Bindlib, version 4.0
A package for abstract syntax with binder

Christophe Raffalli & Rodolphe Lepigre
Université Savoie Mont Blanc

June 10, 2016

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 for 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 examples, 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 types

The main type constructor provided by Bindlib is `('a,'b) binder`. It corresponds to a value of type `'b` with one bound variable of type `'a`. Intuitively, it is very similar to the type `'a -> 'b` used when using higher-order abstract syntax. For free variables, bindlib provides a type `'a variable` for free variable of type `'a`. Using these types, we can define the type of λ-terms as follows.

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

4 How to use a data structure with variables ?

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 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.

Printing application is immediate, just recursive calls. 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 variable -> string` to get the variable name.

For binder, one may use

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

A natural question arises: what is the role of the first argument of `unbind`. It is mandatory when constructing a value of type `'a variable` to provide a function of type `'a variable -> 'a`, which is used when we need to consider the variable as an object of type `'a`.

In the above example, `fVar` is not used and it is possible to replace it by `(fun _ -> assert false)`.

The function `let (x,t) = unbind fVar b in` could be replaced by the following lines:

```
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 functions:

- `binder_name : ('a, 'b) binder -> string` to get the name of the bound variable that is kept and managed by bindlib.
- `new_var` is a function to create a new variable. The name you give is only an indication, it might be changed by bindlib if necessary, but will be always kept as prefix.
- `free_of x` is in fact equivalent to `fVar x`.
- `subst : ('a, 'b) binder -> 'a -> 'b` is finally use to substitute a value in place of the bound variable in `f`.

5 How to construct a data structure with variables ?

To construct data structure with bound variables, we provide a new type

`'a bindbox`

which is the type of a data structure of type `'a` under construction. In fact this type contructor defines a monad. If you do not know what a monad is just consider this is the notion of morphism associated to typed programming language. If you do not know what a morphism is, then just ignore this paragraph !

Using a type `'a bindbox` when constructing a data structure with bind variable 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 buid a value of type `’a` from the value of all these variabes.

Working with `’a bindbox` requires to lift the constructor to this type. Indeed, we need a some kind of way to transform the `App` constructor into a term of type `’a bindbox -> ’a bindbox -> ’a bi` 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 (free_of v)))
```

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

```
box : ’a -> ’a bindbox
apply_box : (’a -> ’b) bindbox -> ’a bindbox -> ’b bindbox
bind_var : ’a variable  -> ’b bindbox -> (’a, ’b) binder bindbox
box_apply : (’a -> ’b) -> ’a bindbox -> ’b bindbox#
box_apply2 : (’a -> ’b -> ’c) -> ’a bindbox -> ’b bindbox -> ’c
```

Remark: it you are used to monad, `box` and `apply_box` are the `return` and `bind` operation of the monad. We did not use the name `bind` ... Because this name is not appropriate hare. This means that `box_apply` and `box_apply2` can easily be defined from `box` and `apply_box`.

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

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

Using these short cut, 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 variable -> 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 variable to be directly of type `term bindbox` as in `lam` or of type `term variable` 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 another example, which performs the following transformation on λ-term (it marks all the application with a variable and the bind this variable):

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

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:

```
(* weak head normal form *)
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

(* call by name normalisation, all step at once *)
let norm t = let rec fn t =
  match whnf t with
  | Lam f ->
      let (x,t) = unbind fVar f in
      vlam (binder_name f) (fun x -> fn (subst f (Var x)))
  | 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)
```

This function is very similar to the previous one, with 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 use the `"as"` keyword in

```
letvar f x as name in t
```

This means that `name_of x` will return `name`. However, when binding `x` using `bindvar x in u`, the suffix of the name may be changed to avoid name conflict. In `letvar f x in t`, the default name for `x` will be `"x"` (that is the string build from the name of the identifier).

To access the name of variables, bindlib provides the following functions:

- `name_of : 'a variable -> string` to access the name of a free variable.
- `binder_name : ('a, 'b) variable -> string` to access the name of a bound variable.
- `match t with bind f x in u -> ...`: if `t : ('a, 'b) binder`, then we have `x` of type `'a variable` and `name_of x` will return the same value as `binder_name t`.

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 can not 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 can not be avoided if one want a reasonable complexity for substitution.
3. By default, bindlib perform minimal renaming. 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 context` is an abstract type.
- `empty_context : context` is the initial empty context.
- `letvar f x as name for ctxt in ...` The argument of `for` must be an Ocaml identifier of type `context`. The name given with `as name` may be changed (only the suffix is changed) into a name not already in `ctxt`. Then, the identifier `ctxt` is rebound, under the scope of `letvar`, to an extended context containing the new variable name.

  This construct can also be used with the `bind f x as name for ctxt in ...` camlp4 extension.

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 `context` as in:

```
let rec print_term ctxt = function
  App(t1,t2) ->
    print_string "(";
    print_term ctxt t1;
    print_string " ";
    print_term ctxt t2;
    print_string ")"
| Abs f ->
    match f with bind fVar x for ctxt in t ->
      print_string "fun ";
      print_string (name_of x);
      print_string " ";
      print_term ctxt t
| FVar(v) ->
    print_string (name_of v)
```

If you prefer minimal renaming (which is required when you really want to refer to names of bound variables), you have nothing to do if you are certain that no substitution have been performed. Otherwise, you need what I call a “lifting” function to copy the data structure before printing as in the following example:

```
let rec lift_term = function
    FVar(y) -> bindbox_of y
  | App(u,v) -> App(^lift_term u, lift_term v^)
  | Abs(f) ->
      match f with bind fVar x in u ->
        Abs(^ bindvar x in lift_term u ^)

let print_term t =
  let rec fn = function
      App(t1,t2) ->
        print_string "(";
        fn t1;
        print_string " ";
        fn t2;
        print_string ")"
    | Abs f ->
        match f with bind fVar x in t ->
          print_string "fun ";
          print_string (name_of x);
          print_string " ";
          fn t
    | FVar(v) ->
        print_string (name_of v)
  in
  fn (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, practise 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.

```
(* a structure to store the information about a function symbol *)
type symbol = {name : string; arity : int }

(* the type of first order term *)
type term =
    Fun of symbol * term array
  | TermVar of term variable
```

```
(* the type of second order formula *)
type form =
  Imply of form * form
| Forall1 of (term, form) binder         (* fitst order quantifier *)
| Forall2 of int * (pred, form) binder   (* second order quantifier *)
                                         (* the int in the arity *)
| FormVar of pred variable * term array  (* a predicate variables and *)
                                         (* its arguments *)

and pred = (term, form) mbinder          (* a predicate is a binder ! *)

let lam1 x = TermVar(x)

let lam2 n x =
  unbox (bind lam1 args(n) in FormVar(^ (^x^), lift_array args^))
```

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.
- `Forall2 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.
- `FormVar of pred variable * term array`: as usual for any free variable, we have to store the variable itself of type `pred variable`. But here, we should also store the terms which are the arguments of the second-order predicate variable.
- `let lam1 x = TermVar(x)`: we introduce the function to construct first-order binder. Every time we will want to construct a first-order quantification, we will write

  `Forall1(^ bind lam1 x in ... ^)`
- `bind lam1 args(n) in ...` : this binds an array of `term variable` of arity `n`. The typing rule of this camlp4 extension is:

$$\frac{\begin{array}{l}\Gamma, \texttt{x : 'a array bindbox} \vdash \texttt{e : 'b bindbox}\\ \Gamma \vdash \texttt{f : 'a variable -> 'a}\\ \Gamma \vdash \texttt{n : int}\end{array}}{\Gamma \vdash \texttt{bind f x(n) in e : ('a, 'b) mbinder bindbox}}$$

- `lift_array` : this is a function of type `'a bindbox array -> 'a array binxbox`, which is often used when binding array of variables.
- `let lam2 n x =`
  `unbox (bind lam1 args(n) in FormVar(^ (^x^), lift_array args^))`: we introduce the function to construct second-order binder. It is a bit complex, but typing gives very little choice: we have `arity : int`, and `x : pred variable`. And we need an object of type `pred`. We have:

- x : pred variable and therefore, (^x^) : pred variable bindbox
- args : term bindbox array gives lift_array args : term array bindbox
- all this gives FormVar(^ ... ^) : form bindbox, using the rule given page ?? for lifted constructor.
- bind lam1 args(arity) in FormVar(^ ... ^) : pred bindbox.
- Finally, unbox (bind lam1 args(arity) in FormVar(^ ... ^)) : pred

Like for first-order variables, every time we will want to construct a second-order quantification of arity n, we will write Forall2(^ unit n, bind lam2 x(n) in ... ^).

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

```
let rec print_term = function
    Fun(sy, ta) ->
      print_string sy.name;
      print_string "(";
      for i = 0 to sy.arity - 1 do
        print_term ta.(i);
        print_string (if i < sy.arity - 1 then "," else ")")
      done
  | TermVar(var) ->
      print_string (name_of var)

let rec print_form lvl = function
    Imply(f1, f2) ->
      if lvl > 0 then print_string "(";
      print_form 1 f1; print_string " => "; print_form lvl f2;
      if lvl > 0 then print_string ")";
  | Forall1 f ->
      match f with bind lam1 t  in g ->
        print_string "Forall1 ";
        print_string (name_of t);
        print_string " ";
        print_form 1 g
  | Forall2 (arity, f) ->
      match f with bind (lam2 arity) x in g ->
      print_string "Forall2 ";
      print_string (name_of x);
      print_string " ";
      print_form 1 g
  | FormVar(var,args) ->
      print_string (name_of var);
      print_string "(";
      let arity = Array.length args in
      for i = 0 to arity - 1 do
        print_term args.(i);
        print_string (if i < arity - 1 then "," else ")")
      done
```

We also write the equality test which is similar:

```
let rec equal_term t t' = match t, t' with
    TermVar(x), TermVar(x') -> x == x'
  | Fun(sy,ta), Fun(sy',ta') when sy = sy' ->
      let r = ref true in
      for i = 0 to sy.arity - 1 do
        r := !r && equal_term ta.(i) ta'.(i)
      done;
      !r
  | _ -> false

let rec equal_form f f' = match f, f' with
    Imply(f,g), Imply(f',g') ->
      equal_form f f' && equal_form g g'
  | Forall1(f), Forall1(f') ->
      match f with bind lam1 t in g ->
      equal_form g (subst f' (free_of t))
  | Forall2(arity,f), Forall2(arity',f') ->
      arity = arity' &&
      match f with bind (lam2 arity) x in g ->
      equal_form g (subst f' (free_of x))
  | FormVar(x,ta), FormVar(x',ta') ->
      x == x' &&
      let r = ref true in
      for i = 0 to Array.length ta - 1 do
        r := !r && equal_term ta.(i) ta'.(i)
      done;
      !r
  | _ -> false
```

One remark here: we do not use the variables names for comparison (because it is slower and for another reason that we will see later), but instead we use physical equality on the free variables we create to substitute to bound variables. It is possible to use structural equality because a variable is a structure whose first field is a unique identifier. But you should be aware that one of the field of this structure is an ML closure (the function of type `'a variable -> 'a` given when creating the variable).

Now, we give the lifting 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 lift_term = function
    TermVar(x) -> bindbox_of x
```

```
  | Fun(sy,ta) -> Fun(^ (^sy^), lift_array (Array.map lift_term ta) ^)

let rec lift_form = function
    Imply(f1,f2) -> Imply(^ lift_form f1, lift_form f2 ^)
  | Forall1 f ->
      match f with bind lam1 t in g ->
       Forall1(^ bindvar t in lift_form g ^)
  | Forall2(arity,f) ->
      match f with bind (lam2 arity) x in g ->
      Forall2(^ (^arity^), bindvar x in lift_form g ^)
  | FormVar(x,args) ->
      mbind_apply (bindbox_of x) (lift_array (Array.map lift_term args))
```

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):

```
type proof =
    Imply_intro of form * (proof,proof) binder
  | Imply_elim of proof * proof
  | Forall1_intro of (term, proof) binder
  | Forall1_elim of proof * term
  | Forall2_intro of int * (pred, proof) binder
  | Forall2_elim of proof * pred
  | Axiom of form * proof variable

let assume f x = Axiom(f,x)
```

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 hyps concl =
  List.iter (
  function
      Axiom(f, v) ->
        print_string (name_of v); print_string ":=";
        print_form 0 (unbox (lift_form f)); print_newline ()
    | _ ->
        failwith ("not an axiom")) hyps;
  print_string "  |- "; print_form 0 concl; print_newline ()
```

Now 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.

Here is the code that we will explain bellow:

```
exception Bad_proof of string

let type_infer p =
  let ctxt = empty_context in
  let rec fn hyps ctxt p =
    let r = match p with
      Imply_intro(f,p) ->
        match p with bind (assume f) ax for ctxt in p' ->
        Imply(^ lift_form f, fn (ax::hyps) ctxt p' ^)
    | Imply_elim(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' -> lift_form f2
          | Imply(f1,f2) ->
              print_form 0 f1; print_string "<>"; print_form 0 f1';
              print_newline ();
              raise (Bad_proof("Imply"))
          | _ ->
              raise (Bad_proof("Imply"))
        end
    | Forall1_intro(p) ->
        match p with bind lam1 t for ctxt in p' ->
        Forall1(^ bindvar t in  fn hyps ctxt p'^)
    | Forall1_elim(p,t) ->
        begin
          match unbox (fn hyps ctxt p) with
            Forall1(f) -> lift_form (subst f t)
          | _ -> raise (Bad_proof("Forall1"))
        end
    | Forall2_intro(arity, f) ->
        match f with bind (lam2 arity) x for ctxt in p' ->
        Forall2(^ (^arity^),  bindvar x in fn hyps ctxt p' ^)
    | Forall2_elim(p,pred) ->
        begin
          match unbox (fn hyps ctxt p) with
n            Forall2(arity, f) when arity = mbinder_arity pred ->
              lift_form (subst f pred)
          | _ -> raise (Bad_proof("Forall2"))
        end
    | Axiom(f,_) ->
```

```
        lift_form f
    in
    print_goal (List.map free_of hyps) (unbox r); print_newline ();
    r
  in
  unbox (fn [] ctxt p)

let type_check p f =
  if not (equal_form (type_infer p) f) then raise (Bad_proof "conclusion")
```

There are two important things to comment in these programs:

1. We care about variable names using a context for the free variables and the `lift_form` function for the bound ones as explained before.

2. The second important point is the use of `unbox` together with the `lift_form` function 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 `lift_form` for that. Important remark: because of the use of `lift_term` and `lift_form` functions, this algorithm is quadratic (at least), because it calls `lift_term` and `lift_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 avoid them in a lot of cases, bringing a substantial gain in efficiency.

8 Semantics

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

- variables are structure of type

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

  that are produced only by a function

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

  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 variable` a value of type `'a`.

  In fact e has type `env` $= \exists$`('a.'a variable * 'a) list`.

  This is not a valid ML type, but it could be coded in ML. However, we will use a search function `assoc` (searching for variables) which should have type `'a variable -> env -> 'a` which is not possible in ML. However, in our case, this is type safe only because the same

$$
\begin{aligned}
\texttt{unbox} &= \texttt{unbox}_{\texttt{[]}}\\
\texttt{unbox}_e\texttt{(box }v\texttt{)} &= v\\
\texttt{unbox}_e\texttt{(apply\_box }f\ v\texttt{)} &= \texttt{(unbox}_e\ f\texttt{)(unbox}_e\ v\texttt{)}\\
\texttt{unbox}_e\texttt{(bind\_apply }f\ v\texttt{)} &= \texttt{(unbox}_e\ f\texttt{)(unbox}_e\ v\texttt{)}\\
\texttt{letvar }f\ id\texttt{ as }s\texttt{ in }p &= \texttt{let }\ id\texttt{ = new\_var }f\ s\texttt{ in }p\\
\texttt{letvar }f\ id\texttt{ as }s\texttt{ for }ctxt\texttt{ in }p &=\\
\texttt{let }s', ctxt = \texttt{fresh }\ s\ ctxt\texttt{ in let }\ & id\texttt{ = new\_var }f\ s\texttt{ in }p\\
\texttt{name\_of }\ v &= \texttt{v.name}\\
\texttt{subst(unbox}_e\texttt{(bindvar }v\texttt{ in}f\texttt{))}a &= \texttt{unbox}_{(v,a)::e}f\\
\texttt{unbox}_e\texttt{(bindbox\_of }v\texttt{)} &= \texttt{try assoc }v\ e\texttt{ with Not\_found -> }v\texttt{.f }v\\
\texttt{unbox}_e\texttt{(\^{}}a_1,\ldots,a_n\texttt{\^{})} &= \texttt{(unbox}_e\ a_1,\ldots,\texttt{unbox}_e\ a_n\texttt{)}\\
\texttt{unbox}_e\texttt{[\^{}}a_1;\ldots;a_n\texttt{\^{}]} &= \texttt{[unbox}_e\ a_1;\ldots;\texttt{unbox}_e\ a_n\texttt{]}\\
\texttt{unbox}_e\texttt{[|\^{}}a_1;\ldots;a_n\texttt{\^{}|]} &= \texttt{[|unbox}_e\ a_1;\ldots;\texttt{unbox}_e\ a_n\texttt{|]}\\
\texttt{unbox}_e\texttt{(Cstr(\^{}}a_1,\ldots,a_n\texttt{\^{}))} &= \texttt{Cstr(unbox}_e\ a_1,\ldots,\texttt{unbox}_e\ a_n\texttt{)}\\
\texttt{letvar }f\ ids\texttt{(}n\texttt{) as }s\texttt{ in }p &= \texttt{let }\ ids\texttt{ = map (new\_var }f\texttt{) }s\texttt{ in }p\\
\texttt{letvar }f\ ids\texttt{(}n\texttt{) as }s\texttt{ for }ctxt\texttt{ in }p &=\\
\texttt{let }s', ctxt = \texttt{fold\_map fresh }\ s\ ctxt\texttt{ in let }\ & ids\texttt{ = map (new\_var }f)\ s\texttt{ in }p
\end{aligned}
$$

Figure 1: Equational semantics for bindlib

value can not have type `'a variable` and `'b variable` if `'a` ≠ `'b`. This is enforced because the type `'a variable` is abstract.

- Value of type `context` will be set of strings. We consider that we have a function `fresh : string -> context -> string * context` 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 `letvar` construct, when the `as` keyword is ommited, the name of the identifier is used as a `string` for the variable name (or as a `string array` with a constant value for multiple binding).

- 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
```