INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Higher-order abstract syntax in Coq

Joëlle Despeyroux , Amy Felty et André Hirschowitz

N° 2556

Mai 1995

PROGRAMME 2

apport de recherche

ISSN 0249-6399

Higher-order abstract syntax in Coq

Joëlle Despeyroux *, Amy Felty ** et André Hirschowitz ***

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Croap

Rapport de recherche n° 2556 — Mai 1995 — 18 pages

Abstract: The terms of the simply-typed λ-calculus can be used to express the *higher-order abstract syntax* of objects such as logical formulas, proofs, and programs. Support for the manipulation of such objects is provided in several programming languages (e.g. λProlog, Elf). Such languages also provide embedded implication, a tool which is widely used for expressing *hypothetical judgments* in natural deduction. In this paper, we show how a restricted form of second-order syntax and embedded implication can be used together with induction in the Coq Proof Development system. We specify typing rules and evaluation for a simple functional language containing only function abstraction and application, and we fully formalize a proof of type soundness in the system. One difficulty we encountered is that expressing the higher-order syntax of an object-language as an inductive type in Coq generates a class of terms that contains more than just those that directly represent objects in the language. We overcome this difficulty by defining a predicate in Coq that holds only for those terms that correspond to programs. We use this predicate to express and prove the adequacy for our syntax.

Key-words: Higher-order abstract syntax, Coq, theorem proving, logical framework, type theory, λ-calculus.

(Résumé : tsvp)

This article appeared in M. Dezani and G. Plotkin, editors, Proceedings of the international conference on Typed Lambda Calculi and Applications (TLCA), volume 902, pages 124-138. Springer-Verlag Lecture Notes in Computer Science, 1995.

*INRIA, Sophia-Antipolis. Email: joelle.despeyroux@sophia.inria.fr
**AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974, USA. felty@research.att.com
***CNRS URA 168, University of Nice, 06108 Nice Cedex 2, France. andre.hirschowitz@sophia.inria.fr

Syntaxe abstraite d'ordre supérieur en Coq

Résumé : Les termes du λ-calcul simplement typé peuvent être utilisés pour décrire en syntaxe abstraite d'ordre supérieur des objets tels que des formules logiques, des preuves ou des programmes. Des langages de programmation tels que λProlog et Elf permettent de manipuler ces objets. Ces langages permettent aussi d'employer l'implication plongée ('embedded implication'), possibilité largement exploitée pour exprimer les jugements de la déduction naturelle. Dans cet article, nous montrons comment utiliser une forme restreinte de syntaxe abstraite d'ordre deux et d'implication, conjointement avec l'induction, dans le système de développement de preuves Coq. Nous spécifions les règles de typage et d'évaluation d'un langage fonctionnel simple contenant seulement l'abstraction et l'application et nous formalisons une preuve de préservation des types dans le système. Le problème est que la syntaxe d'un langage objet donnée à l'ordre supérieur par un type inductif génère trop de termes. La solution présentée ici consiste à définir un prédicat Coq vrai sur les seuls termes qui correspondent à des programmes. Nous utilisons ce prédicat pour exprimer et prouver l'adéquation de notre syntaxe.

Mots-clé : Syntaxe abstraite d'ordre supérieur, Coq, théories typées, λ-calcul.

1 Introduction

Abstraction in the λ-calculus can be used to represent various binding operators such as quantification in formulas or abstraction in functional programs. By making use of the implementation of the λ-calculus in programming languages that support it, the programmer is freed from such concerns as implementing substitution algorithms and correctly handling the scope and names of bound variables. Many examples exist and illustrate the usefulness of higher-order syntax in programming and theorem proving. For example, the Logical Framework (LF) [10] provides a uniform framework for specifying a large class of languages and inference systems. A variety of logics and typed λ-calculi have been specified using it [2]. Theorem provers for several of these logics have been specified and implemented in the logic programming language λProlog, which provides support for the manipulation of objects expressed in higher-order syntax [5, 6]. The λProlog language has also been used to specify program evaluators and transformers [7, 8]. Elf, a logic programming implementation of LF, has been used to specify and verify properties of inference systems [11, 14] and compilers [9]. In many of these examples, embedded implication (*i.e.*, an implication on the left of an implication) is used, providing an elegant mechanism for handling scoping of sets of assumptions during proof construction, or of contexts during program evaluation.

Higher-order syntax and hypothetical judgments can be expressed in many theorem provers. However, there is little experience using them in proofs. In this paper, we illustrate the use of a restricted form of second-order syntax and embedded implication in the Coq Proof Development system [4] by defining typing rules and evaluation for a simple functional language containing only function abstraction and application. We prove type soundness for this language, *i.e.*, that evaluating a term preserves its type. By using this syntax much of the details of proofs, in particular those concerning substitution and names and scopes of variables, are greatly simplified. In addition, this work represents a step towards the goal of providing support for higher-order abstract syntax and allowing both programming and program verification in a unified setting.

We have chosen the Coq Proof Development System because it implements the Calculus of Inductive Constructions (CIC) [13], a type theory which provides a notion of inductive definitions. Defining a type inductively provides a principle of structural induction and an operator for defining functions recursively over the type. These operators can be used directly and there are no requirements placed on the user to prove their correctness. However, in order to use the built-in support for induction, we had to overcome two obstacles.

The first obstacle is that negative occurrences of the type being defined are not allowed in inductive definitions. If L is the type of terms of the language being defined, the usual way to express the higher-order syntax of an abstraction operator such as function abstraction in our example is to introduce a constant such as Lam and assign it the type $(L \rightarrow L) \rightarrow L$. That is, Lam takes one argument of functional type. Thus function abstraction in the object-language is expressed using λ-abstraction in the meta-language. As a result, bound variables in the object-language are identified with bound variables in the meta-language. In inductive types in Coq, negative occurrences such as the first occurrence of L in the above type are disallowed. As in [3], we get around this problem by introducing a separate type *var* for variables and giving *Lam* the type $(var \rightarrow L) \rightarrow L$. We must then add a constructor for injecting variables into terms of L. Thus, in our restricted form of higher-order syntax, we still define function abstraction using λ-abstraction in Coq and it is still the case that α-convertible terms in our object-language map to α-convertible terms in Coq, but we cannot directly define object-level substitution using Coq's β-reduction. Instead we define substitution as an inductive predicate. Its definition is simple and similar to the one found in [12].

The second obstacle is that defining the type L as an inductive type with the usual constructors for application and abstraction plus the special constructor for variables gives a set of terms in Coq that is "too large". That is, there are more terms in L than those that correspond to objects in the object-language. To solve this problem for our functional language, we succeeded in the task of defining an object-level predicate, which we call *valid*, that is true only for those terms that correspond to programs. This predicate, however, does allow some terms that do not directly represent programs, namely, those that are extensionally equal to terms that do. We define extensional equality for the type L in Coq, and consider that each term of the object-language is in fact represented by an equivalence class of terms determined by this equality relation.

This work extends two related projects where higher-order syntax is used in formal proof. In [9], Elf is used in compiler verification. In Elf, there is no quantification over predicates, and thus induction principles cannot be expressed inside the language. As a result, much of the detail of proofs must be done outside the system. Tools such as schema-checking [14] have been developed to help with this task. In [3], a different approach to higher-order syntax in Coq is adopted. There, like here, a separate type *var* for variables is introduced and *Lam* is defined as above. However, instead of directly representing (closed) terms of the object-language by terms of type L, closed and open terms of the object-language are implemented together as functions from lists of arguments (of type L) to terms of type L. Semantics are given on these functional terms. A predicate on these terms is introduced which defines valid terms to be the expected ones. Induction over terms is carried out by using the induction principle for this predicate. Here, we instead define typing and evaluation directly on (closed) terms of type L. We succeed in reasoning about them by directly using the induction principles generated by their definitions.

The rest of the paper is organized as follows. In Section 2, we define the higher-order syntax of our functional language in Coq. In Sections 3, 4, 5, and 6, we show that our syntax adequately represents our object-language. In Section 3, we give a definition of our object-language in LF for which adequacy has already been proved, and in Section 4, we express a translation between the LF and Coq syntaxes. In Section 5, we explain which kind of terms should be ruled out and the need for extensionality. In Section 6, we implement the predicate *valid* which selects terms that represent terms of the object-language and prove the correctness of this implementation. In Section 7, we define and implement substitution in Coq and prove its correctness. Although the definitions of *valid* and substitution are simple, finding them was one of the main challenges of this work. In defining them, and in the proofs in this paper, we succeeded in avoiding any need to refer to variable names or occurrences of variables in terms, or a notion of *fresh* variables not occurring in terms. Section 8 presents the Coq definitions for typing and evaluation in the object-language, which specify the usual natural semantics style presentation of these judgments, and discusses the Coq proof of type soundness. In Section 9, we conclude and discuss future work.

Notation. In proving the adequacy of our representation and correctness of substitution, we will often use notation close to the syntax of Coq. To make the distinction, for those definitions or statements not intended to be Coq or LF definitions, we will use (*) as a superscript on Coq keywords.

2 Specifying Provisional Syntax

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

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

Here $\forall$ is the dependent type constructor and the arrow ($\rightarrow$) is the usual abbreviation when the bound variable does not occur in the body. Of the remaining constants, $Prop$, Set, $Type$, λ, Rec, and $Case$ are primitive, while the others are defined. Rec and $Case$ are the operators for defining inductive ($Case$) and recursive (Rec) functions over inductive types. Equality on Set ($=$) is Leibniz equality.

A parameter is introduced using the Parameter keyword and inductive types are introduced with an Inductive Set or Inductive Definition declaration where each constructor is given with its type, separated by vertical bars.

We specify a provisional syntax for our object-language, the λ-calculus, by introducing a type for variables and defining terms and types inductively.

Parameter $var : Set$.
Inductive Set $L\ =$
  $Var : var \rightarrow L \ \mid\ App : L \rightarrow L \rightarrow L \ \mid\ Lam : (var \rightarrow L) \rightarrow L.$
Inductive Set $tL\ =\ TVar : var \rightarrow tL \ \mid\ Arrow : tL \rightarrow tL \rightarrow tL.$

For instance, $(Lam\ (\lambda x : var.\ (App\ (Var\ x)\ (Var\ x))))$ encodes the function $\lambda x.(x\ x)$. This syntax is provisional since, although it is clear how to encode each term of the object-language as a term of type L, for most instantiations of the type var, the type L contains *exotic* terms, that is, terms that do not encode any λ-term. Describing these terms and finding a way to rule them out is the subject of the next few sections.

The following induction principle for L is generated by the system and proven automatically. It illustrates a general form of induction over higher-order syntax.

$$\begin{array}{l}\forall P : L \rightarrow Prop. \\ \quad (\forall v : var.(P\ (Var\ v))) \rightarrow \\ \quad (\forall m, n : L.(P\ m) \rightarrow (P\ n) \rightarrow (P\ (App\ m\ n))) \rightarrow \\ \quad (\forall E : var \rightarrow L.(\forall v : var.(P\ (E\ v))) \rightarrow (P\ (Lam\ E))) \rightarrow \\ \ \forall e : L.(P\ e).\end{array}$$

By asserting var as a parameter, the theorems we prove will hold for any instantiation of this type. The important ones to consider will be those that satisfy any axioms we assert which make assumptions about this type. All those that we will need here should follow from the var_nat assumption below which asserts a surjective mapping from var to the natural numbers.

Inductive Set $nat \ = \ 0 : nat \ \mid \ S : nat \rightarrow nat$
Axiom $var\_nat : \exists s : var \rightarrow nat.\ \forall n : nat.\ \exists v : var.\ (s\ v) = n.$

3 Specifying Syntax in LF

To prove that our syntax adequately represents our object-language, we begin with an LF signature for the λ-calculus, for which adequacy has already been proven [2]. In LF, the syntax is introduced simply by declaring the type l_0 for terms and two constructors for application and abstraction.

$$\begin{array}{l} l_0 : Type \\ app_0 : l_0 \rightarrow l_0 \rightarrow l_0 \\ lam_0 : (l_0 \rightarrow l_0) \rightarrow l_0. \end{array}$$

The set of ($\alpha\beta\eta$-equivalence classes of) LF terms generated by this signature has three important properties, which we state informally as follows:

1. All terms of the object-language can be represented (as trees) using only the two constructors (induction principle).
2. Each term has a unique representation (injection principle).
3. Any two terms that are extensionally equal are also equal (extensionality principle).

The formulation of these principles, which we will not give here, involves LF terms of type l_0, $l_0 \rightarrow l_0$, $l_0 \rightarrow l_0 \rightarrow l_0$, etc. We will use this sequence of types in the translation of our object-language from the LF representation to the Coq representation in the next section. Thus we give a formal definition:

Definition* $l_n := if\ n = 0\ then\ l_0\ else\ l_0 \rightarrow l_{n-1}.$

In the context of this sequence of types, instead of the original two constructors, the induction and injection principles involve what we call the higher-order constructors, which are defined as follows:

Definition* $ref = \lambda n : nat.\lambda i \in [0..n-1].\lambda x_{n-1}, \ldots, x_0 : l_0.x_i$
Definition* $app = \lambda n : nat.$
$\lambda e, e' : l_n.\lambda x_{n-1}, \ldots, x_0 : l_0.(app_0\ (e\ x_{n-1}\ \ldots\ x_0)\ (e'\ x_{n-1}\ \ldots\ x_0))$
Definition* $lam = \lambda n : nat.Case\ n\ of$
$0 \Rightarrow \lambda e : l_1.\ (lam_0\ e)$
$(S\ m) \Rightarrow \lambda e : l_{m+2}.\ \lambda x_m, \ldots, x_0 : l_0.(lam_0\ (e\ x_m\ \ldots\ x_0)).$

These higher-order constructors will allow us to give a very simple translation from the above LF syntax into Coq (see below). Note that they are not LF terms. However, for each $n \geq 0$ and $i < n$, the terms $(ref\ n\ i)$, $(app\ n)$, and $(lam\ n)$, which we abbreviate as $ref_{n,i}$, app_n, and lam_n, are LF terms. These three families of higher-order λ-terms have natural interpretations in any Cartesian closed category with reflexive objects (cf [1] definition 9.3.1 page 219). The interpretation there is highly semantic in nature. Our purpose here is syntax, and our motivation for the above definitions is not the use of object-level β-reduction.

In [3], our provisional syntax was used to yield and manipulate an implementation of l_0, and in fact of the l_n's. There, adequacy caused no problem, but the final syntax and semantics were invaded by (object-level) lists. Here we will succeed in implementing syntax and semantics without object-level lists.

4 Translation Between LF and Coq Syntaxes

In order to express our translation we define the Coq counterpart of the types l_n and the corresponding higher-order constructors.

Definition* $L_n := if\ n = 0\ then\ L\ else\ var \to L_{n-1}.$
Definition* $\mathcal{R}ef = \lambda n : nat.\lambda i \in [0..n-1].\lambda x_{n-1}, \ldots, x_0 : var.(Var\ x_i)$
Definition* $\mathcal{A}pp = \lambda n : nat.\lambda e, e' : L_n.\lambda x_{n-1}, \ldots, x_0 : var.$
$(App\ (e\ x_{n-1}\ \ldots\ x_0)\ (e'\ x_{n-1}\ \ldots\ x_0))$
Definition* $\mathcal{L}am = \lambda n : nat.Case\ n\ of$
$0 \Rightarrow \lambda e : L_1.\ (Lam\ e)$
$(S\ m) \Rightarrow \lambda e : L_{m+2}.\lambda x_m, \ldots, x_0 : var.(Lam\ (e\ x_m\ \ldots\ x_0)).$

As before, we use abbreviations $\mathcal{R}ef_{n,i}$, $\mathcal{A}pp_n$, and $\mathcal{L}am_n$.

To show the correspondence between Coq terms of type L and LF terms of type l_0, we begin by defining the following natural translation $Trans$ from the l_n's into the L_n's.

Inductive Definition* $Trans : \forall n : nat.\ l_n \to L_n \to Prop$
$= Trans\_ref : \forall n : nat.\forall i \in [0..n-1].(Trans\ n\ ref_{n,i}\ \mathcal{R}ef_{n,i})$
$\mid Trans\_app : \forall n : nat.\forall a, b : l_n.\forall a', b' : L_n.\ (Trans\ n\ a\ a') \to$
$(Trans\ n\ b\ b') \to (Trans\ n\ (app_n\ a\ b)\ (\mathcal{A}pp_n\ a'\ b'))$
$\mid Trans\_lam : \forall n : nat.\forall e :\ l_{n+1}.\forall f :\ L_{n+1}.\ (Trans\ (n+1)\ e\ f) \to$
$(Trans\ n\ (lam_n\ e)\ (\mathcal{L}am_n\ f)).$

Proposition*: The above definition $Trans$ defines, for each n, an injective map from l_n to L_n.

The proof of this proposition relies heavily on the induction and injection principles mentioned earlier. For a similar statement and a fully mechanized proof of it, see [3]. Our next task is to characterize the image of this translation, which in fact is the subset of terms in L_n that specify λ-terms, and thus are the terms we are interested in. The natural definition that selects this subset is the following one (see [3]):

$$\begin{array}{l} \textsf{Inductive Definition}^* \; Valid : \forall n : nat.\; L_n \rightarrow Prop \\ \quad = Valid\_ref : \forall n, i : nat.(i < n) \rightarrow (Valid\ n\ \mathcal{R}ef_{n,i}) \\ \quad \mid Valid\_app : \forall n : nat.\forall e, e' : L_n. \\ \qquad (Valid\ n\ e) \rightarrow (Valid\ n\ e') \rightarrow (Valid\ n\ (\mathcal{A}pp_n\ e\ e')) \\ \quad \mid Valid\_lam : \forall n : nat.\forall e :\ L_{n+1}.\ (Valid\ (n+1)\ e) \rightarrow (Valid\ n\ (\mathcal{L}am_n\ e)). \end{array}$$

Indeed, we have the following result, whose proof is straightforward.

Theorem* : For each integer n, $Trans$ yields a bijection between terms of type l_n and terms of type L_n satisfying $(Valid\ n)$.

Note that $Valid$ is not a Coq predicate. Furthermore, it is not clear how to define a Coq predicate for each instance of n without using the definition for $n+1$ (see the $Valid\_lam$ case). This is precisely the task we will turn to now, at least for $n = 0$. We will succeed only modulo extensionality (see Section 6). Note that the $Valid\_ref$ case becomes irrelevant when $n = 0$ and thus only closed terms satisfy $(Valid\ 0)$. Our solution will replace the proposition $(Valid\ (n+1)\ e)$ with an equivalent one that depends on n instead of $n+1$, thus allowing us to drop n altogether. One obvious candidate is $\forall v : var.(Valid\ n\ (e\ v))$. However, as we will see, this is not sufficient and does not rule out all the necessary terms.

5 Exotic Terms, Extensionality and Extended Validity

There are three kinds of exotic terms that the type L may contain. We illustrate by instantiating var to nat. In this case, we have irreducible functional terms of type $nat \rightarrow L$ that use a $Case$ operator. Exotic terms of type L are generated through the Lam constructor.

The first kind of exotic term is illustrated by the following term:

$$exot_1 = (Lam\ \lambda x : nat.Case\ x : nat\ of\ 0 \Rightarrow (Var\ 0)\ \ (S\ n) \Rightarrow (Var\ (S\ n))).$$

The above term is extensionally equal to the term $(Lam\ \lambda x : nat.(Var\ x))$, and thus we could accept it as a well-formed term. Extensional equality is defined as the following Coq definition.

Inductive Definition $eq_L : L \rightarrow L \rightarrow Prop$
$= \ eq_L\_var : \forall x : var(eq_L\ (Var\ x)\ (Var\ x))$
$\mid \ eq_L\_app : \forall m_1, m_2, n_1, n_2 : L.$
$(eq_L\ m_1\ n_1) \rightarrow (eq_L\ n_2\ m_2) \rightarrow (eq_L\ (App\ m_1\ m_2)\ (App\ n_1\ n_2))$
$\mid \ eq_L\_lam : \forall M, N : var \rightarrow L.$
$(\forall x : var.(eq_L\ (M\ x)\ (N\ x))) \rightarrow (eq_L\ (Lam\ M)\ (Lam\ N)).$

It will be difficult to define predicates which are able to distinguish $Valid$-terms from terms extensionally equal to them. For instance our predicate *subst* (see below) does not. We circumvent this problem by considering that a λ-term is in fact represented by an equivalence class of terms for this eq_L relation.

The second kind of exotic terms are the *open* ones, namely those with "free variables" such as $exot_2 = (Var\ (S\ 0))$. These open terms will play a role in our approach; we will first introduce a meta-level predicate $Valid\_v$, a slight modification of $Valid$ allowing open terms; we will then succeed in defining a Coq predicate *valid* which implements $(Valid\_v\ 0)$ up to extensionality.

The third kind of exotic term is more problematic and we definitely want our *valid* predicate to discard them. These are terms that contain functions that are not "uniform" in their argument. For example, let $exot_3 :=$

$$\lambda f, x : nat.Case\ x : nat\ of\ 0 \Rightarrow (Var\ x)\ \ (S\ n) \Rightarrow (App\ (Var\ f)\ (Var\ n)).$$

The term $(Lam\ \lambda f : nat.(Lam\ \lambda x : nat.(exot_3\ f\ x)))$ does not represent a λ-term.

In order to integrate the first kind of exotic term, we define the meta-level predicate $Valid\_ext$ which selects terms extensionally equal to $Valid$ ones. For this, we have to extend eq_L to the sequence of types L_n. Note that for each integer n, eq_{L_n} is a Coq term.

Definition* $eq_{L_0} = eq_L$.
Definition* $eq_{L_{n+1}} = \lambda e, f : L_{n+1}.\forall v : var.(eq_{L_n}\ (e\ v)\ (f\ v))$.
Definition* $Valid\_ext = \lambda n : nat.\lambda e : L_n.\exists e' : L_n.((eq_{L_n}\ e\ e') \wedge (Valid\ n\ e'))$.

In order to integrate exotic terms of the second kind, we start by introducing a fourth higher-order constructor $\mathcal{V}$.

Definition* $\mathcal{V} = \lambda n : nat.\lambda v : var.\lambda x_1 : var.\ldots\lambda x_n : var.(Var\ v)$.

Note that although $\mathcal{V}$ is not a Coq term, for each n, $(\mathcal{V}\ n)$ is a Coq term in L_{n+1}, which we denote by $\mathcal{V}_n$. Similarly, $\mathcal{V}_{n,v}$ denotes $(\mathcal{V}\ n\ v)$. We are now in a position to mimic our characterization of well-formed closed terms through $Valid$ and $Valid\_ext$ to obtain the following predicates, $Valid\_v$ and $Valid\_v\_ext$, which characterize our open terms.

Inductive Definition* $Valid\_v : \forall n : nat.\ L_n \rightarrow Prop$
$= Valid\_v\_var : \forall n : nat.\forall v : var.(Valid\_v\ n\ (\mathcal{V}_n\ v))$
$\mid Valid\_v\_ref : \forall n, i : nat.(i < n) \rightarrow (Valid\_v\ n\ \mathcal{R}ef_{n,i})$
$\mid Valid\_v\_app : \forall n : nat.\forall e, e' : L_n.$
$(Valid\_v\ n\ e) \rightarrow (Valid\_v\ n\ e') \rightarrow (Valid\_v\ n\ (\mathcal{A}pp_n\ e\ e'))$
$\mid Valid\_v\_lam : \forall n : nat.\forall e :\ L_{n+1}.$
$(Valid\_v\ (n+1)\ e) \rightarrow (Valid\_v\ n\ (\mathcal{L}am_n\ e)).$

Definition* $Valid\_v\_ext =$
$\lambda n : nat.\lambda e : L_n.\exists e' : L_n.((eq_{L_n}\ e\ e') \wedge (Valid\_v\ n\ e')).$

In the next section, we will implement $(Valid\_v\_ext\ 0)$ and $(Valid\_ext\ 0)$.

6 Implementing Validity

As stated earlier, for any n, the challenge of defining a Coq predicate implementing $(Valid\, n)$ is to remove the dependence of the Lam case on $(n+1)$. We show here how we successfully overcome this difficulty for the case when $n = 1$ and define the Coq predicate $valid_1$ implementing $(Valid\ 1)$ modulo extensionality. Since we only need $(Valid\ 0)$, we can then implement it directly (modulo extensionality) using $valid_1$.

We denote by VL_n the subset of $Valid\_v\_ext$-terms in L_n and by VL the union of the VL_n's.

The basis of our implementation of $(Valid\ 1)$ is the following fact which shows that we are able to express quite simply $(Valid\ 2)$ in terms of $(Valid\ 1)$ (modulo extensionality).

Proposition* $Separate\_val : \forall e : var \rightarrow var \rightarrow L.$
$(\forall v : var.(Valid\_v\_ext\ 1\ (e\ v))) \rightarrow$
$(\forall v : var.(Valid\_v\_ext\ 1\ \lambda u : var.(e\ u\ v))) \rightarrow (Valid\_v\_ext\ 2\ e).$

Proof: It follows from the induction and injection principles that equivalence classes (modulo eq_{L_m}) of terms of VL_m are in one-one correspondence with (second-order abstract syntax) trees built from the higher-order constructors $\mathcal{R}ef_{m,i}$, $\mathcal{A}pp_m$, $\mathcal{L}am_m$, and $\mathcal{V}_{m,v}$. Here a tree is a set of (abstract) paths together with a map from this set to our set of constructors.

Let e be a term satisfying the assumptions. We pick three values u,v and w in var. (Note that by the var_nat axiom, we know there are infinitely many terms of type var. Here, we require at least three). By the first assumption, both $(e\ u\ v)$ and $(e\ u\ w)$ are values of the $(Valid\_v\_ext\ 1)$ function $(e\ u)$, thus their associated trees differ at most in some leaves, where they both have a $\mathcal{V}_m$, with different arguments. By the second assumption, a similar statement holds for the trees associated with $(e\ w\ v)$ and $(e\ u\ v)$. By transitivity, we infer that for any four-tuple (u, v, w, x) in var, the trees associated with $(e\ u\ v)$ and $(e\ w\ x)$ differ at most in some leaves, where they both have a $\mathcal{V}_m$ with different arguments. We denote by P the set of paths p where the constructor associated with $(e\ u\ v)$ is $\mathcal{V}$: the first argument of this $\mathcal{V}$ is an integer depending on p, which we denote m_p, while the second is of type var, depending on p, u, and v, and we denote it by $(\phi_p\ u\ v))$. We have to prove that for each p in P, ϕ_p is either one of the two projections or a constant function. We know that for any u, $\lambda v.(\phi_p\ u\ v)$ is either constant or the identity. Similarly, for any u, $\lambda v.(\phi_p\ v\ u)$ is either constant or the identity. The following lemma will complete our proof and illustrate why at least three distinct variables are needed.

Lemma*: Let var be a set with at least three elements. Let ϕ be a function from $var \times var$ into var satisfying the property that for any u, $\lambda v.(\phi\ u\ v)$ and $\lambda v.(\phi\ v\ u)$ are either constant functions or the identity. Then ϕ is either a constant function or one of the two projections.

Proof: First suppose that for some u, $\lambda x.(\phi\ u\ x)$ is a constant w different from u. Then for any v different from w, $\lambda x.(\phi\ x\ v)$ has to be constant and equal to w. Now choose u'. For x different from w, $(\phi\ u'\ x)$ is equal to w. Since there are at least two such x's, $\lambda x.(\phi\ u'\ x)$ has to be constant and equal to w.

The same argument applies when for some u, $\lambda x.(\phi\ x\ u)$ is a constant w different from u. In the remaining cases, $(\phi\ u\ v)$ can only be u or v.

Now suppose that for some u, $\lambda x.(\phi\ u\ x)$ is the constant function $\lambda x.u$. Using the previous assumption, we deduce that for any v different from u, $\lambda x.(\phi\ x\ v)$ is the identity. Now for any u', $\lambda x.(\phi\ u'\ x)$ takes the value u' at least twice, and hence is the constant function $\lambda x.u'$, and we are done.

In the remaining case, $\lambda x.(\phi\ u\ x)$ is the identity for any u, thus ϕ is the second projection.

The above proposition is crucial since it has the corollary below, which concerns the set of terms WL_1 also defined below, and allows a direct implementation of $(Valid\_v\_ext\ 1)$. We denote by VVL_1 the set of $(Valid\_v\ 1)$-terms. Note that VL_1 is the set of terms extensionally equal to terms in VVL_1.

Definition*: We define WL_1 as the smallest among the subsets W of L_1 satisfying the following conditions:

1. W contains $\lambda x.(Var\ x)$.
2. W contains $\lambda x.(Var\ u)$ for any u in var.
3. W contains $\lambda x.(App\ (a\ x)\ (b\ x))$ for any pair (a, b) of terms in W.
4. W contains $\lambda x.(Lam\ (e\ x))$ for any e of type $var \to var \to L$ satisfying the two conditions:
   (a) for any u in var, $(e\ u)$ is in W;
   (b) for any u in var, $\lambda x.(e\ x\ u)$ is in W.

Corollary*: If type var has at least three terms, then WL_1 contains VVL_1 and is contained in VL_1.

Proof: We first check that VL_1 is a subset of L_1 satisfying the above four conditions. It is clear for the first three. For the fourth one, if e is such that for any u in var, $(e\ u)$ and $\lambda x.(e\ x\ u)$ are in VL_1, then by the $Separate\_val$ proposition, e satisfies $(Valid\_v\_ext\ 2)$ and thus is eq_{L_2} equivalent to some $Valid\_v$-term e'. Thus $\lambda x.(Lam\ (e\ x))$ is eq_{L_1} equivalent to $\lambda x.(Lam\ (e'\ x))$ which satisfies $(Valid\_v\ 1)$ by $Valid\_v\_lam$. It follows that $\lambda x.(Lam\ (e\ x))$ satisfies $(Valid\_v\_ext\ 1)$. Since WL_1 is the smallest set satisfying the above conditions, WL_1 is contained in VL_1.

We now check that VVL_1 is contained in any such W. We choose such a W and we prove by induction that any term t satisfying $(Valid\_v\ 1)$ is in W. Induction is on the length of the proof of $(Valid\_v\ 1\ t)$. If t has a height of 1, then t is of the form $\lambda x.(Var\ x)$ or $\lambda x.(Var\ u)$, and hence it is in W. If t has bigger height, then the head constructor is either App or Lam. If $t = \lambda x.(App\ (a\ x)\ (b\ x))$, then by inversion of the definition of $Valid\_v$, a and b both satisfy $(Valid\_v\ 1)$. By the induction hypothesis, they are in W, and thus so is t. If $t = \lambda x.(Lam\ (e\ x))$, then by inversion of the definition of $Valid\_v$, e satisfies $(Valid\_v\ 2)$. This implies that for any u in var, $(e\ u)$ and $\lambda x.(e\ x\ u)$ are in VVL_1, and thus in W since the height of these terms is smaller than the height of t.

We conjecture that WL_1 is in fact equal to VVL_1 but have not yet proved it.

Now we state our Coq definitions. The definition of $valid_1$ is derived directly from the definition of WL_1 above and selects exactly the terms of type $var \to L$ that we want.

Inductive Definition $valid_1 : (var \to L) \to$ Prop
$= valid_1\_var : \forall v : var.(valid_1\ \lambda x : var.(Var\ v))$
$\mid valid_1\_ref : (valid_1\ \lambda x : var.(Var\ x))$
$\mid valid_1\_app : \forall e, e' : var \to L.$
$(valid_1\ e) \to (valid_1\ e') \to (valid_1\ \lambda x : var.(App\ (e\ x)\ (e'\ x)))$
$\mid valid_1\_lam : \forall e : var \to var \to L.$
$(\forall u : var.(valid_1\ \lambda v : var.(e\ u\ v)) \wedge (valid_1\ \lambda v : var.(e\ v\ u))) \to$
$(valid_1\ \lambda x : var.(Lam\ (e\ x)))$.

Inductive Definition $valid_0 : L \to$ Prop
$= valid_0\_var : \forall v : var.(valid_0\ (Var\ v))$
$\mid valid_0\_app : \forall a, b : L.(valid_0\ a) \to\ (valid_0\ b) \to (valid_0\ (App\ a\ b))$
$\mid valid_0\_lam : \forall e : var \to L.(valid_1\ e) \to (valid_0\ (Lam\ e))$.

Definition $valid = \lambda e : L.\exists e' : L.((eq_L\ e\ e') \wedge (valid_0\ e'))$.

Definition $closed = \lambda t : L.((valid_0\ t) \wedge$
$\forall e : var \to L.(valid_1\ e) \to \forall x : var.(t = (e\ x)) \to \forall y : var.(eq_L\ (e\ y)\ t))$.

It follows easily from the above statements that the *valid* predicate implements $(Valid\_v\_ext\ 0)$ and that *closed* implements $(Valid\_ext\ 0)$.

Note that $(valid_0\ exot_1)$ does not hold, but $(valid\ exot_1)$ does if we take e' in the definition of *valid* to be $(Lam\ \lambda x : nat.(Var\ x))$. Note also that in order for $(Lam\ \lambda f : nat.(Lam\ \lambda x : nat.(exot_3\ f\ x)))$ to satisfy $valid_0$, $\lambda f : nat.(Lam\ \lambda x : nat.(exot_3\ f\ x))$ must satisfy $valid_1$. Although it is the case that $\forall u : var.(valid_1\ \lambda v : var.(exot_3\ v\ u))$ holds, $\forall u : var.(valid_1\ \lambda v : var.(exot_3\ u\ v))$ does not. In fact, no term with a *Case* operator will satisfy $valid_0$. However, we must include those that are extensionally equal to terms with no *Case* operator in order to correctly implement substitution, which we define in the next section.

7 Substitution

In order to specify evaluation for our language, we must specify β-reduction which for our representation is the operation that, given a redex of the form $(App\ (Lam\ \lambda x : var.M)\ N)$, replaces all occurrences of $(Var\ x)$ in M by N. To do so, we define a Coq predicate *subst*. We proceed as we did to define *valid*, here starting with a definition of *Subst* of type $\forall n : nat.L_{n+1} \to L_0 \to L_n \to Prop$. The proposition $(Subst\ n\ E\ p\ r)$ holds if E has the form $\lambda x : var.F$ and r is the term obtained by replacing all occurrences of $(Var\ x)$ in F by p. Although we define it relationally, *Subst* is functional on the first three arguments.

$$
\begin{array}{l}
\textsf{Inductive Definition}^{*}\ Subst : \forall n : nat.L_{n+1} \to L_0 \to L_n \to Prop \\
\quad = Subst\_ref\_rename : \forall n : nat.\forall p : L_0(Subst\ n\ \mathcal{R}ef_{n+1,n}\ p\ \lambda x_1.\ldots.\lambda x_n.p) \\
\quad \mid Subst\_ref\_keep : \forall n : nat.\forall p : L_0.\forall i \in [0..n-1]. \\
\qquad (Subst\ n\ \mathcal{R}ef_{n+1,i}\ p\ \mathcal{R}ef_{n,i}) \\
\quad \mid Subst\_var : \forall n : nat.\forall v : var.\forall p : L_0.(Subst\ n\ (\mathcal{V}_{n+1}\ v)\ p\ (\mathcal{V}_n\ v)) \\
\quad \mid Subst\_app : \forall n : nat.\forall p : L_0.\forall A, A' : L_{n+1}\forall B, B' : L_n. \\
\qquad (Subst\ n\ A\ p\ B) \to (Subst\ n\ A'\ p\ B') \to \\
\qquad (Subst\ n\ (\mathcal{A}pp_{n+1}\ A\ A')\ p\ (\mathcal{A}pp_n\ B\ B')) \\
\quad \mid Subst\_lam : \forall n : nat.\forall p : L_0.\forall A : L_{n+2}.\forall B : L_{n+1}. \\
\qquad (Subst\ (n+1)\ A\ p\ B) \to (Subst\ n\ (\mathcal{L}am_{n+1}\ A)\ p\ (\mathcal{L}am_n\ B)).
\end{array}
$$

As before this definition cannot be directly transformed into a Coq definition because it requires an infinite series of definitions where for each n $(Subst\ n)$ requires that of $(Subst\ (n+1))$. As before, we must implement $(Subst\ 0)$. As for $(Valid\ 0)$, we cannot do it directly, but instead must work modulo extensionality. In particular, we implement:

$$
\begin{array}{l}
\textsf{Definition}^{*}\ Subst\_ext = \lambda n : nat.\lambda e : L_{n+1}.\lambda p : L_0.\lambda r : L_n.\exists e' : L_{n+1}.\exists p' : L_0. \\
\quad \exists r' : L_n.(eq_{L_{n+1}}\ e\ e') \to (eq_{L_0}\ p\ p') \to (eq_{L_n}\ r\ r') \to (Subst\ n\ e'\ p'\ r').
\end{array}
$$

Note that if $(Subst\ n\ E\ p\ r)$ holds, then E, p and r are $Valid\_v$-terms, and if $(Subst\_ext\ n\ E\ p\ r)$ holds, then E, p and r are $Valid\_v\_ext$-terms. The property below is crucial and reduces $(Subst\_ext\ (n+1))$ to $(Subst\_ext\ n)$. Here $t@x$ denotes the term $\lambda x_1,\ldots,x_n : var.(t\ x_1 \ldots x_n\ x)$, where the value of n can be determined from context.

Lemma* : $\forall n : nat.\forall E : L_{n+2}.\forall p : L_0.\forall r : L_{n+1}.$
$(Valid\_v\_ext\ (n+2)\ E) \rightarrow (Valid\_v\_ext\ 0\ p) \rightarrow (Valid\_v\_ext\ (n+1)\ r) \rightarrow$
$(\forall v : var(Subst\_ext\ n\ E@v\ p\ r@v)) \rightarrow (Subst\_ext\ (n+1)\ E\ p\ r)$

Proof: Let r' be a term such that $(Subst\_ext\ (n+1)\ E\ p\ r')$. Using the fact that replacing all the arguments (except the variable being substituted) by a value before or after the substitution leads to the same result, we have that $\forall v : var.(Subst\_ext\ n\ E@v\ p\ r'@v)$ holds. From the fact that $Subst$ is functional, we know that for all v of type var, $r@v$ and $r'@v$ are eq_{L_n} equivalent. This directly implies that r and r' are $eq_{L_{n+1}}$ equivalent.

Note that this lemma would not hold without extensionality.

This property leads to the following implementation of $(Subst\_ext\ 0)$.

Inductive Definition $subst : (var \rightarrow L) \rightarrow L \rightarrow L \rightarrow Prop$
$= subst\_ref : \forall p : L.(subst\ \lambda x : var.(Var\ x)\ p\ p)$
$\mid subst\_var : \forall v : var.\forall p : L.(subst\ \lambda x : var.(Var\ v)\ p\ (Var\ v))$
$\mid subst\_app : \forall p : L.\forall A, A' : (var \rightarrow L).\forall B, B' : L.$
$(subst\ A\ p\ B) \rightarrow (subst\ A'\ p\ B') \rightarrow$
$(subst\ \lambda v : var.(App\ (A\ v)\ (A'\ v))\ p\ (App\ B\ B'))$
$\mid subst\_lam : \forall p : L.\forall A : var \rightarrow var \rightarrow L.\forall B : var \rightarrow L.$
$(\forall v : var.(subst\ (\lambda x : var.(A\ x\ v))\ p\ (B\ v))) \rightarrow$
$(subst\ (\lambda x : var.(Lam\ (A\ x)))\ p\ (Lam\ B)).$

Definition $subst\_ext = \lambda e : var \rightarrow L.\lambda p : L.\lambda r : L.\exists e' : var \rightarrow L.\exists p' : L.\exists r' : L.$
$(\forall v : var.(eq_L\ (e\ v)\ (e'\ v))) \rightarrow (eq_L\ p\ p') \rightarrow (eq_L\ r\ r') \rightarrow (subst\ e'\ p'\ r').$

The correctness of these definitions is expressed by the following statement whose proof is straightforward. (The second part follows directly from the lemma above).

Proposition* : $\forall E : var \rightarrow L.\forall p, r : L.(Subst\_ext\ 0\ E\ p\ r) \rightarrow (subst\_ext\ E\ p\ r)$. Conversely, $\forall E : var \rightarrow L.\forall p, r : L.(valid_1\ E) \rightarrow (valid\ p) \rightarrow (valid\ r) \rightarrow (subst\_ext\ E\ p\ r) \rightarrow (Subst\_ext\ 0\ E\ p\ r)$.

In proving properties of our object-language, we may choose to use either $subst$ or $subst\_ext$. In the next section, we choose the former.

8 An Example Proof: the Subject Reduction Theorem

In this section, we specify type assignment and evaluation for our object-language by introducing inductive types for each. We then outline the proof of type soundness (also called subject reduction) which we have fully formalized in Coq.

For typing, we first introduce a predicate for assigning variables to types along with two assumptions about it stating that each variable has a unique type and that there is a variable at every type

Parameter $typvar : var \to tL \to Prop.$
Axiom $uniq\_var\_type : \forall x : var.\forall t, s : tL.(typvar\ x\ t) \to (typvar\ x\ s) \to (s = t).$
Axiom $exists\_new\_var : \forall t : tL.\exists x : var.(typvar\ x\ t).$

Like var, $typvar$ is introduced as a parameter, and thus the theorems we prove will hold for any instantiation. Here, the important ones to consider will be those for which we can prove the above axioms. Note, for example, that these two axioms hold trivially if we take var to be tL and $typvar$ to be equality on tL.

The usual natural deduction style inference rules for assigning simple types to untyped terms is specified by the following inductive type.

Inductive Definition $type : L \to tL \to Prop$
$= type\_Var : \forall x : var.\forall s : tL.(typvar\ x\ s) \to (type\ (Var\ x)\ s)$
$\mid type\_App : \forall e, e' : L.\forall t', t : tL.$
$(type\ e\ (Arrow\ t'\ t)) \to (type\ e'\ t') \to (type\ (App\ e\ e')\ t)$
$\mid type\_Lam : \forall E : var \to L.\forall t, t' : tL.$
$(\forall x : var.(typvar\ x\ t) \to (type\ (E\ x)\ t')) \to (type\ (Lam\ E)\ (Arrow\ t\ t')).$

The third clause in this definition uses a hypothetical judgment with an embedded arrow for typing λ-abstractions. It asserts the fact that $(Lam\ E)$ has functional type $(Arrow\ t\ t')$ if under the assumption that for arbitrary variable x of type t, it can be shown that the expression $(E\ x)$ (the expression obtained by replacing all occurrences of the variable bound by the λ-abstraction at the head of E with x) has type t'.

Similar definitions of *type* have been given in LF and λProlog where the predicate defining typing appears on the left of the embedded arrow. Here this would mean replacing $(typvar\ x\ t)$ by $(type\ (Var\ x)\ t)$. Note that this change results in a negative occurrence of *type* which is disallowed in Coq. For this reason, we need a separate *typvar* predicate, just as we needed a separate type var in the definition of L.

The following induction principle for this definition illustrates the general form of induction over inductively defined predicates, in particular when they involve embedded universal quantification and implication.

$$
\begin{array}{l}
\forall P.L \to tL \to Prop. \\
\quad (\forall x : var.\forall s : tL.(typvar\ x\ s) \to (P\ (Var\ x)\ s)) \to \\
\quad (\forall E : var \to L.\forall t, t' : tL. \\
\qquad (\forall x : var.(typvar\ x\ t) \to (type\ (E\ x)\ t')) \to \\
\qquad (\forall x : var.(typvar\ x\ t) \to (P\ (E\ x)\ t')) \to (P\ (Lam\ E)\ (Arrow\ t\ t'))) \to \\
\quad (\forall e, e' : L.\forall t', t : tL. \\
\qquad (type\ e\ (Arrow\ t'\ t)) \to (P\ e\ (Arrow\ t'\ t)) \to \\
\qquad (type\ e'\ t') \to (P\ e'\ t') \to (P\ (App\ e\ e')\ t)) \to \\
\quad \forall e : L.\forall t : tL.(type\ e\ t) \to (P\ e\ t)
\end{array}
$$

Call by value semantics for our simple functional language is defined by the following inductive definition. Note the use of substitution in the β-redex case.

$$
\begin{array}{l}
\textsf{Inductive Definition}\ eval : L \to L \to Prop \\
\quad = eval\_Lam : \forall E : var \to L.(eval\ (Lam\ E)\ (Lam\ E)) \\
\quad \mid eval\_App : \forall E : var \to L.\forall e_1, e_2, e_3, v_1, v_2 : L.(eval\ e_1\ (Lam\ E)) \to \\
\qquad (eval\ e_2\ v_2) \to (subst\ E\ v_2\ e_3) \to (eval\ e_3\ v_1) \to (eval\ (App\ e_1\ e_2)\ v_1).
\end{array}
$$

The proof of type soundness is quite simple and follows naturally from the definitions and axioms presented in this section, the definitions in Section 2, and the definition of *subst*. The main lemma needed for this theorem is that the predicate *subst* preserves typing. For this lemma, we need to define the notion of two terms having the same type. This definition, lemma, and the main theorem are stated as follows.

$$
\begin{array}{l}
\textsf{Definition}\ same\_type = \lambda m, n : L.\exists t : tL.(type\ m\ t) \wedge (type\ n\ t). \\
\textsf{Lemma}\ subst\_sr : \forall E : var \to L.\forall p, n : L.(subst\ E\ p\ n) \to \\
\quad \forall x : var.(same\_type\ (Var\ x)\ p) \to \forall t : tL.(type\ (E\ x)\ t) \to (type\ n\ t). \\
\textsf{Theorem}\ subj\_reduction : \forall e, v : L.(eval\ e\ v) \to \forall t : tL.(type\ e\ t) \to (type\ v\ t).
\end{array}
$$

The proof of the lemma proceeds by induction on $(subst\ E\ p\ n)$, while the proof of subject reduction proceeds by induction on $(eval\ e\ v)$.

9 Conclusion and Future Work

We have shown by example how higher-order syntax can be used in formal proof. Our method of specification of syntax is in fact quite general. Although we have not yet done it, we plan to generalize it formally as is done in [3]. For any object-language that can be expressed in second-order syntax, it is easy to see how to define the corresponding *valid* and *subst* predicates. Proofs of adequacy follow similarly. In doing proofs, the user is then freed from concerns of α-conversion, and substitution is greatly simplified. In fact, it is possible to automate generation of these definitions and to automate certain aspects of proof search that occur repeatedly in such proofs. Although our proof is already simple (500 lines of Coq script), it would be further simplified by such automated tools.

In the example presented here, we were able to state type soundness without any mention of validity. For other statements, however, this is not possible, and care must be taken to include assumptions about validity where necessary (typicaly on existencial variables). Note that a systematic insertion of *valid* would solve this problem and could be automated. In our case, this would lead to

$\forall e, v : L.(valid\ e) \rightarrow (valid\ v) \rightarrow (eval\ e\ v) \rightarrow \forall t : tL.(type\ e\ t) \rightarrow (type\ v\ t)$,

which is weaker than what we actually proved, but expresses the same object-level property.

In addition to type soundness as presented here, several other examples are in progress including a proof of correctness of a λProlog program that computes the negation normal form of formulas in first-order logic and a proof of the Church-Rosser property for the λ-calculus.

Finally, we have not considered here the question of adequacy for our semantic definitions. This, together with the correctness of the Coq theorems with respect to the corresponding 'meta' theorems will be the subject of future work. The latter will follow naturally (see. Theorem 1. in Section 3.5 in [3]).

References

[1] A. Asperti and G. Longo. *Categories, Types, and Structures.* MIT Press, Foundations of Computing Series, London, England, 1991.

[2] A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, Dec. 1992.

[3] J. Despeyroux and A. Hirschowitz. Higher-order syntax and induction in coq. In *Proceedings of the fifth Int. Conf. on Logic Programming and Automated Reasoning (LPAR 94), Kiev, Ukraine, July 16–21, 1994*, 1994. Also available as an INRIA Research Report RR-2292, Inria-Sophia-Antipolis, France, June 1994.

[4] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide. Technical Report 154, INRIA, 1993.

[5] A. Felty. A logic programming approach to implementing higher-order term rewriting. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the January 1991 Workshop on Extensions to Logic Programming*, pages 135–161. Springer-Verlag LNCS, 1992.

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

[7] J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, Technical Report MS-CIS-91-09, Jan. 1991.

[8] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2:415–459, 1992.

[9] J. Hannan and F. Pfenning. Compiler verification in LF. In *Seventh Annual Symposium on Logic in Computer Science*, pages 407–418, 1992.

[10] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993.

[11] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the January 1991 Workshop on Extensions to Logic Programming*, pages 299–344. Springer-Verlag LNCS, 1992.

[12] D. Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*. MIT Press, 1991.

[13] C. Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 664, pages 328–345. Springer Verlag Lecture Notes in Computer Science, 1993.

[14] F. Pfenning and E. Rohwedder. Implementing the meta-theory of deductive systems. In *Eleventh International Conference on Automated Deduction*, pages 537–551. Springer-Verlag LNCS, 1992.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399