Examen final ILP
durée 3 heures
Revision: 1.9

Christian Queinnec

18 janvier 2005

Conditions générales

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

Le contenu du répertoire nommé *workspace/ilp4exam/* dans votre répertoire personnel sera récupéré par l'équipe système de l'ARI-CCE à l'issue de l'épreuve. Toute votre production dans le cadre de cet examen devra y être placée.

Cette épreuve sera corrigée en partie automatiquement et en partie manuellement. Faites très attention aux noms des fichiers demandés. Toute erreur sera nuisible à votre copie ! Veillez également à introduire des commentaires illuminant les correcteurs humains sur vos intentions. Les classes que vous devez écrire doivent être publiques afin d'être utilisables par des programmes de tests.

Le barème est fixé à 60, la durée de l'épreuve est de 3 heures. Tous les documents sont autorisés. Le répertoire */Infos/lms/2004/master/ue/ilp-2004oct/* est toujours accessible.

Voir éléments de solution[1].

1 Installation

Vous avez 15 minutes pour vous installer c'est-à-dire effectuer les opérations qui suivent. Vous pouvez commencer à lire les fichiers, le reste de l'énoncé ne sera visible qu'à 9h45 environ. Normalement, il faut trois minutes pour faire les opérations qui suivent.

Pour installer les fichiers nécessaires à votre environnement de travail pendant cet examen, veuillez exécuter le script suivant :

```
/Infos/lmd/2004/master/ue/ilp-2004oct/E/installer-examen.sh
```

Les fichiers et répertoires suivants seront installés dans votre répertoire personnel. Ils respectent la hiérarchie usuelle d'ILP telle que visible en */Infos/lmd/2004/master/ue/ilp-2004oct/ILP/* et ci-dessous rappelée :

`~/.emacs`	*pour lire les .rnc et .xml commodément*
`~/C/`	*les bibliotheques C usuelles*
`~/Grammars/`	*les grammaires RelaxNG*
`~/Grammars/Samples/`	*des exemples*
`~/Java/jars/`	*les archives usuelles*
`~/Java/src/`	*les classes en Java*
`~/Java/src/fr/upmc/ilp/ilp2enum/`	*quelques classes pour vous aider*
`~/Java/src/fr/upmc/ilp/ilp4enum/`	*quelques classes pour vous aider*

[1] `#solution`

`~/workspace/` *pour Eclipse*

Lancez `eclipse`. À la question *Select a workspace*, répondez OK. Dans la fenêtre qui s'ouvre, cliquez sur *Workbench*. À partir du menu *File*, *New Project*, *Next* (car le type *Java Project* est déjà sélectionné), indiquez le nom du projet qui doit être `ilp4exam`. Eclipse doit alors découvrir que le projet existe déjà (si ce n'est pas le cas, c'est que vous n'avez pas dû donner le nom attendu correct à savoir `ilp4exam`), cliquez alors sur *Finish*. À la question *Confirm Perspective Switch*, répondez Yes.

Ensuite, sélectionnez le projet à gauche, puis avec le menu contextuel (bouton de droite) choisissez *Properties*, onglet *Java Build path* puis onglet *Libraries*, ajoutez, grâce au bouton *add external jars*, les archives correspondant à `isorelax`, `jing`, `junit`, `saxon`, `trang`, `velocity`, `xercesImpl`, `xml-apis` et `xmlunit`. Comme l'indique le s de *jars*, vous pouvez les ajouter toutes en une seule fois (la touche *SHIFT* permet de sélectionner multiplement). En cliquant sur OK, eclipse recompile le projet tout entier normalement sans erreur sauf les erreurs correspondant aux classes à définir dans le paquetage `fr/upmc/ilp/ilp2enum` ou `fr/upmc/ilp/ilp4enum`.

Vous pouvez écrire vos programmes avec Eclipse ou Emacs (ou tout autre moyen à votre convenance). Dans le répertoire `Java` se trouve un `Makefile` qui recompile les classes Java et lance les tests JUnit se trouvant dans le paquetage `fr/upmc/ilp/ilp2enum` :

```
% ( cd Java/ && make )
```

Dernier point en attendant la suite de l'énoncé, vous pouvez commencer à lire les programmes des nouveaux paquetages. L'examen peut être réalisé en ILP2 ou en ILP4 à votre choix. Mais vous avez a matérialiser ce choix en supprimant soit le paquetage `fr/upmc/ilp/ilp2enum` si vous choisissez ILP4, soit le paquetage `fr/upmc/ilp/ilp4enum` si vous choisissez ILP2 (pour supprimer un paquetage avec Eclipse, menu contextuel du paquetage, *Delete*). L'énoncé est rédigé pour le choix ILP2, quelques phrases distinguent les rares aspects divergents d'ILP4 et sont placées entre crochets carrés. Autrement, l'énoncé concernant ILP4 se déduit du présent énoncé en remplaçant dans ce qui suit 2 par 4.

Les fichiers que vous aurez à créer seront, ainsi qu'indiqué question par question, à placer dans le répertoire `workspace/ilp4exam/`. Vos classes Java seront compilées en mode Java 1.4.

2 ILP2enum

Le but de ce problème est d'ajouter une capacité d'énumération au langage défini par ILP2. En termes de syntaxe concrète, la nouvelle construction ressemble (en syntaxe pseudo Perl) à :

```
foreach v in start .. stop {
  body
}
```

Dans cette énumération, la variable *v* a pour portée le corps de l'énumération. Ce corps, désigné ci-dessus par *body*, est une séquence d'instructions. Les expressions *start* et *stop* ne sont évaluées qu'une seule fois et dans cet ordre et fixent les bornes de l'énumération. Ces bornes doivent impérativement être des nombres (entiers ou flottants). Cette énumération est dite par compréhension car l'intervalle de variation est implicitement défini par ses bornes.

Si la valeur de *start* est inférieure ou égale à la valeur de *stop* alors la variable *v* est liée à la valeur de *start* et le corps est exécuté. Une fois le corps de l'énumération exécuté, la variable *v* est incrémentée de 1. Tant que la valeur de *v* est inférieure ou égale à la valeur de *stop*, le corps de l'énumération est exécuté. L'énumération toute entière rend le booléen Faux (comme la primitive `print`).

Pour vous libérer de certains problèmes, quelques ressources sont disponibles :

- la grammaire reconnaissant le langage ILP2enum vous est donnée en format RNC en `Grammars/grammar2enum.rnc` et en format RNG en `Grammars/grammar2enum.rng`
- un analyseur syntaxique prenant du XML et produisant un DOM *Java/src/fr/upmc/ilp/ilp2enum/CEASTParser.java*
- une classe abstraite dont vous pouvez hériter *Java/src/fr/upmc/ilp/ilp2enum/AbstractCEASTforeach.java*
- une classe de tests JUnit *Java/src/fr/upmc/ilp/ilp2enum/CompilerTest.java*

– une classe de suite de tests JUnit *Java/src/fr/upmc/ilp/ilp2enum/CEASTTestSuite.java*
– des fichiers de tests associés en *Grammars/Samples/e1*-2enum.xml*
– quelques `Makefile`

Le paquetage `fr.upmc.ilp.ilp2enum` sera le paquetage que vous aurez à compléter.

Question 1

Écrire un programme en ILP2enum comportant une énumération et imprimant les nombres 1, 2, 3 et 4.

Livraison

– un fichier XML valide vis-à-vis de la grammaire d'ILP2enum, imprimant 1234 et comportant au moins une énumération. Ce fichier sera nommé *workspace/ilp4exam/e1.xml*.

Notation sur 6 points

– 2 points si votre fichier est valide vis-à-vis de *Grammars/grammar2enum.rng*
– 2 points s'il comporte une énumération en compréhension
– 2 points s'il imprime bien 1234

Question 2

Écrire une classe `fr.upmc.ilp.ilp2enum.CEASTforeachInRange`, héritant de la classe abstraite `fr.upmc.ilp.ilp2enum.Abstract` et implantant une méthode `eval` d'interprétation. [[Pour ILP4, il faut aussi définir une méthode `normalize` appropriée. Les autres méthodes `findGlobalVariables`, `findInvokedFunctions` et `inline` ne sont pas demandées (les versions héritées (quoiqu'incomplètes) d'`fr.upmc.ilp.ilp4enum.AbstractCEASTforeach` peuvent être utilisées)]].

Livraison

– un fichier Java nommé `workspace/ilp4exam/src/fr/upmc/ilp/ilp2enum/CEASTforeachInRange.java`

Notation sur 10 points

– 1 point si votre classe se compile correctement
– 9 points si votre classe passe avec succès les tests fournis

Question 3

Compléter la précédente classe `fr.upmc.ilp.ilp2enum.CEASTforeachInRange` 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 ! [[En ILP4, les méthodes `findGlobalVariables`, `findInvokedFunctions` et `inline` appropriées ne sont pas demandées.]]

Livraison

– un fichier Java nommé `workspace/ilp4exam/src/fr/upmc/ilp/ilp2enum/CEASTforeachInRange.java`

Notation sur 13 points

– 2 points si votre classe se compile correctement
– 11 points si votre classe passe avec succès les tests fournis

3 Transformation

Plutôt que d'écrire des méthodes d'interprétation et de compilation pour les énumérations, on pourrait les transformer en des expressions d'ILP2. Dans le cas d'ILP2, lorsque cette méthode est définie dans la classe `fr.upmc.ilp.ilp2enum.CEASTforeachInRange`, elle est automatiquement invoquée par `fr.upmc.ilp.ilp2enum.CEASTParser`. [[Dans le cas d'ILP4, elle se place naturellement dans la passe de normalisation.]]

Question 4

Compléter la classe `fr.upmc.ilp.ilp2enum.CEASTforeachInRange` en implantant une méthode `transform` transformant une énumération par compréhension en un code équivalent écrit en ILP2 et n'utilisant aucune énumération.

Livraison

- un fichier Java nommé `workspace/ilp4exam/src/fr/upmc/ilp/ilp2enum/CEASTforeachInRange.java`

Notation sur 10 points

- 10 points si votre classe passe avec succès les tests fournis

4 ILP2enumens

Les énumérations précédentes correspondaient à des énumérations par compréhension puisque l'ensemble des valeurs sur lesquelles itérer était fourni implicitement par ses bornes. On désire maintenant ajouter une capacité d'énumération explicite. En termes de syntaxe concrète, la nouvelle construction ressemble à :

`foreach` v `in (` e_1 e_2 `...` `) {`
  body
`}`

Dans cette énumération, la variable v a pour portée le corps de l'énumération. Ce corps, désigné ci-dessus par *body*, est une séquence d'instructions. Les expressions e_1, e_2 etc. ne sont évaluées qu'une seule fois et dans cet ordre. La variable v est liée à la valeur du premier terme e_1 et le corps est exécuté. La variable v est alors liée à la valeur du second terme e_2 et le corps est exécuté. Et ainsi de suite jusqu'à avoir exploité toutes les valeurs des termes spécifiés. L'énumération toute entière ne retourne aucune valeur particulière.

Pour vous libérer de certains problèmes, quelques ressources additionnelles sont disponibles :

- la grammaire reconnaissant le langage ILP2enum vous est donnée en format RNC en `Grammars/grammar2enumens.rnc` et en format RNG en `Grammars/grammar2enumens.rng`
- une classe abstraite `fr.upmc.ilp.ilp2enum.AbstractCEASTforeachInSet` dont votre classe héritera
- [[Pour ILP4, une nouvelle destination `fr.upmc.ilp.ilp2enum.AssignIndexedDestination` qui pourra éventuellement vous servir]]
- des fichiers de tests associés en *Grammars/Samples/e13*-2enumens.xml*

Le paquetage `fr.upmc.ilp.ilp2enum` sera toujours le paquetage que vous aurez à compléter.

Question 5

Écrire un programme en ILP2enumens comportant une telle énumération et imprimant les nombres 1, 12, 129 et 34.

Livraison

- un fichier XML valide vis-à-vis de la grammaire d'ILP2enumens, imprimant 11212934 et comportant l'énumération demandée. Ce fichier sera nommé *workspace/ilp4exam/e2.xml*.

Notation sur 6 points

- 2 points si votre fichier est valide vis-à-vis de *Grammars/grammar2enumens.rng*
- 2 points s'il comporte une énumération explicite
- 2 points s'il imprime bien 11212934

Question 6

Écrire la classe `fr.upmc.ilp.ilp2enum.CEASTforeachInSet`, héritant de la classe `fr.upmc.ilp.ilp2enum.AbstractCEASTforeach` et pourvue d'une méthode `eval` d'interprétation.

Livraison

- un fichier Java nommé `workspace/ilp4exam/src/fr/upmc/ilp/ilp2enum/CEASTforeachInSet.java`

Notation sur 7 points

- 1 point si votre classe se compile correctement
- 6 points si votre classe passe avec succès les tests fournis

Question 7

Compléter la classe `fr.upmc.ilp.ilp2enum.CEASTforeachInSet` avec une méthode `compile` de compilation vers C.

Livraison

- un fichier Java nommé `workspace/ilp4exam/src/fr/upmc/ilp/ilp2enum/CEASTforeachInSet.java`

Notation sur 8 points

- 8 points si votre classe passe avec succès les tests fournis

5 Éléments de solution

Les solutions sont données en version ILP2 puis ILP4.

5.1 Question 1

Il s'agissait d'écrire un programme en ILP2enum imprimant des chiffres de 1 à 4. Ce programme existait pratiquement déjà tout fait dans les programmes exemples, c'était le premier exemple et il suffisait de transformer 3 en 4.

```
<?xml version='1.0' encoding='ISO-8859-15' ?>
<programme2
><enumeration
><variable nom='i'
/><comprehension
><debut
><entier valeur='1'
/></debut
><fin
><entier valeur='4'
/></fin
></comprehension
```

```
><corps
><invocationPrimitive fonction='print'
><variable nom='i'
/></invocationPrimitive
></corps
></enumeration
></programme2
>
```

5.2 Question 2 et 3

Voici la classe pour ILP2enum (avec les méthodes `eval` et snipcompile) :

```
// $Id$

package fr.upmc.ilp.ilp2enum;

import org.w3c.dom.*;
import java.util.*;
import java.math.*;
import fr.upmc.ilp.ilp2.*;

public class CEASTforeachInRange extends AbstractCEASTforeach {

  public CEASTforeachInRange (final CEASTvariable variable,
                              final CEASTExpression start,
                              final CEASTExpression stop,
                              final CEASTInstruction body ) {
    super(variable, body);
    this.start = start;
    this.stop = stop;
  }
  final private CEASTExpression start;
  final private CEASTExpression stop;

  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    Object start_value = start.eval(lexenv, common);
    final Object stop_value = stop.eval(lexenv, common);
    while ( true ) {
      final Object bool = common.applyOperator("<=", start_value, stop_value);
      if ( ((Boolean) bool).booleanValue() ) {
          final ILexicalEnvironment newlexenv =
            lexenv.extend(variable, start_value);
          body.eval(newlexenv, common);
          start_value = common.applyOperator("+", start_value, ONE);
      } else {
        break;
      }
    }
    return Boolean.FALSE;
  }
  private static final BigInteger ONE = new BigInteger("1");

  public void compile_instruction (final StringBuffer buffer,
```

```
                                final ICgenLexicalEnvironment lexenv,
                                final ICgenEnvironment common,
                                final String destination)
  throws CgenerationException {
  buffer.append(" {\n");
  buffer.append(" ILP_Object ");
  variable.compile_expression(buffer, lexenv, common);
  buffer.append(" = ");
  start.compile_expression(buffer, lexenv, common);
  buffer.append(";\n");
  final CEASTvariable tmp_stop = CEASTvariable.generateVariable();
  buffer.append(" ILP_Object ");
  tmp_stop.compile_expression(buffer, lexenv, common);
  buffer.append(" = ");
  stop.compile_expression(buffer, lexenv, common);
  buffer.append(";\n");
  final ICgenLexicalEnvironment new_lexenv =
    lexenv.extend(variable).extend(tmp_stop);

  buffer.append(" while ( ILP_isEquivalentToTrue(ILP_LessThanOrEqual(");
  buffer.append(variable.getName());
  buffer.append(", ");
  buffer.append(tmp_stop.getName());
  buffer.append(")) ) {\n");
  body.compile_instruction(buffer, new_lexenv, common, "(void)");
  buffer.append("\n");
  buffer.append(variable.getName());
  buffer.append(" = ILP_Plus(");
  buffer.append(variable.getName());
  buffer.append(", ILP_Integer2ILP(1)); \n");
  buffer.append("}\n }\n");
  final CEASTInstruction something = CEASTInstruction.voidInstruction();
  something.compile_instruction(buffer, lexenv, common, destination);
}

public CEAST transform () {
  final CEASTvariable fin = CEASTvariable.generateVariable();
  CEASTInstruction result =
    new CEASTunaryBlock(
        variable,
        start,
        new CEASTsequence(
            new CEASTInstruction[] {
                new CEASTunaryBlock(
                    fin,
                    stop,
                    new CEASTsequence(
                        new CEASTInstruction[] {
                            new CEASTwhile(
                                new CEASTbinaryOperation(
                                    "<=",
                                    variable,
                                    fin ),
                                new CEASTsequence(
                                    new CEASTInstruction[] {
                                        body,
```

```
                                        new CEASTassignment(
                                            variable,
                                            new CEASTbinaryOperation(
                                                "+",
                                                variable,
                                                new CEASTinteger("1")))}
                                    ))}))}));
    return result;
  }

}

// end of CEASTforeachInRange.java
```

et l'équivalent pour ILP4enum :

```
// $Id$

package fr.upmc.ilp.ilp4enum;

import org.w3c.dom.*;
import java.util.*;
import java.math.*;
import fr.upmc.ilp.ilp4.*;

public class CEASTforeachInRange extends AbstractCEASTforeach {

  public CEASTforeachInRange (final IVariable variable,
                              final CEASTExpression start,
                              final CEASTExpression stop,
                              final CEASTExpression body ) {
    super(variable, body);
    this.start = start;
    this.stop = stop;
  }
  final private CEASTExpression start;
  final private CEASTExpression stop;

  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    Object start_value = start.eval(lexenv, common);
    final Object stop_value = stop.eval(lexenv, common);
    while ( true ) {
      final Object bool = common.applyOperator("<=", start_value, stop_value);
      if ( ((Boolean) bool).booleanValue() ) {
          final ILexicalEnvironment newlexenv =
            lexenv.extend(variable, start_value);
          body.eval(newlexenv, common);
          start_value = common.applyOperator("+", start_value, ONE);
      } else {
        break;
      }
    }
    return Boolean.FALSE;
  }
  private static final BigInteger ONE = new BigInteger("1");
```

```
public void compile (final StringBuffer buffer,
                     final ICgenLexicalEnvironment lexenv,
                     final ICgenEnvironment common,
                     final IDestination destination)
  throws CgenerationException {
  buffer.append(" {\n");
  variable.compileDeclaration(buffer, lexenv, common);
  final IVariable tmp_stop = CEASTlocalVariable.generateVariable();
  tmp_stop.compileDeclaration(buffer, lexenv, common);
  final ICgenLexicalEnvironment new_lexenv =
    lexenv.extend(variable).extend(tmp_stop);
  start.compile(buffer, new_lexenv, common, new AssignDestination(variable));
  stop.compile(buffer, new_lexenv, common, new AssignDestination(tmp_stop));
  buffer.append(" while ( ILP_isEquivalentToTrue(ILP_LessThanOrEqual(");
  buffer.append(variable.getName());
  buffer.append(", ");
  buffer.append(tmp_stop.getName());
  buffer.append(")) ) {\n");
  body.compile(buffer, new_lexenv, common, VoidDestination.create());
  buffer.append("\n");
  buffer.append(variable.getName());
  buffer.append(" = ILP_Plus(");
  buffer.append(variable.getName());
  buffer.append(", ILP_Integer2ILP(1)); \n");
  buffer.append("}\n }\n");
  final CEASTExpression something = CEASTExpression.voidExpression();
  something.compile(buffer, lexenv, common, destination);
}

public Set findGlobalVariables (final Set globalvars,
                                final ICgenLexicalEnvironment lexenv,
                                final ICgenEnvironment common ) {
  super.findGlobalVariables(globalvars, lexenv, common);
  start.findGlobalVariables(globalvars, lexenv, common);
  stop.findGlobalVariables(globalvars, lexenv, common);
  return globalvars;
}

public CEAST normalize (final INormalizeLexicalEnvironment lexenv,
                        final INormalizeGlobalEnvironment common )
  throws NormalizeException {
  final CEASTExpression start_ =
    (CEASTExpression) start.normalize(lexenv, common);
  final CEASTExpression stop_ =
    (CEASTExpression) stop.normalize(lexenv, common);
  final INormalizeLexicalEnvironment bodylexenv = lexenv.extend(variable);
  final CEASTExpression body_ =
    (CEASTExpression) body.normalize(bodylexenv, common);
  return new CEASTforeachInRange(variable, start_, stop_, body_);
}

public CEAST transform (final INormalizeLexicalEnvironment lexenv,
                        final INormalizeGlobalEnvironment common )
  throws NormalizeException {
  final CEASTlocalVariable deb = new CEASTlocalVariable(variable.getName());
```

```
    final CEASTlocalVariable fin = CEASTlocalVariable.generateVariable();
    CEASTExpression result =
      new CEASTlocalBlock(
          new IVariable[] { deb },
          new CEASTExpression[] { start },
          new CEASTlocalBlock(
              new IVariable[] { fin },
              new CEASTExpression[] { stop },
              new CEASTwhile(
                  new CEASTbinaryOperation(
                      "<=",
                      deb,
                      fin ),
                  new CEASTsequence(
                      new CEASTExpression[] {
                        body,
                        new CEASTassignment(
                            deb,
                            new CEASTbinaryOperation(
                                "+",
                                deb,
                                new CEASTinteger("1")))
                      })));
    return result.normalize(lexenv, common);
  }

  public void findInvokedFunctions () {
    super.findInvokedFunctions();
    start.findInvokedFunctions();
    this.invokedFunctions.addAll(start.getInvokedFunctions());
    stop.findInvokedFunctions();
    this.invokedFunctions.addAll(stop.getInvokedFunctions());
  }

  public void inline () {
    super.inline();
    start.inline();
    stop.inline();
  }

}

// end of CEASTforeachInRange.java
```

Les points délicats étaient de créer le bon environnement et d'y placer la variable d'itération correctement initialisée. Réaliser l'incrémentation de la variable d'itération nécessitait soit d'analyser la valeur (entière ou flottante) et de réaliser la bonne opération, soit d'utiliser l'opérateur + d'ILP qui le fait tout seul. Au passage, on remarquera que l'incrémentation est effectuée par l'introduction d'une nouvelle liaison et non par une affectation.

En revanche la compilation vers C use d'une affectation pour incrémenter la variable d'itération.

5.3 Question 4

La transformation consiste à réécrire la boucle en terme d'ILP2 seulement (d'ailleurs l'un d'entre vous a implanté les méthodes précédentes grâce à `transform`).

```
foreach v in start .. stop {
  body
}
```

peut se réécrire en (deux nouvelles variables nommées *debut* et *fin* ont été introduites pour assurer que les expressions *start* et *fin* ne sont évaluées qu'une seule fois et dans le bon ordre.

```
let v   = start
  let fin = stop
    while ( v <= fin ) {
       body
       v = v + 1;
    }
```

Le code correspondant à cette transformation apparaît dans les classes précédentes.

5.4 Question 5

Voici un tel programme simplement adapté des programmes exemples fournis.

```
<?xml version='1.0' encoding='ISO-8859-15' ?>
<!--

;;; $Id$
(comment "Enumeration par ensemble")
(foreach i (1 12 129 34)
  (print i) )

;;; end of e130-4enum.scm

 -->
<programme2
><!-- test:name description='Enumeration par ensemble'
--><enumeration
><variable nom='i'
/><termes
><terme
><entier valeur='1'
/></terme
><terme
><entier valeur='12'
/></terme
><terme
><entier valeur='129'
/></terme
><terme
><entier valeur='34'
/></terme
></termes
><corps
><invocationPrimitive fonction='print'
><variable nom='i'
/></invocationPrimitive
></corps
></enumeration
></programme2
>
```

5.5 Question 6 et 7

Voici la classe pour ILP2enumens (avec les méthodes `eval` et snipcompile) :

```
// $Id$

package fr.upmc.ilp.ilp2enum;

import org.w3c.dom.*;
import java.util.*;
import fr.upmc.ilp.ilp2.*;

public class CEASTforeachInSet extends AbstractCEASTforeachInSet {

  public CEASTforeachInSet (final CEASTvariable variable,
                            final CEASTlist terms,
                            final CEASTInstruction body ) {
    super(variable, terms, body);
  }

  public CEASTforeachInSet (final CEASTvariable variable,
                            final CEASTExpression[] terms,
                            final CEASTInstruction body ) {
    super(variable, terms, body);
  }

  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    Object values[] = new Object[terms.length];
    for ( int i = 0 ; i<terms.length ; i++ ) {
      values[i] = terms[i].eval(lexenv, common);
    }
    for ( int i = 0 ; i<terms.length ; i++ ) {
      final ILexicalEnvironment newlexenv = lexenv.extend(variable, values[i]);
      body.eval(newlexenv, common);
    }
    return Boolean.FALSE;
  }

  public void compile_instruction (final StringBuffer buffer,
                                   final ICgenLexicalEnvironment lexenv,
                                   final ICgenEnvironment common,
                                   final String destination)
    throws CgenerationException {
    buffer.append(" {\n");
    buffer.append(" ILP_Object ");
    variable.compile_expression(buffer, lexenv, common);
    buffer.append(";\n");
    CEASTvariable tmp_var = CEASTvariable.generateVariable();
    buffer.append("int ");
    buffer.append(tmp_var.getName());
    buffer.append(";\n");
    CEASTvariable term_var = CEASTvariable.generateVariable();
    buffer.append("ILP_Object ");
    buffer.append(term_var.getName());
    buffer.append("[");
```

```
    buffer.append(terms.length);
    buffer.append("];\n");

    for ( int i = 0 ; i<terms.length ; i++ ) {
      term_var.compile_expression(buffer, lexenv, common);
      buffer.append("[");
      buffer.append(i);
      buffer.append("] = ");
      terms[i].compile_expression(buffer, lexenv, common);
      buffer.append(";\n");
    }

    final ICgenLexicalEnvironment new_lexenv = lexenv.extend(variable);
    buffer.append(" for ( ");
    buffer.append(tmp_var.getName());
    buffer.append(" = 0 ; ");
    buffer.append(tmp_var.getName());
    buffer.append("<");
    buffer.append(terms.length);
    buffer.append(" ; ");
    buffer.append(tmp_var.getName());
    buffer.append("++ ) {\n");

    buffer.append(variable.getName());
    buffer.append(" = ");
    buffer.append(term_var.getName());
    buffer.append("[");
    buffer.append(tmp_var.getName());
    buffer.append("];\n");

    body.compile_instruction(buffer, new_lexenv, common, "(void)");
    buffer.append("\n}\n");

    final CEASTInstruction something = CEASTInstruction.voidInstruction();
    something.compile_instruction(buffer, lexenv, common, destination);
    buffer.append("}\n");
  }

}

// end of CEASTforeachInSet.java
```

et le même en ILP4enumens

```
// $Id$

package fr.upmc.ilp.ilp4enum;

import org.w3c.dom.*;
import java.util.*;
import fr.upmc.ilp.ilp4.*;

public class CEASTforeachInSet extends AbstractCEASTforeachInSet {

  public CEASTforeachInSet (final IVariable variable,
                            final CEASTlist terms,
                            final CEASTExpression body ) {
```

```
  super(variable, terms, body);
}

public CEASTforeachInSet (final IVariable variable,
                          final CEASTExpression[] terms,
                          final CEASTExpression body ) {
  super(variable, terms, body);
}

public Object eval (final ILexicalEnvironment lexenv,
                    final ICommon common)
  throws EvaluationException {
  Object values[] = new Object[terms.length];
  for ( int i = 0 ; i<terms.length ; i++ ) {
    values[i] = terms[i].eval(lexenv, common);
  }
  for ( int i = 0 ; i<terms.length ; i++ ) {
    final ILexicalEnvironment newlexenv = lexenv.extend(variable, values[i]);
    body.eval(newlexenv, common);
  }
  return Boolean.FALSE;
}

public void compile (final StringBuffer buffer,
                     final ICgenLexicalEnvironment lexenv,
                     final ICgenEnvironment common,
                     final IDestination destination)
  throws CgenerationException {
  buffer.append(" {\n");
  variable.compileDeclaration(buffer, lexenv, common);
  IVariable tmp_var = CEASTlocalVariable.generateVariable();
  buffer.append("int ");
  buffer.append(tmp_var.getName());
  buffer.append(";\n");
  IVariable term_var = CEASTlocalVariable.generateVariable();
  buffer.append("ILP_Object ");
  buffer.append(term_var.getName());
  buffer.append("[");
  buffer.append(terms.length);
  buffer.append("];\n");

  for ( int i = 0 ; i<terms.length ; i++ ) {
    terms[i].compile(buffer, lexenv, common,
                     new AssignIndexedDestination(term_var, i));
  }

  final ICgenLexicalEnvironment new_lexenv = lexenv.extend(variable);
  buffer.append(" for ( ");
  buffer.append(tmp_var.getName());
  buffer.append(" = 0 ; ");
  buffer.append(tmp_var.getName());
  buffer.append("<");
  buffer.append(terms.length);
  buffer.append(" ; ");
  buffer.append(tmp_var.getName());
  buffer.append("++ ) {\n");
```

```
    buffer.append(variable.getName());
    buffer.append(" = ");
    buffer.append(term_var.getName());
    buffer.append("[");
    buffer.append(tmp_var.getName());
    buffer.append("];\n");

    body.compile(buffer, new_lexenv, common, VoidDestination.create());
    buffer.append("\n}\n");

    final CEASTExpression something = CEASTExpression.voidExpression();
    something.compile(buffer, lexenv, common, destination);
    buffer.append("}\n");
  }

}

// end of CEASTforeachInSet.java
```

L'instruction suivante :

```
foreach v in ( e1 e2 ... ) {
  body
}
```

peut être réécrite en

```
let valeurs = array( e1 e2 ... )
  foreach vindex ( 1 .. array_length(valeurs) ) {
    let v = valeurs[vindex]
      body
      vindex =vindex + 1
```

Ce qui peut aisément se compiler en C puisque les tableaux y existent (d'où en ILP4 la nouvelle destination `AssignIndexedDestination`. Pour l'interprétation, on utilise les tableaux de Java. La « solution » suivante consistant à expanser le code en explicitant tous les cas est bien sûr à proscrire dans un compilateur qui se respecte.

```
let v = e1
   body;
let v = e2
   body;
...
```