Examen du cours "Compilation"
Durée: 3 heures
Tout document autorisé.

25 mai 2016

A. La chaîne de compilation

Cette première partie de l'examen vise à valider votre compréhension de la chaîne de compilation réalisée cette année. Pour cela, on s'intéresse à la compilation du programme P suivant :

```
type list := { Cons : int * list | Nil }.

rec map f l :=
  l ? {
  | Nil => Nil
  | Cons (x, xs) => Cons (f x, map f xs)
  }.

val lsucc := map (\x => x + 1) (Cons (72, Nil)).
```

Il est bien sûr difficile de compiler un programme manuellement, c'est d'ailleurs pour cela que nous écrivons des compilateurs pour le faire pour nous. Dans les exercices suivants, le correcteur sera donc indulgent si quelques erreurs mineures sont insérées dans les programmes compilés. De même, si votre compilateur a lui-même quelques erreurs ou quelques incomplétudes, vous pouvez les passer sous silence en supposant que votre compilateur fonctionne correctement.

Exercice 1 *(Hopix vers Hobix)*

1. *Quel est le rôle de la passe de compilation de* HOPIX *à* HOBIX *?*
2. *Donnez le code de P compilé en* HOBIX *tel que le produirait votre compilateur.*
3. *Que produit l'évaluation du programme compilé ?*

□

Exercice 2 *(Hobix vers Fopix)*

1. *Quel est le rôle de la passe de compilation de* HOBIX *vers* FOPIX *?*
2. *Donnez le code de P compilé en* FOPIX *tel que le produirait votre compilateur.*
3. *Quelles améliorations pourriez-vous faire à votre compilateur pour qu'il produise un code compilé plus efficace pour P ? Donnez un tel code.*

□

Exercice 3 *(Fopix vers Retrolix)*

1. *Quel est le rôle de la passe de compilation de* FOPIX *vers* RETROLIX *?*
   (Ignorez l'allocation des registres pour le moment.)

2. *Donnez le code de P compilé en* RETROLIX *tel que le produirait votre compilateur.*

□

Exercice 4 *(Optimisations du code Retrolix)*

1. *À quoi sert l'allocation des registres ? Comment est-elle réalisée dans* FLAP *?*
2. *Donnez le code de P compilé en* RETROLIX *après allocation des registres. Vous donnerez toutes les étapes qui ont mené à cette allocation des registres.*

□

Exercice 5 *(Retrolix vers MIPS)*

1. *Quel est le rôle de la passe de compilation de* RETROLIX *vers* MIPS *?*
2. *Donnez le code de P compilé en* MIPS *tel que le produirait votre compilateur.*

□

B. Compilation d'un mécanisme de déclenchement et rattrapage erreurs

Dans cette partie, on étudie la compilation de l'extension d'HOPIX par le mécanisme d'erreur décrit dans l'examen d'interprétation des programmes du premier semestre. Cette étude va consister en une comparaison de trois méthodes de compilation distinctes de ce mécanisme.

Les deux premières méthodes s'appuient sur une passe de compilation de HOPIX avec **error/try ... orelse ...** vers HOPIX sans **error/try ... orelse ...** sans changement sur les passes suivantes de compilation. La troisième méthode est plus intrusive car elle nécessite la modification de tous les langages du compilateur et donc aussi de toutes les passes.

Chaque méthode sera décrite de façon informelle. L'objectif de chaque exercice (sauf le dernier) est de vérifier que vous avez compris le principe de chaque méthode proposée. Le dernier exercice vous demande de comparer les avantages et les inconvénients de chaque méthode.

Pour illustrer ces méthodes, on utilisera l'exemple P suivant :

```
rec div a b :=
  if b = 0 then error
  else if a <= 0 then 0
  else 1 + div (a - b) b
  fi fi.

val x := try div 50 10 orelse 0.
```

On rappelle que la construction **error** fait passer l'évaluation du programme dans un mode où tous les calculs sont sautés jusqu'à ce que l'on rencontre un gestionnaire d'erreur. Dans ce cas, le gestionnaire d'erreur est évalué à l'aide de l'expression suivant le mot-clé **orelse**. Si aucune erreur n'est levée alors la partie **orelse** du gestionnaire d'erreur est ignorée.

Exercice 6 *(Monade d'erreur)*

Une façon de supprimer l'usage des erreurs consiste à transformer le programme de façon à ce qu'il retourne une valeur à deux cas : un cas `Normal (v)` *pour les expressions qui se sont exécutées sans erreur et un cas* `Error` *pour les autres. Toute sous-expression d'une expression doit être transformée de façon à prendre en compte les cas d'erreur. Par exemple la déclaration :*

```
val x := (f 0 + g 1).
```

devient :

```
val x :=
   f (Normal 0) ? {
   | Error => Error
   | Normal rf =>
     g (Normal 1) ? {
     | Error => Error
     | Normal rg => Normal (rf + rg)
     }
   }.
```

Pour rendre lisible le code compilé, on peut utiliser les opérateurs suivants (ce sont ceux de la monade dite d'erreur) :

```
val return x := Normal (x).

val bind x f :=
   x ? {
   | Error => Error
   | Normal (x) => f x
   }.
```

Le programme compilé précédent s'écrit alors :

```
val x :=
   bind (f' 0) (\x =>
   bind (g' 1) (\y =>
   return (x + y))).
```

(En supposant que `f'` *et* `g'` *sont les versions compilées de* `f` *et* `g`*.)*

1. *Comment compiler la construction* `error` *dans la monade d'erreur ?*
2. *Comment compiler la construction* `try ... orelse ...` *dans la monade d'erreur ?*
3. *Donnez le code de P compilé avec cette technique.*

□

Exercice 7 *(Compilation en style par continuation avec continuation d'erreur)*
La compilation par continuation explicite la suite du calcul sous la forme d'une fonction pris en argument de façon systématique par toute sous-expression `e` *du calcul. Cette fonction attend le résultat de* `e` *pour continuer le calcul. Ainsi, toute expression est transformée en une fonction de la forme :* `\k => ... k v ...` *où* `k` *est une fonction, la continuation du calcul, qui doit être appliquée au résultat* `v` *de l'évaluation de* `e` *pour continuer normalement le calcul.*

Par exemple, pour compiler :

```
f 0 + g 1.
```

On écrit :

```
\k =>                  ** k is the continuation expecting the result of "f 0 + g 1".
  val after_f x :=     ** x is the result of "f 0".
    val after_g y := ** y is the result of "g 1".
       k (x + y)       ** in the end, the continuation is called with x + y.
    in
    g after_g 1        ** The continuation of "g 0" is returning "f 0 + g 1".
  in
    f after_f 0.       ** The continuation of "f 1" computes "g 1" and returns "f 0 + g 1".
```

Le programme précédent est écrit dans un style dit indirect, ce qui le rend difficile à lire. Encore une fois, une monade peut nous servir à le réécrire de façon plus lisible :
(Ici, il s'agit de la monade dite de continuation.)

```
val return x :=
  \k => k x.

val bind m f :=
  \k =>
    val after_m x := f x k in
    m after_m.
```

En effet, on obtient alors :

```
val x :=
   bind (f' 0) (\x =>
   bind (g' 1) (\y =>
   return (x + y))).
```

(En supposant que `f'` *et* `g'` *sont les versions compilées de* `f` *et* `g`*.)*
Pour traiter les erreurs, il suffit de se doter d'une continuation supplémentaire : elle représente la continuation à exécuter en cas d'erreur. Ainsi, la monade devient :

```
val return x :=
  \k on_error => k x.

val bind m f :=
  \k on_error =>
    val after_m x := f x k on_error in
    m after_m on_error.

val error :=
  \k on_error => on_error 0.

val tryorelse e on_error :=
  \k old_on_error => e k on_error.
```

1. *Expliquez le principe de fonctionnement de* `error`*.*
2. *Expliquez le principe de fonctionnement de* `tryorelse`*.*
3. *Donnez le code compilé de P implémenté à l'aide de continuations.*

□

Exercice 8 *(Marquage de pile)*

Une autre façon de compiler le mécanisme de lancement/rattrapage d'erreurs nécessite une modification plus profonde du compilateur : on étend chaque langage de HOPIX *à* RETROLIX *avec un mécanisme de lancement et de rattrapage d'erreur.*

Comme HOPIX, HOBIX *et* FOPIX *sont des langages à expressions, il n'est pas très compliqué de les étendre avec une construction* `error` *et une construction* `tryorelse` *analogues à celles de* HOPIX.

Pour RETROLIX, *c'est un peu plus compliqué : il faut d'une part étendre l'environnement avec une pile de gestionnaire d'erreurs, et d'autre part, il faut se doter de deux nouvelles instructions.*

Tout d'abord, la pile des gestionnaires d'erreurs est constituée de deux types de données : des étiquettes représentant des positions dans le code et des marques d'entrées de fonction. À chaque fois que l'on rentre dans une fonction, on pousse une marque au sommet de la pile. Quand on sort d'une fonction, on dépile tout ce qui se trouve jusqu'à la marque d'entrée de cette fonction.

La première instruction enregistre un gestionnaire d'erreurs. Elle a la forme suivante :

```
on_error_jump l1 -> l2
```

Cette instruction met à jour l'environnement d'évaluation de RETROLIX *en poussant l'adresse* `l1` *du code du gestionnaire d'erreurs au sommet de la pile des gestionnaires d'erreur puis en continuant l'exécution en* `l2`.

La seconde instruction sert à déclencher une erreur. Elle a la forme suivante :

```
error
```

L'exécution de cette instruction procède comme suit :

- *Si la pile est vide, le programme est stoppé.*
- *Si la pile n'est pas vide et que le sommet de la pile est une étiquette* `l` *de gestionnaire d'erreur alors on saute à cette étiquette.*
- *Si la pile n'est pas vide et que le sommet de la pile est une marque de début de fonction alors on sort de la fonction courante et on essaie de trouver récursivement un gestionnaire d'erreur dans la fonction appellante.*

1. *Comment compiler la construction* `error` *de* FOPIX *en* RETROLIX *?*
2. *Comment compiler la construction* `try ... orelse ...` *de* FOPIX *en* RETROLIX *?*
3. *Proposez une représentation de la pile des gestionnaires d'erreur en* MIPS.
   (Indice : Il suffit d'étendre les blocs d'activation.)
4. *Comment compiler l'instruction* `error` *de* RETROLIX *en* MIPS *?*
5. *Comment compiler l'instruction* `on_error_jump l1 -> l2` *de* RETROLIX *en* MIPS *?*

□

Exercice 9 *(Conclusion)*

1. *Comparez l'efficacité en espace et en temps de la compilation du mécanisme de lancement d'erreur en fonction des trois techniques de compilation précédentes.*
2. *Comparez l'efficacité en espace et en temps de la compilation du mécanisme de rattrapage d'erreur en fonction des trois techniques de compilation précédentes.*
3. *Quelle technique vous semble la plus intéressante à implémenter dans votre projet ? Justifiez votre réponse. N'oubliez pas de prendre aussi en compte le coût de développement de l'extension.*

□