TD

Compilation de langages fonctionnels

Dans ce TD, on part d'un langage fonctionnel de haut niveau, et on étudie sa compilation en un langage de bas niveau (on compilera le C) muni d'une bibliothèque d'aide à l'exécution *(run-time system)*.

Les différentes étapes de cette compilation sont les suivantes :

Inférence de type : on décore chaque nœud de l'AST du programme avec son type. En principe vous savez faire.

Désucrage *(desugaring)* : il s'agit d'enlever le sucre syntaxique pour se ramener à un langage fonctionnel minimal (LFM).

Optimisations sur le LFM : on peut essayer d'évaluer à la compilation tout ce qui peut l'être, de remplacer certains appels de fonction par le corps de la fonction, et de partager certaines sous-expressions communes. Ce genre d'optimisation est aussi utilisé pour la compilation de langages séquentiels.

Génération de code : une fois que l'on sait ce que l'on veut générer c'est facile. Ce sera donc assez dur.

On va faire tout cela dans le cadre de Haskell (mais avec une syntaxe à la Caml quand cela nous arrange), juste parce que c'est fait comme cela dans le bouquin. La grosse différence avec Caml c'est l'évaluation paresseuse (ce qui ne simplifie pas vraiment) et la pureté intégriste au niveau de la transparence référentielle (pas d'effet de bord) – ce qui simplifie beaucoup.

Question 0-1 Quels problèmes de compilation économise-t-on en produisant du C ?

Question 0-2 La première idée pour la génération de code est de produire du C dans lequel les fonctions Caml deviennent des fonctions C. Cela marche jusqu'à un certain point, lequel (donnez des exemples) ? Que fait-on lorsque cela ne marche plus ?

1 Langage fonctionnel minimal

Pour le GBJL, un programme fonctionnel dans le Langage Fonctionnel Minimal (LFM) c'est

- un certain nombre de définitions de fonctions nommées, toutes globales (pas de fonctions locales), parce que nous voulons définir, dans notre code-objet C, une fonction pour chaque fonction du LFM,
- et une expression à évaluer qui appelle ces fonctions.
- Les expressions peuvent contenir des `let` imbriqués mais uniquement pour définir des variables, pas des fonctions auxiliaires. Pour la même raison (on en fera des variables locales de nos fonctions C).
- Il y a un tout petit ensemble de constructeurs de types prédéfinis : `Cons`, le constructeur de listes (comme son nom l'indique ?) d'arité 2, `Nil` (d'arité 0) la liste vide, `Num` pour les entiers.
- Il existe aussi des fonctions prédéfinies (de bibliothèque) comme `add`, `mul`, ... `if_then_else`, ... `fold_left`
- L'application des fonctions est curryfiée. Pas leur définition. On verra pourquoi.

Question 1-1 Définissez l'AST du LFM sous forme d'un type Caml.

Question 1-2 Traduisez en LFM, puis dessinez l'arbre de l'expression pour les programmes suivants :

```
let square x = x*x in square 3
```

```
let twice f x = f (f x) in let square x = x * x in twice square 3
```

```
let list_sum  = fold_left (+) 0  in list_sum [1; 2; 3]
```

2 Désucrage

Question 2-1 Comment désucre-t-on les types nommés comme

```
type t = Bli | Bla of integer*integer*integer
```

Question 2-2 Comment désucre-t-on les *pattern-matching* ? Dans quel ordre faut-il réaliser désucrage et inférence de type ? Comparez

```
let rec fact n = match n with
          0 -> 1
        | n -> n*(fact (n-1))

let rec list_sum l = match l with
         []    -> 0
       | x::rl -> x + (list_sum rl)
in list_sum [1; 2; 3]
```

Le LFM fournira des fonctions booléennes `isCons` et `isNil`.

Question 2-3 Le LFM est restreint à des fonctions toutes définies au même niveau[1], c'est-à-dire sans fonction locale comme la fonction `multscal` de l'exemple suivant :

```
let rec map f l =
  match l with
    [] -> []
  | x::rl -> (f x)::(map f rl)
;;

let svmult scalaire vecteur =
   let multscal x = scalaire * x
   in map multscal vecteur
;;

svmult 3 [1;2;3] ;;
```

Comment désucre-t-on une telle fonction locale ?

Question 2-4 Discutez au passage la paresse dans le `if_then_else`, dans le `mul`.

Question 2-5 Quels sont les autres types de constructions syntaxiques sucrées ? Présentent-elles des difficultés ?

Question 2-6 Si vous avez tout bien fait, la traduction du programme ci-dessus en LFM doit être relativement évidente. Faites-la.

3 Réduction de graphe

Suite aux étapes précédentes, notre programme est constitué d'un certain nombre de fonctions nommées, et d'une expression à réduire.

La réduction part de l'arbre de l'expression du programme de départ sous forme LFM, et le réécrit à l'exécution tant qu'elle peut.

Le résultat du programme est l'arbre irréductible obtenu.

L'étape de calcul dans un programme fonctionnel est la β-réduction, qui consiste à remplacer $f(E)$ par $B_{\{x:=E\}}$, si la définition de f est de la forme $f(x) = B$. Le nœud de l'AST $f(E)$ est appelé le *redex* pour cette β-réduction.

[1] Comme en C, quoi.

Le cœur de l'exécution d'un programme fonctionnel est donc un moteur à trois temps (qu'en France on appelle Machine de Krivine, mais pas dans le GBJL) :

- sélectionner un redex $f(E)$ dans l'AST,
- construire et instancier $B_{\{x:=E\}}$,
- remplacer la racine du redex par l'expression instanciée.

Question 3-1 Définissez précisément l'arité d'une fonction nommée. Définissez ce que l'on doit gérer pour chaque fonction.

On considère à présent le code suivant :

```
let twice f x = f (f x) ;;
let square n = n * n ;;
twice square 3 ;;
```

Question 3-2 Réduisez à la main l'AST correspondant à ce programme. Remarquez chaque fois qu'un objet cesse d'être pointé : il part à la poubelle en attendant le ramasse-miettes *(garbage collector)*.

Question 3-3 En général il existe à un instant donné plusieurs redex que l'on pourrait réduire. Définissez la stratégie de réduction paresseuse.

4 Génération de code

On va produire du C qui correspond exactement aux arbres des questions précédentes.

Question 4-1 Écrivez les déclarations C du type d'un nœud de l'arbre, et du type d'une fonction utilisateur.

Question 4-2 L'expression à évaluer sera un arbre C. Donnez le code généré pour notre expression `twice square 3`.

Question 4-3 Écrivez le C correspondant à une fonction utilisateur, par exemple la fonction `twice` ci-dessus.

Question 4-4 Écrivez un moteur de réduction simplifié.

Question 4-5 Raconte ce que vous savez sur le ramasse-miette.

5 Paresse optimisée

Question 5-1 Le problème avec l'évaluation paresseuse est qu'elle introduit une perte considérable en efficacité. Pourquoi ? On pourra étudier la fonction
`let avg a b = (a+b)/2 ; ;` en produisant son code tel que prévu par notre algorithme, puis tel qu'on peut le compacter. Dans quels cas peut-on et ne peut-on pas faire une telle optimisation ?

Question 5-2 On peut aussi décider, pour chaque fonction utilisateur, d'évaluer tout de suite les arguments dont on peut affirmer de manière certaine qu'ils seront utilisés dans le calcul. Alors le générateur de code pourra évaluer les arguments stricts avant d'évaluer la fonction. Cela économisera quoi au juste ? Le bon exemple est

```
let safe_div a b = if (b=0) then 0 else (a/b);;
```

Une manière de garantir une telle propriété pour certains arguments est l'*analyse de leur caractère strict*. On dit qu'une fonction $f(x)$ est stricte en x ssi, lorsque l'on évalue paresseusement $f(a)$ avec a expression non terminante, alors $f(a)$ est non terminante. Autrement dit, la fonction a toujours besoin de son argument x.

Question 5-3 En lesquels de leurs arguments les fonctions définies ci-dessous sont-elles strictes ?

```
let f x y = x+x+y
let g x y = if x>0 then y else x
let j x = j(0)
```

Question 5-4 Définissez les règles d'inférence du caractère strict. On propagera dans l'AST, des feuilles à la racine, les variables strictes de l'expression. On escamotera la question des fonctions récursives.