Combining de Bruijn Indices and Higher-Order Abstract Syntax in Coq ⋆

Venanzio Capretta⋆⋆ and Amy P. Felty

School of Information Technology and Engineering
and Department of Mathematics and Statistics
University of Ottawa, Canada
venanzio@cs.ru.nl,afelty@site.uottawa.ca

Abstract. The use of *higher-order abstract syntax* is an important approach for the representation of binding constructs in encodings of languages and logics in a logical framework. Formal meta-reasoning about such object languages is a particular challenge. We present a mechanism for such reasoning, formalized in Coq, inspired by the Hybrid tool in Isabelle. At the base level, we define a de Bruijn representation of terms with basic operations and a reasoning framework. At a higher level, we can represent languages and reason about them using higher-order syntax. We take advantage of Coq's constructive logic by formulating many definitions as Coq programs. We illustrate the method on two examples: the untyped lambda calculus and quantified propositional logic. For each language, we can define recursion and induction principles that work directly on the higher-order syntax.

1 Introduction

There are well-known challenges in reasoning within a logical framework about languages encoded using higher-order syntax to represent binding constructs. To illustrate, consider a simple example of an object language – the untyped λ-calculus – encoded in a typed meta-language. We encode λ-terms, in higher-order syntax, by a type term with constructors: abs of type $(\underline{\mathsf{term}} \to \mathsf{term}) \to \mathsf{term}$ and app of type $\mathsf{term} \to \mathsf{term} \to \mathsf{term}$. We represent binding by negative occurrences of the defined type. (Here, the single negative occurrence is underlined.) The Coq system [3, 4] implements the Calculus of Inductive Constructions (CIC) [5, 27]: Like many other systems, it does not allow negative occurrences in constructors of inductive types.

Our approach realizes higher-order syntax encodings of terms with an underlying de Bruijn representation [6]. De Bruijn syntax has two advantages: α-convertibility is just equality and there is no variable capture in substitution.

⋆ In *Proceedings of TYPES 2006 (Conference of the European Types Project),* ©*Springer-Verlag.*

⋆⋆ now at the Computer Science Institute (iCIS), Radboud University Nijmegen, The Netherlands.

A main advantage of higher-order syntax is that it allows substitution by function application at the meta-level. We define higher-order syntax encodings on top of the base level so that they expand to de Bruijn terms.

We provide libraries of operations and lemmas to reason on the higher-order syntax, hiding the details of the de Bruijn representation. This approach is inspired by the Hybrid system [1], implemented in Isabelle [18]. The general structure is the same, but our basic definitions and operators to build a higher level on top of de Bruijn terms are quite different.

Coq's constructive logic allows us to define operators as functions, rather than relations as in Hybrid. This simplifies some of the reasoning and provides more flexibility in specifying object languages. We obtain new induction principles to reason directly on the higher-order syntax, as well as non-dependent recursion principles to define programs on the higher-order syntax.

Our framework includes two parts. The first part is a general library of definitions and lemmas used by any object language. It includes the definition of the de Bruijn representation and of several recursive functions on de Bruijn terms, e.g., substitution. The second part is a methodology to instantiate the library to a particular object language. It includes definitions and lemmas that follow a general pattern and can easily be adapted from one object language to the next. An important result is the validation of induction and recursion principles on the higher-order syntax of the language.

We illustrate our framework on two examples, the untyped λ-calculus (LC) and quantified propositional logic (QPL); the same languages were implemented in Hybrid [1]. For example, we give a Coq function computing the negation normal form of formulas in QPL. This definition uses the recursion principle for the higher-order syntax of QPL. The proof that it does indeed produce normal forms is quite simple, making direct use of our induction principle for QPL. In Hybrid, negation normal forms and many other definitions are given as relations instead. In Coq, they are functions and our higher-order recursion principles allow us to provide simpler, more direct proofs.

Section 2 presents the general part of our framework starting with the de Bruijn syntax and Sect. 3 illustrates how we instantiate this general framework to LC. Section 4 uses this instantiation to prove properties of this object language. In Sect. 5, we apply our framework to QPL. In Sect. 6, we discuss related work, and in Sect. 7, we conclude and discuss future work.

We formalized all the results in this paper in the proof assistant Coq. The files of the formalization are available at:
`http://www.site.uottawa.ca/~afelty/coq/types06_coq.html`

2 De Bruijn Syntax

We describe the lowest level formalization of the syntax. We define a generic type of expressions built on a parameter type of constants `con`. Expressions are built from variables and constants through application and abstraction. There are two

kinds of variables, free and bound. Two types, var and bnd, both instantiated as natural numbers, are used for indexing the two kinds.

Bound variables are treated as de Bruijn indices: this notation eliminates the need to specify the name of the abstracted variable. Thus, the abstraction operator is a simple unary constructor. Expressions are an inductive type:

$$\begin{array}{l}\textbf{Inductive}\ \mathsf{expr} : \mathsf{Set} := \\ \quad \mathsf{CON} : \mathsf{con} \to \mathsf{expr} \\ \quad \mathsf{VAR} : \mathsf{var} \to \mathsf{expr} \\ \quad \mathsf{BND} : \mathsf{bnd} \to \mathsf{expr} \\ \quad \mathsf{APP} : \mathsf{expr} \to \mathsf{expr} \to \mathsf{expr} \\ \quad \mathsf{ABS} : \mathsf{expr} \to \mathsf{expr}\end{array}$$

This is the same definition used in Hybrid [1]. The idea is that the variable $(\mathsf{BND}\, i)$ is bound by the i-th occurrence of ABS above it in the syntax tree. In the following example, we underline all the variable occurrences bound by the first ABS:

$$\underline{\mathsf{ABS}}\,(\mathsf{APP}\,(\mathsf{ABS}\,(\mathsf{APP}\,(\underline{\mathsf{BND}\,1})\,(\mathsf{BND}\,0)))\,(\underline{\mathsf{BND}\,0})).$$

Written in the usual λ-calculus notation, this expression would be $\lambda x.(\lambda y.x\, y)\, x$.

There may occur BND variables with indices higher than the total number of ABSs above them. These are called *dangling* variables. They should not occur in correct terms, but we need to handle them in higher-order binding.

Substitution can be defined for both kinds of variables, but we need it only for the BND variables (substitution of VAR variables can be obtained by first swapping the variable with a fresh BND variable, as shown below, and then substituting the latter).

If j is a de Bruijn index, we define the term $e[j/x]$ (written as `(bsubst e j x)` in the Coq code), obtained by substituting the dangling $\mathsf{BND}\, j$ variable with x. This operation is slightly complicated by the fact that the identity of a BND variable depends on how many ABSs are above it: Every time we go under an ABS, we need to increment the index of the substituted variable and of all the dangling variables in x. In ordinary λ-calculus notation we have $((\lambda y.x\, y)\, x)[x/y] = (\lambda y'.y\, y')\, y$.[1] In de Bruijn syntax, we have, using $\mathsf{BND}\, 0$ for x and $\mathsf{BND}\, 1$ for y:

$$\begin{array}{c}(\mathsf{APP}\,(\mathsf{ABS}\,(\mathsf{APP}\,\underline{(\mathsf{BND}\,1)}\,(\mathsf{BND}\,0)))\,\underline{(\mathsf{BND}\,0)})[0/\mathsf{BND}\,1] \\ = (\mathsf{APP}\,(\mathsf{ABS}\,(\mathsf{APP}\,\underline{(\mathsf{BND}\,2)}\,(\mathsf{BND}\,0)))\,\underline{(\mathsf{BND}\,1)})\end{array}$$

The two underlined occurrences of $\mathsf{BND}\,1$ and $\mathsf{BND}\,0$ on the left-hand side represent the same dangling variable, the one with index 0 at the top level. Similarly, the two occurrences of $\mathsf{BND}\,2$ and $\mathsf{BND}\,1$ on the right-hand side represent the same dangling variable, the one with index 1 at the top level.

[1] Here, y has been renamed y' to avoid capture. Using de Bruijn notation, there is never any variable capture, since renaming is implicit in the syntax.

Call $\hat{x}$ (written as `(bshift x)` in the Coq code) the result of incrementing by one all the dangling variables in x. We define:

$$\begin{aligned}
&(\mathsf{CON}\, c)[j/x] = \mathsf{CON}\, c\\
&(\mathsf{VAR}\, v)[j/x] = \mathsf{VAR}\, v\\
&(\mathsf{BND}\, i)[j/x] = \mathbf{if}\ j = i\ \mathbf{then}\ x\ \mathbf{else}\ \mathsf{BND}\, i\\
&(\mathsf{APP}\, e_1\, e_2)[j/x] = \mathsf{APP}\, (e_1[j/x])\, (e_2[j/x])\\
&(\mathsf{ABS}\, e)[j/x] = \mathsf{ABS}\, (e[j+1/\hat{x}])
\end{aligned}$$

To define binding operators, we need to turn a free variable into a bound one. If e is an expression, j a bound variable, and v a free variable, then we denote by $e[j \leftrightarrow v]$ (written as `(ebind v j e)` in the Coq code) the result of swapping $\mathsf{BND}\, j$ with $\mathsf{VAR}\, v$ in e, taking into account the change in indexing caused by the occurrences of ABS:

$$\begin{aligned}
&(\mathsf{CON}\, c)[j \leftrightarrow v] = \mathsf{CON}\, c\\
&(\mathsf{VAR}\, w)[j \leftrightarrow v] = \mathbf{if}\ w = v\ \mathbf{then}\ \mathsf{BND}\, j\ \mathbf{else}\ \mathsf{VAR}\, w\\
&(\mathsf{BND}\, i)[j \leftrightarrow v] = \mathbf{if}\ i = j\ \mathbf{then}\ \mathsf{VAR}\, v\ \mathbf{else}\ \mathsf{BND}\, i\\
&(\mathsf{APP}\, e_1\, e_2)[j \leftrightarrow v] = \mathsf{APP}\, (e_1[j \leftrightarrow v])\, (e_2[j \leftrightarrow v])\\
&(\mathsf{ABS}\, e)[j \leftrightarrow v] = \mathsf{ABS}\, (e[j+1 \leftrightarrow v])
\end{aligned}$$

We can easily prove that the operation is its own inverse: $e[j \leftrightarrow v][j \leftrightarrow v] = e$.

Finally, the operation $\mathsf{newvar}\, e$ gives the index of the first free variable not occurring in e. Because it is a new variable, if we replace any dangling BND variable with it and then swap the two, we obtain the original term:

Lemma 1. *For* $e : \mathsf{expr}$, $j : \mathsf{bnd}$, *and* $n = \mathsf{newvar}\, e$; $e[j/(\mathsf{VAR}\, n)][j \leftrightarrow n] = e$.

3 Higher-order syntax: The untyped λ-calculus

The first object language that we try to encode is the untyped λ-calculus. Its higher-order definition would be:

$$\begin{aligned}
&\mathbf{Inductive}\ \mathsf{term} : \mathsf{Set} :=\\
&\quad \mathsf{abs} : (\mathsf{term} \to \mathsf{term}) \to \mathsf{term}\\
&\quad \mathsf{app} : \mathsf{term} \to \mathsf{term} \to \mathsf{term}
\end{aligned}$$

This definition is not accepted by Coq, because of the negative occurrence of term in the type of abs. However, we can simulate it on top of the de Bruijn framework. Define lexpr to be the type of expressions obtained by instantiating the type of constants con by the two element type $\mathsf{LCcon} = \{\mathbf{app}, \mathbf{abs}\}$. Not all expressions in lexpr represent valid λ-terms: **app** can be applied only to two expressions and **abs** can be applied only to an abstraction. Therefore there are just two correct forms for a λ-term, besides variables:

$$\mathsf{APP}\, (\mathsf{APP}\, (\mathsf{CON}\, \mathbf{app})\, e_1)\, e_2 \quad \text{and} \quad \mathsf{APP}\, (\mathsf{CON}\, \mathbf{abs})\, (\mathsf{ABS}\, e).^2$$

[2] Do not confuse APP and ABS with **app** and **abs**: the first are constructors for expressions, the second are constants in the syntax of the λ-calculus.

This is easily captured by a boolean function on expressions:

$$
\begin{array}{l}
\mathsf{term_{check}} : \mathsf{lexpr} \to \mathbb{B} \\
\quad \mathsf{term_{check}}\,(\mathsf{VAR}\,v) = \mathsf{true} \\
\quad \mathsf{term_{check}}\,(\mathsf{BND}\,i) = \mathsf{true} \\
\quad \mathsf{term_{check}}\,(\mathsf{APP}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{app})\,e_1)\,e_2) = (\mathsf{term_{check}}\,e_1)\ \mathsf{and}\ (\mathsf{term_{check}}\,e_2) \\
\quad \mathsf{term_{check}}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{abs})\,(\mathsf{ABS}\,e)) = \mathsf{term_{check}}\,e \\
\quad \mathsf{term_{check}}\ \_ = \mathsf{false}
\end{array}
$$

In constructive systems like Coq, a boolean function is not the same as a predicate. However, it can be turned into a predicate Tcheck by $(\mathsf{Tcheck}\,e) = \mathsf{Is\_true}\,(\mathsf{term_{check}}\,e)$, where $\mathsf{Is\_true}\,\mathsf{true} = \mathsf{True}$ and $\mathsf{Is\_true}\,\mathsf{false} = \mathsf{False}$.[3]

Well-formed λ-terms are those expressions satisfying Tcheck. They can be defined in type theory by a record type[4]:

$$
\begin{array}{l}
\mathbf{Record}\ \mathsf{term} := \mathsf{mk\_term} \\
\quad \{\, \mathsf{t\_expr} : \mathsf{lexpr}; \\
\quad\ \ \mathsf{t\_check} : \mathsf{Tcheck}\,\mathsf{t\_expr}\}
\end{array}
$$

The advantage of our definition of Tcheck is that its value is always True for well-formed expressions. This implies that the t_check component of a term must always be I, the only proof of True. This ensures that terms are completely defined by their t_expr component:

Lemma 2 (term unicity). $\forall t_1, t_2 : \mathsf{term}, (\mathsf{t\_expr}\,t_1) = (\mathsf{t\_expr}\,t_2) \to t_1 = t_2$.

Now our aim is to define higher-order syntax for term. It is easy to define notation for variables and application:

$$
\begin{array}{l}
\mathsf{Var}\,v = \mathsf{mk\_term}\,(\mathsf{VAR}\,v)\,\mathsf{I} \\
\mathsf{Bind}\,i = \mathsf{mk\_term}\,(\mathsf{BND}\,i)\,\mathsf{I} \\
t_1 \mathbin{@} t_2 = \mathsf{mk\_term}\,(\mathsf{APP}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{app})\,(\mathsf{t\_expr}\,t_1))\,(\mathsf{t\_expr}\,t_2))\ \bigstar
\end{array}
$$

(The $\bigstar$ symbol stands for a proof of Tcheck that can easily be constructed from $\mathsf{t\_check}\,t_1$ and $\mathsf{t\_check}\,t_2$. In the rest of this paper, we often omit the details of Tcheck proofs and use this symbol instead.)

The crucial problem is the definition of a higher-order notation for abstraction. Let $f : \mathsf{term} \to \mathsf{term}$; we want to define a term $(\mathsf{Fun}\,x, f\,x)$ representing the λ-term $(\lambda x.f\,x)$. The underlying expression of this term must be an abstraction, i.e., it must be in the form $(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{abs})\,(\mathsf{ABS}\,e))$. The idea is that e should be the result of replacing the metavariable x in $f\,x$ by the bound variable $\mathsf{Bind}\,0$. However, the simple solution of applying f to $\mathsf{Bind}\,0$ is incorrect: Different occurrences of the metavariable should be replaced by different de Bruijn indices, according to the number of abstractions above them. The solution is: First apply f to a new free variable $\mathsf{Var}\,n$, and then replace $\mathsf{VAR}\,n$ with

[3] The values true and false are in $\mathbb{B}$ which is a member of Set, while True and False are propositions, i.e., members of Prop.

[4] Here, mk_term is the constructor of term; t_expr and t_check are field names.

$\mathsf{BND}\,0$ in the underlying expression. Formally: $\mathsf{tbind}\,f = \mathsf{t\_expr}\,(f\,(\mathsf{Var}\,n))[0 \leftrightarrow n]$. The proof $\mathsf{tbind\_check}\,f$ of $(\mathsf{Tcheck}\,(\mathsf{tbind}\,f))$ can be constructed from the proof $\mathsf{t\_check}\,(f\,(\mathsf{Var}\,n))$. We can then define a term that we call the *body* of the function f: $\mathsf{tbody}\,f = \mathsf{mk\_term}\,(\mathsf{tbind}\,f)\,(\mathsf{tbind\_check}\,f)$. Finally, we can define the higher-order notation for λ-abstraction:

$$\mathsf{Fun}\,x, f\,x = \mathsf{mk\_term}(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{abs})\,(\mathsf{ABS}\,(\mathsf{tbind}\,(\lambda x, f\,x)))) \bigstar$$

We must clarify what it means for n to be a new variable for a function $f : \mathsf{term} \to \mathsf{term}$. In Coq, the function space $\mathsf{term} \to \mathsf{term}$ includes meta-terms that do not encode terms of the object language LC, often called *exotic terms* (see [7]). Functions that do encode terms are those that work uniformly on all arguments. Since we do not require uniformity, $(f\,x)$ may have a different set of free variables for each argument x. It is in general not possible to find a variable that is new for all the results. We could have, e.g., $f\,x = \mathsf{Var}\,(\mathsf{size}\,x)$ where $(\mathsf{size}\,x)$ is the total number of variables and constants occurring in x. Only if f is uniform, e.g., if $f\,x = (\mathsf{Var}\,1)\,@\,(x\,@\,(\mathsf{Var}\,0))$, we can determine objectively an authentic free variable, in this case $n = 2$. For the general case, we simply define n to be a free variable for $(f\,(\mathsf{Bind}\,0))$. This definition gives an authentic free variable when f is uniform.

To prove some results, we must require that functions are uniform, so we must define this notion formally. Intuitively, uniformity means that all values $(f\,x)$ are defined *by the same expression*. We say that f *is an abstraction* if this happens. We already defined the body of f, $(\mathsf{tbody}\,f)$. We now state that f is an abstraction if, for every term x, $(f\,x)$ is obtained by *applying* the body of f to x. We define the result of applying a function body to an argument by the use of the operation of substitution of bound variables.

$$\begin{array}{l}\text{If } t = \mathsf{mk\_term}\,e_t\,h_t \text{ and } x = \mathsf{mk\_term}\,e_x\,h_x,\\ \text{then } \mathsf{tapp}\,t\,x = \mathsf{mk\_term}\,(e_t[0/e_x]) \bigstar\end{array}$$

We can now link the higher-order abstraction operator Fun to the application of the constant **abs** at the de Bruijn level in an exact sense.

Lemma 3. *For all* $e : \mathsf{t\_expr}$ *and* $h : \mathsf{Tcheck}\,e$*, we have*

$$\mathsf{Fun}\,x, \mathsf{tapp}\,(\mathsf{mk\_term}\,e\,h)\,x = \mathsf{mk\_term}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{abs})\,(\mathsf{ABS}\,e)) \bigstar$$

Proof. Unfold the definitions of Fun, tbind and tapp and then use Lemma 1.

The body of the application of a term is always the term itself. We define a function to be an abstraction if it is equal to the application of its body.

Lemma 4. $\forall t : \mathsf{term}, t = \mathsf{tbody}\,(\mathsf{tapp}\,t)$

Definition 1. *Given a function* $f : \mathsf{term} \to \mathsf{term}$*, we define its* canonical form *as* $\mathsf{funt}\,f = \mathsf{tapp}\,(\mathsf{tbody}\,f) : \mathsf{term} \to \mathsf{term}$*. We say that* f *is an* abstraction*,* $\mathsf{is\_abst}\,f$*, if* $\forall x, f\,x = \mathsf{funt}\,f\,x$*.*

Some definitions and results will hold only in the case that the function is an abstraction. For example, if we want to formalize β-reduction, we should add such a hypothesis: $\mathsf{is\_abst}\, f \;\rightarrow\; ((\mathsf{Fun}\, x, f\, x) \;@\; t) \;\rightsquigarrow_\beta\; f\, t$. This assumption does not appear in informal reasoning, so we would like it to be automatically provable. It would be relatively easy to define a tactic to dispose of such hypotheses mechanically. A different solution is to use always the canonical form in place of the bare function. In the case of β-reduction we would write: $((\mathsf{Fun}\, x, f\, x) @ t) \rightsquigarrow_\beta (\mathsf{funt}\, f)\, t$. Note that if f happens to be an abstraction, then $(\mathsf{funt}\, f)\, t$ and $f\, t$ are convertible at the meta-level, so the two formulations are equivalent and in the second one we are exempted from proving the uniformity of f. In Sect. 4 we follow Hybrid in adopting the first definition, but using the second one would be equally easy.

Coq provides an automatic induction principle on expressions. It would be more convenient to have an induction principle tailored to the higher-order syntax of terms. The first step in this direction is an induction principle on expressions satisfying Tcheck:

Theorem 1 (Induction on well-formed expressions). *Let* $P : \mathsf{lexpr} \rightarrow \mathsf{Prop}$ *be a predicate on expressions such that the following hypotheses hold:*

$$\begin{array}{l}
\forall v : \mathsf{var}, P\,(\mathsf{VAR}\, v) \\
\forall i : \mathsf{bnd}, P\,(\mathsf{BND}\, i) \\
\forall e_1, e_2 : \mathsf{lexpr}, P\, e_1 \rightarrow P\, e_2 \rightarrow P\,(\mathsf{APP}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{app})\, e_1)\, e_2) \\
\forall e : \mathsf{lexpr}, P\, e \rightarrow P\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{abs})\,(\mathsf{ABS}\, e))
\end{array}$$

Then $(P\, e)$ *is true for every* $e : \mathsf{lexpr}$ *such that* $(\mathsf{Tcheck}\, e)$ *holds.*

Proof. By induction on the structure of e. The assumptions provide us with derivations of $(P\, e)$ from the inductive hypotheses that P holds for all subterms of e satisfying Tcheck, but only if e is in one of the four allowed forms. If e is in a different form, the result is obtained by *reductio ad absurdum* from the assumption $(\mathsf{Tcheck}\, e) = \mathsf{False}$. To apply the induction hypotheses to subterms, we need a proof that Tcheck holds for them. This is easily derivable from $(\mathsf{Tcheck}\, e)$.

This induction principle was used to prove several results about terms. A fully higher-order induction principle can be derived from it.

Theorem 2 (Induction on terms). *Let* $P : \mathsf{term} \rightarrow \mathsf{Prop}$ *be a predicate on terms such that the following hypotheses hold:*

$$\begin{array}{l}
\forall v : \mathsf{var}, P\,(\mathsf{Var}\, v) \\
\forall i : \mathsf{bnd}, P\,(\mathsf{Bind}\, i) \\
\forall t_1, t_2 : \mathsf{term}, P\, t_1 \rightarrow P\, t_2 \rightarrow P\,(t_1 \;@\; t_2) \\
\forall f : \mathsf{term} \rightarrow \mathsf{term}, P\,(\mathsf{tbody}\,(\lambda x, f\, x)) \rightarrow P\,(\mathsf{Fun}\, x, f\, x)
\end{array}$$

Then $(P\, t)$ *is true for every* $t : \mathsf{term}$.[5]

[5] The induction hypothesis is formulated for $(\mathsf{tbody}\,(\lambda x, f\, x))$ rather than for $(\mathsf{tbody}\, f)$ because, in Coq, extensionally equal functions are not always provably equal, so we may need to use their η-expansion.

Proof. Since t must be in the form $t = (\mathsf{mk\_term}\, e\, h)$ where $e : \mathsf{lexpr}$ and $h : (\mathsf{Tcheck}\, e)$, we apply Theorem 1. This requires solving two problems.

First, P is a predicate on terms, while Theorem 1 requires a predicate on expressions. We define it by: $\bar{P}\, e = \forall h : (\mathsf{Tcheck}\, e), P\, (\mathsf{mk\_term}\, e\, h)$.

Second, we have to prove the four assumptions about $\bar{P}$ of Theorem 1. The first three are easily derived from the corresponding assumptions in the statement of this theorem. The fourth requires a little more work. We need to prove that $\forall e : \mathsf{lexpr}, \bar{P}\, e \to \bar{P}\, (\mathsf{APP}\, (\mathsf{CON}\, \mathbf{abs})\, (\mathsf{ABS}\, e))$. Let then e be an expression and assume $(\bar{P}\, e) = \forall h : (\mathsf{Tcheck}\, e), P\, (\mathsf{mk\_term}\, e\, h)$ holds. The conclusion of the statement is unfolded to:

$$\forall h : (\mathsf{Tcheck}\, e), P\, (\mathsf{mk\_term}\, (\mathsf{APP}\, (\mathsf{CON}\, \mathbf{abs})\, (\mathsf{ABS}\, e))\, h).$$

By Lemma 3, the above expression has a higher-order equivalent:

$$(\mathsf{mk\_term}\, (\mathsf{APP}\, (\mathsf{CON}\, \mathbf{abs})\, (\mathsf{ABS}\, e))\, h) = \mathsf{Fun}\, x, \mathsf{tapp}\, (\mathsf{mk\_term}\, e\, \bigstar)\, x.$$

By the fourth assumption in the statement, P holds for this term if it holds for $(\mathsf{tbody}\, (\lambda x, \mathsf{tapp}\, (\mathsf{mk\_term}\, e\, \bigstar)\, x))$, which is $(\mathsf{tbody}\, (\mathsf{tapp}\, (\mathsf{mk\_term}\, e\, \bigstar)))$ by extensionality of tbody. This follows from $(P\, (\mathsf{mk\_term}\, e\, \bigstar))$ by Lemma 4, and this last proposition holds by assumption, so we are done.

In Sect. 5 we give a similar induction principle for Quantified Propositional Logic and explain its use on an example.

As stated earlier, although inspired by Hybrid [1], our basic definitions and operators to build the higher-order level on top of de Bruijn terms are quite different. For example, our higher-order notation for λ-abstraction $(\mathsf{Fun}\, x, f\, x)$ is defined only on well-formed terms. In Hybrid, the corresponding lambda is defined on all expressions. A predicate is defined identifying all non-well-formed terms, which are mapped to a default expression. Hybrid also defines an induction principle similar to our Theorem 1. Here we go a step further and show that a fully higher-order induction principle can be derived from it (Theorem 2). In Sect. 5, we also provide a non-dependent recursion principle (Theorem 4).

4 Lazy Evaluation of Untyped λ-Terms

Using the higher-order syntax from the previous section, we give a direct definition of lazy evaluation of closed λ-terms as an inductive relation:

$$\begin{array}{l}
\mathbf{Inductive}\ \Downarrow : \mathsf{term} \to \mathsf{term} \to \mathsf{Prop} := \\
\quad \forall f : \mathsf{term} \to \mathsf{term}, (\mathsf{Fun}\, x, f\, x) \Downarrow (\mathsf{Fun}\, x, f\, x) \\
\quad \forall e_1, e_2, v : \mathsf{term}, \forall f : \mathsf{term} \to \mathsf{term}, \\
\qquad \mathsf{is_abst}\, f \to e_1 \Downarrow\ (\mathsf{Fun}\, x, f\, x) \to (f\, e_2) \Downarrow v \to (e_1 \mathbin{@} e_2) \Downarrow v
\end{array}$$

Notice that in the second rule, expressing β-reduction, substitution is obtained simply by higher-order application: $(f\, e_2)$.

By a straightforward induction on this definition, we can prove that evaluation is a functional relation. The uniformity hypothesis ($\mathsf{is\_abst}\ f$) in the application case is important (see Definition 1); without it, the property does not hold. In particular, the proof uses the following lemma, which holds only for functions which satisfy the is_abst predicate:

Lemma 5 (Fun injectivity). *Let* $f_1, f_2 : \mathsf{term} \to \mathsf{term}$ *such that* $\mathsf{is\_abst}\ f_1$ *and* $\mathsf{is\_abst}\ f_2$*; if* $(\mathsf{Fun}\, x, f_1\, x) = (\mathsf{Fun}\, x, f_2\, x)$*, then* $\forall x : \mathsf{term}, (f_1\, x) = (f_2\, x)$.

Our main theorem follows by direct structural induction on the proof of the evaluation relation.

Theorem 3 (Unique values). *Let* $e, v_1, v_2 : \mathsf{term}$*; if both* $e \Downarrow v_1$ *and* $e \Downarrow v_2$ *hold, then* $v_1 = v_2$.

5 Quantified Propositional Logic

Our second example of an object language, Quantified Propositional Logic (QPL), is also inspired by the Hybrid system [1]. The informal higher-order syntax representation of QPL would be the following:

Inductive formula : Set :=
  Not : formula → formula
  Imp : formula → formula → formula
  And : formula → formula → formula
  Or : formula → formula → formula
  All : (formula → formula) → formula
  Ex : (formula → formula) → formula

As for the case of LC, this definition is not acceptable in type theory because of the negative occurrences of formula in the types of the two quantifiers.

As in Sect. 3, we instantiate the type of de Bruijn expressions with the type of constants $\mathsf{QPLcon} = \{\mathbf{not}, \mathbf{imp}, \mathbf{and}, \mathbf{or}, \mathbf{all}, \mathbf{ex}\}$. We call oo the type of expressions built on these constants.

Similarly to the definition of $\mathsf{term}_{\mathsf{check}}$ and Tcheck, we define a boolean function $\mathsf{formula}_{\mathsf{check}}$ and a predicate Fcheck to restrict the well-formed expressions to those in one of the following forms:

$\mathsf{VAR}\, v$	$\mathsf{APP}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{and})\, e_1)\, e_2$
$\mathsf{BND}\, i$	$\mathsf{APP}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{or})\, e_1)\, e_2$
$\mathsf{APP}\,(\mathsf{CON}\,\mathbf{not})\, e$	$\mathsf{APP}\,(\mathsf{CON}\,\mathbf{all})\,(\mathsf{ABS}\, e)$
$\mathsf{APP}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{imp})\, e_1)\, e_2$	$\mathsf{APP}\,(\mathsf{CON}\,\mathbf{ex})\,(\mathsf{ABS}\, e)$

Then the type of formulas can be defined by:

Record formula := mk_formula
  { f_expr : oo;
    f_check : Fcheck f_expr}

The higher-order syntax of QPL is defined similarly to that of LC:

$$\begin{aligned}
&\mathsf{Var}\, v = \mathsf{mk_formula}\,(\mathsf{VAR}\, v)\,\mathsf{I}\\
&\mathsf{Bind}\, i = \mathsf{mk_formula}\,(\mathsf{BND}\, i)\,\mathsf{I}\\
&\mathsf{Not}\, a = \mathsf{mk_formula}\,(\mathsf{APP}\,\mathbf{not}\,(\mathsf{f_expr}\, a)) \bigstar\\
&a_1\ \mathsf{Imp}\ a_2 = \mathsf{mk_formula}\,(\mathsf{APP}\,(\mathsf{APP}\,\mathbf{imp}\,(\mathsf{f_expr}\, a_1))\,(\mathsf{f_expr}\, a_2)) \bigstar\\
&a_1\ \mathsf{And}\ a_2 = \mathsf{mk_formula}\,(\mathsf{APP}\,(\mathsf{APP}\,\mathbf{and}\,(\mathsf{f_expr}\, a_1))\,(\mathsf{f_expr}\, a_2)) \bigstar\\
&a_1\ \mathsf{Or}\ a_2 = \mathsf{mk_formula}\,(\mathsf{APP}\,(\mathsf{APP}\,\mathbf{or}\,(\mathsf{f_expr}\, a_1))\,(\mathsf{f_expr}\, a_2)) \bigstar\\
&\mathsf{All}\, x, f\, x = \mathsf{mk_formula}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{all})\,(\mathsf{ABS}\,(\mathsf{fbind}\,(\lambda x, f\, x)))) \bigstar\\
&\mathsf{Ex}\, x, f\, x = \mathsf{mk_formula}\,(\mathsf{APP}\,(\mathsf{CON}\,\mathbf{ex})\,(\mathsf{ABS}\,(\mathsf{fbind}\,(\lambda x, f\, x)))) \bigstar
\end{aligned}$$

where fbind is defined exactly as tbind in Sect. 3. As in that case, fbind f satisfies Fcheck, giving us the formula fbody f, the body of the function f. As before, the *canonical form* funf f is the application of the body of f (following Definition 1).

For this example, we carry the encoding of higher-order syntax further by defining a non-dependent recursion principle.

Theorem 4. *For any type B, we can define a function of type* formula $\to B$ *by recursively specifying its results on the higher-order form of formulas:*

$$\begin{aligned}
&\mathsf{Hvar} : \mathsf{var} \to B\\
&\mathsf{Hbind} : \mathsf{bnd} \to B\\
&\mathsf{Hnot} : \mathsf{formula} \to B \to B\\
&\mathsf{Himp} : \mathsf{formula} \to \mathsf{formula} \to B \to B \to B\\
&\mathsf{Hand} : \mathsf{formula} \to \mathsf{formula} \to B \to B \to B\\
&\mathsf{Hor} : \mathsf{formula} \to \mathsf{formula} \to B \to B \to B\\
&\mathsf{Hall} : (\mathsf{formula} \to \mathsf{formula}) \to B \to B\\
&\mathsf{Hex} : (\mathsf{formula} \to \mathsf{formula}) \to B \to B
\end{aligned}$$

If $\mathbf{f} = \mathsf{form\_rec}_{\mathsf{Hvar},\mathsf{Hbind},\mathsf{Hnot},\mathsf{Himp},\mathsf{Hand},\mathsf{Hor},\mathsf{Hall},\mathsf{Hex}} : \mathsf{formula} \to B$ *is the function so defined, then the following reduction equations hold:*

$$\begin{aligned}
&\mathbf{f}\,(\mathsf{Var}\, v) = \mathsf{Hvar}\, v && \mathbf{f}\,(a_1\ \mathsf{Imp}\ a_2) = \mathsf{Himp}\, a_1\, a_2\,(\mathbf{f}\, a_1)\,(\mathbf{f}\, a_2)\\
&\mathbf{f}\,(\mathsf{Bind}\, i) = \mathsf{Hbind}\, i && \mathbf{f}\,(a_1\ \mathsf{And}\ a_2) = \mathsf{Hand}\, a_1\, a_2\,(\mathbf{f}\, a_1)\,(\mathbf{f}\, a_2)\\
&\mathbf{f}\,(\mathsf{Not}\, a) = \mathsf{Hnot}\, a\,(\mathbf{f}\, a) && \mathbf{f}\,(a_1\ \mathsf{Or}\ a_2) = \mathsf{Hor}\, a_1\, a_2\,(\mathbf{f}\, a_1)\,(\mathbf{f}\, a_2)\\
&\mathbf{f}\,(\mathsf{All}\, x, f\, x) = \mathsf{Hall}\,(\mathsf{funf}\,(\lambda x.f\, x))\,(\mathbf{f}\,(\mathsf{fbody}\,(\lambda x.f\, x)))\\
&\mathbf{f}\,(\mathsf{Ex}\, x, f\, x) = \mathsf{Hex}\,(\mathsf{funf}\,(\lambda x.f\, x))\,(\mathbf{f}\,(\mathsf{fbody}\,(\lambda x.f\, x)))
\end{aligned}$$

Proof. Let a : formula; it must be of the form (mk_formula $e\, h$). The definition is by recursion on the structure of e. The assumptions give the inductive steps when e is in one of the allowed forms. If e is in a different form, we can give an arbitrary output, since $h : (\mathsf{Fcheck}\, e) = \mathsf{False}$ is absurd.

For the allowed expressions, we transform e into its equivalent higher-order form in the same manner as was done for λ-terms in the proof of Theorem 2. The reduction equations follow from unfolding definitions.

As illustration, we use this recursion principle to define the negation normal form of a formula. Intuitively, (nnf a) recursively moves negations inside connectives and quantifiers, eliminating double negations. For example, here are four

of the several possible cases:

$$\begin{aligned}
&\mathsf{nnf}\,(\mathsf{Not}\,(\mathsf{Not}\,a)) = a\\
&\mathsf{nnf}\,(\mathsf{Not}\,(a_1\ \mathsf{Imp}\ a_2)) = (\mathsf{nnf}\,a_1)\ \mathsf{And}\ (\mathsf{nnf}\,(\mathsf{Not}\,a_2))\\
&\mathsf{nnf}\,(\mathsf{All}\,x, f\,x) = \mathsf{All}\,x, \mathsf{fapp}\,(\mathsf{nnf}\,(\mathsf{fbody}\,(\lambda x, f\,x)))\,x\\
&\mathsf{nnf}\,(\mathsf{Not}\,(\mathsf{All}\,x, f\,x)) = \mathsf{Ex}\,x, \mathsf{fapp}\,(\mathsf{nnf}\,(\mathsf{Not}\,(\mathsf{fbody}\,(\lambda x, f\,x))))\,x
\end{aligned}$$

Notice, in particular, the way nnf is defined on quantifiers. It would be incorrect to try to define it as $\mathsf{nnf}\,(\mathsf{All}\,x, f\,x) = \mathsf{All}\,x, (\mathsf{nnf}\,(f\,x))$, because this would imply that in order to define nnf on $(\mathsf{All}\,x, x)$, it must already be recursively defined on every x, in particular on $x = (\mathsf{All}\,x, f\,x)$ itself. This definition is circular and therefore incorrect. Instead, we recursively apply nnf only to the body of f and then quantify over the application of the result.

To define nnf formally, we need to give its result simultaneously on a formula a and its negation $(\mathsf{Not}\,a)$. Therefore, we define an auxiliary function $\mathsf{nnf\_aux} : \mathsf{formula} \to \mathsf{formula} \times \mathsf{formula}$ in such a way that $\mathsf{nnf\_aux}\,a = \langle \mathsf{nnf}\,a, \mathsf{nnf}\,(\mathsf{Not}\,a)\rangle$. We apply form_rec with $B := \mathsf{formula} \times \mathsf{formula}$:

$$\begin{aligned}
&\mathsf{Hvar}\,v = \langle \mathsf{Var}\,v, \mathsf{Not}\,(\mathsf{Var}\,v)\rangle\\
&\mathsf{Hbind}\,i = \langle \mathsf{Bind}\,i, \mathsf{Not}\,(\mathsf{Bind}\,i)\rangle\\
&\mathsf{Hnot}\,a\,\langle u, v\rangle = \langle v, u\rangle\\
&\mathsf{Himp}\,a_1\,a_2\,\langle u_1, v_1\rangle\,\langle u_2, v_2\rangle = \langle v_1\ \mathsf{Or}\ u_2, u_1\ \mathsf{And}\ v_2\rangle\\
&\mathsf{Hand}\,a_1\,a_2\,\langle u_1, v_1\rangle\,\langle u_2, v_2\rangle = \langle u_1\ \mathsf{And}\ u_2, v_1\ \mathsf{Or}\ v_2\rangle\\
&\mathsf{Hor}\,a_1\,a_2\,\langle u_1, v_1\rangle\,\langle u_2, v_2\rangle = \langle u_1\ \mathsf{Or}\ u_2, v_1\ \mathsf{And}\ v_2\rangle\\
&\mathsf{Hall}\,f\,\langle u, v\rangle = \langle (\mathsf{All}\,x, \mathsf{fapp}\,u\,x), (\mathsf{Ex}\,x, \mathsf{fapp}\,v\,x)\rangle\\
&\mathsf{Hex}\,f\,\langle u, v\rangle = \langle (\mathsf{Ex}\,x, \mathsf{fapp}\,u\,x), (\mathsf{All}\,x, \mathsf{fapp}\,v\,x)\rangle
\end{aligned}$$

and then define $\mathsf{nnf}\,a = \pi_1\,(\mathsf{nnf\_aux}\,a)$. The arguments in the form $\langle u, v\rangle$ represent the result of the recursive calls on the formula and its negation. For example, in the definition of Hnot, u represents $(\mathsf{nnf}\,a)$ and v represents $(\mathsf{nnf}\,(\mathsf{Not}\,a))$. In the definition of Hall, u represents $(\mathsf{nnf}\,(\mathsf{fbody}\,(\lambda x, f\,x)))$ and v represents $\mathsf{nnf}\,(\mathsf{Not}\,(\mathsf{fbody}\,(\lambda x, f\,x)))$.

We also prove an induction principle on formulas, similar to Theorem 2.

Theorem 5 (Induction on formulas). *Let* $P : \mathsf{formula} \to \mathsf{Prop}$ *be a predicate on formulas such that the following hypotheses hold:*

$$\begin{aligned}
&\forall v : \mathsf{var}, P\,(\mathsf{Var}\,v)\\
&\forall i : \mathsf{bnd}, P\,(\mathsf{Bind}\,i)\\
&\forall a : \mathsf{formula}, P\,a \to P\,(\mathsf{Not}\,a)\\
&\forall a_1, a_2 : \mathsf{formula}, P\,a_1 \to P\,a_2 \to P\,(a_1\ \mathsf{Imp}\ a_2)\\
&\forall a_1, a_2 : \mathsf{formula}, P\,a_1 \to P\,a_2 \to P\,(a_1\ \mathsf{And}\ a_2)\\
&\forall a_1, a_2 : \mathsf{formula}, P\,a_1 \to P\,a_2 \to P\,(a_1\ \mathsf{Or}\ a_2)\\
&\forall f : \mathsf{formula} \to \mathsf{formula}, P\,(\mathsf{fbody}\,(\lambda x, f\,x)) \to P\,(\mathsf{All}\,x, f\,x)\\
&\forall f : \mathsf{formula} \to \mathsf{formula}, P\,(\mathsf{fbody}\,(\lambda x, f\,x)) \to P\,(\mathsf{Ex}\,x, f\,x)
\end{aligned}$$

Then $(P\,a)$ *is true for every* $a : \mathsf{formula}$.

As an application of this principle, we defined an inductive predicate is_Nnf stating that a formula is in negation normal form, i.e., negation can occur only on variables, and proved that the result of nnf is always in normal form.

6 Related Work

There is extensive literature on approaches to representing object languages with higher-order syntax and reasoning about them within the same framework. Pollack's notes on the problem of reasoning about binding [22] give a high-level summary of many of them. Some of them were used to solve the POPLmark challenge problem set [2]. We mention a few here.

Several approaches have used Coq. These include the use of *weak higher-order abstract syntax* [7, 14]. In weak higher-order syntax, the problem of negative occurrences in syntax encodings is handled by replacing them by a new type. For example, the abs constructor for the untyped λ-terms introduced in Sect. 1 has type $(\underline{\mathsf{var}} \to \mathsf{term}) \to \mathsf{term}$, where var is a type of variables. Some additional operations are needed to encode and reason about this new type, which at times is inconvenient. Miculan's approach [14, 15] introduces a "theory of contexts" to handle this representation of variables, with extensive use of axioms whose soundness must be justified independently.

McDowell and Miller [12] introduce a new logic specifically designed for reasoning with higher-order syntax. Their logic is intuitionistic and higher-order with support for natural number induction and definitions. In general, higher-order syntax mainly addresses encodings of term-level abstraction. More recent work by Miller and Tiu [16] includes a new quantifier for this style of logic, which provides an elegant way to handle abstractions at the level of proofs. Another approach uses multi-level encodings [8, 17]. This approach also aims to capture more than term-level abstraction, and is inspired by the work of McDowell and Miller but uses Coq and Isabelle, respectively.

Gabbay and Pitts [9] 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.

The Twelf system [21], which implements the Logical Framework (LF) has also been used as a framework for reasoning using higher-order syntax. In particular Schürmann [23] has developed a logic which extends LF with support for meta-reasoning about object logics expressed in LF. 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.

Schürmann, Despeyroux, and Pfenning [24] develop a modal metatheory that allows the formalization of higher-order abstract syntax with a primitive recursion principle. They introduce a modal operator $\Box$. Intuitively, for every type A there is a type $\Box A$ of *closed* objects of type A. In addition to the regular function type $A \to B$, there is a more restricted type $A \Rightarrow B \equiv \Box A \to B$ of uniform functions. Functions used as arguments for higher-order constructors are of this kind. This allows them to define a recursion principle that avoids the usual circularity problems. The system has not yet been extended to a framework with dependent types.

Schürmann et. al. have also worked on designing a new calculus for defining recursive functions directly on higher-order syntax [25]. 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.

Nogin et. al. [19] build a theory in MetaPRL that includes both a higher-order syntax and a de Bruijn representation of terms, with a translation between the two. Induction principles are defined at the de Bruijn level. Their basic library is more extensive than ours; it provides syntactic infrastructure for reflective reasoning and variable-length bindings. Instantiating the basic theory and proving properties about specific object logics is left as future work.

Solutions to the POPLmark challenge also include first-order approaches which adopt de Bruijn representations, such as the one by Stump [26] that uses named bound variables and indices for free variables, and solves part 1a of POPLmark. Another earlier first-order approach by Melham avoids de Bruijn syntax altogether and encodes abstractions using names paired with expressions [13]. Working at this level requires dealing with low-level details about α-conversion, free and bound variables, substitution, etc. Gordon and Melham [11] generalize this name-carrying syntax approach and develop a general theory of untyped λ-terms up to α-conversion, including induction and recursion principles. They illustrate that their theory can be used as a meta-language for representing object languages in such a way that the user is free from concerns of α-conversion. Norrish [20] improves the recursion principles, allowing greater flexibility in defining recursive functions on this syntax. Gordon [10] was able to take a step further in improving the name-carrying syntax approach by defining this kind of syntax in terms of an underlying de Bruijn notation. Gordon's work was the starting point for Hybrid [1], which kept the underlying de Bruijn notation, but used a higher-order representation at the higher-level.

7 Conclusion

We have presented a new method for reasoning in Coq about object languages represented using higher-order syntax. We have shown how to structure proof development so that reasoning about object languages takes place at the level of higher-order syntax, even though the underlying syntax uses de Bruijn notation. An important advantage of our approach is the ability to define induction and recursion principles directly on the higher-order syntax representation of terms. Our examples illustrate the use of this framework for encoding object languages and their properties in a manner that allows direct and simple reasoning.

Future work includes considering a wider variety of object languages and completing more extensive proofs. It also includes adapting the preliminary ideas from Hybrid [1] to show that our representation of the λ-calculus is an adequate one. It is unlikely that these results will carry over directly to our constructive setting, so further work will be required. We also plan to generalize the methodology to define object languages with binding and prove their higher-

order recursion and induction principles: the user should be able to define a new language and reason about it using only higher-order syntax, without having to look at all at the lower de Bruijn level. In a forthcoming article, we develop a higher-order universal algebra in which the user can define a language by giving typing rules for the constants. Higher-order terms and associated induction and recursion principles are then automatically derived.

Acknowledgments

The authors would like to thank Alberto Momigliano for useful comments on an earlier draft of this paper. Thanks also go to the Natural Sciences and Engineering Research Council of Canada for providing support for this work.

References

1. S. J. Ambler, R. L. Crole, and A. Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In *Fifteenth International Conference on Theorem Proving in Higher-Order Logics*, pages 13–30. Springer-Verlag Lecture Notes in Computer Science, Aug. 2002.
2. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Eighteenth International Conference on Theorem Proving in Higher-Order Logics*, pages 50–65. Springer-Verlag Lecture Notes in Computer Science, 2005. http://fling-l.seas.upenn.edu/∼plclub/cgi-bin/-poplmark/index.php?title=The_POPLmark_Challenge.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
4. Coq Development Team, LogiCal Project. The Coq Proof Assistant reference manual: Version 8.0. Technical report, INRIA, 2006.
5. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.
6. N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.
7. J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138. Springer-Verlag Lecture Notes in Computer Science, Apr. 1995.
8. A. P. Felty. Two-level meta-reasoning in Coq. In *Fifteenth International Conference on Theorem Proving in Higher-Order Logics*, pages 198–213. Springer-Verlag Lecture Notes in Computer Science, Aug. 2002.
9. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
10. A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *Sixth International Workshop on Higher-Order Logic Theorem Proving and its Applications*, pages 413–425. Springer-Verlag Lecture Notes in Computer Science, 1993.

11. A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In *Ninth International Conference on Theorem Proving in Higher Order Logics*, pages 173–190. Springer-Verlag Lecture Notes in Computer Science, 1996.
12. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, Jan. 2002.
13. T. F. Melham. A mechanized theory of the Π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.
14. M. Miculan. Developing (meta)theory of λ-calculus in the theory of contexts. *Electronic Notes in Theoretical Computer Science*, 58(1):37–58, Nov. 2001. MERLIN 2001: Mechanized Reasoning about Languages with Variable Binding.
15. M. Miculan. On the formalization of the modal μ-calculus in the calculus of inductive constructions. *Information and Computation*, 164(1):199–231, 2001.
16. D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, Oct. 2005.
17. A. Momigliano and S. J. Ambler. Multi-level meta-reasoning with higher-order abstract syntax. In *Sixth International Conference on Foundations of Software Science and Computational Structures*, pages 375–391. Springer-Verlag Lecture Notes in Computer Science, Apr. 2003.
18. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer-Verlag Lecture Notes in Computer Science, 2002.
19. A. Nogin, A. Kopylov, X. Yu, and J. Hickey. A computational approach to reflective meta-reasoning about languages with bindings. In *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN Workshop on MEchanized Reasoning about Languages with varIable biNding*, pages 2–12. ACM Press, 2005.
20. M. Norrish. Recursive function definition for types with binders. In *Seventeenth International Conference on Theorem Proving in Higher Order Logics*, pages 241–256. Springer-Verlag Lecture Notes in Computer Science, 2004.
21. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Sixteenth International Conference on Automated Deduction*, pages 202–206. Springer-Verlag Lecture Notes in Artificial Intelligence, 1999.
22. R. Pollack. Reasoning about languages with binding. Presentation available at http://homepages.inf.ed.ac.uk/rap/export/bindingChallenge_slides.pdf, 2006.
23. C. Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000.
24. C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.
25. C. Schürmann, A. Poswolsky, and J. Sarnat. The $\bigtriangledown$-calculus. Functional programming with higher-order encodings. In *Seventh International Conference on Typed Lambda Calculi and Applications*, pages 339–353. Springer-Verlag Lecture Notes in Computer Science, Apr. 2005.
26. A. Stump. POPLmark 1a with named bound variables. Presented at the Progress on Poplmark Workshop, Jan. 2006.
27. B. Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.