Projet 2 : Fouine

Lucas Escot, Romain Liautaud

May 16, 2017

Types

Notre implémentation de `Fouine` supporte tous les types suivants :

- **`unit`**, **`int`**, **`bool`**, **`string`**, **`char`**
- `'a ref`
- `'a list`, `'a array`, `('a1 * ... * 'an)`
- `exn`
- `'a -> 'b`

```
>>> [|true; false|];;
- : bool array = [|true; false|]

>>> ref (fun x -> x);;
- : ('_a -> '_a) ref = { contents = <fun> }

>>> E 3;;
- : exn = E (3)

>>> "ok", "cool", 3;;
- : string * string * int = ("ok", "cool", 3)
```

Environnement et Opérateurs

Il n'y a aucune distinction entre les opérateurs binaires, unaires et les fonctions `fouine`.

$$\texttt{a OP b} \longrightarrow \texttt{Call (Call (OP, a), b)}$$

```
>>> ref;;
- : '_a -> '_a ref = <fun>

>>> (:=);;
- : '_a ref -> '_a -> unit = <fun>

>>> (!);;
- : '_a ref -> '_a = <fun>

>>> aMake;;
- : int -> int array = <fun>
```

Types somme

```
>>> type ('a, 'b) pair = Left of 'a | Right of 'b;;
>>> type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree;;

>>> let l = Leaf;;
val l : 'a tree = Leaf
>>> Node (true, Leaf, Node (false, Leaf, Leaf));;
- : bool tree = Node (true, Leaf, Node (false, Leaf, Leaf))

>>> type exn = E of int;;
>>> type 'a list = [] | (::) of 'a * 'a list;;

>>> [1; 2; 3];;
- : int list = (::) (1, (::) (2, (::) (3, [])))
```

Pattern matching

(Presque) toutes les affectations font intervenir les patterns.

```
>>> let x, (y, _), h :: t = 1, (true, "ok"), [1; 2];;
val x : int = 1
val y : bool = true
val h : int = 1
val t : int list = (::) (2, [])

>>> let rec length l =
  match l with
  | [] -> 0
  | _ :: t -> 1 + length t;;
val length : 'a list -> int = <fun>

>>> let rec map f l =
  match l with
  | [] -> []
  | x :: t -> f x :: map f t;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Remarques

- La suppression de la distinction opérateurs/fonctions simplifie l'AST.

```
type t =
  | Var  of identifier
  | Const of constant
  | Tuple of t list
  | Array of t list
  | Constructor of string * t list

  | Let of pattern * t * t
  | LetRec of identifier * t * t

  | IfThenElse of t * t * t
  | Fun of pattern * t
  | Call of t * t
  | TryWith of t * pattern * t
  | MatchWith of t * (pattern * t) list
  | Raise of t
  | Seq of t * t
  | ArraySet of t * t * t
  | ArrayRead of t * t
```

Remarques

- On peut faire plein de choses marrantes.

```
>>> let (+) = (-);;
val + : int -> int -> int = <builtin>
>>> 2 + 2;;
- : int = 0

>>> let plusTwo = (+) 2;;
val plusTwo : int -> int = <builtin>
>>> plusTwo 4;;
- : int = -2

>>> let (-->) x y = (x, y);;
val --> : 'a -> 'b -> 'a * 'b = <fun>
>>> 1 --> 2;;
- : int * int = (1, 2)
```

Remarques

- Mais ça rend les transformations beaucoup plus compliquées.

```ocaml
let rec rem t v =
  match t, v with
  | TArrow (_, ty), CMetaClosure f ->
      CMetaClosure (function
        | CTuple [x; CTuple [CMetaClosure k; _]] ->
          k (rem ty (f x))
        | CTuple [x; CTuple [CClosure (p, e, env); _]] ->
          let env' = match_pattern env p (rem ty (f x)) in
          eval_expr env' e
        | _ -> raise InterpretationError
      )
  | _, x -> x in

Env.mapi
  (fun name v -> rem (List.assoc name !Infer.env) v)
  Base.base
```