Examen ILP 2ème session (Revision: 1.9)

Christian Queinnec

25 janvier 2007

Conditions générales

Cet examen est formé de deux problèmes assez indépendants, vous pouvez répondre dans l'ordre qui vous plait.

Le barème est fixé à 20 ; la durée de l'épreuve est de 3 heures. Tous les documents sont autorisés et notamment ceux du cours. Les clés USB sont autorisées.

Votre copie sera formée de fichiers textuels que vous laisserez aux endroits spécifiés dans votre espace de travail pour Eclipse. Cet espace de travail pour Eclipse sera obligatoirement nommé `workspace` et devra être un sous-répertoire direct de votre répertoire personnel `HOME`. Seuls les répertoires mentionnés dans les livraisons à effectuer seront ramassés (je n'irai pas, comme pour le premier examen, rechercher des fichiers textuels créés ou modifiés pendant l'examen et placés ailleurs !)

Pour vous aider, un script vous est proposé qui vous installera la dernière distribution d'ILP ainsi que quelques fichiers utiles. Vous le lancerez ainsi :

```
/Infos/lmd/2006/master/ue/ilp-2006oct/E2/install.sh
```

Ce script lancera Eclipse après coup, ne le lancez donc pas avant !

L'examen sera corrigé à la main, il est donc absolument inutile de s'acharner sur un problème de compilation. Il est beaucoup plus important de rendre aisé, voire plaisant, le travail du correcteur et de lui indiquer, par tout moyen à votre convenance, de manière claire, compréhensible et terminologiquement précise, comment vous surmontez cette épreuve. À ce sujet, vos fichiers n'auront que des lignes de moins de 80 caractères[1] et n'utiliseront que le code ASCII, latin-1 (ISO-8859-1) ou latin-9 (ISO-8859-15)[2]. Vous vous attacherez aussi à présenter des programmes correctement agencés et disposés (il y a des outils pour cela sous Eclipse).

Vous pouvez vous aider des machines pour naviguer dans la documentation ou dans le code d'ILP.

1 Introduction

Le langage à étendre est ILP4, le langage étendu sera nommé ilp4gfv. Le paquetage Java correspondant sera donc `fr.upmc.ilp.ilp4gfv`.

Le programme suivant pose, en ILP4, des problèmes de compilation.

```
function deuxfois (x) {
  2 * x
}
function twice (f x) {
  f(f(x))
}

((deuxfois ? deuxfois : deuxfois)(3))    /* Problème */
```

[1] Eclipse affiche une ligne verticale grise pour indiquer la colonne 80 : activez-la (menu Window, Preferences, General, Editors, Text Editors, Show Print Margin) et respectez-la !

[2] Sur les machines de l'ARI, ceci peut se faire avec le menu Window, Preferences, General, workspace, Text File Encoding, ISO-8859-1.

```
let f = deuxfois                          /* Problème */
in f(4)

twice(deuxfois, 5)                        /* Problème */
```

Il y a deux sortes de problèmes :
- Des problèmes de typage au niveau de C lors de la manipulation de fonctions comme valeurs de première classe.
- Des problèmes d'intégration trop « agressive » (comme disent les américains) : dans le programme précédent, la fonction `deuxfois` disparaît puisque nulle part statiquement invoquée (il n'y a aucun nœud `CEASTglobalInvocation` où la fonction invoquée est `deuxfois`).

Cet examen s'intéresse à ces deux points que vous pouvez vous-même explorer plus en détails avec les programmes fournis en `Grammars/Samples/u59*-4gfv.xml`.

2 Meilleure intégration

Question 1

L'intégration, aujourd'hui en ILP4, retire toute fonction non statiquement invoquée (c'est-à-dire qui ne figure pas en position de fonction dans un nœud `CEASTglobalInvocation`). Or, une fonction peut être référencée par son nom pour être ensuite appliquée : il ne faut donc pas supprimer ces fonctions ! Dans l'exemple précédent, il ne faut donc pas supprimer la fonction `deuxfois`.

L'objet de la question est de bâtir une nouvelle analyse, dans une classe nommée `fr.upmc.ilp.ilp4gfv.UsefulGlobalFunctionVisitor` et implantant l'interface `fr.upmc.ilp.ilp4gfv.IUsefulGlobalFunctionVisitor` (ce fichier est fourni, vous pouvez y lire les commentaires qui s'y trouvent concernant le constructeur), déterminant les seules fonctions utiles (invoquées statiquement ou référencées). Le but est que le compilateur C ne produise aucun avertissement signalant une fonction superflue ou une fonction manquante.

Vous détaillerez la stratégie (c'est-à-dire les grandes lignes de votre solution) de votre analyse dans le commentaire global du fichier `fr.upmc.ilp.ilp4gfv.UsefulGlobalFunctionVisitor`. Vous adjoindrez au fichier demandé les fichiers techniques d'accompagnement, vraisemblablement `fr.upmc.ilp.ilp4gfv.CEASTprogram`, `fr.upmc.ilp.ilp4gfv.CEASTParser` et `fr.upmc.ilp.ilp4gfv.Process`.

Livraison

- un fichier Java nommé `fr.upmc.ilp.ilp4gfv.UsefulGlobalFunctionVisitor.java`
- tous les autres fichiers nécessaires placés dans le paquetage `fr.upmc.ilp.ilp4gfv`

Notation sur 10 points

- 10 points

3 Fonctions de première classe

Quelle est la raison profonde des problèmes de typage lors de la compilation du programme donné en exemple ci-dessus ? Pour vous aider dans votre réflexion, il vous est demandé d'implanter une nouvelle primitive, nommée `isFunction`, un prédicat unaire vérifiant si son argument est une fonction définie par l'utilisateur ou pas. Ainsi,

```
function deuxfois (x) {
  2 * x
}
let g = 1
in print(isFunction(deuxfois))    -- imprime true
```

```
print(isFunction(g+1))        -- imprime false
```

Question 2

Écrire une classe `fr.upmc.ilp.ilp4gfv.IsFunctionStuff.java` implantant cette nouvelle primitive. Vous détaillerez la stratégie d'implantation de cette primitive en commentaire du fichier java demandé et notamment comment vous interprétez et compilez cette primitive et ses usages. Vous en profiterez pour expliquer les problèmes de typage et la (ou les) solution(s) que vous préconiseriez. Il n'est pas demandé de programmes associés).

Vous trouverez quelques exemples de programmes invoquant la primitive `isFunction` en `u5996-4gfv.xml` et `u5997-4gfv.xml`.

Livraison

- un fichier Java nommé `fr.upmc.ilp.ilp4gfv.IsFunctionStuff.java`
- tous les autres fichiers nécessaires placés dans le paquetage `fr.upmc.ilp.ilp4gfv`

Notation sur 10 points

- 10 points