TP 15 avril 2022

Langages Formels – Projet Analyse Syntaxique

Dans le projet, il s'agit d'implémenter un petit langage de programmation. Un programme consiste de plusieurs processus et peut être équipé de plusieurs spécifications. En fonction de vos ambitions, vous pouvez implémenter un parmi trois objectifs ; la notation prendra en compte l'objectif que vous avez choisi.

- Niveau 1 : Tester si un programme est syntaxiquement correct, et en faire un "pretty print".
- Niveau 2 : Simuler un programme aléatoirement pour tester si les spécifications sont satisfaites ou pas.
- Niveau 3 : Vérifier si les spécifications sont satisfaites en explorant systématiquement tous les états atteignables.

Vous pouvez travailler **en groupes de deux** et utiliser les fichiers du TP (notamment `langlex.l` et `lang.y`) comme base ; ces fichiers contiennent déjà quelques structures utiles, mais il faudra les adapter.

Modalités de soumission. La date pour soumettre votre projet est le dimanche 15 mai à 18h. Envoyez un mail à `schwoon@lsv.fr` avec un seul archive (zip ou tgz) qui contient les fichiers sources ainsi qu'un readme avec toutes les explications qui vous paraissent utiles, et surtout quel niveau vous avez choisi. Si vous travaillez en groupes de deux, la personne qui envoye le mail mettra son parténaire en copie.

1 Le langage

Un programme consiste d'un ou plusieurs processus. Les processus partagent des variables globales, et chaque processus peut aussi manipuler ses variables locales. Toutes les variables sont du type entier (`int`), et les opérations telles que `+`, `==`, `&&` correspondent aux opérateurs de C. Comme dans C, on équivaut 0 à 'faux' et toute valeur non-zéro à 'vrai'. Toute variable est 0 initialement.

Avec le projet, vous trouverez trois exemples (`peterson.prog`, `sort.prog`, `lock.prog`) qui illustrent la syntaxe. C'est à vous d'en dériver une grammaire concrète, et vous y avez un certain dégré de liberté, mais à minima, votre analyseur devrait comprendre ces trois programmes.

Regardons ces programmes pour illustrer quelques points particuliers :

— Les commentaires sont indiqués par `//` et vont jusqu'à la fin de la ligne.

— Les variables globales sont déclarées avant les processus. Il peut y avoir plusieurs lignes, p.ex. dans `sort.prog`.

```
var x,y,z;
var fin;
```

— Un processus possède un nom et une liste d'instructions, il termine par le mot-clé `end`. Si le processus possède des variables locales, elles sont déclarées avant les instructions.

```
proc p
   var tmp;
   ...
end
```

— Un instruction peut être (i) une affectation de variable `x := expression` ; (ii) `skip` (instruction nulle) ; (iii) `do/od` (une boucle) ; (iv) `if/fi` (alternative) ; (v) `break` (sortir d'une boucle).

— Les instructions `do` et `if` contiennent une ou plusieurs choix. Tout choix est indiqué par `::` puis une expression (garde), une fleche (`->`) et un bloc d'instructions. Un bloc peut-être exécuté si son garde est vrai. Un seul garde peut valoir `else` dans quel cas il sera pris si tous les autres gardes sont faux.

```
if
:: x > y -> tmp := x; x := y; y := tmp
:: else -> skip
fi;
```

Attention, les gardes ne sont pas forcement exclusifs dans quel cas l'un des gardes est choisi aléatoirement :

```
if
:: 1 -> skip
:: 1 -> lock := lock-1; todo := 1
fi
```

— Un `if` est exécuté une fois, un `do` est exécuté en boucle. Une instruction `break` sort du `do` le plus proche qui l'englobe.

Sémantique. L'état d'un programme est défini par le compteur d'instructions de tout processus, et par la valeur de toutes les variables globales et locales.

On passe d'un état à l'autre en exécutant une instruction dans n'importe quel processus. L'exécution d'un `if` ou `do` consiste à évaluer les gardes et choisir un bloc d'instructions à exécuter prochainement si son garde est vrai. Si tous les gardes sont faux et qu'il n'y a pas de `else`, le processus est bloqué.

2 Les spécifications

Après les processus, le programme peut contenir des spécifications, chacune indiqué par le mot-clé `reach` et une expression sur les variables globales.

```
reach   c == 1  // accessible
reach   c == 2  // accessible
reach   c == 3  // non accessible
```

Une spécification est satisfaite s'il existe un état accessible dans lequel son expression est vraie. Dans les exemples, les commentaires indiquent si les spécifications devraient être satisfaites ou pas.

3 Les objectifs

Quelques détails sur les objectifs parmi lesquels vous pouvez choisir. Il suffit d'en réaliser un seul, p.ex. si vous faites le niveau 2, inutile de faire le niveau 1 aussi.

Niveau 1. Vous devez tester si un programme donné est syntaxiquement correct. Du coup il faut au moins implémenter une grammaire et créer un arbre syntaxique qui le représente en mémoire. Une représentation textuelle de cet arbre ("pretty print") sera alors affichée sur l'écran.

Il n'y a pas d'exigence particulière sur la ≪beauté≫ du pretty-print, mais le programme affiché doit au moins être sémantiquement équivalent au programme de départ.

À noter que le pretty-print doit être réalisé à partir de l'arbre syntaxqiue, notamment, il n'est pas permis de réaliser cet affichage *pendant* l'analyse syntaxique.

Niveau 2. Après l'analyse syntaxique, livrez-vous à une méthode «Monte Carlo» pour tester si les spécifications sont satisfaites. Pour cela, on simule les calculs du programme donné. Notons qu'il y a deux sources de non-déterminisme : (i) le choix de processus qui exécutera la prochaine instruction ; (ii) le choix de bloc dans `if`/`do` si plusieurs gardes sont vrais. Dans ce cas il faut faire des choix aléatoires dans la simulation. Il est donc utile de faire un grand nombre d'exécutions suffisamment longues (p.ex. 100 calculs à 100 instructions). Dans tout état ainsi atteint, on teste si les spécifications sont satisfaites. À la fin, on affichera pour toute spécification si elle était vraie pendant l'un des calculs.

Comme au niveau 1, il faut créer une structure de données qui représente le programme. En plus, il faut réfléchir à la sémantique et à comment représenter l'état actuel du programme.

Niveau 3. Ici, on énumère systématiquement tous les états accessibles du programme donné. Ensuite, pour toute spécification, on teste s'il existe un état accessible satisfaisant. Le nombre d'états accessibles est toujours fini, et on va travailler sous l'hypothèse que ce nombre est «petit» (on peut tous les stocker en mémoire).

En plus du niveau 2, il faut mémoriser les états déjà explorés. Il est recommandé d'utiliser une table de hachage pour ce faire et d'implémenter une «worklist» qui contient les états qu'il reste à explorer. Dans les fichers du projet, vous trouvez une implémentation d'un table de hachâge ainsi qu'une copie de crc32 (du projet GNU) qui convient comme fonction de hachage sur un bloc de mémoire. Vous avez le choix d'utiliser ces implémentations dans votre projet, ou de les remplacer par votre propre code.

4 Améliorations diverses

Quelque soit le niveau que vous choisissez, vous pouvez complémenter votre implementation par quelques améliorations que vous documenterez dans le readme. En voici des propositions :

- Si le programme donné est syntaxiquement incorrect, donner un message d'erreur utile (p.ex. indiquer la ligne et colonne ou l'analyse syntaxique a échoué).
- Permettre aux instructions de porter une étiquette, p.ex.

```
proc p
```

```
        ...
        ici: skip
        ...
    end
```

Ces étiquettes figureraient dans le pretty-print ou bien dans les spécifications qui contiennent alors des prédicats genre `p@ici` (le processus `p` est à l'instruction `ici`).

— Au niveau 2 ou 3, si une spécification est satisfaite dans un état, afficher un calcul qui mène à un tel état.