

TP 8— LANGAGES DE PROGRAMMATION <http://www.nicollet.net/colles/>

Les énoncés des travaux pratiques seront disponibles sur internet quelques jours avant la séance correspondante, à l'adresse indiquée ci-dessus. Les questions en rapport avec les TP, Caml Light ou l'informatique sont les bienvenues et peuvent être envoyées à victor.nicollet@ens.fr

Un programme écrit dans un langage de programmation peut être décrit comme un arbre dont chaque noeud est un élément : par exemple une opération, une valeur ou une variable. Nous allons ici créer un petit langage de programmation simple et écrire un programme qui comprend ce langage et permet d'en exécuter les programmes.

1 ARBRE DE SYNTAXE

Pour représenter un programme, on définit un type d'arbre qui décrit sa syntaxe (on parle également de grammaire). Pour commencer, notre langage ne contient que des additions, des soustractions, et des entiers. Cela le rend très simple: exécuter un programme consiste alors uniquement à effectuer les opérations mathématiques et à renvoyer le résultat, comme sur une calculatrice.

```
type expr =
  | EVal of int
  | EAdd of expr * expr
  | ESub of expr * expr;;
```

- ▷ **Question 1.** Écrire une fonction récursive `eval` qui exécute un tel programme. S'assurer que le programme `1+2` renvoie bien 3. ◀

2 VARIABLES ET CONTEXTE

Un langage de programmation contient des variables. Dans un langage comme Caml Light, on dispose de deux opérations: `let` associe un nom à une valeur, et utiliser le nom permet d'accéder à la valeur associée. On ajoute à l'arbre les nœuds `ELet` et `EVar` pour représenter ces deux opérations.

```
...
| ELet of string * expr * expr
| EVar of string
```

Lors de l'exécution, il faut maintenant se souvenir de quelle variable est associée à quel nom: on appelle cela le contexte d'exécution. Pour le représenter, on utilise une liste `(string * int) list` et la fonction `assoc`.

- ▷ **Question 2.** Modifier `eval` pour prendre en compte le contexte (qui sera fourni comme un argument supplémentaire, initialement vide) et les deux nouvelles opérations.

S'assurer que le programme `let x = 3 in x * x` renvoie bien 9, et que le programme `1 + x` indique une erreur (car `x` est non défini). ◀

3 BOOLÉENS ET CONDITIONNELLES

On travaille maintenant avec des valeurs qui sont soit des entiers, soit des booléens, à l'aide du type ci-contre. On souhaite pouvoir gérer les opérations `=`, `<`, `&&`, `not` et `if .. then .. else ..`, qu'il va donc falloir intégrer au type de l'arbre de syntaxe.

```
type value =
  | VInt of int
  | VBool of bool
```

Attention: il faut maintenant vérifier le type des valeurs au moment de l'exécution. Par exemple, essayer d'additionner un entier et un booléen doit renvoyer une erreur.

- ▷ **Question 3.** Modifier `eval` et `expr` pour prendre en compte les booléens et les nouvelles opérations. S'assurer que le programme `let n = 3 in if n = 2 then n else n + n` renvoie bien 6. ◀

4 VÉRIFICATION DES TYPES

Afin d'éviter les erreurs pendant l'exécution, on veut vérifier à l'avance que le programme est correct: les bons types sont utilisés et toutes les variables sont définies. Ainsi, les opérations arithmétiques utilisent des entiers, les opérations logiques des booléens, les deux membres d'une égalité (et les deux résultats d'un `if`) ont le même type et une variable a le type de la valeur qui lui est associée—il va donc falloir conserver un contexte de type pour la vérification.

▷ **Question 4.** Écrire une fonction `check` qui vérifie qu'un programme est bien typé (on pourra s'inspirer de `eval`). ◀

5 FONCTIONS SIMPLES

On veut maintenant représenter les fonctions comme `fun x -> x + y`. En terme d'expression, une fonction est un nom de variable (le paramètre `x`) et une sous-expression (`x+y`). En terme de valeur, une fonction contient le nom de la variable, la sous-expression, et le contexte dans lequel la fonction a été définie (et qui contient par exemple la valeur de `y`). C'est pour cela que la fonction `let y = 3 in fun x -> x + y` est la fonction qui ajoute 3 à son argument: elle se souvient de l'association `y=3` en conservant le contexte de sa définition, ce que l'on appelle une *clôture*.

Appeler une fonction consiste alors à évaluer son argument, puis à l'ajouter au contexte contenu dans la fonction en l'associant au nom du paramètre, et à exécuter la sous-expression.

▷ **Question 5.** Modifier `value`, `expr` et `eval` pour pouvoir définir et appeler des fonctions.

Tester `let double = fun x -> x+x in double (double 3)` et s'assurer que cela vaut bien 12.

Peut-on également transformer `check` facilement? ◀

6 FONCTIONS RÉCURSIVES

Problème: avec notre `ELet`, on ne peut pas associer une valeur à un nom pendant la définition de la valeur. Cela interdit donc les fonctions récursives. On modifie donc la définition des fonctions: dorénavant, on n'autorisera plus les définitions `fun x -> ..` mais uniquement les définitions `let rec f x = .. in f`.

Comment réaliser cela en Caml Light? Puisqu'on ne peut associer la valeur au nom au moment où la fonction est définie, on choisit de le faire au moment où elle est appelée.

Pour cela, une fonction contient également son propre nom. Au moment de l'appel, on ajoute au contexte la fonction elle-même, associée à son nom. Elle devient ainsi utilisable à l'intérieur de sa propre sous-expression.

▷ **Question 6.** Modifier `value`, `expr` et `eval` pour pouvoir définir et appeler des fonctions récursives.

Tester la fonction 91 de MacCarthy: $mc(x) = x - 10$ si $x > 100$ et $mc(x) = mc(mc(x+11))$ sinon. Pourquoi l'appelle-t-on fonction 91? ◀

7 SYSTÈME DE TYPES

Les fonctions introduisent des difficultés au niveau de la vérification des types: en effet, on ne peut deviner le type d'un paramètre tant qu'on n'a pas appelé la fonction. Il faut donc choisir un autre mode de vérification.

▷ **Question 7.** Proposer un système de types inspiré de celui de Caml Light. Quel genre d'algorithme pourrait-on utiliser pour vérifier ces types? Quelles sont les contraintes de complexité sur cet algorithme? ◀