Lucas Escot et Romain Liautaud

Compte-rendu de PROJ2

Vous avez entre les mains le rendu 4 du groupe de Lucas Escot et Romain Liautaud. Ce rendu est une version remaniée du fichier `README.md`, aussi vous ne serez sûrement pas dépaysés. Dans le texte qui suit, nous essayons de présenter au mieux les choix que nous avons effectués. Une citation forcée en passant, pour que ça apparaisse dans la biblio [2].

1 Inférence de type.

Un système d'inférence de type a été rajouté au cours du rendu 3. Il est très fortement inspiré du système de types introduit par Didier Rémy[1] pour le langage OCaml, fonctionnant sur le principe des "niveaux" de code. Nous avons porté une attention particulière à rendre ce système de type "sound" en pensant à ne pas généraliser les types de conteneurs mutables.

```
>>> let a = ref [];;
val a : ('_b list) ref = { contents = [] }
>>> 1 :: !a;;
- : int list = [1]
>>> a;;
- : (int list) ref = { contents = [] }
```

```
>>> let a = ref (fun x -> x);;
val a : ('_b -> '_b) ref = { contents = <fun> }
>>> a := fun x -> x + 1;;
- : unit = ()
>>> a;;
- : (int -> int) ref = { contents = <fun> }
```

A toute expression `fouine` est ainsi associé un type fouine : nous supportons `int`, `bool`, `string`, `char`, `'a list`, `'a array`, `'a ref`, `'a -> 'b`, `exn`, `unit` et `'a1 * ...* 'an`.

2 Types somme.

En plus de ces types proposés par défaut, il est maintenant possible de définir ses propres types sommes avec la syntaxe usuelle Caml.

```
type 'a tree =
        | Leaf
        | Node of 'a * 'a tree * 'a tree;;

type ('a, 'b) pair = Left of 'a | Right of 'b;;
```

Notons qu'il existe une distinction entre `Machin of int * int` et `Machin of (int * int)`. Cette différence est prise en compte dans `fouine` et détaillée dans le `README.md`.

Il existe d'ailleurs dans `fouine` des constructeurs par défaut qui, à l'instar de OCaml, permettent de définir les listes et les exceptions. Les types `'a list` et `exn` sont ainsi définis en `fouine` comme :

```
type 'a list = [] | (::) of 'a * 'a list;;

type exn = E of int;;
```

3 Pattern matching.

Pour être fidèle à la puissance des `let` de OCaml, nous avons dès le début voulu implémenter le **pattern matching**. Ainsi, que ce soit au sein d'une expression `let` ou comme argument d'une fonction, les patterns sont omniprésents dans `fouine`, et permette une grande expressivité. Ils ont récemment été étendus pour permettre le matching sur les constructeurs, et plus spécifiquement le matching sur les listes plus esthétique.

```
let l = [1; 2; 3];;
let x :: t = l;;
let a, (b, c, _) = 1, (2, true, "false");;
let E x = E 3;;
```

Après cet ajout et celui des constructeurs, l'absence du `match . with` dans `fouine` s'est fait cruellement ressentir, aussi nous sommes nous pressés de l'ajouter.

```
let rec length l =
  match l with
  | [] -> 0
  | x :: t -> 1 + length t
;;
```

4 Fonctions built-in.

Le choix le plus radical que nous avons effectué (peut-être à tord) est la suppression complète des opérateurs binaires, unaires et des fonctions de base (`prInt`, `ref`, `aMake`) de notre AST, pour les traiter comme des fonctions standards. Cette décision à amené à une simplification extrême de notre type `Ast.t`, et la possibilité de manipuler les opérateurs comme des fonctions standards. On peut donc notamment currifier les opérateurs par défaut, les réassigner à d'autres variables, ou encore définir nos propres opérateurs.

```
>>> let plusTwo = (+) 2 in plusTwo 3;;
- : int = 5
>>> let f = (+) in f 1 2;;
- : int = 3
>>> let (+++) x y = x - y in 5 +++ 4;;
- : int = 1
```

On a donc choisi de munir nos différents interpréteurs d'un environnement par défaut contenant toutes les fonctions et opérateurs de base, sous forme d'une valeur `fouine` `CMetaClosure` qui encapsule du code OCaml (le metalangage). Ces fonctions sont également ajoutées à l'environnement de type, pour pouvoir faire correctement l'inférence.

Si ce changement est très pratique à l'usage, il a été la source de beaucoup de frustration lors du travail sur les transformations d'AST vers AST (passage par continuation et suppression des références). En effet, dans les deux cas, il nous était alors impossible d'une part d'isoler les appels à `raise`, `ref`, `:=` ou `!` dans l'AST, puisque ceux-ci sont traités exactement comme des appels de fonctions classiques; et d'autre part de traiter les cas du style `[[ e1 + e2 ]]`. Notre solution, dans chacun des cas, est présentée plus en détail ci-dessous.

5 Transformations.

5.1 CPS et suppression des exceptions.

Pour résoudre le problème d'isoler les appels à `raise` dans l'AST, nous nous sommes résolus à ne plus le traiter comme une fonction mais bien comme un mot-clé spécial, avec son token `RAISE` et son noeud d'AST `Raise of t`.

Le reste de l'implémentation est somme toute assez classique, si ce n'est le problème des fonctions récursives. Une astuce a toutefois été mise au point pour permettre la définition de fonctions récursives après transformation, en ne faisant pas disparaître le noeud `LetIn` de l'AST. D'aucun diront que ce n'est pas du lambda-calcul pur mais bon, eh, que voulez-vous.

5.2 Suppression des références.

Pour résoudre le problème d'isoler les appels à `ref`, `!=` et `!`, par contre, nous avons décidé d'utiliser une approche un peu différente. Au lieu de faire des cas particuliers pour chacun d'entre eux dans l'AST, nous incluons dans notre transformation des nouvelles définitions de ces trois fonctions, qui font appel aux primitives `read`, `modify` et `allocate` de l'énoncé.

Concrètement, l'énoncé suggérait de transformer `ref 2` en :

```
[[ref 2]] = fun s ->
  let (v,s1) = (fun s -> (2, s)) s in
  let (l,s2) = allocate v s1 in (l,s2)
```

Mais nous le transformons plutôt en :

```
let (!) = fun _r -> fun _s ->
    (((read) (_s)) (_r), _s) in

let (:=) = fun _r -> fun _s ->
        (fun _v -> fun _s ->
            ((), (((modify) (_s)) (_r)) (_v)), _s) in

let (ref) = fun _v -> fun _s ->
    ((allocate) (_s)) (_v) in

let (_v, _) = (fun _s ->
    let (_v2, _s2) = (fun _s -> (2, _s)) (_s) in
    let (_v1, _s1) = (fun _s -> (ref, _s)) (_s2) in
    ((_v1) (_v2)) (_s1)) ((empty) ()) in
_v;;
```

Nous avons ensuite rencontré un second problème : tous les autres opérateurs (+ ou > par exemple) avaient une signature qui n'était pas adaptée à la transformation des références. Concrètement, pour un opérateur de signature `a -> b`, il a fallu passer à une signature `a -> state -> (b * state)`, et généraliser cette modification pour les fonctions currifiées. L'idée est que l'on veut pouvoir faire ceci lorsque l'on transforme `2 + 2` :

```
let (_v2, _s2) = (fun _s -> (2, _s)) (_s) in
let (_v1, _s1) = (fun _s ->
    let (_v2, _s2) = (fun _s -> (1, _s)) (_s) in
    let (_v1, _s1) = (fun _s -> (+, _s)) (_s2) in
    ((_v1) (_v2)) (_s1)) (_s2) in

((_v1) (_v2)) (_s1)
```

Notez que cette transformation est quelque peu limitée. En particulier, par construction, elle ne conserve pas l'état d'une commande à l'autre dans le REPL, puisque l'état est détruit en fin d'évalutation d'une expression.

6 Compilation et machine SECD.

Nous avons souhaité supporter, dans la partie compilation, la *quasi*-totalité des fonctionnalités proposées par le langage lorsqu'il est interpreté – par exemple la gestion des références et des exceptions, des types, des constructeurs, des tuples, des arrays déclarés en inline ou du pattern matching.

Nous utilisons donc un jeu d'instructions un peu plus complet que celui d'une SECD "classique".

```
type bytecode =
  instruction list

and instruction
  = BConst of Ast.constant
  | BTuple of int
  | BArray of int
  | BConstructor of string * int
  | BArraySet
  | BArrayRead
  | BAccess of identifier
  | BEncap of bytecode
  | BTry of pattern
  | BRaise
  | BClosure of pattern * bytecode
  | BRecClosure of identifier * pattern * bytecode
  | BLet of pattern
  | BEndLet
  | BApply
  | BBranch
  | BReturn
```

Notons que, une fois n'est pas coûtume, notre choix initial de supprimer la distinction entre opérateurs et fonctions nous a quelque peu ralenti. Concrètement, nous n'avons pas pu passer à une implémentation utilisant uniquement des entiers de De Bruijn, puisqu'il nous fallait tout de même toujours faire référence à nos opérateurs par leur nom ; et nous avons aussi essayé, sans succès, de passer à la machine Zinc, du fait du nombre de transitions à adapter pour faire fonctionner les `CMetaClosures`.

Bibliographie

[1] Oleg Kiselyov. Efficient and insightful generalization.

[2] Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3) :157–166, 1966.