Sémantique des langages de programmation
TP $\mathbb{K}$

Semestre 5, 2023–24

Le but de ce TP est d'apprendre à se servir de l'outil $\mathbb{K}$ (http://k-framework.org/), qui permet de spécifier des sémantiques de façon formelle et de pouvoir les manipuler. On essaiera d'écrire la sémantique du langage IMP.

1 Expressions arithmétiques

Dans le fichier http://www.ensiie.fr/~guillaume.burel/download/arith.k (à télécharger), dans le module `ARITH-SYNTAX`, on trouvera la définition de la syntaxe des expressions arithmétiques en $\mathbb{K}$. Nous allons le compléter pour y donner leur sémantique. Quelques remarques à propos de la syntaxe :

— `Int` est l'ensemble des entiers (`0 1 42` ...) ;
— `Id` est l'ensemble des identifiants (`x y my_var` ...) ;
— les opérateurs sont parenthésés à gauche.

1. Nous allons tout d'abord définir la configuration utilisée dans la sémantique. Rappelons qu'il s'agit d'un couple formé de l'expression arithmétique et d'une représentation de l'état mémoire par une valuation qui associe des valeurs entières à des variables.

   La structure des configurations est donnée à l'aide d'une syntaxe à la XML. Dans notre cas, ajoutons au fichier, dans le module `ARITH`, après la ligne `syntax KResult ::= Int`

   ```
   configuration <T>
                   <k> $PGM:K </k>
                   <state> .Map </state>
                 </T>
   ```

   La configuration comporte bien deux parties : `k` contient l'expression dont on cherche la sémantique et `state` contient la valuation courante, représentée par une Map, qui est au départ la Map vide (`.Map`). `$PGM` recevra le programme passé en entrée, il devra être de type `K` qui est un sur-type de `Aexp` .

2. On va maintenant ajouter des règles définissant la sémantique. La technique de $\mathbb{K}$ se rapproche de la sémantique opérationnelle à petits pas, même si elles sont un peu différentes. On définit donc des étapes élémentaires de calcul.

   La règle AS_4 du cours pour l'addition est traduite par la syntaxe suivante :

   ```
   rule N1:Int + N2:Int => N1 +Int N2
   ```

   `+Int` désignant l'addition sur les entiers qui est built-in dans $\mathbb{K}$.

   Ajouter cette règle ainsi que celles pour `-` `*` et `/`. Pour cette dernière, on pourra spécifier une condition (à la fin de la règle) `when N2 =/=Int 0`

3. Nous pouvons déjà évaluer la sémantique de quelques programmes. Pour cela, compiler `arith.k` avec la commande

   ```
   kompile arith.k
   ```

   Écrire ensuite un fichier `exp.arith` contenant `2 + 5`. Puis lancer l'évaluation avec la commande

   ```
   krun exp.arith
   ```

4. Réessayer avec l'expression `2 + 3 + 5`. Que se passe-t-il ?

   Il faut rajouter les étapes de simplification pour pouvoir réduire à l'intérieur des expressions. Pour cela, $\mathbb{K}$ permet l'utilisation de contextes d'évaluation.

   On peut rajouter des définitions de contexte en utilisant la syntaxe suivante :

   ```
   context HOLE + _
   context _:Int + HOLE
   ```

   `HOLE` désigne le trou du contexte, et on force la partie gauche à être un `Int` avant de réduire la partie droite. Ces déclarations de contexte remplacent donc les règles AS_7 et AS_8.

5. Il nous reste la règle AS_1, qui fait intervenir la valuation. La syntaxe est alors la suivante :

   ```
   rule <k> X:Id => N ...</k><state>... X |-> N ...</state>
   ```

   Cela doit être lu : dans `state` j'ai (entre autres `(... ...)`) une association de `X` à `N` donc l'identifiant `X` est réduit en `N`.

2 Expressions booléennes

Dans le module `BEXP-SYNTAX` on trouvera la syntaxe des expressions booléennes, qui utilisent les expressions arithmétiques dont vous venez de définir la sémantique. Dans le module `BEXP` :

1. Ajouter des règles et des définitions de contexte pour définir la sémantique des expressions booléennes, en respectant le comportement coupe-circuit des `and` et `or`.
2. Compiler à l'aide de la commande

   ```
   kompile --main-module BEXP arith.k
   ```

   pour spécifier que le module principal est `BEXP`, alors que c'est le nom du fichier sans préfixe par défaut.
3. Tester avec l'expression booléenne `2 <= 1 and 0 / 0 <= 3 or true`

3 IMP

Dans le module `IMP-SYNTAX` on trouvera la syntaxe des programmes IMP. On notera la présence de `begin end` qui permettent de parenthéser des programmes (ce qui est inutile au niveau de la syntaxe abstraite, mais indispensable au niveau de la syntaxe concrète). Dans ce fichier :

1. Ajouter les deux règles pour l'affectation (suivant que `X` soit déjà défini dans `state` ou pas) :

   ```
   rule <k>X := A:Int => skip ...</k> <state>... X |-> (_ => A) ...</state>

   rule <k>X := A:Int => skip ...</k> <state> Rho:Map (.Map => X |-> A) </state>
     when notBool(X in keys(Rho))
   ```

   Ajouter également une définition de contexte pour l'affectation.
2. Ajouter des règles et des définitions de contexte pour la séquence, le `if then else` et le `begin end`.
3. Pour le `while`, on réduira `while b do c` vers
   `if b then begin c ; while b do c end else skip`.
4. Tester avec les fichiers `simple.imp` et `euclide.imp` disponibles depuis `http://www.ensiie.fr/~guillaume.burel/download/`.