Representation of Binders

Using the Bindlib (OCaml) Library

Rodolphe Lepigre & Christophe Raffalli

LFMTP Workshop – 07/07/2018 – Oxford

Motivations: binders are (were?) a pain

We need to develop programming languages / proof assistants

This requires many technical but boring elements:

- Source code parsing (notations, unicode)
- Representation of binders (functions, quantifiers)
- Computation of dependencies, management of modules

There are many applications of binders:

- Functions, type abstraction, polymorphic types
- Quantifiers (possibly higher-order), predicates
- Pattern-matching, unification variables, metavariables

```
module DB = struct
  (* With De Bruijn indices. *)
  type term =
    | Var of string           (* Free variable.          *)
    | Idx of int              (* Bound variable index.   *)
    | Abs of term             (* Abstraction (function). *)
    | App of term * term      (* Function application.   *)
end

module HOAS = struct
  (* With higher-order abstract syntax. *)
  type term =
    | Var of string           (* Free variable.          *)
    | Abs of (term -> term)   (* Abstraction (function). *)
    | App of term * term      (* Function application.   *)
end
```

Bindlib was designed by Christophe Raffalli in the nineties

I contributed several improvements:

- New, well-documented implementation (almost) from scratch
- Bindlib without the old (`camlp4`) syntax extension
- Lighter free variable management, "`unbind`" function, ...

Implemented systems relying on Bindlib:

- Lambdapi (new version of the Dedukti logical framework)
- PML proof system, SubML language (with subtyping)
- Pure type systems (PTS) and combinatory reduction systems (CRS)

```
type term =
  | TVar of term Bindlib.var                   (* Free variable.             *)
  | LAbs of (term, term) Bindlib.binder (* Abstraction (function). *)
  | Appl of term * term                        (* Function application.   *)
  | MAbs of (stack,term) Bindlib.binder (* Save operation.            *)
  | Name of stack * term                       (* Restore operation.        *)

and stack =
  | Epsi                                       (* Empty stack.                *)
  | SVar of stack Bindlib.var                  (* Stack variable.            *)
  | Push of term * stack                       (* Term pushed on stack.   *)
```

```
val subst  : ('a,'b) Bindlib.binder -> 'a -> 'b
val unbind : ('a,'b) Bindlib.binder -> 'a var * 'b

let rec eval : term * stack -> term * stack = function
  | (Appl(t,u) , pi        ) -> eval (t                , Push(u,pi))
  | (LAbs(f)   , Push(t,pi)) -> eval (Bindlib.subst f t , pi        )
  | (MAbs(f)   , pi        ) -> eval (Bindlib.subst f pi, pi        )
  | (Name(pi,t), _         ) -> eval (t                , pi        )
  | whnf                     -> whnf

let rec to_string : term -> string = function
  | TVar(x)    -> Bindlib.name_of x
  | LAbs(f)    -> let (x,t) = Bindlib.unbind f in
                  Printf.sprintf "\\%s.%s" (Bindlib.name_of x) (to_string t)
  | Appl(t,u) -> Printf.sprintf "(%s) %s" (to_string t) (to_string u)
  | _          -> "<...>"
```

```
(* There is no generic function like the following. *)
val bind_var : 'a Bindlib.var -> 'b -> ('a,'b) Bindlib.binder

(* However, Bindlib provides the following function. *)
val bind_var : 'a Bindlib.var -> 'b Bindlib.box
               -> ('a,'b) Bindlib.binder Bindlib.box


(* A value of the type ['a Bindlib.box] represents:
     - an element of type ['a] under construction,
     - its free variables are available for binding. *)

(* The ['a Bindlib.box] type is an applicative functor. *)
val box       : 'a -> 'a Bindlib.box
val apply_box : ('a -> 'b) Bindlib.box -> 'a Bindlib.box -> 'b Bindlib.box
val box_var   : 'a Bindlib.var -> 'a Bindlib.box
```

```
let _TVar : term Bindlib.var -> term Bindlib.box =
  fun x -> Bindlib.box_var x

let _LAbs : (term, term) Bindlib.binder Bindlib.box -> term Bindlib.box =
  fun b -> Bindlib.box_apply (fun f -> LAbs(f)) b

let _Appl : term Bindlib.box -> term Bindlib.box -> term Bindlib.box =
  fun t u -> Bindlib.box_apply2 (fun t u -> Appl(t,u)) t u


let rec lift_term : term -> term Bindlib.box = function
  | TVar(x)     -> _TVar x
  | LAbs(b)     -> _LAbs (Bindlib.box_binder lift_term b)
  | Appl(t,u)   -> _Appl (lift_term t) (lift_term u)
  | _           -> failwith "Not implemented..."
```

Examples of terms

```
(* Fresh (free variables). *)
let x : term Bindlib.var = Bindlib.new_var (fun x -> TVar(x)) "x"
let y : term Bindlib.var = Bindlib.new_var (fun x -> TVar(x)) "y"

(* Usual terms. *)
let id : term Bindlib.box =
  _LAbs (Bindlib.bind_var x (_TVar x))

let fst : term Bindlib.box =
  _LAbs (Bindlib.bind_var x (_LAbs (Bindlib.bind_var y (_TVar x))))

let delta : term Bindlib.box =
  _LAbs (Bindlib.bind_var x (_Appl (_TVar x) (_TVar x)))

(* Unboxed term (fully constructed). *)
let omega : term = Bindlib.unbox (_Appl delta delta)
```

```
let rec snf : term -> term = function
  | Appl(t,u) ->
      begin
        let v = snf u in
        match snf t with
        | LAbs(b) -> snf (Bindlib.subst b v)
        | h       -> Appl(h,v)
      end
  | LAbs(b)   ->
      begin
        let (x,t) = Bindlib.unbind b in
        let v = snf t in
        Bindlib.unbox (_LAbs (Bindlib.bind_var x (lift_term v)))
      end
  | TVar(x)   -> TVar(x)
  | _         -> failwith "not a lambda-term"
```

```ocaml
type ('a,'b) binder =
  { b_name   : string          (* Name of the bound variable.               *)
  ; b_bind   : bool            (* Indicates whether the variable occurs.    *)
  ; b_rank   : int             (* Number of remaining free variables.       *)
  ; b_mkfree : 'a var -> 'a (* Injection of variables into domain.       *)
  ; b_value  : 'a -> 'b        (* Substitution function.                    *) }


let subst : ('a,'b) binder -> 'a -> 'b =
  fun b v -> b.b_value v

let unbind : ('a,'b) binder -> 'a var * 'b = fun b ->
    let x = new_var b.b_mkfree (binder_name b) in
    (x, subst b (b.b_mkfree x))
```

Internal representation: variables and box

```
type 'a closure = varpos -> Env.t -> 'a

type 'a box =
  | Box of 'a
  (* Element of type ['a] with no free variable. *)
  | Env of any_var list * int * 'a closure
  (* Element of type ['a] with free variables stored in an environment. *)

and 'a var =
  { var_key         : int          (* Unique identifier.                *)
  ; var_prefix      : string       (* Name as a free variable (prefix). *)
  ; var_suffix      : int          (* Integer suffix.                   *)
  ; var_mkfree      : 'a var -> 'a (* Function to build a term.         *)
  ; mutable var_box : 'a box       (* Bindbox containing the variable.  *) }

let box_var : 'a var -> 'a box = fun x -> x.var_box
```

Internal representation: variable creation

```
(* type any_var = Any : 'a var -> any [@@ unboxed] FIXME *)
type any_var = Obj.t var

let new_var_closure key = fun vp -> Env.get (IMap.find key vp).index
let new_var : ('a var -> 'a) -> string -> 'a var =
  fun var_mkfree name ->
    let var_key = fresh_key () in
    let (var_prefix, var_suffix) = split_name name in
 (* let rec x =
      { var_key; var_prefix; var_suffix; var_mkfree
      ; var_box = Env([Any x], 0, new_var_closure var_key) }
    in x *)
    let var_box = Env([], 0, fun _ -> assert false) in
    let x = {var_key; var_prefix; var_suffix; var_mkfree; var_box} in
    x.var_box <- Env([Obj.magic x], 0, new_var_closure var_key); x
```

```ocaml
module Env : sig
  type t
  val create : int -> t
  val set : t -> int -> 'a -> unit
  val get : int -> t -> 'a

  (* ... *)
end = struct
  (* ... *)

  (* Safe as soon as we write/read at a fixed type for each index. *)
  let set env i e = Array.set env.tab i (Obj.repr e)
  let get i env = Obj.obj (Array.get env.tab i)

  (* ... *)
end
```

Formal proof of correctness:

- Coq implementation of Bindlib (Bruno Barras)
- With axiomatized environment operations
- PML implementation of Bindlib? (Bootstrap)

Complexity analysis (Bruno Barras)

We need feedback from new users!

Thanks!

https://github.com/rlepigre/ocaml-bindlib