Bindlib
A package for abstract syntax with binder.
version 3.2

Christophe Raffalli
Universit de Savoie

March 3, 2008

Abstract

`bindlib` is a library and a camlp4 syntax extension for the OCaml language. It proposes a set of tools to manage data structures with bound and free variables. It includes fast substitution and management of variables names including renaming.

Contents

1 Introduction.

Data structures with bound and free variables are not so rare in computer science. For instance, computer programs and mathematical formulae are data structures using bound and free variables and are needed to write compilers and computer algebra system.

Representing these variables, implementing the necessary primitives (substitution of a variables, renaming to avoid captures, etc.) is not so easy and often the simple implementation result in poor performance, especially when substitution is needed.

Bindlib aims at providing a library resulting in both simple and efficient code. It provides tools to deal with substitution and renaming.

The version 3.0 of bindlib is simpler to use than the previous ones because of its camlp4 syntax extension and completely rewritten documentation.

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

This library only works with ocaml version 3.09.x., 3.10.x and after

To install the library follow the following steps:

`cp Makefile-3.XX Makefile` copy Makefile-3.09 or Makefile-3.10 to Makefile depending upon your ocaml version

`make checkconfig` : check the guessed value of BINDIR and LIBDIR. If it does not suit you, edit the Makefile.

`make` : this compiles the library.

`make check` : optional, to test the library.

`make install` : to install everything.

To use the library, compile your files with one of the following commands:

`ocamlc -pp camlp4bo -c foo.ml` : to produce `foo.cmo` from `foo.ml`, if it uses bindlib.

`ocamlc -pp camlp4bop -c foo.ml` : to produce `foo.cmo` from `foo.ml`, if it uses both bindlib and stream pattern matching.

`ocamlc -pp "camlp4 pa_bindlib.cmo ..." -c foo.ml` : if you want to use bindlib together with your other favorite extension.

The same options work with `ocamlopt`.

Finally, link with `bindlib.cma` or `bindlib.cmxa`.

3 Basic types

Before considering bound and free variables in data structures, we should say what a data structure is. For this documentation, we will consider that a data structure is an ML value (like a list of integers, a tree, etc.) with no ML function inside.

Now, the main type constructor of the bindlib library is:

`('a,'b) binder`.

This is the type of a data structure of type `'b` with one bound variable of type `'a`.

For variables, bindlib provide a type

`'a variable`

for variable in data structure of type `'a`.

With these type constructors, we can define easily the type of lambda-terms:

```
type term =
  App of term * term         (* application of two terms *)
| Abs of (term, term) binder (* abstraction of a variable in a term *)
| FVar of term variable      (* free variable *)
```

4 How to use a data structure with variables ?

Now let us start by using data structure with bound and free variables. Consider the following function to print lambda-terms:

```
let fVar x = FVar(x)

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

The first line creates a function synonymous to the `FVar` constructor. Then, printing of application is straight forward.

For abstraction, we use one of our camlp4 extension: `match f with bind fVar x in t` destruct `f` which is an expression e with one bound variable v (remark: the programmer has no direct access to e and v, this is why we do not use a typewriter font for them). A new variable `x` is created by this construct, and `t` denotes e where v is replaced by `fVar x`. This explains the two last line of the program. Notice the function `name_of` to get the variable name.

Let us now give an equivalent to the camlp4 syntax extension we use. The line

```
match f with bind fVar x in t ->
```

is equivalent to the following three lines:

```
let name = binder_name f in
letvar fVar x as name 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.
- `letvar fVar x as name in` is yet another a camlp4 extension creating a new variable `x : term variable`.

  `fVar` means that `free_of x` evaluates to `fVar x` which evaluates itself to `FVar(x)`

  `as name` means that the name of the variable `x` will be `name`. This means that `name_of x` evaluates to `name`.

  This camlp4 syntax extension is equivalent to `let x = new_var fVar name in` where

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

  (we will see later the interest of the `letvar` camlp4 extension)
- `free_of : 'a variable -> 'a` has been explained above with `letvar`.
- `subst : ('a, 'b) binder -> 'a -> 'b` to substitute a value to the bound variable in `f`.

n

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 is 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 !

Let use give some examples:

```
let fVar x = FVar(x)

let idt = unbox (Abs(^ bind fVar x in x ^))
let delta = unbox (Abs(^ bind fVar x in App(^x,x^) ^))
```

Here are the basic idea in the above examples: we use `Abs(^ ... ^)` and `App(^ ... ^)` to construct value of type `term bindbox` instead of `term` directly. Then, the camlp4 syntax extension `bind fVar x in` allows to construct a value of type `(term, term) binder bindbox`. Finally, `unbox` finishes the work and produces a value of type `term` from a value of type `term bindbox`.

Here is a list with more detailed explanation of the common functions and camlp4 syntax extensions to construct data structure with bound and free variables:

- `unit : 'a -> 'a bindbox` : you use this function when you have an object of type `'a` that you want to extend.

  Warning: with `unit t`, you will not be able to bind free variables in `t`.

- `(^t^)` : a camlp4 syntax extension for `unit t` (t is parsed at the priority `LEVEL ":="`).

- `(^t1, ... , tn^)` : to construct tuples in the `bindbox` type. If `t1 : 'a1 bindbox`, ..., `tn : 'an bindbox`, then `(^t1, ..., tn^) : ('a1 * ... * 'an) bindbox`

- `[^t1; ... ; tn^]` to construct lists in the bindbox type. If `t1 : 'a bindbox`, ..., `tn : 'a bindbox`, then `[^t1; ...; tn^] : 'a list bindbox`

- `[|^t1; ... ;tn^|]` to construct arrays in the bindbox type. If `t1 : 'a bindbox`, ..., `tn : 'a bindbox`, then `[|^t1; ...; tn^|] : 'a array bindbox`

- `( ^:: ) : 'a bindbox -> 'a list bindbox -> 'a list  bindbox`: to construct list cells. Remark: this is a camlp4 extension to have the same priority as `::`.

- `Cstr(^t1, ... , tn^)` : to apply a variant constructor in the `bindbox` type. This also works for polymorphic variant constructors.

  Example: if `t1` and `t2` are of type `term bindbox`, then `App(^t1,t2^)` is also of type `term bindbox`.

  More generally, if you have a type constructor `Cstr` with the following typing rule:

  $$\frac{x_1 : t_1 \qquad \dots \qquad x_n : t_n}{\texttt{Cstr}(x_1, \dots, x_n) : u}$$

  Then, you also have

  $$\frac{x_1 : t_1\ \texttt{bindbox} \qquad \dots \qquad x_n : t_n\ \texttt{bindbox}}{\mathrm{Cstr}\texttt{(\^{}}x_1, \dots, x_n\texttt{\^{})} : u\ \texttt{bindbox}}$$

- `{^ l1 = t1; ...;  ln = tn ^}` : to construct a structure in the bindbox type.

  Example: if one defines the type `defi = { name : string;  value : term}` and if `t : term bindbox`, then `{^ name = (^ "foo" ^);  value = t ^} : defi bindbox`.

Now one also need the following functions and camlp4 extension to deal with variables:

- `bindbox_of : 'a variable -> 'a bindbox` : this is the correct (and unique) way to use a variable in order to be able to bind it.
- `bindvar x in t` : this is a camlp4 extension. If `x : 'a variable` and `t : 'b bindbox`, then `bindvar x in t : ('a, 'b) binder bindbox`.
- `bind f x in t` : this is a camlp4 extension (the variable `x` is bound in `t`) and it is a short cut for:

  ```
  letvar f x' in
  let x = bindbox_of x' in
  bindvar x' in t
  ```

- `unbox : 'a bindbox -> 'a` : this is the function to produce the final data-structure. In `unbox t`, all the variables that were not bound using the `bind` or `bindvar` camlp4 extension will become free variables by calling the function `f : 'b variable -> 'b` that you had to give when creating the variable with the `letvar` or `bind` camlp4 extension.

Here is another example, which performs the following transformation on λ-term (it marks all the application with a 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:

```
let mark t =
  let rec phi x = function
    FVar(y) -> bindbox_of y
  | App(u,v) -> App(^App(^x,phi x u^),phi x v^)
  | Abs(f) ->
      match f with bind fVar y in f' ->
        Abs(^ bindvar y in phi x f' ^)
  in
  unbox(Abs(^ bind fVar x in phi x t^))
```

This code is very similar to the mathematical definition. 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
      Abs 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 *)
let norm t = let rec fn t =
  match whnf t with
    Abs f ->
      match f with bind fVar x in u ->
      Abs(^ bindvar x in fn u ^)
  | t ->
      let rec unwind = function
          FVar(x) -> bindbox_of 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. Understanding this function is more a question of knowledge of λ-calculus than a problem of the library itself.

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 5 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.

A Module `Bindlib` : Bindlib library

A.1 Basic functions

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

this is the type of an expression of type `'b` with a bound variables of type `'a`

`type 'a variable`

`type 'a mvariable = 'a variable array`

type of variables and array of variables

`type context`

context, used to store names to rename free variables

`val empty_context : context`

`val subst : ('a, 'b) binder -> 'a -> 'b`

this is the subtitution function: it takes an expression with a bound variable of type `'a` and a value for this variable and replace all the occurrences of this variable by this value

`val binder_name : ('a, 'b) binder -> string` binder_name f returns the name of the variable bound in f

```
val name_of : 'a variable -> string
```

name_of v returns the name of the free variable v

```
val compare_variables : 'a variable -> 'a variable -> int
```

a safe comarison for variables

```
type +'a bindbox
```

type inhabited by data structures of type 'a with variables under construction

```
val unbox : 'a bindbox -> 'a
```

the function to call when the construction of an expression of type 'a is finished.

```
val bindbox_of : 'a variable -> 'a bindbox
```

the function to use a variable inside a data structure

```
val free_of : 'a variable -> 'a
```

the function to use a variable as free, identical to the composition of unbox and bindbox_of

```
val unit : 'a -> 'a bindbox
```

unit allows you to use an expression of type 'a in a larger expression constructed with free variables

```
val apply : ('a -> 'b) bindbox -> 'a bindbox -> 'b bindbox
```

this is the function that allows you to construct expressions by allowing the application of a function with free variables to an argument with free variables

```
val dummy_bindbox : 'a bindbox
```

It is sometimes usefull to have a dummy value, for instance to initialize arrays

If one use dummy_bindbox in a data structure, calling unbox will raise the exception Failure "Invalid use of dummy_bindbox"

```
val is_binder_closed : ('a, 'b) binder -> bool
```

check if the bound variable occurs or not

A.2 Multiple binders

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

this is the type of an expression of type 'b with several bound variables of type 'a

```
val mbinder_arity : ('a, 'b) mbinder -> int
```

mbinder_arity f returns the number of variables bound in f

```
val binder_arity : ('a, 'b) mbinder -> int
val mbinder_names : ('a, 'b) mbinder -> string array
```

mbinder_names f returns the names of the variables bound in f

```
val binder_names : ('a, 'b) mbinder -> string array
val msubst : ('a, 'b) mbinder -> 'a array -> 'b
```

this is the subtitution function: it takes an expression with several bound variables of type 'a and an array of values for these variables and replace all the occurrences by the given values

```
val is_mbinder_closed : ('a, 'b) mbinder -> bool
```

check if one of the bound variables occurs or not. There are no way to tell if a specific variable bound in a multiple binder occurs or not

A.3 Other functions

```
val is_closed : 'a bindbox -> bool
```

this function tells you if a 'a bindbox is closed. This means it has no free variables. This may be useful when optimizing a program

```
val bind_apply : ('a, 'b) binder bindbox ->
  'a bindbox -> 'b bindbox
val mbind_apply :
  ('a, 'b) mbinder bindbox ->
  'a array bindbox -> 'b bindbox
```

These functions are usefull when using "higher order variables". That is variables that represent itself binder and therefore that can be applied to arguments

```
val unit_apply : ('a -> 'b) -> 'a bindbox -> 'b bindbox
```

This function and the following ones can be written using unit and apply but are given because they are very often used. Moreover, some of them are optimised

unit_apply f a = apply (unit f) a

```
val unit_apply2 :
  ('a -> 'b -> 'c) ->
  'a bindbox -> 'b bindbox -> 'c bindbox
```

unit_apply2 f a b = apply (apply (unit f) a) b

```
val unit_apply3 :
  ('a -> 'b -> 'c -> 'd) ->
  'a bindbox ->
  'b bindbox -> 'c bindbox -> 'd bindbox
```

```
unit_apply3 f a b c = apply (apply (apply (unit f) a) b) c
```

```
val lift_pair : 'a bindbox -> 'b bindbox -> ('a * 'b) bindbox
```

```
lift_pair (x,y) = unit_apply2 (,) x y
```

```
val fixpoint :
  (('a, 'b) binder, ('a, 'b) binder) binder
  bindbox -> ('a, 'b) binder bindbox
```

Very advanced feature: binder fixpoint !

The following structures allow you to define function like `lift_list` or `lift_array` for your own types, if you can provide a `map` function for your types. These are given for polymorphic types with one or two parameters

```
module type Map =
  sig
    type 'a t
    val map : ('a -> 'b) -> 'a t -> 'b t
  end
module Lift :
functor (M : Map) ->   sig
    val f : 'a bindbox M.t -> 'a M.t bindbox
  end
module type Map2 =
  sig
    type ('a, 'b) t
    val map : ('a -> 'b) ->
      ('c -> 'd) -> ('a, 'c) t -> ('b, 'd) t
  end
module Lift2 :
functor (M : Map2) ->   sig
    val f : ('a bindbox, 'b bindbox) M.t -> ('a, 'b) M.t bindbox
  end
val lift_list : 'a bindbox list -> 'a list bindbox
```

```
lift_list =
  let module M = struct
    type 'a t = 'a list
    let map = List.map end
  in Lift(M).f
```

```
val lift_array : 'a bindbox array -> 'a array bindbox
```

```
lift_array =
  let module M = struct
    type 'a t = 'a array
    let map = Array.map end
  in Lift(M).f
```

A.4 Syntactic extension of the language for bindlib

We describe here the syntactic extension provided by `pa_bindlib.cmo`.

expr ::= ...
| `(^` *expr* `^)`
| `(^` *expr*`,` *expr* `,` *expr* `^)`
| `[^ ^]`
| `[^` *expr* `;` *expr* `^]`
| `[|^ ^|]`
| `[|^` *expr* `;` *expr* `^|]`
| *constr*`(^` *expr* `^)`
| *constr*`(^` *expr*`,` *expr* `,` *expr* `^)`
| `{^` *field* `=` *expr* {`;` *field* `=` *expr*} `^}`
| `{^` *expr* `with` *field* `=` *expr* {`;` *field* `=` *expr*} `^}`
| `letvar` *expr* *lowercase-ident*[`(` *expr* `)`] [`as` *expr*] [`for` *lowercase-ident*] `in` *expr*
| `bindvar` *lowercase-ident*[`()`] `in` *expr*
| `bind` *expr* *lowercase-ident*[`(` *expr* `)`] [`as` *expr*] [`for` *lowercase-ident*] `in` *expr*
| `match` *expr* `with` `bind` *expr* *lowercase-ident*[`(` *lowercase-ident* `)`]
    [`as` *expr*] [`for` *lowercase-ident*] `in` *lowercase-ident* `->`*expr*
| *expr* `^^` *expr*
| *expr* `^|^` *expr*

Remarks and comment:

- `(^e^)` is equivalent to `unit e`
- In `letvar` and `bind`, the first *expr* is parser at priority level "simple" and therefore it should be parenthesised if it is not an identifier.
- in `match ... bind`, the [`as` *expr*] option is not a pattern to hold the name of the bound variable, but a string you can give to rename the variable.

- `e ^^ e'` is equivalent to `bind_apply e e'` and it is left associative and parsed at the priority level of application. This is why it is a syntactic extension and not an infix operator.
- `e ^|^ e'` is equivalent to `mbind_apply e e'` and it is left associative and parsed at the priority level of application.

B Module `Nvbindlib` : Bindlib library

Author(s): Christophe Raffalli

This libraries provide functions and type to use binder (that is construct that binds a variable in a data structure)

```
type ('a, 'b) binder = 'a -> 'b
```

this is the type of an expression of type 'b with a bound variables of type 'a

```
type 'a variable
type 'a var = 'a variable
type 'a mvariable = 'a variable array
val subst : ('a, 'b) binder -> 'a -> 'b
```

this is the subtitution function: it takes an expression with a bound variable of type 'a and a value for this variable and replace all the occurrences of this variable by this value

```
type 'a bindbox
```

the type of an object of type 'a being constructed : this object may have free variables

```
val bindbox_of : 'a variable -> 'a bindbox
```

the function to call when the construction of an expression of type 'a is finished. This two functions are identical, and can also be used as a prefixoperator !!

```
val free_of : 'a variable -> 'a
val unbox : 'a bindbox -> 'a
val unlift : 'a bindbox -> 'a
val (!!) : 'a bindbox -> 'a
val unit : 'a -> 'a bindbox
```

unit allows you to use an expression of type 'a in a larger expression begin constructed with free variables

```
val lift : 'a -> 'a bindbox
val (!^) : 'a -> 'a bindbox
val bind :
```

```
  ('a variable -> 'a) ->
  ('a bindbox -> 'b bindbox) ->
  ('a, 'b) binder bindbox
```

this is THE function constructing binder. If takes an expression of type 'a bindbox → 'b bindbox in general written (fun x → expr) (we say that x is a free variable of expr). And it constructs the expression where x is bound. The first argument is a function build a free variables. The second argument is the name of the variable

```
val new_var : ('a variable -> 'a) -> 'a variable
val bind_var : 'a variable ->
  'b bindbox -> ('a, 'b) binder bindbox
val apply : ('a -> 'b) bindbox -> 'a bindbox -> 'b bindbox
```

this is THE function that allows you to construct expressions by allowing the application of a function with free variables to an argument with free variables

```
val dummy_bindbox : 'a bindbox
type ('a, 'b) mbinder
```

this is the type of an expression of type 'b with "n" bound variables of type 'a

```
val mbinder_arity : ('a, 'b) mbinder -> int
```

mbinder_arity f return the number of variables bound by

```
val msubst : ('a, 'b) mbinder -> 'a array -> 'b
```

this is the subtitution function: it takes an expression with a bound variable of type 'a and a value for this variable and replace all the occurrences of this variable by this value

this is THE function constructing mbinder. If takes an expression of type 'a bindbox array → 'b bindbox in general written (fun x → expr) (we say that x is vector of free variables of expr). And it constructs the expression where x's are bound. The first argument is a function to build a free variables. The second argument are the names of the variable

```
val mbind :
  ('a variable -> 'a) ->
  int ->
  ('a bindbox array -> 'b bindbox) ->
  ('a, 'b) mbinder bindbox
val new_mvar : ('a variable -> 'a) -> int -> 'a mvariable
val bind_mvar : 'a mvariable ->
  'b bindbox -> ('a, 'b) mbinder bindbox
val is_binder_closed : ('a, 'b) binder -> bool
```

check if the bound variable occurs or not

```
val is_mbinder_closed : ('a, 'b) mbinder -> bool
```

check if one of the bound variables occurs or not

```
val bind_apply : ('a, 'b) binder bindbox ->
  'a bindbox -> 'b bindbox
```

Used in some rare cases !

```
val mbind_apply :
  ('a, 'b) mbinder bindbox ->
  'a array bindbox -> 'b bindbox
val fixpoint :
  (('a, 'b) binder, ('a, 'b) binder) binder
  bindbox -> ('a, 'b) binder bindbox
val unit_apply : ('a -> 'b) -> 'a bindbox -> 'b bindbox
```

The following function can be written using unit and apply but are given because they are very usefull. Moreover, some of them are optimised

```
val unit_apply2 :
  ('a -> 'b -> 'c) ->
  'a bindbox -> 'b bindbox -> 'c bindbox
val unit_apply3 :
  ('a -> 'b -> 'c -> 'd) ->
  'a bindbox ->
  'b bindbox -> 'c bindbox -> 'd bindbox
val lift_pair : 'a bindbox -> 'b bindbox -> ('a * 'b) bindbox
val lift_list : 'a bindbox list -> 'a list bindbox
val (^::) : 'a bindbox ->
  'a list bindbox -> 'a list bindbox
val lift_array : 'a bindbox array -> 'a array bindbox
val is_closed : 'a bindbox -> bool
```

this function tells you if a 'a bindbox is closed. This may be useful when optimizing a program

the following structures allow you to define function like lift_pair or lift_list for your own types, if you can provide a "map" function for your types. These are given for polymorphic types with one or two arguments

```
module type Map =
  sig
     type 'a t
     val map : ('a -> 'b) -> 'a t -> 'b t
  end
module Lift :
functor (M : Map) ->   sig
```

```
    val f : 'a Nvbindlib.bindbox M.t -> 'a M.t Nvbindlib.bindbox
  end

module type Map2 =
  sig
    type ('a, 'b) t
    val map : ('a -> 'b) ->
      ('c -> 'd) -> ('a, 'c) t -> ('b, 'd) t
  end

module Lift2 :
functor (M : Map2) ->   sig
    val f :
      ('a Nvbindlib.bindbox, 'b Nvbindlib.bindbox) M.t ->
      ('a, 'b) M.t Nvbindlib.bindbox
  end

type environment
type varpos
type 'a env_term = varpos -> environment -> 'a
val special_apply :
  unit bindbox ->
  'a bindbox -> unit bindbox * 'a env_term
val special_start : unit bindbox
val special_end : unit bindbox -> 'a env_term -> 'a bindbox
```

B.1 Syntactic extension of the language for nvbindlib

The syntactic extension provided by `pa_nvbindlib.cmo` are the same as for bindlib, except that the [`as` *expr*] and [`for` *lowercase-ident*] options dealing with variables are not allowed.

C 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 value can not have type `'a variable` and `'b variable` if `'a` $\neq$ `'b`. This is enforced when 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
  ```

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

Figure 1: Equational semantics for bindlib