Compilation - TD8

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 (insultons 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. Ce problème a été survolé dans le TD précédent, et nous n'y reviendrons pas.

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. Donc ce sera dur.

1 Vue d'en haut

Question 1-1 Définissez un langage fonctionnel minimal adapté à Caml, avec son AST.

Question 1-2 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 ?

Question 1-3 Quelles fonctionnalités devra fournir l'environnement d'aide à l'exécution ?

Question 1-4 Quels problèmes de compilation économise-t-on en produisant du C ?

2 Désucrage

Question 2-1 Comment désucre-t-on les *pattern-matching* ? Dans quel ordre faut-il réaliser désucrage et inférence de type ?

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

```
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] ;;
```

[1] Comme en C, quoi.

Comment désucre-t-on une telle fonction locale ?

Question 2-3 Quels sont les autres types de constructions syntaxiques sucrées ? Présentent-elles des difficultés ?

Question 2-4 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, et d'une expression à réduire. Les opérateurs arithmétiques seront considérés comme des fonctions à deux arguments curryfiées dans un premier temps.

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.

Le cœur de l'exécution d'un programme fonctionnel est donc un moteur à trois temps :
- 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 ce que l'on doit stocker 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 on peut à un instant donné réduire plusieurs redex. Définissez une stratégie qui choisit un redex utile : si la fonction d'un redex est une fonction utilisateur, on peut l'appliquer quelques soient ses arguments, mais si c'est un opérateur prédéfini, il faut que les arguments soient des feuilles.

La suite discute l'ordre de réduction.

4 Évaluation en appel par valeur ou évaluation paresseuse

Question 4-1 Soient les deux programmes suivants, le programme de droite étant le résultat d'une β-réduction appliquée au programme de gauche :

```
let rec loop z = if z>0 then z
else loop z
in
let f x = if y>8 then x else (-y)
in
f (loop y)
```

```
let rec loop z = if z>0 then z
else loop z
in
let f x = if y>8 then x else (-y)
in
if y>8 then (loop y) else (-y)
```

Pourquoi ne peut-on pas dire que les deux programmes ont strictement le même comportement, si l'on suppose une évaluation en appel par valeur ?

Question 4-2 On peut retrouver le même comportement en adoptant une stratégie d'évaluation *paresseuse*. L'idée est alors de n'évaluer l'argument d'une fonction que lorsque l'on en a effectivement besoin

(au lieu de l'évaluer systématiquement comme en appel par valeur). Quel choix de redex permet l'évaluation paresseuse ? Quel choix de redex permettait l'évaluation par appel par valeur ?

Question 4-3 Caml, il est paresseux ?

5 Génération de code

On va produire du C qui correspond exactement à l'AST précédent.

Question 5-1 Écrivez les déclarations C du type d'un nœud de l'arbre, et du type d'une fonction utilisateur.

Question 5-2 L'expression à évaluer sera un arbre C. Donnez le code généré pour notre expression `twice square 3`.

Question 5-3 Écrivez le C correspondant à une fonction utilisateur, par exemple la fonction `twice` ci-dessus.

Question 5-4 Écrivez un moteur de réduction simplifié.

Question 5-5 Le ramasse-miette, c'est le prochain TD...

6 Paresse optimisée

Question 6-1 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 6-2 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 6-3 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 6-4 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.