level, or *meta-logic*, and a separate *specification logic* is encoded as an inductive definition in Coq. Then, in our method as in $FO\lambda^{\Delta\mathbb{N}}$, the deductive systems that we want to reason about are the *object logics* which are encoded in the specification logic. We first give an approach to reasoning in Coq which very closely mimics reasoning in $FO\lambda^{\Delta\mathbb{N}}$ illustrating a close correspondence between the two frameworks. We then generalize the approach to take advantage of other constructs in Coq such as the use of direct structural induction provided by inductive types.

1 Introduction

Higher-order abstract syntax encodings of object logics are usually given using a typed meta-language. The terms of the untyped λ-calculus can be encoded using higher-order syntax, for instance, by introducing a type tm and two constructors: abs of type $(\boldsymbol{tm} \to tm) \to tm$ and app of type $tm \to tm \to tm$. As this example shows, it is often useful to use negative occurrences of the type introduced for representing the terms of the object logic. (Here the single negative occurrence is in boldface.) Predicates of the meta-logic are used to express judgments in the object logic such as "term M has type t". Embedded implication is often used to represent *hypothetical judgments*, which can result in negative occurrences of such predicates. For example the following rule which defines typing for λ-

* In *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics, August 2002,* ©*Springer-Verlag.*

abstraction in the object logic

$$\frac{\begin{array}{c}(x : \tau_1)\\ M : \tau_2\end{array}}{\lambda x.M : \tau_1 \to \tau_2}$$

can be expressed using the *typeof* predicate in the following formula.

$$\begin{array}{l}\forall M : tm \to tm. \forall \tau_1, \tau_2 : tm.\\ \quad (\forall x : tm.(typeof\ x\ \tau_1) \supset (typeof\ (M\ x)\ \tau_2))\\ \quad\quad \supset (typeof\ (abs\ M)\ (\tau_1 \to \tau_2))\end{array}$$

The Coq system [21] implements the Calculus of Inductive Constructions (CIC) and is one of many systems in which such negative occurrences cause difficulty. In particular, the inductive types of the language cannot be used directly for this kind of encoding of syntax or inference rules.

$FO\lambda^{\Delta\mathbb{N}}$ is a logical framework capable of specifying a wide variety of deductive systems [13]. It is one of the first to overcome various challenges and allow both specification of deductive systems and reasoning about them within a single framework. It is a higher-order intuitionistic logic with support for natural number induction and definitions. A rule of definitional reflection is included and is central to reasoning in the logic [8]. This rule in particular represents a significant departure from the kinds of primitive inference rules found in Coq and a variety of other systems that implement similar logics. Our methodology illustrates that, for a large class of theorems, reasoning via this rule can be replaced by reasoning with inductive types together with a small number of assumptions about the constants that are introduced to encode a particular deductive system.

We define both the specification logic and the object logic as inductive definitions in Coq. Although there are no inductive definitions in $FO\lambda^{\Delta\mathbb{N}}$, our Coq definitions of specification and object logics closely resemble the corresponding $FO\lambda^{\Delta\mathbb{N}}$ definitions of the same logics. The use of a two-level logic in both $FO\lambda^{\Delta\mathbb{N}}$ and Coq solves the problem of inductive reasoning in the presence of negative occurrences in hypothetical judgments. Hypothetical judgments are expressed at the level of the object logic, while inductive reasoning about these object logics takes place at the level of the specification logic and meta-logic. More specifically, in $FO\lambda^{\Delta\mathbb{N}}$, a combination of natural number induction and definitional reflection provides induction on the height of proofs in the specification logic. For the class of theorems we consider, we can mimic the natural number induction of $FO\lambda^{\Delta\mathbb{N}}$ fairly directly in Coq. In addition, the Coq environment provides the extra flexibility of allowing reasoning via direct induction using the theorems generated by the inductive definitions. For example, we can use direct structural induction on proof trees at both the specification level and the object-level.

One of our main goals in this work is to provide a system that allows programming and reasoning about programs and programming languages within a single framework. The Centaur System [3] is an early example of such a system. We are interested in a proof and program development environment that supports higher-order syntax. In particular, we are interested in the application of

such a system to building proof-carrying code (PCC) systems. PCC [17] is an approach to software safety where a producer of code delivers both a program and a formal proof that verifies that the code meets desired safety policies. We have built prototype PCC systems [1, 2] in both λProlog [16] and Twelf [19] and have found higher-order syntax to be useful in both programming and expressing safety properties. Definitional reflection as in $FO\lambda^{\Delta\mathbb{N}}$ is difficult to program directly in λProlog and Twelf. On the other hand, support for inductive types similar to that of Coq is straightforward to implement. We hope to carry over the methodology we describe here to provide more flexibility in constructing proofs in the PCC setting.

In this paper, after presenting the Calculus of Inductive Constructions in Sect. 2, we begin with the example proof of subject reduction for the untyped λ-calculus from McDowell and Miller [13] (also used in Despeyroux et al. [5]). For this example, we use a sequent calculus for a second-order minimal logic as our specification logic. We present a version of the proof that uses natural number induction in Sect. 3. By using natural number induction, we are able to mimic the corresponding $FO\lambda^{\Delta\mathbb{N}}$ proof, and in Sect. 4 we discuss how the $FO\lambda^{\Delta\mathbb{N}}$ proof illustrates the correspondence in reasoning in the two systems. In Sect. 5, we present an alternate proof which illustrates reasoning by direct structural induction in Coq. In Sect. 6, we conclude as well as discuss related and future work.

2 The Calculus of Inductive Constructions

We assume some familiarity with the Calculus of Inductive Constructions. We note here the notation used in this paper, much of which is taken from the Coq system. Let x represent variables and M, N represent terms of CIC. The syntax of terms is as follows.

$$\begin{gathered} Prop \mid Set \mid Type \mid x \mid MN \mid \lambda x : M.N \mid \\ \forall x : M.N \mid M \to N \mid M \wedge N \mid M \vee N \mid \\ \exists x : M.N \mid \neg M \mid M = N \mid True \mid Ind\ x : M\ \{N_1 | \cdots | N_n\} \mid \\ Rec\ M\ N \mid Case\ x : M\ of\ M_1 \Rightarrow N_1, \ldots, M_n \Rightarrow N_n \end{gathered}$$

Here $\forall$ is the dependent type constructor and the arrow ($\to$) is the usual abbreviation when the bound variable does not occur in the body. Of the remaining constants, $Prop$, Set, $Type$, λ, Ind, Rec, and $Case$ are primitive, while the others are defined. $Prop$ is the type of logical propositions, whereas Set is the type of data types. $Type$ is the type of both $Prop$ and Set. Ind is used to build inductive definitions where M is the type of the class of terms being defined and $N_1, \ldots, N_n$ where $n \geq 0$ are the types of the constructors. Rec and $Case$ are the operators for defining recursive and inductive functions, respectively, over inductive types. Equality on Set ($=$) is Leibnitz equality.

A constant is introduced using the Parameter keyword and ordinary definitions which introduce a new constant and the term it represents are defined using the Definition keyword. Inductive definitions are introduced with an Inductive

declaration where each constructor is given with its type separated by vertical bars. When an inductive definition is made, Coq automatically generates operators for reasoning by structural induction and for defining recursive functions on objects of the new type. We use the section mechanism of the system which provides support for developing theories modularly. The Variable keyword provides a way to introduce constants that will be discharged at the end of a section. Axiom is used to introduce formulas that are assumed to hold and Theorem introduces formulas which are immediately followed by a proof or a series of commands (*tactics*) that indicate how to construct the proof.

3 An Example: Subject Reduction for the Untyped λ-Calculus

A variety of specification logics can be defined. In this paper, we use a simple minimal logic taken from McDowell and Miller [13]. In Coq, we introduce the type *prp* to encode formulas of the specification logic, and the type *atm* (left as a parameter at this stage) to encode the atomic formulas of the object logic.

Variable $atm : Set.$
Variable $tau : Set.$
Inductive $prp : Set :=$
$\langle\rangle : atm \to prp \mid tt : prp \mid \& : prp \to prp \to prp \mid$
$\Rightarrow : atm \to prp \to prp \mid \bigwedge : (tau \to prp) \to prp \mid \bigvee : (tau \to prp) \to prp.$

The operator $\langle\rangle$ is used to coerce objects of type *atm* to *prp*. The other constructors of the inductive definition of *prp* define the logical connectives of the specification logic. We use a higher-order syntax encoding of the quantifiers $\bigwedge$ (forall) and $\bigvee$ (exists), i.e., each quantifier takes one argument which is a λ-term so that binding of quantifiers in the specification logic is encoded as λ-binding at the meta-level. Note that we parameterize the quantification type; this version of the specification logic limits quantification to a single type *tau*. This is not a serious restriction here since we encode all syntactic objects in our examples using the single type *tm*; also, it can be extended to include other types if necessary. Here, we freely use infix and prefix/postfix operators, without discussing the details of using them in Coq.

For illustration purposes, we show the induction principle generated by Coq resulting from the above definition of *prp*.

$$\begin{aligned}
&\forall P : prp \to Prop. \\
&\quad [(\forall A : atm. P\langle A\rangle) \to \\
&\quad\ P(tt) \to \\
&\quad\ (\forall B : prp. PB \to \forall C : prp. PC \to P(B \& C)) \to \\
&\quad\ (\forall A : atm. \forall B : prp. PB \to P(A \Rightarrow B)) \to \\
&\quad\ (\forall B : tau \to prp. (\forall x : tau. P(Bx)) \to P(\textstyle\bigwedge B)) \to \\
&\quad\ (\forall B : tau \to prp. (\forall x : tau. P(Bx)) \to P(\textstyle\bigvee B))] \to \forall B : prp. PB
\end{aligned}$$

After closing the section containing the above definitions, *prp* will have type $Set \to Set \to Set$ because *atm* and *tau* are discharged.

The Coq inductive definition in Fig. 1 is a direct encoding of the specification logic. The predicate *prog* is used to declare the object-level deductive system. It

```
Variable prog : atm → prp → Prop.
Inductive seq : nat → list atm → prp → Prop :=
    sbc : ∀i : nat.∀A : atm.∀L : list atm.∀b : prp.
          (prog A b) → (seq i L b) → (seq (S i) L ⟨A⟩)
  | sinit : ∀i : nat.∀A, A' : atm.∀L : list atm.
          (element A (A' :: L)) → (seq i (A' :: L) ⟨A⟩)
  | strue : ∀i : nat.∀L : list atm.(seq i L tt)
  | sand : ∀i : nat.∀B, C : prp.∀L : list atm.
          (seq i L B) → (seq i L C) → (seq (S i) L (B&C))
  | simp : ∀i : nat.∀A : atm.∀B : prp.∀L : list atm.
          (seq i (A :: L) B) → (seq (S i) L (A ⇒ B))
  | sall : ∀i : nat.∀B : tau →  prp.∀L : list atm.
          (∀x : tau.(seq i L (B x))) → (seq (S i) L (⋀ B))
  | ssome : ∀i : nat.∀B : tau → prp.∀L : list atm.
          ∀x : tau.(seq i L (B x)) → (seq (S i) L (⋁ B)).
Definition ▷ : list atm → prp → Prop := λl : list atm.λB : prp.∃i : nat.(seq i l B).
Definition ▷0 : prp → Prop := λB : prp.∃i : nat.(seq i nil B).
```

Fig. 1. Definition of the Specification Logic in Coq

is a parameter at this stage. A formula of the form (*prog A b*) as part of the object logic means roughly that b implies A where A is an atom. We will see shortly how *prog* is used to define an object logic. Most of the clauses of this definition encode rules of a sequent calculus which introduce connectives on the right of a sequent. For example, the *sand* clause specifies the following $\wedge$-R rule.

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

In the Coq definition, the natural number i indicates that the proofs of the premises have height at most i and the proof of the conclusion has height at most $i+1$. (S is the successor function from the Coq libraries.) The *sinit* clause specifies when a sequent is initial (i.e., the formula on the right appears in the list of hypotheses on the left). We omit the definition of *element*, which is straightforward. The *sbc* clause represents backchaining. A backward reading of this rule states that A is provable from hypotheses L in at most $i+1$ steps if b is provable from hypotheses L in at most i steps, where "A implies B" is a statement in the object logic. The definitions of $\triangleright$ and $\triangleright_0$ at the end of the figure are made for convenience in expressing properties later. The former is written using infix notation.

Theorems which *invert* this definition can be directly proved using the induction and recursion operators for the type *seq*. For example, it is clear that if a proof ends in a sequent with an atomic formula on the right, then the sequent was either derived using the rule for *prog* (*sbc*) or the atom is an element of the

list of formulas on the left ($sinit$). This theorem is expressed as follows.

$$\begin{array}{l}\textsf{Theorem } seq\_atom\_inv : \forall i : nat.\forall A : atm.\forall l : list\ atm.(seq\ i\ l\ \langle A\rangle) \rightarrow \\ \quad [\exists j : nat.\exists b : prp.(i = (S\ j) \wedge (prog\ A\ b) \wedge (seq\ j\ l\ b)) \vee \\ \quad \exists A' : atm.\exists l' : list\ atm.(l = (A' :: l') \wedge (element\ A\ l))].\end{array}$$

Induction principles generated by Coq are also useful for reasoning by case analysis. For example, case analysis on seq can be used to prove the seq_cut property below, which is an essential part of our proof development.

$$\textsf{Theorem } seq\_cut : \forall a : atm.\forall b : prp.\forall l : list\ atm.(a :: l) \rhd b \rightarrow l \rhd \langle a\rangle \rightarrow l \rhd b.$$

This theorem can also be proven by case analysis on prp using the induction principle shown earlier. In fact, for this particular theorem, case analysis on prp leads to a somewhat simpler proof than case analysis on seq.

Our object logic consists of untyped λ-terms, types, and rules for assigning types to terms. Terms and types are encoded using the parameter declarations below.

$$\begin{array}{ll}\textsf{Parameter } tm : Set. & \\ \textsf{Parameter } gnd : tm. & \textsf{Parameter } abs : (tm \rightarrow tm) \rightarrow tm. \\ \textsf{Parameter } arr : tm \rightarrow tm \rightarrow tm. & \textsf{Parameter } app : tm \rightarrow tm \rightarrow tm. \\ \textsf{Axiom } gnd\_arr : \forall t, u : tm.\neg(gnd = (arr\ t\ u)). & \\ \textsf{Axiom } abs\_app : \forall R : tm \rightarrow tm.\forall M, N : tm.\neg((abs\ R) = (app\ M\ N)). & \\ \textsf{Axiom } arr\_inj : \forall t, t', u, u' : tm.(arr\ t\ u) = (arr\ t'\ u') \rightarrow t = t' \wedge u = u'. & \\ \textsf{Axiom } abs\_inj : \forall R, R' : tm \rightarrow tm.(abs\ R) = (abs\ R') \rightarrow R = R'. & \\ \textsf{Axiom } app\_inj : \forall M, M', N, N' : tm. & \\ \qquad (app\ M\ N) = (app\ M'\ N') \rightarrow M = M' \wedge N = N'. & \end{array}$$

The five axioms following them express properties about distinctness and injectivity of constructors. For example, a term beginning with abs is always distinct from one beginning with app. Also, if two terms $(abs\ R)$ and $(abs\ R')$ are equal then so are R and R'. For objects defined inductively in Coq, such properties are derivable. Here, we cannot define tm inductively because of the negative occurrence in the type of the abs constant, so we must include them explicitly. They are the only axioms we require for proving properties about this object logic. The type tm is the type which instantiates tau in the definitions of prp and seq above.

Note that by introducing constants and axioms, we are restricting the context in which reasoning in Coq is valid and actually corresponds to reasoning about the deduction systems we encode. For example, we cannot discharge these constants and instantiate them with arbitrary objects such as inductively defined elements of Set. We do not want to be able to prove any properties about these constants other than the ones we assume and properties that follow from them.

The definitions for atomic formulas and for the $prog$ predicate, which encode typing and evaluation of our object logic are given in Fig. 2. An example of an inversion theorem that follows from this definition is the following. Its proof

$$
\begin{array}{l}
\mathsf{Inductive}\ atm : Set := typeof : tm \to tm \to atm \mid \Downarrow : tm \to tm \to atm. \\
\mathsf{Inductive}\ prog : atm \to prp \to Prop := \\
\quad tabs : \forall t, u : tm. \forall R : tm \to tm. \\
\qquad (prog\ (typeof\ (abs\ R)\ (arr\ t\ u)) \\
\qquad\quad (\bigwedge\ \lambda n : tm. ((typeof\ n\ t) \Rightarrow \langle typeof\ (R\ n)\ u\rangle))) \\
\quad \mid tapp : \forall M, N, t : tm. \\
\qquad (prog\ (typeof\ (app\ M\ N)\ t) \\
\qquad\quad (\bigvee\ \lambda u : tm. (\langle typeof\ M\ (arr\ u\ t)\rangle \& \langle typeof\ N\ u\rangle))) \\
\quad \mid eabs : \forall R : tm \to tm. (prog\ ((abs\ R) \Downarrow (abs\ R))\ tt) \\
\quad \mid eapp : \forall M, N, V : tm. \forall R : tm \to tm. \\
\qquad (prog\ ((app\ M\ N) \Downarrow V)\ (\langle M \Downarrow (abs\ R)\rangle \& \langle (R\ N) \Downarrow V\rangle))
\end{array}
$$

Fig. 2. Definition of the Object Logic in Coq

requires seq_atom_inv above.

$$
\begin{array}{l}
\mathsf{Theorem}\ eval\_nil\_inv : \forall j : nat. \forall M, V : tm. (seq\ j\ nil\ \langle M \Downarrow V\rangle) \to \\
\quad [(\exists R : tm \to tm. M = (abs\ R) \wedge V = (abs\ R)) \vee \\
\quad\ (\exists k : nat. \exists R : tm \to tm. \exists P, N : tm. j = (S\ (S\ k)) \wedge M = (app\ P\ N) \wedge \\
\qquad (seq\ k\ nil\ \langle P \Downarrow (abs\ R)\rangle) \wedge (seq\ k\ nil\ \langle (R\ N) \Downarrow V\rangle))].
\end{array}
$$

We are now ready to express and prove the subject reduction property.

$$
\mathsf{Theorem}\ sr : \forall p, v : tm.\ \triangleright_0\ \langle p \Downarrow v\rangle \to \forall t : tm.\ \triangleright_0\ \langle typeof\ p\ t\rangle \to \triangleright_0 \langle typeof\ v\ t\rangle.
$$

Our proof of this theorem corresponds directly to the one given by Miller and McDowell [13]. We show a few steps to illustrate. After one definition expansion of $\triangleright_0$, several introduction rules for universal quantification and implication, and an elimination of the existential quantifier on one of the assumptions, we obtain the sequent

$$
(seq\ i\ nil\ \langle p \Downarrow v\rangle), \triangleright_0 \langle typeof\ p\ t\rangle \longrightarrow \triangleright_0 \langle typeof\ v\ t\rangle. \tag{1}
$$

(We display meta-level sequents differently than the Coq system. We omit type declarations and names of hypotheses, and we separate the hypotheses from the conclusion with a sequent arrow.) We now apply complete induction, which comes from the basic Coq libraries and is stated:

$$
\begin{array}{l}
\mathsf{Theorem}\ lt\_wf\_ind : \forall k : nat. \forall P : nat \to Prop. \\
\quad (\forall n : nat. (\forall m : nat. m < n \to Pm) \to Pn) \to Pk.
\end{array}
$$

After solving the trivial subgoals, we are left to prove $\forall k. (k < j \supset (IP\ k)) \supset (IP\ j)$, where IP denotes the formula

$$
\lambda i : nat. \forall p, v : tm. (seq\ i\ nil\ \langle p \Downarrow v\rangle) \to \forall t : tm.\ \triangleright_0\ \langle typeof\ p\ t\rangle \to \triangleright_0 \langle typeof\ v\ t\rangle.
$$

After clearing the old assumptions and applying a few more intro/elim rules we get the following sequent.

$$
\forall k. k < j \to (IP\ k), (seq\ j\ nil\ \langle p \Downarrow v\rangle), \triangleright_0 \langle typeof\ p\ t\rangle \longrightarrow \triangleright_0 \langle typeof\ v\ t\rangle. \tag{2}
$$

Note that a proof of $(seq\ j\ nil\ \langle p \Downarrow v\rangle)$ must end with the first clause for *seq* (containing *prog*). Here, we apply the *eval_nil_inv* inversion theorem to obtain

$$\begin{array}{l}\forall k.k < j \to (IP\ k),\\ [(\exists R : tm \to tm.p = (abs\ R) \wedge v = (abs\ R)) \vee\\ \ (\exists k' : nat.\exists R : tm \to tm.\exists P, N : tm.j = (S\ (S\ k')) \wedge p = (app\ P\ N) \wedge\\ \quad (seq\ k'\ nil\ \langle P \Downarrow (abs\ R)\rangle) \wedge (seq\ k'\ nil\ \langle (R\ N) \Downarrow v\rangle))],\\ \rhd_0\langle typeof\ p\ t\rangle \longrightarrow \rhd_0\langle typeof\ v\ t\rangle.\end{array}$$

Then after eliminating the disjunction, existential quantifiers, and conjunction, as well as performing the substitutions using the equalities we obtain the following two sequents.

$$\forall k.k < j \to (IP\ k), \rhd_0\langle typeof\ (abs\ R)\ t\rangle \longrightarrow \rhd_0\langle typeof\ (abs\ R)\ t\rangle \quad (3)$$

$$\begin{array}{l}\forall k.k < (S\ (S\ k')) \to (IP\ k), (seq\ k'\ nil\ \langle P \Downarrow (abs\ R)\rangle),\\ \quad (seq\ k'\ nil\ \langle (R\ N) \Downarrow v\rangle), \rhd_0\langle typeof\ (app\ P\ N)\ t\rangle \longrightarrow \rhd_0\langle typeof\ v\ t\rangle\end{array} \quad (4)$$

Note that the first is directly provable.

We carry out one more step of the Coq proof of (4) to illustrate the use of a distinctness axiom. In particular, we show the two sequents that result from applying an inversion theorem to the formula just before the sequent arrow in (4) and then applying all possible introductions, eliminations, and substitutions. (We abbreviate $(S\ (S\ (S\ k')))$ as $(S^3\ k')$ and similarly for other such expressions.)

$$\begin{array}{l}\forall k.k < (S^5\ k') \to (IP\ k), (seq\ (S^3\ k')\ nil\ \langle P \Downarrow (abs\ R)\rangle),\\ \quad (seq\ (S^3\ k')\ nil\ \langle (R\ N) \Downarrow v\rangle), (app\ P\ N) = (abs\ R), t = (arr\ T'\ U),\\ \quad (seq\ k'\ ((typeof\ M\ T') :: nil)\ \langle (typeof\ (R\ M)\ U)\rangle)) \longrightarrow \rhd_0\langle typeof\ v\ t\rangle\\ \forall k.k < (S^5\ k') \to (IP\ k), (seq\ (S^3\ k')\ nil\ \langle P \Downarrow (abs\ R)\rangle),\\ \quad (seq\ (S^3\ k')\ nil\ \langle (R\ N) \Downarrow v\rangle), (seq\ k'\ nil\ \langle typeof\ P\ (arr\ u\ t)\rangle)\\ \quad (seq\ k'\ nil\ \langle typeof\ N\ u\rangle)) \longrightarrow \rhd_0\langle typeof\ v\ t\rangle\end{array}$$

Note the occurrence of $(app\ P\ N) = (abs\ R)$ in the first sequent. This sequent must be ruled out using the *abs_app* axiom.

The remainder of the Coq proof continues using the same operations of applying inversion theorems, and using introduction and elimination rules. It also includes applications of lemmas such as *seq_cut* mentioned above.

4 A Comparison to $FO\lambda^{\Delta^{\mathbb{N}}}$

The basic logic of $FO\lambda^{\Delta^{\mathbb{N}}}$ is an intuitionistic version of a subset of Church's Simple Theory of Types with logical connectives $\bot$, $\top$, $\wedge$, $\vee$, $\supset$, $\forall_\tau$, and $\exists_\tau$. Quantification is over any type τ not containing o, which is the type of meta-level formulas. The inference rules of the logic include the usual left and right sequent rules for the connectives and rules that support natural number induction. Note that these sequent rules are at the meta-level, and thus we have sequent calculi both at the meta-level and specification level in our example proof. $FO\lambda^{\Delta^{\mathbb{N}}}$ also

has the following rules to support definitions.

$$\frac{\Gamma \longrightarrow B\Theta}{\Gamma \longrightarrow A}\,\mathrm{def}\mathcal{R}, \qquad \text{where } A = A'\Theta \text{ for some clause } \forall\bar{x}[A' =_\Delta B]$$

$$\frac{\{B\Theta, \Gamma\Theta \longrightarrow C\Theta | \Theta \in CSU(A, A') \text{ for some clause } \forall\bar{x}[A' =_\Delta B]\}}{A, \Gamma \longrightarrow C}\,\mathrm{def}\mathcal{L}$$

A definition is denoted by $\forall\bar{x}[A' =_\Delta B]$ where the symbol $=_\Delta$ is used to separate the object being defined from the body of the definition. Here, A' has the form $(p\ \bar{t})$ where p is a predicate constant, every free variable in B is also free in $(p\ \bar{t})$, and all variables free in $\bar{t}$ are contained in the list $\bar{x}$. The first rule provides "backchaining" on a clause of a definition. The second rule is the rule of *definitional reflection* and uses complete sets of unifiers (*CSU*). When this set is infinite, there will be an infinite number of premises. In practice, such as in the proofs in McDowell and Miller's work [12, 13], this rule is used only in finite cases.

Fig. 3 illustrates how *seq* and *prog* are specified as $FO\lambda^{\Delta\mathbb{N}}$ definitions. Each

$$\begin{array}{rl}
seq\ (SI)\ L\ \langle A\rangle & =_\Delta \exists b.[prog\ A\ b \wedge seq\ I\ L\ b] \\
seq\ I\ (A' :: L)\ \langle A\rangle & =_\Delta element\ A\ (A' :: L) \\
seq\ I\ L\ tt & =_\Delta \top \\
seq\ (SI)\ L\ (B\&C) & =_\Delta seq\ I\ L\ B \wedge seq\ I\ L\ C \\
seq\ (SI)\ L\ (A \Rightarrow B) & =_\Delta seq\ I\ (A :: L)\ B \\
seq\ (SI)\ L\ (\bigwedge_\tau\ B) & =_\Delta \forall_\tau x.[seq\ I\ L\ (Bx)] \\
seq\ (SI)\ L\ (\bigvee_\tau\ B) & =_\Delta \exists_\tau x.[seq\ I\ L\ (Bx)]
\end{array}$$

$$\begin{array}{ll}
prog\ (typeof\ (abs\ R)\ (arr\ T\ U)) & \bigwedge_{tm} \lambda N.((typeof\ N\ T) \Rightarrow \langle typeof\ (R\ N)\ U\rangle) \\
prog\ (typeof\ (ap\ M\ N)\ T) & \bigvee_{tm} \lambda U.(\langle typeof\ M\ (arr\ U\ T)\rangle \& \langle typeof\ N\ U\rangle) \\
prog\ ((abs\ R) \Downarrow (abs\ R)) & tt \\
prog\ ((app\ M\ N) \Downarrow V) & \langle M \Downarrow (abs\ R)\rangle \& \langle (R\ N) \Downarrow V\rangle
\end{array}$$

Fig. 3. Definitions of Specification and Object Logics in $FO\lambda^{\Delta\mathbb{N}}$

of the clauses for *prog* ends in $=_\Delta \top$ which is omitted.

To a large extent, the inversion and case analysis theorems we proved in Coq were introduced to provide the possibility to reason in Coq in a manner which corresponds closely to reasoning directly in $FO\lambda^{\Delta\mathbb{N}}$. In particular, they allow us to mimic steps that are directly provided by definitional reflection in $FO\lambda^{\Delta\mathbb{N}}$. For types that cannot be defined inductively in Coq such as *tm*, the axioms expressing distinctness and injectivity of constructors are needed for this kind of reasoning.

To illustrate the correspondence between proofs in $FO\lambda^{\Delta\mathbb{N}}$ and Coq, we discuss how several of the steps in the proof outlined in Sect. 3 correspond to steps in a $FO\lambda^{\Delta\mathbb{N}}$ proof of the same theorem. For instance, application of sequent rules in $FO\lambda^{\Delta\mathbb{N}}$ correspond directly to introduction and elimination rules in Coq. Thus, we can begin the $FO\lambda^{\Delta\mathbb{N}}$ proof of the *sr* theorem similarly to the Coq proof, in this case with applications of $\forall$-R, $\supset$ -R, $\exists$-L at the meta-level, from which we obtain sequent (1) in Sect. 3.

Complete induction is derivable in $FO\lambda^{\Delta\mathbb{N}}$, so using this theorem as well as additional sequent rules allows us to obtain sequent (2) in the $FO\lambda^{\Delta\mathbb{N}}$ proof similarly to how it was obtained in the Coq proof.

It is at this point that the first use of definitional reflection occurs in the $FO\lambda^{\Delta\mathbb{N}}$ proof. Applying def$\mathcal{L}$ to the middle assumption on the left of the sequent arrow in (2), we see that this formula only unifies with the left hand side of the first clause of the definition of sequents in Fig. 3. We obtain

$$\forall k.k < (S\ j') \to (IP\ k), \exists d.[(prog\ (p \Downarrow v)\ d) \wedge (seq\ j'\ nil\ d)], \triangleright_0\langle typeof\ p\ t\rangle \longrightarrow \triangleright_0\langle typeof\ v\ t\rangle.$$

Then applying left sequent rules, followed by def$\mathcal{L}$ on $(prog\ (p \Downarrow v)\ d)$, we get two sequents.

$$\begin{array}{l}\forall k.k < (S\ j') \to (IP\ k), (seq\ j'\ nil\ tt),\\ \quad \triangleright_0\ \langle typeof\ (abs\ R)\ t\rangle \longrightarrow \triangleright_0\langle typeof\ (abs\ R)\ t\rangle\\ \forall k.k < (S\ j') \to (IP\ k), (seq\ j'\ nil\ (\langle P \Downarrow (abs\ R)\rangle \& \langle (R\ N) \Downarrow v\rangle)),\\ \quad \triangleright_0\ \langle typeof\ (app\ P\ N)\ t\rangle \longrightarrow \triangleright_0\langle typeof\ v\ t\rangle\end{array}$$

Like sequent (3) in Sect. 3, the first is directly provable. The def$\mathcal{L}$ rule is applied again, this time on the middle assumption of the second sequent. Only the fourth clause of the definition of sequents in Fig. 3 can be used in unification. Following this application by conjunction elimination yields a sequent very similar to (4). The *eval_nil_inv* theorem used in the Coq proof and applied to (2) at this stage encompasses all three def$\mathcal{L}$ applications. Its proof, in fact, uses three inversion theorems. Note that because of the unification operation, there is no need for existential quantifiers and equations as in the Coq version.

The use of inductive types in Coq together with distinctness and injectivity axioms are sufficient to handle most of the examples in McDowell and Miller's paper [13]. One specification logic given there relies on extensionality of equality which holds in $FO\lambda^{\Delta\mathbb{N}}$. In Coq, however, equality is not extensional, which causes some difficulty in reasoning using this specification logic. Assuming extensional equality at certain types in Coq will be necessary. Other axioms may be needed as well. In the application of the definitional reflection rule, higher-order unification is a central operation. Examples which we cannot handle also include those in McDowell's thesis [12] which require complex uses of such unification. For instance, we cannot handle applications for which there are multiple solutions to a single unification problem.

We have claimed that Coq provides extra flexibility by allowing reasoning via direct induction using the induction principles generated by the system. A simple example of this extra flexibility is in the proof of the *seq_cut* theorem mentioned in Sect. 3. The $FO\lambda^{\Delta\mathbb{N}}$ proof is similar to the Coq proof that does case analysis using the induction principle for *seq*. In Coq, we were also able to do a simpler proof via case analysis provided by structural induction on *prp*, which is not possible in $FO\lambda^{\Delta\mathbb{N}}$. The next section illustrates other examples of this extra flexibility.

5 Structural Induction on Sequents

In this section, we discuss an alternate proof of theorem sr that uses Coq's induction principle for seq. Since we do not do induction on the height of the proof for this example, the natural number argument is not needed, so we omit it and use the definition given in Fig. 4 instead.

$$
\begin{array}{l}
\mathsf{Inductive}\ seq : list\ atm \to prp \to Prop := \\
\quad sbc : \forall A : atm. \forall L : list\ atm. \forall b : prp. \\
\qquad\quad (prog\ A\ b) \to (seq\ L\ b) \to (seq\ L\ \langle A \rangle) \\
\quad \mid sinit : \forall A, A' : atm. \forall L : list\ atm. \\
\qquad\quad (element\ A\ (A' :: L)) \to (seq\ (A' :: L)\ \langle A \rangle) \\
\quad \mid strue : \forall L : list\ atm. (seq\ L\ tt) \\
\quad \mid sand : \forall B, C : prp. \forall L : list\ atm. \\
\qquad\quad (seq\ L\ B) \to (seq\ L\ C) \to (seq\ L\ (B \& C)) \\
\quad \mid simp : \forall A : atm. \forall B : prp. \forall L : list\ atm. \\
\qquad\quad (seq\ (A :: L)\ B) \to (seq\ L\ (A \Rightarrow B)) \\
\quad \mid sall : \forall B : tau \to\ prp. \forall L : list\ atm. \\
\qquad\quad (\forall x : tau. (seq\ L\ (B\ x))) \to (seq\ L\ (\bigwedge B)) \\
\quad \mid ssome : \forall B : tau \to prp. \forall L : list\ atm. \\
\qquad\quad \forall x : tau. (seq\ L\ (B\ x)) \to (seq\ L\ (\bigvee\ B)).
\end{array}
$$

Fig. 4. Definition of the Specification Logic without Natural Numbers

Note that in the statement of the sr theorem, all of the sequents have empty assumption lists and atomic formulas on the right. Using the induction principle for seq to prove such properties often requires generalizing the induction hypothesis to handle sequents with a non-empty assumption list and a non-atomic formula on the right. We provide an inductive definition to facilitate proofs that require these extensions and we parameterize this definition with two properties, one which represents the desired property restricted to atomic formulas (denoted here as P), and one which represents the property that must hold of formulas in the assumption list (denoted here as $Phyp$). We require that the property on atomic formulas follows from the property on assumptions, so that for the base case when the $sinit$ rule is applied to a sequent of the form $(seq\ L\ A)$, it will follow from the fact that A is in L that the desired property holds. (In many proofs, the two properties are the same, which means that this requirement is trivially satisfied.) The following are the two properties that we use in the proof of sr.

$$
\begin{array}{l}
\mathsf{Definition}\ P := \lambda l : list\ atm. \lambda A : atm. \mathsf{Cases}\ A\ of \\
\quad (typeof\ m\ t) \Rightarrow True \\
\quad \mid (p \Downarrow v) \Rightarrow \forall t : tm. (seq\ l\ \langle typeof\ p\ t \rangle) \to (seq\ l\ \langle typeof\ v\ t \rangle)\ \mathsf{end}. \\
\mathsf{Definition}\ Phyp := \lambda l : list\ atm. \lambda A : atm. \exists p, t : tm. A = (typeof\ p\ t) \wedge \\
\quad (seq\ nil\ \langle A \rangle) \wedge (\forall v : tm. (seq\ l\ \langle p \Downarrow v \rangle) \to (seq\ l\ \langle typeof\ v\ t \rangle)).
\end{array}
$$

The proof of sr (in this and in the previous section) uses induction on the height of the proof of the evaluation judgment $\rhd_0 \langle p \Downarrow v \rangle$. Thus in defining P,

we ignore *typeof* judgments. The clause for $\Downarrow$ simply states a version of the subject reduction property but with assumption list l. The property that we require of assumption lists is that they only contain atomic formulas of the form $(typeof\ p\ t)$ and that each such assumption can be proven from an empty set of assumptions and itself satisfies the subject reduction property.

The inductive definition which handles generalized induction hypotheses (parameterized by P and $Phyp$) is given in Fig. 5. The definition of $mapP$ mimics

$$
\begin{array}{l}
\mathsf{Variable}\ P : list\ atm \to atm \to Prop. \\
\mathsf{Variable}\ Phyp : list\ atm \to atm \to Prop. \\
\mathsf{Inductive}\ mapP : list\ atm \to prp \to Prop := \\
\quad mbc : \forall A : atm. \forall L : list\ atm. (P\ L\ A) \to (mapP\ L\ \langle A \rangle) \\
\quad \mid minit : \forall A : atm. \forall L : list\ atm. (element\ A\ L) \to (mapP\ L\ \langle A \rangle) \\
\quad \mid mtrue : \forall L : list\ atm. (mapP\ L\ tt) \\
\quad \mid mand : \forall B, C : prp. \forall L : list\ atm. \\
\qquad\quad (mapP\ L\ B) \to (mapP\ L\ C) \to (mapP\ L\ (B \& C)) \\
\quad \mid mimp : \forall A : atm. \forall B : prp. \forall L : list\ atm. \\
\qquad\quad ((Phyp\ L\ A) \to (mapP\ (A :: L)\ B)) \to (mapP\ L\ (A \Rightarrow B)) \\
\quad \mid mall : \forall B : tau \to\ prp. \forall L : list\ atm. \\
\qquad\quad (\forall x : tau. (mapP\ L\ (B\ x))) \to (mapP\ L\ (\bigwedge B)) \\
\quad \mid msome : \forall B : tau \to prp. \forall L : list\ atm. \\
\qquad\quad \forall x : tau. (mapP\ L\ (B\ x)) \to (mapP\ L\ (\bigvee\ B)).
\end{array}
$$

Fig. 5. A Definition for Extending Properties on Atoms to Properties on Propositions

the definition of seq except where atomic properties appear. For example, the clause $mand$ can be read as: if the generalized property holds for sequents with arbitrary propositions B and C under assumptions L, then it holds for their conjunction under the same set of assumptions. In mbc, the general property holds of an atomic proposition under the condition that the property P holds of the atom. In $minit$, the general property holds simply because the atom is in the list of assumptions. The only other clause involving an atom is $mimp$. The general property holds of an implication $(A \Rightarrow B)$ as long as whenever $Phyp$ holds of A, the general property holds of B under the list of assumptions extended with A.

Using the definition of $mapP$, we can structure proofs of many properties so that they involve a direct induction on the definition of the specification logic, and a subinduction on $prog$ for the atomic formula case. The theorem below takes care of the first induction.

$$
\begin{array}{l}
\mathsf{Definition}\ PhypL := \lambda L : list\ atm. (\forall a : atm. (element\ a\ L) \to (Phyp\ L\ a)). \\
\mathsf{Theorem}\ seq\_mapP : \\
\quad (\forall L : list\ atm. \forall A : atm. (PhypL\ L) \to \\
\qquad (Phyp\ L\ A) \to \forall A' : atm. (element\ A'\ (A :: L)) \to (Phyp\ (A :: L)\ A')) \to \\
\quad (\forall L : list\ atm. \forall A : atm. \forall b : prp. (PhypL\ L) \to \\
\qquad (prog\ A\ b) \to (seq\ L\ b) \to (mapP\ L\ b) \to (P\ L\ A)) \to \\
\quad \forall l : list\ atm. \forall B : prp. (PhypL\ l) \to (seq\ l\ B) \to (mapP\ l\ B).
\end{array}
$$

The first two lines of the seq_mapP statement roughly state that $Phyp$ must be preserved as new atomic formulas are added to the list of assumptions. Here,

$PhypL$ states that $Phyp$ holds of all elements of a list. More specifically, these lines state that whenever $(Phyp\ L\ A')$ holds for all A' already in L, and it is also the case that $(Phyp\ L\ A)$ holds for some new A, then $(Phyp\ (A :: L)\ A')$ also holds for every A' in the list L extended with A. The next two lines of the theorem state the base case, which is likely to be proved by a subinduction on $prog$. Under these two conditions, we can conclude that the generalized property holds of l and B whenever $Phyp$ holds of all assumptions in l.

For the new proof of sr, the definitions of prp and $prog$ remain exactly as in Sect. 3, as do the definitions of the parameters that represent syntax along with their distinctness and injectivity axioms. Inversion theorems such as seq_atom_inv and $eval\_nil\_inv$ are stated and proved similarly as in the previous section, but without the additional natural number arguments.

Now we can state the generalized version of the sr property which is just:

$$\begin{array}{l}\textsf{Theorem}\ sr\_mapP : \forall L : list\ atm.\forall B : prp.\\ \quad (seq\ L\ B) \rightarrow (PhypL\ L) \rightarrow (mapP\ L\ B).\end{array}$$

To prove this theorem, we directly apply seq_mapP. The proof that $Phyp$ is preserved under the addition of new assumptions is straightforward. The base case for atomic formulas is proved by induction on $prog$. This induction gives us four cases. The two cases which instantiate atom A from seq_mapP to formulas of the form $(typeof\ m\ t)$ cause $(P\ L\ A)$ to be reduced to $True$. The details of the two cases for $(prog\ (p \Downarrow v)\ b)$ are similar to the corresponding cases in the proof in Sect. 3.

Finally, we can show that the desired sr theorem is a fairly direct consequence of sr_mapP.

$$\begin{array}{l}\textsf{Theorem}\ sr : \forall p, v : tm.(seq\ nil\ \langle p \Downarrow v\rangle) \rightarrow\\ \quad \forall t : tm.(seq\ nil\ \langle typeof\ p\ t\rangle) \rightarrow (seq\ nil\ \langle typeof\ v\ t\rangle).\end{array}$$

The proof begins with an application of the sr_mapP theorem to conclude that $(mapP\ nil\ \langle p \Downarrow v\rangle)$ holds. Now note that the only way such a formula can hold is by the first clause of the definition of $mapP$ since the assumption list is empty and the formula is atomic. This fact is captured by the following inversion theorem.

$$\textsf{Theorem}\ mapP\_nil\_atom\_inv : \forall A : atm.(mapP\ nil\ \langle A\rangle) \rightarrow (P\ nil\ A).$$

Applying this theorem and expanding P, we can conclude that

$$\forall t : tm.(seq\ nil\ \langle typeof\ p\ t\rangle) \rightarrow (seq\ nil\ \langle typeof\ p\ v\rangle)$$

which is exactly what is needed to complete the proof.

We have also used the $mapP$ definition to prove the following theorem for a functional language which includes app and abs as well as many other primitives such as booleans, natural numbers, a conditional statement, and a recursion operator:

$$\begin{array}{l}\textsf{Theorem}\ type\_unicity : \forall M, t : tm(seq\ nil\ \langle typeof\ M\ t\rangle) \rightarrow\\ \quad \forall t' : tm(seq\ nil\ \langle typeof\ M\ t'\rangle) \rightarrow (equiv\ t\ t').\end{array}$$

where the *equiv* predicate is defined simply as

$$\mathsf{Inductive}\ equiv : tm \to tm \to Prop := refl : \forall t : tm(equiv\ t\ t).$$

To do this proof, we of course had to first define P and $Phyp$ specialized to this theorem. We do not discuss the details here.

6 Conclusion, Related Work, and Future Work

We have described a methodology for proving properties of objects expressed using higher-order syntax in Coq. Because of the similarity of reasoning in Coq and reasoning in the object logic we use in our PCC system, we hope to be able to carry over this methodology to the PCC setting. In particular, in both our λProlog and Twelf prototypes, we use an object logic that is currently specified directly, but could be specified as *prog* clauses, allowing the kind of reasoning described here.

In addition to our practical goal of building proofs in the PCC domain as well as other domains which use meta-theoretical reasoning about logics and programming languages, another goal of this paper was to provide insight into how a class of proofs in the relatively new logic $FO\lambda^{\Delta\mathbb{N}}$ correspond to proofs in a class of logics that have been around somewhat longer, namely logics that contain dependent types and inductive definitions.

Certainly, many more examples are needed to illustrate that our approach scales to prove all properties that we are interested in. In addition to carrying out more examples, our future work includes providing more flexible support for reasoning in this setting. The *mapP* predicate in Sect. 5 was introduced to provide one kind of support for induction on sequents. Other kinds of support are worth exploring. For example, we have started to investigate the possibility of generating induction principles for various object-level predicates such as $typeof$. One goal is to find induction principles whose proofs would likely use *mapP* and *seq*, but when they are used in proofs, all traces of the middle layer specification logic would be absent.

As in any formal encoding of one system in another, we need to express and prove adequacy theorems for both the specification and object-level logics. Proofs of adequacy of these encodings should follow similarly to the one for the π-calculus in Honsell et al. [11]. Such a proof would require that we do not discharge the type *tm* in Coq, thus preventing it from being instantiated with an inductively defined type, which could violate adequacy.

In related work, there are several other syntactic approaches to using higher-order syntax and induction in proofs, which either do not scale well, or more work is needed to show that they can. For example, Coq was used by Despeyroux et al. [5] to do a proof of the subject reduction property for the untyped λ-calculus. There, the problem of negative occurrences in definitions used for syntax encodings was handled by replacing such occurrences by a new type. As a result, some additional operations were needed to encode and reason about these types, which at times was inconvenient. Miculan uses a similar approach

to handling negative occurrences in formalizing meta-theory of both modal μ-calculus [15] and the lazy call-by-name λ-calculus [14] in Coq. These proofs require fairly extensive use of axioms, more complex than those used here, whose soundness are justified intuitively.

Honsell et al. [11] and Despeyroux [4] define a higher-order encoding of the syntax of the π-calculus in Coq and use it formalize various aspects of the metatheory. Although they use higher-order syntax to encode processes, there is no negative occurrence of the type being defined, and so they are able to define processes inductively.

Despeyroux and Hirschowitz [6] studied another approach using Coq. Again, a different type replacing negative occurrences is used, but instead of directly representing the syntax of (closed) terms of the encoded language by terms of types such as tm, closed and open terms of the object language are implemented together as functions from lists of arguments (of type tm) to terms of type tm. Examples are described, but it is not clear how well the approach scales.

Hofmann [10] shows that a particular induction principle for tm, which is derived from a straightforward extension of the inductive types in Coq to include negative occurrences, can be justified semantically under certain conditions (though not within Coq). Although he cannot prove the subject reduction property shown here, he shows that it is possible to express a straightfoward elegant proof of a different property: that every typed λ-term reduces to a term in weak-head normal form.

In Pfenning and Rohwedder [18], the technique of schema checking is added to the Elf system, a precursor to the Twelf system mentioned earlier. Both systems implement the Logical Framework (LF) [9]. Induction cannot be expressed in LF, so proofs like those shown here cannot be fully formalized inside the system. However, each of the cases of a proof by induction can be represented. The schema checking technique works outside the system and checks that all cases are handled.

Despeyroux et al. [7] present a λ-calculus with a modal operator which allows primitive recursive functionals over encodings with negative occurrences. This work is a first step toward a new type theory that is powerful enough to express and reason about deductive systems, but is not yet powerful enough to handle the kinds of theorems presented here.

Schürmann [20] has developed a logic which extends LF with support for meta-reasoning about object logics expressed in LF. It has been used to prove the Church-Rosser theorem for the simply-typed λ-calculus and many other examples. The design of the component for reasoning by induction does not include induction principles for higher-order encodings. Instead, it is based on a realizability interpretation of proof terms. This logic has been implemented in Twelf, and includes powerful automated support for inductive proofs.

References

[1] A. W. Appel and A. P. Felty. Lightweight lemmas in λProlog. In *International Conference on Logic Programming*, Nov. 1999.

[2] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.

[3] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of SIGSOFT'88: Third Annual Symposium on Software Development Environments (SDE3)*, Boston, 1988.

[4] J. Despeyroux. A higher-order specification of the π-calculus. In *First IFIP International Conference on Theoretical Computer Science*. Springer-Verlag Lecture Notes in Computer Science, 2000.

[5] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*. Springer-Verlag Lecture Notes in Computer Science, Apr. 1995.

[6] J. Despeyroux and A. Hirschowitz. Higher-order syntax and induction in coq. In *Fifth International Conference on Logic Programming and Automated Reasoning*. Springer-Verlag Lecture Notes in Computer Science, 1994.

[7] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications*. Springer-Verlag Lecture Notes in Computer Science, 1997.

[8] L.-H. 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*. Springer-Verlag Lecture Notes in Artificial Intelligence, 1992.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1), Jan. 1993.

[10] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Fourteenth Annual Symposium on Logic in Computer Science*, 1999.

[11] F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theories. *Theoretical Computer Science*, 253(2), 2001.

[12] R. McDowell. *Reasoning in a Logic with Definitions and Induction*. PhD thesis, University of Pennsylvania, December 1997.

[13] R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1), Jan. 2002.

[14] M. Miculan. Developing (meta)theory of λ-calculus in the theory of contexts. *Electronic Notes on Theoretical Computer Science*, 58, 2001.

[15] M. Miculan. On the formalization of the modal μ-calculus in the calculus of inductive constructions. *Information and Computation*, 164(1), 2001.

[16] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*. MIT Press, 1988.

[17] G. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Jan. 1997.

[18] F. Pfenning and E. Rohwedder. Implementing the meta-theory of deductive systems. In *Eleventh International Conference on Automated Deduction*, volume 607. Lecture Notes in Computer Science, 1992.

[19] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Sixteenth International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1999.

[20] C. Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000.

[21] The Coq Development Team. The Coq Proof Assistant reference manual: Version 7.2. Technical report, INRIA, 2002.