Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 174 (2007) 69–77

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Nominal Reasoning Techniques in Coq

(Extended Abstract)

Brian Aydemir Aaron Bohannon Stephanie Weirich

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA

Abstract

We explore an axiomatized nominal approach to variable binding in Coq, using an untyped lambda-calculus as our test case. In our nominal approach, alpha-equality of lambda terms coincides with Coq's built-in equality. Our axiomatization includes a nominal induction principle and functions for calculating free variables and substitution. These axioms are collected in a module signature and proved sound using locally nameless terms as the underlying representation. Our experience so far suggests that it is feasible to work from such axiomatized theories in Coq and that the nominal style of variable binding corresponds closely with paper proofs. We are currently working on proving the soundness of a primitive recursion combinator and developing a method of generating these axioms and their proof of soundness from a grammar describing the syntax of terms and binding.

Keywords: Coq, nominal reasoning techniques, variable binding.

1 Introduction

We present here work on implementing within the Coq proof assistant [2] a "nominal" approach to formalizing syntax with variable binding. This approach is characterized by a close correspondence between common practice on paper and reasoning within Coq. For example:

(i) All occurrences of object-level variables of a given sort (binding, bound, and free) are represented uniformly using *atoms*, an infinite set of objects with decidable equality.

(ii) Alpha-equivalence of object-level terms is represented by Coq's built-in equality, not a separately defined equivalence relation.

Both of these points reflect common practice with pencil and paper formalizations. More generally, our nominal approach is designed to eliminate the need to reason about any terms that do not actually appear in paper proofs, e.g., pre-terms, shifted terms, and exotic terms.

1571-0661/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2007.01.028

Our ultimate goal is to provide a system that takes as input a specification of a language and produces as output a Coq signature providing the term constructors for this language and axioms about their behavior, including a natural induction principle. The system should also generate a module implementing the signature, thereby proving the signature's soundness. The signature will not define object-level terms as an inductive datatype in Coq. Nevertheless, we believe that the axioms in the signature can be made easily usable by generating a specialized library of tactics and lemmas. Our framework also includes a library containing concepts, such as atoms and swapping (introduced in Section 2), common to all languages.

The primary contributions of this paper are to demonstrate that a nominal approach to variable binding is indeed possible in Coq and to highlight the issues that arise when implementing such an approach in a dependently typed type theory. While we do not yet have the system described above, we have assessed the theoretical and practical viability of this approach in the particular instance of an untyped lambda calculus, while bearing in mind the issues that arise in more complex languages. We feel that our experience with this specific case will allow us to build a complete system as described above.

The rest of this paper is structured as follows. We first describe the foundational components of our approach in Section 2 and the design and implementation of our signature for an untyped lambda calculus in Section 3. We then give some empirical observations about using this signature in Section 4. We discuss related work in Section 5 and conclude in Section 6 with an overview of our ongoing work.

2 Foundations for nominal signatures

As in previous work on nominal approaches for variable binding [7,8,9], we base our work on atoms, swapping, and support. Since swapping and support cannot be defined parametrically for all types, we use an encoding of Haskell-like type classes to quantify over all types for which these notions are defined. We discuss each of these components in this section.

Our development makes extensive use of dependently typed records to capture types which possess certain properties and operations. In the case of atom swapping, the ability to abstract over such records is critical. In other cases, it simply makes our code more flexible. For example, we describe a type for finite sets with extensional equality using the record type `ExtFset`, part of which is shown below.

```
Record ExtFset (T : Set) : Type := mkExtFset {
  extFset : Set;  In : T -> extFset -> Prop;  ...  }
```

The record type is parameterized by `T`, the type of elements carried by the sets. The actual type of finite sets over `T` is given by the field `extFset`, and `In` is a set-membership predicate. The names of these fields are constants (i.e., record field selectors) whose full types are

```
extFset : ∀ T : Set, ExtFset T -> Set
In : ∀ (T : Set) (R : ExtFset T), T -> extFset T R -> Prop .
```

```
Record AtomT : Type := mkAtom {
  atom : Set;   asetR : ExtFset atom;   aset := extFset asetR;
  atom_eqdec : ∀ a b : atom, {a = b} + {a <> b};
  atom_infinite : ∀ F : aset, { b : atom | b ∉ F }  }.
```

Fig. 1. Atoms.

We use Coq's implicit arguments mechanism to infer the arguments `T` and `R` when possible, and we write `x ∉ F` for `not (In x F)` when this can be done.

In general, our use of records implements a dictionary-passing semantics for type classes. Each record type defines a type class, and fields of the record type are fields of the type class. To quantify over only those types which are members of a given type class, we quantify over its dictionary. We do not use modules for this purpose because we cannot quantify over all modules implementing a given signature.

We also use a record type to capture the essential qualities of the variable names in our object languages, namely that there are an infinite number of names and that equality on names is decidable. We call objects with these properties *atoms*; records of type `AtomT`, shown in Figure 1, consist of a type and proofs that the type is a collection of atoms. The field `atom` is the type of the atoms, `aset` is the type of finite sets of atoms, and `atom_eqdec` asserts that equality on the atoms is decidable. The function `atom_infinite`, when supplied any finite set `F` of atoms, produces an atom `b` paired with a proof that `b` is not in `F`. Note that this function requires that the type `atom` be infinite and implements "choosing a fresh atom," an operation whose details are typically left unspecified on paper. With Coq's implicit coercions, for any `A : AtomT`, we may write `A` wherever `atom A` is required. Specifically, whenever `A` occurs in a location where a term of type `Set` is required, Coq implicitly inserts an application of `atom`.

Having characterized atoms, we need to construct a definition for swapping a pair of atoms in arbitrary expressions. Atom swapping is a central concept in nominal approaches for two reasons. First, it is easy to define an appropriate method of swapping atoms on almost any type, including function types and types with nominal binding. Second, it gives us a means to generically specify which names are fresh for any such type. The important properties of atom swapping for any type `X` are specified by the record `SwapT` in Figure 2. The property `swap_same` asserts that swapping an atom with itself must always leave the expression unchanged. The next property states that swapping must be an involution. The final property allows nested swaps to be reordered.

In theory, the user may use any definition of swapping for a given type that satisfies the properties in `SwapT`, but in practice there is usually a natural one defined by the structure of the type. The simplest form of swapping is the swap of atoms `a` and `b` of type `atom A` applied to the atom `c`, also of type `atom A`, denoted by `swapa A (a, b) c`. We provide the constructor `mkAtomSwap` that uses the `swapa` function to construct the `SwapT` record. For types where no atoms (of the sort being swapped) appear (e.g., the type `nat`), the only reasonable definition of applying a swap is to leave the object unchanged.

```
Record SwapT (A : AtomT) (X : Set) : Set := mkSwap {
  swap : (A * A) -> X -> X;
  swap_same : ∀ a x, swap (a, a) x = x;
  swap_invol : ∀ a b x, swap (a, b) (swap (a, b) x) = x;
  swap_distrib : ∀ a b c d x,
    swap (a, b) (swap (c, d) x) =
    swap (swapa A (a, b) c, swapa A (a, b) d) (swap (a, b) x) }.
```

Fig. 2. Swapping.

```
Parameters (tmvar : AtomT) (tm : Set).
Parameter var : tmvar -> tm.
Parameter app : tm -> tm -> tm.
Parameter lam : tmvar -> tm -> tm.
Axiom tm_induction : ∀ (P : tm -> Prop) (F : aset tmvar),
  (∀ x : tmvar, P (var x)) ->
  (∀ t : tm, P t -> ∀ u : tm, P u -> P (t @ u)) ->
  (∀ x : tmvar, x ∉ F -> ∀ t : tm, P t -> P (λ x . t)) ->
  (∀ t : tm, P t).
```

Fig. 3. Term constructors and induction principle.

Defining how to apply a swap to an expression with a function type is not too difficult, either. Our definition follows Pitts [8] and satisfies the properties in the `SwapT` record (if we allow ourselves an axiom of functional extensionality):

```
Variables (A : AtomT).
Variables (X : Set) (XS : SwapT A X) (Y : Set) (YS : SwapT A Y).
Definition func_swap (a b : A) (f : X -> Y) :=
    fun x => swap YS (a, b) (f (swap XS (a, b) x)).
```

Our framework for atom swapping allows users to define swapping on any *non-dependent* type that lives in the sort `Set`. It is currently unclear whether there is a good way to specify what it means to swap over a dependent type.

3 Signature for an untyped lambda calculus

In this section, we describe the main components of our signature for terms of the untyped lambda calculus. First, our signature includes a declaration of a type for terms, which live in the sort `Set`, and introduction and elimination forms for this type, as shown in Figure 3. Using Coq's notation mechanism, we write `t @ u` for `app t u` and `λ x . t` for `lam x t`. We would like the type `tm` to resemble an inductive type, so our introduction and elimination forms for it are similar to those of types defined by Coq's `Inductive` keyword.

For a natively defined inductive type `X`, Coq generates the definition of a term `X_rect` (using the language constructs `fix` and `match`), which serves as a recursion combinator that can produce results with a dependent type. When specialized to the sort `Prop`, the type of this combinator serves as an induction principle. How-

```
Parameter fvar : tm -> aset tmvar.
Axiom fvar_lam : ∀ (x : tmvar) (s : tm),
  fvar (lam x s) = remove x (fvar s)
Parameter subst : tm -> tmvar -> tm -> tm
Axiom subst_lam : ∀ (x y : tmvar) (s t : tm),
  x <> y -> x ∉ (fvar t) ->
    (λ x . s) [y := t] = λ x . (s [y := t]).
```

Fig. 4. Free variables and substitution on terms.

```
Axiom swap_var : ∀ (x y z : tmvar),
  (x, y) • (var z) = var ((x, y) ∘ z).
Axiom swap_app : ∀ (x y : tmvar) (t u : tm),
  (x, y) • (t @ u) = ((x, y) • t) @ ((x, y) • u).
Axiom swap_lam : ∀ (x y z : tmvar) (t : tm),
  (x, y) • (λ z . t) = λ ((x, y) ∘ z) . ((x, y) • t).
Axiom eq_lam : ∀ (x y : tmvar) (t : tm),
  y ∉ fvar t -> λ x . t = λ y . ((x, y) • t).
Axiom injection_lam : ∀ (x x' : tmvar) (t t' : tm),
  λ x . t = λ x' . t' ->
    (x = x' ∧ t = t') ∨ (x ∉ fvar t' ∧ t = (x, x') • t').
```

Fig. 5. Axioms for swapping and equality.

ever, it is not clear how to perform swapping on terms with dependent types, so we cannot axiomatize such a powerful recursion operator in this signature. Instead we axiomatize an independent induction principle. Importantly, this induction principle allows us to reason only about fresh names for the bound variable in the `lam` case, by taking a finite set of names from which the bound variable is guaranteed to be distinct (recall that `aset tmvar` is the type of *finite* sets of `tmvars`).

Our signature does not yet include a recursion combinator—we are currently working to provide such an operator (see Section 6). However, for lambda calculus terms, the main use of a recursion combinator is for the definitions of substitution and free variable functions. Therefore, our signature axiomatizes these operations—the axioms for the `lam` cases are shown in Figure 4. Even with a recursion operator, it may make sense to include these operations in a generated signature. Again, we use Coq's notation to write `s [y := t]` for `subst s y t`. Note that `subst_lam` is the only axiom defining the behavior of `subst` on `lam`-abstractions, yet `subst` must be a total function by virtue of its type. Therefore we also axiomatize alpha-equivalence for lambda terms (see Figure 5). Given a `lam`-abstraction, we can use always `eq_lam` to rename the bound variable so that `subst_lam` applies, as on paper.

Axiomatizing equivalence requires a canonical notion of swapping on lambda-terms. Thus, our signature includes the following:

```
Definition tvS := mkAtomSwap tmvar.
Parameter tmS : SwapT tmvar tm.
```

The first line constructs a default definition of swapping for `tmvar` atoms. The second asserts the existence of a definition of swapping on terms. We use Coq's notation mechanism to write `(x, y) ∘ z` for `swap tvS (x, y) z`, which applies a swap of the variable names `x` and `y` to the variable `z`, and `(x, y) • t` for `swap tmS (x, y) t`, which applies the swap to the term `t`. The result of applying a swap to a term is given by three axioms—one for each constructor—and is also shown in Figure 5. Note that this definition simply applies the swap to the arguments of the constructor, even in the `lam` case.

We have implemented a module with this signature (thereby proving the soundness of our axioms) using a locally nameless [5] implementation of lambda terms where free variables are named and bound variables are encoded using de Bruijn indices. We define `tm` to be the type of locally nameless terms paired with well-formedness proofs indicating that all indices refer to bound variables, and we use an axiom of proof irrelevance to equate well-formedness proofs when comparing terms for equality. Thus, our induction principle allows one to prove properties about all well-formed terms without having to explicitly prove anything about indices. When proving that this principle holds, we assume its premises, in particular that

```
∀ x : tmvar, x ∉ F -> ∀ t : tm, P t -> P (λ x . t) ,
```

and then show that `P` holds for all `x` by induction on the *size* of `x`. The interesting case is when `x` is a locally nameless lambda abstraction, where we need to use the above premise to show that `P x` holds. In the abstraction's body, we instantiate the bound index to a sufficiently fresh name `y`, resulting in a term `t` such that `x = λ y . t`. Since `P t` holds by the induction hypothesis, the above premise implies that `P (λ y . t)` holds. Structural induction on `x` would fail here since `t` is not a subterm of `x`. The remainder of the signature is straightforward to implement.

4 Experience using the signature

The statements of theorems in the nominal style are about as close to those on paper as one could hope for. For example, the following two theorems can be proved from our signature by nominal induction on `M`.

```
Theorem subst_not_fv  : ∀ x M N, x ∉ (fvar M) -> M [x := N] = M.
Theorem subst_comm : ∀ x y M N L, x <> y -> x ∉ (fvar L) ->
  M [x := N] [y := L] = M [y := L] [x := N [y := L]].
```

Proof by induction using the `tm_induction` principle is not significantly different from proofs that would use the `induction` tactic on a standard inductive type. The reasoning in inductive proofs is very similar to that done on paper, too, but does require that we be precise in the lambda case about the set of variables from which the name of the binder must be distinct. Conservatively, we often assert that the bound variable is distinct from all free names appearing in any expression in our context. Using such assumptions requires a little more detail and care than is seen in paper proofs, but seems consistent with the general overhead of mechanization. Furthermore, we hope to automate this process.

Another critical issue that we have attempted to assess is whether it is practical to work from axiomatized equalities in Coq. For instance, since the behavior of `fvar` is axiomatized rather than defined concretely, tactics such as `simpl` cannot unfold its definition. Additionally, since alpha-equivalent terms are not convertible under our signature, there may be cases when it is necessary to use `eq_lam` to rewrite a term in order to apply a given lemma or hypothesis. We have, however, found Coq's `autorewrite` tactic to be quite powerful, allowing common simplifications to be performed automatically, even in cases where the rewrites have preconditions, and convertibility was not an issue in the proofs of the theorems above. Coq's tactic language has even allowed us to easily perform more complex combinations of simplification and case analysis. There is some room for improvement, but we have found no serious obstacles to working in this style.

5 Related work

Our work is inspired by a nominal datatype package for Isabelle/HOL [1,9]. However, in addition to the common goal of providing automated tools for reasoning about datatypes with binding, we seek to explore the issues that arise when using nominal techniques in a dependently-typed type theory and to make explicit the "signature" required to provide an effective and practically usable formalization of syntax with binding. As in the Isabelle/HOL package, and unlike in nominal logic [7], wherever we require equivariance (the invariance of a relation under swapping) or finite support, we state that requirement explicitly rather than making a global assumption.

As our signature is an axiomatization of lambda-terms and related functions, it is very similar in spirit to Gordon and Melham's axiomatization [3]. It is not clear whether a direct translation of Gordon and Melham's iteration operator could be used to derive a natural induction principle in Coq, even if the development were augmented with axioms from higher-order logic. Additionally, in the `lam` case of their iteration operator, instead of quantifying over the name of the bound variable, they quantify over functions from names to terms. In other words, they provide a "nominal" introduction form for the type of terms, and a weak-HOAS elimination form. Taking into account Norrish's experience using Gordon and Melham's axioms [6], our approach avoids making a direct connection between meta- and object-level binders in favor of a pure nominal approach.

We are not the first to use a "locally nameless" approach to representing syntax with binding. McBride and McKinna [5] give a brief history of the technique, and Leroy used it in his solution [4] to the PoplMark challenge. Our use of this approach, in addition to an axiom of proof irrelevance, is crucial in making Coq's built-in equality coincide with alpha-equality on object-level terms.

```
Variables (A : AtomT) (X : Set) (S : SwapT A X).
Definition supports (F : aset A) (x : X) : Prop :=
  ∀ a b : A, a ∉ F -> b ∉ F -> swap S (a, b) x = x.
Definition fresh (b : A) (x : X) : Prop :=
  ∃ F : aset A, supports F x ∧ b ∉ F.
Parameter tm_rec : ∀ (R : Set) (PR : SwapT tmvar R),
  ∀ f_var : tmvar -> R,
  ∀ f_app : tm -> R -> tm -> R -> R,
  ∀ f_lam : tmvar -> tm -> R -> R,
  ∀ F : aset tmvar, (supports ... F (f_var, f_app, f_lam)) ->
  (∃ b : tmvar, (b ∉ F ∧ ∀ x y, fresh PR b (f_lam b x y))) ->
  (tm -> R).
Axiom tm_rec_lam : ∀ R PR F f_var f_app f_lam supp fcb,
  let f := (tm_rec R PR F f_var f_app f_lam supp fcb) in
    ∀ b t, b ∉ F -> f (lam b t) = f_lam b t (f t).
```

Fig. 6. A recursion operator and related definitions. Ellipses indicate an omitted dictionary argument.

6 Ongoing and future work

Our ongoing work includes implementing a combinator for defining functions on terms by primitive recursion, developing a tool to generate signatures and implementations from user-provided grammar specifications, and investigating swapping on dependent types. We discuss below our progress on the recursion combinator.

Taking the work of Pitts [8] as inspiration, we begin by defining what it means for a finite set of atoms to support an object. Intuitively, an object is supported by a set of atoms when the set includes the free names of the object. Freshness then generalizes the idea of when a name is free for an object. Precise definitions are given in Figure 6. Note that the sets that support an object change depending on the definition of swapping used, and hence so do the atoms that may be considered fresh for an object.

Based on our initial attempts to define a recursion combinator, we expect that `tm_rec`, shown in Figure 6, can be implemented and `tm_rec_lam` can be proved sound. Except for the side condition `b ∉ F`, the axiom `tm_rec_lam` takes the usual form for a function defined by primitive recursion. The arguments

```
∀ F : aset tmvar, (supports ... F (f_var, f_app, f_lam)) and
∃ b : tmvar, (b ∉ F ∧ ∀ x y, fresh PR b (f_lam b x y))
```

to `tm_rec` follow Pitts. The `supports` proposition concisely captures Norrish's requirements on his recursion operator that the functions `f_var`, `f_app`, and `f_lam` "respect permutation" and "not create too many fresh names" [6]. Finally, whereas `tm_rec` can be used to define only non-dependently typed functions, we plan on investigating a combinator for defining dependently typed functions.

References

[1] Berghofer, S. and C. Urban, *Nominal datatype package for Isabelle/HOL*, http://isabelle.in.tum.de/nominal/.

[2] Bertot, Y. and P. Castéran, "Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions," Springer-Verlag, 2004.

[3] Gordon, A. and T. Melham, *Five axioms of alpha-conversion*, in: J. von Wright, J. Grundy and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, LNCS **1125** (1996), pp. 173–190.

[4] Leroy, X., *A locally nameless solution to the POPLmark challenge*, http://cristal.inria.fr/~xleroy/POPLmark/locally-nameless/.

[5] McBride, C. and J. McKinna, *Functional pearl: I am not a number—I am a free variable*, in: *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (2004), pp. 1–9.

[6] Norrish, M., *Recursive function definition for types with binders*, in: K. Slind, A. Bunker and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004*, LNCS **3223** (2004), pp. 241–256.

[7] Pitts, A. M., *Nominal logic, a first order theory of names and binding*, Information and Computation **186** (2003), pp. 165–193.

[8] Pitts, A. M., *Alpha-structural recursion and induction (extended abstract)*, in: J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005*, LNCS **3603** (2005), pp. 17–34.

[9] Urban, C. and C. Tasson, *Nominal techniques in Isabelle/HOL*, in: R. Nieuwenhuis, editor, *Automated Deduction — CADE-20: 20th International Conference on Automated Deduction*, LNAI **3632** (2005), pp. 38–53.