Representation of Binders

Using the Bindlib (OCaml) Library

Rodolphe Lepigre & Christophe Raffalli

Séminaire Gallium – 01/10/2018 – Paris

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

Standard techniques to deal with binders

```
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, a History

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)

Abstract syntax representation

```
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.    *)
```

Substitution and destructive traversal

```
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)
  | _         -> failwith "Not implemented..."
```

Thinking inside the box

```
(* 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
```

Smart constructors and lifting

```
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)
```

Working under binders

```
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"
```

Binders and related operations

```
type ('a, 'b) binder

(* Widely used operation. *)
val subst : ('a, 'b) binder -> 'a -> 'b

(* Access to some properties (not used very often) *)
val binder_name   : ('a, 'b) binder -> string
val binder_occur  : ('a, 'b) binder -> bool
val binder_closed : ('a, 'b) binder -> bool
val binder_rank   : ('a, 'b) binder -> int

(* Convenient (but definable) operations. *)
val unbind    : ('a, 'b) binder -> 'a var * 'b
val eq_binder : ('b -> 'b -> bool) -> ('a, 'b) binder
                                   -> ('a, 'b) binder -> bool
```

Internal representation: binders

```
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))

let eq_binder eq_body b1 b2 =
  let (x, t1) = unbind b1 in
  eq_body t1 (subst b2 (b2.b_mkfree x))
```

Variables and related operations

```
type 'a var

(* Widely used operation. *)
val new_var : ('a var -> 'a) -> string -> 'a var
val name_of : 'a var -> string
val eq_vars : 'a var -> 'a var -> bool

(* Other useful operation. *)
val hash_var : 'a var -> int
```

Boxes and related operations

```
type 'a box

(* Widely used operations. *)
val unbox     : 'a box -> 'a
val box       : 'a -> 'a box
val apply_box : ('a -> 'b) box -> 'a box -> 'b box
val box_var   : 'a var -> 'a box
val bind_var  : 'a var -> 'b box -> ('a,'b) binder box

(* Other useful operations. *)
val is_closed : 'a box -> bool
val occur     : 'a var -> 'b box -> bool

(* Convenient (but definable) operations. *)
val box_apply  : ('a -> 'b) -> 'a box -> 'b box
val box_apply2 : ('a -> 'b -> 'c) -> 'a box -> 'b box -> 'c box
val box_opt    : 'a box option -> 'a option box
```

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
```

Internal representation: unboxing

```
let unbox : 'a box -> 'a = function
  | Box(t)       -> t
  | Env(vs,nb,t) ->
      let nbvs = List.length vs in
      let env = Env.create ~next_free:nbvs (nbvs + nb) in
      let cur = ref 0 in
      let fn vp x =
        let i = !cur in incr cur;
        Env.set env i (x.var_mkfree x);
        IMap.add x.var_key {index=i; suffix=x.var_suffix} vp
      in
      t (List.fold_left fn IMap.empty vs) env
```

Internal representation: unboxing

```
let bind_var : 'a var -> 'b box -> ('a, 'b) binder box = fun x b ->
  match b with
  | Box(t)                                      ->
      (* No free variable, variable does not occur. *)
      Box(build_binder x 0 false (fun _ -> t))
  | Env([y],n,t) when y.var_key = x.var_key ->
      (* The variable to bind is the last one. *)
      let r = {index = 0; suffix = x.var_suffix} in
      let t = t (IMap.singleton x.var_key r) in
      let value arg =
        let env = Env.create ~next_free:1 (n+1) in
        Env.set env 0 arg; t env
      in
      Box(build_binder x 0 true value)
  | Env(vs,n,t)                                 ->
      failwith "Code removed"
```

The (Obj.)magic of Bindlib

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

  (* ... *)
end = struct
  type t = Obj.t array

  (* 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
```

Bindlib is (type) safe

Only one questionable part: representation of environments

Environment cells are written and accessed at the same type:

- A `Bindlib.binder` is always built using `Bindlib.bind_var`
- (The variable and the binder must have matching types)
- The invariant is maintained by `Bindlib.bind_var`
- (Environment indices only live in its definition)

Environments cannot contain floating point values:

- Elements of type `float Bindlib.var` can be created
- But there is no way they can be bound in non-trivial structures
- Note that `Bindlib.bind_var x (Bindlib.box_var x)` is fine
- (Only bound variables have a reserved space in environments)

Related and future work

Formal proof of correctness:

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

Complexity analysis (Bruno Barras)

Feedback is welcome, especially from new users!

Thanks!

https://github.com/rlepigre/ocaml-bindlib (git repository)

http://eptcs.web.cse.unsw.edu.au/paper.cgi?LFMTP2018.4 (paper)