Conception de langage
Examen final

janvier 2010

1 Machine à registres illimités

La machine est dotée d'une mémoire constituée d'un ensemble illimité de registres. Chaque registre contient une valeur entière (positive ou négative). Chaque registre est simplement identifié par un numéro (entier positif). Le langage de la machine ne comprend que des instructions assez élémentaires et pas d'expressions du tout. Les constantes numériques désignes soit des registres soit une valeur immédiate (dans un cas seulement, voir ci-dessous).
Les instructions sont de trois sorte :

1. les transferts avec ou sans opération :
   - `i == n` signifie le transfert de la constante `n` dans le registre `i`
   - `i1 := i2` signifie le simple transfert du contenu de `i2` dans `i1`
   - `i1 += i2` signifie le transfert dans `i1` de la somme des valeurs contenues dans `i1` et `i2`.
     En d'autres termes : `i1` reçoit `i1` + `i2`.
     On a le même genre d'instruction pour les trois autres opérations arithmétiques.
2. les alternatives selon la valeur d'un registre par rapport à 0 :
   - `ifeq i [ p1 ] [ p2 ]` signifie si le contenu de `i` est nul alors exécuter `p1` sinon exécuter `p2`.
     On a le même genre d'instruction pour les tests d'infériorité (`iflt`) et de supériorité (`ifgt`) par rapport à 0.
3. les boucles selon la valeur d'un registre par rapport à 0
   - `looplt i [ p ]` signifie tant que `i` est plus petit que 0 faire `p`.
   - `loopgt i [ p ]` signifie tant que `i` est plus grand que 0 faire `p`.

Le résultat de l'exécution d'un programme est l'état mémoire obtenu lorsque le programme s'arrête.

Lexique :

```
num = -?[0-9]+
'==' ':=' '+=' '*=' '-=' '/='
'ifeq' 'iflt' 'ifgt'
'looplt' 'loopgt'
'[' ']' ';'
```

Syntaxe

```
Prog ::= num '==' num ';'
     | num ':=' num ';'
     | num '+=' num ';'
     | num '-=' num ';'
     | num '*=' num ';'
     | num '/=' num ';'
```

```
      | 'ifeq' num '[' Prog ']' '[' Prog ']'
      | 'iflt' num '[' Prog ']' '[' Prog ']'
      | 'ifgt' num '[' Prog ']' '[' Prog ']'
      | 'looplt' num '[' Prog ']'
      | 'loopgt' num '[' Prog ']'
      | Prog Prog
```

Questions :

1. Comment modilisez vous la mémoire de la machine, son accès, sa mise à jour ?
2. Donnez la signature et les équations de la fonction sémantique P d'interprétation des programmes.

Pour définir ces équations, vous pouvez utilisez, outre ce que vous avez défini sur la mémoire, tout ce que vous savez d'arithmétique (il en faut peu), l'alternative if-then-else, la lambda abstraction et le point fixe.

2 Instructions et expressions

L'instruction `return` est utilisée dans les langages de programmation pour définir des fonctions en termes procédureaux. Par exemple, on rêverait de

```
Fun fac(n) = {
  Var r;
  r := 1;
  while (n > 0) { r := r*n; n := n-1; }
  return r;
}
```

Cependant, introduire un `return` comme nouvelle instruction nous engagerait trop loin : il faudrait savoir donner la sémantique de cette nouvelle instruction dans un contexte quelconque ; alors qu'elle ne nous interresse que dans le cas de définition de fonctions. De plus, on peut toujours faire en sorte que, dans le corps d'une définition procédurale de fonction, il n'y ait qu'une seule instruction `return` à la fin ; de surcroît cette instruction peu ne porter que sur une variable. C'est cette solution que nous envisageons.

Nous nous plaçons dans le cadre du langage impératif sans continuations où

```
P: Prog -> Mem
Ds: Decs -> (Env * Mem) -> (Env * Mem)
D: Dec -> (Env * Mem) -> (Env * Mem)
S: Stat -> (Env * Mem) -> Mem
E: Exp -> (Env * Mem) -> Data#
```

Extension de la syntaxe.
Lexique : nouveaux mots clefs `'return' 'from'`
Syntaxe : nouvelle expression

```
Exp ::= ...
      | '{' 'return' ident 'from' Instr '}'
```

Avec cette nouvelle construction, on peut définir

```
Fun fac(n) = {
  return r
  from
    r := 1;
    while (n > 0) { r := r*n; n := n-1; }
}
```

Notez que dans la syntaxe, la déclaration de r est implicite, sans valeur initiale.

Questions :

1. Donnez l'équation sémantique de cette nouvelle expression. N'hésitez pas à indiquez toute autre modification que vous jugerez nécessaire.
2. ≪Testez≫ votre nouvelle sémantique avec le programme

```
Fun f(x) = return z from z := x+1; Var x = 0; x := f(4);
```

3 Variables persistantes (statiques)

On appelle variable *persistante* ou *statique* une variable locale à une fonction qui continue à exister et conserve sa valeur lorsque la fonction a cessé d'être active. Une activation suivante de la fonction retrouvera la dernière valeur affectée à une telle variable. C'est ce qui permet par exemple d'implanter une fonction "compteur" qui donne à chaque appel la nouvelle valeur d'un compteur dont la référence reste locale à la fonction.

Par exemple, en OCaml, on écrira :

```
let cpt =
  let c = ref 0 in
  (fun () -> incr c; !c)
```

en C, on écrira :

```
int cpt() {
  static int c = 0;
  c = c+1;
  return c;
}
```

Dans notre langage modèle, on voudra écrire

```
Fun cpt() =
  return static r = 0
  from r := r + 1 ;
```

Notez qu'ici, on donne une valeur initiale à r.

Question :

1. Proposez une sémantique pour ce nouveau trait de langage.