ISTIC
UFR INFORMATIQUE - ELECTRONIQUE

Licence STS 2ème année mentions Informatique & Mathématiques

CAL - TP 3 - UN ELIMINATEUR D'EXPRESSIONS COMPLEXES

Un utilisateur du langage WHILE peut toujours se limiter à n'utiliser que des expressions limitées à un seul constructeur. La fin du chapitre 5 expose les principes et motivations de cette contrainte.

On se propose de développer un programme qui prend en paramètre un programme WHILE, plus précisément son arbre de syntaxe abstraite, et retourne l'arbre de syntaxe abstraite d'un programme de même sémantique, mais qui n'utilise que des expressions comptant au plus un constructeur. Ce programme est un compilateur du langage WHILE vers sa variante sans expressions imbriquées, qu'on peut donc désigner par $\texttt{compil}^{Scala}_{\text{WHILE},\text{WHILE}_{1cons}}$. Ce sujet peut être traité en trois séances.

Spécification

On fixe le cahier des charges suivant :

— On convient d'appeler *expression complexe* toute expression du programme source qui comporte plusieurs constructeurs. Sont considérés comme constructeurs : `nil`, `hd`, `tl`, `cons`, `=?`, et toutes les constantes, par exemple `fact` et `dofact`. On dit qu'une expression est *simple* si elle comporte au plus un constructeur. Ainsi, `(hd (tl X))` et `(hd nil)` sont complexes, mais `(tl X)` et `nil` sont simples.

— Le programme résultat ne comporte que des expressions simples.

— Le programme résultat suit fidèlement la structure du programme source. Il n'en diffère que par des séquences d'affectations qui remplacent les expressions complexes. Par exemple `X := (hd (hd Z))` est remplacé par `A0 := (hd Z) ; X := (hd A0)`.

— Les séquences d'affectations calculées pour remplacer les expressions complexes utilisent des variables nouvelles pour stocker les résultats intermédiaires (`A0` dans l'exemple précédent). On ne cherche pas à économiser les variables nouvelles, et on en alloue une pour chaque résultat intermédiaire sans se soucier de savoir si une variable déjà allouée pourrait être réutilisée.

— On évitera toutefois d'introduire des séquences d'affectations de la forme `Ai := expr ; X := Ai` car la commande `X := expr` a le même effet et ne comporte qu'une expression simple.

— Par contre, on évitera que les expressions qui contrôlent les boucles `while` et `for` ainsi que la conditionnelle `if` comportent des constructeurs ; c'est-à-dire qu'elles ne peuvent être que des variables. Cet objectif et le précédent sont antagonistes.

— On construit les nouvelles variables à partir d'une racine commune `A`, et d'un numéro d'identification. On ne vérifiera pas que la racine n'est pas déjà employée dans le programme traité.

Par exemple, le programme source

```
read X %
  Y := nil ;
  while X do
        Y := (cons (hd X) Y) ;
        X := (tl X)
  od
% write Y
```

est recomposé de la façon suivante :

```
read X %
  Y := nil ;
  while X do
        A0 := (hd X) ;
        Y := (cons A0 Y) ;
        X := (tl X)
  od
% write Y
```

Il convient de prendre garde à l'expression qui contrôle la boucle WHILE. Si elle est complexe et doit être remplacée par une séquence d'affectations de même effet, le remplacement doit avoir lieu avant la boucle *et* à la fin du corps de la boucle. En effet, on entre dans une boucle en venant des commandes qui la précèdent, ou en venant de son corps. Par exemple, le programme

```
read X %
  Y := nil ;
  while X =? nil do
        Y := (cons (hd X) Y) ;
        X := (tl X)
  od
% write Y
```

sera transformé en

```
read X %
  Y := nil ;
  A0 := nil ;
  A1 := X =? A0 ;
  while A1 do
        A2 := (hd X) ;
        Y := (cons A2 Y) ;
        X := (tl X) ;
        A0 := nil ;
        A1 := X =? A0
  od
% write Y
```

Si on omettait la séquence de commandes `A0 := nil ; A1 := X =? A0` à la fin du corps de la boucle WHILE, l'expression qui contrôle la boucle ne serait jamais réévaluée, et l'exécution de la boucle ne terminerait jamais.

Le principe de la transformation est le suivant. à chaque expression source e est associée une paire constituée d'une séquence d'affectations s et d'une expression simple e'. L'idée est que les expressions e et e' ne sont pas équivalentes, mais évaluer l'expression simple e', après avoir exécuté les affectations s, donne le même résultat que pour l'expression source e. Par exemple, l'expression `(hd (tl (hd X)))` pourra être associée à la paire $\langle$`A0 := (hd X) ; A1 := (tl A0) , (hd A1)`$\rangle$ où `A0` et `A1` sont des variables nouvelles.

Formellement, l'invariant de la transformation d'une expression complexe e en une paire $\langle s, e' \rangle$ est

$$\mathcal{SEM}_e(e, m) = \mathcal{SEM}_e(e', \mathcal{SEM}_c(s, m))$$

Chaque paire peut être construite récursivement en suivant la structure des expressions. Explicitons un cas de la définition de la fonction associant expressions et paires ; les autres peuvent facilement s'en déduire. On définit deux fonctions :

— *while1ConsExprV* si l'expression produite est réduite à une variable.
— *while1ConsExprSE* si l'expression produite est une expression simple, autre qu'une variable

On donne un exemple très partiel de la définition de ces fonctions ci-dessous :

$$\begin{array}{rrcl} \textit{while1ConsExprV}: & \text{Expression} & \to & \text{List[command]} \times \text{Variable} \\ & Hd(e) & \mapsto & \langle s \,;\, v' \texttt{:=(hd } v\texttt{)}, v' \rangle \\ & & & \text{où } \textit{while1ConsExprV}(e) = \langle s \,,\, v \rangle \\ & & & \text{et où } v' \text{ est une } \textit{nouvelle} \text{ variable} \end{array}$$

$$\begin{array}{rrcl} \textit{while1ConsExprSE}: & \text{Expression} & \to & \text{List[command]} \times \text{Expression} \\ & Hd(e) & \mapsto & (\, s \,,\, Hd(v) \,) \\ & & & \text{où } \textit{while1ConsExprV}(e) = \langle s \,,\, v \rangle \end{array}$$

On construit ces paires pour toutes les expressions, puis le programme résultat en remplaçant les expressions e par les e' et en insérant les séquences s de telle sorte qu'elles soient toujours exécutées juste avant que l'expression e' correspondante ne soit évaluée.

Plan de développement

Les définitions et fonctions de manipulation de variables nouvelles sont disponibles dans le fichier `newvar.scala`. L'objet `Newvar` offre les services suivants :

— `** mise à zéro de l'indice des variables`
`* reset: -> Unit`
L'appel `NewVar.reset()` réinitialise à 0 le compteur de variables nouvelles. Cette fonction est utile pour rendre les tests indépendants des calculs qui les ont précédés et auraient pu modifier le compteur de variables nouvelles.
— `** construction d'une nouvelle variable temporaire`
`* make: -> Variable`
L'appel `NewVar.make()` retourne l'arbre de syntaxe abstraite d'une variable nouvelle, jamais encore utilisée depuis le dernier appel à `NewVar.reset` .
Cette fonction Scala ne réalise pas une fonction au sens mathématique du terme. Cela vient de ce que le résultat de `NewVar.make()` dépend du nombre de fois où la fonction a été appelée auparavant. On dit qu'une telle fonction est *impure*. Une fonction impure contamine les fonctions qui l'utilisent dans leur définition. Les fonctions qui restent à définir seront toutes impures parce qu'elles appellent `NewVar.make`.

On propose le plan de développement suivant.

1. Définir une fonction `while1ConsExprV` qui prend une expression e et retourne une paire $\langle s, v \rangle$ où s est une séquence d'affectations et v est une variable. On doit avoir $\mathcal{SEM}_e(e, m) = \mathcal{SEM}_e(v, \mathcal{SEM}_c(s, m))$. Cette fonction est spécifiée comme suit :

   ```
   /**
   * @param expression : un AST décrivant une expression du langage WHILE
   * @return une paire constituée d'une liste d'affectations ayant le même effet
   * que l'expression et de la variable qui contient le résultat
   */
   def while1ConsExprV(expression: Expression): (List[Command], Variable)
   ```

   On fera attention aux tests. En effet, deux programmes peuvent mettre en œuvre correctement la spécification tout en produisant des séquences d'instructions différentes. Celles-ci peuvent différer par l'ordre des opérations et par la numérotation des nouvelles variables. En conséquence, un programme correct peut échouer aux tests fournis avec cette spécification ; il faut savoir s'en rendre compte et corriger l'oracle du test, et non pas le programme.

2. Définir une fonction `while1ConsExprSE` qui prend une expression e en paramètre et retourne une paire $\langle s, e' \rangle$ où s est une séquence d'affectations et e' est une expression simple. On doit avoir $\mathcal{SEM}_e(e, m) = \mathcal{SEM}_e(e', \mathcal{SEM}_c(s, m))$. Il s'agit d'une variante de la fonction précédente. Dans le cas d'expressions complexes, cette fonction pourra appeler la précédente. Sa spécification est comme suit :

   ```
   /**
   * @param expression : un AST décrivant une expression du langage WHILE
   * @return une paire constituée d'une liste d'affectations et une expression simple
   * qui, combinées, ont le même effet que l'expression initiale
   */
   def while1ConsExprSE(expression: Expression): (List[Command], Expression)
   ```

3. Définir une fonction `While1ConsCommand` qui prend une commande c en paramètre et retourne une séquence s de commandes de même sémantique, $\forall m. \mathcal{SEM}_c(c, m) = \mathcal{SEM}_c(s, m)$, mais qui ne comporte que des expressions simples (voir détails dans le cahier des charges). Cette fonction, qui utilise les précédentes, est spécifiée comme suit :

   ```
   /**
   * @param command : un AST décrivant une commande du langage WHILE
   * @return une liste de commandes ayant un seul constructeur par expression
   * et ayant le même effet que la commande initiale
   */
   def while1ConsCommand(command: Command): List[Command]
   ```

4. Définir une fonction variante de la précédente qui prend en paramètre une liste de commandes plutôt qu'une commande unique.

   ```
   /**
   * @param commands : une liste non vide d'AST décrivant une liste non vide de commandes du
   langage WHILE
   * @return une liste de commandes ayant un seul constructeur par expression
   * et ayant le même effet que les commandes initiales
   */
   def while1ConsCommands(commands: List[Command]): List[Command]
   ```

5. Définir une fonction qui prend un programme p en paramètre et retourne un programme p' équivalent ne comportant que des expressions simples.

```
/**
* @param program : un AST décrivant un programme du langage WHILE
* @return un AST décrivant un programme du langage WHILE
* de même sémantique que le programme initial mais ne contenant
* que des expressions simples
*/
def while1ConsProgr(program: Program): Program
```