Principles for Functional Abstract Syntax

Thierry Despeyroux[1], André Hirschowitz[2]

[1] INRIA-Sophia Antipolis, B.P 93, F-06902 Sophia Antipolis Cedex, France
`thierry.despeyroux@sophia.inria.fr`
[2] CNRS, URA 168, University of Nice, F-06108 Nice Cedex 2, France
`andre.hirschowitz@sophia.inria.fr`

Abstract In this paper, we give a new, categorical definition for first-order and higher-order abstract syntax (that we call functional abstract syntax), and we state the corresponding induction, separation and recursion principles. We also present a practical application of functional abstract syntax by means of syntactical editing.

Introduction

As compared to concrete syntax, abstract syntax focuses on the tree structure of terms. Indeed, this structure is suited for "syntax-oriented" edition, as well as for semantics specification. It is also the relevant part of the syntax for meta-theoretical reasoning about the object language: an omnipresent ingredient of such reasoning is the principle of induction on the tree structure, together with the companion separation and recursion principles.

Accordingly, trees play a central role in a generic programming environment like Centaur [1].

- Concerning edition: once an abstract syntax is declared, a corresponding editor is generated; this editor acts as client of a (generic) *tree processor* which receives terms (encoded as trees) together with commands or queries, and returns trees or booleans;

- Concerning semantics: the philosophy of "natural semantics" is precisely to specify semantics in terms of elementary (tree) constructions of the language;

- Concerning meta-level reasoning: from a syntax+semantics specification, a Coq file is generated, making available the desired induction and companion principles [15].

The trees concerned in the above processes may contain variables: substitution and unification are two fundamental and delicate operations on these trees.

Recent works on the semantics of programming languages and on proof development systems have shown the need to incorporate the notion of binding into abstract syntax, from which emerged what is called *higher-order abstract syntax* [13]. Higher-order abstract syntax has already been used in some systems such as λ-Prolog [10,11] or Elf [12]. In higher-order abstract syntax, object-level constructions are interleaved with basic meta-level constructions, namely abstraction and application. The corresponding tree structure for terms breaks bindings, and admits sub-trees with "free" variables.

In the present work, we study a slightly different tree structure for terms of a higher-order abstract syntax (that we call rather functional abstract syntax), which we claim better suited for a higher-level handling of the three above mentioned tasks of edition, semantics specification and meta-theoretical reasoning. The main feature of our *higher-order (functional) trees* is that they represent functions: bindings are inherited by sub-terms so that there are no free variables. Accordingly, each of our trees is labeled with a refined meta-level functional type: in low-level language, this functional type keeps track of the ordering of the variables, and of their types, especially in case of variables which eventually do not occur. In other words, our approach replaces low-level basic constructions (namely constructors of the syntax and variables) by their higher-order (functional) counterpart, which we call higher-order constructors and higher-order projections respectively, and which govern higher-order induction and companion principles suited for higher-order meta-theoretical reasoning. As for edition, we specify what should be a functional syntax-oriented editor.

All our definitions are settled in the language of categories. In particular, we start with a categorical definition of functional abstract syntax. Indeed, while some formal definitions of first-order abstract syntax are available (for example through order-sorted algebra in [5,4,8]), higher-order abstract syntax has been considered till now rather as a methodology. Note that abstract syntax as implemented in LF [6] for instance, avoids products by an extensive use of curryfication (as we plan to explain elsewhere, this is one main obstacle for the implementation of higher-order semantics in this system). Instead of mimicing this implementation, we reinsert products in the picture, linking tightly our definitions to the familiar framework of cartesian (closed) categories. Roughly speaking, we define an abstract syntax as a free cartesian (closed in the higher-order case) category. At this point, we had to select a precise notion or cartesian (closed) category, namely to decide for instance whether the product were defined (or commutative, or associative) up to isomorphism or strictly. Here we have departed from the usual track and experienced to identify canonically isomorphic objects as much as possible. Finally, we had to give a convincing definition for freeness of a CCC. For this we have followed a standard track, namely we have built a left-adjoint for a forgetful functor. The correspondence between cartesian closed categories and the semantics of typed λ-calculi has been widely studied (see for example [14]). This is because the notion of category captures and integrates perfectly the notions of function and application. Here we take these notions as fully understood, and build our notion of syntax upon them, thus selecting special kinds of CCCs.

In the study of a first-order abstract syntax, constant use is made of the induction, separation and recursion principles, which roughly say:

- what is true for the terminals and stable by the constructors is true for any term of the syntax

- two terms with different head constructors are different, as well as two terms with the same head constructor but different sons (in some place)

- in order to define a function on terms, it is sufficient to specify it on terminals, and to give, for each constructor, a *recursion* formula in terms of the sons and the value of the function on these sons.

The extension of these three principles to the higher-order case is not straightforward, because in this case, the sons are no more ground terms, but functional ones. Our approach is to consider the functional terms altogether, and to extend naturally the action of the constructors of our syntax to these functional terms. We call these extensions **higher-order constructors** and use them systematically to express our three principles. Since we consider functional terms, when de-structuring trees, we eventually meet *bound variables*. In our approach, these are functional constructions which we call **higher-order projections** and these are the second new ingredient in our three principles. We give special attention to the second-order case, which is the most usual, where we give alternative versions of the induction and recursion principles. These alternative principles are apparently weaker, but they have the advantage of being expressible "in ground terms".

This paper does not give the proofs of the described properties. This may be done by giving an explicit description of the terms of the categories using trees of composable arrows as it is done in [9] p. 50 for strings of composable arrows.

After a first section devoted to our categorical definitions and three sections devoted to the above discussed principles in the first-order, second-order and general cases, we present what could be a practical application of functional abstract syntax by means of syntax-oriented editing.

1 Categorical Preliminaries

The definitions of FOAS (first-order abstract syntax) and HOAS (higher-order abstract syntax) need some preliminary definitions on graphs and categories.

1.1 First-Order Case

In the following, we will write "category" for what is usually called small category (objects from a set), and "Category" for categories that are not necessarily small.

A **cartesian graph** is a graph (see [9] p.10) whose set of vertices is a commutative monoïd (see [9] p.2). We say that a cartesian graph is **free** if so is the monoïd of its vertices. We will also call a free cartesian graph a **factorial graph**.

A **morphism of cartesian graphs** is a morphism of graphs which induces on vertices a morphism of monoïds. The Category $\mathcal{F}G$ of factorial graphs is included in the Category of cartesian graphs ([14]).

We will consider here that a **cartesian category** is a category with distinguished products (see [14] p.18).

We call **factorial category** a cartesian category X where the product makes $obj(X)$ into a free commutative monoïd (in such a way that the categorical identification between $A \times B$ and $B \times A$ is the identity).

We call **base** of a factorial category X the set of indecomposable elements (that we call **atomic**) of $obj(X)$ (The neutral element is not in the base.)

A **morphism of factorial categories** is a functor which induces on objects a morphism of monoïds such that the following diagram (of trifunctors) commutes:

$$\begin{tikzcd} Hom(X, A \times B) \arrow[r, phantom, "\simeq"] \arrow[d] & Hom(X, A) \times Hom(X, B) \arrow[d] \\ Hom(F(X), F(A \times B)) = Hom(F(X), F(A) \times F(B)) \arrow[r, "\sim"] & Hom(F(X), F(A)) \times Hom(F(X), F(B)) \end{tikzcd}$$

The factorial categories form a Category $\mathcal{FC}$ and we have a forgetful functor $\mathcal{FC} \rightarrow \mathcal{FG}$.

Proposition 1. *This functor admits a left adjoint $F_{\mathcal{FC}} : G \Rightarrow F_{\mathcal{FC}}(G)$.*

Definition 2. If G is a factorial graph, we say that $F_{\mathcal{FC}}(G)$ is the **first-order abstract syntax** (FOAS) associated with G, and we call **first-order abstract syntax** (FOAS) any factorial category isomorphic to $F_{\mathcal{FC}}(G)$ for some factorial graph G.

We say that a factorial graph G is **simple** if the target of any of its arrows is a base object.

Proposition 3. *Any FOAS is isomorphic to a FOAS associated with a simple graph.*

Example Let $P_1 ... P_n$ be a set of names (to be understood as type or phylum names), and consider a set names $c_1, ..., c_m$ for arrows in the monoïd generated by the P_i's :

$$c_i : P_1^{a_{i,1}} \times ... \times P_n^{a_{i,n}} \rightarrow P_{b_i}$$

(to be understood as a set of operators given by their signatures). This form a simple factorial graph, and thus specifies a FOAS.

The objects of this category correspond to the terms (trees) of the abstract syntax. Ground terms are objects of type $P_1^0 \times ... \times P_n^0 \rightarrow P_j$ for some j.

To fix ideas, let us take the set of names $\{Nat\}$ containing only one element, and the three operators

$zero : Nat^0 \rightarrow Nat$
$succ : Nat \rightarrow Nat$
$plus : Nat^2 \rightarrow Nat$

We write `s(zero)` a ground term of the category of type $Nat^0 \rightarrow Nat$. λ`a:Nat.plus(a,s(zero))` is a functional term (also called scheme) of type $Nat \rightarrow Nat$. With our representation of FOAS, the sons of a functional term are themselves functional; in the present example λ`a:Nat.a` and λ`a:Nat.s(zero)`.

1.2 Higher-Order Case

For the higher-order case, the structure of monoïd is not sufficient. We call **higher-order monoïd** (HO-monoïd) a commutative monoïd $(E, \times, 1)$ endowed with a second composition law $\rightarrow$ satisfying the four following conditions:

$$\begin{gathered}
A \rightarrow (B \times C) = (A \rightarrow B) \times (A \rightarrow C) \\
B \rightarrow (C \rightarrow A) = (B \times C) \rightarrow A \\
1 \rightarrow A = A \\
A \rightarrow 1 = 1.
\end{gathered}$$

A **morphism of HO-monoïds** is a morphism of monoïds which also respects the second law. HO-monoïds form a Category $\mathcal{HOM}$ and the forgetful functor $\mathcal{HOM} \rightarrow Sets$ admits a left adjoint functor $F_{\mathcal{HOM}}$.

We call **free HO-monoïd** a HO-monoïd isomorphic to $F_{\mathcal{HOM}}(S)$ for some set S. We call **base** of a free HO-monoïd the (unique) minimal set of generators and we say that elements of the base are **atomic**.

In a HO-monoïd M there is a notion of **normal form**: any element of M different from 1 and from the atomic elements decomposes in a unique way (up to order) as a product of indecomposable (for $\times$) elements, and any indecomposable element can be expressed in a unique way as $U \rightarrow V$ with V atomic [2].

We call **factorial closed graph** a graph whose set of vertices is a HO-monoïd.

A **morphism of factorial closed graphs** is a morphism of graphs which induces on vertices a morphism of HO-monoïds.

Factorial closed graphs form a Category $\mathcal{FCG}$.

A **cartesian closed category** is a cartesian category with (distinguished) exponentiation (see [9] p.95).

A **factorial closed category** is a cartesian closed category X where product and exponentiation make $objects(X)$ into a free HO-monoïd in such a way that the four structural (Yoneda) isomorphisms between:

$$\begin{gathered}
A \rightarrow (B \times C) \quad and \quad (A \rightarrow B) \times (A \rightarrow C); \\
B \rightarrow (C \rightarrow A) \quad and \quad (B \times C) \rightarrow A; \\
1 \rightarrow A \quad and \quad A; \\
A \rightarrow 1 \quad and \quad 1;
\end{gathered}$$

are identities.

A **morphism of factorial closed categories** is a morphism of factorial categories which induces on objects a morphism of HO-monoïds such that the following diagram commutes:

$$\begin{tikzcd}
Hom(A \times B, C) \arrow[r, "\sim"] \arrow[d] & Hom(A, B \rightarrow C) \arrow[d] \\
Hom(F(A) \times F(B), F(C)) \arrow[r, "\sim"] & Hom(F(A), F(B \rightarrow C)) = Hom(F(A), F(B) \rightarrow F(C))
\end{tikzcd}$$

Factorial closed categories form a Category $\mathcal{F}CC$ with a forgetful functor $U : \mathcal{F}CC \to \mathcal{F}CG$.

Proposition 4. *This functor admits a left adjoint $F_{\mathcal{F}CC}$.*

The proof produces for each factorial closed graph a concrete category made of (higher-order) trees. We will meet again these trees later on.

Definition 5. If G is a factorial closed graph, we say that $F_{\mathcal{F}CC}(G)$ is the **higher-order abstract syntax** (HOAS) associated with G, and we call **higher-order abstract syntax** (HOAS) any factorial closed category isomorphic to $F_{\mathcal{F}CC}(G)$ for some factorial closed graph G.

We say that a factorial closed graph G is **simple** if the target of any of its arrows is atomic.

Proposition 6. *Any HOAS is isomorphic to a HOAS associated with a simple factorial closed graph.*

In a HOAS, we thus have atomic objects and **base** arrows: these are the images of arrows of G by the unit of adjunction ([9] p.81). The target of each of them is an atomic object.

We say that an abstract syntax is **effective** iff it contains an arrow of type $1 \to L_i$ for every atomic type L_i.

We say that an abstract syntax is **extensional** iff it contains at least two arrows from the final object 1 to L for every atomic object L. For example, the abstract syntax defined by $z : L$ is not extensional because the two (different) terms $\lambda x.x$ and $\lambda x.z$ are equal when applied to the only element z of the syntax.

For an extensional abstract syntax $\mathcal{C}$, the functor $T \to Hom(1, T)$ from $\mathcal{C}$ to $Sets$ is faithful, thus in this case, $\mathcal{C}$ is a concrete category (see [9] p.26). This statement may be called **extensionality principle**.

1.3 Vocabulary

In the syntactical terminology, objects are called **types**. The base arrows are called **constructors**. Morphisms from A to B, with atomic B are called terms of type $A \to B$. We call **terminal** any constructor of type $1 \to L$ with L atomic.

We say that terms of type $1 \to L$ (with L is atomic) are **ground** (or **closed**), and we also say that they have type L.

1.4 Examples

We consider a factorial closed graph with one single atomic type $\{Exp\}$ and three constructors:

$lambda : (Exp \to Exp) \to Exp$
$apply : Exp^2 \to Exp$
$zero : Exp^0 \to Exp.$

For instance, `λa:Exp.apply(lambda(λx:Exp.x), a)` is a term of type $Exp \to Exp$, and the son of its first son is a function of two arguments: `λa,x:Exp.x`.

Higher-order abstract syntax has mainly been used to express the semantics of languages based on a λ-calculus. However it can be used also for imperative languages as shown in the following example.

Consider a language with instructions, expressions and (integer) variables. The base of types is $\{Inst, Exp, Var\}$, and let us define the following operators:

$decl : (Var \to Inst) \to Inst$
$assign : Var \times Exp \to Inst$
$plus : Exp^2 \to Exp$
$var : Var \to Exp$
$zero : Exp^0 \to Exp$

The functional tree representation of the program "declare X; X=X+0" will be `decl(λa:Var.assign(var(a),plus(var(a),zero)))`. Note that in this example there is no operator of type Var. Thus, only an object variable previously declared by means of the `decl` operator can be used in the instruction. This feature can be used for syntactical editing as explained in the last section.

2 FOAS (First-Order Abstract Syntax)

2.1 Setting

In this section we review what are the main features of first-order abstract syntaxes, with respect to induction, separation (injection), extensionality, unification and recursion. As far as open terms are concerned, induction, separation and recursion involve the functional counterparts of variables, namely projections.

Thus, we consider a FOAS S, with its set $\mathcal{T}(S) = \mathcal{T}$ of atomic types, and its set $\mathcal{C}(S) = \mathcal{C}$ of constructors. We also have its set of *arities* $\mathcal{A}(S) = \mathcal{A}$, that is to say the set of applications with finite support from $\mathcal{T}$ to $\mathbf{N}$. We denote by $\mathbf{N}$ the set of non-negative integers, and for n such an integer, we write $\underline{n}$ for the interval $[1, ..., n]$ (if $n = 0$ the interval is empty).

For any arity $a \in \mathcal{A}$, we denote by T^a the corresponding object in S, namely $\Pi_{t \in \mathcal{T}} t^{a(t)}$, as well as the corresponding set of (closed) terms. For instance if $\mathcal{T}$ consists of the two elements *nat* and *bool*, and if $a(nat) = 2$ and $a(bool) = 1$, then T^a is simply $nat^2 \times bool$. Furthermore we denote by T_a^t the set of morphisms from T^a to t. More generally, for b another arity, we denote by T_a^b the set of morphisms from T^a to T^b. Notice that each type is of the form T_a^b.

The set $\mathcal{C}$ is equipped with the two maps *dom*, from $\mathcal{C}$ to $\mathcal{A}$, and *codom*, from $\mathcal{C}$ to $\mathcal{T}$. Finally, we have the set of projections $\mathcal{P}(S) = \mathcal{P}$: this one too is equipped with the two maps *dom*, from $\mathcal{P}$ to $\mathcal{A}$, and *codom*, from $\mathcal{P}$ to $\mathcal{T}$, and we denote by $\mathcal{C}_a$ (resp. $\mathcal{P}_a$) (resp. $\mathcal{C}_{a,t}$ (resp. $\mathcal{P}_{a,t}$)) the subset of $\mathcal{C}$ (resp. $\mathcal{P}$) of elements with domain a (resp. with domain a and codomain t).

2.2 Induction Principles

Already for a FOAS, we have two induction principles, one for closed and one for functional terms.

Proposition 7. (closed FO induction principle) *Let B a collection indexed by $\mathcal{T}$ of sets B_t, B_t being a subset of t. For a an arity, we set $B^a := \Pi_t(B_t)^{a(t)}$.*

We suppose that B is stable by all the constructors, more precisely if C_c is a constructor mapping T^a to t, then it sends B^a into B_t;

Then, for every t, B_t is the whole t.

The induction principle allows inductive proofs on FOAS: if a property is stable by composition with the operators of the syntax, then it is verified by all the closed terms of the syntax. In particular, it allows inductive proofs concerning a single type (or any set of types) of the syntax: just extend the desired property from the concerned type(s) to all types, by declaring that it is true for non concerned types.

For the open induction principle, we need to introduce the *second-order constructors*. For each constructor C_c mapping with domain d and codomain t and each arity a, we denote by C_c^a the corresponding map from T_a^d into T_a^t, and we call it a second-order constructor of the syntax.

Proposition 8. (open FO induction principle) *Let B a collection indexed by $\mathcal{A} \times \mathcal{T}$ of sets B_a^t, B_a^t being a subset of T_a^t.*

We suppose that

(i) B contains the projections, more precisely if a projection $p \in \mathcal{P}$ is in T_a^t, then it is in B_a^t.

(ii) B is stable by all second-order constructors, more precisely if C_c^a is a second-order constructor mapping T_a^d to T_a^k, it sends B_a^d to B_a^k.

Then, for each arity a and atomic type t B_a^t is the whole T_a^t.

The open induction principle is stronger than the closed one in this sense that any property of closed terms which can be proved by the latter can be extended to be true by definition on open terms and proved by the former. For the other way around, there is no such a trivial reduction. One way to connect properties of open terms to the domain of the closed induction principle is to prove that they are **extensional**: the property P is extensional if for any term x of type $\Pi A_i \to A$, with atomic A_i's, $P(x)$ holds if and only if, for any collection (x_i) of closed terms of types (A_i) satisfying the collection of conditions $(P(x_i))$, $P(x((x_i))$ holds.

2.3 Separation Principles

Proposition 9. (closed separation principle): *Let A_1 and A_2 be two ground terms of a FOAS such that $A_j = C_j \prod_{i=1}^{n_j} x_{i,j}$, where C_1 and C_2 are constructors and the $x_{i,j}$ are ground terms. Then $A_1 = A_2$ iff $C_1 = C_2$, $n_1 = n_2$, and $\forall i \in [1, n_1], x_{i,1} = x_{i,2}$.*

Together with the closed induction principle, the closed separation principle endows each closed term with a **tree structure**, through a well defined **head constructor** and well-defined **sons**.

In order to state the open separation principle, we need to identify more carefully the projections.Among morphisms in T_a^t, we have projections: these are labelled by $\underline{a}(t)$. Thus projections are parametrized by the set Θ of triples $\theta = (a, t, k)$ where a is an arity, t is an atomic type, and k is an element of $\underline{a}(t)$.

Proposition 10. (open separation principle) *Let A_1 and A_2 be two terms of a S such that $A_j = C_{c_j}^a \prod_{i=1}^{n_j} x_{i,j}$, where the c_j are in $\mathcal{C}$ and the $x_{i,j}$ are terms. Then $A_1 = A_2$ iff $c_1 = c_2$, $n_1 = n_2$, and $\forall i \in [1, n_1], x_{i,1} = x_{i,2}$ (this concerns in particular terminal constructors, for which n_j is zero). Furthermore, such a term with a head constructor cannot be equal to a projection; and two projections p_θ and $p_{\theta'}$ are equal if and only if the two triples θ and θ' are equal.*

Together with the open induction principle, the open separation principle endows each term with a **first-order tree structure**, through a well defined second-order **head constructor** and well-defined **sons**.

2.4 Recursion

The closed **recursion principle** reads as follows.

Proposition 11. (closed FO recursion principle) *Given*

- a collection B of sets B_t, indexed by T;

- a collection M of maps M_c indexed by $\mathcal{C}$, M_c mapping $T^{dom(c)} \times B^{dom(c)}$ into $B_{codom(c)}$, where $B^{dom(c)}$ is defined as $\Pi_{t \in T} B_t^{dom(c)(t)}$;

there exists a unique collection $r := rec(B, M)$ of maps r_t indexed by T, r_t sending t into B^t, satisfying the collection indexed by $\mathcal{C}$ of reduction rules:

$r_{codom(c)}(C_c((x_{t,k})_{k \in \underline{d}om(c)(t)})) = M_c((x_{t,k})_{k \in \underline{d}om(c)}, (r_t(x_{t,k}))_{k \in \underline{d}om(c)})$.

The open recursion principle generates maps from sets of open terms instead of maps from sets of closed terms; and accordingly, it involves projections. It also involves what we call the *second-order constructors*: these are naturally associated with the constructors in the following way: il C_c has domain d and codomain k, and a is an arity, then the second-order constructor C_c^a is the map from T_a^d to T_a^k defined by composition with C_c.

Proposition 12. (open FO recursion principle) *Given*

- a collection B, indexed by $\mathcal{A} \times \mathcal{T}$ of sets $B_{a,t}$;

- a collection M, indexed by $\mathcal{C} \times \mathcal{A}$ of maps $M_{a,c}$ sending $T_a^{dom(c)} \times B_a^{dom(c)}$ into $B_{a,codom(c)}$, where B_a^b stands for $\Pi_{t \in \mathcal{T}} B_{a,t}^{b(t)}$;

and in addition

- a collection M', indexed by $\mathcal{P}$ of elements M'_p, with $M'_p \in B_{dom(p),codom(p)}$;

there exists a unique collection $r := rec(B, M)$ of maps $r_{a,t}$ indexed by $\mathcal{A} \times \mathcal{T}$, $r_{a,t}$ sending T_a^t into $B_{a,t}$, satisfying the two collections of reduction rules:

$r_{a,codom(c)}(C_c^a((x_{t,k})_{k \in \underline{d}om(c)(t)})) = M_{a,c}((x_{t,k})_{k \in \underline{d}om(c)(t)}, (r_{a,t}(x_{t,k}))_{k \in \underline{d}om(c)(t)})$

and

$r_{dom(p),codom(p)}(p) = M'_p$.

Remark This recursion principle builds for each a a collection r_a of maps $r_{a,t}$, and all of these constructions are independent. This will no more hold true in the second-order case, to be considered next.

2.5 Unification

Definition 13. Two terms A and B of the same type **unify** iff there exists a term C called **unifier** of A and B such that $A \circ C = B \circ C$. A **most general unifier** is an equalizer (see [9] p. 70).

Proposition 14. (unification principle) *If two terms unify, a most general unifier for these two terms exists.*

3 SOAS (Second-Order Abstract Syntax)

3.1 Setting

In this section we review what are the main features of second- order abstract syntaxes. At first we focus on first-order terms of such a syntax.

Thus, we consider a SOAS S, with its set $\mathcal{T}^0(S) = \mathcal{T}^0$ of atomic types, and its set $\mathcal{C}(S) = \mathcal{C}$ of constructors. We also have its set of 0-*arities* $\mathcal{A}^0(S) = \mathcal{A}^0$, which is as above the set of applications with finite support from $\mathcal{T}^0$ to $\mathbf{N}$. We also have the set $\mathcal{T}^1$ of first-order simple types: $\mathcal{T}^1$ is identified with $\mathcal{A}^0 \times \mathcal{T}^0$, by the pair of maps: $(dom, codom)$. Note that $\mathcal{T}^1$ contains $\mathcal{T}^0$ embedded by $t \to (0, t)$. We then have the set $\mathcal{A}^1$ of 1-*arities*, which is as above the set of applications with finite support from $\mathcal{T}^1$ to $\mathbf{N}$. Note that $\mathcal{A}^1$ contains $\mathcal{A}^0$: zeroth-order arities may be extended by zero to become first-order arities.

For a a 0-arity, t an atomic type, and b a 1-arity, we use the notations T^a , T^t_a and T^b_a in the same way as above. Note that T^b_a decomposes in a single way as a product of simple first-order types of the form $T^{t'}_{a'}$.

Now the set $\mathcal{C}$ is equipped with the two maps dom, from $\mathcal{C}$ to $\mathcal{A}^1$, and $codom$, from $\mathcal{C}$ to $\mathcal{T}^0$, and the *second-order constructors* are defined in the following way: il C_c has domain d and codomain k, and a is a 0-arity, then the second-order constructor C^a_c is the map from T^d_a to T^k_a defined by composition with C_c.

Finally, we have the set of projections $\mathcal{P}^1(S) = \mathcal{P}^1$: this one too is equipped with the two maps dom, from $\mathcal{P}^1$ to $\mathcal{A}^0$, and $codom$, from $\mathcal{P}^1$ to $\mathcal{T}^0$, and we denote by $\mathcal{C}_a$ (resp. $\mathcal{P}_a$) (resp. $\mathcal{C}_{a,t}$ (resp. $\mathcal{P}_{a,t}$)) the subset of $\mathcal{C}$ (resp. $\mathcal{P}$) of elements with domain a (resp. with domain a and codomain t).

3.2 Induction Principles

For our SOAS S, we have at least two induction principles, one for closed and one for functional terms.

Proposition 15. (closed SO induction principle) *Let B a collection indexed by $\mathcal{T}^0$ of sets B_t, B_t being a subset of t.*

We suppose that for each constructor C_c the following holds: writing a for $dom(c)$ and t for $codom(c)$, for any collection $f = (f_{\theta,k})_{k \in \underline{a}(\theta)}$, with $f_{\theta,k}) \in T_\theta$, satisfying, for each (θ, k) with $k \in \underline{a}(\theta)$, writing b for $dom(\theta)$ and t' for $codom(\theta)$, for any collection $x = (x_{s,m})_{m \in \underline{b}(s)}$ in $\Pi_{m \in \underline{b}(s)} B_s$, $f_{\theta,k}(x)$ is in $B_{t'}$. Then, for each $t \in \mathcal{T}^0$, B_t is the whole T_t.

Proposition 16. (open SO induction principle) *Let B a collection indexed by $\mathcal{T}^1$ of sets B_θ, B_θ being a subset of θ.*

We suppose that

(i) B contains the projections, more precisely if a projection $p \in \mathcal{P}$ is in T_θ, it is in B_θ.

(ii) B is stable by all the second-order constructors, more precisely if C^a_t is a second-order constructor sending T^d_a to T^k_a and if $T^d_a = \Pi_i T_{\theta_i}$, then, for any family $f = (f_i)_i$ with $f_i \in B_{\theta_i}$, $C^a_c(f)$ is in B^a_k.

Then, for each $\theta \in \mathcal{T}^1$, B_θ is the whole T_θ.

The open induction principle is stronger than the closed one in this sense that any property of closed terms which can be proved by the latter can be extended to be true by definition on terms and proved by the former. For the other way around, there is no such a trivial reduction. One way to connect properties of open terms to the domain of the closed induction principle is to prove that they are **extensional**: the property P is extensional if for any term x of type $\Pi A_i \to A$, with atomic A_i's, $P(x)$ holds if and only if, for any collection (x_i) of closed terms of types (A_i) satisfying the collection of conditions $(P(x_i))$, $P(x((x_i)))$ holds.

3.3 Separation Principle

Proposition 17. (first-order separation principle): *Let A_1 and A_2 be two (first-order) terms of our SOAS such that $A_j = C_j \prod_{i=1}^{n_j} x_{i,j}$, where the C_j are* **second-order** *constructors and the $x_{i,j}$ are (first-order) terms. Then $A_1 = A_2$ iff $C_1 = C_2$, $n_1 = n_2$, and $\forall i \in [1, n_1], x_{i,1} = x_{i,2}$ (This concerns in particular terminal constructors, for which n_j is zero). Furthermore, such a term with a head constructor cannot be equal to a projection; and two projections p_θ and $p_{\theta'}$ are equal if and only if the two triples θ and θ' are equal.*

3.4 Recursion

Proposition 18. (first-order SO recursion principle) *Given*

- a collection B, indexed by $\mathcal{A}^0 \times \mathcal{T}^0$ of sets B^t_a;

- a collection M, indexed by the set $\mathcal{A}^0 \times \mathcal{C}$ of second-order constructors, of maps M^a_c such that if C^a_c sends $T^d_a = \Pi_i T^{k_i}_{a_i}$ into T^k_a, then M^a_c sends $\Pi_i T^{k_i}_{a_i} \times \Pi_i B^{k_i}_{a_i}$ into B^k_a;

- a collection M', indexed by $\mathcal{P}$ of elements M'_p, with $M'_p \in B^{codom(p)}_{dom(p)}$;

there exists a unique collection $r := rec(B, M, M')$ *of maps* r_a^t *indexed by* $\mathcal{A}^0 \times \mathcal{T}^0$, r_a^t *mapping* T_a^t *into* B_a^t, *satisfying the two collections of reduction rules:*

for each C_c^a *sending* $T_a^d = \Pi_i T_{a_i}^{k_i}$ *into* T_a^k,
$r_a^k(C_c^a((x_i)_i)) = M_c^a((x_i)_i, (r_{a_i}^{k_i}(x_i))_i)$, *and*
$r_{dom(p)}^{codom(p)}(p) = M'_p$.

Remark This general principle has the weak feature that the arguments of the meta-level construction rec have a quite complicated nature (type). Here are some tricks inspired by [3] for simplifying this nature. First we choose the collection of sets B_a^t in the following way: we first choose a collection D_t indexed by $\mathcal{T}^0$, and we set $B_a^t = T^a \times D^a \Rightarrow D_t$ where D^a stands for $\Pi_t D_t^{a(t)}$. The presence of D^a in this formula offers a possibility to settle once and for all a (default) value for the images of projections:

$$r_{dom(p)}^{codom(p)}(p) = p,$$

where the right hand side refers to the projection corresponding to p from D^a to D_t. The presence of T^a in this formula offers similarly a possibility of giving a (default) formula for M_c^a in terms of a substitute m_c^a of type $\Pi_i T_{k_i} \Pi_i B_{a'_i}^{k_i}$ into B_k, where we have written $a_i = a + a'_i$, so that the type of C_c is $\Pi_i T_{a'_i}^{k_i} \rightarrow T_k$. Accordingly, the type of M_c^a may be written $\Pi_i T_{a_i}^{k_i} \times \Pi_i T^{a_i} \Pi_i B^a \rightarrow B_{a'_i}^{k_i} \Rightarrow B_a^k$ and the default formula for M is

$M_c^a((f_i)_i, (x_i)_i, (g_i)_i) = [\lambda z : B^a] m_c^a((f_i(x_i))_i, (g_i(z))_i)$.
Each of these two tricks may be used separately, with
$B_a^t = D^a \Rightarrow D_t$
or
$B_a^t = T^a \Rightarrow D_t$,
so that all together we have three variants here.

3.5 Unification

Proposition 19. (unification principle) *If two terms unify, a most general unifier for these two terms exists.*

Remark This is not the usual first-order unification principle but a slight variant. There is a corresponding algorithm.

4 General Case

4.1 Setting

Thus, we consider a general HOAS S, with its set $\mathcal{T}^0(S) = \mathcal{T}^0$ of atomic types, and its set $\mathcal{C}(S) = \mathcal{C}$ of constructors. We also have its set $\mathcal{T}$ of simple types, and its set of arities $\mathcal{A}$ which is the set of applications with finite support from $\mathcal{T}$ to

$\mathbf{N}$, and $\mathcal{T}$ is identified with $\mathcal{A} \times \mathcal{T}^0$, by the pair of maps: $(dom, codom)$. Note that $\mathcal{T}$ contains $\mathcal{T}^0$, t being embedded as $(0, t)$. For a an arity, t an atomic type, and b a 1-arity, we use the notations T^a , T^t_a and T^b_a in the same way as above. Note that T^b_a decomposes in a single way as a product of simple first-order types of the form $T^{t'}_{a'}$.

Now the set $\mathcal{C}$ is equipped with the two maps dom, from $\mathcal{C}$ to $\mathcal{A}$, and $codom$, from $\mathcal{C}$ to $\mathcal{T}^0$, and the *higher-order constructors* are defined in the following way: il C_c has domain d and codomain k, and a is an arity, then the higher-order constructor C^a_c is the (meta-level) map from T^d_a to T^k_a defined by composition with C_c.

Finally, we have the set of higher-order projections $\mathcal{P}(S) = \mathcal{P}$: this one too is equipped with the two maps dom, from $\mathcal{P}$ to $\mathcal{A}$, and $vcodom$, from $\mathcal{P}$ to $\mathcal{T}$ (in $vcodom$, v stands for virtual). We denote by $\mathcal{C}_a$ (resp. $\mathcal{P}_a$) (resp. $\mathcal{C}_{a,\theta}$ (resp. $\mathcal{P}_{a,\theta}$)) the subset of $\mathcal{C}$ (resp. $\mathcal{P}$) of elements with domain a (resp. with domain a and codomain θ). The subset $\mathcal{P}_{a,\theta}$ is naturally identified with $\underline{a}(\theta)$. Thus, for $k \in \underline{a}(\theta)$, we have a higher-order projection $p^{\theta,k}_a$ and we now explain what it is:

We first explain what it is not, namely the natural naive projection in T^θ_a, which we denote by ${p'}^{\theta,k}_a$. Indeed, we prefer the following kind of "long η form". We write $\theta = T^t_b$ and we set

$p^{\theta,k}_a = [\Lambda z : T^b_a][\lambda x : T^a]{p'}^{\theta,k}_a(x)(z(x))$. Here Λ refers to the meta-level abstraction, while λ refers to the abstraction in our HOAS. So this ho-projection is a meta-level map sending T^b_a into T^t_a.

4.2 Induction principle

For a general HOAS S, there is a closed induction principle but its formulation is intricated and we turn directly to the following:

Proposition 20. (open HO induction principle) *Let B a collection indexed by $\mathcal{T}$ of sets B^θ, B^θ being a subset of T^θ. If a is an arity and θ a simple type, we set $B^\theta_a = B^{\theta'}$, where $\theta' = a \to \theta$. If a and b are arities, we define B^b_a as $\Pi_\theta (B^\theta_a)^{b(\theta)}$. We suppose that*

(i) B is stable by the ho-projections, more precisely if $p^{\theta,k}_a$ sends T^b_a into T^t_a, then it also sends B^b_a into B^t_a.

(ii) B is stable by the ho-constructors, more precisely if C^a_t is a ho-constructor sending T^a_d to T^a_k,

then it also sends B^d_a into B^k_a.

4.3 Separation Principle

Proposition 21. (higher-order separation principle): *Let A_1 and A_2 be two terms of our HOAS such that $A_j = C_j \prod_{i=1}^{n_j} x_{i,j}$, where the C_j are ho-constructors and the $x_{i,j}$ are terms. Then $A_1 = A_2$ iff $C_1 = C_2$, $n_1 = n_2$, and $\forall i \in [1, n_1], x_{i,1} = x_{i,2}$ (This concerns in particular terminal constructors, for which n_j is zero). Furthermore, such a term with a head constructor cannot be* *equal to an image by a ho-projection; and two such images* $p_a^{\theta,k}(x_1,\ldots,x_n)$ *and* $p_{a'}^{\theta',k'}(x'_1,\ldots,x'_{n'})$ *are equal if and only if* $a = a'$, $\theta = \theta'$ *and* $k = k'$ *(so that* $n = n'$*), and* $x_1 = x'_1 \ldots x_n = x'_n$.

4.4 Recursion.

Proposition 22. (HO recursion principle) *Given*

- a collection B*, indexed by* $\mathcal{T} = \mathcal{A} \times \mathcal{T}^0$ *of sets* B_a^t*;*

- a collection M*, indexed by the set* $\mathcal{A} \times \mathcal{C}$ *of ho-constructors, of maps* M_c^a *such that if* C_c^a *maps* $T_a^d = \Pi_i T_{a_i}^{k_i}$ *into* T_a^k*, then* M_c^a *maps* $\Pi_i T_{a_i}^{k_i} \times \Pi_i B_{a_i}^{k_i}$ *into* B_a^k*;*

- a collection M'*, indexed by* $\mathcal{P}$ *of maps* M'_p*: if* $p_a^{\theta,k}$ *maps* $T_a^b = \Pi_i T_{a_i}^{k_i}$ *into* T_a^t*, then* M'_p *maps* $T_a^b \times B_a^b$ *into* B_a^t*, where* B_a^b *stands for* $\Pi_i B_{a_i}^{k_i}$*,*

there exists a unique collection $r := rec(B, M, M')$ *of maps* r_t^a *indexed by* $\mathcal{A} \times \mathcal{T}^0$, r_a^t *mapping* T_a^t *into* B_a^t*, satisfying the two collections of reduction rules: for each* C_a^c*, mapping* $T_a^d = \Pi_i T_{a_i}^{k_i}$ *into* T_a^k*,*

$$r_a^k(C_c^a((x_i)_i)) = M_c^a((x_i)_i, (r_{a_i}^{k_i}(x_i))_i)$$

and, for each ho-projection $p = p_a^{\theta,k}$*, mapping* $T_a^d = \Pi_i T_{a_i}^{k_i}$ *into* T_a^k*,*

$$r_a^k(p((x_i)_i)) = M'_p((x_i)_i, (r_{a_i}^{k_i}(x_i))_i).$$

4.5 Unification

There is no most general unifier in general. However there are some weaker solution for this problem.

5 Editing Higher-Order Terms

Up to now, higher-order abstract syntax has been used only to specify the semantics of programming languages, in particular for functional ones. The goal of this section is to describe how it can be used for very practical applications, as syntactical editing.

The goal of syntactical editing is to guarantee at any time that a program that is edited is syntactically correct. The abstract syntax definition can be used to generate a menu driven mode that will help the user to construct and modify programs. In Centaur [7], a first-order abstract syntax and a notion of meta-variable are used to achieve this goal. At any place in an abstract tree the user can be prompted with a list of operators or transformation rules of the correct type. This is completely independent of the context and can be done completely locally. The limit of this technique is reached when one wants to edit a leaf of a program, for example an identifier. One would like, for example to be prompted with the list of objects that are accessible, taking into account the scope rules of the language. But this is not possible since with first-order abstract syntax the notion of scope is part of the semantic specification.

Integrating the notion of binder in the syntax of a programming language can improve this aspect of editing. Our goal in this section is to describe the basic actions of editing in the context of an higher-order abstract syntax. To fix ideas, we will use the previous Exp example, and for simplicity we will forget in the following the type of the arguments (which will be always *Exp*). In the following example, some pieces of the text are in boldface to enhance our notion of selection.

Selection: Giving a term and a legal path we can select a sub-term of this term. Note that the arity of the selected subterm will be greater than the arity of the source term if we cross some binders along the given path.

select(λ**a.apply(lambda(λx.x),a)**, []) = λ**a.apply(lambda(λx.x),a)**
select(λ**a**.apply(**lambda(λx.x)**,a), 1) = λ**a.lambda(λx.x)**
select(λ**a**.apply(lambda(λ**x.x**),a), 1.1) = λ**a,x.x**
select(λ**a**.apply(lambda(λx.x),**a**), 2) = λ**a.a**

Arity Modification: Higher-order terms are functional. Type coercion consists in seeing one particular term that is a function of n arguments as a function of one more or one less argument. In the last case, this can be done only for functions that do no depend of one particular argument (i.e. that argument is not referenced (used) in the term). We have to give an index to designate the rank of the new (resp. old) argument.

new(λa.apply(lambdaλx.x,a),2) = λa,**b**.apply(lambdaλx.x,a)
del(λa,**b**.apply(lambdaλx.x,a),2) = λa.apply(lambdaλx.x,a)
del(λ**a**,b.apply(lambdaλx.x,**a**),1) is an error

Cut: The cut operation takes two arguments: the initial term and a path to one of its sub-terms; it returns two results: the first one is the initial term in which the designated subtree has been replaced by a place-holder (or meta-variable) that is in fact a new argument, the second one is equal to the designated subtree. In the following, meta-variables (i.e. arguments that are generated by the system, not by the user) are denoted by capitals.

cut(λ**a**.apply(lambda(λ**x.x**),a), 1.1) =
  (λ**a,U**.apply(lambda(λx.**U**),a), λ**a,x.x**)

Application: This operation is very natural. One can think of a variant of application in which one give the rank of the variable to which one argument must be applied. The traditional paste operation is an application. The traditional *change* operation is the composition of a cut and an application as explained in the section on menus.

app(λa,**b**.apply(lambda(λx.**b**),a), 2, λa,b.b) =
  λa,b,c.apply(lambda(λx.c,a))

Menus: As it is traditionally done in first order syntactical editing, it is natural to propose the list of (higher-order) operators that can be put at a certain place (selection). This list is extended by the so called projections. Among these projections one can find the arguments of the term when it is functional, and also those that have been introduced by binders on the path going from the root to the current location, allowing, for exemple the use of declared objects.

menu(λ**a.**apply(**lambda(**λ**x.**x),a), 1) =
  (**a**, λU,V.**apply**(U,V), λU.**lambda**(λy.U))
menu(λ**a.**apply(lambda(λ**x.**x),a), 1.1) =
  (**a**, **x**, λU,V.**apply**(U,V), λU.**lambda**(λy.U))

When a projection proposed by a menu is choosen, the current tree is replaced by the corresponding argument. In this way, it is not possible to use a terminal object that has not been introduced before.

choose(λ**a.**apply(**lambda(**λ**x.x**),a), 1, 1) = λ**a.**apply(**a**,a)
choose(λ**a.**apply(lambda(λ**x.**x),a), 1.1, 1) = λ**a.**apply(lambda(λ**x.a**),a)

When an operator proposed by a menu is selected, two consecutive operation are performed: first the current tree is cut, generating a temporary meta-variable, second the whole tree is applied to the selected (functional) operator using the rank of this temporary meta-variable. Depending on the arity of the operator, some new meta-variables will appear.

choose(λ**a.**apply(lambda(λ**x.**x),a), 1.1, 4) =
  λ**a,U.**apply(lambda(λ**x.lambda(**λ**y.U**),a)

All the previous operations operate modulo α-conversion. A complete editor should allow to visualize arguments by user-defined names, checking for eventual captures.

6 Conclusion

We have better understood what is a higher-order abstract syntax, and we have formulated its main features in three principles, making clear the need for an implementation of higher-order abstract syntax in a logical framework where these principles could be available at the object-level. Our next goals now will be to implement a prototype of a higher-order syntactical editor and to use functional abstract syntax for semantics.

Thanks are due to Joëlle Despeyroux and Pierre Leleu for discussions and reading of the previous version of this paper.

References

1. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of the 3rd Symp. on Software Development Environments*, Boston, USA, November 1988. Rapport de Recherche INRIA 777, Inria-Sophia-Antipolis, France, December 1987.
2. R. D. Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhaüser, ISBN 0-8176-3763-X, 1995.
3. J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In *To appear in the Proceedings of the TLCA 97 Int. Conference on Typed Lambda Calculi and Applications, Nancy, France, April 2–4*. Springer-Verlag LNCS, April 1997. An extended version is available as CMU Technical Report CMU-CS-96-172.
4. J. A. Goguen and J. Meseguer. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. In *Symposium on Logic in Computer Science, IEEE Press*, 1987.
5. J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance , overloading, exceptions and partial operations. *Technical Monograph PRG-80, Programming Research Group, Oxford University Computing Laboratory*, December 1989.
6. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Journal of the ACM, 40(1):143-184*, January 1993.
7. I. Jacobs. The centaur 2.0 manual. Technical report, Inria, Sophia-Antipolis, France, 1994.
8. J.-P. J. Joseph A. Goguen and J. Meseguer. Operational semantics of order-sorted algebra. In *International Conference on Automata, Languages and Programming, W. Brauer, Ed. Springer Verlag, LNCS 194.*, 1985.
9. S. M. Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer Verlag, 1971.
10. G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, Aug. 1988. MIT Press.
11. G. Nadathur and F. Pfenning. The type system of a higher-order logic programming language. In F. Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.
12. F. Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.
13. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.
14. B. C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. MIT Press, 1991.
15. D. Terrasse. Encoding natural semantics in coq. In *Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST '95)*. Springer-Verlag LNCS, July 1995.