Modularité : TD et TME

LU3IN032 : Programmation comparée

Basile Pesin

15 mars 2021

Table des matières

1 TD : Récurseurs fonctionnels modulaires

Ce sujet est librement inspiré de [1].

1.1 Structure d'expressions

On va travailler sur un petit langage d'expressions arithmétiques et logiques. La grammaire du language est donnée en 1. Le langage comportes des constantes (entière ou booléennes), des opérateurs unaires et binaires (on prendra quelques opérateurs arithmétiques et logiques tels que `+`, `*`, `==`, `<`, `||`, `&&`). En OCaml, on utilisera des types sommes pour exprimer ces opérateurs : `type unop_t = Neg | Minus` pour les opérateurs unaires, et `type binop_t = Plus | Mult | ...` pour les opérateurs binaires.

On peut également exprimer une alternative sur une valeur booléene. Enfin, le programme peut comporter des variables, qui peuvent être déclarées par let-binding.

```
<e> := <int>
    | true | false
    | <x>
    | ◇ <e>
    | <e> ⊕ <e>
    | if <e> then <e> else <e>
    | let <x> = <e> in <e>
```

Listing 1 – Grammaire du langage

On peux écrire de nombreux algorithmes manipulant cette structure de données. Le but de cette séance est donc de montrer qu'on peut factoriser les traitements communs à ces algorithmes. Pour ce faire, on s'appuiera sur le système de modules d'OCaml.

Donnez un premier encodage de cette structure d'expression dans un ADT OCaml.

1.2 Récurseur : `fold_expr`

Quand on travaille avec des structures arborescentes (comme nos expressions) un traitement usuel est le suivant : descendre récursivement dans les fils d'un nœud pour calculer un résultat, puis combiner les résultats issus des différents appels récursifs. C'est alors la manière de combiner ces résultats qui diffère d'un algorithme à l'autre.

Par exemple, pour une structure d'arbres binaires, les fonctions `hauteur` (qui calcule la hauteur, ou profondeur maximale d'un arbre) et `to_list` (qui convertit un arbre en liste par parcours infixe) peuvent toutes deux être vues sous cet angle :

```
type 'a btree = Leaf | Node of ('a btree * 'a * 'a btree)

let rec hauteur t =
  match t with
  | Leaf -> 0
  | Node (l, _, r) -> 1 + max (hauteur l) (hauteur r)

let rec to_list t =
 match t with
 | Leaf -> []
 | Node (l, d, r) -> (to_list l)@[d]@(to_list r)
```

Listing 2 – Parcours d'arbres

C'est aussi le cas de beaucoup d'autres fonctions. Pour simplifier le travail du programmeur implémentant ces différents algorithmes, il serait utile de factoriser la "structure de la récursion", qu'on appellera le récurseur. Le programmeur n'aurait alors plus qu'à fournir les fonctions remplissant les cas de bases, et combinant les résultats des appels récursifs.

1.2.1 Signature de `Folder`

Ecrire la signature de module `Folder`. Celui ci contient un type `t`, qui correspond au résultat final de l'opération de fold sur une expression. Cette signature définit les fonctions `fold_litint`, `fold_litbool`, etc... qui seront instanciées par l'utilisateur. Ces fonctions donnent les cas de base et permettent de combiner les résultats d'appels récursifs. Par exemple, la fonction combinant les appels récursifs sur les sous-expressions d'un opérateur binaire aura la signature `fold_binop : t binop -> t`, où `t binop` est un raccourci pour `(binop_op * t * t)`.

1.2.2 Module `Fold`

Ecrire le foncteur `Fold`, qui prend en paramètre une instance de `Folder`. Le module expose une unique fonction récursive `fold_expr : expr -> t` qui descends recursivement dans l'expression et applique les fonctions du `Folder` pour calculer le résultat.

1.2.3 Exemple 1 : `string_of_expr`

Ecrire un module `ToString` instantiant `Folder` avec `string` comme type de retour. Utiliser `fold_expr` instantiée avec le module `ToString` permet de produire une représentation de l'expression dans une chaîne de caractères, pour affichage. Par exemple,

```
module TS = Fold(ToString)
let _ =
  print_endline (TS.fold_expr (BinOp (Plus, (LitInt 2), (Variable "x")))); (* (2 + x) *)
  print_endline (TS.fold_expr (Let ("x", LitInt 42, Variable "x"))) (* let x = 42 in x *)
```

1.2.4 Exemple 2 : Variables libres dans une expression

Les variables "libres" d'une expression sont les variables apparaissant dans l'expression qui n'ont pas été liées par un let-binding. Par exemple, dans l'expression `let x = 42 in x + y`, `y` est une variable libre mais pas `x`. Ecrire, sous forme de `Folder`, une fonction collectant les variables libres d'une expression. Quel sera le type de retour `t` ? Autrement dit, comment représenter un ensemble de variables libres ?

1.3 Récurseur avec environnement

Le problème de notre `Folder` actuel est que l'utilisateur n'a accès à aucune donnée pendant l'évaluation, si ce n'est la structure de l'expression traitée. Puisque notre langage comporte des variables, on aura souvent besoin de manipuler un environnement (associant les variables à des valeurs) pendant la récursion.

1.3.1 Signature de `EnvFolder`

Ecrire la signature de module `EnvFolder` : comme `Folder`, elle doit être instanciée par l'utilisateur qui fournit les fonctions `fold_litint`, `fold_litbool`, etc... Ces fonctions prennent toutes un paramètre en plus : l'environnement, qui contient des valeurs du même type `t` que le résultat. Par exemple, on a `fold_binop : t env -> t binop -> t`. Comment implémenter le type `t env` ?

1.3.2 Module `EnvFold`

Ecrire le module `EnvFold`, qui prend en paramètre une instance de `EnvFolder`. Comme le module `Fold`, il expose une unique fonction récursive `fold_expr : (env t) -> expr -> t` qui parcourt l'expression. Dans le cas particulier du let-binding, c'est cette fonction qui aura la charge de mettre à jour l'environnement.

1.4 Constructeur récursif : `unfold_expr`

Dans l'exercice précédant, on a vu comment utiliser un `Folder` pour calculer une valeur à partir d'une expression. On aimerait maintenant faire le contraire : construire une expression à partir d'une valeur passée en paramètre. Cela permettrait par exemple de définir un parseur, qui produirait l'arbre de l'expression à partir d'une chaîne de caractères donnant sa représentation textuelle, on d'écrire un générateur aléatoire de programmes.

Comme pour les récurseurs vus précédemment, on va essayer de factoriser la récursion dans un module `Unfold`, et de permettre à l'utilisateur de fournir uniquement la partie "contrôle" par une signature `Unfolder`

1.4.1 Signature de `Unfolder`

Donner la signature de module `Unfolder` instantiable par l'utilisateur. Un `Unfolder` prend une donnée de type `t` en entrée.

Attention ! Même si ce n'est pas l'`Unfolder` qui effectue les appels récursifs, il doit quand même choisir les expressions construites : autrement dit, il doit renvoyer une valeur construite comme une expression, avant d'avoir pu calculer les sous-expressions. Quel doit donc être le type de retour de l'`Unfolder` ? Vous aurez peut-être besoin de modifier l'AST des expressions.

1.4.2 Module `Unfold`

Ecrire le foncteur `Unfold`, qui prend en paramètre une instance d'`Unfolder`, et expose une fonction `unfold_expr : t -> expr` qui construit une expression.

1.5 Cas particulier : expressions à expressions

On a vu ci-dessus un `Folder`, qui construit une valeur à partir d'une expression, et un `Unfolder`, qui construit une expression à partir d'une valeur. Peut on imaginer construire une expression à partir d'une expression ? En fait oui, on peut considérer un tel module (qu'on appellera `Mapper`) comme un cas particulier de ces deux modules, ou la valeur en sortie / entrée est également une expression.

1.5.1 Définition d'un `Mapper`

Donnez deux définitions de la signature `Mapper` comme spécialisation du `Folder` et de l'`Unfolder`. Quelle est la différence fondamentale entre ces deux versions de `Mapper` ?

2 TME : Applications

2.1 Evaluation et Typage

On s'intéresse dans cette section aux expressions "closes", c'est-à-dire qui n'ont pas de variables libres.

2.1.1 Evaluation

Ecrire un module d'évaluation pour de telles expressions. On utilisera l'un des récurseurs définis précédemment (lequel ?). Le type de retour `t` sera `type value = Bool of bool | Int of int`. Attention, il est possible que la fonction échoue pour certaines expressions. Vous utiliserez le mécanisme d'exceptions d'OCaml pour traiter ces cas d'échec.

Quelques exemple d'évaluation :

— `let x = 40 in (x + 2)` s'évalue à `42`
— `if (12 < 30) then 42 else 24` s'évalue à `42`
— `let x = false in x || x` s'évalue à `false`

2.1.2 Vérification de type

Dans la fonction précédente, on a vu que l'évaluation échoue pour certaines expressions. Un moyen de se protéger de ces cas d'echec est d'établir une discipline de typage statique auquel les expressions doivent se conformer. On présente ci-dessous notre (simple) système de type. La règle `E |- e : t` se lit "l'expression `e` à le type `t` dans l'environnement `E`". `E` associe les variables à leur type.

— `E |- <int> : int`

— `E |- <bool> : bool`
— si `E(x) = t`, alors `E |- x : t`
— si `E |- e1 : int` et `E |- e2 : int` alors `E |- e1 + e2 : int` (et de même, avec les types raisonnables pour les autres opérateurs)
— si `E |- c : bool` et `E |- e1 : t` et `E |- e2 : t` alors `E |- if c then e1 else e2 : t`
— si `E |- e1 : t1` et `E+(x : t1) |- e2 : t2` alors `E |- let x = e1 in e2 : t2`

En utilisant le récurseur avec environnement (où l'environnement sera celui de typage) écrire un module de vérification de types. Le type de retour `t` sera `type typ = Tbool | Tint`.

Y-a-t-il des expressions bien typées dont l'évaluation peut échouer ?

2.2 Transformations de programmes

On va maintenant écrire quelques modules permettant de transformer des expressions, en s'appuyant sur l'interface `Mapper` définie dans 1.5.1. Dans les questions suivantes, vous utiliserez l'interface de `Mapper` basée sur le `Folder`.

2.2.1 Elimination de code mort 1

Dans une expression `let x = e1 in e2`, où `x` n'apparaît pas dans `e2`, on dit que `e1` est un code "mort" : il ne sert à rien puisque la valeur calculée n'est jamais utilisée. On peut donc raisonnablement transformer cette expression en `e2`. Ecrire un `Mapper DeadCodeElim1` qui effectue cette transformation, et le tester. Vous pourrez utiliser le collecteur de variables libres définit en 1.2.4.

2.2.2 Elimination de code mort 2

Dans la même démarche d'élimination du code mort, on peut transformer une expression de la forme `if true then e1 else e2` en `e1` et `if false then e2 else e1` en `e2`. Ecrire un `Mapper DeadCodeElim2` qui effectue cette transformation, et le tester.

2.2.3 Inlining

Dans une expression `let x = e1 in e2`, où `x` est utilisée une et une seule fois dans `e2`, il est raisonnable d'éliminer le let-binding, et de remplacer textuellement `x` par `e1` dans `e2`. Ecrirer un mapper qui effectue cette transformation. Vous ne pourrez pas utiliser directement l'interface `Mapper` car vous aurez besoin de vous appuyer sur `FolderEnv`.

Vous pouvez modifier le collecteur de variables libres pour lui permettre de compter le nombre d'occurrences d'une variable libre dans une expression.

2.2.4 Bonus : mapper par défaut

Les mappers définis ci-dessus partagent encore beaucoup de code en commun : en effet, pour chacun d'entre eux, seul un ou deux constructeurs des expressions déclenche un traitement particulier. Dans les autres cas, les fonctions ne font qu'appliquer le constructeur approprié.

On aimerait pouvoir factoriser ces cas "par défaut" et proposer à l'utilisateur de ne définir que les fonctions opérant des traitements particuliers.

Définir un `Mapper DefaultMapper` qui effectue les traitements par défaut décrits ci-dessus. Redéfinir `DeadCodeElim1` et `DeadCodeElim2` pour qu'ils utilisent ce mapper. Vous pourrez utiliser l'inclusion de modules.

2.3 Génération de programmes

Pour tester toutes nos fonctions, on aimerait pouvoir générer des programmes. Ecrivez un `Unfolder` qui génère des programmes bien typés aléatoires. Pour cela, vous pourrez utiliser les fonctions du module Random d'OCaml. La fonction de génération produite aura la signature suivante : `unfold_expr : (int * typ * (typ env)) -> expr`. Le premier paramètre du triplet est la profondeur maximum de l'expression à génerer, le second le type de l'expression à génerer et le troisième un environnement contenant les variables utilisables par l'expression, et leur type.

Références

[1] Vladimir Keleshev. Map as a recursion scheme in ocaml. 2018.