HIGHER-ORDER ABSTRACT SYNTAX IN TYPE THEORY

VENANZIO CAPRETTA AND AMY P. FELTY

Abstract. We develop a general tool to formalize and reason about languages expressed using higher-order abstract syntax in a proof-tool based on type theory (Coq). A language is specified by its signature, which consists of sets of sort and operation names and typing rules. These rules prescribe the sorts and bindings of each operation. An algebra of terms is associated to a signature, using de Bruijn notation. Then a higher-order notation is built on top of the de Bruijn level, so that the user can work with meta-variables instead of de Bruijn indices. We also provide recursion and induction principles formulated directly on the higher-order syntax. This generalizes work on the Hybrid approach to higher-order syntax in Isabelle and our earlier work on a constructive extension to Hybrid formalized in Coq. In particular, a large class of theorems that must be repeated for each object language in Hybrid is done once in the present work and can be applied directly to each object language.

§1. Introduction. We aim to use proof assistants (in our specific case Coq [9, 6]) to formally represent and reason about languages using higher-order syntax, i.e. object languages where binding operators are expressed using binding at the meta-level. This is an active and fertile field of research. Several methods contend to become the most elegant, efficient, and easy to use. The differences stem from the approach of the researchers and the characteristics of the proof tool used.

Our starting point was the work on the Hybrid tool in Isabelle/HOL by Ambler, Crole, and Momigliano [2]. We began by replicating their development step by step in Coq, but soon realized that the different underlying meta-theory (the Calculus of Inductive Constructions [10, 38], as opposed to higher-order logic) provided us with different tools and led us to diverge from a simple translation of their work. The final result [8] exploits the computational content of the Coq logic: we prove a recursion principle that can be used to program functions on the object language. These functions can be directly computed inside Coq. In the present work, we extend it and provide a general tool in which the user can easily define a language by giving its signature, and a set of tools (higher-order notation, recursion and induction principles) are automatically available.

Let us introduce the problem of representing languages with bindings in a logical framework. While a first-order language contains operation symbols that take just elements of the domain(s) as arguments, a second-order language may have operations whose input consists of functions of arbitrary complexity. From a syntactic point of view these operations bind some of the free variables in their arguments. The simplest example of this phenomenon is the abstraction operation in λ-calculus. We can see the operation λ, syntactically, as a binder;

its use is: $\lambda x.t$, where we indicate that the free occurrences of the variable x in t are now bound. Alternatively, we can see it as taking a function as input; its use is: $\lambda(f)$, where f maps λ-terms to λ-terms.

Both outlooks have their strong and weak points. First-order approaches are unproblematic from the logical point of view and most proof tools allow their formalization by direct inductive definitions. However, a series of problems arise. Terms are purely syntactical objects and therefore are considered equal only when they are syntactically the same; while terms that differ only by the name of bound variables should be considered identical (α-conversion). Substitution becomes problematic because binders may capture free variables, so we need to use some renaming mechanism. These approaches implement object languages by a first-order encoding with some bookkeeping mechanism to keep track of bound variables and avoid problems of variable capture and renaming. In general, many definitions and lemmas must be formalized for each implemented object language.

The higher-order approach has the advantage that the implementation of the meta-theory can be reused: In implementing the proof tool, the developers were already faced with the decision of how to represent functions and operators on functions. Higher-Order Abstract Syntax (HOAS) aims at reusing this implementation work by seeing arguments as functions at the meta-level, thus delegating the issues of α-conversion, substitution, and renaming to the logical framework. The main disadvantage is that higher-order data-types are simply not allowed in most systems. For example, a higher-order representation of the λ-calculus would be an inductive data-type with two constructors:

$$\begin{array}{l}\textbf{Inductive } \Lambda := \\ \quad \mathsf{abs} : (\underline{\Lambda} \to \Lambda) \to \Lambda \\ \quad \mathsf{app} : \Lambda \to \Lambda \to \Lambda\end{array}$$

But the constructor for abstraction abs is not allowed in most systems, since it contains a negative occurrence (underlined) of the defined type. Changing the meta-theory to permit such definitions would just make the logic inconsistent. Some constraint must be imposed on negative occurrences to avoid the kind of circularity that results in inconsistency of the logical framework. Several ways of doing this are documented in the literature.

Another drawback of HOAS is the appearance of *exotic terms.* A term is called exotic if it results from the application of a binder to a function that is not uniform in its argument. For example $\lambda(X \mapsto F^{|X|}\, X)$, where F is any term and $|\_|$ is the length function on terms, is exotic: its body does not itself represent a term, but only reduces to a term for every concrete input. Exotic terms do not exist in the informal treatment of object languages and should be precluded.

The Hybrid strategy (which we appropriated) is a combination of the two approaches: It uses an internal first-order syntactic representation (de Bruijn syntax [12]) combined with a higher-order user interface.

The original elements of the present work with respect to [2] and [8] are:

- *Definition of a type of signatures for Universal Algebra with binding operations:* The general shape of formalizations of various case studies in [2] and

[8] was informally explained, but the low level work had to be repeated for each new formal system. Our higher-order signatures generalize Plotkin's *binding signatures* [29, 15]. To our knowledge, there is no other formalization of this notion.

- *Automatic definition of the families of types of terms from every signature:* The user need not define explicitly the types of formal terms of the object language anymore. It is enough to specify the construction rules for terms and the higher-order syntax is automatically generated.
- *Generation of typed terms:* In [2] and [8] terms were generated in an untyped way, any term could be applied to any other term. The set of correct terms was defined by using a well-formedness predicate. This meant that proofs needed to explicitly manipulate proofs of well-formedness. In the present work, terms are well-formed from the start, thanks to the use of an inductive family that directly implements the typing rules given in the signature.
- *Generation of the recursion principle and proof of its correctness for every signature:* In our previous work we formalize recursion principles for specific object languages. Now we have a parameterized version of it; it can be used to define computable functions on the higher-order syntax of any signature.
- *General proof of the induction principle for all signatures:* This gives appropriate reasoning principles for every object language, derived automatically by the system from the specification of the signature. In our previous work, this principle had to be derived separately for every object language. Neither the recursion nor the induction principle were proved in the original Hybrid system, neither for the general case nor for specific case studies.
- *Higher-order recursive calls:* In the recursion principle, the recursive call is a function on all possible results on the bound variables, rather that just the result on the body of the abstraction. This means, for example, that in defining a recursive function f on the typed λ-calculus, in the abstraction case $f\,(\lambda x.b)$ the recursive call isn't just the value of f on b, but a function mapping all possible results for the bound variable x to the corresponding value for b.

In Section 2 we present our tool from the user point of view. We describe how to define an object language by giving its signature and how to obtain higher-order notation, recursion and induction principles on it. In Section 3 we describe the implementation of our method in Coq and explain the main ideas and problems involved. In Section 4, we discuss related work, and in Section 5, after a review of our work, we state our goals for future research.

Prerequisites: We assume that the reader is familiar with Dependent Type Theory and (Co)Inductive types. Good introductions are Barendregt [5] and the books by Luo [21] and Sørensen and Urzyczyn [34]. An introduction to inductive and coinductive types, as used here, is Part I of the first author's PhD thesis [7].

More specifically, we formalized everything in the proof assistant Coq. A good introduction to this system is the book by Bertot and Castéran [6]. A complete formal description is in the Coq manual [9]. The Coq files of our development are available at: `http://www.cs.ru.nl/~venanzio/Coq/HOUA.html`.

In this paper we avoid giving specific Coq code. Instead we have tried to adopt notation and terminology accessible to people familiar with other versions of dependent type theory and (co)inductive definitions. In addition, we have tried to make our treatment comprehensible to users of other systems, like Lego [1], HOL [19], and Isabelle [24].

In this paper, Prop is the type of logical propositions, whereas Set is the type of data types. We use the standard logical connectives for formulas in Prop. We write $\{A\} + \{B\}$ for Coq's constructive disjunction (in Set). We write $:=$ for definitional equality.

We need several notations for lists and tuples. The standard type of lists of elements of set A is $\mathsf{list}\, A$ and a typical element of it is $[a_1, \ldots, a_n]$. A Cartesian product $A_1 \times \cdots \times A_n$ has as elements n-tuples of the form $\langle t_1, \ldots, t_n \rangle$. Beside these standard types, we need to use dependent lists, in which every element has a different type belonging to a class of types called *argument types:* the type of such a lists is denoted by $\{\!|A_1, \ldots, A_n|\!\}$, where the A_is belongs to the class of argument types, and its elements are n-tuples of the form $\{a_1, \ldots, a_n\}$. Braces in conjunction with :: denote the cons operator, e.g., if $\vec{a} = \{a_1, \ldots, a_n\}$, then $\{a_0 :: \vec{a}\} = \{a_0, a_1, \ldots, a_n\}$. Finally, we have a different notation $\langle\!\langle t_1, \ldots, t_n \rangle\!\rangle$ for list of *terms with bindings.* These types of lists are defined rigorously at the moment of their introduction in Section 3. The reader who is not interested in their specific implementation may think of them as being equivalent to Cartesian products.

The notation $(\mathsf{fun}\, x \mapsto b)$ denotes λ-abstraction: variable x is bound in body b. Abstraction over an n-tuple is often written $(\mathsf{fun}\, \langle x_1, \ldots, x_n \rangle \mapsto b)$. The keyword **Inductive** introduces an inductive type; the keyword **Record** introduces a record type and is syntactic sugar for a dependent tuple type. We freely use infix notations, without explicit declarations. An underscore _ denotes an implicit type parameter that can be inferred from context. Other notations are described as they are introduced.

§2. Higher-order universal algebra. We describe our Coq development of Universal Algebra with bindings from a user point of view. In the next section we give the details of the implementation. We use the simply typed λ-calculus as a running example. We use Coq as a logical framework. An object language is specified by giving its signature, which is a tuple consisting of a set of names for sorts, a set of names for operations, and an assignment of a typing rule to every operation; in addition, we require equality on the sorts to be decidable. This notion is a multisorted extension of Plotkin's binding signatures [29, 15]. Our tool automatically generates the family of types of terms for the object language and provides programming and reasoning support for it.

Before describing the general development, let us illustrate how a user defines the simply typed λ-calculus in our tool. As stated, the user must give a signature, that is, a 4-tuple:

$$\mathsf{sig}_\lambda : \mathsf{Signature} := \mathsf{signature}\, \mathsf{type}_\lambda\, \mathsf{dec}_\lambda\, \mathsf{operation}_\lambda\, \mathsf{rule}_\lambda.$$

Below we explain the meaning of the components of the signature sig_λ.

First we must give the set type_λ of sorts, that are, in our case, codes for simple types: they are generated from the base type o_λ by applying the arrow constructor, if A and B are codes for types, then $(\mathsf{arrow}_\lambda\, A\, B)$ is also a code for a type. We call attention to the distinction between Coq types, like Signature and type_λ themselves, that represent sets in the metalanguage, and elements of type_λ, like o_λ and $(\mathsf{arrow}_\lambda\, \mathsf{o}_\lambda\, \mathsf{o}_\lambda)$, that are codes for types of the object language.

Then we define the set $\mathsf{operation}_\lambda$ of names of operations: abstraction and application are parameterized over types, so there are operations names $(\mathsf{abs}_\lambda\, A\, B)$ and $(\mathsf{app}_\lambda\, A\, B)$ for every pair of type codes A and B.

In type theory, the sets of codes for types and names of operations are defined by the following inductive declarations:

Inductive type_λ : Set :=
  $\mathsf{o}_\lambda : \mathsf{type}_\lambda$
  $\mathsf{arrow}_\lambda : \mathsf{type}_\lambda \to \mathsf{type}_\lambda \to \mathsf{type}_\lambda$

Inductive $\mathsf{operation}_\lambda$: Set :=
  $\mathsf{abs}_\lambda : \mathsf{type}_\lambda \to \mathsf{type}_\lambda \to \mathsf{operation}_\lambda$
  $\mathsf{app}_\lambda : \mathsf{type}_\lambda \to \mathsf{type}_\lambda \to \mathsf{operation}_\lambda$.

It is necessary, in order to prove some of the results of the formal language, that the set of sorts, type_λ in the example, has a decidable equality. This requirement is expressed in type theory by:

$$\mathsf{dec}_\lambda : \forall t_1, t_2 : \mathsf{type}_\lambda, \{t_1 = t_2\} + \{t_1 \neq t_2\}.$$

Precisely, dec_λ is a computable function that maps every pair of codes for types t_1 and t_2 to either a proof that they are equal or a proof that they are distinct.

The last component of the signature is a specification of the typing rules for the operations. Every sort code A will be associated to a set of terms $\mathsf{Term}\, A$ and every operation name f will be associated to a (possibly higher-order) function $\mathsf{Opr}\, f$ on terms. Every operation name is associated to a rule specifying the sorts and bindings of its arguments and the sort of its result. In our case, our goal is to obtain the following typing rules for abstraction and application:

$$\frac{\begin{array}{c}[x : \mathsf{Term}\, A] \\ \vdots \\ b : \mathsf{Term}\, B\end{array}}{\mathsf{Opr}\,(\mathsf{abs}_\lambda\, A\, B)\,\{\mathsf{fun}\, x \mapsto b\} : \mathsf{Term}\,(\mathsf{arrow}_\lambda\, A\, B)};$$

$$\frac{f : \mathsf{Term}\,(\mathsf{arrow}_\lambda\, A\, B) \quad a : \mathsf{Term}\, A}{\mathsf{Opr}\,(\mathsf{app}_\lambda\, A\, B)\,\{f, a\} : \mathsf{Term}\, B}.$$

Technically, rules are themselves formal objects, that is, they are codes specifying the intended use of the operation names. We define the exact form of rules later; here we just show how the user specifies the two previous informal rules for the operations of the simply typed λ-calculus:

$$\begin{array}{l}\mathsf{rule}_\lambda : \mathsf{operation}_\lambda \to \mathsf{operation\_type}\,\mathsf{type}_\lambda \\ \mathsf{rule}_\lambda\,(\mathsf{abs}_\lambda\, A\, B) = [[A] \vdash B] \mathbin{/\!/} (\mathsf{arrow}_\lambda\, A\, B); \\ \mathsf{rule}_\lambda\,(\mathsf{app}_\lambda\, A\, B) = [[] \vdash (\mathsf{arrow}_\lambda\, A\, B), [] \vdash A] \mathbin{/\!/} B.\end{array}$$

Informally, rule_λ states that: $(\mathsf{abs}_\lambda\, A\, B)$ is the name of an operation taking as argument a term of type B, binding a variable of type A, giving a result of type $(\mathsf{arrow}_\lambda\, A\, B)$; $(\mathsf{app}_\lambda\, A\, B)$ is the name of an operation taking two arguments of type $(\mathsf{arrow}_\lambda\, A\, B)$ and A, without any binding, giving a result of type B.

Thus, signatures are defined as records:

$$\begin{array}{l}\textbf{Record}\ \mathsf{Signature} := \mathsf{signature}\\ \quad\{\ \mathsf{sig\_sort} : \mathsf{Set};\\ \quad\ \mathsf{sig\_dec} : \forall s_1, s_2 : \mathsf{sig\_sort}, \{s_1 = s_2\} + \{s_1 \neq s_2\};\\ \quad\ \mathsf{sig\_operation} : \mathsf{Set};\\ \quad\ \mathsf{sig\_rule} : \mathsf{sig\_operation} \to \mathsf{operation\_type}\,\mathsf{sig\_sort}\ \}\end{array}$$

Let a signature $\sigma : \mathsf{Signature}$ be fixed from now on, and take $\mathsf{sort} := \mathsf{sig\_sort}\,\sigma$, $\mathsf{operation} := \mathsf{sig\_operation}\,\sigma$, and $\mathsf{rule} := \mathsf{sig\_rule}\,\sigma$. The function rule maps every operation f to an element of $(\mathsf{operation\_type}\,\mathsf{sort})$, whose general form is:

$$(\mathsf{rule}\,f) = [[A_{11}, \ldots, A_{1k_1}] \vdash B_1, \ldots, [A_{n1}, \ldots, A_{nk_n}] \vdash B_n] /\!/ C$$

where the A_{ij}s, B_is, and C are sorts.

Let us explain informally the meaning of this rule. It is comprised of a list of argument specifications and a result sort; each argument has a list of bindings and a sort. This states that f is an operation that takes n arguments. The ith argument is a function from the sorts $A_{i1}, \ldots, A_{ik_i}$ to B_i; or, in more syntactical terms, it is a term of sort B_i with bindings of variables of sorts $A_{i1}, \ldots, A_{ik_i}$. Formally we define $(\mathsf{arg\_type}\,\mathsf{sort})$ as the set of argument types: its elements are pairs $\langle [A_1, \ldots, A_k], B\rangle$ consisting of a list of sorts (the types of bindings) and and sort (the type of the argument); we use the symbol $\vdash$ as a pairing operator. Then $(\mathsf{rule}\,f)$ is a pair $\langle [T_1, \ldots, T_n], C\rangle$ consisting of a list of argument types and a result sort; we use $/\!/$ as a pairing operator. We use the square bracket notation exclusively for lists of elements of sort and of $(\mathsf{arg\_type}\,\mathsf{sort})$, to distinguish them from other kinds of lists. The various components of the rule can be extracted by the following functions:

$$\begin{array}{l}\mathsf{op\_arguments}\,(\mathsf{rule}\,f) = [T_1, \ldots, T_n] : \mathsf{list}\,(\mathsf{arg\_type}\,\mathsf{sort})\\ \mathsf{op\_result}\,(\mathsf{rule}\,f) = C : \mathsf{sort}\\ \mathsf{arg\_bindings}\,T_i = [A_{i1}, \ldots, A_{ik_i}] : \mathsf{list}\,\mathsf{sort}\\ \mathsf{arg\_result}\,T_i = B_i : \mathsf{sort}.\end{array}$$

Therefore, the specification of f is equivalent to the following higher-order introduction rule for the type $(\mathsf{Term}\,C)$ of terms of sort C, where Opr is the higher-order application operator:

$$\frac{\begin{array}{ccc}[x_{1j} : \mathsf{Term}\,A_{1j}]_{j=1}^{k_1} & & [x_{nj} : \mathsf{Term}\,A_{nj}]_{j=1}^{k_n}\\ \vdots & \cdots & \vdots\\ b_1 : \mathsf{Term}\,B_1 & & b_n : \mathsf{Term}\,B_n\end{array}}{\mathsf{Opr}\,f\,\{\mathsf{fun}\,\langle x_{11}, \ldots, x_{1k_1}\rangle \mapsto b_1, \cdots, \mathsf{fun}\,\langle x_{n1}, \ldots, x_{nk_n}\rangle \mapsto b_n\} : \mathsf{Term}\,C}.$$

We can exploit Coq's type inference mechanism to define, straightforwardly, a compact notation that reflects the informal way of writing (Church style [5]) λ-terms. In the case of the simply typed λ-calculus, we defined the following notations:

informal	**formal compact**	**formal long**
$(\lambda x : A.b)$	$(\mathsf{Lambda}\,x\,\mathsf{in}\,A\,\mathsf{gives}\,b)$	$(\mathsf{Opr}\,(\mathsf{abs}_\lambda\,A\,B)\,(\mathsf{fun}\,x \mapsto b))$
$(f\,a)$	$(f\,\mathsf{of}\,a)$	$(\mathsf{Opr}\,(\mathsf{app}_\lambda\,A\,B)\,f\,a)$.

Once the user declares the signature σ, the system automatically generates, for each sort A : sort, the set of terms $\mathsf{Term}\,A$. The user-accessible notation for these terms is higher-order syntax, generated starting from free and bound variables using the binding operators of the form $\mathsf{Opr}\,f$ for f : $\mathsf{operation}$. The user never needs to see the internal de Bruijn syntax representation of the term. It will be explained in the next section.

There are two distinct sets of variables, free and bound. Each is indexed on the sorts and the natural numbers. Free variables, in the form v_i^A for A a sort and i a natural number, are treated as parameters whose interpretation is arbitrary but fixed. Bound variables, in the form x_j^A, are internally represented as de Bruijn indices, and their interpretation may change.

The argument functions $(\mathsf{fun}\,\langle x_{i1}, \ldots, x_{ik_i}\rangle \mapsto b_i)$ in the introduction rule for operations are not restricted. The user may exploit every construction allowed in the logical framework to define such functions, opening the door to exotic terms. We want to prevent this from happening: b_i should be uniform in the variables, that is, it should be a term constructed solely by the given rules, other operators and recursors from the logical framework should be banned. Two options are available. First, we could simply add to the rule for Opr some conditions requiring the uniformity of the arguments. This has the disadvantage that even in the case when the uniformity is trivial (always in practice) a proof must be provided, cluttering the syntax. Furthermore, in an intensional system like Coq, terms would depend on the proof of uniformity and we may have that two terms differing only by the proofs of uniformity cannot be proven equal. The second solution, which we adopt, is to perform an automatic normalization of the functions. As an example, let's consider the following non-uniform function allowable in our encoding of the simply typed λ-calculus:

$$\begin{aligned} &h : \mathsf{Term}\,A \to \mathsf{Term}\,B \\ &h\,x = \mathbf{if}\ x = \mathsf{x}_0^A\ \mathbf{then}\ \mathsf{v}_0^B\ \mathbf{else}\ \mathsf{v}_1^B. \end{aligned}$$

The function h maps the term x_0^A to v_0^B and all other terms to v_1^B. In a naive implementation of higher-order syntax, the abstraction $\lambda(h)$ would be an exotic term: Inside the formal language of the simply typed λ-calculus, it is not possible to make a case distinction on the syntactic form of a term in this way. In our implementation, the higher-order term $\mathsf{Opr}\,(\mathsf{abs}_\lambda\,A\,B)\,h$ performs an automatic uniformization of h by choosing a fresh variable, v_0^A will do in this case, and forcing h to behave on every argument in the same way as it behaves on the fresh variable:

$$h \rightsquigarrow \underline{h} = (\mathsf{fun}\,x \mapsto (h\,\mathsf{v}_0^A)[\mathsf{v}_0^A := x]) = (\mathsf{fun}\,x \mapsto \mathsf{v}_1^B[\mathsf{v}_0^A := x]) = (\mathsf{fun}\,x \mapsto \mathsf{v}_1^B).$$

It is important to keep in mind here that x is a *metavariable*, i.e. a Coq variable, of type $\mathsf{Term}\,A$, while x_0^A and v_0^A are object variables of sort A. The notation $[\mathsf{v} := x]$ expresses a defined substitution operation that replaces occurrences of free variable v with x; in its general form it allows simultaneous substitution of multiple variables. In our tool the function h is transformed into $\underline{h}$ behind the scene, so $\mathsf{Opr}\,(\mathsf{abs}_\lambda\,A\,B)\,h$ is actually equivalent to $\mathsf{Opr}\,(\mathsf{abs}_\lambda\,A\,B)\,\underline{h}$. Since $\underline{h}$ is uniform, the exotic term has disappeared.

Now for the general case: Given a function $h = \mathsf{fun}\,\langle x_{i1}, \ldots, x_{ik_i}\rangle \mapsto b_i$, we define its *uniformization* $\underline{h}$ by simply taking arbitrary fresh variables $\mathsf{v}_{i1}, \ldots, \mathsf{v}_{ik_i}$, applying h to them and then stating that $\underline{h}$ acts on every input in the same way that h acts on these variables. In symbols:

$$\underline{h}\,\langle x_{i1}, \ldots, x_{ik_i}\rangle = (h\,\langle \mathsf{v}_{i1}, \ldots, \mathsf{v}_{ik_i}\rangle)[\mathsf{v}_{i1} := x_{i1}, \ldots, \mathsf{v}_{ik_i} := x_{ik_i}].$$

Given a list of functions $\vec{h} = \{\mathsf{fun}\,\vec{x_1} \mapsto b_1, \cdots, \mathsf{fun}\,\vec{x_n} \mapsto b_n\}$, before applying $\mathsf{Opr}\,f$ to it, we normalize it to $\underline{\vec{h}}$. See Subsection 3.3 for a rigorous definition of uniformization.

The system automatically generates recursion and induction principles on the higher-order syntax. The (non-dependent) recursion principle also provides lemmas validating the recursion equations.

When defining a recursive function on the object language, we want to map each term to a result in a certain type. In general, it is possible that terms of different sorts are mapped to results with different types. Therefore the general type of a recursive function is $\forall A : \mathsf{sort}, \mathsf{Term}A \to F\,A$, where F is a family of sets indexed on sorts, $F : \mathsf{sort} \to \mathsf{Set}$.

Let us give a couple of examples on the simply typed λ-calculus.

As a first example, suppose that we want to define a size function on terms that counts the number of operations occurring in them. In this case every term is mapped to a natural number, so $F = \mathsf{fun}\,s \mapsto \mathbb{N}$ and $\mathsf{size} : \forall A : \mathsf{type}_\lambda, \mathsf{Term}A \to \mathbb{N}$.

For the second example, we consider the definition of a set-theoretic model of the λ-calculus. In this case, every sort (that is, every simple type) is mapped to a set and every term is mapped to an element of the corresponding set. Given a fixed set X to interpret the base type o_λ, we define a family of sets Model_X by recursion on type_λ:

$$\begin{aligned}&\mathsf{Model}_X : \mathsf{type}_\lambda \to \mathsf{Set}\\ &\mathsf{Model}_X\,\mathsf{o}_\lambda = X\\ &\mathsf{Model}_X\,(\mathsf{arrow}_\lambda\,A\,B) = (\mathsf{Model}_X\,A) \to (\mathsf{Model}_X\,B).\end{aligned}$$

Then an interpretation of the typed λ-calculus can be defined by recursion with $F = \mathsf{Model}_X$, as a map $\mathsf{interpret}_X : \forall A : \mathsf{sort}, \mathsf{Term}A \to \mathsf{Model}_X\,A$. To be precise, we need also an extra argument giving the assignment of a value in the model for all dangling variables. See the formal definition at the end of this section.

Before formulating the recursion principle, we need to introduce some notation. Given a family of sets indexed on the sorts, $F : \mathsf{sort} \to \mathsf{Set}$, we define a type of dependent lists of elements of F indexed on lists of sorts:

$$\mathsf{sort\_list}\,F\,[A_1, \ldots, A_k] \cong (F\,A_1) \times \cdots \times (F\,A_k).$$

We use $\cong$ to denote provable equivalence; we omit the exact definition of $\mathsf{sort\_list}$. An element of this type has the form $\langle a_1, \ldots, a_k\rangle$, with $a_i : F\,A_i$. Functions within the family F can be specified by an argument type. To an argument type $[A_1, \ldots, A_k] \vdash B$ we associate the type of functions from $(F\,A_1) \times \ldots \times (F\,A_k)$ to $F\,B$:

$$\begin{aligned}&\mathsf{arg\_map}\,F : \mathsf{arg\_type} \to \mathsf{Set}\\ &\mathsf{arg\_map}\,F\,T = \mathsf{sort\_list}\,F\,(\mathsf{arg\_bindings}\,T) \to F\,(\mathsf{arg\_result}\,T).\end{aligned}$$

In other words, the $\mathsf{arg\_map}\, F\, ([A_1, \ldots, A_k] \vdash B)$ is the type of functions $(F\, A_1) \times \cdots \times (F\, A_k) \to F\, B$. We extend it to lists of argument types:

$$\begin{aligned}&\mathsf{args\_map}\, F : \mathsf{list\ arg\_type} \to \mathsf{Set}\\ &\mathsf{args\_map}\, F\, [T_1, \ldots, T_n] = \{\!|\mathsf{arg\_map}\, F\, T_1, \ldots, \mathsf{arg\_map}\, F\, T_n|\!\}.\end{aligned}$$

The notation means that an element of this type is a dependent list $\{g_1, \ldots, g_n\}$, where g_i has type $(\mathsf{sort\_list}\, F\, \vec{A_i} \to F\, B_i)$ if $T_i = \vec{A_i} \vdash B_i$, for $1 \leq i \leq n$.

If we instantiate the previous definitions using the family Term for F, we obtain the representation of higher-order arguments as functions on dependent lists of terms. We call $\mathsf{Term\_arg}$ and $\mathsf{Term\_args}$ the instantiations of $\mathsf{arg\_map}$ and $\mathsf{args\_map}$ for $F := \mathsf{Term}$. An argument with specification $[A_1, \ldots, A_k] \vdash B$ will have the type:

$$\begin{aligned}\mathsf{Term\_arg}\, ([A_1, \ldots, A_k] \vdash B) &:= \mathsf{arg\_map\ Term}\, ([A_1, \ldots, A_k] \vdash B)\\ &= \mathsf{sort\_list\ Term}\, [A_1, \ldots, A_k] \to \mathsf{Term}\, B\\ &\cong (\mathsf{Term}\, A_1) \times \cdots \times (\mathsf{Term}\, A_k) \to \mathsf{Term}\, B.\end{aligned}$$

And a list of arguments will have the following type:

$$\begin{aligned}&\mathsf{Term\_args} : \mathsf{list\ arg\_type} \to \mathsf{Set}\\ &\mathsf{Term\_args}\, [\vec{A_1} \vdash B_1, \ldots, \vec{A_n} \vdash B_n]\\ &\quad := \mathsf{args\_map\ Term}\, [\vec{A_1} \vdash B_1, \ldots, \vec{A_n} \vdash B_n]\\ &\quad = \{\!|\mathsf{sort\_list\ Term}\, \vec{A_1} \to \mathsf{Term}\, B_1, \ldots, \mathsf{sort\_list\ Term}\, \vec{A_n} \to \mathsf{Term}\, B_n|\!\}.\end{aligned}$$

When defining a recursive function on terms, the results associated to bound variables are stored in an assignment. Formally assignments are defined using *streams*: a stream $s : \mathsf{Stream}\, X$ is an infinite sequence of elements of X. An assignment is a family of streams.

DEFINITION 1. *An* assignment *is a family of streams of the family F indexed on the sorts:* $\mathsf{Assignment}\, F = \forall A : \mathsf{sort}, \mathsf{Stream}\, (F\, A)$. *We use the symbol α to denote a generic assignment.*

This definition means that for every sort name A, we have an infinite sequence $\alpha_A = a_0, a_1, a_2, \ldots$ of elements of $F\, A$.

Assignments are used to give interpretations for the de Bruijn variables during the definition of a function by recursion. So the variable x_j^A will be interpreted as $(\alpha\, A)_i$, the ith element of the stream associated with the sort A. Representing assignments as streams harmonizes nicely with the use of de Bruijn indices: whenever we go under a binder, the interpretations of the new bound variables can be simply appended in front of the stream; the old variables will be shifted to the right, automatically performing the required index increment. If $\vec{a} : \mathsf{sort\_list}\, F\, \vec{A}$, then we denote by $[\vec{a}]\alpha$ the assignment obtained by appending the elements of $\vec{a}$ in front of the streams in α.

With these notions we are ready to formulate the recursion principle.

THEOREM 1. *Given the step functions* Fvar *for free variables and* FOpr *for operation application, of the following types:*

$$\begin{aligned}
&\mathsf{Fvar} : \forall A : \mathsf{sort}, \mathbb{N} \to F\,A\\
&\mathsf{FOpr} : \ \forall f : \mathsf{operation},\\
&\qquad \mathsf{Term_args}\,(\mathsf{op_arguments}\,(\mathsf{rule}\,f)) \to\\
&\qquad \mathsf{args_map}\,F\,(\mathsf{op_arguments}\,(\mathsf{rule}\,f)) \to\\
&\qquad F\,(\mathsf{op_result}\,(\mathsf{rule}\,f)).
\end{aligned}$$

we can construct recursive functions ϕ *and* $\vec{\phi}$ *of the following types:*

$$\begin{aligned}
&\phi : \mathsf{Assignment} \to \forall A : \mathsf{sort}, \mathsf{Term}\,A \to F\,A\\
&\vec{\phi} : \mathsf{Assignment} \to \forall \vec{T} : \mathsf{list\ arg_type}, \mathsf{Term_args}\,\vec{T} \to \mathsf{args_map}\,F\,\vec{T}
\end{aligned}$$

satisfying the reduction behaviour given by the equations:

$$\begin{aligned}
&\phi\,\alpha\,A\,\mathsf{v}_i^A = \mathsf{Fvar}\,A\,i\\
&\phi\,\alpha\,A\,\mathsf{x}_j^A = (\alpha\,A)_j\\
&\phi\,\alpha\,C\,(\mathsf{Opr}\,f\,\vec{h}) = \mathsf{FOpr}\,f\,\underline{\vec{h}}\,(\vec{\phi}\,\alpha\,_\,\vec{h})\\
&\\
&\vec{\phi}\,\alpha\,[]\,\{\} = \{\}\\
&\vec{\phi}\,\alpha\,(\vec{A} \vdash B :: \vec{T})\,\{h :: \vec{h}\} = \{(\mathsf{fun}\,\vec{a} \mapsto \phi\,[\vec{a}]\alpha\,(\mathsf{bind}\,_\,h)) :: \vec{\phi}\,\alpha\,\vec{T}\,\vec{h}\}.
\end{aligned}$$

PROOF. See Subsection 3.4. ⊣

The operation bind takes a higher-order (functional) argument h and *flattens* it by replacing the binding meta-variables with de Bruijn indices:

$$\begin{aligned}
&h : \mathsf{Term_arg}\,([A_1, \ldots, A_k] \vdash B) \cong (\mathsf{Term}\,A_1) \times \cdots \times (\mathsf{Term}\,A_k) \to (\mathsf{Term}\,B)\\
&(\mathsf{bind}\,_\,h) : \mathsf{Term}\,B := \ \text{compute } h(x_1, \ldots, x_k) \text{ and then replace}\\
&\qquad\qquad\qquad\qquad \text{metavariable } x_i \text{ with a de Bruijn variable } \mathsf{x}_{i'}^{A_i}.
\end{aligned}$$

This intuitive definition is enough to follow the discussion in this section. In Section 3, we give a formal definition.

At the higher level, the user does not need to know how de Bruijn indices work. To understand the last recursion equation, it is enough to know that the function $(\mathsf{fun}\,\vec{a} \mapsto \phi\,[\vec{a}]\alpha\,(\mathsf{bind}\,\_\,h))$ maps a list $\vec{a} = \{a_1, \ldots, a_k\} : \mathsf{sort\_list}\,F\,[A_1, \ldots, A_k]$ to the result of the recursive call of ϕ on h where the occurrences of the meta-variables of type $\mathsf{Term}\,A_1, \ldots, \mathsf{Term}\,A_k$ are mapped to $a_1, \ldots, a_k$, respectively.

We also get an induction principle on the higher-order abstract syntax. Let $P : \forall A : \mathsf{sort}, \mathsf{Term}\,A \to \mathsf{Prop}$ be a predicate on terms. We can extend it to lists of higher-order arguments in the following way:

$$\begin{aligned}
&\text{If } \{g_1, \ldots, g_n\} : \mathsf{Term_args}\,[[A_{11}, \ldots, A_{1k_1}] \vdash B_1, \ldots, [A_{n1}, \ldots, A_{nk_n}] \vdash B_n],\\
&\text{then } \vec{P}\,\{g_1, \ldots, g_n\} = (P\,_\,(\mathsf{bind}\,_\,g_1)) \wedge \cdots \wedge (P\,_\,(\mathsf{bind}\,_\,g_n)).
\end{aligned}$$

THEOREM 2. *If the following hypotheses are true:*

$$\begin{aligned}
&\forall (A : \mathsf{sort})(i : \mathbb{N}), P\,A\,\mathsf{v}_i^A\\
&\forall (A : \mathsf{sort})(i : \mathbb{N}), P\,A\,\mathsf{x}_i^A\\
&\forall (f : \mathsf{operation})(\vec{g} : \mathsf{Term_args}\,(\mathsf{op_arguments}\,(\mathsf{rule}\,f))),\\
&\qquad \vec{P}\,\vec{g} \to P\,_\,(\mathsf{Opr}\,f\,\vec{g});
\end{aligned}$$

then, for every sort A *and term* $t : \mathsf{Term}\,A$, $P\,A\,t$ *is true.*

PROOF. See Subsection 3.5. ⊣

Theorem 2 instantiates directly to an induction principle for our running example. For use in practice, it is convenient if the inductive hypothesis for Opr is formulated as two separate hypotheses for abs_λ and app_λ. In the case of abs_λ we also define the *body of a function* $F : \mathsf{Term}\, A \to \mathsf{Term}\, B$ to be:

$$\mathsf{body}\ h = \mathsf{bind}\,([A] \vdash B)\,(\mathsf{fun}\,\langle x\rangle \mapsto h\,x) : \mathsf{Term}\, B.$$

Then we can easily convert the general induction principle to one for the typed λ-calculus.

THEOREM 3. *Let* $P : \forall A : \mathsf{type}_\lambda, \mathsf{Term}\, A \to \mathsf{Prop}$ *be a predicate on terms. If the following hypotheses are true:*

$$\begin{array}{l}
\forall (A : \mathsf{type}_\lambda)(i : \mathbb{N}), P\, A\, \mathsf{v}_i^A \\
\forall (A : \mathsf{type}_\lambda)(i : \mathbb{N}), P\, A\, \mathsf{x}_i^A \\
\forall (A, B : \mathsf{type}_\lambda)(h : \mathsf{Term}\, A \to \mathsf{Term} B), \\
\quad P\, B\, (\mathsf{body}\, h) \to P\, (\mathsf{arrow}_\lambda\, A\, B)\, (\mathsf{Lambda}\, x\, \mathsf{in}\, A\, \mathsf{gives}\, (h\, x)) \\
\forall (A, B : \mathsf{type}_\lambda)(f : \mathsf{Term}\, (\mathsf{arrow}_\lambda\, A\, B))(a : \mathsf{Term}\, A), \\
\quad P\, (\mathsf{arrow}_\lambda\, A\, B)\, f \to P\, A\, a \to P\, B\, (f\, \mathsf{of}\, a)
\end{array}$$

Then, for every $A : \mathsf{type}_\lambda$ *and* λ*-term* $t : \mathsf{Term}\, A$, $P\, A\, t$ *is true.*

In a similar way we can adapt the recursion principle to our specific object language. Here is its instantiation for the simply typed λ-calculus.

THEOREM 4. *Given the step functions* Fvar *for free variables,* Fabs *for abstractions, and* Fapp *for application, of the following types:*

$$\begin{array}{ll}
\mathsf{Fvar}: & \forall A : \mathsf{type}_\lambda, \mathbb{N} \to F\, A \\
\mathsf{Fabs}: & \forall A, B : \mathsf{type}_\lambda, (\mathsf{Term}\, A \to \mathsf{Term}\, B) \\
& \quad \to (F\, A \to F\, B) \to F\, (\mathsf{arrow}_\lambda\, A\, B) \\
\mathsf{Fapp}: & \forall A, B : \mathsf{type}_\lambda, (\mathsf{Term}\, (\mathsf{arrow}_\lambda\, A\, B) \times \mathsf{Term}\, B) \\
& \quad \to (F\, (\mathsf{arrow}_\lambda\, A\, B) \times F\, B) \to F\, B
\end{array}$$

we can construct a recursive function ϕ *of the following type:*

$$\phi : \mathsf{Assignment} \to \forall A : \mathsf{type}_\lambda, \mathsf{Term}\, A \to F\, A$$

satisfying the reduction behaviour given by the equations:

$$\begin{array}{l}
\phi\, \alpha\, A\, \mathsf{v}_i^A = \mathsf{Fvar}\, A\, i \\
\phi\, \alpha\, A\, \mathsf{x}_j^A = (\alpha\, A)_j \\
\phi\, \alpha\, (\mathsf{arrow}_\lambda\, A\, B)\, (\mathsf{Lambda}\, x\, \mathsf{in}\, A\, \mathsf{gives}\, (h\, x)) \\
\qquad = \mathsf{Fabs}\, A\, B\, \underline{h}\, (\mathsf{fun}\, a \mapsto \phi\, [a] \alpha\, (\mathsf{body}\, h)) \\
\phi\, \alpha\, B\, (f\, \mathsf{of}\, a) = \mathsf{Fapp}\, A\, B\, \langle f, a\rangle\, \langle \phi\, \alpha\, (\mathsf{arrow}_\lambda\, A\, B)\, f, \phi\, \alpha\, A\, a\rangle
\end{array}$$

We omitted the extension $\vec{\phi}$ of ϕ to higher-order arguments, and instead directly unfolded its occurrence in the reduction rule for $(\mathsf{Lambda}\, x\, \mathsf{in}\, A\, \mathsf{gives}\, (h\, x))$.

Notice the true higher-order nature of this recursion principle: in the recursion step Fabs, the recursive argument is not just the result on the body of the function, but a mapping giving the result for all possible assignments to the abstracted variable.

For example, the function size mentioned earlier is defined by the following step functions:

$$\begin{aligned}&\mathsf{Fvar}\,A\,i = 1\\&\mathsf{Fabs}\,A\,B\,h\,g = (g\,1) + 1\\&\mathsf{Fapp}\,A\,B\,\langle f, a\rangle\,\langle n_f, n_a\rangle = n_f + n_a + 1.\end{aligned}$$

In this case we want to give to every bound variable the size 1, so we define $\mathsf{size} = \phi\,\vec{1}$, where $\vec{1}$ is the constant stream where all elements are 1, resulting in the following reduction behaviour:

$$\begin{aligned}&\mathsf{size}\,A\,\mathsf{v}_j^A = 1\\&\mathsf{size}\,A\,\mathsf{x}_j^A = 1\\&\mathsf{size}\,(\mathsf{arrow}_\lambda\,A\,B)\,(\mathsf{Lambda}\,x\,\mathsf{in}\,A\,\mathsf{gives}\,(h\,x)) = (\mathsf{size}\,(\mathsf{body}\,h)) + 1\\&\mathsf{size}\,B\,(f\,\mathsf{of}\,a) = (\mathsf{size}\,(\mathsf{arrow}_\lambda\,A\,B)\,f) + (\mathsf{size}\,A\,a) + 1\end{aligned}$$

as desired.

The size example is quite simple in that we didn't use the higher-order power of the recursion principle and the assignment argument. The second order recursive call g in the abstraction case was used only with constant argument 1, and the assignment is constantly equal to $\vec{1}$ during reduction.

To illustrate what a truly higher-order application of the recursion principle is, let us show the definition of the model construction $\mathsf{interpret}_X$ mentioned earlier. We assume as parameter a fixed interpretation of the free variables: $\mathsf{Fvar}\,A\,i : \forall A : \mathsf{type}_\lambda, \mathbb{N} \to \mathsf{Model}_X\,A$. The step functions are the following:

$$\begin{aligned}&\mathsf{Fabs}\,A\,B\,h\,g = g\\&\mathsf{Fapp}\,A\,B\,\langle f, a\rangle\,\langle g, x\rangle = g\,x\end{aligned}$$

Setting $\mathsf{interpret}_X = \phi$, we obtain the following reduction rules:

$$\begin{aligned}&\mathsf{interpret}_X\,\alpha\,A\,\mathsf{v}_i^A = \mathsf{Fvar}\,A\,i\\&\mathsf{interpret}_X\,\alpha\,A\,\mathsf{x}_j^A = (\alpha\,A)_j\\&\mathsf{interpret}_X\,\alpha\,(\mathsf{arrow}_\lambda\,A\,B)\,(\mathsf{Lambda}\,x\,\mathsf{in}\,A\,\mathsf{gives}\,(h\,x))\\&\qquad = \mathsf{fun}\,a \mapsto \mathsf{interpret}_X\,[a]\alpha\,(\mathsf{body}\,h)\\&\mathsf{interpret}_X\alpha\,B\,(f\,\mathsf{of}\,a) = (\mathsf{interpret}_X\,\alpha\,(\mathsf{arrow}_\lambda\,A\,B)\,f)\,(\mathsf{interpret}_X\,\alpha\,A\,a).\end{aligned}$$

Notice how the usual interpretation of abstraction in a set-theoretic model, as the function mapping an argument to the interpretation of the body under a modified assignment, is automatically validated by the recursion principle from the simple definition of Fabs.

§3. Technical details. Let us explain some details of our Coq implementation. Under the higher-order syntax described in the previous section, we have a de Bruijn syntax defined as a standard Coq inductive type.

3.1. De Bruijn syntax. To define terms over the signature σ, we need to define the type of terms simultaneously with the type of term lists with bindings. A list of terms with bindings is an object of the form:

$$\langle\!\langle [\mathsf{x}_{11}^{A_{11}}, \ldots, \mathsf{x}_{1k_1}^{A_{1k_1}}]t_1, \ldots, [\mathsf{x}_{n1}^{A_{n1}}, \ldots, \mathsf{x}_{nk_n}^{A_{nk_n}}]t_n \rangle\!\rangle.$$

It is the list of the terms $t_1, \ldots, t_n$, each binding a list of variables: the term t_i binds the variables $\mathsf{x}_{i1}, \ldots, \mathsf{x}_{ik_i}$, where the variable x_{ij} has sort A_{ij}. In keeping with the de Bruijn convention, we don't actually need to specify the names of

the abstracted variables, but they will automatically be determined as indices, starting with index 0 for the rightmost variable.

Informally, terms and term lists are defined by the following rules (in the operation rule, assume that $(\mathsf{rule}\, f) = [\vec{A_1} \vdash B_1, \ldots, \vec{A_n} \vdash B_n] /\!\!/ C$):

$$\begin{array}{l}\mathsf{Term} : \mathsf{sort} \to \mathsf{Set}\\ \mathsf{TermList} : \mathsf{list}\,(\mathsf{arg\_type}\,\mathsf{sort}) \to \mathsf{Set}\end{array}$$

$$\frac{A : \mathsf{sort} \quad i : \mathbb{N}}{\mathsf{v}_i^A : \mathsf{Term}\, A}\ (\text{free variable}) \qquad \frac{A : \mathsf{sort} \quad i : \mathbb{N}}{\mathsf{x}_i^A : \mathsf{Term}\, A}\ (\text{bound variable})$$

$$\frac{f : \mathsf{operation} \quad \langle\!\langle t_1, \ldots, t_n \rangle\!\rangle : \mathsf{TermList}\,[\vec{A_1} \vdash B_1, \ldots, \vec{A_n} \vdash B_n]}{(\mathsf{opr}\, f\, \langle\!\langle t_1, \ldots, t_n \rangle\!\rangle) : \mathsf{Term}\, C}$$

$$\frac{t_1 : \mathsf{Term}\, B_1 \quad \cdots \quad t_n : \mathsf{Term}\, B_n}{\langle\!\langle [\mathsf{x}_{1j}^{A_{1j}}]_{j=1}^{k_1} t_1, \ldots, [\mathsf{x}_{nj}^{A_{nj}}]_{j=1}^{k_n} t_n \rangle\!\rangle : \mathsf{TermList}\,[\vec{A_1} \vdash B_1, \ldots, \vec{A_n} \vdash B_n]}.$$

In the formal definition, as mentioned, we don't need to explicitly mention the names of the variables. Also, in the last rule the list of bound sorts for each term in the list is specified by the type of the list. Therefore, we need to put explicitly in the list just the terms. The formal definition of terms and term lists is the following:

$$\begin{array}{rl}\textbf{Inductive}\ \mathsf{Term} : & \mathsf{sort} \to \mathsf{Set}\\ \mathsf{TermList} : & \mathsf{list}\,(\mathsf{arg\_type}\,\mathsf{sort}) \to \mathsf{Set}\\ & \\ \mathsf{var} : & \forall A : \mathsf{sort}, \mathbb{N} \to \mathsf{Term}\, A\\ \mathsf{bnd} : & \forall A : \mathsf{sort}, \mathbb{N} \to \mathsf{Term}\, A\\ \mathsf{opr} : & \forall f : \mathsf{operation},\\ & \mathsf{TermList}\,(\mathsf{op\_arguments}\,(\mathsf{rule}\, f)) \to \mathsf{Term}\,(\mathsf{op\_result}\,(\mathsf{rule}\, f))\\ & \\ \mathsf{nil\_term} : & \mathsf{TermList}\,[]\\ \mathsf{cons\_term} : & \forall T : \mathsf{arg\_type}\,\mathsf{sort}, \forall \vec{T} : \mathsf{list}\,(\mathsf{arg\_type}\,\mathsf{sort}),\\ & \mathsf{Term}\,(\mathsf{arg\_result}\, T) \to \mathsf{TermList}\,\vec{T} \to \mathsf{TermList}\,(T :: \vec{T}).\end{array}$$

So cons_term doesn't explicitly mention the bound variables in the term t : $\mathsf{Term}\,(\mathsf{arg\_result}\, T)$. The binding list is implicitly given by the argument type T using the de Bruijn convention for indexing. For example, if $T = [A, A', A, A, A'] \vdash B$, then the bound variables are $\mathsf{x}_2^A, \mathsf{x}_1^{A'}, \mathsf{x}_1^A, \mathsf{x}_0^A, \mathsf{x}_0^{A'}$. Bound variables are numbered from right to left starting from 0; each sort has an independent indexing.

We must translate the higher-order notation of the previous section into this de Bruijn syntax. The fundamental point is to establish a correspondence between the two ways to apply an operation:

$$\mathsf{Opr}\, f\, \{g_1, \ldots\, g_n\} \leftrightsquigarrow \mathsf{opr}\, f\, \langle\!\langle a_1, \ldots, a_n \rangle\!\rangle.$$

This transformation is performed by the mentioned bind operation. Before defining it, we need some definitions and results about variables and substitution.

3.2. Variables and binding. The operator newvar finds a new free variable of a specific sort with respect to a term. If $t : \mathsf{Term}\, A$ is a term and A' is a sort, then $(\mathsf{newvar}\, A'\, t)$ gives an index i such that $\mathsf{v}_i^{A'}$ does not occur in t. More precisely, all free variables of sort A' occurring in t have indices smaller that i. It is defined by recursion on the structure of t.

The identity of bound variables is determined by the number of bindings of their sort above them. If we have a term with bindings of sorts $[A_1, \ldots, A_k]$, we want to know what index the variable $\mathsf{x}_i^{A'}$ will have under the bindings:

$$\cdots\ \mathsf{x}_i^{A'}\ \cdots (\mathsf{opr}\, f\, \langle\!\langle \cdots, \ldots \mathsf{x}_j^{A'} \ldots, \cdots \rangle\!\rangle) \cdots$$

In this expression $\mathsf{x}_i^{A'}$ and $\mathsf{x}_j^{A'}$ are meant to represent occurrences of the same de Bruijn variable. The indices of the bound variables are determined by the de Bruijn convention. The value of j depends on the number of occurrences of A' in the binding list of the argument of f where $\mathsf{x}_j^{A'}$ occurs. Suppose there are h occurrences of A' in it; then they bind the variables $\mathsf{x}_{h-1}^{A'}, \ldots, \mathsf{x}_0^{A'}$. Therefore, the indices of all other variables of sort A' are shifted by h, so we must have $j = i + h$.

This shifting is performed by the function bind_inc: $(\mathsf{bind\_inc}\, [A_1, \ldots, A_k]\, A'\, i)$ gives the value $j = i + h$. It is defined by recursion on $[A_1, \ldots, A_k]$, the binding list of the argument of f. It uses the decidability of equality of sorts to check whether A' is equal to the A_js. Here is where the assumption sig_dec is used.

Given a term $t : \mathsf{Term}\, B$, a free variable v_i^A and a de Bruijn variable x_j^A, we can define the operation of swapping the two variables in t: $t[\mathsf{v}_i^A \leftrightarrow \mathsf{x}_j^A]$. It is defined by recursion on the structure of t, taking care to increment the index of x_j^A with bind_inc every time we go under a binding operation.

Note: We could have chosen to have only one indexing for the variables, regardless of their sorts. This would have avoided the need for decidability. However, it would have required carrying around an assignment of sorts to the indices everywhere, so we opted for independent indexing of every sort.

Now we have the machinery to bind meta-variables to de Bruijn indices. We start with the simple example of a single variable. We keep track of variables already bound with an argument $\vec{A} : \mathsf{list\ sort}$.

The index of the new bound variable of sort A must be $j = (\mathsf{bind\_inc}\, \vec{A}\, A\, 0)$:

$$\begin{aligned}
&\mathsf{one_arg_bind}\, A\, B\, \vec{A} : (\mathsf{Term}\, A \to \mathsf{Term}\, B) \to \mathsf{Term}\, B \\
&\mathsf{one_arg_bind}\, A\, B\, \vec{A}\, g = (g\, \mathsf{v}_i^A)[\mathsf{v}_i^A \leftrightarrow \mathsf{x}_j^A] \\
&\qquad \text{where}\ \ i = \mathsf{newvar}\, A\, (g\, \mathsf{x}_0^A) \\
&\qquad\qquad\ \ j = \mathsf{bind_inc}\, \vec{A}\, A\, 0
\end{aligned}$$

Note: We must require that v_i^A is a new variable for g. Since g is a function, it is not immediately clear what this means. It does **not** mean that v_i^A is new for every instance of g (for example, if g is the identity, no variable is new for all instances). We are mainly interested in the case where g is uniform, that is, g is of the form $\mathsf{fun}\, X : \mathsf{Term}\, A \mapsto C[X]$ where $C[\_]$ is a term build up using only variables and operations of the signature. We want that v_i^A doesn't occur in C. So it is enough to apply g to a dummy argument that doesn't add

new free variables and find a new variable for that term. That is why we use $i = (\mathsf{newvar}\, A\, (g\, \mathsf{x}_0^A))$.

For bindings of several variables, we use the same process simultaneously on all the variables. Similarly to the definition of swapping for a single variable, we can swap several variables simultaneously. We use the notation $\mathbb{N}^{[A_1,\dots,A_k]}$ (Coq notation: `(indices [A1,... ,Ak])`) for the type $(\mathsf{sort\_list}\, (\mathsf{fun}\, A : \mathsf{sort} \mapsto \mathbb{N})\, [A_1, \dots, A_k])$. Its elements are lists of indices $\langle i_1, \dots, i_k \rangle$ denoting either free or de Bruijn variables; let us use the notation $\mathbf{v}_{\langle i_1,\dots,i_k\rangle}$ (Coq: `vars As is`) for $\langle \mathsf{v}_{i_1}^{A_1}, \dots, \mathsf{v}_{i_k}^{A_k} \rangle$. We use the similar notation $\mathbf{x}_{\langle i_1,\dots,i_k\rangle}$ (Coq: `bnds As is`) for de Bruijn variables.

Let now $\langle i_1, \dots, i_k \rangle$ and $\langle j_1, \dots, j_k \rangle$ be two elements of $\mathbb{N}^{[A_1,\dots,A_k]}$, denoting a list of indices for free variables and a list of indices for de Bruijn variables, respectively. We then define the operation of simultaneously swapping each free variable with the corresponding de Bruijn variable:

$$\mathsf{vars\_swap}\, [A_1, \dots, A_k]\, \langle i_1, \dots, i_k \rangle\, \langle j_1, \dots, j_k \rangle$$

which we also denote by the notation

$$t[\mathsf{v}_{i_1}^{A_1} \leftrightarrow \mathsf{x}_{j_1}^{A_1}, \dots, \mathsf{v}_{i_k}^{A_k} \leftrightarrow \mathsf{x}_{j_k}^{A_k}] \quad \text{or} \quad t[\mathbf{v}_{\langle i_1,\dots,i_k\rangle} \leftrightarrow \mathbf{x}_{\langle j_1,\dots,j_k\rangle}].$$

As mentioned earlier, given a list of sorts $[A_1, \dots, A_k]$, the indices of the corresponding bound variables are determined by the de Bruijn convention. This is formalized by the following operator $\mathsf{bind\_vars}\, [A_1, \dots, A_k] : \mathbb{N}^{[A_1,\dots,A_k]}$ for which we use the notation $\mathbf{x}^{[A_1,\dots,A_k]}$. For example $\mathbf{x}^{[A,A',A,A,A']} = \langle 2, 1, 1, 0, 0 \rangle$.

We also have an operator that defines a list of new variables of specified sorts with respect to a given term: $(\mathsf{newvars}\, [A_1, \dots, A_k]\, t) = \langle i_1, \dots, i_k \rangle$ such that the variables $\mathsf{v}_{i_1}^{A_1}, \dots, \mathsf{v}_{i_k}^{A_k}$ do not occur in t.

We define the operation of binding meta-variables to de Bruijn variables in a similar way to what we have done for a single variable:

$$\begin{aligned}
&\mathsf{bind_args}\, [A_1, \dots, A_k]\, B : (\mathsf{sort_list}\, \mathsf{Term}\, [A_1, \dots, A_k] \to \mathsf{Term}\, B) \to \mathsf{Term}\, B \\
&\mathsf{bind_args}\, [A_1, \dots, A_k]\, B\, g = (g\, \mathbf{v}_{\langle i_1,\dots,i_k\rangle})[\mathbf{v}_{\langle i_1,\dots,i_k\rangle} \leftrightarrow \mathbf{x}_{\langle j_1,\dots,j_k\rangle}] \\
&\qquad \text{where } \langle i_1, \dots, i_k \rangle = \mathsf{newvars}\, [A_1, \dots, A_k]\, (g\, \mathbf{x}_{\langle j_1,\dots,j_k\rangle}) \\
&\qquad\qquad \langle j_1, \dots, j_k \rangle = \mathbf{x}^{[A_1,\dots,A_k]}.
\end{aligned}$$

If we apply this binding operator directly to meta-arguments, we obtain the bind operation that we mentioned before:

$$\begin{aligned}
&\mathsf{bind} : \forall T : \mathsf{arg_type}, (\mathsf{Term_arg}\, T) \to \mathsf{Term}\, (\mathsf{arg_result}\, T) \\
&\mathsf{bind}\, T = \mathsf{bind_args}\, (\mathsf{arg_bindings}\, T)\, (\mathsf{arg_result}\, T).
\end{aligned}$$

We extend it to lists of argument types, so we can simultaneously bind different variables in different arguments.

$$\begin{aligned}
&\mathsf{binds}\, \vec{T} : \mathsf{Term_args}\, \vec{T} \to \mathsf{TermList}\, \vec{T} \\
&\mathsf{binds}\, [T_1, \dots, T_n]\, \{g_1, \dots, g_n\} = \langle\!\langle \mathsf{bind}\, T_1\, g_1, \dots, \mathsf{bind}\, T_n\, g_n \rangle\!\rangle.
\end{aligned}$$

We have now all the tools needed to define the higher-order application operator:

$$\begin{aligned}
&\mathsf{Opr}\, f : \mathsf{Term_args}\, (\mathsf{op_arguments}\, (\mathsf{rule}\, f)) \to \mathsf{Term}\, (\mathsf{op_result}\, (\mathsf{rule}\, f)) \\
&\mathsf{Opr}\, f\, \vec{g} = \mathsf{opr}\, f\, (\mathsf{binds}\, (\mathsf{op_arguments}\, (\mathsf{rule}\, f))\, \vec{g}).
\end{aligned}$$

Assume that the operation f has the rule $[\vec{A_1} \vdash B_1, \ldots, \vec{A_n} \vdash B_n] /\!\!/ C$. Then:

$$\begin{aligned}&\mathsf{Opr}\, f : \{\!|\mathsf{sort\_list\ Term}\ \vec{A_1} \to B_1, \ldots, \mathsf{sort\_list\ Term}\ \vec{A_n} \to B_n|\!\} \to \mathsf{Term}\, C\\ &\mathsf{Opr}\, f\, \{g_1, \ldots, g_n\} = \mathsf{opr}\, f\, \langle\!|\mathsf{bind}\, (\vec{A_1} \vdash B_1)\, g_1, \ldots, \mathsf{bind}\, (\vec{A_n} \vdash B_n)\, g_n|\!\rangle.\end{aligned}$$

3.3. Application. The inverse operation of binding is application of a de Bruijn term with bindings to a list of arguments. This is the same as substitution of a de Bruijn variable with terms, keeping track of the increase of the variable index under abstraction. We want to do this simultaneously for several variables. We use the notation $t[\mathbf{x}_{\vec{i}}/\vec{a}]$ (Coq: `sub t As is al`) for the simultaneous substitution of the list of de Bruijn variables $\mathbf{x}_{\vec{i}}$ with the list of terms $\vec{a}$ of the correct sort. It is defined by recursion on t, mutually with its extension subs to lists of terms with bindings.

Application is just the substitution of the abstracted variables: $t[\mathbf{x}^{\vec{A}}/\bullet]$, where $\vec{A}$ is the list of the variables abstracted in t. The bullet $\bullet$ indicates that the substituted terms are meta-abstracted: $t[\mathbf{x}^{\vec{A}}/\bullet] := (\mathsf{fun}\, \vec{x} \mapsto t[\mathbf{x}^{\vec{A}}/\vec{x}])$. We define application directly on lists of terms with bindings:

$$\begin{aligned}&\mathsf{aps}\, \vec{T} : \mathsf{TermList}\, \vec{T} \to \mathsf{Term\_args}\, \vec{T}\\ &\mathsf{aps}\, []\, \langle\!|\rangle\!| = \{\}\\ &\mathsf{aps}\, (T :: \vec{T})\, \langle\!|t :: \vec{t}|\!\rangle = \{t[\mathbf{x}^{\mathsf{arg\_bindings}\, T}/\bullet] :: (\mathsf{aps}\, \vec{T}\, \vec{t})\}.\end{aligned}$$

THEOREM 5. *Application is a right inverse of binding:*

$$\forall (\vec{T} : \mathsf{list\ arg\_type})(\vec{t} : \mathsf{TermList}\, \vec{T}), \vec{t} = \mathsf{binds}\, \vec{T}\, (\mathsf{aps}\, \vec{T}\, \vec{t}).$$

PROOF. We have that:

$$\begin{aligned}&\mathsf{binds}\, [T_1 \ldots, T_n]\, (\mathsf{aps}\, [T_1 \ldots, T_n]\, \langle\!|t_1, \ldots t_n|\!\rangle)\\ &= \langle\!|\mathsf{bind}\, T_1\, (t_1[\mathbf{x}^{\mathsf{arg\_bindings}\, T_1}/\bullet]), \ldots, \mathsf{bind}\, T_n\, (t_n[\mathbf{x}^{\mathsf{arg\_bindings}\, T_n}/\bullet])|\!\rangle\end{aligned}$$

So it is sufficient to prove that for every $T : \mathsf{arg\_type}$ and $t : \mathsf{Term}\, (\mathsf{op\_result}\, T)$ we have $t = \mathsf{bind}\, T\, (t[\mathbf{x}^{\mathsf{arg\_bindings}\, T}/\bullet])$. Assume that $T = \vec{A} \vdash B$, then:

$$\mathsf{bind}\, T\, (t[\mathbf{x}^{\mathsf{arg\_bindings}\, T}/\bullet]) = \mathsf{bind\_args}\, \vec{A}\, B\, (t[\mathbf{x}^{\vec{A}}/\bullet]) = t[\mathbf{x}^{\vec{A}}/\mathbf{v}_{\vec{i}}][\mathbf{v}_{\vec{i}} \leftrightarrow \mathbf{x}_{\vec{j}}]$$

where $\mathbf{x}_{\vec{j}} = \mathbf{x}^{\vec{A}}$ and $\mathbf{v}_{\vec{i}} = \mathsf{newvars}\, \vec{A}\, t[\mathbf{x}^{\vec{A}}/\mathbf{x}_{\vec{j}}] = \mathsf{newvars}\, \vec{A}\, t$. Since the variables $\mathbf{v}_{\vec{i}}$ are fresh for t, we have $t[\mathbf{x}^{\vec{A}}/\mathbf{v}_{\vec{i}}][\mathbf{v}_{\vec{i}} \leftrightarrow \mathbf{x}_{\vec{j}}] = t$ as desired. ⊣

Note that in general aps is not the left inverse of binds. In fact, if we start with a list of functions, bind their meta-arguments, and then lift the result again to the meta-level; we won't in general get back the original functions. If some of them were not uniform, we will instead obtain their *uniformization*, that is, the system will choose one instance of the function and generalize it to all other arguments.

DEFINITION 2. *The* uniformization *of a list of functions* $\vec{h} : \mathsf{Term\_args}\, \vec{T}$ *is defined as the list of functions:* $\underline{\vec{h}} = \mathsf{aps}\, \vec{T}\, (\mathsf{binds}\, \vec{T}\, \vec{h})$.

We expect lists of functions used as arguments of operations in higher-order syntax to be extensionally equal to their uniformization. If they are not, then we have exotic terms.

3.4. The Recursion Principle. The higher-order recursion principle is translated internally into the structural recursion principle on de Bruijn notation. This is a standard principle that can easily be derived in Coq.

Let $H : \mathsf{sort} \to \mathsf{Set}$ be a family of types indexed on the sorts. The *recursion principle on the de Bruijn notation* is the standard structural recursion in Coq (but it has to be explicitly given by a Fixpoint definition, because the automatic recursor does not perform mutual recursion).

Let the following step functions be given:

$$\begin{array}{rl}
\mathsf{Hvar}: & \forall A : \mathsf{sort}, \mathbb{N} \to H\,A \\
\mathsf{Hbnd}: & \forall A : \mathsf{sort}, \mathbb{N} \to H\,A \\
\mathsf{Hopr}: & \forall (f : \mathsf{operation}), \\
& \quad \mathsf{TermList}\,(\mathsf{op_arguments}\,(\mathsf{rule}\,f)) \to \\
& \quad \mathsf{sort_list}\,H\,(\mathsf{op_arguments}\,(\mathsf{rule}\,f)) \to H\,(\mathsf{op_result}\,(\mathsf{rule}\,f)).
\end{array}$$

(The actual principle is a bit more general in that you can use any family on term lists in place of $\mathsf{sort\_list}\,H$ and have recursion steps on it for lists; but we only need the case with $\mathsf{sort\_list}\,H$.) We obtain two recursive functions on Term and TermList:

$$\begin{array}{l}
\mathsf{debruijn_term_recursion}\,H\,\mathsf{Hvar}\,\mathsf{Hbnd}\,\mathsf{Hopr}: \\
\qquad \forall A : \mathsf{sort}, \mathsf{Term}\,A \to H\,A \\
\mathsf{debruijn_term_list_recursion}\,H\,\mathsf{Hvar}\,\mathsf{Hbnd}\,\mathsf{Hopr}: \\
\qquad \forall \vec{T} : \mathsf{list\,arg_type}, \mathsf{TermList}\,\vec{T} \to \mathsf{sort_list}\,H\,\vec{T}.
\end{array}$$

For brevity, let us denote them by θ and $\vec{\theta}$, respectively. They satisfy the recursive equations:

$$\begin{array}{l}
\theta\,A\,\mathsf{v}_i^A = \mathsf{Hvar}\,A\,i \\
\theta\,A\,\mathsf{v}_j^A = \mathsf{Hbnd}\,A\,j \\
\theta\,_\,(\mathsf{opr}\,f\,\vec{t}) = \mathsf{Hopr}\,f\,\vec{t}\,(\vec{\theta}\,_\,\vec{t})
\end{array}$$

$$\begin{array}{l}
\vec{\theta}\,[]\,\langle\!\langle\rangle\!\rangle = \langle\rangle \\
\vec{\theta}\,(\vec{A} \vdash B :: \vec{T})\,\langle\!\langle t :: \vec{t}\rangle\!\rangle = \langle(\theta\,B\,t) :: (\vec{\theta}\,\vec{T}\,\vec{t})\rangle
\end{array}$$

So in conclusion we have that:

$$\theta\,\_\,(\mathsf{opr}\,f\,\langle\!\langle t_1, \ldots, t_n\rangle\!\rangle) = \mathsf{Hopr}\,f\,\langle\!\langle t_1, \ldots, t_n\rangle\!\rangle\,\langle\theta\,\_\,t_1, \ldots, \theta\,\_\,t_n\rangle$$

Notice that this is a purely syntactic recursion principle; the bindings in the term lists are completely ignored. In particular, the results on de Bruijn variables are computed by Hbnd, which may give different outputs for different de Bruijn indices, even if they happen to correspond to the same bound variable.

We must show how the higher-order function ϕ can be defined in terms of θ. First of all, let us define an appropriate H in terms of F. We take: $H\,A = \mathsf{Assignment} \to F\,A$. Then we need to define Hvar, Hbnd, and Hopr in terms of Fvar and FOpr:

$$\begin{array}{l}
\mathsf{Hvar}\,A\,i = \mathsf{fun}\,\alpha \mapsto \mathsf{Fvar}\,A\,i \\
\mathsf{Hbnd}\,A\,i = \mathsf{fun}\,\alpha \mapsto (\alpha\,A)_i \\
\mathsf{Hopr}\,f\,\vec{t}\,\vec{r} = \mathsf{fun}\,\alpha \mapsto \mathsf{FOpr}\,f\,(\mathsf{aps}\,_\,\vec{t})\,(\vec{r} \bullet \alpha).
\end{array}$$

where $\vec{g} = \vec{r} \bullet \alpha$ is the list of functions defined in the following way, assuming $(\mathsf{rule}\, f) = [[A_{11}, \ldots, A_{1k_1}] \vdash B_1, \ldots, [A_{n1}, \ldots, A_{nk_n}] \vdash B_n] /\!\!/ C$ and $\vec{r} = \langle r_1, \ldots, r_n \rangle$:

$$\begin{array}{l} \vec{g} : \mathsf{args\_map}\, F\, [[A_{11}, \ldots, A_{1k_1}] \vdash B_1, \ldots, [A_{n1}, \ldots, A_{nk_n}] \vdash B_n] \\ \vec{g} = \langle g_1, \ldots, g_n \rangle \\ g_i = \mathsf{fun}\, \vec{a}_i : \mathsf{sort\_list}\, F\, [A_{i1}, \ldots, A_{ik_i}] \mapsto r_i\, [\vec{a}_i]\alpha \end{array}$$

Having defined θ using these parameters, we define ϕ and $\vec{\phi}$ as:

$$\begin{array}{l} \phi\, \alpha\, A\, t = \theta\, A\, t\, \alpha \\ \vec{\phi}\, \alpha\, \vec{T}\, \vec{h} = (\vec{\theta}\, \vec{T}\, (\mathsf{binds}\, \vec{T}\, \vec{h})) \bullet \alpha \end{array}$$

We have to prove that they satisfy the higher-order recursion relations of Theorem 1. The equations for free and bound variables are immediate. The proof of the equation for operation application follows from Theorem 5 in the following way:

$$\begin{array}{rl} \phi\, \alpha\, C\, (\mathsf{Opr}\, f\, \vec{h}) & = \theta\, C\, (\mathsf{Opr}\, f\, \vec{h})\, \alpha \\ & = \theta\, C\, (\mathsf{opr}\, f\, (\mathsf{binds}\, \_\, \vec{h}))\, \alpha \\ & = \mathsf{Hopr}\, f\, (\mathsf{binds}\, \_\, \vec{h})\, (\vec{\theta}\, \_\, (\mathsf{binds}\, \_\, \vec{h}))\, \alpha \\ & = \mathsf{FOpr}\, f\, (\mathsf{aps}\, \_\, (\mathsf{binds}\, \_\, \vec{h}))\, ((\vec{\theta}\, \_\, (\mathsf{binds}\, \_\, \vec{h})) \bullet \alpha) \\ & = \mathsf{FOpr}\, f\, \underline{\vec{h}}\, (\vec{\phi}\, \alpha\, \_\, \vec{h}). \end{array}$$

The equations for $\vec{\phi}$ can easily be proved by noticing that they are equivalent to the single equation $\vec{\phi}\, \alpha\, \vec{T}\, \vec{h} = \vec{r} \bullet \alpha$, with $r_i = \mathsf{fun}\, \alpha' \mapsto \phi\, \alpha'\, \_\, (\mathsf{bind}\, \_\, h_i)$.

3.5. The Induction Principle. The proof of the induction principle is very similar to that of the recursion principle. Therefore, we only point out two differences.

In one respect the proof is easier, because we don't need to use an assignment for de Bruijn variables: It is enough that the predicate P is true for the variables, we are not interested in changing those proofs when going under an abstraction. We also don't need to prove reduction equations.

In another respect the proof is slightly more difficult: The conclusion $(P\, A\, t)$ depends on the term t, while in the recursion principle the conclusion $(F\, A)$ only depended on the sort A. As a consequence, when we apply structural induction on the de Bruijn syntax, the inductive case for operation application requires a proof of $(P\, \_\, (\mathsf{opr}\, f\, \vec{t}))$ for any operation f and argument list $\vec{t}$. We cannot directly apply the induction hypothesis given by the statement of Theorem 2, because its conclusion is $(P\, \_\, (\mathsf{Opr}\, f\, \vec{g}))$. We must first convert $(\mathsf{opr}\, f\, \vec{t})$ to the form $(\mathsf{Opr}\, f\, \vec{g})$. We do this by applying Theorem 5 and the definition of Opr:

$$\begin{array}{rl} \mathsf{opr}\, f\, \vec{t} & = \mathsf{opr}\, f\, (\mathsf{binds}\, (\mathsf{op\_arguments}\, (\mathsf{rule}\, f))\, (\mathsf{aps}\, \_\, \vec{t})) \\ & = \mathsf{Opr}\, f\, (\mathsf{aps}\, \_\, \vec{t}) \end{array}$$

We can then apply the hypothesis with $\vec{g} = (\mathsf{aps}\, \_\, \vec{t})$ and the proof goes through without problems.

3.6. Adequacy. It is important to show that our encodings of de Bruijn terms and object languages are adequate, i.e., that there is a bijection between the language we are encoding and its encoded form such that substitution commutes with the encoding. The first adequacy result for the original Hybrid [2] appears in [11], and since we use a standard de Bruijn encoding, the adequacy of our representation should follow directly. Adequacy for object languages such as the simply-typed λ-calculus should also follow similarly from adequacy results for Hybrid. The main difference here is that when descending through binders via the bind operator, a meta-variable becomes a *dangling index*, i.e., an index representing a bound variable that does not have a corresponding binder in the term. Establishing adequacy in this case requires mapping free variables of the object language to two kinds of variables in the formalization: free variables *and* dangling indices.

§4. Related work. The slides of a recent talk by Randy Pollack [30] give a good summary of the literature on approaches to the implementation of languages with binders. We discuss here some that are most closely related to ours.

One of the basic ideas used in this work is the notion of translation between a high-level notation and a lower-level de Bruijn representation. This idea appears in the work of Gordon [17], where bound variables are presented to the user as strings. The idea of replacing strings with a binding operator was introduced by Ambler et. al. [2], and adopted directly here. Gordon and Melham [18] used the name-as-strings syntax approach and developed a general theory of untyped λ-terms up to α-conversion, including induction and recursion principles. Norrish building on this work, improves the recursion principles [25], allowing greater flexibility in defining recursive functions on this syntax. Schürmann et. al. have also worked on designing a new calculus for defining recursive functions directly on higher-order syntax [33]. Built-in primitives are provided for the reduction equations for the higher-order case, in contrast to our approach where we define the recursion principle on top of the base level de Bruijn encoding, and prove the reduction equations as lemmas.

Multi-level approaches [14, 23] in Coq and Isabelle, respectively, have been adopted to facilitate reasoning by induction on object-level judgments. We should be able to directly adopt these ideas to create a multi-level version of our system. This kind of approach is inspired by logics such as $FO\lambda^{\Delta\mathbb{N}}$ [22], which was developed specifically to reason using higher-order syntax. In such logics, one can encode an intermediate "specification logic" between the meta-logic and the object language. Negative occurrences in inductive types representing object-level judgments are avoided by using the specification logic instead.

The Twelf system [26], which implements the Logical Framework (LF) has also been used as a framework for reasoning using higher-order syntax. In particular, support for meta-reasoning about object logics expressed in LF has been added [27, 32]. The design of the component for reasoning by induction does not include induction principles for higher-order encodings. Instead, it is based on a realizability interpretation of proof terms. The Twelf implementation of this

approach includes powerful automated support for inductive proofs such as as termination and coverage checkers.

Weak Higher-Order Abstract Syntax (WHOAS) tries to solve the problems of HOAS by turning the negative occurrences of the type of terms in the definition of a data-type into a parameter. In the case of the λ-calculus, the abstraction operator has type:

$$\mathsf{abs} : (\mathsf{Var} \to \Lambda) \to \Lambda.$$

where Var is a type parameter. This approach was introduced by Despeyroux, Felty, and Hirschowitz [13]. Exotic terms were discussed and a predicate was defined to factor them out. Honsell, Miculan, and Scagnetto [20] use a WHOAS approach; they have considered a variety of examples and developed a Theory of Contexts to aid reasoning about variables. A drawback of this approach is that it needs to assume axiomatically several properties of Var.

Gabbay and Pitts [16, 28] define a variant of classical set theory that includes primitives for variable renaming and variable freshness, and a new "freshness quantifier." Using this set theory, it is possible to prove properties by structural induction and also to define functions by recursion over syntax. This approach has been used by Urban and others to solve unification problems [36] and to formalize results on the λ-calculus in Isabelle/HOL [37].

Every object of a nominal set is associated with a *support*, a set of *atoms* (variable names) which generalizes the notion of the set of free variables of a term. There is a notion of swapping of atoms: $(a\,b) \cdot t$ intuitively interchanges the free occurrences of the atoms a and b in t. This notion can be extended to standard type constructors like pairs and functions. In particular, if we have a notion of swapping for nominal sets A and B, we can define swapping for the type of functions $A \to B$: $(a\,b) \cdot f = \lambda x.(a\,b) \cdot (f\,(a\,b) \cdot x)$. Once swapping is defined, the support of an object (its free variables) can be defined as:

$$\mathsf{support}\, t = \{a \mid \{b \mid (a\,b) \cdot t \neq t\} \text{ is infinite}\}.$$

Therefore it is possible to define a notion of free variable also for functions. The freshness relation $a \# t$ expresses the fact that $a \notin \mathsf{support}\, t$ (a is a fresh variable for t). The fact that these definitions generalize to function and product types allows the authors to impose freshness conditions on the recursive definition of functions on the syntax, thus guaranteeing the preservation of α-equality. The possibility of defining a set of free variables for a function is of interest for our work. However, this method is not constructive and cannot be used to generate effectively a fresh variable for a function. For this reason we are forced, in our formalization, to adopt a different, less elegant but computable solution. The nominal approach has been implemented to give representations of object languages with easy recursion principles on α-equivalence classes in Isabelle/HOL [35] and in Coq [3].

Schürmann, Despeyroux, and Pfenning [31] develop a modal meta-theory that allows the formalization of higher-order abstract syntax with a primitive recursive principle. They introduce a modal operator $\Box$. Intuitively, for every type A there is a type $\Box A$ of *closed* objects of type A. Besides the regular function type $A \to B$, there is a more restricted type $A \Rightarrow B = \Box A \to B$ of uniform functions. Functions used as arguments for higher-order constructors are of this kind. For

example, in formalizing the pure λ-calculus, the abstraction operator has type

$$\mathsf{abs} : (\Lambda \Rightarrow \Lambda) \to \Lambda.$$

A structural recursion principle is provided. It can be used to define functions of the regular type $\Lambda \to B$. On the other hand, we are not allowed to use structural recursion to define a function of type $\Box\Lambda \to B$. This avoids the usual paradoxes associated with recursion for types with non-positive occurrences in their definition. Intuitively, we can explain the method as follows: While defining a type A, we cannot assume knowledge of the type as a whole. Think of $\Box A$ as a non-completed version of A, that is, a type that contains some elements of A but may still be extended in the future. Since $\Box A$ is not complete, we are not allowed to do recursion on it. If a constructor requires an argument that is a function of A, we must use $\Box A$, because the function should be compatible with future extensions.

§5. Conclusion. We have developed an approach to reasoning using higher-order abstract syntax which is built on a formalization of a higher-order universal algebra with bindings. This approach generalizes the Hybrid approach where an underlying de Bruijn notation is used. Higher-order syntax encodings are defined in such a way that expanding definitions results in the low-level de Bruijn representation of terms. Reasoning, however, is carried out at the level of higher-order syntax, allowing details of the lower-level implementation to be hidden. In our generalized version, an object language is defined by simply giving its signature, and the resulting tools for reasoning about the object language, such as a higher-order notation, induction principles, and recursion principles are directly available by simply instantiating the general theorems.

Future work includes considering a variety of object languages and completing more extensive proofs. In our earlier work [8], we expressed induction and recursion principles more directly for each object language, but proving them was not just simple instantiation and instead required some proof effort. In that setting, we illustrated the approach with examples showing that reasoning about object languages was direct and simple. In the new setting, after instantiating our general induction and recursion theorems, we expect the reasoning to be equally direct and simple. We also plan to apply our approach to more complex examples such as the PoplMark challenge [4].

Acknowledgment. The authors would like to thank Alberto Momigliano for useful discussions on Hybrid. The work described here is supported in part by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] *The LEGO proof assistant*, www.dcs.ed.ac.uk/home/lego/, 2001.

[2] Simon J. Ambler, Roy L. Crole, and Alberto Momigliano, *Combining higher order abstract syntax with tactical theorem proving and (co)induction*, ***Fifteenth international conference on theorem proving in higher-order logics***, Lecture Notes in Computer Science, vol. 2410, Springer-Verlag, 2002, pp. 13–30.

[3] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich, *Nominal reasoning techniques in Coq*, ***International workshop on logical frameworks and meta-languages: Theory and practice***, 2006, To appear in Electronic Notes in Theoretical Computer Science.

[4] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic, *Mechanized metatheory for the masses: The* poplmark *challenge*, ***Eighteenth international conference on theorem proving in higher-order logics***, Lecture Notes in Computer Science, vol. 3605, Springer-Verlag, 2005, http://fling-l.seas.upenn.edu/∼plclub/cgi-bin/poplmark/index.php?title=The_POPLmark_Challenge, pp. 50–65.

[5] Henk Barendregt, *Lambda calculi with types*, ***Handbook of logic in computer science, volume 2*** (S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors), Oxford University Press, 1992, pp. 117–309.

[6] Yves Bertot and Pierre Castéran, ***Interactive theorem proving and program development. coq'art: The calculus of inductive constructions***, Springer, 2004.

[7] Venanzio Capretta, *Abstraction and computation*, ***Ph.D. thesis***, Computing Science Institute, University of Nijmegen, 2002.

[8] Venanzio Capretta and Amy P. Felty, *Combining de Bruijn indices and higher-order abstract syntax in Coq*, ***Proceedings of types 2006***, Lecture Notes in Computer Science, Springer-Verlag, 2007, To appear.

[9] Coq Development Team, LogiCal Project, *The Coq Proof Assistant reference manual: Version 8.0*, ***Technical report***, INRIA, 2006.

[10] Thierry Coquand and Gérard Huet, *The Calculus of Constructions*, ***Information and Computation***, vol. 76 (1988), pp. 95–120.

[11] Roy L. Crole, *A Representational Adequacy Result for Hybrid*, Submitted for journal publication, April 2006.

[12] N. G. de Bruijn, *Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem*, ***Indagationes Mathematicae***, vol. 34 (1972), pp. 381–392.

[13] Joëlle Despeyroux, Amy Felty, and André Hirschowitz, *Higher-order abstract syntax in Coq*, ***Second international conference on typed lambda calculi and applications***, Lecture Notes in Computer Science, vol. 902, Springer-Verlag, April 1995, pp. 124–138.

[14] Amy P. Felty, *Two-level meta-reasoning in Coq*, ***Fifteenth international conference on theorem proving in higher-order logics***, Lecture Notes in Computer Science, Springer-Verlag, August 2002, pp. 198–213.

[15] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi, *Abstract syntax and variable binding.*, ***Fourteenth annual IEEE symposium on logic in computer science***, IEEE-Computer Society, 1999, pp. 193–202.

[16] Murdoch J. Gabbay and Andrew M. Pitts, *A new approach to abstract syntax with variable binding*, ***Formal Aspects of Computing***, vol. 13 (2001), pp. 341–363.

[17] Andrew D. Gordon, *A mechanisation of name-carrying syntax up to alpha-conversion*, ***Higher-order logic theorem proving and its applications***, Lecture Notes in Computer Science, vol. 780, Springer-Verlag, 1993, pp. 414–426.

[18] Andrew D. Gordon and Tom Melham, *Five axioms of alpha-conversion*, ***9th international conference on higher-order logic theorem proving and its applications***, Lecture Notes in Computer Science, vol. 1125, Springer-Verlag, 1996, pp. 173–191.

[19] Michael J. C. Gordon and Tom F. Melham, ***Introduction to HOL: A theorem proving environment for higher-order logic***, Cambridge University Press, 1993.

[20] Furio Honsell, Marino Miculan, and Ivan Scagnetto, *An axiomatic approach to metareasoning on nominal algebras in HOAS*, ***28th international colloquium on automata, languages and programming***, Lecture Notes in Computer Science, vol. 2076, Springer Verlag, 2001, pp. 963–978.

[21] Zhaohui Luo, ***Computation and reasoning: A type theory for computer science***, International Series of Monographs on Computer Science, vol. 11, Oxford University Press, 1994.

[22] Raymond McDowell and Dale Miller, *Reasoning with higher-order abstract syntax in a logical framework*, ***ACM Transactions on Computational Logic***, vol. 3 (2002), no. 1, pp. 80–136.

[23] Alberto Momigliano and Simon J. Ambler, *Multi-level meta-reasoning with higher-order abstract syntax*, ***Sixth international conference on foundations of software science and computational structures***, Lecture Notes in Artificial Intelligence, vol. 2620, Springer-Verlag, 2003, pp. 375–391.

[24] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel, ***Isabelle/HOL: A proof assistant for higher-order logic***, Lecture Notes in Computer Science, vol. 2283, Springer Verlag, 2002.

[25] Michael Norrish, *Recursive function definition for types with binders*, ***Seventeenth international conference on theorem proving in higher order logics***, Lecture Notes in Computer Science, vol. 3223, Springer-Verlag, 2004, pp. 241–256.

[26] Frank Pfenning and Carsten Schürmann, *System description: Twelf — a meta-logical framework for deductive systems*, ***Sixteenth international conference on automated deduction***, Lecture Notes in Computer Science, vol. 1632, Springer-Verlag, 1999, pp. 202–206.

[27] Brigitte Pientka, *Verifying termination and reduction properties about higher-order logic programs*, ***Journal of Automated Reasoning***, vol. 34 (2005), no. 2, pp. 179–207.

[28] Andrew M. Pitts, *Alpha-structural recursion and induction*, ***Journal of the Association of Computing Machinery***, vol. 53 (2006), no. 3, pp. 459–506.

[29] Gordon D. Plotkin, *An illative theory of relations*, ***Situation theory and its applications, volume 1*** (Robin Cooper, Kuniaki Mukai, and John Perry, editors), CSLI, 1990, pp. 133–146.

[30] Randy Pollack, *Reasoning about languages with binding*, Presentation available at http://homepages.inf.ed.ac.uk/rap/export/bindingChallenge_slides.pdf, 2006.

[31] Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning, *Primitive recursion for higher-order abstract syntax*, ***Theoretical Computer Science***, vol. 266 (2001), pp. 1–57.

[32] Carsten Schürmann and Frank Pfenning, *A coverage checking algorithm for LF*, ***Sixteenth international conference on theorem proving in higher order logics***, Lecture Notes in Computer Science, vol. 2758, Springer-Verlag, 2003, pp. 120–135.

[33] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat, *The $\triangledown$-calculus. functional programming with higher-order encodings*, ***Seventh international conference on typed lambda calculi and applications***, Lecture Notes in Computer Science, vol. 3461, Springer-Verlag, 2005, pp. 339–353.

[34] Morten Heine B. Sørensen and P. Urzyczyn, ***Lectures on the Curry-Howard isomorphism***, Elsevier Science, 2006.

[35] Christian Urban and Stefan Berghofer, *A recursion combinator for nominal datatypes implemented in Isabelle/HOL*, ***Third international joint conference on automated reasoning***, Lecture Notes in Computer Science, vol. 4130, Springer-Verlag, 2006, pp. 498–512.

[36] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay, *Nominal unification*, ***Theoretical Computer Science***, vol. 323 (2004), pp. 473–497.

[37] Christian Urban and Christine Tasson, *Nominal techniques in Isabelle/HOL*, ***Twentieth international conference on automated deduction***, Lecture Notes in Computer Science, vol. 3632, Springer-Verlag, 2005, pp. 38–53.

[38] Benjamin Werner, *Méta-théorie du calcul des constructions inductives*, ***Ph.D. thesis***, Université Paris 7, 1994.

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING
AND DEPARTMENT OF MATHEMATICS AND STATISTICS
UNIVERSITY OF OTTAWA, CANADA

E-mail: venanzio@cs.ru.nl, afelty@site.uottawa.ca

Current address of first author: Computer Science Institute (iCIS), Radboud University Nijmegen, The Netherlands.