INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Higher-order abstract syntax with induction in Coq

Joëlle Despeyroux , André Hirschowitz

N° 2292

June 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

Rapport de recherche

1994

ISSN 0249-6399

Higher-order abstract syntax with induction in Coq

Joëlle Despeyroux *, André Hirschowitz **

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

Rapport de recherche n° 2292 — June 1994 — 18 pages

Abstract: Three important properties of Higher-Order Abstract Syntax are the (higher-order) induction principle, which allows proofs by induction, the (higher-order) injection principle, which asserts that equal terms have equal heads and equal sons, and the extensionality principle, which asserts that functional terms which are pointwise equal are equal. Higher-order abstract syntax is implemented for instance in the Edinburgh Logical Framework and the above principles are satisfied by this implementation. But although they can be proved at the meta level, they cannot be proved at the object level and furthermore, it is not so easy to know how to formulate them in a simple way at the object level. We explain here how Second-Order Abstract Syntax can be implemented in a more powerful type system (Coq) in such a way as to make available or provable (at the object level) the corresponding induction, injection and extensionality principles.

Key-words: Higher-Order Abstract Syntax, Induction, Type Systems, Theorem provers, Calculus of Inductive Constructions.

(Résumé : tsvp)

To appear in the Proc. of the Int. Conf. on Logic Programming and Automated Reasoning, LPAR'94, Kiev, Ukraine, July 1994

*joelle.despeyroux@sophia.inria.fr. INRIA-Sophia
**andre.hirschowitz@sophia.inria.fr. CNRS URA 168, University of Nice, F-06108 Nice Cedex 2, France

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Syntaxe abstraite d'ordre supérieur avec induction dans Coq

Résumé : En syntaxe abstraite d'ordre supérieur, on dispose de trois principes importants. Le premier est le principe d'induction (d'ordre supérieur), qui permet de faire des preuves par induction. Le second est le principe d'injection, qui dit que deux termes égaux ont des opérateurs de tête égaux et des fils égaux. Le troisième est le principe d'extensionalité, qui dit que deux termes fonctionnels égaux en tous points sont égaux. La notion de syntaxe abstraite d'ordre supérieur est implementée dans le 'Logical Framework' d'Edinbourg (LF), òu ces trois principes sont valides. Mais bien qu'ils soient prouvables au niveau meta, ces principes ne peuvent etre prouvés (ni même exprimés de manière simple) au niveau objet. Nous donnons ici une méthode d'implémentation de la syntaxe abstraite d'ordre deux, dans un système de types plus fort que LF, le système Coq, de manière à rendre ces principes disponibles au niveau objet.

Mots-clé : Syntaxe abstraite d'ordre supérieur , Induction, Systèmes de types, Systèmes de développement de preuves, Calcul des constructions inductives.

1 Introduction

The original motivation of our work is to investigate how to use a powerful theorem prover to perform proofs in Natural Semantics [Kah87] written in the LF [AHM87][HHP87][HHP91] style. We have chosen the system Coq [DFH⁺93] rather than other systems, like Elf [Pfe89] or Isabelle [Pau89] for example, because Coq is equipped with a notion of inductive definitions [PM92] which provides induction and a recursion operator (called Match) both on expressions and on proofs.

The problem of the implementation of second-order syntax together with induction in Coq is not trivial because declarations like the following one are not legal:

'Inductive Set' $L_0 = Lam_0 : (L_0 \rightarrow L_0) \rightarrow L_0 \mid App_0 : L_0 \rightarrow L_0 \rightarrow L_0$.

Indeed, in the definition of the constructor $Lam_0 : (L_0 \rightarrow L_0) \rightarrow L_0$, the first occurrence of L_0 is *negative* and this is not allowed in an inductive definition.

Let us explain, on the above example, the main features of our solution.

In order to avoid negative positions, we use the following trick. We use a set *var* for variables, introduced by the following declaration:

Variable var : Set.

together with an axiom providing *var* with two distinct values. This assumption is much weaker than the usual one concerning variables. This reflects the fact that our treatment of variables is genuinely higher-order. Our lam-constructor has for its argument a function from *var* to L instead of from L to L. In order to complete the picture, we also need a constructor from *var* to L. This yields the following *inductive* declaration:

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

For instance, the term $\lambda x.(x\ x)$ is encoded as

$(Lam\ (\lambda x : var.\ (App\ (Var\ x)\ (Var\ x))))$,

using the meta-level operator λ. Terms of L form our *provisional* syntax. Now we want to implement altogether closed and open terms. We see open terms as higher-order terms, in other words as functions of an arbitrary number of variables. We know of at least three approaches for implementing such functions:

- through implicit lists, by glueing in a single dependent type the sequence L_n of types: L, $L \rightarrow L$, $L \rightarrow L \rightarrow L$, ... Such a definition is not possible in the current Coq system, since the rule $Nodep_{Set,Type_{Set}}$ is not yet available;
- through a term *list* of type $nat \rightarrow Set \rightarrow Set$, where the intended meaning of $(list\ n\ A)$ is the type of lists of length n of elements of A;
- through infinite lists via the term $list = \lambda A : Set.nat \rightarrow A$.

Here below we explore the latter solution, which appeared to be the simplest one. Our infinite lists are easily equipped with the usual terms *cons*, *car* and *cdr*. Thus our final syntax is made of terms of type $(list\ L) \rightarrow L$. Not all such terms are convenient and we have to rule out exotic terms through a predicate *Valid* of type $nat \rightarrow ((list\ L) \rightarrow L) \rightarrow Prop$. Finally, we have to identify *Valid* terms which are *extensionally* equal in some natural sense; this is because Coq's object equality (more precisely the polymorphic equality on Sets provided in Coq) is not extensional.

The desired induction principle is generated by the inductive definition of *Valid*. The extensionality principle is given for free by our ad-hoc equality. As for the injection principle, we are able to prove it at the object level.

Our implementation of second-order syntax would be meaningless from our point of view if it did not allow convincing formulations and proofs in semantics. We wish to address this problem systematically in the future. In this paper, we present briefly a significant example, namely an implementation of a translation from a first-order (de Bruijn) version of our simply typed λ-calculus to our second-order description of the same calculus, together with the formulation and proof (at the object level) of the correctness of this translation. This proof illustrates nicely the use of the induction and injection principles on both sides.

The rest of the paper is organized as follows. In section 2, we explain our implementation of the above example of a simply-typed λ-calculus. In subsection 2.1, we review a first-order implementation of our λ-calculus in Coq. In subsection 2.2, we introduce our provisional syntax and discuss its properties. In subsection 2.3, we present our final syntax. In order to build a complete proof of the injection principle, we were led to introduce ad-hoc extensional notions of equality, which are studied here. In subsection 2.4, we present our object-level proof of translation. The statements proved in this subsection 2.4 are not the natural ones, since they involve our notion of equality, instead of Coq's object equality. This makes them more cumbersome. In subsection 2.5, we discuss alternative approaches through extensionality axioms needed for proofs of the original statements (those involving Coq's object equality). In section 3, we explain how to implement similarly a large class of second-order syntaxes. Related works are discussed in section 4, while future works are presented in the conclusion.

Note: Throughout the paper, Coq terms are pretty-printed using λ, $\forall$ and $\exists$ in place of [_], (_) and exists([_]_). We also omit the type information in the Match $((< T > Match\ \_))$ and in Coq's object equality on Sets $(< T > \_ = \_)$.

2 An example

We explain in this section our implementation of the example of the simply-typed λ-calculus considered in the above introduction.

2.1 First-order setting

First, let us review first-order syntax as it can be implemented in a theorem prover such as Coq. For a first-order implementation of a λ-calculus without variables, the de Bruijn approach is simple and efficient: in Coq, we may declare, following Huet [Hue92]:

$\mathsf{Inductive\ \ Set}\ fol =\ ref : nat \to fol \mid lam : fol \to fol \mid app : fol \to fol \to fol.$

Induction and Match. This declaration generates, for terms of type fol, both a Match operator and the following induction principle, available for object proofs:

$\forall P : fol \to Prop.$
$(\forall n : nat.\ (P\ (ref\ n)))$
$\to\ (\forall e : fol.\ (P\ e) \to (P\ (lam\ e)))$
$\to\ (\forall a : fol.\ (P\ a) \to \forall b : fol.\ (P\ b) \to (P\ (app\ a\ b)))$
$\to\ \forall e : fol.\ (P\ e).$

Injection principle. These two tools make it possible to prove at the object level the following injection principle, made of six theorems (we only give two of them here):

$\mathsf{Theorem}\ lam\_app : \forall e : fol.\forall a, b : fol.\ \neg((lam\ e) = (app\ a\ b)).$
$\mathsf{Theorem}\ lam\_lam : \forall a, b : fol.\ ((lam\ a) = (lam\ b)) \to a = b.$

Extensionality. In this setting already, the extensionality principle does not hold. We can easily produce two terms of type $fol \to fol$ which are extensionally equal but not equal. For instance, the identity function on fol can naturally be written as $\lambda x : fol.x$ or using the Match operator, which is a combination of a recursor operator and a case operator ($(*\_*)$ denotes comments in Coq):

$\lambda x : fol.\ (Match\ x\ with$
$\quad (*\ (ref\ n)\ *)\ \lambda n : nat.(ref\ n)$
$\quad (*\ (lam\ y)\ *)\ \lambda y : fol.\lambda h_y : fol.(lam\ y)$
$\quad (*\ (app\ y\ z)\ *)\ \lambda y : fol.\lambda h_y : fol.\lambda z : fol.\lambda h_z : fol.(app\ y\ z)).$

This is not such a drawback since, in this setting, we do not plan to manipulate higher-order terms.

Inversion. This is an important tool in Coq. As induction is not allowed on partially instanciated terms, a standard way to simulate a double induction on two predicates is to use induction on the first predicate, followed by the use of the inversion rules of the second predicate. We give here as an example, the inversion rules for the following predicate *valid*, which makes it possible to characterize closed terms (through the predicate $(valid\ 0)$).

$\mathsf{Inductive\ Definition}\ valid : nat \to fol \to Prop$
$= valid\_ref : \forall n, m : nat.\ (m < n) \to (valid\ n\ (ref\ m))$
$\mid valid\_lam : \forall n : nat.\forall e : fol.\ (valid\ (S\ n)\ e) \to (valid\ n\ (lam\ e))$
$\mid valid\_app : \forall n : nat.\forall a, b : fol.\ (valid\ n\ a) \to (valid\ n\ b) \to (valid\ n\ (app\ a\ b)).$

Since the conclusions of the different rules do not unify, the inversion package simply reads as the following one:

$valid\_inv\_ref : \forall n, m : nat.(valid\ n\ (ref\ m)) \to\ (\ m <\ n)$
$valid\_inv\_lam : \forall n : nat.\forall e : fol.(valid\ n\ (lam\ e)) \to (valid\ (S\ n)\ e)$
$valid\_inv\_app : \forall n : nat.\forall a, b : fol.(valid\ n\ (app\ a\ b)) \to (valid\ n\ a) \wedge (valid\ n\ b).$

The proof of this package, which is more or less standard (cf. eg. [DFH+93]), uses the injection principle (here on fol) in an essential way.

2.2 Provisional setting

In this section, we explain our *provisional* syntax. At first, we introduce a type var for variables.

Variable var : Set.

The keyword Variable makes var universally quantified for the rest of the session. Next we make sure that var is inhabited by at least two distinct values:

Axiom $var_2 : \exists x, y : var.\ (x \neq y).$

Now our provisional syntax L is as follows:

Inductive Set $L = Var : var \to L \mid Lam : (var \to L) \to L \mid App : L \to L \to L.$

This definition generates some exotic terms: indeed, we want the type var to be used only for (bound) meta-variables; those terms using (ground) values of type var have to be ruled out. Also, for var an inductive type, we could have irreducible terms of type $var \to L$ different from Var, and, through Lam, these would again generate exotic terms. And it seems quite hard to formulate an assumption on var which makes it possible to prove that the terms of type $var \to L$ are the expected ones only. Here below (higher-order setting), we shall show how to rule out exotic terms.

Induction and Match. The previous declaration generates, for terms of type L, both a Match operator and the following induction principle, available for object proofs:

$\forall P : L \to Prop.$
$(\forall x : var.\ (P\ (Var\ x)))$
$\to\ (\forall e : (var \to L).\ (\forall x : var.\ (P\ (e\ x))) \to (P\ (Lam\ e)))$
$\to\ (\forall a : L.\ (P\ a) \to \forall b : L.\ (P\ b) \to (P\ (App\ a\ b)))$
$\to\ \forall e : L.\ (P\ e).$

First injection principle. The two previous tools make it possible to prove the injection principle for the type L, which is a package of six theorems:

Theorem $Lam\_app : \forall e : (var \to L).\forall a, b : L.\ \neg((Lam\ e) = (App\ a\ b)).$
Theorem $Lam\_lam : \forall a, b : (var \to L).\ ((Lam\ a) = (Lam\ b)) \to (a = b).\ \ldots$

Extensionality. As in the first-order case, the Match operator will generate for instance a term of type $L \to L$ which is extensionally equal but not equal to the identity on L. Even after having ruled out exotic terms and gone to the higher-order setting, it seems quite hard, and maybe impossible, to prove the desired extensionality principle. That is why we introduce the ad-hoc notions of equality, which give the desired extensionality principle for free.

Inductive Definition $eq_L : L \to L \to Prop$
$= eq_L\_var : \forall x : var.(eq_L\ (Var\ x)\ (Var\ x))$
$\mid eq_L\_lam : \forall a, b : (var \to L).(\forall x : var.(eq_L\ (a\ x)\ (a'\ x)))$
$\to (eq_L\ (Lam\ a)\ (Lam\ a'))$
$\mid eq_L\_app : \forall a, a', b, b' : L.\ (eq_L\ a\ a') \to (eq_L\ b\ b') \to (eq_L\ (App\ a\ b)\ (App\ a'\ b'))$.

As in the first-order case, where we have given the inversion rules for the *valid* predicate, the previous first injection principle for L makes it possible to prove the rules for inversion of eq_L.

Second injection principle. We now have to suit our injection principle to this new equality. We get a package of six theorems, the proofs of which mainly use the inversion of eq_L.

Theorem $Lam\_app\_eq : \forall e : (var \to L).\forall a, b : L.\ \neg(eq_L\ (Lam\ e)\ (App\ a\ b))$.
Theorem $Lam\_lam\_eq : \forall e, e' : (var \to L).(eq_L\ (Lam\ e)\ (Lam\ e'))$
$\to \forall x : var.(eq_L\ (e\ x)\ (e'\ x))$. ...

2.3 Higher-order setting

As mentioned in the introduction, it is not yet possible to implement the list of types L_n as a dependent type in Coq. However, by curryfication, we may identify L_n with a type $(list\ n\ L) \to L$. This is how lists enter the picture. Now it turns out to be very uncomfortable to work with such a dependant type, because the type $(list\ n\ L)$ is not equal to the type $(list\ m\ L)$, even when n and m are dynamically equal. This is the reason why we chose infinite lists.

Lists. We simply define lists through: Definition $list = \lambda A : Set.nat \to A$.

Lists are defined together with the usual terms *cons* , *car*, *cdr* and *map*. An additional basic definition is provided: $proj_p$, that gives the p-th element of a list. A *cst* function builds a list from a given element. A bunch of theorems has been proved on these lists. A typical one is the following:

Theorem $cons\_car\_cdr_{ext} : \forall A : Set.\forall l : (list\ A)$.
$\forall n : nat.\ (l\ n) = (cons\ A\ (car\ A\ l)\ (cdr\ A\ l\ )\ n)$.

which states that a list is essentially the *cons* of its *car* and its *cdr*. We do not state it as follows:

'Theorem' $cons\_car\_cdr : \forall A : Set.\forall l : (list\ A).\ l = (cons\ A\ (car\ A\ l)\ (cdr\ A\ l\ ))$.

because the proof of the latter statement needs some extensionality axiom, like the following one:

Axiom $ext_l : \forall A : Set.\forall f, g : (list\ A).\ (\forall x : nat.(f\ x) = (g\ x)) \to f = g$.

Actually, this problem of extensionality is recurrent all over our work.

Higher-order constructors. We shall implement our higher-order syntax within the type $(list\ L) \rightarrow L$, which will be denoted as LL. Exotic terms will be ruled out easily, and we shall be able to define a suitable equality. We introduce what we call the higher-order constructors, which will make apparent the tree structure of our higher-order terms.

Definition $\mathcal{R}ef = \lambda i : nat.\lambda x : (list\ L).\ (x\ i)$.
Definition $\mathcal{L}am = \lambda e : LL.\lambda x : (list\ L).\ (Lam\ (\lambda y : var.\ (e\ (cons\ L\ (Var\ y)\ x))))$.
Definition $\mathcal{A}pp = \lambda a,\ b : LL.\lambda x : (list\ L).\ (App\ (a\ x)\ (b\ x))$.

These higher-order constructors are the exact counterpart of the constructors of fol. Indeed, we may define naturally the translation relating fol and LL:

Inductive Definition $trans : fol \rightarrow LL \rightarrow Prop$
$= trans\_ref : \forall n : nat.(trans\ (ref\ n)\ (\mathcal{R}ef\ n))$
$\mid trans\_lam :\ \forall e : fol.\forall e' : LL.(trans\ e\ e') \rightarrow (trans\ (lam\ e)\ (\mathcal{L}am\ e'))$
$\mid trans\_app :\ \forall a : fol.\forall a' : LL.(trans\ a\ a') \rightarrow$
$\forall b : fol.\ \forall b' : LL.(trans\ b\ b') \rightarrow (trans\ (app\ a\ b)\ (\mathcal{A}pp\ a'\ b'))$.

In fact, as for the injection principle given at the end of the previous section, we have to use a modified version of the translation given above, involving our ad hoc equality eq_{LL} given below, instead of Coq's object equality. For example, the first rule of $trans$ has to be read as:

$trans\_ref : \forall n : nat.\forall e : LL.(eq_{LL}\ e\ (\mathcal{R}ef\ n)) \rightarrow (trans\ (ref\ n)\ e)$.

In the rest of this section, for each definition we shall give (more exactly for wf and $Valid$), we shall only give the simplest version, for clarity.

Coarse equality. In order to prove the desired extensionality principle, we shall define two notions of equality which we shall prove to coincide on well-formed terms. We start with the coarser one: two terms are made equal by this definition if they associate equal (in the sense of eq_L) values to any list of variables:

Definition $eq_{LLv} = \lambda a, b : LL.\forall x : (list\ var)$.
$(eq_L\ (a\ (map\ var\ L\ Var\ x))(b\ (map\ var\ L\ Var\ x)))$.

Extensional equality. We now turn to our final notion of equality. Two terms are equal by this definition if they associate equal (in the sense of eq_L) values to any list of terms. This notion is finer than the previous one.

Definition $eq_{LL} = \lambda a, b : LL.\forall x : (list\ L).(eq_L\ (a\ x)\ (b\ x))$.
Theorem $eq_{LL}\_eq_{LLv} : \forall a, b : LL.\ (eq_{LL}\ a\ b) \rightarrow (eq_{LLv}\ a\ b)$.

Ruling out exotic terms and the induction principle. It is easy now, using the higher-order constructors, to define inductively the *well-formed* terms of type LL:

Inductive Definition $wf :\ LL \rightarrow$ Prop
$= wf\_ref : \forall n : nat.(wf\ (\mathcal{R}ef\ n))$
$\mid wf\_lam :\ \forall e : LL.\ (wf\ e) \rightarrow (wf\ (\mathcal{L}am\ e))$
$\mid wf\_app : \forall a, b : LL.\ (wf\ a) \rightarrow (wf\ b) \rightarrow (wf\ (\mathcal{A}pp\ a\ b))$.

From this definition, Coq generates the following induction principle (more exactly an equivalent version of it):

$\forall P : LL \rightarrow Prop.$
$(\forall n : nat.(P\ (\mathcal{R}ef\ n)))$
$\rightarrow\ (\forall e : LL.\ (P\ e) \rightarrow (P\ (\mathcal{L}am\ e)))$
$\rightarrow\ (\forall a : LL.\ (P\ a) \rightarrow \forall b : LL.\ (P\ b) \rightarrow (P\ (\mathcal{A}pp\ a\ b)))$
$\rightarrow\ \forall e : LL.(wf\ e) \rightarrow (P\ e).$

This induction principle is the exact counterpart of the induction principle generated by the definition of fol. We shall also need the counterpart of the induction principle generated by the definition of *valid*. For that, we just have to define the counterpart of *valid*:

Inductive Definition $Valid : \forall n : nat.\ LL \rightarrow$ Prop
$= Valid\_ref : \forall n, i : nat.(i < n) \rightarrow (Valid\ n\ (\mathcal{R}ef\ i))$
$\mid Valid\_lam : \forall n : nat.\ \forall e : LL.\ (Valid\ (S\ n)\ e) \rightarrow (Valid\ n\ (\mathcal{L}am\ e))$
$\mid Valid\_app : \forall n : nat. \forall a, b : LL.(Valid\ n\ a) \rightarrow (Valid\ n\ b) \rightarrow (Valid\ n(\mathcal{A}pp\ a\ b))$

Coarse injection principle. The injection principle corresponding to our coarse equality is composed as usual of six theorems. Only one of them need a wf assumption:

Theorem $Lam\_app\_var : \forall e : LL. \forall a, b : LL.\ \neg(eq_{LLv}\ (\mathcal{L}am\ e)\ (\mathcal{A}pp\ a\ b)).$
Theorem $Lam\_lam\_var : \forall a, b : LL.(wf\ a) \rightarrow (wf\ b) \rightarrow$
$(eq_{LLv}\ (\mathcal{L}am\ a)\ (\mathcal{L}am\ b)) \rightarrow (eq_{LLv}\ a\ b).\ \ldots$

The proofs of these theorems mainly use the inversion of eq_L and the second injection principle given above. In addition to that the proof of theorem *Lam_lam_var* uses the following lemma (whose proof is straightforward):

Theorem $wf\_ext\_eq_L = \forall e : LL.(wf\ e) \rightarrow \forall x, y : (list\ L).(\forall n : nat.(x\ n) = (y\ n))$
$\rightarrow (eq_L\ (e\ x)\ (e\ y)).$

Thanks to the coarse injection principle, we can prove that our two notions of equality coincide on *well-formed* terms:

Theorem $eq_{LLv}\_eq_{LL} : \forall a : LL.(wf\ a) \rightarrow \forall b : LL.(wf\ b) \rightarrow$
$(eq_{LLv}\ a\ b) \rightarrow (eq_{LL}\ a\ b).$

Final injection principle. The second and coarse injection principles make it possible to prove our final injection principle. The theorem $eq_{LLv}\_eq_{LL}$ given in the previous paragraph is a decisive tool in the proof of the theorems $\mathcal{L}am\_lam$ and $\mathcal{A}pp\_app$, which are the only ones who need a *well-formed* assumption:

Theorem $\mathcal{L}am\_app : \forall e : LL. \forall a, b : LL.\ \neg(eq_{LL}\ (\mathcal{L}am\ e)\ (\mathcal{A}pp\ a\ b)).$
Theorem $\mathcal{L}am\_lam : \forall a : LL.(wf\ a) \rightarrow \forall b : LL.(wf\ b)$
$\rightarrow\ (eq_{LL}\ (\mathcal{L}am\ a)\ (\mathcal{L}am\ b)) \rightarrow (eq_{LL}\ a\ b).\ \ldots$

2.4 Proof of translation

In this section we shall describe a proof of correctness of our translation from the first-order syntax fol into the higher-order syntax L. Remember that we consider the variant of the translation given in 2.3 suited to our notion of equality. This example of proof illustrates two of our goals. Firstly, it is an example of a proof of adequacy of syntaxes. Secondly, it is an example of a proof in semantics, that makes intensive use of the tools that we have developed in the previous sections.

The correctness of our translation consists in four theorems, stating that the $trans$ relation is a surjective function in both directions:

Theorem $trans\_sur\_l : \forall e : fol.\forall n : nat.(valid\ n\ e)$
$\rightarrow \exists e' : LL.(Valid\ n\ e') \wedge (trans\ e\ e')$.
Theorem $trans\_sur\_r : \forall e' : LL.\forall n : nat.(Valid\ n\ e')$
$\rightarrow \exists e : fol.(valid\ n\ e) \wedge (trans\ e\ e')$.
Theorem $trans\_lr : \forall n : nat.\forall e : fol.(valid\ n\ e) \rightarrow \forall a : LL.(Valid\ n\ a)$
$\rightarrow (trans\ e\ a) \rightarrow \forall b : LL.(Valid\ n\ b) \rightarrow (trans\ e\ b) \rightarrow (eq_{LL}\ a\ b)$.
Theorem $trans\_rl : \forall n : nat.\forall e : fol.(valid\ n\ e) \rightarrow \forall e' : LL.(Valid\ n\ e')$
$\rightarrow (trans\ e\ e') \rightarrow \forall f : fol.(valid\ n\ f) \rightarrow (trans\ f\ e') \rightarrow (e = f)$.

These four theorems are easily reduced to corresponding lemmas which do not involve the $valid$ or $Valid$ conditions. The proofs of the first two lemmas, stating surjectivity, are straightforward. The proofs of the last two lemmas proceed by a double induction on trans, (more exactly induction on $trans$ followed by an inversion of $trans$). These proofs use all the injection principles given in the previous subsections.

2.5 Alternative approaches

The reason why we had to introduce ad-hoc equalities could be concentrated in the following statement:

Axiom $ext_l : \forall A : Set.\forall f, g : (list\ A).\ (\forall x : nat.(f\ x) = (g\ x)) \rightarrow f = g$.

This axiom is certainly not provable, since it modifies Coq's object equality by identifying for example the identity function on nat and the following term of type $nat \rightarrow nat$, which are extensionally equal:

$\lambda x : nat.\ (Match\ x\ with\ \ (*\ 0\ *)\ 0\ \ (*\ (S\ x)\ *)\ \lambda x, h_x : nat.(S\ x))$.

If we assume this ext_l axiom, we are able to implement our language and to prove the higher-order injection principle in a simpler way, without introducing ad-hoc equalities. Thus we wonder if this very natural axiom could be assumed without making the whole system inconsistent.

A less controversial -and still sufficient- axiom is the following one (where wf is defined using $=$ instead of eq_{LL}):

Axiom $ext_{wf} = \forall e : LL.(wf\ e) \rightarrow \forall x, y : (list\ L)$.
$(\forall n : nat.(x\ n) = (y\ n)) \rightarrow (e\ x) = (e\ y)$.

It seems quite difficult to prove this axiom at the object-level. However, it is possible to prove it at the meta-level.

3 Generalization

In this section, we explain how the ideas described above make it possible to implement in Coq, any second-order abstract syntax in the sense of [DH94], together with the corresponding induction, injection and extensionality principles.

3.1 The data

We consider here an arbitrary second-order abstract syntax, given for instance by a LF signature as follows:

$L_1, \dots, L_n : Type;$
$c_1 : T_1;\ \cdots\ ; c_m : T_m;$

where the L_i's (denoting types) and the c_i's (denoting constructors of these types) are identifiers, while the T_i's belong to the grammar T defined as follows:

$L = L_1 \mid \dots \mid L_n;$
$A = L \mid L \to A;$
$T = L \mid A \to T;$

To the previous sequence of grammars, we associate the following one, which is suited for replacing, in terms of T, negative occurrences of L's by the corresponding V's:

$L = L_1 \mid \dots \mid L_n;$
$V = V_1 \mid \dots \mid V_n;$
$A^v = L \mid V \to A^v;$
$T^v = L \mid A^v \to T^v;$

Since there is a natural bijection between the terms in L and the terms in V, there is also a natural bijection between the terms in A and the terms in A^v, and also between the terms in T and the terms in T^v. We denote by X^v the term associated with the term X by this bijection. We write $T_i = m_{i,1} \to \dots \to m_{i,a_i} \to L_{p_i}$ and $m_{i,j} = L_{q_{i,j,1}} \to \dots \to L_{q_{i,j,b_{i,j}}} \to L_{q_{i,j,0}}$.

Thus c_i has type:

$(L_{q_{i,1,1}} \to \dots \to L_{q_{i,1,b_{i,1}}} \to L_{q_{i,1,0}}) \to \dots$
$\to (L_{q_{i,j,1}} \to \dots \to L_{q_{i,j,b_{i,j}}} \to L_{q_{i,j,0}}) \to \dots$
$\to (L_{q_{i,a_i,1}} \to \dots \to L_{q_{i,a_i,b_{i,a_i}}} \to L_{q_{i,a_i,0}})$
$\to L_{p_i}.$

The p, q, a, b's are, together with n and m, the integers encoding the syntax (its arity in the terminology of [DH94]). Observe that a and b are families of arbitrary integers (depending on one and two integers respectively) and that p and q are in $1 \dots n$ (depending on one and three integers respectively). We fix the natural convention that if $(a_i = 0)$ then $c_i = L_{p_i}$ and if $(b_{i,j} = 0)$ then $m_{i,j} = L_{q_{i,j,0}}$.

3.2 Preliminary declarations

Now we describe a list of Coq declarations which are necessary for our implementation of this syntax. For this we introduce some further notations.

At first, because there is no mutual inductive definitions in Coq, we have to define our types L_i as a single dependent type. There is a standard way[DFH+93] to do this. First we define the type for parameters as an inductive type with n elements:

Inductive Set $Param = P_1 : Param \mid \dots \mid P_n : Param.$

Then we use as above a type var, inhabited by at least two distinct values:

Variable var : Set. Axiom $var_2 : \exists x, y : var.\ (x \neq y).$

We build typed variables by multiplying the type var and the type $Param$:

Inductive Type $pvar : Param \rightarrow Set = Pvar : \forall p : Param.var \rightarrow (pvar\ p).$

3.3 Provisional syntax

Now we can introduce the provisional syntax as follows:

Inductive Type $L : Param \rightarrow Set$

$= Var : \forall i : Param.(pvar\ i) \rightarrow (L\ i) \mid C_1 : T_1^w \mid \dots \mid C_m : T_m^w.$

where T_i^w is the term built from T_i^v by substituting $(L\ j)$ for occurrences of L_j, and $(pvar\ j)$ for occurrences of V_j (for all values of j).

Thus C_i has type T_i^w. We write $T_i^w = t_{i,1} \rightarrow \dots \rightarrow t_{i,a_i} \rightarrow L_{p_i}$ where $t_{i,j} = (pvar\ q_{i,j,1}) \rightarrow \ \dots\ \rightarrow (pvar\ q_{i,j,b_{i,j}}) \rightarrow (L\ q_{i,j,0}).$

The injection principle for the type L consists of four series of theorems. The first series contains only the following:

Theorem $Var\_Var : \forall p : Param.\forall x, y : (pvar\ p).$
$((Var\ p\ x) = (Var\ p\ y)) \rightarrow (x = y).$

Then for each $i \in 1 \dots m$ we have:

Theorem $C_i\_C_i : \forall x_1, y_1 : t_{i,1}.\dots\forall x_{a_i}, y_{a_i} : t_{i,a_i}.$
$((C_i\ x_1 \dots x_{a_i}) = (C_i\ y_1 \dots y_{a_i})) \rightarrow (x_1 = y_1) \wedge \dots \wedge (x_{a_i} = y_{a_i}).$

For each $j \in 1 \dots m$ we also have:

Theorem $Var\_C_j : \forall x : (pvar\ p_j).\forall y_1 : t_{j,1}.\dots\forall y_{a_j} : t_{j,a_j}.$
$\neg((Var\ p_j\ x) = (C_j\ y_1 \dots y_{a_j})).$

And for any couple $i < j$ in $1 \dots m$ satisfying $p_i = p_j$, we have:

Theorem $C_i\_C_j : \forall x_1 : t_{i,1}.\dots\forall x_{a_i} : t_{i,a_i}.\forall y_1 : t_{j,1}.\dots\forall y_{a_j} : t_{j,a_j}.$
$\neg((C_i\ x_1 \dots x_{a_i}) = (C_j\ y_1 \dots y_{a_j})).$

The ad-hoc equality is introduced by the following definition:

Inductive Definition $eq_L : \forall i : Param.(L\ i) \rightarrow (L\ i) \rightarrow Prop$
$= eq_L\_var : \forall i : Param.\forall x : (pvar\ i).(eq_L\ i\ (Var\ i\ x)\ (Var\ i\ x))$
$\mid eq_L\_C_1 : \ldots$
$\mid \ldots$
$\mid eq_L\_C_i : \forall x_1, y_1 : t_{i,1}.(\forall z_1 : (pvar\ q_{i,1,1}).\ldots\forall z_{b_{i,1}} : (pvar\ q_{i,1,b_{i,1}}).$
$(eqL\ q_{i,1,0}\ (x_1\ z_1\ \ldots\ z_{b_{i,1}})\ (y_1\ z_1\ \ldots\ z_{b_{i,1}})))$
$\rightarrow \ldots$
$\rightarrow \forall x_{a_i}, y_{a_i} : t_{i,a_i}.(\forall z_1 : (pvar\ q_{i,a_i,1}).\ldots\forall z_{b_{i,a_i}} : (pvar\ q_{i,a_i,b_{i,a_i}}).$
$(eqL\ q_{i,a_i,0}\ (x_{a_i}\ z_1 \ldots z_{b_{i,a_i}})(y_{a_i}\ z_1 \ldots z_{b_{i,a_i}})))$
$\rightarrow (eq_L\ p_i\ (C_i\ x_1 \ldots x_{a_i})\ (C_i\ y_1 \ldots y_{a_i}))$
$\mid \ldots$
$\mid eq_L\_C_m : \ldots$

The second ground injection principle is again a package of four series of theorems. We only give one of them here. The others are a simple modification of the injection principle given above, where Coq's object equality on L is replaced by our eq_L equality. For each $i \in 1 \cdots m$ we have:

Theorem $C_i\_C_i\_eq : \forall x_1, y_1 : t_{i,1}.\cdots\forall x_{a_i}, y_{a_i} : t_{i,a_i}.$
$(eq_L\ p_i\ (C_i\ x_1 \cdots x_{a_i})\ (C_i\ y_1 \cdots y_{a_i}))$
$\rightarrow (\forall z_1 : (pvar\ q_{i,1,1}).\cdots\forall z_{b_{i,1}} : (pvar\ q_{i,1,b_{i,1}}).$
$(eqL\ q_{i,1,0}\ (x_1\ z_1\ \cdots\ z_{b_{i,1}})\ (y_1\ z_1\ \cdots\ z_{b_{i,1}})))$
$\wedge \cdots$
$\wedge(\forall z_1 : (pvar\ q_{i,a_i,1}).\cdots\forall z_{b_{i,a_i}} : (pvar\ q_{i,a_i,b_{i,a_i}}).$
$(eq_L\ q_{i,a_i,0}\ (x_{a_i}\ z_1\ \cdots\ z_{b_{i,a_i}})\ (y_{a_i}\ z_1\ \cdots\ z_{b_{i,a_i}}))).$

3.4 Higher-order setting

Next, our final type is the type of functions with values in $(L\ p)$ depending on n lists of arguments (one for each type):

Definition $mlist = \lambda L : (Param \rightarrow Set).\forall p : Param.nat \rightarrow (L\ p).$
Definition $LL = \lambda p : Param.(mlist\ L) \rightarrow (L\ p).$

Next, we define the higher-order constructors $\mathcal{C}_i$.

We define in general $\mathcal{C}$ for each term C of a type T of the form:

$T = \quad ((pvar\ q_{1,1}) \rightarrow \cdots \rightarrow (pvar\ q_{1,b_1}) \rightarrow (L\ q_{1,0})) \rightarrow \cdots$
$\rightarrow ((pvar\ q_{a,1}) \rightarrow \cdots \rightarrow (pvar\ q_{a,b_a}) \rightarrow (L\ q_{a,0})) \rightarrow (L\ p).$

The term $\mathcal{C}$ is the following Coq term of type $(LL\ q_{1,0}) \rightarrow \cdots \rightarrow (LL\ q_{a,0}) \rightarrow (LL\ p)$:

$\mathcal{C} = \quad \lambda z_1 : (LL\ q_{1,0}).\ \cdots\lambda z_a : (LL\ q_{a,0}).\lambda z : (mlist\ L).$
$(C\ \lambda x_1 : (pvar\ q_{1,1}).\cdots\lambda x_{b_1} : (pvar\ q_{1,b_1}).$
$(z_1\ (append\ ((Var\ q_{1,1}\ x_1)\cdots(Var\ q_{1,b_1}\ x_{b_1}))\ z))\ \ldots$
$\lambda x_1 : (pvar\ q_{a,1}).\cdots\lambda x_{b_a} : (pvar\ q_{a,b_a}).$
$(z_a\ (append\ ((Var\ q_{a,1}\ x_1)\cdots(Var\ q_{a,b_a}\ x_{b_a}))\ z))).$

Where $(append\ (s_1\ \cdots s_p)\ z)$ denotes the Coq term z in case $p = 0$, and the Coq term $(append\ (s_1\ \cdots s_{p-1})\ (mcons\ s_p\ z))$ otherwise; where again, for s of type $(L\ P_i)$, $(mcons\ s\ z)$ denotes the Coq term

$\lambda p : Param.\ (Match\ p\ with$
$(*\ p_1\ *)\ (z\ p_1)\ \cdots\ \ (*\ p_i\ *)\ (cons\ (L\ p_i)\ s\ (z\ p_i))\ \cdots\ \ (*\ p_n\ *)\ (z\ p_n)).$
The projections $\mathcal{R}ef : \forall i : Param.\forall n : nat.(mlist\ L) \rightarrow (L\ i)$ are defined by:
$\mathcal{R}ef = \lambda i : Param.\lambda n : nat.\lambda z : (mlist\ L).(z\ i\ n).$

In the following definition of our coarse equality, the term $mmap$, standing for 'multi-map', is the evident one:
Definition $eq_{LL_v} = \lambda p : Param.\lambda E, E' : (LL\ p).\forall x : (mlist\ pvar).$
$(eq_L\ p\ (E\ (mmap\ var\ L\ Var\ x))\ (E'\ (mmap\ var\ L\ Var\ x))).$

Here is now our final notion of equality, which is finer than the previous one:
Definition $eq_{LL} = \lambda p : Param.\lambda e, e' : (LL\ p).\forall x : (mlist\ L).(eq_L\ p\ (e\ x)\ (e'\ x)).$
Theorem $eq_{LL}\_eq_{LL_v} : \forall p : Param.\forall e, e' : (LL\ p).(eq_{LL}\ p\ e\ e') \rightarrow (eq_{LL_v}\ p\ e\ e').$

Next we define the *well-formed* terms of type LL. This definition generates the desired induction principle.
Inductive Definition $wf :\ \forall p : Param.(LL\ p) \rightarrow$ Prop
$= wf\_ref : \forall p : Param.\forall n : nat.\forall E : (LL\ p).(eq_{LL}\ p\ E\ (\mathcal{R}ef\ p\ n)) \rightarrow (wf\ p\ E)$
$|\ wf\_C_1 :\ \forall E : (LL\ p_1).\forall E_1 : (LL\ q_{1,1,0}).\cdots\forall E_{a_1} : (LL\ q_{1,a_1,0}).$
$(eq_{LL}\ p_1\ E\ (\mathcal{C}_1\ E_1\ \cdots E_{a_1})) \rightarrow (wf\ q_{1,1,0}\ E_1) \rightarrow \cdots \rightarrow (wf\ q_{1,a_1,0}\ E_{a_1})$
$\rightarrow (wf\ p_1\ E)$
$|\ \cdots$
$|\ wf\_C_m :\ \forall E : (LL\ p_m).\forall E_1 : (LL\ q_{m,1,0}).\cdots\forall E_{a_m} : (LL\ q_{m,a_m,0}).$
$(eq_{LL}\ p_m\ E\ (\mathcal{C}_m\ E_1\ \cdots E_{a_m})) \rightarrow (wf\ q_{m,1,0}\ E_1) \rightarrow \cdots \rightarrow (wf\ q_{m,a_m,0}\ E_{a_m})$
$\rightarrow (wf\ p_m\ E).$

For the definition of $Valid$ we need the numbers $\delta_{i,j,p}$ of arguments of type L_p of terms of type $m_{i,j}$. In the following definition, for a meta-level integer d and an object-level integer v, we denote by $v+d$ the object-level integer which is defined as v if $d = 0$ and as $(S\ v+(d-1))$ otherwise.
Inductive Definition $Valid :\ \forall v_1, \cdots, v_n : nat.\forall p : Param.(LL\ p) \rightarrow$ Prop
$= Valid\_ref\_1 :\ \ldots$
$|\ Valid\_ref\_i : \forall E : (LL\ P_i).\forall v_1, \cdots, v_n : nat.\forall k : nat.\ (k < v_i)$
$\rightarrow (eq_{LL}\ P_i\ E\ (\mathcal{R}ef\ P_i\ k)) \rightarrow (Valid\ v_1 \cdots v_n\ P_i\ E)$
$|\ \cdots$
$|\ Valid\_C_i :\ \forall E : (LL\ p_i).\forall E_1 : (LL\ q_{i,1,0}).\cdots\ \forall E_{a_i} : (LL\ q_{i,a_i,0}).$
$(eq_{LL}\ p_i\ E\ (\mathcal{C}_i\ E_1\ \cdots E_{a_i})) \rightarrow$
$(Valid\ (v_1 + \delta_{i,1,1}) \cdots (v_n + \delta_{i,1,n})\ q_{i,1,0}\ E_1) \rightarrow \cdots \rightarrow$
$(Valid\ (v_1 + \delta_{i,a_i,1}) \cdots (v_n + \delta_{i,a_i,n})\ q_{i,a_i,0}\ E_{a_i}) \rightarrow (Valid\ v_1 \cdots v_n\ p_i\ E).$
$|\ \cdots$
$|\ Valid\_C_m :\ \ldots$

The first part of the injection principle corresponding to our coarse equality, namely the theorems

Var_Var_var, $Var\_C_j\_var$ and $C_i\_C_j\_var$ do not contain any wf-assumption. We expand the last series $C_i\_C_i\_var$ which do contain these assumptions.

Theorem $C_i\_C_i\_var : \forall x_1, y_1 : (LL\ q_{i,1,0}). \cdots \forall x_{a_i}, y_{a_i} : (LL\ q_{i,a_i,0}).$
$(wf\ q_{i,1,0}\ x_1) \rightarrow (wf\ q_{i,1,0}\ y_1) \rightarrow \cdots (wf\ q_{i,a_i,0}\ x_{a_i}) \rightarrow (wf\ q_{i,a_i,0}\ y_{a_i})$
$\rightarrow (eq_{LLv}\ p_i\ (\mathcal{C}_i\ x_1 \cdots x_{a_i})\ (\mathcal{C}_i\ y_1 \cdots y_{a_i}))$
$\rightarrow (eq_{LLv}\ q_{i,1,0}\ x_1\ y_1) \wedge \cdots \wedge (eq_{LLv}\ q_{i,a_i,0}\ x_{a_i}\ y_{a_i}).$

The proof of the above theorem uses the following lemma:

Theorem $wf\_ext\_eq_L = \forall p : Param.\forall E : (LL\ p).(wf\ p\ E) \rightarrow \forall x, y : (mlist\ L).$
$(\forall q : Param.\forall n : nat.(x\ q\ n) = (y\ q\ n)) \rightarrow (eq_L\ p\ (E\ x)\ (E\ y)).$

Thanks to the coarse injection principle, we can state that our two notions of equality coincide on *well-formed* terms:

Theorem $eq_{LLv}\_eq_{LL} : \forall p : Param.\forall a : (LL\ p).(wf\ p\ a) \rightarrow \forall b : (LL\ p).(wf\ p\ b)$
$\rightarrow (eq_{LLv}\ p\ a\ b) \rightarrow (eq_{LL}\ p\ a\ b).$

As before, the first part of the injection principle corresponding to eq_{LL}, namely the theorems $\mathcal{R}ef\_ref$, $\mathcal{R}ef\_\mathcal{C}_j$ and $\mathcal{C}_i\_\mathcal{C}_j$ do not contain any wf-assumption. We expand the last series $\mathcal{C}_i\_\mathcal{C}_i$ which do contain such an assumption.

Theorem $\mathcal{C}_i\_\mathcal{C}_i : \forall x_1 : (LL\ q_{i,1,0}).(wf\ q_{i,1,0}\ x_1) \rightarrow \forall y_1 : (LL\ q_{i,1,0}).(wf\ q_{i,1,0}\ y_1)$
$\rightarrow \cdots \rightarrow \forall x_{a_i} : (LL\ q_{i,a_i,0}).(wf\ q_{i,a_i,0}\ x_{a_i}) \rightarrow \forall y_{a_i} : (LL\ q_{i,a_i,0}).(wf\ q_{i,a_i,0}\ y_{a_i})$
$\rightarrow (eq_{LL}\ p_i\ (\mathcal{C}_i\ x_1 \cdots x_{a_i})\ (\mathcal{C}_i\ y_1 \cdots y_{a_i}))$
$\rightarrow (eq_{LL}\ q_{i,1,0}\ x_1\ y_1) \wedge \cdots \wedge (eq_{LL}\ q_{i,a_i,0}\ x_{a_i}\ y_{a_i}).$

3.5 Adequacy

In this subsection, we discuss correctness and adequacy of the generalization proposed in the previous subsections. The proofs of the two theorems stated in this subsection are straightforward, although tedious. We do not state them here because of space limitation.

The correctness is expressed by the following (meta) theorem:

Theorem 1. *Consider a second-order abstract syntax (given for instance through its arity, namely the package of integers n, m, a, b, p, q as above). Then:*

(i) The corresponding sequence of definitions (inductive or not) listed in this section is a correct sequence of Coq definitions.

(ii) The types of the corresponding sequence of theorems listed above are correct Coq types.

(iii) All these theorems have object proofs.

(iv) The meta-form of all these theorems is true.

In order to state adequacy, we have to build some category [DH94]. Let S be a second-order abstract syntax given as in the previous statement, with the integers n, m, a, b, p, q. The corresponding sequence of definitions listed above generates Coq types $(L\ P_i)$'s and

$(LL\ P_j)$'s, the predicate $Valid$ and equalities eq_L and eq_{LL} from which we build a Cartesian category S_{Coq} as follows.

The objects of S_{Coq} are (indexed by) sequences of n natural integers, and product of objects corresponds to addition of sequences. We denote by L^I the object indexed by the sequence $I = (i_1, \ldots,\ i_n)$. In order to describe morphisms in S_{Coq}, since it is Cartesian, it is sufficient to describe $Hom(L^I, X)$ for indecomposable X's. The indecomposable objects are those indexed by the indecomposable indices, namely $I_1 := (1,\ 0, \ldots,\ 0), \ldots,\ I_n := (0, \ldots,\ 0,\ 1)$. We take for $Hom(L^I, L^{I_j})$ the set of Coq terms t of type $(LL\ P_j)$ satisfying $(Valid\, i_1 \cdots i_n\ t)$, modulo the equivalence relation $(eq_{LL}\ P_j)$ (we hope the category structure is sufficiently apparent; to settle it, one should use the properties of eq_{LL} listed above). Now, we can state the adequacy statement.

Theorem 2. *Let S be a second-order abstract syntax given as in the previous statement. Then the cartesian category S_{Coq} is naturally isomorphic with the first-order part of S.*

Thus, for a second-order abstract syntax given by a LF signature as in 3.1, we have implemented first order terms (i.e. terms whose type has shape $L_{i_1} \to \cdots \to L_{i_n} \to L_{i_0}$) as classes (with respect to some object-level equivalence relation, here eq_{LL}) of (tuples of) terms of some object-level type (here, roughly speaking, LL) satisfying some object-level predicate (here $Valid$).

Note that, in contrast with the natural LF implementation, we only implement first-order terms. This is sufficient for semantics purposes.

4 Related Work

The general notion of Higher-Order Abstract Syntax has been introduced in [EP88] and is currently being revisited in [DH94]. Higher-order abstract syntax is now commonly used, at least by people who use either the λ-Prolog language [MN88],or the Elf language [Pfe89] (an implementation of the LF Logical Framework), both for writing semantics of languages [HM88][Har90] and for developing proofs in those semantics [HP92][MP91][PR91]. Proofs in Elf [HP92][MP91][PR91] use induction. All these proofs rely on the introduction of an adequate induction principle.

On the other hand, using systematically Coq's inductive types, Gerard Huet developed in Coq a theory of simply-typed λ-calculus with complete proofs in the first-order setting.

To our knowledge, the method described in the present paper is the first one which allows writing semantics on higher-order abstract syntax in a system which provides inductive types.

5 Conclusions and Future Work

We have explained how to implement in Coq any second-order abstract syntax together with the corresponding induction, injection and extensionality principles. In performing this task, our main trouble came from the fact that Coq's object equality is not extensional. We have also produced samples of proofs using extensively these principles. Our work would not be relevant if our implementation of second-order syntax did not allow smooth formulation and object proofs for semantics. Although not presented here, we have already gathered a lot of positive experience about this and our next task is to present them in a systematic treatment. Just to satisfy the curiosity of the reader, we give here a rule for β-reduction:

$red_\beta : \forall e : LL.\forall v : LL.(red\ (\mathcal{A}pp\ (\mathcal{L}am\ e)\ v)\ \lambda x : (list\ L).\ (e\ (cons\ (v\ x)\ x)))$.

Another task is to design and implement a top-level over Coq providing user-friendly support for implementing in our way object second-order syntaxes and performing object proofs on them. On the other hand, before tightening definitely our project to Coq, it seems reasonable to explore other theorem provers equipped with induction, in particular HOL and Isabelle, in order to verify that the difficulties we have encountered could not be overcome there in an easier way.

Acknowledgements Thanks go to Frank Pfenning for useful comments on an earlier draft version of the paper. We would like to thank Amy Felty and Christine Paulin-Mohring for many fruitful discussions, and more generally the Coq team for their quick and helpful e-mail answers. Finally we thank Yves Bertot for his very comfortable interface for the Coq system.

References

[AHM87] A. Avron, F. Honsell, and A. Mason. Using typed λ-calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, Edinburgh University, July 1987.

[DFH+93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, Ch. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide, version 5.8. Technical Report 154, Inria, Rocquencourt, France, May 1993.

[DH94] Th. Despeyroux and A. Hirschowitz. A categorical approach to higher-order abstract syntax. forthcoming paper, 1994.

[EP88] C. Elliot and F. Pfenning. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN' 88 International Conference on Programming Language Design and Implementation, Atlanta, Georgia, USA*, June 22-24, 1988.

[Har90] R. Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In IEEE, editor, *Proceedings of the second LICS International Conference on Logic In Computer Sciences, Cornell, USA*, pages 194–204, 1987.

[HHP91] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. Technical Report ECS-LFCS-91-162, Edinburgh University, June 1991.

[HM88] J. Hannan and D. Miller. Enriching a meta-language with higher-order features. In *Proceedings of the Workshop on Meta-Programming in Logic Programming, Bristol*, June 1988.

[HP92] J. Hannan and F. Pfenning. Compiler verification in LF. In IEEE, editor, *Proceedings of the LICS International Conference on Logic In Computer Sciences, Santa Cruz, California*, June 1992.

[Hue92] G. Huet. Constructive computation theory. part I. Lecture notes. October 1992.

[Kah87] G. Kahn. Natural semantics. In *Proceedings of the Symp. on Theorical Aspects of Computer Science, Passau, Germany*, 1987. also available as a Research Report RR-601, Inria, Sophia-Antipolis, February 1987.

[MN88] D. Miller and G. Nadathur. An overview of λ-prolog. In MIT Press, editor, *Proceedings of the International Logic Programming Conference, Seattle, Washington*, pages 910–827, August 1988.

[MP91] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in elf. In Lars Halln"as, editor, *Proceedings of the Second Workshop on Extentions of Logic Programing*, Springer-Verlag LNCS, 1991. also available as a Technical Report MPI–I–91–211, Max-Planck-Institute for Computer Science,Saarbrucken, Germany, August 1991.

[Pau89] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[Pfe89] F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the fourth ACM-IEEE Symp. on Logic In Computer Science, Asilomar, California, USA*, June 1989.

[PM92] Ch. Paulin-Mohring. Inductive definitions in the system coq. rules and properties. In J.F. Groote M. Bezem, editor, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, Springer-Verlag LNCS 664, pages 328–345, 1992. also available as a Research Report RR-92-49, Dec. 1992, ENS Lyon, France.

[PR91] F. Pfenning and E. Rohwedder. Implementing the meta-theory of deductive systems. In *Proceedings of the CADE-11 Conference*, 1991.

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