Verified Programs with Binders [*]

Martin Clochard[1,2,3] Claude Marché[2,3] Andrei Paskevich[3,2]
[1]ENS Paris, [2]Inria Saclay – Île-de-France, [3]LRI (CNRS & Université Paris-Sud), France

Abstract

Programs that treat datatypes with binders, such as theorem provers or higher-order compilers, are regularly used for mission-critical purposes, and must be both reliable and performant. Formally proving such programs using as much automation as possible is highly desirable. In this paper, we propose a generic approach to handle datatypes with binders both in the program and its specification in a way that facilitates automated reasoning about such datatypes and also leads to a reasonably efficient code. Our method is implemented in the Why3 environment for program verification. We validate it on the examples of a lambda-interpreter with several reduction strategies and a simple tableaux-based theorem prover.

Categories and Subject Descriptors F.3.1 [*Theory of Computation*]: Logics and Meanings of Programs—Specifying and Verifying and Reasoning about Programs

Keywords Formal Verification, Binders, Verified Symbolic Computations, Automated Theorem Proving

1. Introduction

This work is about developing programs involving datatypes with binders, in a safe manner. Such datatypes appear when one wants to represent symbolic expressions, such as logic formulas, algebraic expressions, or abstract syntax trees of programs. As binders are the natural way to model quantifiers and anonymous function expressions, they are widely used to formalize logic and programming languages.

In this context, we propose an approach aiming at both producing reasonably efficient programs manipulating binders, and formally verifying their correctness, using as much automation as possible. Typical examples of such programs include theorem provers, higher-order language interpreters and compilers, and computer algebra systems. Some of these tools are used to produce mission-critical code, requiring a high level of trust. Formally proving such programs is thus desirable, and handling datatypes with binders is a major challenge in this task.

Dealing with binders is an active research area since a long time. On the one hand, there are several generic approaches and tools for programming datatypes with binders in a systematic way, so as to avoid classical traps such as variable capture (see [24, 25] for an overview). On the other hand, there are also several approaches proposed for reasoning about datatypes with binders (e.g. de Bruijn indices, locally nameless, nominal or nested representations), typically using highly expressive logical frameworks such as those implemented in interactive proof assistants, as exemplified by the POPLmark challenge [3]. However, there has been significantly less work considering simultaneously the issues for developing and formally proving programs with binders.

Our aim is thus to bridge a gap between two seemingly opposite objectives: reasoning easily about datatypes with binders (hence the representation should be simple) and implementing them with a reasonably efficient structure (hence the representation should be clever). Instead of looking for a single representation of binders that would be fit for both tasks, we introduce two different representations: one on the logic side to perform reasoning, one on the implementation side to perform efficient computations. The implementation representation is interpreted to the logic one, assigning to every program object a logical model, which is used in program specifications. The idea is that almost all logical reasoning is done using the model, without losing efficiency on the program side.

Our method is implemented using the Why3 environment for program verification [12] that generates proof obligations discharged by external automated and interactive provers. This environment is presented in Section 2. Why3 allows extraction of programs to OCaml code, thereby allowing efficient execution of the proved code. In order to deal with datatypes with binders in a generic way, we designed and implemented a general scheme of definitions of such types, together with a procedure for translating them into Why3 code defining types, operations, lemmas about them, and also hints for proving those lemmas. This method is detailed in Section 4.

Our approach is illustrated on several examples: the terms of lambda-calculus, the formulas of first-order logic, the terms of system $F_{<:}$ (from the POPLmark challenge). For each example, all properties and programs that are automatically generated are formally proved correct with quite limited human assistance after generation. We experimented with the generated representations on two case studies: a verified interpreter for lambda-calculus using several reduction strategies (Section 5), and certified sound tableaux-based theorem prover (Section 6). The source files for our developments are available at `http://www.lri.fr/~clochard/`. The obtained results validate the fact that the implementation of datatypes with binders, as generated by our tool, is competitive with hand-written ones. Notice that in order to handle these examples, our approach has to treat the problem of substitution under binders, making it possible to implement, for example, an innermost reduction strategy or a skolemization procedure.

[*] Work partly supported by the Bware project of the French national research organization (ANR-12-INSE-0010, `http://bware.lri.fr/`)

PLPV '14, January 21, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2567-7/14/01...$15.00.
http://dx.doi.org/10.1145/2541568.2541571

2. Preliminaries

2.1 The Why3 Environment

Why3 is a platform for deductive program verification, providing a rich language for specification and programming, called WhyML. It relies on external provers, both automated and interactive, in order to discharge the auxiliary lemmas and verification conditions. WhyML is used as an intermediate language for verification of C, Java or Ada programs, and is also intended to be comfortable as a primary programming language.

The specification component of WhyML [6], used to write program annotations and background theories, is an extension of first-order logic. It features ML-style polymorphic types, algebraic datatypes, inductive and co-inductive predicates, recursive definitions over algebraic types. Constructions like pattern matching, let-binding and conditionals can be used directly inside formulas and terms. A type, function, or predicate can be either defined or declared abstract and axiomatized. The specification part of the language can serve as a common format for theorem proving problems, suitable for multiple provers. The Why3 tool generates proof obligations from lemmas and goals, then dispatches them to multiple provers, including Alt-Ergo, CVC4, CVC3, Z3, E, SPASS, Vampire, Coq, and PVS. As most of the provers do not support some of the language features, typically pattern matching, polymorphic types, or recursion, Why3 applies a series of encoding transformations to eliminate unsupported constructions before dispatching a proof obligation. Other transformations can also be imposed by the user in order to simplify the proof search.

The programming part of WhyML [12] is a dialect of ML with a number of restrictions to make automated proving viable. In order to keep proof obligations in first-order logic, higher-order procedures are not supported. Memory aliasing is restricted by static typing, ensuring that every left-value has a statically known finite number of mutable fields. In practice this means that recursive datatypes cannot have mutable components.

WhyML function definitions are annotated with pre- and post-conditions both for normal and exceptional termination, and loops are also annotated with invariants. In order to ensure termination, recursive definitions and while-loops can be supplied with variants, *i.e.* values decreasing at each iteration according to a well-founded order. Statically-checked assertions can also be inserted at arbitrary points in the program. The Why3 tool generates proof obligations, called verification conditions, from those annotations using a standard weakest-precondition procedure. Also, most of the elements defined in the specification part: types, functions, and predicates, can be used inside a WhyML program.

For more details about Why3 and WhyML, we refer the reader to the project's web site `http://why3.lri.fr` which provides a extensive tutorial and a large collection of examples. The WhyML source code in this paper is mostly self-explanatory for those familiar with functional programming. Additionally, in our development, we make use of a currently experimental feature of Why3, namely a support for higher-order logic in specifications. This is done via a built-in library `HighOrd` providing a datatype `(func 'a 'b)` for functions from `'a` to `'b`, and the syntax `(\x:ty. t)` for the function that maps any `ty`-typed argument `x` to `t`. These constructions are encoded to first-order when calling external first-order provers. For the sake of readability, below we abbreviate `(HighOrd.func 'a 'b)` as `('a -> 'b)`.

2.2 A Taxonomy of Binder Representations

Datatypes with binders are algebraic datatypes extended with a notion of variable binding. Programming with binders, as well as formal reasoning about them, gives rise to a common difficulty: the representation of the binding structure. The traditional approach, representing every variable by an identifier, causes a lot of trouble when dealing with operations like substitution, as undesirable captures can arise during a naive substitution. Getting it right typically involves reasoning modulo α-equivalence by renaming bound variables into fresh ones, which is inefficient and error-prone. Also, reasoning modulo α-equivalence requires to state and prove a great number of invariance results.

Considering a datatype with binders, five operations stand out as basic blocks, for which there is a general scheme independent of the actual datatype.

- Construction, in order to build the values of the datatype. It includes the important case of variable binding.
- Decomposition, usually via pattern-matching. As the inverse of construction, the case of variable unbinding must be considered.
- Equality test—usually modulo α-equivalence.
- Substitution, in a way that avoids capture.
- Testing whether a variable occurs free in a term, or collecting the set of free variables.

So, from the logic side, a good representation of such a datatype is one where those operations are easy to reason about. On the implementation side, efficiency comes first.

Let us consider several possible representations of binding, using the pure lambda-calculus and the term $\lambda x.(\lambda y.xya)xa$ as an example.

Named representation

This is the standard representation of lambda-terms, with explicitly named variables in the abstraction. In Why3, it is written as

```
type term = Var id | App term term | Lambda id term
```

Here, `id` is the type of variables, usually an abstract infinite set. We identify it with `string` in the examples. The representation of the example lambda-term is straightforward[1]:

```
L "x" (A (A (L y (A (A (V "x") (V "y")) (V "a")))
          (V "x")) (V "a"))
```

Though construction, decomposition, and free variable testing are both easy to define and efficient, this representation has the above-mentioned problems with substitution. Moreover, equality is not trivial as it amounts to α-equivalence checking.

De Bruijn indices

The variables are represented as non-negative integers, called de Bruijn [10] indices.

```
type term = Var int | App term term | Lambda term
```

An index represents the binding distance to the abstraction actually binding this variable. Precisely, the variable i is bound by the $(i+1)^{\text{th}}$ abstraction crossed on the path from the variable to the root of the term. If there is no such abstraction, it corresponds to a free variable whose index is the remaining distance. Namely, it corresponds to the $(i-j)^{\text{th}}$ free variable, with j the total number of abstractions encountered on the path to the root. Assuming `a` is mapped to the integer `42`, the example is represented by:

```
L (A (A (L (A (A (V 1) (V 0)) (V 44))) (V 0)) (V 43))
```

Construction and decomposition are trivial except in the case of abstraction. In order to bind a variable, it must be first renamed to 0, and every other free variable incremented, before applying the abstraction constructor. Unbinding is the inverse: after removing the

[1] In order to get readable representations, the constructors are shortened to their first letter.

abstraction constructor, 0 is renamed to a fresh variable and other free variables are decremented. However for some definitions, like substitution or beta-reduction, this renaming is not necessary, as we can readily use 0 as the next fresh variable. It also avoids fresh variable quantification during reasoning. In other words, depending on purpose, the construction and decomposition operations inherited from the algebraic structure may be used instead of the others.

Equality test is trivial, as the structure gives a canonical representation, and free variable test is quite easy. Substitution of a term into another can be defined inductively on the structure of the second one, but avoiding captures requires to shift the first one by incrementing each free variable by one while putting it under a binding. As an implementation structure, it is slightly less efficient than the named representation because the shifting during substitution can cause performance loss greater than renaming the bound variables to rule out captures.

However, this representation is quite counter-intuitive. As variables are necessarily integers, the meaning of a free variable can be quite cryptic from the human point of view, except by explicitly carrying a context. Because of this, it is also easy to make mistakes while handling such representation—both on logic and implementation side. POPLmark [3] says about de Bruijn indices: "have two major flaws" in particular regarding readability.

Locally nameless

The locally nameless representation [7] is hybrid. Its syntax distinguishes free and bound variables. Bound variables are represented by de Bruijn indices, while free ones are named.

```
type term = BruijnVar int | FreeVar id
          | App term term | Lambda term
```

The example term is represented as follows.

```
L (A (A (L (A (A (B 1) (B 0)) (F "a"))) (B 0)) (F "a"))
```

Again, this is a canonical representation, one which makes much clearer the meaning of free variables. Equality test, free variable testing, and even substitution are trivial, as captures are prevented by the structure invariant. There is, however, no longer any canonical way to define the decomposition function in the abstraction case: it should rename the de Bruijn index 0 to some fresh variable, all of them being equally valid. It can be easier to reason on a canonical fresh variable, as it can be done with 0 in the de Bruijn representation. However, this is not possible in this setting, as the de Bruijn variables must be bound. All the reasoning must be done modulo a non-trivial well-formedness invariant, which prevents reasoning on open terms.

However, it is good as an implementation structure: it inherits the advantages of the de Bruijn representation, while removing any need for shifting.

Nested datatype

Basically, the nested datatype representation [5] is locally nameless with invariants enforced by the type system itself, or de Bruijn indices in unary representation.

```
type option 'a = None | Some 'a
type term 'a = Var 'a | App (term 'a) (term 'a)
             | Lambda (term (option 'a))
```

De Bruijn indices are turned into corresponding unary integers `Some`^{n} `None`, while an occurrence of a free variable x is translated into `Some`^{n} `x`, where n is the number of abstractions on the path from the occurrence to the root of the term. So a value of type `term 'a` represents a lambda-term with free variables of type `'a`.

This yields the following representation for the example:

```
L (A (A (L (A (A (V (S N)) (V N)) (V (S (S "a")))))
        (V N)) (V (S "a")))
```

This representation method inherits most of the advantages of de Bruijn indices without having cryptic free variables. The representation is canonical, which makes the equality test trivial. The free variable testing is quite easy as well, and though shifting is still necessary, captures during substitutions are ruled out by the type system itself.

Finally, construction and decompositions work as de Bruijn, considering `None` to be 0 and `Some` to be the successor. Binding x is done by renaming it to `None`, and adding a `Some` constructor to other variables. Unbinding is the inverse.

```
function bind (x:'a) (t:term 'a) : term (option 'a)
function unbind (t:term (option 'a)) (x:'a) : term 'a
```

As discussed above for de Bruijn representation, there are cases where using the construction and decompositions inherited from the algebraic structure is more practical. As all fresh variables are equivalent in order to decompose an abstraction, `None` is usually the best choice as it is a canonical choice, is evidently fresh (it was not part of the type of free variables), and it avoids both reasoning about the renaming coming from unbinding and also about the quantification over fresh variables. Moreover, binding it afterward avoids renaming as well.

Consequently, this representation is a good candidate for reasoning on datatypes with binders.

3. Our Approach Illustrated on the case of Pure Lambda-Calculus

In this section, we present our approach through the example of the pure lambda-calculus, as it features most of the problems encountered when dealing with binders, while having a relatively simple structure. We first focus on the theory used for specification, then describe the implementation structure. In Section 5, we will show a practical example of a program using such structures: a procedure performing beta-reduction.

3.1 Specification

Choice of a specification representation

The first thing to decide upon is which representation of datatypes to choose in order to reason about them. Such a representation should have several properties:

- Representation should be canonical, first of all to avoid invariance lemmas. Not only it avoids to state them, but it also helps a lot automated proving, as automated provers can use specialized decision procedures for equality instead of trying to instantiate lemmas. Moreover, in the particular case of α-equivalence, getting a proof of invariance lemmas is non-trivial.
- There should be a canonical way to unbind variables. Though any reasonable notion should not depend on the choice of the fresh variable, getting it proved is another matter. Experiments on the beta-reduction predicate in Why3 showed that it was difficult if there was any quantifier on fresh variables. Having such a canonical fresh variable for the unbinding prevents that problem, especially if the unbinding and binding this variable back correspond to simpler operations.
- As the specifications will ultimately be written and read by human beings, the representation should not be cryptic.
- The operations must be kept as simple as possible, in order to simplify proof search—and also human understanding.

We choose the nested datatype representation for the specification purpose, as it seems to fit the best those requirements. Named representation clearly conflicts with the first and fourth points. The best fit with these requirements are the de Bruijn indices and the nested representation. Additionally the typing constraints enforced by the nested representation are an advantage. As for the locally nameless representation, the problem is the second point, as there is no canonical decomposition. This point is easy to deal with in the nested datatype setting, where unbinding modifies the type of a free variable and the canonical fresh variable is just `None`. The above-mentioned "fresh variable independence" results can be derived immediately from commutation with bijective renamings, which is fairly simple to prove while using the canonical decomposition.

Another representation which we do not present here is the canonical locally named representation [22]. This approach seems to fit most of our requirements, in particular, the existence of a canonical fresh variable to perform unbinding. However, this fresh variable actually depends on the term, which imposes additional renaming work when defining a relation between several terms. Moreover, this canonical choice may conflict with a variable we are carrying outside the term, in which case it may be incorrect to use it. This issue does not show up with the nested datatype as the fresh variable is out of the current type for free variables.

Defined notions

Considering only the binding structure, there are only two interesting notions to define: free variables and substitutions. Equality is given for free by the representation, and the renamings necessary for construction and decomposition derive from substitution.

It is straightforward to write the predicate "being a free variable in" by pattern-matching on the type `term`:

```
predicate is_free_var (x:'a) (t:term 'a) = match t with
 | Var y -> x = y
 | App u v -> is_free_var x u \/ is_free_var x v
 | Lambda u -> is_free_var (Some x) u
end
```

The substitution operation is slightly more complicated. We define substitutions as functions that simultaneously map every variable to a term. Prior to defining the substitution operation, we need to handle the special case of variable renaming, which is done by a structural recursion.

```
function rename (t:term 'a) (sigma:'a -> 'b) : term 'b =
 match t with
 | Var x -> Var (sigma x)
 | App u v -> App (rename u sigma) (rename v sigma)
 | Lambda u ->
   let sigma' = \ x:option 'a. match x with
     | None -> None
     | Some x -> Some (sigma x)
     end
   in Lambda (rename u sigma')
 end
```

The first two cases are natural. In the case of abstraction, the substitution must be modified to take into account the new variable `None`, which should be replaced by itself. This corresponds exactly to the definition of `sigma'` above.

The substitution operation is quite similar:

```
constant lift_renaming = (\ x:'a. Some x)
function subst (t:term 'a) (sigma: 'a -> term 'b) :
  term 'b =
 match t with
 | Var x -> sigma x
 | App u v -> App (subst u sigma) (subst v sigma)
 | Lambda u ->
   let sigma' = \ x:option 'a. match x with
     | None -> Var None
     | Some y -> rename (sigma y) lift_renaming
     end
   in Lambda (subst u sigma')
 end
```

where the substitution `sigma'` to apply under the binder is now defined with a renaming, in order to add a `Some` constructor over the variables of substituted terms (which corresponds to the shifting of de Bruijn indices). Notice that a definition using `subst` instead of `rename` would be rejected by Why3 since it would not be structurally decreasing.

Interestingly, the type system prevents us from writing an incorrect version of substitution lifting, which could cause captures. The new input type forces the case of the new fresh variable to be considered, while the output type prevents us from forgetting lifting the variables in the second case. It is possible to write an erroneous version of the function, but not easily (or intentionally). This would not have been true for de Bruijn indices, so those errors could have gone undetected until we try to prove properties of the substitution.

Although substitution and renaming are the principal interesting functions, there are also some other functions or constants that are interesting to name: first, composition of substitutions, defined by pointwise substitution

```
function subst_compose (sigma1: 'a -> term 'b)
  (sigma2: 'b -> term 'c) : 'a -> term 'c =
    (\ x:'a. subst (sigma1 x) sigma2)
```

and variants when one or both substitutions are renamings; second, the identity substitution:

```
constant subst_id: 'a -> term 'a = (\ x:'a. Var x)
```

Proved properties

Considering only the binding structure, there are several properties that are interesting to have in any case.

- Substituting twice into a term is equivalent to apply the composition of both substitutions, that is mathematically

$$(t\sigma_1)\sigma_2 = t(\sigma_1 \circ \sigma_2)$$

```
lemma subst_consecutive:
 forall t:term 'a, s1:'a -> term 'b,
        s2:'b -> term 'c.
  subst (subst t s1) s2 =
    subst t (subst_compose s1 s2)
```

- As an immediate consequence (by extensionality) of the previous property, composition of substitutions is associative.

```
lemma subst_compose_associative:
 forall s1:'a -> term 'b, s2:'b -> term 'c,
        s3:'c -> term 'd.
  subst_compose (subst_compose s1 s2) s3
  = subst_compose s1 (subst_compose s2 s3)
```

- Characterization of the free variables after a substitution.

$$x \in \mathrm{fv}(t\sigma) \leftrightarrow (\exists y \in \mathrm{fv}(t).\ x \in \mathrm{fv}(\sigma(y)))$$

```
lemma subst_free_var:
 forall t:term 'a, sigma:'a -> term 'b, x:'b.
  is_free_var x (subst t sigma) <->
  exists y:'a. is_free_var y t /\
               is_free_var x (sigma y)
```

- The identity substitution preserves lambda-terms.

```
lemma subst_identity:
 forall t:term 'a. subst t subst_id = t
```

- Result of substitution depends only on the instances of free variables

$$t\sigma_1 = t\sigma_2 \leftrightarrow (\forall x \in \mathrm{fv}(t).\ \sigma_1(x) = \sigma_2(x))$$

```
lemma subst_equality:
 forall t:term 'a, s1 s2:'a -> term 'b.
  subst t s1 = subst t s2 <->
  (forall x:'a. is_free_var x t -> s1 x = s2 x)
```

i.e. if two substitutions are equal on every free variable of a term, then applying them to this term yields the same result ; and conversely.

The lemmas above have several variants, considering the cases where the substitutions are renamings. The variants involving renamings must be proved first and then used for the main lemmas.

Two of those properties, the fact that substitution depends only on free variables and that substitution by identity preserves terms, can be proven by direct induction. The other properties require more work. In the case of the property about consecutive substitutions into a term, we have to prove that the lifting operation preserves composition of substitution. As it involves two renamings interleaved with substitution, proving this property requires the two variants of this lemma where one of the substitutions is replaced by a renaming. Unraveling completely the proof, the four variants of the lemma are necessary. For more details about how many hints are given inside Why3 in order to get the lemmas proved automatically, see Section 4.4.

Actually, because of the presence of a renaming in the lifting operation, in this formalization it is a common pattern to first prove whatever property we want for substitution with renamings. In this case, this means that it is necessary to prove the four lemmas in the order of increasing strength.

The characterization of the free variables follows the same pattern: in order to be able to relate free variables in the substitutions before and after lifting, it is first necessary to prove its variant with renamings. The remaining lemmas are straightforward consequences.

3.2 Implementation

As for specification, we first have to choose a representation. There is a trade-off between efficiency and difficulty to prove the relation between the implementation and the specification. The difficulty prevented us from using overly complex data structures (delayed substitution in the structure, director strings [11], etc.), while efficiency was the obvious reason for not using directly the specification data structure. The encoding of variables in the latter consumes a lot of memory (potentially quadratic in the size of the term), and the shifting operation causes similar performance loss.

Also, as the logical reasoning is not done with the implementation structure, it is no longer needed for the representation to be canonical, neither for unbinding to have a simple canonical variant.

The type of lambda-terms of our implementation is a record as follows.

```
type id = int
type iterm = {
  repr : Nameless.term;
  mfv : int;
  ghost term_model : Nested.term id
}
```

The first field is the locally nameless representation of the term, as defined in Section 2.2. In order to generate fresh variables efficiently, we also carry in the second field an upper approximation of its set of free variables, that is in practice, as we use integers for the type of free variables, we carry an upper bound of the free variables of the term, which is very easy to keep up-to-date. Finally, we use a so-called *ghost* field, that is seen only in specifications, that is the image of that term in the logic side, that we call its *model*. The characterization of the model is done thanks to a function that maps a term in the nameless representation into a term in the nested one. This is defined recursively, and takes as extra parameters two evaluations for free and de Bruijn variables into lambda-terms:

```
type nameless = Nameless.term
function model (t: nameless)
  (fr:id -> Nested.term 'a)
  (b:int -> Nested.term 'a) : Nested.term 'a =
 match t with
 | BruijnVar n -> b n
 | FreeVar x -> fr x
 | App u v -> Nested.App (model u fr b) (model v fr b)
 | Lambda u ->
   Nested.Lambda
     (model u (\ x:id. rename (fr x) lift_renaming)
              (\ n:int. if n = 0 then Var None else
                 rename (b (n-1)) lift_renaming))
end
```

A property of this evaluation function is that it is independent of the values in the environment for de Bruijn indices which are greater than every free de Bruijn variable in the locally nameless representation. The model of a well-formed locally nameless representation (well-formed meaning without free de Bruijn variables) is thus obtained by taking the model with the evaluation composed of the identity substitution for free variables and any evaluation for de Bruijn variables, since it does not depend on it. Thanks to this property, we can define the following predicate which plays the role of an invariant for our implementation type `iterm`:

```
predicate impl_ok (t:iterm) =
  no_free_debruijn t.repr /\
  t.term_model = model t.repr subst_id subst_dummy /\
  (forall x:id. is_free_var x t.term_model -> x < t.mfv)
```

where `no_free_debruijn` is a predicate checking that each de Bruijn index i in the term is below at least i binders.

Another direct property of the `model` function that can be shown by induction is that it commutes with substitution:

```
lemma model_commute_subst :
 forall t:nameless, fr:id -> term 'a,
       b:int -> term 'a, sigma:'a -> term 'b.
  subst (model t fr b) sigma =
    model t (subst_compose fr sigma)
            (subst_compose b sigma)
```

Both properties are crucial to show that binding and unbinding operations coincide with particular substitutions in the model.

As we want to use this representation for lambda-terms in our implementation, we implement the five basic operations mentioned in Section 2.2. Construction and decomposition for non-binder cases, as well as equality, are straightforward. Free variable testing is easy to write as a recursive function but a little trickier to verify correct because the model is given by the evaluation function. In order to do it, we introduce two ghost arguments corresponding to evaluations into variables, and a precondition ensuring that free de Bruijn variables of the term were always interpreted differently than any possible free variable. On a well-formed nameless term, this coincides exactly with free variable testing. Finally, the code for substitution is extremely similar to the one for unbinding (especially unbinding with a term), so let us focus on binding and unbinding operations.

In order to implement those functions, we have to be able to transform the de Bruijn variable 0 to a free variable and the other way, since the de Bruijn variable 0 corresponds to the variable bound by the abstraction constructor. We implement two recursive functions over locally nameless representation to achieve that: one that takes a free variable name and changes it to a given de Bruijn index, and another that transforms a given de Bruijn index to a well-formed locally nameless term. The second one is more powerful than needed, but it is not harder to prove neither less efficient, and it can replace efficiently the pattern of unbinding followed by substitution. Both functions are straightforward to write:

```
function bind (t:nameless) (x:id) (i:int) : nameless =
 match t with
 | BruijnVar j -> BruijnVar j
 | FreeVar y -> if x = y then BruijnVar i else FreeVar y
 | App u v -> App (bind u x i) (bind v x i)
 | Lambda u -> Lambda (bind u x (i+1))
 end

function unbind (t:nameless) (i:int) (u:nameless) :
  nameless =
 match t with
 | BruijnVar j -> if i = j then u else BruijnVar j
 | FreeVar y -> FreeVar y
 | App v w -> App (unbind v i u) (unbind w i u)
 | Lambda v -> Lambda (unbind v (i+1) u)
 end
```

By induction (as they can be written as logic functions), or equivalently by adding ghost evaluation arguments and writing those two properties in the postconditions, we get the expected lemmas:

```
function update (f:'a->'b) (x:'a) (v:'b) =
  (\ y:'a. if x = y then v else f x)

lemma model_bind :
 forall t:nameless, x:id, i:int, f:id -> term 'a,
        b:int -> term 'a.
  model (bind t x i) f b = model t (update f x (b i)) b

lemma model_unbind :
 forall t:nameless, i:int, u:nameless,
        f:id -> term 'a, b b':int -> term 'a.
  no_free_debruijn u ->
   model (unbind t i u) f b =
   model t f (update b i (model u f b'))
```

In the particular case of abstraction construction/decomposition with terms respecting the locally nameless invariants (no free de Bruijn variables), those equalities correspond to the bindings and unbinding operations over the model. Also, it is straightforward to prove that the invariants are preserved.

4. A Generator of Binder Specification and Implementation

The work required to model and implement a datatype with binders is complex but mostly generic: the needed operations such as substitution are always the same, and the lemmas to reason on such structures are also the same. Still, adapting this work to a particular datatype is error-prone and takes a lot of time. This stresses the need for some automated generic method. However, as it involves reasoning over inductive structures of datatypes, this is hardly doable in the programming language itself, and calls instead for some kind of meta-programming. The Why3 environment is no different in this respect: its current lack of reflection capabilities makes it impossible to describe a datatype with binders, in a generic style, directly in Why3.

So instead, we wrote a standalone independent tool that generates Why3 code, from a high-level description of datatypes with binders. Technically, the tool generates type declarations, functions, predicates, procedures, lemmas and hints to automatically prove those lemmas, corresponding to the representations of the datatypes with binders, the operations over them, and their standard properties. Our tool is written in OCaml and is roughly 6000 lines of code long.

The class of datatypes with binders accepted by this tool is the class of finite mutually inductive families of algebraic datatypes extended with potentially multiple occurrences of bindings inside constructors, and with at most one kind of variable per declared datatype. The idea is that a kind of a variable corresponds to the datatype of terms that may be substituted for it. It would probably be possible to extend the class with several/infinitely many kinds per datatype, but this is mostly irrelevant from the point of view of the binding structure, as all those kinds have the same behavior. Moreover, there is a simple way to refine kinds by encoding the kind information in the variables themselves.

4.1 Description of a datatype with binders

An algebraic datatype with binders is described by its list of constructors. There are two kinds of constructors:

- The variable constructor Var. It declares a constructor taking a variable name corresponding to the kind of variable for the current datatype. It is not allowed more than once, because of the correspondence between variable kinds and the types of terms that can be substituted for it. It is only possible to replace such a variable with a term from the datatype currently defined.
- A constructor declared under the form $\mathsf{C}\ p_1 \cdots p_n$ where each parameter p_i is either a type or a binding of some kind denoted as $\sharp kind$. A binding $p_i = \sharp k$ corresponds to a variable of kind k bound in every arguments corresponding to the types in $p_{i+1} \cdots p_n$. In other words, a binding binds to the right.

EXAMPLE 1. *The pure lambda-calculus is described as*

$$term \ ::= \ \mathsf{Var} \mid \mathsf{App}\ term\ term \mid \mathsf{Lambda}\ \sharp term\ term$$

EXAMPLE 2. *Formulas of first-order logic are described as*

$$\begin{array}{rcl} term & ::= & \mathsf{Var} \mid \mathsf{FApp}\ function\_symbol\ term\_list \\ term\_list & ::= & \mathsf{Nil} \mid \mathsf{Cons}\ term\ term\_list \\ fmla & ::= & \mathsf{Forall}\ \sharp term\ fmla \mid \mathsf{Exists}\ \sharp term\ fmla \\ & \mid & \mathsf{And}\ fmla\ fmla \mid \mathsf{Or}\ fmla\ fmla \mid \mathsf{Not}\ fmla \\ & \mid & \mathsf{PApp}\ predicate\_symbol\ term\_list \end{array}$$

Notice that the types term_list and fmla do not have a Var *constructor, meaning that there are no variables of these kinds.*

EXAMPLE 3. *The system* $F_{<:}$ *is described by the grammar*

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \top \mid \forall\alpha <: \tau.\,\tau \mid \tau \to \tau \\ t & ::= & x \mid t\ t \mid \lambda x : \tau.t \mid \Lambda\alpha <: \tau.t \end{array}$$

where τ *denotes types,* α *denotes type variables,* t *denotes terms and* x *denotes term variables. In our setting, it is described as follows.*

$$\begin{array}{rcl} ftype & ::= & \mathsf{Var} \mid \mathsf{Top} \mid \mathsf{Forall}\ ftype\ \sharp ftype\ ftype \\ & \mid & \mathsf{Arrow}\ ftype\ ftype \\ fterm & ::= & \mathsf{Var} \mid \mathsf{App}\ fterm\ fterm \\ & \mid & \mathsf{Lambda}\ ftype\ \sharp fterm\ fterm \\ & \mid & \mathsf{TLambda}\ ftype\ \sharp ftype\ fterm \end{array}$$

Notice the order of arguments of Forall, Lambda *and* $\mathsf{TLambda}$, *making explicit the sub-terms where the variable is bound, e.g. in the notation* $\forall\alpha <: \tau_1.\,\tau_2$, *the type variable* α *is bound only in* τ_2.

4.2 Specification scheme

Encoding of the binding structure

Following the idea of the nested datatype encoding, each declared type $type_i$ is translated into a parameterized Why3 type $\mathtt{TYPE}_i$. The type parameters of $\mathtt{TYPE}_i$ correspond to the types of the free variables of each kind that may appear in a value of this type. More precisely, the Why3 declaration of $\mathtt{TYPE}_i$ has the form

```
type TYPE$_i$ ('a$_k$)$_{k\in\mathrm{kinds}(i)}$ = Var 'a$_i$ (| C $\cdots$)$_C$
```

where $\mathrm{kinds}(i)$ is the least fix-point of the equation

$$\{i\} \cup \bigcup_{C,j \mid p_j \text{ not a binding}} \mathrm{kinds}(j) = \mathrm{kinds}(i)$$

The Var constructor is omitted if there is no Var in the description, and in that case the $\{i\}$ is also omitted in the previous fix-point equation. The arguments of a constructor C correspond to the non-bindings $p_i = type_j$, with each p_i replaced by

```
TYPE$_j$ (option$^{n_{i,k}}$ 'a$_k$)$_{k\in\mathrm{kinds}(j)}$
```

where $n_{i,k}$ is the number of bindings of kind k occurring in $p_1 \cdots p_{i-1}$.

Using this encoding, the unary integer (Some^m None), with $m < n$, corresponds to the m-th binding found when going to the left from the argument. It can also be interpreted in terms of applying binding operations from right to left, starting from the current argument.

EXAMPLE 4 (Example 3 continued). *The Why3 generated code for types and terms of $F_{<:}$ is as follows.*

```
type ftype 'a =
  | Var_ftype 'a
  | Top
  | Forall (ftype 'a) (ftype (option 'a))
  | Arrow (ftype 'a) (ftype 'a)

type fterm 'a 'b =
  | Var_fterm 'b
  | App (fterm 'a 'b) (fterm 'a 'b)
  | Lambda (ftype 'a) (fterm 'a (option 'b))
  | TLambda (ftype 'a) (fterm (option 'a) 'b)
```

We have $\mathrm{kinds}(\mathit{ftype}) = \{\mathit{ftype}\}$ *and* $\mathrm{kinds}(\mathit{fterm}) = \{\mathit{ftype}, \mathit{fterm}\}$ *and thus the type* ftype *is parameterized only for the kind corresponding to* ftype.

Operations of binders

The predicate checking if a variable occurs free in a term is generalized in a natural way from the one for lambda-calculus. The only new aspect is that there are several predicates, one for each variable kind and each datatype where variables of that kind may occur.

The operations of renaming and substitution are generalized following the same guidelines. There are as many such functions as types in the given family of datatypes with binders. Moreover, the renaming (resp. substitution) operation for a given type is parametrized by substitution functions given for every type. In other words, a renaming or a substitution is a tuple of functions, which gives terms to instantiate every possible free variable. The general scheme is as follows.

```
function rename$_i$ (t:TYPE$_i$ ('a$_j$)$_{j\in\mathrm{kinds}(i)}$)
 ((sigma$_j$:'a$_j$ -> 'b$_j$))$_{j\in\mathrm{kinds}(i)}$ : TYPE$_i$ ('b$_j$)$_{j\in\mathrm{kinds}(i)}$ =
 match t with
 | Var x -> Var (sigma$_i$ x)
 | C (t$_1$:TYPE$_{l_1}$)...(t$_m$:TYPE$_{l_m}$) ->
   let t'$_1$ =
     rename$_{l_1}$ t$_1$ (lift$_{1,k}$ sigma$_k$)$_{k\in\mathrm{kinds}(l_1)}$
   in
   ...
   let t'$_m$ =
     rename$_{l_m}$ t$_m$ (lift$_{m,k}$ sigma$_k$)$_{k\in\mathrm{kinds}(l_m)}$
   in
   C t'$_1$...t'$_m$
 | ...
 end
```

```
function subst$_i$ (t:TYPE$_i$ ('a$_j$)$_{j\in\mathrm{kinds}(i)}$)
 ((sigma$_j$:'a$_j$ -> TYPE$_j$ ('b$_k$)$_{k\in\mathrm{kinds}(j)}$ )$_{j\in\mathrm{kinds}(i)}$ :
     TYPE$_i$ ('b$_j$)$_{j\in\mathrm{kinds}(i)}$ =
 match t with
 | Var x -> sigma$_i$ x
 | C (t$_1$:TYPE$_{l_1}$)...(t$_m$:TYPE$_{l_m}$) ->
   let t'$_1$ =
     subst$_{l_1}$ t$_1$ (lifts$_{1,k}$ sigma$_k$)$_{k\in\mathrm{kinds}(l_1)}$
   in
   ...
   let t'$_m$ =
     subst$_{l_m}$ t$_m$ (lifts$_{m,k}$ sigma$_k$)$_{k\in\mathrm{kinds}(l_m)}$
   in
   C t'$_1$...t'$_m$
 | ...
 end
```

Functions lift$_{i,k}$ and lifts$_{i,k}$ corresponds to the lifting operations that should be applied to renaming/substitution functions because of the bindings. This is because the bound variable must be considered as part of the free variables, and the substituted terms/-variables must be shifted accordingly.

They are defined from the two base lifting functions:

```
function liftb (f:'a -> 'b) :
 option 'a -> option 'b =
  (\ x:option 'a. match x with
    | None -> None
    | Some x -> Some (f x) end)

function liftbs$_i$ (sigma:'a -> TYPE$_i$ ('b$_j$)$_{j\in\mathrm{kinds}(i)}$) :
  option 'a -> TYPE$_i$ ('b$_j$)$_{j\in\mathrm{kinds}(i),j\neq i}$ (option 'b$_i$) =
  (\ x:option 'a. match x with
    | None -> Var$_i$ None
    | Some x ->
      rename$_i$ (sigma x) (identity)$_{j\in\mathrm{kinds}(i),j\neq i}$
        lift_renaming
  end)
```

For renaming, lift$_{i,k}$ is the power of liftb to the exponent $n_{i,k}$ defined above. For substitution, (lifts$_{i,k}$ sigma) is:

```
(\ x:'a$_k$. rename$_k$ (liftbs$_k^{n_{i,k}}$)
  ((\ x:'a$_l$. Some$^{n_{i,l}}$ x))$_{l\in\mathrm{kinds}(i),l\neq k}$ identity
```

The substitution composition is still defined as pointwise substitution, so it is modified in the same way as substitution is. For each kind of variable k, the composition of a substitution of kind k is done with substitution functions for every kind in $\mathrm{kinds}(k)$. The renaming of a substitution is modified in the same way, while other composition functions are unchanged.

The identity substitution definitions, one for each kind of variable, are the same as for the lambda-calculus example.

Stated Properties

Our generator automatically produces Why3 lemmas as direct adaptation of those stated for lambda-calculus: (1) Renamings are special cases of substitutions; (2) Applying consecutive substitutions is equivalent to substitution by the composition; (3) Associativity of substitution composition; (4) Characterization of the free variables after a substitution; (5) Identity substitution preserves term; (6) Substitution depends only on the free variables of a term, and conversely.

4.3 Implementation scheme

Our generator tool also generates the implementation types and the code corresponding to the nameless representation of a datatype with binders. The generated implementation follows the same structure as for the lambda-calculus example.

The evaluation function for a locally nameless term follows directly from the generalization of substitution: there are two evaluations for free and de Bruijn variable of each kind.

```
function model_i (t:nameless_i)
  ( (f_j:id_j -> TYPE_j ('a_k)_{k∈kinds(j)})
    (b_j:int -> TYPE_j ('a_k)_{k∈kinds(j)}) )_{j∈kinds(i)} :
    TYPE_i ('a_j)_{j∈kinds(i)}
```

The $\texttt{nameless}_i$ type corresponds to the locally nameless encoding of the datatype of index i, while $\texttt{id}_i$ is the type used to represent free variables of kind i (`int` in practice).

The evaluation function definition follows the same pattern as the one defined for lambda-calculus, with the same generalization as substitution for the non-variable cases. We only detail the lifting operations for evaluations.

- The free variable evaluations are directly renamed by shifting every variable with the correct powers of the `Some` constructor, as in the locally nameless representation free variables are unconcerned by the binders— there is no new free variable in the open term to consider.
- The de Bruijn variable evaluations are lifted in the same way as substitutions, except for the base lifting operation which is adapted from options to integers:

```
function liftbb_i (b: int -> TYPE_i ('a_j)_{j∈kinds(i)}) :
  int -> TYPE_i ('a_j)_{j∈kinds(i),j≠i} (option 'a_i) =
    (\ n:int. if n = 0 then Var_i None
      else rename_i (b (n-1)) (identity)_{j∈kinds(i),j≠i}
    lift_renaming )
```

From this definition, the properties which already held in the lambda-calculus example, commutation with substitution and independence from evaluations on non-free de Bruijn variables, can still be proved directly by induction.

To deal with binder construction and decomposition, it is enough to transform the two (un)binding functions into two families of functions defined similarly for each pair of type and kind:

```
function bind_{j,i} (t:nameless_i) (x:id_j) (n:int) :
  nameless_i
function unbind_{j,i} (t:nameless_i) (n:int) (u:nameless_j) :
  nameless_i
```

Of course, the same lemmas can be proved about the interpretation on both functions. These are given below.

```
lemma model_bind_{j,i} :
 forall t:nameless_i,x:id_j,n:int,
  ( f_l: id_l -> TYPE ('a_k)_{k∈kinds(l)},
    b_l: int -> TYPE ('a_k)_{k∈kinds(l)} )_{l∈kinds(i)}.
    model_i (bind_{j,i} t x n) (f_l b_l)_{l∈kinds(i)}
      = model_i t (f_l b_l)_{l∈kinds(i),l≠j} (update f_j x (b n)) b_j

lemma model_unbind_{j,i} :
 forall t:nameless_i,n:int,u:nameless_j,
  ( f_l: id_l -> TYPE ('a_k)_{k∈kinds(l)},
    b_l: int -> TYPE ('a_k)_{k∈kinds(l)} )_{l∈kinds(i)},
  ( b'_l: int -> TYPE ('a_k)_{k∈kinds(l)} )_{l∈kinds(j)}.
    model_i (unbind_{j,i} t n u) (f_l b_l)_{l∈kinds(i)}
      = model_i t (f_l b_l)_{l∈kinds(i),l≠j} f_j
             (update b_j n (model_j u (f_k b'_k)_{k∈kinds(j)}))
```

As expected, the use cases for (multi-)binders constructions and decompositions correspond to the binding and unbinding operations in the model.

Finally, the other operations can be written and proved exactly in the same way as it was done for the lambda-calculus example. Equality test is still trivial, and free variable test is proved in the same way. Substitution is an immediate variant of unbinding to a term, and is proved nearly identically.

4.4 Proof scheme

Let us now detail how all the proofs are actually done, keeping in mind that we strive for full proof automation.

Generalities

First, we adopted the axiom of functional extensionality. Everything could have been done without it, but this would have implied proving a lot of congruence properties. More importantly, this would have forced the automated provers to use these properties instead of the specialized decision procedures for equality.

Second, Why3 support for first-class logic functions is still experimental, especially handling of lambda-expressions, so we decided not to rely on it. Although we have used lambda-expressions in this paper for the sake of readability, in the Why3 code, they are lifted out and axiomatized:

```
function f p_1...p_n : t1 -> t2 = (\ x:t1. expr)
```

is replaced with

```
function f p_1...p_n : t1 -> t2
axiom f_def : forall p_1...p_n,x:t1. f p_1...p_n x = expr
```

Importantly, both functional extensionality and this axiom scheme are consistent with Why3 logic.

Next, the large majority of the proofs needs polymorphic mutual induction, which requires human help as such proofs cannot be done by an automated prover. The natural way to do that in Why3 without using an interactive prover is to make explicit the induction proof structure using mutually recursive lemma functions, which consist only of recursive calls instantiating the induction hypothesis. The induction cases are then dispatched to the automated provers as the verification conditions of the procedures.

The lemma functions are also used in order to help provers. When a goal is too hard to be proved directly by an automated prover, and does not require some technique beyond its scope (like induction), the usual way to get the proof done is to split the goal by giving explicitly a cut rule. In lemma functions, this can be done simply by adding an assertion, which will not be remembered outside the proof. It can also be done outside by adding an auxiliary lemma, but the lemma will be remembered later, potentially degrading automated prover performance. Moreover, some of the hypotheses have to be stated again.

Here are some frequent kinds of cuts that had to be added:

- Intermediate lemmas when the proof is too big to be found.
- Universally quantified hypothesis, as automated provers will usually not try to prove them. A typical example of such hypothesis is extensional equality.
- Explicitly giving the instance for existential quantifiers. Unless they can be derived immediately from the context, they will usually not be found. Existentially quantified goals cause a lot of trouble to automated provers.
- Explicit case disjunction: it is not uncommon to split explicitly the reasoning between several cases. The typical case is reasoning over the option type, where doing case analysis usually gets the goal proved immediately. However, most of the time the automated provers do not instantiate the inversion axiom, that is generated by Why3 when encoding algebraic datatypes (for provers that do not know them).

The latter three problems mostly concern the SMT solvers with trigger-based instantiation.

Proofs of the Generated Code

The first proved lemmas are the ones over consecutive substitutions and renamings, in order of increasing strength, interleaved with associativity lemmas. As explained in Section 3.1, it must start with the lemmas about renamings. Here is the complete order:

- Prove compositionality of the base renaming lifting operation.
- Prove the lemma about consecutive renamings by mutual induction.
- Using the previous lemma and extensionality, derive the three variants of the associativity lemma about composing two renamings and a substitution in all possible orders.
- Prove the compositionality of the base substitution lifting operations for compositions of a substitution and a renaming.
- Using the previous associativity lemmas, prove the lemma about consecutive renamings and substitution in both orders by mutual induction.
- Using the two previous lemma and extensionality again, derive the three variants of the associativity lemma about composing two substitutions and a renaming.
- Prove the compositionality of the base substitution lifting operation for composition of substitutions.
- Derive the lemma about substitution associativity.

For our three examples (lambda-calculus, first-order logic formulas, $F_{<:}$ terms), these lemmas were proved without real difficulty by the automated provers. Interestingly, some of them are required only in cases with more than one kind of variable.

The next proved ones are the identity substitution lemma, which are not as easy as one could expect in the cases with several kinds of variable. Because of the structure of the lifting operation, its renaming variant must be proved first, which amounts to prove that lifting preserves identity before an immediate inductive proof. For the same reason, it was also necessary to prove that the identity substitution commutes with renamings. With one kind of variable, the proof can be done directly from the last step as the lifting operation is only a power of the base substitution lifting operation.

The remaining lemmas can be proved in any order. Defining renamings in terms of substitution follows immediately from the previous lemmas, and the characterization of equality of terms after substitution can be derived by immediate mutual induction. In order to simplify the provers' task, as one side of the equivalence is universally quantified, this induction is split in two for each direction of the equivalence. The only challenging property is the characterization of free variables after substitution, as one side of the equivalence is existentially quantified. One direction can still be proved quite easily by two inductions, one for renamings and one for substitution, since the existential quantification is eliminated. The other is not proved in general without an unreasonable number of cuts—practically giving the complete proof structure before dispatching any proof obligation. To avoid that problem, we generated a constructive version of the proof: two recursive procedures which returns an instance for the existential variable, one for renamings then one for substitution. We complete the proof by calling the final procedure inside a lemma function.

5. Case study: an interpreter for lambda-calculus

In order to check that those constructions were enough to handle lambda-calculus, we implemented and proved a procedure for beta-normalizing terms (up to a given bound, to prevent from non-termination). There are two major steps:

- Proving that the beta-reduction is preserved under (bijective) renaming, in order to propagate the relation across abstraction. This corresponds exactly to an independence result relatively to the choice of a fresh variable.
- In order to prove that the redex search terminates, finding a variant over the implementation structure. The problem is that decomposing the abstraction constructor is not a structurally decreasing operation, neither on the implementation side nor on the specification side. It was enough to write a variant function which is preserved under renamings, such as term size.

After those steps, writing the function was straightforward.

The specification of one-step beta-reduction is given by an inductive predicate as follows.

```
function unbindt (u:term 'a) : option 'a -> term 'a =
  \ x:option 'a. match x with
    | None -> u | Some y -> Var y
  end

inductive beta_red (term 'a) (term 'a) =
| BAppL : forall u v w:term 'a.
  beta_red u v -> beta_red (App u w) (App v w)
| BAppR : forall u v w:term 'a.
  beta_red v w -> beta_red (App u v) (App u w)
| BLamC : forall u v:term (option 'a).
  beta_red u v -> beta_red (Lam u) (Lam v)
| BRedex : forall u:term (option 'a),v:term 'a.
  beta_red (App (Lam u) v) (subst u (unbindt v))
```

Interestingly, thanks to the nested representation, this definition of beta-reduction does not require to explicitly bind or unbind variables, neither any kind of fresh variable quantification. This is the case in particular when defining reduction under a lambda (case `BLamC`) where the hypothesis applies to terms `u` and `v` of type `term (option 'a)`.

The procedure performing one step of beta-reduction on the program side is then specified as follows.

```
exception Irreducible
val reduce (u:term_impl) : term_impl
  requires { impl_ok u }
  ensures { impl_ok result }
  ensures { beta_red (model u) (model result) }
  raises { Irreducible ->
   forall v:term id. not(beta_red (model u) v) }
```

That is, if the function returns normally then the result is a one-step reduction of the input, whereas of it raises the exception `Irreducible` then the input term cannot be reduced at all. We wrote and proved four implementations of this function, corresponding to various strategies: innermost or outermost, leftmost or rightmost.

The reduction up to a given number of steps is specified as

```
val reduce_n_steps (u:term_impl) (n:int) :
   (term_impl,int)
  requires { n >= 0 /\ impl_ok u }
  ensures { let (r,n0) = result in
   0 <= n0 <= n /\ term_ok r /\
   beta_red_power n0 u.term_model r.term_model /\
   (n0 < n ->
     forall v:term id. not(beta_red r.term_model v)
```

That is, the result is a pair (r, n_0) such that n_0 is at most n, r is a reduction in n_0 steps of the input, and if $n_0 < n$ then r cannot be reduced anymore.

In all, the generated part plus the additional specifications and code for the interpreter amounts to around 1,000 lines of Why3 source. The number of verification conditions is around 450, and these are all proved by at least one of the automated provers Alt-Ergo, CVC3, CVC4, Eprover and Yices.

Experimental results

To validate the efficiency of the generated verified interpreter, we tested against examples of lambda-terms representing the application of standard functions over integers (exponential, factorial, Ackermann) using several reduction strategies. The integers are represented using Church's encoding, and factorial and Ackermann functions are encoded using a fix-point combinator. In each case, the interpreter was used to normalize the term while computing every intermediate term of the reduction sequence.

	4^6	5^7	7!	8!	ack 2 5	ack 3 1
inner right	29 0.0s	40 1.2s	2173 4.1s	- >5m	10480 4.1s	13793 9.1s
inner left	29 0.0s	40 1.4s	2173 4.8s	- >5m	10480 2.7s	13793 8.0s
outer right	2732 1.9s	- >5m	1618 0.2s	3116 3.3s	25688 2.9s	27480 4.2s
outer left	2732 2.4s	- >5m	30901 48.7s	- >5m	- >5m	- >5m

The table above gives for each test the number of reduction steps and the execution time. The experiments were carried out using a processor Intel Core 2 Duo 2.26 GHz. Notice that for being really efficient, an interpreter of lambda-calculus should not, at each step, restart the search of a redex from the top of the term, as we do here. Our purpose is not to provide the best possible lambda-interpreter, but just to show that our automatically generated term representation and operations are adequate for the practical use.

Also notice that the extraction mechanism of Why3 maps integers into the OCaml implementation of arbitrary precision numbers (library `Big_ints`). We believe we could have a significant speed-up by using a more advanced library such as ZArith (`http://forge.ocamlcore.org/projects/zarith`), or even better, using machine integers and proving that our WhyML code is safe with respect to integer overflow.

6. Case study: a Tableaux-Based Theorem Prover

The tableaux method [15] is a procedure to find a proof of inconsistency for a set of first-order formulas. Basically, it works by decomposing formulas and exploring the possible disjunctive branches, using unification to generate contradictions in those. Existential quantifiers are eliminated by skolemization, and the universal ones are eliminated during decomposition by generating a fresh name for the bound variable, which can get instantiated in unification. If the procedure succeed in deriving contradictions in every branch, then the set of initial formulas is unsatisfiable.

Using our framework, we wrote a simple theorem prover based on the tableaux method and proved its soundness: the main procedure takes a set of first-order formulas as arguments, and if it succeeds, then this set is guaranteed to be unsatisfiable.

In order to develop the prover, it is necessary to formalize first-order logic in this framework. It is done by interpreting formulas over a first-order structure directly into their Why3 equivalent. A first-order structure is represented as a parametrized type containing two interpretations for functions and predicate symbols over the type parameter intended as the domain of the structure.

The theorem prover is developed in several steps:

- Writing and proving a unification procedure for first-order atomic formulas, as unification is the base operation used in order to find contradictions in a tableaux prover. This was the longest step, as unification maintains complex invariants. The implemented algorithm is a variant of Robinson algorithm.
- Proving that unification, as a substitution, preserved the semantics of the branches explored by the prover. This corresponds exactly to the substitution lemma for the semantics of first-order logic.
- Proving satisfiability preservation for skolemization, the preprocessing transformation used in order to remove the existential quantifiers. It was done assuming a variant of the axiom of choice in Why3.
- Proving lemmas about other preprocessing transformations and implications for formula decomposition. As purely propositional transformations/implications were trivial for automated provers, it is only necessary to state a lemma for universal quantifier instantiation.
- Writing the prover itself.

The specification of the prover is that it is correct, in the sense that if it terminates, then the input first-order formula set is unsatisfiable. We do not prove any completeness result, though for our implementation it is likely to be true.

6.1 Automatically Generated Objects

Running our generator on the description of first-order formulas produces around 16,000 lines of Why3 code, including types, logic functions, lemmas and program code (and lemma functions). From these objects, the number of verification conditions is 3,051.

6.2 The Unification Algorithm

On top of the generated Why3 code, we designed a unification algorithm, that we proved correct in the sense that whenever unification of two terms succeeds, returning some substitution σ, then it is true that the models of those terms instantiated by σ coincide. For our purpose, we did not have to prove that our algorithm is complete.

6.3 Semantics of Formulas

To specify our prover, we need to formalize the semantics of first-order logic. It is parametrized by the signature, the interpretation domain and the interpretation of each symbol:

```
type interpretation 'fsymb 'psymb 'var 'dom = {
  interp_fun : 'fsymb -> list 'dom -> 'dom;
  interp_pred : 'psymb -> list 'dom -> bool;
  interp_var : 'var -> 'dom;
}
```

Semantics of terms and formulas are then defined by recursive functions in a natural way, here is an excerpt of the definitions:

```
function term_semantics (t:term 'fsymb 'var)
  (rho:interpretation 'fsymb 'psymb 'var 'dom) : 'dom =
  match t with
  | Var x -> rho.interp_var x
  | FApp f l ->
    rho.interp_fun f (term_list_semantics l rho)
  end

predicate fmla_semantics (f:fmla 'fsymb 'psymb 'var)
  (rho:interpretation 'fsymb 'psymb 'var 'dom) :
  list 'dom =
 match f with
 | Forall f0 -> forall x:'dom.
   let iv = (\ y:option 'var. match y with
             None -> x | Some y -> rho.interp_var y
             end)
   in fmla_semantics f0 { rho with interp_var = iv }
 | And f0 f1 ->
   fmla_semantics f0 rho /\ fmla_semantics f1 rho
 | PApp p l ->
   rho.interp_pred p (term_list_semantics l rho)
 | ...
 end
```

It is important to note that using Why3 types in order to represent the domains of first-order structures is correct only because Why3 types are always non-empty.

6.4 The Prover

Here is the actual specification of the theorem prover.

```
val prove (fl:fmla_list_impl) : unit
 requires { impl_ok_fmla_list fl }
 ensures { forall rho:interpretation int int int 'dom.
  not (fmla_list_semantics (model_fmla_list fl) rho) }
```

Notice that this procedure may not terminate. To produce a reasonably efficient code for finding proofs, we implemented the proof search method known as the *regular connection tableaux* [15].

In all, the number of lines of Why3 code written by hand on top of the generated part is around 6,000. From these additional specifications and code, 4,303 verification conditions are generated. Globally for this case study, the total of 7,354 proof obligations are automatically proved by at least one automated prover among Alt-Ergo, CVC3, CVC4, Eprover, Spass, and Z3, given a limit of at most 20s.

6.5 Experimentation

Since we do not only seek for a certified sound prover but also a reasonably efficient one, we extracted the Why3 code to OCaml and experiment it on a family of problems. After testing it on some simple examples, we evaluated its efficiency on a family of examples of increasing difficulty. We chose the following family

$$(\forall x. R\, x \vee R(f\, x)) \rightarrow \exists x. R\, x \wedge R\,(f^{2n}\, x)$$

from the TPTP library (`http://www.cs.miami.edu/~tptp/`). All these formulas are valid, hence we tried to prove that their negation is unsatisfiable. Here is a summary of results:

n	3	4	5	6
time (sec.)	0.02	0.55	3.36	19.67
nb calls to unification	2,438	69,614	381,542	2,018,934
(per sec.)	122,229	126,814	113,659	102,643
nb of nodes generated	502	9,506	42,898	197,244
(per sec.)	25,134	17,316	12,779	10,028

The first fact is that our prover is significantly slower than state-of-the-art provers which can solve all these problem instances in a fraction of a second. This should not be surprising because the search strategy in our prover is a basic one. Our main goal is to validate the efficiency of the generated code handling binders. This is why we provide the number of calls to unification and the number of proof nodes generated during search. This makes evident that the source of inefficiency is not the representation of terms with binders, but the search strategy. Roughly speaking, our code can perform around 100,000 term unification per seconds and explore 10,000 proof nodes per second. Moreover, the memory consumption remains very low.

7. Related Work

7.1 Meta-Programming with Binders

The genericity of issues related to datatypes with binders in programming lead to several approaches aiming at provided generic support for binders in programming languages. This can take the form of specialized type systems like is the FreshML proposal [25]. It can take the form of meta-programming, in the sense that support for binders can be provided by a layer on top of the considered programming language, like what is proposed by the Cαml code generation tool [23]. A recent approach of meta-programming is proposed by Pouillard [24], under the form of suitable program interface built upon the Agda environment.

Our own tool that generates code from a higher level description clearly belongs to the same family of approaches, this time producing both Why3 specifications and WhyML code. The use of the nested representation in the logic side is inspired by the work of Pouillard.

7.2 Mechanical Reasoning with Binders

The problem of reasoning about datatypes with binders is also a long-standing problem, and its difficulties motivated the design of the POPLmark challenge [3], in order to help comparison of different approaches. In the solutions proposed for that challenge, the locally nameless representation was quite popular (4 solutions, see `http://www.seas.upenn.edu/~plclub/poplmark/`). There is also one solution based on the nested representation by Hirschowitz and Maggesi [16].

The LNgen tool [4] is similar to our own generator. It generates definitions and lemmas for datatypes with binders using the locally nameless representation, for the Coq proof assistant. LNgen does not aim at generating an efficient representation from an implementation point of view.

Going further, the nominal logic by Gabbay et al. [14] is an attempt to incorporate the binders directly into the logic. It is implemented for example in the Nominal-Isabelle framework, with which another solution to POPLmark is provided (Urban et al., see `http://isabelle.in.tum.de/nominal/`). Given that we wanted to use the existing Why3 environment for the proofs, an approach with a different logic was not an option for us.

It may be surprising that we decided to use the nested representation instead of the locally nameless representation, despite the apparent success of the latter. We believe that nested representation solves elegantly the issue of generation of a fresh name when opening a binder, and thus is a great advantage when one wants to perform proofs with automated provers only.

All the approaches mentioned above are built on top of interactive proof assistants. Our approach makes use of automated provers only, and we believe it greatly helped us, in the sense that it significantly shortens the time needed for development of proofs.

7.3 Development of Verified Programs

In a broader scope, there exists a plethora of approaches for development of verified programs. A large effort was put towards verification of code written in general-purpose programming languages: Java (ESC/Java, KeY, KIV, Krakatoa, etc.), C (KeY-C, VCC, Frama-C, etc.), C# (Spec#) and Ada (SparkAda). These projects are specialized in relatively low-level pointer programs, and are typically applied on applications that do not involve symbolic computations, not to mention datatypes with binders. Also, in this context, program verification is made difficult largely because of the intricate semantics of the underlying language.

More recently, several projects were aiming to propose environments for simultaneously programming and proving: Plaid[2] [1, 2], Trellys[3], ATS[4], Guru [26], Fstar[5]. These environments are promising in the sense that since the language is designed with proving in mind, it is usually simpler to achieve the development of a certified code. For example, a verified SAT solver was developed in Guru [21]. We are not aware of any attempt of a generic support for binders in such a context.

[2] `http://www.cs.cmu.edu/~aldrich/plaid.html`

[3] `http://code.google.com/p/trellys/`

[4] `http://www.ats-lang.org/`

[5] `http://research.microsoft.com/en-us/projects/fstar/`

The most famous success stories of development of verified programs are probably the L4-verified project [17] developing a guaranteed secure micro-kernel (proved in Isabelle), and the CompCert project [19] developing a verified C compiler (proved in Coq). As far as we know, there are no binders involved in those projects. Regarding compilers, Flatau [13] and later Myreen and Gordon [20] developed verified Lisp compilers using Nqthm and HOL4, respectively. There were no important issues with binders and variable capture, since a dynamic scope semantics was used in both projects. Kumar et al. developed a verified ML compiler [18], with static scoping, but since the evaluation strategy is call-by-value, this project, too, does not have to deal with binders. Similarly, Chlipala proposed a higher-order encoding of binding structures in order to verify functional language compilers [8, 9] without the usual hassle of binding issues; substitution under binders is not considered in this work.

We believe that our approach using Why3 makes one further step towards making the development of large verified projects (such as state-of-the-art theorem provers) easier, in particular in terms of proof automation.

8. Conclusions and Perspectives

This work presents a new way to prove programs manipulating binders, through a general approach to binders. To validate this approach, we developed and certified two examples of such programs: a lambda-calculus interpreter and a simple tableaux-based theorem prover. The key point here is the separation between the logic and the implementation representation of the binder datatypes, which allows us to reason on a simple structure while using a more efficient one.

This contribution is subject to the usual assumption of a trusted code base that arises when considering mechanical proofs. With our approach, the standalone tool that generates Why3 specifications and code do not need to be trusted: instead, one should be convinced that the generated definition of datatypes and operations (substitution and such) are the ones expected. The generic scheme given in Section 4 is complex, but that is internal to our tool. User sees instances, which are simpler, e.g. for first-order logic, it is not hard to review the generated code, which is reasonably understandable. Then, the part that needs of course to be trusted is the Why3 environment and its external automated provers.

As future work, we could extend the class of supported generic definitions. Currently it is not possible to describe complex binding structures like for pattern-matching, that binds an unknown number of variables. Another possibility would be to extend Why3 in order to generate types with binders directly in Why3, and to get rid both of an external tool and of the need to prove everything from scratch every time.

In the long term, we plan to use this representation in order to prove real-life programs manipulating binders, such as compilers, program verifiers, etc. In particular, every element mentioned above as part of the trusted code base (Why3 itself, SMT provers) is a potential use case for this approach.

Acknowledgments

We gratefully thank J.-C. Filliâtre and A. Charguéraud for their comments and suggestions, and anonymous referees for pointing out additional relevant related work.

References

[1] J. Aldrich. Resource-based programming in Plaid. Fun Ideas and Thoughts, June 2010.

[2] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *OOPSLA*, pages 1015–1022, Oct. 2009.

[3] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *TPHOLs*, number 3603 in LNCS, pages 50–65. Springer, 2005.

[4] B. Aydemir and S. Weirich. LNgen: Tool support for locally nameless representations. Technical report, University of Pennsylvania, 2010. `http://www.cis.upenn.edu/~sweirich/papers/lngen/`.

[5] R. S. Bird and R. Paterson. De Bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, Jan. 1999.

[6] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie*, pages 53–64, August 2011.

[7] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.

[8] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*, pages 143–156. ACM Press, Sept. 2008.

[9] A. Chlipala. A verified compiler for an impure functional language. In *POPL*. ACM Press, Jan. 2010.

[10] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. of the Koninklijke Nederlands Akademie*, 75(5):380–392, 1972.

[11] M. Fernández, I. Mackie, and F.-R. Sinot. Lambda-calculus with director strings. *Applicable Algebra in Engineering, Communication and Computing*, 15(6):393–437, 2005.

[12] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, Mar. 2013.

[13] A. Flatau. *A verified implementation of an applicative language with dynamic storage allocation*. PhD thesis, University of Austin, Texas, 1992.

[14] M. J. Gabbay. Nominal terms and nominal logics: from foundations to meta-mathematics. In *Handbook of Philosophical Logic*, volume 17. Kluwer, 2013.

[15] R. Hähnle. Tableaux and related methods. In *Handbook of Automated Reasoning*, pages 100–178. 2001.

[16] A. Hirschowitz and M. Maggesi. Nested abstract syntax in Coq. *Journal of Automated Reasoning*, 49(3):409–426, 2012.

[17] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[18] R. Kumar, M. O. Myreen, S. Owens, and M. Norrish. CakeML: A verified implementation of ML. In *POPL*, 2014.

[19] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[20] M. O. Myreen and M. J. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In *TPHOLs*, LNCS, pages 359–374. Springer, 2009.

[21] D. Oe, A. Stump, C. Oliver, and K. Clancy. versat: A verified modern SAT solver. In *VMCAI*, volume 7148 of *LNCS*, pages 363–378. Springer, 2012.

[22] R. Pollack, M. Sato, and W. Ricciotti. A canonical locally named representation of binding. *Journal of Automated Reasoning*, 49(2):185–207, 2012.

[23] F. Pottier. *Cαml reference manual*. `http://cristal.inria.fr/~fpottier/alphaCaml/alphaCaml.pdf`.

[24] N. Pouillard. Nameless, painless. *SIGPLAN Not.*, 46(9):320–332, Sept. 2011.

[25] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: programming with binders made simple. *SIGPLAN Not.*, 38(9):263–274, 2003.

[26] A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified programming in Guru. In *PLPV*, pages 49–58. ACM, 2009.