Examen de rattrapage d'ILP

durée 2 heures (Revision: 1.6)

Christian Queinnec

08 septembre 2006

Conditions générales

Cet examen est formé d'un unique problème en plusieurs questions auxquelles vous pouvez répondre dans l'ordre qui vous plait.

Le barème est fixé à 20 ; la durée de l'épreuve est de 2 heures. Tous les documents sont autorisés et notamment ceux du cours[1].

Votre copie sera formée de fichiers (ASCII ou ISO-latin 1 ou 15 seulement) dont les lignes contiendront au plus 78 caractères. Vous rassemblerez ces fichiers dans un répertoire nommé `ilp` placé immédiatement dans votre répertoire personnel (ou `HOME`). Le répertoire `ilp` (pas votre `HOME`) sera ramassé automatiquement en fin d'examen par le centre de calcul. Seuls les fichiers mentionnés dans les livraisons à effectuer seront ramassés. Aucun fichier des sources d'ILP n'est à modifier (sauf si demandé).

L'examen sera corrigé à la main, il n'est donc pas utile de s'acharner sur un problème de compilation. Il est beaucoup plus important de rendre aisé, voire plaisant, le travail du correcteur et, notamment, de l'aider, par de judicieux commentaires, à comprendre vos intentions.

Vous pouvez vous aider des machines pour naviguer dans la documentation ou dans le code d'ILP4 (avec Emacs (`etags`) ou Eclipse). Attention, il est peu conseillé que vous lanciez tous Eclipse en même temps : Eclipse n'est pas nécessaire pour se balader dans le code.

1 Vérification statique d'arité

Le but de ce problème est de vérifier au plus vite que les fonctions globales (non primitives) sont correctement invoquées : lorsqu'une fonction globale f est définie (par le programmeur) et possède une arité n alors toute invocation à f doit comporter exactement n arguments. Cette vérification est nommée « vérification d'arité ».

Rappelons que, pour cet examen, le langage de référence est ILP4. On n'oubliera pas la force de persuasion que peut revêtir un croquis (en ASCII-art) bien pensé ou un commentaire pertinent.

Question 1

Après qu'un programme est lu comme un DOM, quelles sont, dans le cas de l'interprétation et de la compilation, les phases, transformations ou passes successivement appliquées sur ce DOM ? Dans quelles classes ou méthodes ou autres entités d'ILP4 est **effectivement** réalisée la vérification d'arité concernant les fonctions primitives ? La vérification est-elle statique ou dynamique ?

Livraison

– un fichier textuel nommé `q1` contenant votre réponse.

Notation sur 4 points

– 4 points

[1] `http://www-master.ufr-info-p6.jussieu.fr/2005/Ext/queinnec/`

Question 2

Dans quelles classes ou méthodes ou autres entités d'ILP4 est **effectivement** réalisée la vérification d'arité concernant les fonctions globales non primitives ? La vérification est-elle statique ou dynamique ?

Livraison

– un fichier textuel nommé `q2` contenant votre réponse.

Notation sur 4 points

– 4 points

Question 3

Que ce soit pour l'interprétation ou la compilation, on souhaite détecter statiquement les invocations à des fonctions globales définies par le programmeur dont le nombre d'arguments n'est pas approprié. Dans quelle passe insériez-vous cette analyse ? Argumentez votre réponse en détaillant notamment la technique que vous comptez mettre en œuvre, les informations dont vous avez besoin et la disponibilité de ces informations.

Livraison

– un fichier textuel nommé `q3` contenant votre réponse.

Notation sur 3 points

– 3 points

Question 4

Si vous aviez à modifier les fichiers d'ILP4, quelles modifications apporteriez-vous pour implanter l'analyse statique de vérification d'arité concernant les fonctions globales ?

Livraison

– un fichier textuel nommé `q4` contenant votre réponse.

Notation sur 4 points

– 4 points

Question 5

Vous ne pouvez modifier les sources d'ILP4, écrivez le code Java implantant l'analyse statique de vérification d'arité concernant les fonctions globales. Les classes que vous aurez à écrire devront appartenir au paquetage `fr.upmc.ilp.ilp4fns`. Pensez à expliquer les nouveaux problèmes qui surgissent et comment vous les résolvez.

Livraison

– des fichiers Java. Ces fichiers Java seront placés dans `ilp/fr/upmc/ilp/ilp4fns/`

Notation sur 5 points

– 5 points

2 Éléments de solution

Voici quelques éléments de solution (`Revision: 1.3`).

2.1 Question 1 : vérification d'arité des primitives

La première sous-question était une pure question de cours. Lorsqu'un programme est lu comme un DOM, les phases communes à l'interprétation et à la compilation sont :

1. normalisation
2. graphe d'appel
3. intégration
4. collecte des variables globales

L'interprétation ajoute une phase d'évaluation. La compilation une phase de génération de code.

2.1.1 Vérification d'arité des invocations aux primitives

La vérification d'arité des invocations aux fonctions primitives a lieu a plusieurs endroits.

- à la normalisation d'une `CEASTprimitiveInvocation`, une méthode `checkArity` pourrait permettre de vérifier ce point mais à deux conditions :
  1. que le code du corps de cette méthode soit présent,
  2. que l'arité de la primitive vérifiée soit présente dans l'environnement global de normalisation.

  À ces deux conditions, la vérification serait statique (il n'est pas nécessaire d'évaluer le programme pour savoir s'il est erroné ou pas).
- À l'interprétation (donc de façon dynamique), une `ilp4.ast.CEASTprimitiveInvocation` hérite de la méthode `eval` de `ilp4.ast.CEASTexpression` qui délègue à la méthode `eval` des `ilp2.ast.CEASTprimitiveInvocation`. Cette méthode évalue les arguments, les réunit en un vecteur et invoque la primitive suivant la méthode `invoke` de l'interface `Invokable`.
  Dans `ilp4.runtime.PrintStuff` la définition des deux primitives `print` et `newline` hérite de la classe abstraite `AbstractInvokable` (implantant `Invokable`). L'invocation avec un vecteur de n arguments renvoie sur la méthode `invoke` à n arguments. Par défaut toutes les méthodes `invoke` d'`AbstractInvokable` signalent une erreur d'arité. La primitive `print` redéfinit la méthode `invoke` unaire qui est donc la seule à ne pas provoquer d'erreur d'arité. Pour la primitive `newline`, seule la méthode `invoke` zéro-aire (sans argument) ne provoque pas d'erreur d'arité.
  Cette vérification est faite à l'exécution : elle est donc dynamique.
- En ce qui concerne la compilation, l'invocation est traduite en C sans vérification. En revanche, le compilateur C vérifiera (à l'aide du fichier `ilp.h` qui contient les prototypes (les signatures) d'`ILP_print` et `ILP_newline`) que les invocations à ces fonctions C ont des arités correctes. La vérification est statique pour C, elle n'est (actuellement) pas statique pour ILP qui n'a rien vérifié.

p

2.2 Question 2 : vérification d'arité des invocations aux fonctions globales

La vérification d'arité des invocations aux fonctions globales a lieu a plusieurs endroits.

- à la normalisation d'une `CEASTglobalInvocation`, une méthode `checkArity` vérifie que le nombre d'arguments est égal au nombre de variables de la fonction. La vérification est donc statique.
- À l'interprétation (donc de façon dynamique), une `ilp4.ast.CEASTglobalInvocation` hérite de la méthode `eval` de `ilp4.ast.CEASTexpression` qui délègue à la méthode `eval` de

`ilp2.CEASTinvocation`. Cette méthode évalue les arguments, les réunit en un vecteur et invoque la fonction avec la méthode `invoke` de l'interface `IUserFunction`.
Les fonctions globales sont implantées avec `UserGlobalFunction` qui vérifie l'arité dans sa méthode `invoke`.
Cette vérification (totalement inutile puisque déjà effectuée) est faite à l'exécution : elle est donc dynamique.

– En ce qui concerne la compilation, l'invocation est traduite en C sans vérification (puisque cela a été vérifié auparavant). En revanche, le compilateur C vérifiera à son tour que la fonction est utilisée avec une arité correcte. Cette vérification (encore une fois totalement inutile) est statique pour C.

2.3 Question 3 et 4 : analyse et code

Le code est déjà présent dans ILP4 (il y a été introduit suite à cet examen). Auparavant la méthode `checkArity` était présente mais non invoquée.

Cette passe a lieu lors de la normalisation mais pourrait faire l'objet d'une analyse statique séparée postérieure (à l'aide par exemple d'un visiteur).

2.4 Question 5 : sans modification

Si le code précédent (ajourd'hui intégré à ILP4) n'était pas présent et devait être défini dans un paquetage séparé (`ilp4fns`), il faudrait :

1. Écrire une classe `ilp4fns.CEASTParser` qui transformerait les invocations à des fonctions globales en des
2. `ilp4fns.CEASTglobalInvocation` : classe héritant de `ilp4.CEASTglobalInvocation` et dotée du bon comportement
3. c'est-à-dire d'une méthode de normalisation qui vérifie l'arité.
4. Il faudrait également que la classe `ilp4fns.CEASTParser` transforme les définitions de fonctions en des
5. `ilp4fns.CEASTfunctionDefinition` : classe héritant de `ilp4.CEASTfunctionDefinition` dont la nouvelle méthode de normalisation stocke, dans l'environnement global de normalisation, l'arité de la fonction définie afin que cette arité puisse être retrouvée par les invocations de `ilp4fns.CEASTglobalInvocation`.