Abstract Representation of Binders in OCaml using the Bindlib Library

Rodolphe Lepigre
Inria, LSV, CNRS, Université Paris-Saclay
Cachan, France
rodolphe.lepigre@inria.fr

Christophe Raffalli
LAMA, CNRS, Université Savoie Mont Blanc
Chambéry, France
christophe.raffalli@univ-smb.fr

The Bindlib library for OCaml provides a set of tools for the manipulation of data structures with variable binding. It is very well suited for the representation of abstract syntax trees, and has already been used for the implementation of half a dozen languages and proof assistants (including a new version of the logical framework Dedukti). Bindlib is optimised for fast substitution, and it supports variable renaming. Since the representation of binders is based on higher-order abstract syntax, variable capture cannot arise during substitution. As a consequence, variable names are not updated at substitution time. They can however be explicitly recomputed to avoid "visual capture" (i.e., distinct variables with the same apparent name) when a data structure is displayed.

1 Introduction

The implementation of programming languages and/or theorem provers plays an important role in our community. It allows us to concretely illustrate the mathematical models that we consider, and also helps us discovering new intuitions through experimentations. In practice, implementing languages is more difficult than one could expect. It requires the combination of specific techniques, ranging from the parsing of source files to unification algorithms, through the representation of data structures with variable bindings. We here focus on the latter point and introduce the Bindlib library [16] for OCaml.

Variable binding is very common in computer science, as it is used for the representation of computer programs and mathematical formulas as abstract syntax trees. However, implementing the necessary primitives, including variable renaming and capture-avoiding substitution, is cumbersome and error-prone. Moreover, naive implementations generally result in poor performances, especially when many substitutions must be performed. The Bindlib library solves these problems thanks to an abstract representation of binders with an efficient substitution operation. It is based on a form of higher-order abstract syntax [12],[1] which eliminates the possibility of variable capture. Variable names are managed using a distinct mechanism, and bound variables are not effectively renamed when performing a substitution. As a consequence, displaying a data structure in which substitutions have occurred may introduce "visual capture", or distinct variables with the same apparent name. To avoid this unfortunate, but harmless situation, variable names must be explicitly updated.[2] This is achieved using a glorified "identity function", which injects the data structure into a specific Bindlib type constructor.

[1]The idea of higher-order abstract syntax is to represent binders as functions in the host language. As a consequence, the binding of a value of type `'a` in a value of type `'b` is represented as a function of type `'a -> 'b`.

[2]Bound variable names are changed in the minimal way, and they may still be overloaded as in the λ-term $\lambda x.\lambda x.x$ (which is α-equivalent to $\lambda x.\lambda y.y$). It is possible to enforce a "stronger" form of renaming like Barendregt's convention.

Submitted to:
LFMTP 2018

1.1 Main working principles

The Bindlib library provides type constructors `'a var` and `('a,'b) binder`, for representing free variables (of type `'a`) and binders (of a value of type `'a` in an element of type `'b`) respectively. A bound variable can be substituted using the `subst` function.

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

However, it is not possible to bind a free variable directly, as there is no built-in function of type `'a var -> 'b -> ('a,'b) binder`. This is not surprising because this function does not have any specific information on the structure of the elements of type `'b`. To bind variables in a data structure, it must first be injected in a specific type constructor `'a box`. Intuitively, an element of type `'a box` corresponds to a value of type `'a` under construction, and its free variables can be bound easily using `bind_var`.

```
val bind_var : 'a var -> 'b box -> ('a,'b) binder box
```

When an element of type `'a box` has been fully constructed, and all the desired variables have been bound, it can be "unboxed" to a value of type `'a` using the `unbox` function.

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

In the process, its free variables are set to remain free. Indeed, binding such variables would require lifting the value to the `'a box` type again, which is not very efficient as it requires a traversal of the data structure. Nonetheless, two representations of an element of type `'a` must often coexist and interact. The type `'a` itself is used for pattern-matching and for substituting binders, and the type `'a box` is used whenever variables need to be bound. The latter case arises not only at the construction of a term, but also when one needs to work under binders (e.g., to compute the strong normal form of a λ-term).

The Bindlib library aims at being very flexible, and it does not impose any restriction on the types of variables that can be bound, nor on the type of elements in which they are bound. As a consequence, it is the responsibility of the programmer to implement the lifting functions transforming values of type `'a` into values of type `'a box` for every type `'a` that may contain bound variables. This limited amount of boiler-plate code is generally straightforward, and mainly amounts to providing *smart constructors* for each constructor of the abstract syntax tree. In other words, every constructor of the type `'a` must be lifted to the `'a box` type using provided Bindlib functions. In an earlier version of Bindlib, this process was partly automated using an OCaml syntax extension. However, this approach proved to be rather confusing for new users, and the benefits did not really outweigh the costs.

1.2 Origins and applications of Bindlib

The development of Bindlib [16] was initiated by the second author in the nineties. An early version of the library was used to implement an efficient normaliser for the λ-calculus [5]. It was then made into a separate library, and used in the implementation of the first version of the PML language [6]. Starting from version 4, Bindlib was mostly rewritten by the first author, who proposed some simplifications an made a significant documentation effort. Further simplifications and improvements have been put into the version 5 of Bindlib [16], which is being released at the time of writing. Bindlib is now finally ready to be used by a larger community. The latest version can be easily installed with the Opam package manager, using the following command.

```
opam update && opam install bindlib
```

Bindlib has been and is used for the implementation of half a dozen programming languages and/or proof assistants, as well as for a large number of small prototypes, including an implementation of the pure type systems (PTS), and an implementation of the combinatory reduction systems (CRS). We give below a list of the most recent and relevant systems relying on Bindlib.

The SubML language. The SubML language [10] implements a rich extension of System F with Subtyping [17]. It features polymorphic, existential, inductive and coinductive types, which all require variable binding. Variable binding is also used for the standard λ-abstraction constructor, as well as a fixpoint for general recursion. The SubML language makes a particularly extensive use of binders as the system relies on choice operators (similar to Hilbert's Epsilon operator) in its syntax. Moreover, the language allows quantification over sizes for inductive types, which requires binding a syntactic representation of ordinals into types (as it is usually done for sized types).

The PML$_2$ proof system. The PML$_2$ proof system [11] implements the type system described in the PhD thesis of the first author [9]. This project certainly contains the most advanced uses of Bindlib so far, as it mixes many different types of binders. In particular, the language admits a higher-order type system with several base sorts (values, terms, stacks, propositions and ordinals), as well as an arrow sort. In particular, Bindlib binders are mixed with GADTs in order to implement multiple binders with non-homogeneous types (i.e., binding values of different types at once into a term).

A new version of Dedukti. Bindlib was most recently used by the first author to propose a new implementation of the logical framework Dedukti [1], called Lambdapi [8]. Although the abstract syntax of the language has a rather simple binding structure, with only λ-abstractions and dependent product types, the system makes a singular use of Bindlib for representing rewriting rules. A rewriting rule $l \hookrightarrow r$ is formed of a *left-hand side* (LHS) l, which corresponds to a pattern, and a *right-hand side* (RHS) r, in which the free variables of l are bound. For example, the term "`Plus` 12 (`Succ` 29)" matches the pattern of the rewriting rule "`Plus` n (`Succ` m) $\hookrightarrow$ `Succ` (`Plus` n m)", associating the values 12 and 29 to variables n and m, thus resulting in the term "`Succ` (`Plus` 12 29)" after the application of the rewriting rule. In the implementation, the RHS is effectively represented as a binder ranging over the free variables of the LHS. Applying a rule thus simply amounts to substituting this binder with the values gathered for the variables during pattern-matching.

1.3 Existing approaches to data structures with bound variables

Naive approach. There exist several different approaches to the representation of data structures with variable bindings. A first possibility is to use the naive approach, which is very close to our pen and paper intuitions. In this presentation, variables are represented with names, and they are bound by simply referencing these names. For example, the pure λ-terms can be encoded as follows.[3]

```
(** Naive representation of pure λ-terms. *)
type term =
  | Var of string        (** Bound or free variable. *)
  | Abs of string * term (** Abstraction. *)
  | App of term * term   (** Application. *)
```

[3]Note that bound and free variables are represented uniformly (using a single constructor), despite the fact that these objects have very different status. This will not be the case in other representations.

Note that terms can be parsed and injected in the above data type in a straightforward way. For example, the term $\lambda x.\lambda y.x\ y$ is represented as `Abs("x", Abs("y", App(Var("x"), Var("y"))))`. Although this representation is convenient for the construction of terms, it is not very well suited for implementation. Indeed, the capture-avoiding substitution operation cannot be implemented efficiently, and it is also relatively hard to get right. Overall, the main interest of the naive approach is purely pedagogical, since it illustrates the usual notions of renaming and capture-avoiding substitution at a low level.

De Bruijn indices The most widely used technique for implementing binders is certainly De Bruijn indices [2], in which bound variables are replaced with natural numbers giving the "distance" between the variable and the linked binder. The corresponding representation of pure λ-terms can be expressed as follows.[4]

```
(** De Bruijn representation of pure λ-terms. *)
type term =
  | Var of string      (** Free variable.          *)
  | Ind of int         (** Index (bound variable). *)
  | Abs of term        (** Abstraction.            *)
  | App of term * term (** Application.            *)
```

Using De Bruijn indices, the term $\lambda x.\lambda y.x\ y$ is represented as `Abs(Abs(App(Ind(2),Ind(1))))`.[5] Note that α-equivalent terms have a unique representation using De Bruijn indices, but that bound variable names are lost in the process (if they are not managed using a specific mechanism). Substitution using De Bruijn representation is well-defined, but hard to get right in practice, especially if several kind of objects can be bound. Indeed, the index to substitute (initially `1`) increases when moving under binders, and some index shifting is also necessary in the substituted term when it contains indices that are bound outside the scope of the substitution.

Higher-order abstract syntax and its variations Another alternative is the use of higher-order abstract syntax (or HOAS) [12], in which a binder is represented using a function of the host language. As it relies on the binders of the meta-language, the correctness of this approach is immediate. A HOAS representation of pure λ-terms is given below.

```
(** Higher-order abstract syntax representation of pure λ-terms. *)
type term =
  | Var of string          (** Free variable. *)
  | Abs of (term -> term)  (** Abstraction.   *)
  | App of term * term     (** Application.   *)
```

The term $\lambda x.\lambda y.x\ y$ is here encoded as `Abs(fun x -> Abs(fun y -> App(x,y)))`. As with De Bruijn indices, the name of bound variables are not stored. As a consequence, they must be handled using a separate mechanism. A variation of this technique is used in the internals of Bindlib, but it is not visible to the user.

An important remark about the use of HOAS is that the domain of binders appears negatively (i.e., to the left of an odd number of arrows). As a consequence, the definition of the type `term` above falls

[4]The use of De Bruijn indices is sometimes referred to as *locally nameless*, referring to the fact that free variables are named. This idea was already present in the work of De Bruijn, as remarked by Charguéraud [3].

[5]De Bruijn indices most often start at `0` for the immediate binder, but we here stick to `1` which was used in the original presentation [2]. Both approaches are isomorphic, but using `0` allows some optimisations.

Syntax of terms and types:

$$t,u ::= x \mid \lambda x:A.t \mid t\ u \mid \Lambda X.t \mid t\ A$$
$$A,B ::= X \mid A \Rightarrow B \mid \forall X.A$$

Operational semantics:

$$\frac{}{(\lambda x:A.t)\ u \longrightarrow t[x:=u]} \qquad \frac{}{(\Lambda X.t)\ A \longrightarrow t[X:=A]} \qquad \frac{t_1 \longrightarrow t_2}{t_1\ u \longrightarrow t_2\ u}$$

$$\frac{u_1 \longrightarrow u_2}{t\ u_1 \longrightarrow t\ u_2} \qquad \frac{t_1 \longrightarrow t_2}{\lambda x:A.t_1 \longrightarrow \lambda x:A.t_2} \qquad \frac{t_1 \longrightarrow t_2}{\Lambda X.t_1 \longrightarrow \Lambda X.t_2}$$

Typing rules:

$$\frac{}{\Gamma, x:A \vdash x:A} \qquad \frac{\Gamma, x:A \vdash t:B}{\Gamma \vdash \lambda x:A.t : A \Rightarrow B} \qquad \frac{\Gamma \vdash t:A \Rightarrow B \qquad \Gamma \vdash u:A}{\Gamma \vdash t\ u:B}$$

$$\frac{\Gamma \vdash t:A \qquad X \notin \Gamma}{\Gamma \vdash \Lambda X.t : \forall X.A} \qquad \frac{\Gamma \vdash t:\forall X.A}{\Gamma \vdash t\ B : A[X:=B]}$$

Figure 1: Syntax, operational semantics and typing rules for Church-style System F.

into the limitations of languages such as Coq or Agda, where positivity is enforced on inductive data types (this is required for their soundness). To solve this issue, variations of HOAS such as *parametric higher-order abstract syntax* [4] or *nested abstract syntax* [7] have been designed. Although they are more well-behaved than general HOAS (in the sense that the corresponding types only contain terms of the represented abstract syntax), such techniques are not very flexible.

1.4 Related works and similar tools

There are only very few available tools or libraries for the representation of binders in the OCaml language. The only one that seems to be a serious alternative is François Pottier's Cαml (or alpha-Caml) [14, 15], which follows a completely different approach. Indeed, Cαml is not an OCaml library, but a meta-programming tool that generates OCaml code from a "binding specification". Yet another difference with Bindlib is that Cαml relies on De Bruijn indices, not a form of HOAS.

Another approach is that of the FreshML family of languages [13, 18, 19], which extend OCaml with specific binding facilities. There seem to be some similarities between these tools and the Bindlib approach, most notably in the idea of generating fresh names (or rather, variable) to substitute binders. One advantage of FreshML over Bindlib is that the user does not have to provide any code (with Bindlib, a lifting function must be written by the user, as we will shortly see). However, this requires a modification of the host language, while Bindlib is a simple library.

2 Application to Church-style System F

We will now consider the implementation of the Church-style (explicitly typed) version of System F. This language is more interesting than the usual, pure λ-calculus example because it requires the binding of types in types (polymorphism), the binding of terms in terms (λ-abstractions), and the binding of types in terms (type abstractions). The syntax of the language, its operational semantics and its typing rules are recalled in Figure 1.

2.1 Abstract syntax tree

The abstract syntax tree of our language can be encoded as follow, using the Bindlib data types. Free variables are represented using the `'a var` type, and binders using the `('a,'b) binder` type.

```
(** Representation of a type. *)
type ty =
  | TyVar of ty var                (** Free type variable.   *)
  | TyArr of ty * ty               (** Arrow type.           *)
  | TyAll of (ty,ty) binder        (** Universal quantifier. *)

(** Representation of a term. *)
type te =
  | TeVar of te var                (** Free lambda-variable. *)
  | TeAbs of ty * (te,te) binder (** Lambda-abstraction.   *)
  | TeApp of te * te               (** Application.          *)
  | TeLam of (ty,te) binder        (** Type abstraction.     *)
  | TeSpe of te * ty               (** Type specialisation.  *)
```

The definition of these types closely follows the BNF grammar given at the top of Figure 1 page 5, and similar definitions can be made for richer languages.

In the following sections, we will demonstrate the construction of terms and types using this representation, and the implementation of various functions. In particular, we will see how it is possible to work under binders, using user-defined lifting functions. Note that such functions are not needed when we do not want to work under binders, or when binders only have to be substituted. For example, it is possible to define the following head-normalisation function on terms.

```
(** Head-normalisation function. *)
let rec hnf : te -> te =
  function
  | TeApp(t,u) ->
      begin
        let v = hnf u in
        match hnf t with
        | TeAbs(_,b) -> hnf (subst b v)
        | h          -> TeApp(h,v)
      end
  | TeSpe(t,a) ->
      begin
        match hnf t with
        | TeLam(b) -> hnf (subst b a)
        | h        -> TeSpe(h,a)
      end
  | t          -> t
```

On the other hand, we will only be able to implement a strong normalisation function after the lifting operations (related to the construction of binders) have been defined.

Of course, it is possible to implement variations of the head normalisation function by changing the evaluation strategy, which is here right-to-left call-by-value. It can also be implemented in the form of a stack machine (e.g., a Krivine abstract machine).

2.2 Smart constructors and lifting functions

As discussed in the introduction, it is the responsibility of the programmer to define the lifting functions for every type that may contain free variables. Here, this is the case for both the `te` and `ty` types. We will thus need to give two lifting functions which types will be `te -> te box` and `ty -> ty box` respectively. To do so, it is a good practice to first define a set of smart constructors, that will also be useful for building terms and types.

Smart constructors are generally straightforward to define, as this only requires lifting the corresponding constructors to boxed types. To this aim, Bindlib provides functions such as the following.[6]

```
val box_apply  : ('a -> 'b) -> 'a box -> 'b box
val box_apply2 : ('a -> 'b -> 'c) -> 'a box -> 'b box -> 'c box
```

Using these functions, we can define the following smart constructors for our abstract syntax.

```
let _TyArr : ty box -> ty box -> ty box =
  box_apply2 (fun a b -> TyArr(a,b))

let _TyAll : (ty,ty) binder box -> ty box =
  box_apply (fun f -> TyAll(f))

let _TeAbs : ty box -> (te,te) binder box -> te box =
  box_apply2 (fun a f -> TeAbs(a,f))

let _TeApp : te box -> te box -> te box =
  box_apply2 (fun t u -> TeApp(t,u))

let _TeLam : (ty,te) binder box -> te box =
  box_apply (fun f -> TeLam(f))

let _TeSpe : te box -> ty box -> te box =
  box_apply2 (fun t a -> TeSpe(t,a))
```

In the above, we did not provide any smart constructors for free variables. They are handled using a specific function `box_var`, which is used to make variables available for binding.

```
val box_var : 'a var -> 'a box
```

As every variable can potentially be bound in a boxed type, the variable constructors of the abstract syntax do not make sense at this level. We can however complete our set of smart constructors by using synonyms of `box_var`. This is obviously not necessary, but it makes the use of smart constructors more uniform during the construction of terms.

```
let _TeVar : te var -> te box = box_var

let _TyVar : ty var -> ty box = box_var
```

To be able to construct terms, we still need to introduce two Bindlib functions. The first one is called `new_var`, and allows the creation of a new free variable. The second one is the `bind_var` function, which

[6]Various similar functions can be used to lift usual type constructor to their boxed forms. In fact, they can all be implemented using two primitives, thanks to the fact that the `'a box` has an applicative functor structure (this will be explained later).

is used to effectively bind a variable in a boxed value. The obtain boxed binder can then be fed to smart constructors like `_TyAll`.

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

val bind_var : 'a var -> 'b box -> ('a,'b) binder box
```

Note that the `new_var` function requires a `string`, corresponding to a preferred name for the bound variable, and a function for injecting a `'a var` into the `'a` type.

Before defining the lifting functions for our abstract syntax representation we will demonstrate the construction of types and terms. We will construct the type $\forall X.\forall Y.(X \Rightarrow Y) \Rightarrow X \Rightarrow Y$, and a corresponding application combinator $\Lambda X.\Lambda Y.\lambda f : X \Rightarrow Y.\lambda a : X.f\ a$. They can be defined as follows.

```
(* Creation of variables. *)
let _X = new_var (fun x -> TyVar(x)) "X"
let _Y = new_var (fun x -> TyVar(x)) "Y"
let f  = new_var (fun x -> TeVar(x)) "f"
let a  = new_var (fun x -> TeVar(x)) "a"

(* Representation of the type X ⇒ Y. *)
let _X_arr_Y : ty box = _TyArr (_TyVar _X) (_TyVar _Y)

(* Representation of the type ∀X.∀Y.(X ⇒ Y) ⇒ X ⇒ Y. *)
let appl_ty : ty box =
  _TyAll (bind_var _X (_TyAll (bind_var _Y (_TyArr _X_arr_Y _X_arr_Y))))

(* Representation of the term ΛX.ΛY.λf:X⇒Y.λa:X.f a. *)
let appl_te : te box =
  _TeLam (bind_var _X (_TeLam (bind_var _Y (
    _TeAbs _X_arr_Y (bind_var f (_TeAbs (_TyVar _X) (bind_var a (
      _TeApp (_TeVar f) (_TeVar a)))))))))
```

Note that the construction of these terms can then be completed by calling the `unbox` function, which type is `'a box -> 'a`. It would then be possible to pattern-match on these elements, as we did in the implementation of the `hnf` function.

The lifting functions can then be defined in a straightforward way using the smart constructors. Note that the `box_binder` function is used to propagates the lifting operation under binders, provided a suitable lifting function for their codomain. To do so, `box_binder` simply substitutes the given binder using a fresh variable, applies the lifting function, and reconstructs the binder using the `bind_var` function.

```
let rec lift_ty : ty -> ty box = fun a ->
  match a with
  | TyVar(x)   -> _TyVar x
  | TyArr(a,b) -> _TyArr (lift_ty a) (lift_ty b)
  | TyAll(f)   -> _TyAll (box_binder lift_ty f)

let rec lift_te : te -> te box = fun t ->
  match t with
  | TeVar(x)   -> _TeVar x
  | TeAbs(a,f) -> _TeAbs (lift_ty a) (box_binder lift_te f)
  | TeApp(t,u) -> _TeApp (lift_te t) (lift_te u)
  | TeLam(f)   -> _TeLam (box_binder lift_te f)
  | TeSpe(t,a) -> _TeSpe (lift_te t) (lift_ty a)
```

2.3 Basic functions: normalisation, printing and equality

At the beginning of the current section, we were able to define a head normalisation function `hnf`, since it did not require working under binders. We will now consider several examples of functions which need to do so, starting with a strong normalisation.

```
let rec nf : te -> te = fun t ->
  match t with
  | TeVar(_)   -> t
  | TeAbs(a,f) ->
      let (x,t) = unbind f in
      TeAbs(a, unbox (bind_var x (lift_te (nf t))))
  | TeApp(t,u) ->
      let u = nf u in
      begin
        match nf t with
        | TeAbs(_,f) -> nf (subst f u)
        | t          -> TeApp(t,u)
      end
  | TeLam(f)   ->
      let (x,t) = unbind f in
      TeLam(unbox (bind_var x (lift_te (nf t))))
  | TeSpe(t,a) ->
      begin
        match nf t with
        | TeLam(f) -> nf (subst f a)
        | t        -> TeSpe(t,a)
      end
```

Note that this function is similar to `hnf` in most cases, except when the term is a binder (i.e., when it is a `TeAbs` or a `TeLam` constructor). To handle these, the corresponding binder is substituted using a fresh variable using the `unbind` function, which returns a couple of the fresh variable and the term obtained after substitution. It is then possible to call the normalisation function on this term, and then reconstruct the binder. To do so, the term must be lifted using our `lift_te` function so that the variable `x` is available for binding. The binder is then reconstructed using the usual `bind_var` function, and it is then immediately unboxed.

Let us now consider the case of printing functions for our abstract syntax. Although they need to work under binders, there is no boxing involved since no binder must be reconstructed. We only provide the printing function for the `ty` type, as printing terms can be achieved in a very similar way.[7]

```
let rec print_ty : out_channel -> ty -> unit = fun oc a ->
  match a with
  | TyVar(x)   -> output_string oc (name_of x)
  | TyArr(a,b) -> Printf.fprintf oc "(%a) ⇒ (%a)" print_ty a print_ty b
  | TyAll(f)   -> let (x,a) = unbind f in
                  Printf.fprintf oc "∀%s.%a" (name_of x) print_ty a
```

As for `nf`, the `unbind` function is used to decompose binders into pairs of a fresh variable and a term. However, the printing function relies on `name_of` to obtain the name of free variable. The name of the variable `x` returned by `unbind` is built according to its bound counterpart.

[7]Note that we always add parentheses in functions types to avoid ambiguities. It would of course be possible to limit the number of such parentheses, but this is not important here.

To conclude this section, we provide an equality function for the types of our abstract syntax. As for printing, there is no boxing involved because equality can be tested in a “destructive” way. The function below relies on the `eq_vars` for variable comparison (which amounts to comparing their unique keys), and the function `eq_binder` is used for binders.

```
let rec eq_ty : ty -> ty -> bool = fun a b -> a == b ||
  match (a, b) with
  | (TyVar(x1)   , TyVar(x2)   ) -> eq_vars x1 x2
  | (TyArr(a1,b1), TyArr(a2,b2)) -> eq_ty a1 a2 && eq_ty b1 b2
  | (TyAll(f1)   , TyAll(f2)   ) -> eq_binder eq_ty f1 f2
  | (_           , _           ) -> false
```

Note that `eq_binder` simply substitutes the two given binders with the same, fresh variable. The obtained bodies can then be compared using the given function. In particular, the comparison is automatically performed modulo α-equivalence in this way.

2.4 Type-checking and type inference

To complete our example, we will now consider the implementation of the type system of Figure 1 page 5. We will here represent a typing context as an association list, mapping free term variables to types. Note that the corresponding lookup function needs to use `eq_vars` to compare the keys.

```
type context = (te var * ty) list

let find_ctxt : te var -> context -> ty option = fun x ctx ->
  try Some(snd (List.find (fun (y,_) -> eq_vars x y) ctx))
  with Not_found -> None
```

The (mutually defined) type-inference and type-checking functions can then be implemented as follows, using a standard bidirectional type-checking approach. Note that Bindlib-specific functions only need to be used in the cases related to binders. For instance, inferring the type of a λ-abstraction requires a call to the `unbind` function, and the returned variable is then used to extend the context. In some cases, it is also necessary to establish a binder. This is the case when inferring the type of a type abstraction, where we start by decomposing the binder with `unbind`, infer the type of the body, and finally bind the variable returned by `unbind` in the type to produce a polymorphic type.

```
let rec infer : context -> te -> ty = fun ctx t ->
  match t with
  | TeVar(x)   ->
      begin
        match find_ctxt x ctx with
        | None    -> failwith "[infer] variable not in context..."
        | Some(a) -> a
      end
  | TeAbs(a,f) ->
      let (x,t) = unbind f in
      let b = infer ((x,a)::ctx) t in
      TyArr(a,b)
  | TeApp(t,u) ->
      begin
        match infer ctx t with
        | TyArr(a,b) -> check ctx u a; b
        | _           -> failwith "[infer] expected arrow type..."
      end
  | TeLam(f)    ->
      let (x,t) = unbind f in
      let a = infer ctx t in
      TyAll(unbox (bind_var x (lift_ty a)))
  | TeSpe(t,b) ->
      begin
        match infer ctx t with
        | TyAll(f) -> subst f b
        | _         -> failwith "[infer] expected quantifier..."
      end
```

The type-checking function relies on similar techniques, and it uses the previously defined `eq_ty` function. In the case of a more expressive language, like Martin-Löf dependent type theory, this equality function would be replaced by a convertibility test that would involve evaluation (and hence substitution). Examples of such situations can also be found in the implementation of Lambdapi [8], where conversion plays an even greater role.

```
and check : context -> te -> ty -> unit = fun ctx t a ->
  match (t, a) with
  | (TeVar(x)   , b          ) ->
      let a =
        match find_ctxt x ctx with
        | None     -> failwith "[check] variable not in context..."
        | Some(a) -> a
      in
      if not (eq_ty a b) then
        failwith "[check] type mismatch... (var)"
  | (TeAbs(c,f), TyArr(a,b)) ->
      if not (eq_ty c a) then
        failwith "[check] type mismatch... (abs)";
      let (x,t) = unbind f in
      check ((x,a)::ctx) t b
  | (TeApp(t,u), b          ) ->
      let a = infer ctx u in
      check ctx t (TyArr(a,b))
  | (TeLam(f1) , TyAll(f2) ) ->
      let (_,t,a) = unbind2 f1 f2 in
      check ctx t a
  | (TeSpe(t,b), a          ) ->
      begin
        match infer ctx t with
        | TyAll(f) ->
            let c = subst f b in
            if not (eq_ty c a) then
              failwith "[check] type mismatch... (spe)"
        | _         -> failwith "[infer] expected quantifier..."
      end
  | (_          , _          ) ->
      failwith "[check] not typable..."
```

3 Overview of the implementation of Bindlib

We will now consider the core aspects of the implementation of the Bindlib library, which can be downloaded at https://github.com/rlepigre/ocaml-bindlib. Note that the full library contains less than 1000 lines of generously documented code.

3.1 Main data structures

At the core of Bindlib, the two most important data types are `'a var` (representing a free variable of type `'a`) and `'a box` (representing an element of type `'a` under construction). Their (mutual) definitions are given and discussed below.

```
type 'a var =
  { var_key          : int          (** Unique identifier.              *)
  ; var_prefix       : string       (** Prefix of the variable name.    *)
  ; var_suffix       : int          (** Suffix of the variable name.    *)
  ; var_mkfree       : 'a var -> 'a (** Free variable constructor in ['a]. *)
  ; mutable var_box : 'a box        (** Bindbox containing the variable.   *) }
```

A free variable is uniquely identified by an integer, stored in the `var_key` field. It is not only used for referencing variables, but also for comparing them. The name of a variable is split into a fixed prefix, and an integer suffix which can be modified to avoid name clashes when a variable is in the position of being bound. Note that a variable also carries a function of type `'a var -> 'a`, which is called for injecting the (free) variable into the corresponding type. Conversely, a variable also stores its boxed counterpart, so that it only needs to be computed once.

```
and  '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. *)
```

A boxed element of type `'a` is represented as `Box(e)` in the case where it does not contain any free variable. Otherwise, it is represented as `Env(vs,n,cl)`, which contains the list `vs` of all the variables that it effectively contains[8] (sorted by key), an integer `n` giving the number of variables that have effectively been bound, and the value represented as a closure `cl`, which type is the following.

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

A closure expects two arguments. The former is a map, associating variable keys to positions in the environment. The latter is the environment itself, which is implemented as an array of heterogeneous values. The efficiency of the substitution operation of Bindlib (`subst` function) has to do with the fact that closures are constructed in two steps. The first step only involves the first argument of the closure, and consists in computing indices in the environment. The second step only consists in accessing or manipulating the environment, using the precomputed indices. In particular, the `varpos` map is only used once for each variable, even if the variable appears many times.

[8]In the current implementation, `any_var` is defined as `Obj.t var` and elements of `'a var` are coerced to `any_var` using `Obj.magic`. We could use an existential type instead (as suggested by Bruno Barras), but this would require an extra block of memory due to limitations on unboxing (see https://caml.inria.fr/mantis/view.php?id=7774). Another advantage of using an existential type would be that the `var_box` field would not need to be mutable anymore.

3.2 Variable manipulation and binding

As mentioned in earlier sections, fresh variable are created using the `new_var` function, which has the type `('a var -> 'a) -> string -> 'a var`. Its first argument is used as the value of the injection function in the `var_mkfree` field of the `'a var` type, and the `string` name given as second argument is decomposed into a prefix and an integer suffix for the `var_prefix` and `var_suffix` fields. The variable key, stored in the `var_key` field, is set using a new unique identifier. Most interestingly, the `var_box` field of the created variable `x` is first initialised with a dummy value, and then set as follows.

```
let cl vp = Env.get (IMap.find var_key vp).index in
x.var_box <- Env([to_any x], 0, cl)
```

The only variable in the box is `x` itself, and `0` variables were bound. The closure computes the index corresponding to `x` in the environment, and then relies on `Env.get` (of type `int -> Env.t -> 'a`) to access the value in the environment in constant time.

We will now go into the construction of binders, which are created using the `bind_var` function, of type `'a var -> 'b box -> ('a,'b) binder box`. The type `('a,'b) binder` itself is defined as follows, and contains a name for the bound variable and a function of type `'a -> 'b` (in the spirit of higher-order abstract syntax), along with several less important informations on the binder.

```
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.                 *) }
```

Note that the `b_mkfree` field is initialised with the `mk_free` field of the variable being bound, so that the binder can be substituted with a fresh variable when calling functions such as `unbind`.

The implementation of the `bind_var` function cannot be explained in full here, because it contains four different cases. In particular, the computation of `bind_var x t` depends on whether `x` occurs in `t`, and whether if is the last free variable of `t`. If `x` does not occur in `t`, then it may be that `t` is closed (i.e., of the form `Box(v)`) or that `x` is not in the list of its free variables (i.e., `t` is of the form `Env(vs,n,cl)`, with `x` not in `vs`). In both cases, the `b_value` field of the binder is set to be a constant function. In the case where the variable occurs, the then term `t` must be of the form `Env(vs,n,cl)` with `x` appearing in `vs`. In this case, a position is reserved for the variable in the environment, and associated to `x.var_key` in the `varpos` map.

3.3 Boxing as an applicative functor

The `'a box` type constructor of Bindlib is an applicative functor which unit is the `box` function, and which application is the `apply_box` function. The former corresponds to an application of the `Box` constructor, and can be used to inject any value of type `'a` in the `'a box` type, assuming that it does not contain any Bindlib variable. The latter applies a boxed function to a boxed argument, obtaining a boxed result. Note that these two primitives, along with `box_var` and `bind_var` are the only four functions that are necessary for the manipulation of boxed values, and the construction of binders. In particular, `box` and `apply_box` can be combined to obtain convenient boxing functions for usual data types. For instance, we can define the function `box_apply` and `box_apply2` (that we used previously) as follows.

```
let box_apply : ('a -> 'b) -> 'a box -> 'b box =
  fun f a -> apply_box (box f) a

let box_apply2 : ('a -> 'b -> 'c) -> 'a box -> 'b box -> 'c box =
  fun f ta tb -> apply_box (box_apply f ta) tb
```

We can also define boxing functions for the `'a option` type or the `'a list` type as follows.

```
let box_opt : 'a box option -> 'a option box =
  function
  | None    -> box None
  | Some(e) -> box_apply (fun e -> Some(e)) e

let box_list : 'a box list -> 'a list box =
  fun l -> List.fold_right (box_apply2 (fun x l -> x::l)) l (box [])
```

In the implementation, some of the provided boxing functions are not defined from `box` and `box_apply` for performance reasons. This is the case for the `box_list` function, which is implemented using a more general and optimised (imperative) functor mechanism, which only requires a `map` function.

3.4 On the use of magic

The implementation of Bindlib relies on the `Obj` module[9] of OCaml to store values of different types in a single array, intuitively corresponding to the environments of a closure. However, the implementation of Bindlib satisfies the invariant that the value at every given index is always read or written at a fixed type. As mentioned earlier, the `Obj.magic` function is also used to transform variables of type `'a var` into value of type `any_var` (defined as `Obj.t var`), which are only used in a type-safe way.

Although the lists of variables with heterogeneous types used in the `'a box` type could be encoded with an existential type, this is not the case for the environments. Ensuring (more) type safety in the handling of environments would require the user to provide a runtime representation of all the type of variables that can be bound. We chose not to pursue this directions as this would make the use of Bindlib cumbersome, and the only benefit would be to have an exception raised instead of a segmentation fault, which could only happen in case of a Bindlib bug.

4 Conclusion and future work

The Bindlib library has been around for more than twenty years, but it was only used in a very restricted community centered around the second author. The main reason for that was that the library was not very well documented, and hence not easily accessible to potential users. This problem has now been fixed, and the library is now extensively documented.

In a recent work, Bruno Barras has initiated the implementation of a Coq version of Bindlib. Although it is not publicly available yet, this work opens the way to the validation of the Bindlib model with a formal specification and formal proofs. Due to the specificities of Bindlib, this requires an axiomatic presentation, notably for the representation and the manipulation of environments.

[9]This module gives access to the low-level representation of data structures, and allows us to bypass type-checking.

References

[1] Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard (2016): *Expressing theories in the $\lambda\Pi$-calculus modulo theory and in the Dedukti system.* In: 22nd International Conference on Types for Proofs and Programs, TYPES 2016, Novi SAd, Serbia. Available at https://hal-mines-paristech.archives-ouvertes.fr/hal-01441751.

[2] N.G de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem.* Indagationes Mathematicae (Proceedings) 75(5), pp. 381 – 392. Available at http://www.sciencedirect.com/science/article/pii/1385725872900340.

[3] Arthur Charguéraud (2012): *The Locally Nameless Representation.* J. Autom. Reasoning 49(3), pp. 363–408. Available at https://doi.org/10.1007/s10817-011-9225-2.

[4] Adam Chlipala (2008): *Parametric higher-order abstract syntax for mechanized semantics.* In: ICFP, ACM, pp. 143–156. Available at http://doi.acm.org/10.1145/1411204.1411226.

[5] Christophe Raffalli (1995): *A Normaliser for Pure and Typed λ-Calculus.* Available at https://lama.univ-savoie.fr/~raffalli/normaliser.html. First version implemented in Caml Light.

[6] Christophe Raffalli (2007): *PML: a new proof assistant.* Available at http://lama.univ-savoie.fr/~raffalli/pml. Prototype implementation.

[7] André Hirschowitz & Marco Maggesi (2012): *Nested Abstract Syntax in Coq.* J. Autom. Reasoning 49(3), pp. 409–426. Available at https://doi.org/10.1007/s10817-010-9207-9.

[8] Rodolphe Lepigre (2017): *Lambdapi: a new implementation of Dedukti.* Available at https://github.com/rlepigre/lambdapi.

[9] Rodolphe Lepigre (2017): *Semantics and Implementation of an Extension of ML for Proving Programs. (Sémantique et Implantation d'une Extension de ML pour la Preuve de Programmes).* Ph.D. thesis, Grenoble Alpes University, France.

[10] Rodolphe Lepigre & Christophe Raffalli (2016): *Implementation of the SubML language.* Available at https://github.com/rlepigre/subml.

[11] Rodolphe Lepigre & Christophe Raffalli (2017): *Implementation of the PML_2 proof system.* Available at https://github.com/rlepigre/pml.

[12] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax.* In: PLDI, ACM, pp. 199–208.

[13] Andrew M. Pitts & Mark R. Shinwell (2008): *Generative Unbinding of Names.* Logical Methods in Computer Science 4(1). Available at https://doi.org/10.2168/LMCS-4(1:4)2008.

[14] François Pottier (2005): *Cαml.* Available at http://gallium.inria.fr/~fpottier/alphaCaml/.

[15] François Pottier (2006): *An overview of Cαml.* In: ACM Workshop on ML, Electronic Notes in Theoretical Computer Science 148, pp. 27–52. Available at http://gallium.inria.fr/~fpottier/publis/fpottier-alphacaml.pdf.

[16] Rodolphe Lepigre and Christophe Raffalli (2015): *The Bindlib OCaml Library Version 5.* Available at https://github.com/rlepigre/ocaml-bindlib. May be installed using the Opam package manager.

[17] Rodolphe Lepigre and Christophe Raffalli (2017-2018): *Practical Subtyping for Curry-Style Languages.* Available at http://lepigre.fr/files/docs/lepigre2017_subml.pdf. Under revision for publication in the TOPLAS journal.

[18] Mark R. Shinwell (2006): *Fresh O'Caml: Nominal Abstract Syntax for the Masses.* Electr. Notes Theor. Comput. Sci. 148(2), pp. 53–77. Available at https://doi.org/10.1016/j.entcs.2005.11.040.

[19] Mark R. Shinwell, Andrew M. Pitts & Murdoch James Gabbay (2003): *FreshML: programming with binders made simple.* SIGPLAN Notices 38(9), pp. 263–274.