N1 - Tutorial on Bindlib

February 1, 2019

The structure of this document is as follows. We first introduce the syntax of the language we work with $\lambda^{\mathbb{B},\rightarrow}$. Then we provide a brief tutorial on using the Bindlib library.

1 Syntax

The concrete syntax of $\lambda^{\mathbb{B},\rightarrow}$ is:

$$\begin{array}{rcl} M, N, P, Q & ::= & x \\ & | & \mathit{true} \\ & | & \mathit{false} \\ & | & \mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q \\ & | & \lambda x.M \\ & | & M\ N \end{array}$$

2 Naive (First-Order) Encoding of AST

The type representing the naive (first-order) representation of ASTs would be:

```
type term =
  | CstTrue
  | CstFalse
  | Var of string
  | App of term * term
  | Abs of string*term
  | ITE of term*term*term
```

As mentioned above, operations for performing renaming, substitution, generating fresh names, computing free variables and checking for α-equivalence must all be implemented from scratch and are not easy to get right. Bindlib provides support for all these operations.

3 Encoding of AST using Bindlib

We begin this tutorial by describing two type constructors, among others discussed later, that are defined in Bindlib:

- `'a var`: The type of variables of type `'a`. For example, if `term` is the type of our terms, then `term var` is the type of a variable over terms.

- `('a,'b) binder`: The type of a binder from elements of type `'a` to elements of type `'b`. For example, the type of the `Abs` constructor (for representing lambda abstractions) will be declared to be `(term,term) binder`. Indeed, in a term such as $\lambda x.M$, the variable x is a term variable and M is a term.

Below is the type declaration for `term` for encoding the AST of $\lambda^{\mathbb{B},\rightarrow}$ that uses the above mentioned Bindlib types. It will be the one we will work with for the rest of this assignment.

```
type term =
  | CstTrue
  | CstFalse
  | Var of term var            (* note use of var type *)
  | App of term * term
  | Abs of (term,term) binder (* note use of binder type *)
  | ITE of term*term*term
```

Before actually building expressions of type `term`, let us first illustrate how to traverse them. In order to inspect the argument of a `Var` or of an `Abs` we will use two operations defined in the Bindlib library:

- `val name_of : 'a var -> string`. The expression `name_of x` returns a printable name for variable `x`. This name will of course have to be provided when the variable is constructed, as we will see later.

- `val unbind : ('a, 'b)binder -> 'a var * 'b`. The expression `unbind e` substitutes the binder `e` using a fresh variable. The variable and the result of the substitution are returned.

The following code computes the list (possibly with duplicates) of the names of the free variables of a term. It makes use of a helper function `remove` (whose code is not supplied) that simply removes all copies of a value from a list of values:

```
let rec fv : term -> string list = fun t ->
  match t with
  | CstTrue | CstFalse -> []
  | Var(x) -> [name_of x]    (* note use of name_of *)
  | Abs(f) ->
      let (x,t) = unbind f in  (* note use of unbind *)
      remove (name_of x) (fv t)
  | App(t,u) ->
      (fv t) @ (fv u)
  | ITE(tc,tthen,telse) ->
      (fv tc) @ (fv tthen) @ (fv telse)
```

Here is another example:

```
let rec string_of_term : term -> string = function
  | CstTrue -> "True"
  | CstFalse -> "False"
  | Var(x) -> name_of x  (* note use of name_of *)
  | Abs(b) ->
     let (x,a) = unbind b  (* note use of unbind *)
     in "(lam "^(name_of x) ^"."^ string_of_term a ^")"
  | App(t,u) ->
     "("^string_of_term t ^"  "^string_of_term u^")"
  | ITE(t,u,v) ->
     "if "^string_of_term t ^" then "^string_of_term u ^" else "^
         ↪ string_of_term v
```

There are many techniques to address binders: de Bruijn indices, HOAS (higher-order abstract syntax), nominal sets, etc. Bindlib uses a form of HOAS.

3.1 Building Terms

Up until now we haven't constructed any terms and hence have not being able to test the functions `fv` and `string_of_term`. We will do so now. Constructing terms without binders is straightforward, so we first focus on these; later we'll show how to construct terms involving binders.

3.1.1 Terms without Binders

A simple example of a term without binders is `App(CstTrue,CstFalse)`. This is a valid expression of type `term`. If we want to include variables in our terms too then we need a way of constructing expressions of type `term var`. Bindlib provides the following operation for this purpose:

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

The first argument allows to inject values of type `'a var` into `'a`; its role will be explained soon. The second argument of `new_var` is a name that we assign the variable. Here is the term $(x\ x)$ represented as an expression of type `term`:

```
let t1 =
     let var_x = new_var (fun x -> Var(x)) "x"
     in App(Var(var_x),Var(var_x))
```

Here is an example of how to compute its free variables or turn it into a string:

```
utop # fv t1;;
- : string list = ["x"; "x"]
utop # string_of_term t1;;
- : string = "(x  x)"
```

In summary, we have seen how to construct values of type `term` which do *not* involve abstractions, that is, which do not involve the constructor `Abs`.

3.1.2 Terms with Binders

In order to include abstractions we need to know how to construct expressions of type `('a,'b) binder`. If we think of the type `term` we might expect a function:

```
val bind_var : term var -> term -> (term,term)binder
```

to construct such binders for terms. However, consider `bind_var x t` for a moment. In order to construct a binder and support operations that allow to rename variables and substitute them Bindlib needs to add extra structure to `t`. For this purpose, Bindlib requires its own, internal encoding of expressions of type `term`. The type `term box` represents exactly that: an encoding of an expression of type `term` with some "extra" book-keeping information. Therefore, the operation offered by Bindlib is:

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

As may be seen from the type, every time we need to create a binder in our running example of lambda calculus expressions, rather than supply an expression of type `term` as second argument, it will have to be **lifted** to `term box`. Bindlib supplies a number of such lifting operations which are described below (eg. `box_var`, `box_apply`, etc.). For example, to lift `CstTrue` we use write `box CstTrue`. Since we'll use this again later we'll introduce a function that does this for us.

```
let _CstTrue : term box =
  box CstTrue
let _CstFalse : term box =
  box CstFalse
```

In order to lift a variable of type `'a var` we use the `box_var` operation:

```
let _Var : term var -> term box =
  box_var
```

The remaining operations for lifting an expression from `term` to `term box` are:

```
let _Abs : (term,term) binder box -> term box =
  box_apply (fun f -> Abs(f))
let _App : term box -> term box -> term box =
  box_apply2 (fun t u -> App(t,u))
let _ITE : term box -> term box -> term box -> term box =
  box_apply3 (fun t u v -> ITE(t,u,v))
```

Here is a summary of the built-in operations for lifting that we have used:

```
val box_var : 'a var -> 'a box
(* box_var x builds a 'a box from the 'a var x. *)

val box : 'a -> 'a box
(* box e injects the value e into the 'a box type, assuming that it is
closed. *)

val box_apply : ('a -> 'b) -> 'a box -> 'b box
(*box_apply f ba applies the function f to a boxed argument ba. *)

val box_apply2 : ('a -> 'b -> 'c) -> 'a box -> 'b box -> 'c box
(* box_apply2 f ba bb applies the function f to two boxed arguments ba
and bb. *)

val box_apply3 : ('a -> 'b -> 'c -> 'd) -> 'a box -> 'b box -> 'c box -> 'd
    ↪ box
```

We summarize through an example. Suppose we want to construct the Church numeral 2 (we'll call it `t2`) and then compute its set of free variables via `fv`:

$$\lambda f.\lambda x.f(fx).$$

Notice that the term requires using `Abs` since there are abstractions in it.

```
let var_x : term var = new_var (fun x -> Var(x)) "x"
let var_f : term var = new_var (fun x -> Var(x)) "f"

let _t2 : term box =
  _Abs (bind_var var_f
    (_Abs (bind_var var_x
         (_App (_Var var_f) (_App (_Var var_f) (_Var var_x))))))

let t2 : term = Bindlib.unbox _t2
```

Above, we use a leading underscore for the names of expressions of type `term box`. This is just for notational clarity. Below we compute the free variables of `t2` and also convert it to a string. Notice the use of `Bindlib.unbox` for finally injecting it back to the type `term`; it has the type:

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

It is this operation that makes use of the second argument of `new_var`: it uses it to inject back (free) variables into `'a`. We conclude by computing the free variables of `t2`.

```
utop # fv t2;;
- : string list = []
# string_of_term t2;;
- : string = "(lam f.(lam x.(f  (f  x))))"
```

Exercise. Construct the term $\lambda f.\lambda x.f\,(x\,x)$ and call it `t3`. Then execute `string_of_term` on it.

3.2 An Example: Strong Evaluation

This section develops another example that uses the operations of Bindlib mentioned above. It concerns *strong evaluation*. Evaluation in functional programming languages never reduces under an abstraction. Indeed, if an abstraction is ever reached during the process of evaluation, then it stops and reports back the abstraction itself as the result. However, in many situations evaluating under abstractions makes sense. Some examples are when implementing partial evaluation techniques, when implementing proof assistants and when implementing type-checkers for dependent types.

We next define a strong evaluation judgement $M \Downarrow \mathcal{V}$. It should be read: "M (strongly) evaluates to the value $\mathcal{V}$". A value is either a neutral term or an abstraction:

$$\begin{array}{lrcl} \text{(values)} & \mathcal{V} & ::= & \mathcal{N} \mid \lambda x.\mathcal{V} \\ \text{(neutral terms)} & \mathcal{N} & ::= & x \mid \mathit{true} \mid \mathit{false} \mid \mathit{if}\ \mathcal{V}\ \mathit{then}\ \mathcal{V}\ \mathit{else}\ \mathcal{V} \mid \mathcal{N}\ \mathcal{V} \end{array}$$

where the condition $\mathcal{V}$ in ITE is not *true* or *false*. Next we present the rules defining the evaluation judgement:

$$\frac{}{x \Downarrow x}\ (\text{SE-Var})$$

$$\frac{M \Downarrow \lambda x : \sigma.P \quad P\{x := N\} \Downarrow \mathcal{V}}{M\,N \Downarrow \mathcal{V}}\ (\text{SE-App})$$

$$\frac{M \Downarrow \mathcal{N} \quad N \Downarrow \mathcal{V}}{M\,N \Downarrow \mathcal{N}\,\mathcal{V}}\ (\text{SE-App2})$$

$$\frac{M \Downarrow \mathcal{V}}{\lambda x.M \Downarrow \lambda x.\mathcal{V}}\ (\text{SE-Lam})$$

$$\frac{M \Downarrow \mathit{true} \quad P \Downarrow \mathcal{V}}{\mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q \Downarrow \mathcal{V}}\ (\text{SE-IfT}) \qquad \frac{M \Downarrow \mathit{false} \quad Q \Downarrow \mathcal{V}}{\mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q \Downarrow \mathcal{V}}\ (\text{SE-IfF})$$

$$\frac{M \Downarrow \mathcal{V}_1 \quad \mathcal{V}_1 \neq \mathit{true}, \mathit{false} \quad P \Downarrow \mathcal{V}_2 \quad Q \Downarrow \mathcal{V}_3}{\mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q \Downarrow \mathit{if}\ \mathcal{V}_1\ \mathit{then}\ \mathcal{V}_2\ \mathit{else}\ \mathcal{V}_3}\ (\text{SE-IfF})$$

Note that strong evaluation may require renaming. Renaming requires creating fresh (i.e. globally unused variables) variables. Fresh variable creation requires maintaining some notion of state. For example, evaluate the term $(\lambda z.z\,z)\ (\lambda x.\lambda y.x\,y)$ by constructing a derivation of

$$(\lambda z.z\,z)\ (\lambda x.\lambda y.x\,y) \Downarrow \mathcal{V}$$

for an appropriate $\mathcal{V}$.

Here is an implementation of strong evaluation using our type `term` that relies on the Bindlib library:

```
let rec nf : term -> term = fun t ->
  match t with
  | CstTrue | CstFalse | Var(_)   -> t
  | Abs(f) ->
      let (x,t) = unbind f in
      Abs(unbox (bind_var x (lift_term (nf t))))  (* new binder
      constructed after evaluating under it *)
  | App(t,u) ->
      begin
        match nf t with
        | Abs(f) -> nf (subst f u) (* note use of subst *)
        | v      -> App(v, nf u)
      end
  | ITE(tc,tthen,telse) ->
    match nf tc with
    | CstTrue -> nf tthen
    | CstFalse -> nf telse
    | v -> ITE(v,nf tthen, nf telse)
```

Some comments:

- First notice the use of `subst f u`, in the `App` clause, to perform substitution. This is handled entirely by Bindlib for us.

- The clause for `Abs` is interesting. According to rule (SE-Lam), we must evaluate under binders. This requires "unbinding" the argument `f` in `Abs(f)` using `unbind f`. Note how this allows `nf` to continue working under the binder: `nf t`. Note also how, the binder is rebuilt from `nf t` and variable `x`. Since `bind_var` requires its second argument to be `term box` but `nf t` has type `term`, we have to lift `nf t` to an expression of type `term box`. This is achieved with `lift_term` which is defined as follows:

```
let rec lift_term : term -> term box = fun t ->
  match t with
  | CstTrue   -> _CstTrue
  | CstFalse   -> _CstFalse
  | Var(x)   -> _Var x
  | Abs(f) -> _Abs (box_binder lift_term f)
  | App(t,u) -> _App (lift_term t) (lift_term u)
  | ITE(t,u,v) -> _ITE (lift_term t) (lift_term u) (lift_term v)
```

Let's run our evaluator on an example, namely computing 2^2:

```
utop # string_of_term (nf @@ App(t2,t2));;
- : string = "(lam x.(lam x0.(x  (x  (x  (x  x0))))))"
```

Notice that α-conversion took place during reduction since there was no variable names `x0` in the original term.