Éléments de correction
Examen réparti de mi-semestre d'ILP

Christian Queinnec

10 novembre 2011

Un corrigé est déjà disponible en ligne. Ce document n'ajoute que quelques remarques plus ou moins générales, suite à la correction des copies.

Présentez vos programmes correctement, non seulement je n'aime pas les programmes mal présentés mais en plus vous vous pénalisez vous-même car les programmes mal présentés sont bien plus souvent faux que ceux bien présentés.

Puisque d'après l'énoncé, il fallait « rendre erroné toute tentative de placer une valeur non entière dans une variable dont le nom commence par une lettre comprise entre i et n (minuscule ou majuscule) », il faut s'intéresser à toutes les possibilités qu'offre ILP3 de placer une valeur quelconque dans une variable quelconque.

On passe donc en revue les classes d'AST d'ILP3 et on trouve que peuvent placer des valeurs (méthodes de type `extend` ou `update` dans `lexenv` ou `common`) : l'affectation, les blocs unaires ou n-aires, l'invocation de fonction et le rattrapage d'exception.

Il fallait donc changer les méthodes `eval` et `compile*` de ces classes donc modifier la fabrique pour que `parse` convertisse le DOM vers l'AST utilisant les nouvelles classes `CEASTassignment`, `CEASTunaryBlock`, `CEASTlocalBlock` et `CEASTtry`. Une nouvelle classe `Process` permet de stipuler la fabrique à utiliser.

Les méthodes `eval` se ressemblent toutes. Dès qu'une valeur a été calculée et que la variable où elle va être placée a un nom débutant par $i..n$, on vérifie que la valeur est bien entière avec `value instanceof BigInteger`. Si ce n'est pas le cas, on signale une exception `EvaluationException` (ou une sous-classe). Si c'est le cas, on procède comme dans la classe parente.

Attention, le test est `value instanceof BigInteger` ! Ce n'est pas `value instanceof CEASTinteger`, ce n'est pas non plus `value.getClass().getName().equals('BigInteger')`, ni `Integer.parseInt(value.toStr`

Vérifier que la valeur à placer est (ou n'est pas) un entier ne peut se faire à *parse-time* ! Les grammaires RNG ne permettent pas de faire de la vérification de types d'un langage comme ILP3. On ne peut pas non plus parler de la valeur d'une expression à *parse-time* (ou à *compile-time* même si certains supposent que pour compiler un programme, il faudrait déjà l'avoir interprété !).

Les tests se déduisent assez simplement des cas précédents. Ce ne sont que des tests négatifs vérifiant que ce qui est interdit l'est bien. On enserre donc le code faux dans un try-catch ILP3. On vérifie donc qu'affecter une variable, la lier dans un bloc ou dans un rattrapeur d'erreur à une valeur incompatible est bien une anomalie signalée.

Attention aussi à ne pas dupliquer les calculs comme ici :

```
Object value = getXXX().eval(lexenv, common);
if ( value instanceof BigInteger ) {
   ... lexenv.update(getVariable(), getXXX().eval(lexenv, common))...
```

Ce n'est pas une recherche d'efficacité, c'est une propriété sémantique qui est violée : le programme `n = (print('cou'),3)` ne doit imprimer `cou` qu'une seule fois.

Une solution originale, pour l'interprétation, est de ne pas modifier les méthodes `eval` mais les environnements d'interprétation afin que `extend`, `update` et les autres vérifient la propriété demandée. Ça ne fonctionne toutefois pas avec la compilation.

Enfin, je n'ai pratiquement trouvé aucun schéma de compilation correct. Le schéma doit être un patron C (donc sans try-catch, sans foreach qui n'existent pas en C). En cas d'erreur, on signale une exception et l'on n'émet surtout pas d'`exit` qui interdirait tout rattrapage d'erreur.

Savoir si une variable a un nom débutant par une lettre entre i et n est une propriété statique : il est donc totalement inutile d'engendrer du code en C pour vérifier cela !

À moins de stocker le nom des variables ILP dans des chaînes de caractères en C, on ne peut, en C, obtenir, à l'exécution, le nom de la variable.

Il y avait deux difficultés : l'une est que `ILP_CheckInteger` est une instruction et non une expression ce qui rend son emploi délicat dans la compilation d'une expression. La seconde difficulté est la vérification dans le try-catch qui nécessite de se placer sous le contrôle du bon rattrapeur d'erreur.