Bindlib, version 4.0
A package for abstract syntax with binder

Christophe Raffalli & Rodolphe Lepigre
Université Savoie Mont Blanc

January 27, 2017

Abstract

Bindlib is an `OCaml` library providing a set of tools for managing data structures with (bound and free) variables. It is very well suited for defining abstract syntax trees. Bindlib includes support for fast substitutions and for the management of variables names. In particular, bound variables are renamed in a minimal way to avoid capture.

Contents

1 Introduction

Data structures with bound and free variables are very common in computer science. For instance, they are required to encode mathematical formulas or computer programs into so-called abstract syntax trees. The manipulation of bound and free variables is hence essential

when it comes to writing compilers or proof assistants. However, implementing the necessary primitives (mainly capture-avoiding substitution and renaming) is cumbersome. Moreover, the simplest implementations often result in poor performance, especially when a lot of substitutions are required. Bindlib aims at providing a library that enables both simplicity and efficiency. It provides high-level tools to deal with binders, substitutions and free variables.

This documentation is in two parts: an informal presentation introducing each concept step by step illustrated with examples. Then, a more formal presentation in the appendix which includes the syntax and an equational semantics of the library.

For our first example, we assume basic knowledge of the λ-calculus which is the simplest data structure with bound variables (Wikipedia has a good introduction on this topic). For the second example, we will give the required mathematical definitions.

2 Using and installing

Bindlib works with `OCaml` version 3.12.1 or later, but it has not been tested with earlier versions. It can be compiled and installed using the command

```
make && make install
```

in the source directory. Alternatively, Bindlib can be installed using the `opam` package manager using the following command.

```
opam install bindlib
```

Several examples of applications (including an implementation of the λ-calculus) are provided in Bindlib's source directory (or in `opam`'s doc directory if you installed via `opam`).

3 Basic principles

The main type constructors and function provided by Bindlib are

- `'a var` represent a variable in the type `'a`.
- `new_var : ('a var -> 'a) -> string -> 'a var` creating a new variable from an initial name (that may change if necessary) and a function whose role is explained below when we introduce `unbox` (it can often be `fun _ -> assert false`).
- `('a,'b) binder` corresponds to a value of type `'b` with one bound variable of type `'a`
- `subst : ('a,'b) binder -> 'a -> 'b` performing substitution of bound variables.
- `'a bindbox` represents a term in constructor with free variables that might become eventually bound. This is the main concept to understand, because value of type `('a,'b) binder` cannot be constructed directly. On must construct them with type `('a,'b) binder bindbox`. Therefore, bindlib provide the following functions to be able to build value of type `'a bindbox`.

  Using a type `'a bindbox` when constructing a data structure with bound variables is the main idea behind bindlib. As an approximation, you can view a value of type `'a bindbox` as a pair with a set of bound variables, and a function that builds a value of type `'a` from the value of all these variabes (this is mostly what the implementation does).

- `box_of_var : 'a var -> 'a bindbox` to use a variable in a value of type `'a bindbox`.
- `bind_var : 'a var -> 'b bindbox -> ('a,b) binder bindbox`.
  to bind a variable in a term.
- Bindlib also provides a few functions to use predefined OCaml functions under the `bindbox` type constructor like:
  - `apply_box : ('a -> 'b) bindbox -> 'a bindbox -> ''b bindbox`
  - `box : 'a -> 'a bindbox`
  - `box_array : 'a bindbox array -> 'a array bindbox`
  - ...
- `unbox : 'a bindbox -> 'a` must be called when the construction is finished. When calling `unbox t`, if `t` still has some free variables (variables not bound using `bind_var`), then the function provided as first argument of `new_var` is used to convert the free variable to a value with a type not using `bindbox`. This function may also raise an exception if we want to ensure that free variables are not allowed.

4 How to use a data structure with variables ?

For now, we will analyse an exemple using a data structure with binders. We will therefore not need the `bindbox` type constructor.

We first define the type of λ-terms as follows.

```
open Bindlib

type term =
  | App of term * term          (* application                    *)
  | Lam of (term, term) binder  (* abstraction (i.e. function)    *)
  | Var of term var             (* free variable                  *)
```

Consider the following function to convert lambda-terms to string:

```
let fVar x = Var(x)

let rec term_to_string = function
  | Var x    -> name_of x
  | Lam b    -> let (x,t) = unbind fVar b in
                "\\" ^ (name_of x) ^ "." ^ (term_to_string t)
  | App(t,u) -> (term_to_string t) ^ "(" ^ (term_to_string u) ^ ")"
```

The function `fVar` is just a synonymous of the type constructor `Var` that can be easily passed in argument to functions like `new_var` or `vbind` because it as the expected type `term var -> term`. If we call `unbox t` when they are still some free variables in `t`, these variables will be placed une the `Var` constructor.

In some case, it could be usefull to use `(fun _ -> assert false)`. For instance if we know that normalisation below will only be used on closed term, this will enforce an assertion if we do not respect this constraint.

A printing is now given. All cases not dealing with bindlib's type constructor should be straightforward. For variables, bindlib manages function name and renaming and provide a function `name_of : 'a var -> string` to get the variable name.

For binder, one may use

```
unbind : ('a,'b) binder -> ('a var -> 'a) -> 'a var *'b
```

The line `let (x,t) = unbind fVar b in` could be replaced by the following lines which are the definition of `unbind`:

```
let name = binder_name f in
let x = new_var name fVar x in
let t = subst f (free_of x) in
```

These lines use the following new functions:

- `binder_name : ('a,'b) binder -> string` to get the name of the bound variable that is kept and managed by bindlib.
- `free_of x` is in fact equivalent to `unbox (box_of_var x)` and therefore in this case to `Var x`.

5 How to construct a data structure with variables ?

To construct data structure with bound variables, we now need the `'a bindbox` type constructor.

Working with `'a bindbox` requires to lift the constructor to this type. Indeed, we need some kind of way to transform the, for instance, the `App` constructor into a term

`app : term bindbox -> term bindbox -> term bindbox`.

The bindlib library provide the necessary function to do that in a few lines:

```
let app : term bindbox -> term bindbox -> term bindbox =
  fun x y -> box_apply2 (fun x y -> App(x,y)) x y
let lam : string -> (term bindbox -> term bindbox) -> term bindbox =
  fun name f -> box_apply (fun x -> Lam(x))
    (let v = new_var fVar name in bind_var v (f (box_of_var v)))
```

Those *smart constructors* are build using the following new functions:

```
bind_var : 'a var  -> 'b bindbox -> ('a,'b) binder bindbox
box_apply : ('a -> 'b) -> 'a bindbox -> 'b bindbox#
box_apply2 : ('a -> 'b -> 'c) -> 'a bindbox -> 'b bindbox -> 'c
```

The key function to bind variables is `bind_var`. Two short cuts are provided:

```
let bind fv name f =
  let v = new_var fVar name in bind_var v (f (box_of_var v))
let vbind fv name f =
  let v = new_var fVar name in bind_var v (f v)
```

Using these short cuts, we could give a shorter definition and an alternative version of `lam`:

```
let lam : string -> (term bindbox -> term bindbox) -> term bindbox =
  fun name f -> box_apply (fun x -> Lam(x)) (bind fVar name f)
let vlam : string -> (term var -> term bindbox) -> term bindbox =
  fun name f -> box_apply (fun x -> Lam(x)) (vbind fVar name f)
```

Depending what we do when we construct a term, we might prefer the bound variables to be directly of type `term bindbox` as in `lam` or of type `term var` as in the latest definition.

Using these *smart constructors*, we can start to define value of type term. We also need the function `unbox : 'a bindbox -> 'a` to finalise the construction.

```
let idt = unbox (lam "x" (fun x -> x))
let delta = unbox (lam "x" (fun x -> app x x))
let omega = App(delta,delta)
```

Remark: in the last definition, because we are not binding any new variables, we do not need to work inside the type `'a bindbox`.

Here is an example, which performs the following transformation on λ-term (it marks all the applications with a variable and bind this variable):

$$\begin{aligned} \mathrm{mark}(t) &= \lambda x.\phi_x(t) \\ \phi_x(y) &= y \\ \phi_x(u\,v) &= x\,\phi_x(u)\,\phi_x(v) \\ \phi_x(\lambda y.u) &= \lambda y.(\phi_x u) \end{aligned}$$

Here is the corresponding code, which illustrates the different use of `lam` and `vlam`:

```
let mark t =
  let rec phi x = function
  | Var(y) -> box_of_var y
  | App(u,v) -> app (app x (phi x u)) (phi x v)
  | Lam(f) -> vlam (binder_name f) (fun y -> phi x (subst f (Var y)))
  in
  unbox (lam "x" (fun x -> phi x t))
```

This code is very similar to the mathematical definition. The use of `binder_name f` allows to use the original name (eventually with a changed suffix).

Here is another example: the computation of the normal form of a term in call-by-name:

```
let rec whnf = function
  App(t1,t2) as t0 -> (
    match (whnf t1) with
    | Lam f -> whnf (subst f t2)
    | t1' ->
       (* a small optimization here when the term is in whnf *)
       if t1' == t1 then t0 else App(t1', t2))
| t -> t

let norm t = let rec fn t =
  match whnf t with
  | Lam f ->
      let (x,t) = unbind fVar f in
      vlam (name_of x) (fun x -> fn t)
  | t ->
      let rec unwind = function
        | Var(x) -> box_of_var x
        | App(t1,t2) -> app (unwind t1) (fn t2)
        | t -> assert false
      in unwind t
in unbox (fn t)

let _ = print_endline (term_to_string (norm (App(delta,idt))))
```

This program uses one function to compute the "weak head normal form", which is used in the second function to compute the full normal form.

6 Naming of variables

The bindlib library uses `string` for variable names. Names are considered as the concatenation of a prefix and a possibly empty suffix. The suffix is the longest terminal substring of the name composed only of digits.

Example: in `"toto0"` the suffix is `"0"`.

To choose the initial name of a variable, you passe it to `new_var`, `bind` or `vbind` as second argument.

When binding `x` using `bind_var v t`, the suffix of the name may be changed to avoid name conflict. To access the name of variables, bindlib provides the following functions:

- `name_of : 'a var -> string` to access the name of a free variable.
- `binder_name : ('a,'b) binder -> string` to access the name of a bound variable.

Using bindlib you are sure that bound variables are renamed to avoid variable conflict. But this is not enough:

1. Distinct free variables may have the same name. Renaming of free variables cannot be done automatically because there is no way to know the variables that are used in the same "context".
2. The `subst` function does not perform renaming. Therefore, it is only just after the call to `unbox` that the bound variables are named correctly. If you use the result of `unbox` and perform substitution then, the naming may become incorrect.

   This cannot be avoided if one want a reasonable complexity for substitution.

   In the above example, there is no problem when printing the resulf of `norm t`, because the term is fully reconstructed. However, if we call `whnf t` and if `t` is not closed, the name stored in the result may be incorrect if the free variables are moved under binders with the same name.
3. By default, bindlib perform minimal renaming when binding variables. This means it accepts name collision in `fun x -> fun x -> x` that you might prefer printed as `fun x -> fun x0 -> x0`.

To solve these problems, bindlib provides an abstract notion of context which are "sets" of free variables which should have distinct names.

- `type ctxt` is an abstract type.
- `empty_ctxt : ctxt` is the initial empty context.
- `new_var_in : ctxt -> ('a var -> 'a) -> string -> 'a var * ctxt`
  which is the same as `new_var`, except that it receives a context, renames the variable so that its name is not used in the context and returns a new context where this variable was added.
- There exists also two variants of `bind` and `vbind` named `bind_in` and `vbind_in`

This fixes point (1). For the two other points, there are two solutions, depending if you prefer minimal renaming or the so called Barendregt convention (no bound variable should have the same name than a free variable, even is the later does not occur in the scope of the former).

If you want to follow Barendregt convention, this is easy, your printing functions should use a `ctxt` as in:

```
let rec term_to_string2 ctxt = function
  | Var x    -> name_of x
  | Lam b    -> let (x,t,ctxt) = unbind_in ctxt fVar b in
                "\\" ^ (name_of x) ^ "." ^ (term_to_string2 ctxt t)
  | App(t,u) -> (term_to_string2 ctxt t) ^ "(" ^ (term_to_string2 ctxt u) ^ ")"
```

If you prefer minimal renaming, you have nothing to do if you are certain that no substitution have been performed. Otherwise, you need what we call a "lifting" or "boxing" function to copy the data structure before printing as in the following example:

```
let rec lift_term = function
  | Var(y) -> box_of_var y
  | App(u,v) -> app (lift_term u) (lift_term v)
  | Lam(f) ->
      vlam (binder_name f) (fun x -> lift_term (subst f (Var x)))

let term_to_string3 t =
  term_to_string (unbox (lift_term t))
```

7 A more complete example and advanced features

This section covers advanced feature of the library. The reader is advised to read, practice and understand the previous section before reading this.

We will consider second order predicate logic. We chose this example, because the definition of second-order substitution is non trivial ... and this is a very good example for the power of `bindlib`.

Here is the mathematical definition of terms and formulas, and the corresponding definition using `bindlib`:

Definition 1 (Syntax of second order logic) We assume a signature

$$\Sigma = \{(f, 1), (g, 2), (a, 0), \ldots\}$$

with various constants and function symbols of various arity. An infinite set of first-order variables (written $x, y, z \ldots$) and for each natural number n an infinite set of second-order variables of arity n (written $X, Y, Z, \ldots$).

Terms are defined by

- x is a term if it is a first order variable
- $f(t_1, \ldots, t_n)$ is a term if f is a function symbol of arity n and if $t_1, \ldots, t_n$ are terms.

Formulas are defined by

- $X(t_1, \ldots, t_n)$ is a formula if X is a second order variable of arity n and if $t_1, \ldots, t_n$ are terms.
- $A \to B$ is a formula if A and B are formulas.
- $\forall x\, A$ is a formula with x bound if A is a formula and x is a first-order variable.
- $\forall X\, A$ is a formula with X bound if A is a formula and X is a second-order variable.

We now give the type for terms and formulas and their smart constructor.

```
open Bindlib

(* A structure representing a function symbol. *)
type symbol = { name : string ; arity : int }

(* The type of first order terms. *)
type term =
  | Var of term var
  | Fun of symbol * term array

(* The type of formulas. *)
type form =
  (* Implication. *)
  | Imply of form * form
  (* First-order universal quantification. *)
  | Univ1 of (term, form) binder
  (* Second-order universal quantification. *)
  | Univ2 of int * (pred, form) binder
  (* Variable. *)
  | FVari of pred var * term array

(* Predicate (implemented as a binder). *)
and pred = (term, form) mbinder

let imply = box_apply2 (fun f g -> Imply(f,g))

let univ1 = box_apply (fun f -> Univ1 f)

let univ2 arity = box_apply (fun f -> Univ2(arity,f))

let fvar1 : term var -> term = fun x -> Var x

let fvar2 : int -> pred var -> pred = fun arity x ->
  let vs = Array.init arity (Printf.sprintf "x%i") in
  let f xs = box_apply (fun y -> FVari(x,y)) (box_array xs) in
  unbox (mbind fvar1 vs f)

let unbind1 : (term, form) binder -> term var * form =
  unbind (fun x -> Var x)

let unbind2 : int -> (pred, form) binder -> pred var * form = fun a b ->
  unbind (fvar2 a) b
```

Let us review these definitions:

- `pred = (term, form) mbinder`: this is the type of an object of type `form` with an array of bound variables. We use this to represent predicates, that is formula with n parameters.

- `Univ2 of int * (pred, form) binder`: for second order quantification, we bind a variable of arity n. The arity is the first argument of the constructor. For the second argument, `(pred, form) binder`, we mean that we bind a "predicate" variable. This variable is itself a `mbinder`, this is why it is a "second-order" variable.

- `FVari of pred var * term array`: as usual for any free variable, we have to store the variable itself of type `pred var`. But here, we should also store the terms which are the arguments of the second-order predicate variable.

- `fvar1 : term var -> term = fun x -> Var x`: we introduce the function to construct first-order variable. Every time we will create a first order variable we will use `new_var fvar1`.

- `fvar2 : int -> pred var -> pred`: is the anlogous function for second order variables. It is more complex, because a second order variable is a binder. Mathematically, we provide the arity n of the variable and the predicate variable X and we build the predicate $\lambda x_1, \ldots, x_n X(x_1, \ldots, x_n)$.

  We therefore must construct a multiple binder with `mbind` decribed below to bind the variables $x_1, \ldots, x_n$. And we need the function `box_array` to build passe the argument to X, as second argument to the constructor `FVari`.

  This seems complex, but in general, the composition of function is not that difficult to get right thanks to the types that allow in fact very few simple combinations.

- `mbind : ('a var -> 'a) -> string array ->`
  `('a bindbox array -> 'b bindbox) -> ('a,'b) mbinder bindbox`
  is the main function to build multiple binder. The array of names give the name of the bound variables and the arity of this multiple binder.

- `box_array` : this is a function of type `'a bindbox array -> 'a array binxbox`, which is often used when binding array of variables.

Now, we write the printing function for terms and formulas:

```
let print_array print_elt sep och a =
  let f i e =
    let pref = if i = 0 then "" else sep in
    Printf.fprintf och "%s%a" pref print_elt e
  in
  Array.iteri f a

let rec print_term : out_channel -> term -> unit = fun och t ->
  match t with
  | Var(x)   -> output_string och (name_of x)
  | Fun(s,a) -> Printf.fprintf och "%s(%a)" s.name
                (print_array print_term ",") a

let rec print_form : out_channel -> form -> unit = fun och f ->
  match f with
  | Imply(a,b) -> Printf.fprintf och "(%a) => (%a)" print_form a print_form b
  | Univ1(b)   -> let (x,f) = unbind1 b in
                  Printf.fprintf och "All1 %s.(%a)" (name_of x) print_form f
  | Univ2(a,b) -> let (x,f) = unbind2 a b in
                  Printf.fprintf och "All2 %s.(%a)" (name_of x) print_form f
```

```
  | FVari(x,a) -> Printf.fprintf och "%s(%a)" (name_of x)
                    (print_array print_term ",") a
```

We also write the equality test which is similar:

```
let eq_arrays : ('a -> 'a -> bool) -> 'a array -> 'a array -> bool =
  fun eq_elt a1 a2 ->
    let l = Array.length a1 in
    if Array.length a2 <> l then false else
    let eq = ref true in
    let i = ref 0 in
    while !i < l && !eq do
      eq := !eq && eq_elt a1.(!i) a2.(!i); incr i
    done; !eq

let rec equal_term t u =
  match (t, u) with
  | (Var(x)      , Var(y)      )              -> compare_vars x y = 0
  | (Fun(s1,ta1), Fun(s2,ta2)) when s1 = s2 -> eq_arrays equal_term ta1 ta2
  | _                                        -> false

let rec equal_form f g =
  match (f, g) with
  | (Imply(f1,g1), Imply(f2,g2)) -> equal_form f1 f2 && equal_form g1 g2
  | (Univ1(b1)   , Univ1(b2)   ) ->
      let x = free_of (new_var fvar1 "") in
      equal_form (subst b1 x) (subst b2 x)
  | (Univ2(a1,b1), Univ2(a2,b2)) when a1 = a2 ->
      let x = free_of (new_var (fvar2 a1) "") in
      equal_form (subst b1 x) (subst b2 x)
  | (FVari(x,a1), FVari(y,a2)) ->
      compare_vars x y = 0 && eq_arrays equal_term a1 a2
  | _                          -> false
```

One remark here: we do not use the variables names for comparison, it could be wrong as renaming is often necessary. We use the provided function `eq_vars` or `compare_vars`. Remark: because of the function `copy_var`, physical equality and `eq_vars` are not equivalent.

It is also a bad idea to use structural equality because one of the field of the structure is an ML closure (the function of type `'a var -> 'a` given when creating the variable).

Now, we give the boxing functions (already mentioned about naming): very often, we have an object `o` of type `t bindbox` that we want to read/match. Therefore, we will use `unbox`. But then, we will want to reuse the subterms of `o` with type `t bindbox` to continue the construction of an object of type `t` with some bound variables. For this, we need this kind of copying functions:

```
let rec box_term : term -> term bindbox = function
  | Var(x)    -> box_of_var x
  | Fun(s,ta) -> let ta = Array.map box_term ta in
                 box_apply (fun a -> Fun(s,a)) (box_array ta)

let rec box_form : form -> form bindbox = function
  | Imply(a,b) -> box_apply2 (fun a b -> Imply(a,b)) (box_form a) (box_form b)
  | Univ1(b)   ->
      let (x,a) = unbind1 b in
      box_apply (fun b -> Univ1 b) (bind_var x (box_form a))
  | Univ2(a,b) ->
      let (x,f) = unbind2 a b in
```

```
      box_apply (fun b -> Univ2(a,b)) (bind_var x (box_form f))
  | FVari(x,a) ->
      mbind_apply (box_of_var x) (box_array (Array.map box_term a))
```

The functions `bind_apply: ('a->'b) binder bindbox -> 'a bindbox -> 'b bindbox` and `mbind_apply: ('a->'b) mbinder bindbox -> 'a bindbox array -> 'b bindbox` for multiple binder are used to apply to its arguments a variable representing a binder.

Now we define proofs (for natural deduction) and their smart constructors :

```
type proof =
  | Imply_i of form * (proof, proof) binder
  | Imply_e of proof * proof
  | Univ1_i of (term, proof) binder
  | Univ1_e of proof * term
  | Univ2_i of int * (pred, proof) binder
  | Univ2_e of proof * pred
  | Axiom   of form * proof var

let imply_i = box_apply2 (fun p q -> Imply_i(p,q))
let imply_e = box_apply2 (fun p q -> Imply_e(p,q))
let univ1_i = box_apply (fun p -> Univ1_i p)
let univ1_e = box_apply2 (fun p q -> Univ1_e(p,q))
let univ2_i arity = box_apply (fun p -> Univ2_i(arity,p))
let univ2_e = box_apply2 (fun p q -> Univ2_e(p,q))
let axiom f v = Axiom(f,v)
```

This rule correspond to proof in natural deduction with an elimination rule and an introduction rule for each connective and the axiom rule. The type checking algotihm below shoud make the meaning of the rule clear.

We remark that all introduction rules are binder and the implication introduction rule binds a proof inside a proof. For the function `assume` constructing the variables of type `proof`, we also store the formula that it "assumes" when we do the introduction of an implication.

Now, we give a simple function to print goals (or sequent), that is a list of named hypotheses, represented by proof variables, and a conclusion. We need to copy the hypotheses because they result from a substitution. This is not the case for the conclusion of the sequent which is passed to `print_goal` just after the call to `unbox`.

```
let print_goal och hyps concl =
  let print_hyp och = function
    | Axiom(a,x) -> Printf.fprintf och " %s : %a\n" (name_of x) print_form a
    | _          -> failwith "Not an axiom..."
  in
  List.iter (print_hyp och) hyps;
  output_string och "————————————————————————————————————————\n";
  Printf.fprintf och " %a\n" print_form concl
```

The main function, that checks if a proof is correct. The first function builds the formula which is proved by a proof, or raise the exception `Bad_proof` if the proof is incorrect. The second function calls the first one and checks if the produced formula is equal to a given formula.

Moreover, to illustrate the problem of variables names, we print the goal which is obtained after each rule and use a context to rename the variables and respect Barendregt convention.

Here is the code that we will explain below:

```
exception Bad_proof of string
```

```
let unbind1_in ctxt = unbind_in ctxt (fun x -> Var x)

let unbind2_in ctxt a = unbind_in ctxt (fvar2 a)

let type_infer p =
  let ctxt = empty_ctxt in
  let rec fn hyps ctxt p =
    let r = match p with
      | Imply_i(f,p) ->
        let ax, ctxt = new_var_in ctxt (axiom f) (binder_name p) in
        let p' = subst p (free_of ax) in
        imply (box_form f) (fn (ax::hyps) ctxt p')
      | Imply_e(p1,p2) ->
        begin
          let f1' = unbox (fn hyps ctxt p2) in
          match unbox (fn hyps ctxt p1) with
          | Imply(f1,f2) when equal_form f1 f1' -> box_form f2
          | Imply(f1,f2) ->
             Printf.eprintf "%a <> %a\n%!" print_form f1 print_form f1';
             raise (Bad_proof("Imply"))
          | _ ->
             raise (Bad_proof("Imply"))
        end
      | Univ1_i(p) ->
        let x,t,ctxt = unbind1_in ctxt p in
        univ1 (bind_var x (fn hyps ctxt t))
      | Univ1_e(p,t) ->
        begin
          match unbox (fn hyps ctxt p) with
          | Univ1(f) -> box_form (subst f t)
          | _ -> raise (Bad_proof("Univ1"))
        end
      | Univ2_i(arity,f) ->
        let x,t,ctxt = unbind2_in ctxt arity f in
        univ2 arity (bind_var x (fn hyps ctxt t))
      | Univ2_e(p,pred) ->
        begin
          match unbox (fn hyps ctxt p) with
          | Univ2(arity, f) when arity = mbinder_arity pred ->
              box_form (subst f pred)
          | _ -> raise (Bad_proof("Univ2"))
        end
      | Axiom(f,_) -> box_form f
    in print_goal stdout (List.map free_of hyps) (unbox r); r
  in
  unbox (fn [] ctxt p)

let type_check p f =
  if not (equal_form (type_infer p) f) then
    raise (Bad_proof "Infered type does not match expected type.")
```

There are two important things to comment in these programs:

1. We care about variable names using a context for the free variables .

2. The second important point is the use of `unbox` together with the `box_form` and `box_term` functions to type-check the elimination rules.

We must use `unbox` to match the formula coming from the type-checking of the principal premise of the rule. Then, one sub-formula of the matched formula must be used and we have to use `box_form` for that. Important remark: because of the use of `box_term` and `box_form` functions, this algorithm is quadratic (at least), because it calls `box_term` and `box_form` which are linear at each elimination rule. As an exercise, the reader could rewrite the `type_infer` function, using a stack, to avoid this!

It is in fact a general problem when writing programs using bound variables, we have often to make copy of objects (to adjust DeBruijn indices, to rename variables, to "relift" them). And this is important that `bindlib` allow to easily notice this when using the "lifting" functions and to allow to try to avoid them in a lot of cases, bringing a substantial gain in efficiency.

8 Semantics (work in progress)

Here is an equational specification of bindlib. To give the semantics, we will use the following convention:

- variables are structure of type

  `'a var = {id : int; name : string; f : 'a var -> 'a}`

  that are produced only by a function

  `new_var : ('a var -> 'a) -> string -> 'a var`

  which always generates fresh `id`.

- We will use values written $\texttt{unbox}_e$ where e is an association list associating to a value of type `'a var` a value of type `'a`. We will write $e(v)$ for the value of a variable v in the environment e.

  In fact e has type `env` $= \exists$`('a.'a var * 'a) list`. This is now a valid ML type thanks to GADT, but this is another story.

- Value of type `ctxt` will be set of strings equiped with a function

  `fresh : string -> ctxt -> string * ctxt` such that

  `fresh` $s\,c = s', c'$ where `s'` is not member of c, has the same prefix that s (only the numerical suffix of s is changed) and c' is the addition of s' to the set c.

- In the semantics, we also use `map`, the standard map function on array (`Array.map`), and `fold_map : ('a -> 'b -> 'c * 'b) -> 'a array -> 'b -> 'c array * 'b` which definition follows

```
let fold_map f tbl acc =
  let acc = ref acc in
  let fn x =
    let x', acc' = f x !acc in
    acc := acc';
    x'
  in
  let tbl' = Array.map fn tbl in
  tbl', !acc
```

$$
\begin{array}{rcl}
\texttt{unbox} & = & \texttt{unbox}_{\texttt{[]}} \\
\texttt{unbox}_e\texttt{(box}\ v\texttt{)} & = & e(v) \\
\texttt{unbox}_e\texttt{(apply\_box}\ f\ v\texttt{)} & = & \texttt{(unbox}_e\ f\texttt{)(unbox}_e\ v\texttt{)} \\
\texttt{unbox}_e\texttt{(bind\_apply}\ f\ v\texttt{)} & = & \texttt{(unbox}_e\ f\texttt{)(unbox}_e\ v\texttt{)} \\
\texttt{new\_var}\ c\ f\ s & = & \texttt{let}\ s', c = \texttt{fresh}s\ c\ (\texttt{=}\ \texttt{new\_var}\ f\ s, c) \\
\texttt{name\_of}\ v & = & v\texttt{.name} \\
\texttt{subst(unbox}_e\texttt{(bindvar}\ v\ \texttt{in}f\texttt{))}a & = & \texttt{unbox}_{(v,a)::e}f \\
\texttt{unbox}_e\texttt{(bindbox\_of}\ v\texttt{)} & = & \texttt{try assoc}\ v\ e\ \texttt{with Not\_found ->}\ v\texttt{.f}\ v \\
\texttt{unbox}_e\texttt{(box\_pair(}a_1, \ldots, a_n\texttt{))} & = & \texttt{(unbox}_e\ a_1, \ldots, \texttt{unbox}_e\ a_n\texttt{)} \\
\texttt{unbox}_e\texttt{(box\_list[}a_1; \ldots; a_n\texttt{])} & = & \texttt{[unbox}_e\ a_1; \ldots; \texttt{unbox}_e\ a_n\texttt{]} \\
\texttt{unbox}_e\texttt{(box\_array[|}a_1; \ldots; a_n\texttt{|])} & = & \texttt{[|unbox}_e\ a_1; \ldots; \texttt{unbox}_e\ a_n\texttt{|]}
\end{array}
$$

Figure 1: Equational semantics for bindlib