Examen final d'ILP
1ère session

Christian Queinnec

19 décembre 2008

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 3 heures. Tous les documents sont autorisés et notamment ceux du cours.

Votre copie sera formée de fichiers textuels que vous laisserez aux endroits spécifiés dans votre espace de travail pour Eclipse. L'espace de travail pour Eclipse sera obligatoirement nommé `workspace` et devra être un sous-répertoire direct de votre répertoire personnel.

À l'exception des clés USB en lecture seule, tous les appareils électroniques sont prohibés (y compris les téléphones portables, les assistants numériques personnels et les agendas électroniques).

L'examen sera corrigé à la main, il est donc absolument inutile de s'acharner sur un problème de compilation ou sur des méthodes à contenu informatif faible. 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, n'utiliseront que le codage ASCII ou UTF-8 enfin, s'abstiendront de tout caractère de tabulation.

Le langage à étendre est ILP4. Le paquetage Java correspondant à cet examen sera donc nommé `fr.upmc.ilp.ilp4array`. Sera ramassé, à partir de votre *workspace*, tout répertoire ou fichier ayant le fragment `4array` dans son nom.

1 Introduction de tableaux

On souhaite étendre ILP4 avec des tableaux unidimensionnels (des vecteurs) statiquement nommés. Le programme (inepte) suivant illustre les opérations (déclarations ou expressions) possibles sur les tableaux :

```
dimension tab[40]             // définition du tableau tab
dimension vec[1]              // définition du tableau vec
function foo (i, j) {
  i + vec[0]*tab[j]           // lecture
}
let k = 0
  while ( k < tab'size ) {    // taille
     tab[k+1] = foo(k, k-1)   // écriture
     k = k+1
  }
```

Les tableaux doivent être définis (ici avec une notation à la Fortran) avant les fonctions, avec une taille qui ne peut être qu'une constante. Les positions dans le tableau peuvent être lues ou écrites, elles sont indexées à partir de zéro (comme en C). La taille du tableau peut être obtenue (ici avec une notation à la Ada). Cette extension doit être sûre : toute lecture ou écriture en dehors du tableau signalera une exception.

Question 1 – Grammaire (2 points)

Étendre la grammaire d'ILP4 pour inclure les nouvelles déclarations et expressions autour des tableaux. Vous pouvez abuser de commentaires pour y insérer les points les plus pertinents de vos réflexions.

Livraison

– un fichier `grammar4array.rnc` placé dans le répertoire `Grammars`.

[2/20]

```
# -*- coding: utf-8 -*-
# Ajout de vecteurs statiques alloues. On etend ILP4 pour ne pas s'encombrer
# avec ILP6 (mais cela aurait pu se faire).

include "grammar4.rnc"
start |= programme4array

# On ajoute quelques expressions permettant de manipuler des tableaux

expression |=
   lectureTableau
 | ecritureTableau
 | tailleTableau

# Un programme peut aussi contenir des definitions de classes en tete.

programme4array = element programme4array {
   definitionTableau +,
   definitionFonction *,
   expression +
}

definitionTableau = element definitionTableau {
   attribute nom { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   attribute taille { xsd:nonNegativeInteger }
}

lectureTableau = element lectureTableau {
   attribute nom { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   element index { expression }
}

ecritureTableau = element ecritureTableau {
   attribute nom { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   element index { expression },
   element valeur { expression }
}

tailleTableau = element tailleTableau {
   attribute nom { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) }
}

# fin de grammar4array.rnc
```

Question 2 – Programme (2 points)

Écrire un programme en syntaxe XML conforme à la grammaire précédente et utilisant toutes ces nouvelles extensions.

Livraison

– un fichier `a-4array.xml` placé dans le répertoire `Grammars/Samples`.

Question 3 – Interprétation (5 points)

Écrire les méthodes d'évaluation des expressions ajoutées. Vous préciserez en commentaire dans ces classes comment vous représentez les tableaux, où vous les créez et où vous les stockez.

Livraison

– les fichiers correspondant à ces classes placés dans le paquetage `fr.upmc.ilp.ilp4array`.

[5/20]
On créera les classes dérivées de la grammaire c'est-à-dire : CEASTarrayDefinition(String nom, int taille), CEASTarrayRead(String nom, IAST4expression index), CEASTarrayWrite(String nom, IAST4expression index, IAST4expression value), CEASTarraySize(String nom). On ajoutera à l'analyseur syntaxique la transformation des nuds XML correspondant en ces classes (on peut aussi créer les interfaces correspondantes).
L'évaluation d'un programme passe par une première phase où l'on alloue dans l'environnement global d'évaluation, les tableaux qui sont définis en tête du programme. Pour ce faire, on peut sous-classer l'environnement global d'évaluation pour y ajouter un champ `arrays` menant vers une table associative associant à chaque nom de tableau un tableau unidimensionnel d'objets Java (la taille de ce tableau est un entier contenu dans le nœud `CEASTarrayDefinition`) contenant des valeurs ILP c'est-à-dire des objets légaux au sens d'ILP : on prendra par exemple la constante booléenne `Boolean.FALSE`. Puisque les tableaux `Object[]` existent en Java, il n'est pas nécessaire d'inventer une classe pour eux dans la bibliothèque d'exécution d'ILP.
On remarquera qu'il n'est pas dit ce que vaut une case non initialisée d'un tableau. On pourrait soit signaler une erreur, soit rendre le contenu (qui existe au sens de Java) de cette case. Par simplicité, nous avons choisi cette dernière solution.
L'évaluation du programme se poursuit par la définition des fonctions globales puis l'évaluation du corps du programme.
L'obtention de la taille d'un tableau s'effectue simplement. Le nom du tableau (qui figure dans le nœud `tailleTableau`) est recherché dans l'environnement global d'évaluation (qui est visible de partout lors de l'évaluation). Le tableau Java associé est recherché dans la table associative `arrays` stockée dans cet environnement global d'évaluation. Enfin, la taille du tableau Java est retournée en résultat convenablement transformée en un `BigInteger`.
La lecture d'un tableau est un nœud particulier contenant le nom d'un tableau et une expression qu'il faut évaluer pour obtenir l'index (un entier naturel). Il faut vérifier que le nom du tableau figure bien dans la table des tableaux (cette vérification est statique), que l'index a pour valeur (vérification dynamique) un entier positif ou nul compatible avec la taille du tableau Java.
Pour l'écriture d'un tableau, il faut en plus évaluer l'expression à stocker dans le tableau mais il n'y a pas de contrainte sur cette valeur puisqu'ILP est un langage non typé statiquement.

Question 4 – Suppression de tableaux inutiles (3 points)

Lorsqu'un tableau est déclaré mais qu'il n'est pas utilisé c'est-à-dire qu'il n'est ni lu ni écrit (mais que peut-être sa taille est consultée), on peut transformer le programme en un nouveau programme débarrassé de ce tableau. Vous préciserez, en commentaire de la classe `DeadArrayRemoval`, plus finement cette transformation (et notamment les schémas de transformations tels qu'utilisés dans le cours), quand elle doit être effectuée ; vous indiquerez aussi comment cette transformation pourrait être implantée.

Livraison

– une unique classe `DeadArrayRemoval` placé dans le paquetage `fr.upmc.ilp.ilp4array`.

[3/20]

Pour réaliser cette analyse, il faut

1. préparer l'analyse statique (disons `removeDeadArrays`) qui peut s'insérer juste après la normalisation. Pour cette analyse, l'environnement lexical d'analyse est sans utilité car on ne s'occupe que des noms globaux des tableaux. En revanche, il faut un environnement global d'analyse associant au nom des tableaux, leur taille et un booléen initialement faux indiquant si le tableau est utilisé dans le programme.
2. Arpenter l'AST entier à la recherche des nœuds de type `definitionTableau`, `lectureTableau`, `ecritureTableau` et `tailleTableau`.
   - Un nœud de type `definitionTableau` enrichit l'environnement global d'analyse en ajoutant le tableau défini, sa taille et le booléen faux. Les définitions de tableau doivent être passées en revue avant les fonctions.
   - Un nœud de type `tailleTableau` sera remplacé par la constante entière (`IAST2integer`) formée par la taille du tableau.
   - un nœud de type `lectureTableau` ou `ecritureTableau` vérifie que le tableau figure bien dans l'environnement global d'analyse et bascule le booléen à Vrai pour indiquer que le tableau est utile.
3. Lorsque tout l'AST a été parcouru, supprimer tous les tableaux n'ayant aucune utilité (cette utilité est indiquée par le booléen associé dans l'environnement global d'analyse). Comme les tailles ont déjà été remplacées par une expression équivalente, il ne reste plus aucune référence aux tableaux qui peuvent ainsi disparaître. Le plus simple est alors de supprimer de l'AST du programme entier les définitions de tableaux superflues.

Un visiteur générique peut être utilisé pour cette analyse.

Question 5 – Suppression de petits tableaux (2 points)

Lorsqu'un tableau est déclaré avec une taille de 1, une variable globale peut aisément le remplacer. Vous préciserez, en commentaire de la classe `SmallArrayRemoval`, plus finement cette transformation (et notamment les schémas de transformations tels qu'utilisés dans le cours), quand elle doit être effectuée ; vous indiquerez aussi comment cette transformation pourrait être implantée.

Livraison

- une unique classe `SmallArrayRemoval` placé dans le paquetage `fr.upmc.ilp.ilp4array`.

[2/20]

Si un tableau (disons *A*) a une taille de 1, alors *A'taille* peut être remplacé par `1` (comme le fait la précédente analyse) et *A[0]* peut être partout remplacé par une variable globale avec un nom ne créant pas de conflit (disons *gvA*). L'expression *A[0]* peut être remplacée par *gvA* à la fois à gauche ou à droite du signe d'affectation.
Toutefois la difficulté est que l'expression *A[0]* est probablement rare et que l'on observe plutôt *A[expression]*. L'analyse est donc une tranformation de programme où l'on transforme les lectures/écritures du tableau par les expressions ILP équivalentes suivantes (en syntaxe ML-ienne) :

```
A[expression]            // lectureTableau

          let tmp = expression          // blocUnaire
          in if tmp == 0
             then gvA
             else throw BadArrayException

A[expression] = val      // ecritureTableau

          let tmp = expression          // blocUnaire
          in if tmp == 0
             then gvA = val
             else throw BadArrayException
```

Un visiteur générique peut être utilisé pour cette transformation.

Question 6 – Compilation (6 points)

Écrire les schémas de compilation (tels qu'utilisés dans le cours) correspondant aux expressions ajoutées et au nouveau patron de génération du fichier C. Vous les préciserez dans les commentaires des classes concernées.

Livraison

- les fichiers correspondant à ces classes placés dans le paquetage `fr.upmc.ilp.ilp4array`.

Voici les schémas de compilation :

```
-----------------------------> d      // d ici ne peut etre qu'en place.
definitionTableau(nom, taille)

      #define TAILLE taille+1
      static ILP_Object nom[TAILLE] = {    // Au 'mangling' pres
         ILP_FALSE, /* indice 0 */
         ILP_FALSE, /* indice 1 */
         ...
         ILP_FALSE  /* indice taille-1  */
         ILP_FALSE  /* finale */
      };
// Toutes les definitions des tableaux doivent apparaitre avant les
// definitions de fonctions. Les tableaux peuvent etre declares avant
// ou apres les variables globales. Les noms des tableaux doivent etre
// adaptes au jeu de caracteres de C (mangling).
// Au fait, C77 ne permet pas de declarer un tableau de taille nulle
// d'ou l'ajout systematique d'une case ce que represente le pseudo
// #define initial.

-----------------> d
tailleTableau(nom)

      d taille                   // On trouve la taille associee au tableau
                                 // nom dans l'environnement global de
                                 // compilation. C'est aussi valable
                                 // pour les 2 schemas qui suivent:

------------------------> d
lectureTableau(nom, exp)


      { ILP_Object tmp;          // Au 'mangling' pres
        ----> tmp
        exp;
        if ( 0 <= tmp && tmp < taille ) {
           d nom[tmp];
        } else {
           ILP_arrayBadIndex("nom", taille, tmp);
        }
      }
// On ajoute a la bibliotheque d'execution (runtime) la fonction (ou
// macro) ILP_arrayBadIndex signalant une exception lorsque l'index
// sort des bornes du tableau.

-------------------------> d
ecritureTableau(nom, exp1, exp2)


      { ILP_Object tmp;          // Au 'mangling' pres
        ----> tmp
        exp1;
        if ( 0 <= tmp && tmp < taille ) {
           ILP_Object val;      // Au 'mangling' pres
           -----> val
           exp2;
           d (nom[tmp] = val);
        } else {
           ILP_arrayBadIndex("nom", taille, tmp);
        }
      }
// Ici le choix a ete fait de verifier l'index avant de calculer la
// valeur a stocker dans le tableau.
```

Le patron général en C ressemble est le même que celui d'ILP3.