Examen ILP 1ère session (Revision: 1.17)

Christian Queinnec

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

Pour des raisons de compatibilité de calendriers d'examens, vous n'êtes pas autorisés à sortir de cette épreuve avant 15h45.

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.

Pour vous aider, un script vous est proposé qui vous installera quelques fichiers utiles. Vous le lancerez ainsi :

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

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 et n'utiliseront que le code ASCII, latin-1 (ISO-8859-1) ou latin-9 (ISO-8859-15).

Vous pouvez vous aider des machines pour naviguer dans la documentation ou dans le code d'ILP (avec Emacs (`etags`) ou Eclipse).

Le langage à étendre est ILP4, le langage étendu sera nommé ilp4tr. Le paquetage Java correspondant sera donc `fr.upmc.ilp.ilp4tr`.

1 Trace

L'examen tout entier vise à tracer des fonctions c'est-à-dire faire en sorte d'imprimer comment elles sont invoquées et quelle valeur elles renvoient afin notamment de suivre à la trace ce qui se passe dans un programme. Voici un exemple où le programme est à gauche et où les impressions produites sont à droite.

```
function f (x, y) {
  return x * y
}
function g (x) {
  let z = f(x, x)
  return f(z, x)
}
trace f
trace print
let un = 1
    deux = 2
in print(un + g(deux))
```

```
-> f 2 2
<- f 4
-> f 4 2
<- f 8
-> print 9
<- print false
```

Les traces devraient, bien sûr, être produites sur le flux standard d'erreur. Néanmoins il sera possible, pour simplifier, d'utiliser aussi le flux standard de sortie.

Question 1

Une fonction globale (prédéfinie ou définie par le programmeur) nommée f sera tracée suite à une déclaration de trace `trace f`. Écrire une grammaire RelaxNG pour `ilp4tr`. La nouvelle forme syntaxique sera nommée `traceDeclaration`. Les déclarations de trace ne peuvent apparaître qu'après les définitions et avant les instructions.

Livraison

– un fichier `grammar4tr.rnc` placé dans le répertoire `Grammars`.

Notation sur 2 points

– 2 points

Question 2

La propriété « la fonction globale f est-elle tracée ? » est-elle une propriété statiquement décidable ou pas ? Justifiez votre réponse.

Livraison

– un fichier textuel nommé `q2.txt` dans le répertoire `Java/src/fr/upmc/ilp/ilp4tr/`.

Notation sur 2 points

– 2 points

Question 3

La proposition « tracer $f1$ » est ambiguë ainsi que le montre le programme suivant. Deux traces possibles figurent à droite. Dans l'une, on trace les appels à la valeur de la variable globale `f1`, dans l'autre, on trace les appels à la fonction globale nommée `f1`.

Dans le programme qui suit, on ne s'interrogera pas sur la stricte légalité, en ILP4, de l'affectation `f1 = f2`. Il s'agit juste de changer la valeur de la variable globale `f1` pour devenir la fonction définie sous le nom de `f2`.

```
function f1 (x) {
   return 11 * x
}
function f2 (y) {
   return 5 * y
}
trace f1
f1(11)
f1 = f2
f1(22)
```

```
-> f1 11
<- f1 121
-> f2 22
<- f2 110
```

ou

```
-> f1 11
<- f1 121
```

Vous avez la possibilité de choisir l'une ou l'autre des interprétations. Indiquez votre choix (ainsi qu'éventuellement les raisons) dans le fichier `q3.txt`. Vous avez bien sûr intérêt à choisir le plus simple à implanter.

Livraison

- un fichier textuel nommé `q3.txt` dans le répertoire `Java/src/fr/upmc/ilp/ilp4tr/`

Notation sur 1 point

- 1 point
- **Bonus:** 1 point Si jamais vous imaginez encore une autre sémantique possible, indiquez-la dans ce même fichier.

Question 4

Expliquez les grandes lignes de la stratégie d'implantation que vous allez suivre dans les questions qui suivent. Rappelez-vous qu'une explication textuelle bien écrite vaut mieux qu'un code approximatif. Souvenez-vous également que des schémas bien faits peuvent aussi aider.

Cette question est la plus importante de l'examen ! L'idéal est, qu'après cette question, la suite de cet examen ne soit plus que la simple mise en œuvre de ce que vous avez dit : on sait alors quelles sont les classes à écrire, ce qu'elles doivent faire, comment elles s'intègrent techniquement à ILP4, etc.

Livraison

- un fichier textuel nommé `q4.txt` dans le répertoire `Java/src/fr/upmc/ilp/ilp4tr/`.

Notation sur 5 points

- 5 points

Question 5

Écrire la classe `fr.upmc.ilp.ilp4tr.CEASTtraceDeclaration` et sa méthode `parse`. On n'oubliera pas la force de persuasion que peut revêtir un commentaire pertinent.

Livraison

- un fichier Java nommé `CEASTtraceDeclaration.java` dans le répertoire approprié sous `Java/src`.

Notation sur 1 point

- 1 point

Question 6

Compléter la précédente classe `fr.upmc.ilp.ilp4tr.CEASTtraceDeclaration` avec une méthode `normalize`. Attention en modifiant cette classe de ne pas détruire le comportement obtenu à la question précédente !

Livraison

- un fichier Java nommé `CEASTtraceDeclaration.java` dans le répertoire approprié sous `Java/src`.

Notation sur 2 points

- 2 points

Question 7

Compléter la précédente classe `fr.upmc.ilp.ilp4tr.CEASTtraceDeclaration` avec une méthode `compile` implantant la compilation vers C. Attention en modifiant cette classe de ne pas détruire le comportement obtenu à la question précédente ! On n'oubliera pas la force de persuasion que peut revêtir un croquis bien pensé ou un commentaire pertinent. Il sera notamment plus qu'utile d'indiquer la forme générale du résultat de compilation avant de se lancer dans le détail du code C.

D'autres classes peuvent intervenir ! Elles doivent appartenir au même paquetage.

Pour vous aider dans cette tâche, vous utiliserez les fonctions utilitaires (en `C/ilpTrace.c`) `ILP_beforeInvocation` et `ILP_afterInvocation`.

Livraison

- un fichier Java nommé `CEASTtraceDeclaration.java` ainsi que les éventuels autres fichiers Java dans le répertoire approprié sous `Java/src`.

Notation sur 3 points

- 3 points

Question 8

Compléter la précédente classe `fr.upmc.ilp.ilp4tr.CEASTtraceDeclaration` et la méthode d'interprétation associée. On n'oubliera pas la force de persuasion que peut revêtir un croquis bien pensé ou un commentaire pertinent.

D'autres classes peuvent intervenir ! Elles doivent appartenir au même paquetage.

Livraison

- un fichier Java nommé `CEASTtraceDeclaration.java` ainsi que les éventuels autres fichiers Java dans le répertoire approprié sous `Java/src`.

Notation sur 3 points

- 3 points

Question 9

Ceci est une question subsidiaire destinée à départager les éventuels ex-aequo.

On souhaite ne plus mettre en séquence les définitions de fonctions et les déclarations de trace mais pouvoir les mélanger sans restriction. En particulier, une déclaration de trace de f peut apparaître avant la définition de f. Définissez une nouvelle grammaire pour ce faire et commentez (dans le fichier `q9.txt`) comment vous implanteriez cette nouvelle caractéristique.

Livraison

- une grammaire nommée `grammar4trc.rnc` placé dans le répertoire `Grammars`.
- un fichier textuel nommé `q9.txt` dans le répertoire `Java/src/fr/upmc/ilp/ilp4tr/`.

Notation sur 1 point

- 1 point

2 Éléments de solution

Voici quelques éléments de solution (`Revision: 1.5`).

2.1 Question 1 : grammaire

Voici la grammaire. À noter toutefois que la déclaration de trace n'est pas une instruction. Si c'était une instruction, elle pourrait apparaître partout où une instruction est possible ce qui n'est pas le cas.

```
# $Id: grammar4tr.rnc 573 2006-12-28 17:15:40Z queinnec $
# On ajoute "trace␣nomFonction" a la grammaire d'ILP4.

include "grammar4.rnc"

start |= programme4tr

programme4tr = element programme4tr {
   definitionFonction *,
   traceDeclaration *,
   expression +
}

# Tracer la fonction 'nom':
traceDeclaration = element traceDeclaration {
   attribute nom { xsd:Name }
}

# fin de grammar4tr.rnc
```

2.2 Question 2 : Qualité

La propriété « la fonction globale f est-elle tracée ? » est statiquement décidable (ou pour faire plus court : statique). Pour savoir, dès la lecture du programme, si une fonction est tracée, il suffit de chercher si la déclaration `trace $f$` est présente. Il n'est pas besoin d'exécuter le programme pour cela, ce n'est donc point dynamique. À noter que savoir statiquement si une fonction est tracée ne nécessite pas de connaître la valeur des arguments lors des appels à cette fonction à l'exécution !

La propriété « la fonction globale f est-elle invoquée ? », quant à elle, n'est, en général, pas statiquement décidable (même si souvent il est possible de répondre) mais ce n'était pas la question posée. Ce n'est pas statiquement décidable car il faudrait connaître l'exécution toute entière pour connaître la réponse. Cependant, dans l'état de la bibliothèque prédéfinie qui ne contient pas de fonction acquérant dynamiquement de l'information (pas de fonction `read`), les programmes ne dépendent pas d'information externe à leur code et donc pourraient être calculés dès la compilation !

2.3 Question 3 : Sémantique

Il y a aussi d'autres interprétations différentes à côté des variantes A et B qu'introduisait l'énoncé.

Tracer tous les appels à `f1` pourrait aussi produire la trace suivante (variante C) :

```
-> f1 11
<- f1 121
-> f1 22
<- f1 110
```

sans référence à `f2` puisque c'est `f1` que l'on trace. Enfin, on pourrait imaginer aussi (variante D) que le programme suivant à gauche trace aussi les fonctions nommées par des variables locales :

```
function f (x) {                -> g 3
  return 2 * x                  <- g 6
}
trace g
let g = f
in g(3)
```

La sémantique que nous choisirons est celle correspondant à la variante C : seront tracées toutes les invocations où la fonction est obtenue par la valeur d'une variable globale mentionnée dans une déclaration de trace. Ainsi,

```
function f1 (x) {               -> f1 11
   return 11 * x                <- f1 121
}                               -> f1 22
function f2 (y) {               <- f1 110
   return 5 * y
}
trace f1
f1(11)      // tracée
savef1 = f1
savef1(33)  // non tracée
f1 = f2
f1(22)      // tracée
let f1 = savef1
in f1(44)   // non tracée
```

Naturellement, avec cette sémantique, il faut modifier la sémantique des invocations afin de prendre en compte le cas où l'expression en position fonctionnelle est une variable globale qu'il faut tracer ou pas.

Une autre sémantique aurait été de modifier le corps des fonctions tracées par une transformation de programme qui aurait ajouté les instructions d'impression des arguments et du résultat. Cette sémantique ne correspond pas à la sémantique retenue (variante C) car le corps d'une fonction ne sait pas déterminer par quelle expression elle a été invoquée (par une variable globale ou locale ou encore par une expression plus complexe).

Autre sujet de réflexion : avez-vous noté dans le premier exemple donné en tête de l'examen que les fonctions prédéfinies telles que `print` ou `newline` pouvaient être tracées ? Et l'on ne peut instrumenter leur corps pour les tracer puisque, prédéfinies, elles n'ont pas de corps !

2.4 Question 4 : stratégie

C'était la question la plus importante de l'examen !

La sémantique retenue consiste à modifier l'évaluation (interprétation, compilation) des invocations globales, c'est-à-dire les invocations où la fonction est obtenue par la valeur d'une variable globale (nœud syntaxique `CEASTglobalInvocation`). Pour tracer les invocations à des primitives, les mêmes traitements pourront être appliqués aux nœuds syntaxiques `CEASTprimitiveInvocation`.

Tracer une invocation consiste à imprimer le nom de la variable globale suivi des arguments d'appel, à effectuer l'invocation proprement dite (comme dans ilp4) puis à récupérer le résultat et à l'imprimer précédé du nom de la variable globale. En d'autres termes, cela revient (sur l'exemple suivant) à transformer l'invocation à gauche en l'expression à droite (les variables locales `v1`, `v2` et `r` sont *fraîches* c'est-à-dire non susceptibles de conflits ou captures, la substitution est aussi considérée comme *hygiénique* c'est-à-dire que `print` est bien la fonction d'impression et non une autre valeur qu'un `let` emboîtant aurait lié). L'usage de variables fraîches permet de n'évaluer qu'une seule fois les arguments de la fonction.

```
f(expr1, expr2)                 let v1 = expr1
                                    v2 = expr2
                                in
                                  print "->␣f" v1 v2
                                  let r = f v1 v2
                                  in
                                     print "<-␣f" r
                                     r
```

Voici une des nombreuses solutions possibles.

Lors de l'analyse syntaxique (`CEASTParser`, toutes les invocations seront associées à un booléen indiquant s'il faudra tracer l'invocation ou pas. Comme on ne peut ajouter un champ booléen aux invocations d'ILP4 (`ilp4.CEASTinvocation`), on créera une nouveau nœud `ilp4tr.CEASTinvocation` qui aura pour délégué l'invocation ILP4 (ou plutôt une de ses sous-classes plus précise obtenue après normalisation).

Le traitement d'une déclaration consiste à mémoriser quelles sont les variables globales dont l'invocation sera à tracer. Cet ensemble de variables sera stocké dans le nœud syntaxique `CEASTprogram` puisqu'il sera obtenu lors de l'analyse des sous-nœuds qui peuvent être des définitions de fonctions suivies des déclarations de trace suivies d'expressions.

Enfin, une nouvelle analyse syntaxique sera ajoutée à `Process` pour, à partir de l'ensemble des variables globales à tracer, parcourir le programme tout entier et marquer (en basculant le booléen associé) toutes les invocations à tracer. À ce propos, la nouvelle analyse devra être effectuée avant l'intégration, celle-ci ayant tendance à faire disparaître les invocations ! Au passage, identifier les invocations à tracer permet, par exemple et si on le souhaitait, de ne tracer que les appels à une fonction dans une certaine portée. La déclaration actuelle est une demande globale de trace, on pourrait imaginer que la déclaration puisse être insérée au début d'un bloc pour ne tracer que les appels présents dans ce bloc.

Donc, pour récapituler :

1. Une nouvelle classe `ilp4tr.CEASTprogram` avec un champ supplémentaire déterminant l'ensemble des variables globales à tracer et une méthode `parse` initialisant cet ensemble.
2. une nouvelle classe `ilp4tr.CEASTinvocation` avec un champ booléen indiquant si l'invocation doit être tracée ou pas. Ce champ sera associé à des accesseurs en lecture et écriture pour lire ou basculer ce booléen. Hormis ce champ additionnel, tout sera délégué vers une `ilp4.IAST4invocation`.
3. une classe `ilp4tr.CEASTtraceDeclaration` pour représenter syntaxiquement une déclaration de trace. La seule méthode intéressante est celle de normalisation qui doit assurer l'unicité de la représentation des variables globales.
4. une classe `ilp4tr.InvocationMarker` implantant la nouvelle analyse statique (un visiteur) qui parcourt l'AST et positionne à vrai le booléen de toute invocation globale dont la variable globale appartient à la liste des variables à tracer trouvée dans `ilp4tr.CEASTprogram`.

2.5 Question 5 : analyse syntaxique

Voici la classe `ilp4tr.CEASTtraceDeclaration` :

```
/* ****************************************************************
 * ILP -- Implantation d'un langage de programmation.
 * Copyright (C) 2006 <Christian.Queinnec@lip6.fr>
 * $Id: CEASTtraceDeclaration.java 572 2006-12-24 17:29:33Z queinnec $
 * GPL version>=2
 * ****************************************************************/

package fr.upmc.ilp.ilp4tr;
```

```
import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.ast.NormalizeException;
import fr.upmc.ilp.ilp4.interfaces.IAST4;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.tool.IParser;

public class CEASTtraceDeclaration
extends fr.upmc.ilp.ilp4.ast.CEAST
implements IAST4 {

  public CEASTtraceDeclaration (final IAST4variable variable) {
    this.variable = variable;
  }
  // ce peut etre une globalFunctionVariable ou une predefinedVariable,
  // voire meme une localVariable !
  private IAST4variable variable;

  public IAST4variable getVariable () {
      return this.variable;
  }

  public IAST4 getDelegate () {
      throw new RuntimeException("Needless!");
  }

  public static CEASTtraceDeclaration parse (Element e, IParser parser)
  throws CEASTparseException  {
    final IAST4variable variable =
        fr.upmc.ilp.ilp4.ast.CEASTvariable.parse(e, parser);
    return new CEASTtraceDeclaration(variable);
  }

  @Override
  public CEASTtraceDeclaration normalize (
          final INormalizeLexicalEnvironment lexenv,
          final INormalizeGlobalEnvironment common )
  throws NormalizeException {
    final IAST4variable variable =
        fr.upmc.ilp.ilp4.ast.CEAST.narrowToIAST4variable(
                this.variable.normalize(lexenv, common) );
    return new CEASTtraceDeclaration(variable);
  }

  public void accept (IAST4visitor visitor) {
          // Rien a faire.
  }

  public Object eval (ILexicalEnvironment lexenv, ICommon common)
  throws EvaluationException {
      throw new EvaluationException("Not␣evaluable␣since␣static!");
```

```
  }

}

// end of CEASTtraceDeclaration.java
```

et l'analyseur syntaxique tout entier :

```
/* ****************************************************************
 * ILP -- Implantation d'un langage de programmation.
 * Copyright (C) 2006 <Christian.Queinnec@lip6.fr>
 * $Id: CEASTParser.java 572 2006-12-24 17:29:33Z queinnec $
 * GPL version>=2
 * ****************************************************************/

package fr.upmc.ilp.ilp4tr;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;

public class CEASTParser
extends fr.upmc.ilp.ilp4.ast.CEASTParser {

  public CEASTParser ()
  throws CEASTparseException {
    super();
    addParser("programme1",         CEASTprogram.class);
    addParser("programme2",         CEASTprogram.class);
    addParser("programme3",         CEASTprogram.class);
    addParser("programme4",         CEASTprogram.class);
    addParser("programme4tr",       CEASTprogram.class);
    // l'invocation d'ilp4tr et non celle d'ilp4.ast:
    addParser("invocation",         CEASTinvocation.class);
    addParser("traceDeclaration",   CEASTtraceDeclaration.class);
  }

}

// end of CEASTParser.java
```

2.6 Question 6 : normalisation

Cf. ci-dessus. Voici, au passage, la classe `ilp4tr.CEASTprogram.java`.

```
/* ****************************************************************
 * ILP -- Implantation d'un langage de programmation.
 * Copyright (C) 2006 <Christian.Queinnec@lip6.fr>
 * $Id: CEASTprogram.java 572 2006-12-24 17:29:33Z queinnec $
 * GPL version>=2
 * ****************************************************************/

package fr.upmc.ilp.ilp4tr;

import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.Vector;

import org.w3c.dom.Element;
```

```
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2;
import fr.upmc.ilp.ilp4.ast.CEAST;
import fr.upmc.ilp.ilp4.ast.CEASTexpression;
import fr.upmc.ilp.ilp4.ast.CEASTfunctionDefinition;
import fr.upmc.ilp.ilp4.ast.CEASTglobalFunctionVariable;
import fr.upmc.ilp.ilp4.ast.CEASTglobalInvocation;
import fr.upmc.ilp.ilp4.ast.CEASTsequence;
import fr.upmc.ilp.ilp4.ast.NormalizeException;
import fr.upmc.ilp.ilp4.interfaces.IAST4;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4functionDefinition;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.tool.IParser;

public class CEASTprogram
extends fr.upmc.ilp.ilp4.ast.CEASTprogram {

  public CEASTprogram (IAST4functionDefinition[] definitions,
                       IAST4expression body,
                       CEASTtraceDeclaration[] traces ) {
    super(definitions, body);
    this.traces = traces;
  }
  private CEASTtraceDeclaration[] traces;

  /** Le constructeur analysant syntaxiquement un DOM. */

  public static CEASTprogram parse (final Element e, final IParser parser)
          throws CEASTparseException {
      List<IAST2> itemsAsList = parser.parseList(e.getChildNodes());
      IAST4[] items = itemsAsList.toArray(new IAST4[0]);
      final List<IAST4functionDefinition> definitions =
          new Vector<IAST4functionDefinition>();
      final List<CEASTtraceDeclaration> traces =
          new Vector<CEASTtraceDeclaration>();
      final List<IAST4expression> instructions =
          new Vector<IAST4expression>();
      for ( IAST4 item : items ) {
          if ( item instanceof IAST4functionDefinition ) {
              definitions.add((IAST4functionDefinition) item);
          } else if ( item instanceof CEASTtraceDeclaration ) {
              // Accumuler les declarations de trace:
              traces.add((CEASTtraceDeclaration) item);
          } else if ( item instanceof IAST4expression ) {
              instructions.add((IAST4expression) item);
          } else {
              final String msg = "Should␣never␣occur!";
              assert false : msg;
              throw new CEASTparseException(msg);
          }
      }
      IAST4functionDefinition[] defs =
          definitions.toArray(new IAST4functionDefinition[0]);
      CEASTtraceDeclaration[] trs =
```

```
        traces.toArray(new CEASTtraceDeclaration[0]);
    IAST4expression instrs =
        new CEASTsequence(
                instructions
                    .toArray(CEASTexpression.EMPTY_EXPRESSION_ARRAY) );
    return new CEASTprogram(defs, instrs, trs);
}

/** Normaliser un programme dans un environnement lexical et global
 * particuliers. */

@Override
public CEASTprogram normalize (
        final INormalizeLexicalEnvironment lexenv,
        final INormalizeGlobalEnvironment common )
  throws NormalizeException {
    // Introduire d'abord toutes les variables globales nommant les
    // fonctions globales:
    IAST4functionDefinition[] definitions = getFunctionDefinitions();
    for ( int i = 0 ; i<definitions.length ; i++ ) {
        IAST4globalFunctionVariable gfv =
            new CEASTglobalFunctionVariable(
                    definitions[i].getDefinedVariable().getName(),
                    definitions[i] );
        common.add(gfv);
    }
    // On normalise toutes les definitions
    final IAST4functionDefinition[] definitions_ =
        new IAST4functionDefinition[definitions.length + 1];
    for ( int i = 0 ; i<definitions.length ; i++ ) {
        definitions_[i] = CEAST.narrowToIAST4functionDefinition(
            definitions[i].normalize(lexenv, common));
    }
    // Empaqueter le code hors fonction en une fonction globale:
    final IAST4expression oldBody = CEAST.narrowToIAST4expression(
            getBody().normalize(lexenv, common));
    final IAST4globalFunctionVariable program =
        CEASTglobalFunctionVariable.generateGlobalFunctionVariable();
    common.add(program);
    final IAST4functionDefinition bodyAsFunction =
        new CEASTfunctionDefinition(program, new IAST4variable[0], oldBody);
    program.setFunctionDefinition(bodyAsFunction);
    definitions_[definitions.length] = CEAST.narrowToIAST4functionDefinition(
        bodyAsFunction.normalize(lexenv, common));
    final IAST4expression body_ =
        // devrait passer par une fabrique pour simplifier InvocationMarker:
        new CEASTglobalInvocation(program, new CEASTexpression[0]);

    // Normaliser les traces:
    final CEASTtraceDeclaration[] traces_ =
        new CEASTtraceDeclaration[this.traces.length];
    for ( int i=0 ; i<traces_.length ; i++ ) {
        traces_[i] = this.traces[i].normalize(lexenv, common);
    }

    return new CEASTprogram(definitions_, body_, traces_);
}
```

```
  // l'analyse statique supplementaire identifiant les invocations a tracer:

  public void markTracedInvocations () {
      Set<IAST4variable> variables = new HashSet<IAST4variable>();
      for ( CEASTtraceDeclaration traceDecl : this.traces ) {
          variables.add(traceDecl.getVariable());
      }
      final InvocationMarker visitor = new InvocationMarker(variables);
      this.accept(visitor);
  }

}

// end of CEASTprogram.java
```

2.7 Question 7 : compilation

Comme la déclaration de trace est une propriété statique, elle ne correspond *per se* à aucune méthode de compilation. Voici, par contre, la compilation d'une invocation qui prend en compte l'éventuelle présence d'une déclaration de trace.

```
/* ****************************************************************
 * ILP -- Implantation d'un langage de programmation.
 * Copyright (C) 2006 <Christian.Queinnec@lip6.fr>
 * $Id: CEASTinvocation.java 572 2006-12-24 17:29:33Z queinnec $
 * GPL version>=2
 * ****************************************************************/

package fr.upmc.ilp.ilp4tr;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.cgen.NoDestination;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IUserFunction;
import fr.upmc.ilp.ilp4.ast.CEASTlocalVariable;
import fr.upmc.ilp.ilp4.ast.NormalizeException;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4invocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.tool.IParser;

public class CEASTinvocation
extends fr.upmc.ilp.ilp4.ast.CEASTexpression
implements IAST4invocation {
```

```
public CEASTinvocation (final IAST4expression function,
                        final IAST4expression[] arguments ) {
    this.delegate =
        new fr.upmc.ilp.ilp4.ast.CEASTinvocation(function, arguments);
    this.traced = false;
}
private boolean traced;
private String /* or null */ functionName;
private IAST4invocation delegate;

protected CEASTinvocation (final IAST4invocation delegate) {
    this.delegate = delegate;
    this.traced = false;
}

// On trace l'appel en indiquant quel est le nom de la fonction:
public void setTraced (final String functionName) {
    this.functionName = functionName;
    this.traced = true;
}
public boolean isTraced () {
    return this.traced;
}

@Override
public IAST4invocation getDelegate () {
    return this.delegate;
}

/* La methode imposée pour le visiteur: */
public void accept (IAST4visitor visitor) {
    visitor.visit(this);
}

// Les methodes passant par le délégué:
public IAST2expression getArgument (int i) {
    return getDelegate().getArgument(i);
}
public int getArgumentsLength () {
    return getDelegate().getArgumentsLength();
}
public IAST4expression[] getArguments () {
    return getDelegate().getArguments();
}
public IAST4expression getFunction () {
    return getDelegate().getFunction();
}

// Note: le délégué raffine ilp4.CEASTinvocation:
public static CEASTinvocation parse (Element e, IParser parser)
throws CEASTparseException {
  IAST4invocation old = fr.upmc.ilp.ilp4.ast.CEASTinvocation.parse(e, parser);
  return new CEASTinvocation(old);
}

@Override
public IAST4invocation normalize (
        final INormalizeLexicalEnvironment lexenv,
```

```
        final INormalizeGlobalEnvironment common )
throws NormalizeException {
  this.delegate = (IAST4invocation) this.delegate.normalize(lexenv, common);
  return this;
}

@Override
public Object eval (final ILexicalEnvironment lexenv,
                    final ICommon common )
throws EvaluationException {
  final Object fn = getFunction().eval(lexenv, common);
  if ( fn instanceof IUserFunction ) {
    final IUserFunction function = (IUserFunction) fn;
    final IAST4expression[] arguments = getArguments();
    final Object[] args = new Object[arguments.length];
    for ( int i=0 ; i<args.length ; i++ ) {
      args[i] = arguments[i].eval(lexenv, common);
    }
    if ( isTraced() ) {
      beforeInvocation(args);
    };
    Object value = function.invoke(args, common);
    if ( isTraced() ) {
      afterInvocation(value);
    };
    return value;
  } else {
    final String msg = "Not␣a␣function␣" + fn;
    throw new EvaluationException(msg);
  }
}

// On ecrit sur stderr car c'est plus simple que de trouver le flux actuel
// d'erreurs dans common car il ne s'y trouve pas!
public void beforeInvocation (Object[] args ) {
    System.err.print("->␣");
    System.err.print(this.functionName);
    for ( Object arg : args ) {
        System.err.print("␣");
        System.err.print(arg.toString());
    }
    System.err.println();
}
public Object afterInvocation (Object value) {
    System.err.print("<-␣");
    System.err.print(this.functionName);
    System.err.print("␣");
    System.err.print(value.toString());
    System.err.println();
    return value;
}

@Override
public void compile (final StringBuffer buffer,
                     final ICgenLexicalEnvironment lexenv,
                     final ICgenEnvironment common,
                     final IDestination destination )
throws CgenerationException {
```

```
    if ( isTraced() ) {
        compileTracedInvocation(buffer, lexenv, common, destination);
    } else {
        getDelegate().compile(buffer, lexenv, common, destination);
    }
}

public void compileTracedInvocation (
        final StringBuffer buffer,
        final ICgenLexicalEnvironment lexenv,
        final ICgenEnvironment common,
        final IDestination destination )
throws CgenerationException {
    IAST4localVariable result = CEASTlocalVariable.generateVariable();
    IAST4expression[] args = getArguments();
    IAST4localVariable[] tmps = new IAST4localVariable[args.length];
    ICgenLexicalEnvironment bodyLexenv = lexenv;
    buffer.append("{␣\n");
    buffer.append("#include␣\"ilpTrace.h\"\n");
    result.compileDeclaration(buffer, lexenv, common);
    bodyLexenv = bodyLexenv.extend(result);
    for ( int i=0; i<args.length ; i++ ) {
        tmps[i] = CEASTlocalVariable.generateVariable();
        tmps[i].compileDeclaration(buffer, lexenv, common);
        bodyLexenv = bodyLexenv.extend(tmps[i]);
    }
    for ( int i=0 ; i<args.length ; i++ ) {
        args[i].compile(buffer, bodyLexenv, common,
                new AssignDestination(tmps[i]) );
        buffer.append(";\n");
    }
    // Trace des arguments:
    buffer.append("ILP_beforeInvocation(\"");
    buffer.append(this.functionName);
    buffer.append("\"");
    for ( IAST4localVariable lv : tmps ) {
        buffer.append(",␣");
        buffer.append(lv.getMangledName());
    }
    buffer.append(",␣NULL);\n");
    // Recuperation du resultat:
    (new AssignDestination(result)).compile(buffer, bodyLexenv, common);
    IDestination no = NoDestination.create();
    getFunction().compile(buffer, bodyLexenv, common, no);
    buffer.append("(");
    for ( int i=0 ; i<(args.length-1) ; i++ ) {
        tmps[i].compile(buffer, bodyLexenv, common, no);
        buffer.append(",␣");
    }
    if ( args.length > 0 ) {
        tmps[args.length-1].compile(buffer, bodyLexenv, common, no);
    };
    buffer.append(");\n");
    // Trace du resultat:
    buffer.append("ILP_afterInvocation(\"");
    buffer.append(this.functionName);
    buffer.append("\",␣");
    buffer.append(result.getMangledName());
```

```
      buffer.append(");\n");
      //
      result.compile(buffer, lexenv, common, destination);
      buffer.append(";\n}\n");
  }

}

// end of CEASTinvocation.java
```

Notez la mémorisation du nom de la variable menant à la fonction pour l'impression des arguments d'appel et du résultat (dans les méthodes `beforeInvocation` et `afterInvocation`).

Notez aussi l'inclusion systématique du fichier `ilpTrace.h` dans la portée du bloc enchâssant l'invocation.

Notez aussi la différence de traitement entre interprétation et compilation vis-à-vis des canaux de sortie. La compilation produit ses traces sur *stdout* parce que `ILP_print` ne sait imprimer que sur ce flux tandis que l'interprétation produit ses traces sur `System.err` parce que le flux de sortie est encapsulé dans les primitives `print` et `newline` et de ce fait assez inaccessible.

Et voici la nouvelle analyse statique identifiant les invocations à tracer. C'est un visiteur classique qui ne s'intéresse qu'aux invocations (en général) et aux invocations globales (en particulier) :

```
/* ****************************************************************
 * ILP -- Implantation d'un langage de programmation.
 * Copyright (C) 2006 <Christian.Queinnec@lip6.fr>
 * $Id: InvocationMarker.java 572 2006-12-24 17:29:33Z queinnec $
 * GPL version>=2
 * ****************************************************************/

package fr.upmc.ilp.ilp4tr;

import java.util.Set;

import fr.upmc.ilp.ilp4.interfaces.AbstractExplicitVisitor;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4invocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;

public class InvocationMarker
extends AbstractExplicitVisitor {

    public InvocationMarker (Set<IAST4variable> variables) {
        this.variables = variables;
    }
    protected Set<IAST4variable> variables;

    @Override
    public void visit (IAST4invocation iast) {
        // Visiter recursivement les sous-noeuds:
        iast.getFunction().accept(this);
        for ( IAST4expression arg : iast.getArguments() ) {
            arg.accept(this);
        }
        // Traitement particulier de l'invocation:
        markInvocation(iast);
    }
```

```
    @Override
    public void visit (IAST4globalInvocation iast) {
        for ( IAST4expression arg : iast.getArguments() ) {
            arg.accept(this);
        }
        markInvocation(iast);
    }

    protected void markInvocation (IAST4invocation iast) {
        if ( iast instanceof CEASTinvocation ) {
            if ( iast.getFunction() instanceof IAST4globalFunctionVariable ) {
                IAST4globalFunctionVariable gfv =
                    fr.upmc.ilp.ilp4.ast.CEAST.narrowToIAST4globalFunctionVariable(
                            iast.getFunction() );
                // Doit avoir lieu apres normalisation:
                if ( variables.contains(gfv) ) {
                    ((CEASTinvocation)iast).setTraced(gfv.getName());
                }
            };
        } else {
            // Cela peut arriver! notamment dans l'invocation globale
            // creee dans ilp4.CEASTprogram. Vraiment besoin d'une fabrique
            // generalisee.
        }
    }

}
```

2.8 Question 8 : interprétation

Cf. ci-dessus.

2.9 Question 9 : méli-mélo

Comme il serait compliqué de traiter, en ordre dispersé, déclaration de trace et définition de fonctions, il suffit, lors de l'analyse syntaxique, de réordonner les unes et les autres ce qui permet de se ramener au problème précédent. Ainsi, il faut juste changer la grammaire pour autoriser ce mélange puisque la méthode `ilp4tr.CEASTprogram.parse` ne se préoccupe pas de l'ordre des composantes d'un programme.

Conclusions

Un nouveau TGZ incorpore le paquetage `ilp4tr`.

Suggestions

Étendre la solution à la trace des fonctions prédéfinies comme `print` et `newline`.
Étudier la méthode consistant à modifier le corps des fonctions.
Étudier la faisabilité des sémantiques non retenues.