Revision: 1.6

Master d'informatique 2009-2010
Spécialité STL
« Implantation de langages »
ILP – MI016
épisode ILP4

C.Queinnec

Buts

- Une analyse statique : l'intégration (ou *inlining*) et de nouvelles analyses statiques préparatoires
  - α-conversion
  - approximation du graphe d'appel
- Techniques Java :
  - Délégation
  - Visiteur réflexif
  - Discrimination par table associative

Plan du cours 7

Intégration (pour *inlining*) de fonctions

- Préliminaires
- Normalisation
- Graphe d'appel
- Intégration
- Réflexions

ILP4 = ILP3 + intégration.

Intégration

L'optimisation reine : 20% d'amélioration.
Transformation de programme. On remplace des appels de fonctions par leur corps, après substitution de leurs variables par les paramètres d'appel.

```
function f (x y) {
   let t = x + y
   in 2*t;
}

let z = 34 in
let y = f(1, z) in ...
```

```
let z = 34 in
let y = ({ let t = 1 + z;
           in 2*t;
         }) in ...
```


Mais les variables peuvent aussi être modifiées :

```
function f (x y) {
   x = x + y;
   in 2*x;
}

let z = 34 in
let y = f(1, z) in ...
```

```
let z = 34 in
let y = ({ x = 1;
           y = z;

           x = x + y;
           in 2*x;
         }) in ...
```

Il faut donc recréer les liaisons correspondantes pour les affectations.

Mais attention aux portées :

```
z = 2;
function f (x y) {
   return x + y + z;
}

let z = 34 in
let y = f(1, z) in ...
```

```
z = 2;
let z = 34 in
let  y = ({ x = 1;
            y = z;

            return x + y + z;
          }) in ...
```

Capture de la variable locale `z` par la référence libre à la variable globale `z` depuis la fonction `f`.

Intérêts

- Supprimer des invocations (sauvegarde registres, allocation en pile, restauration registres, responsabilité des registres (appelant ou appelé ?) etc.)
- Rapprocher des fragments de code indépendants (surtout avec précalculs statiques (*constant folding*) et suppression du code mort (*dead code elimination*).

```
function f(x, y) {
  if (x > 1 ) {
    x + y
  } else {
    x
  }
}
let z = 2 in
  print(3 + f(z, t));
```

```
print(3 + (2 + t));
```


Problèmes

- Respect des portées
- Équivalence expression/instruction
- Prévention des conflits de noms

Solution

Renommage de toutes les variables locales (alpha-conversion) :

```
z = 2;
function f (x y) {
   let z = x + y + z;
   in 2*z;
}

let z = 34 in
let x = 5 in
let y = f(1, x+z)
in ...
```

```
z = 2;
function f (x1 y1) {
   let z1 = x1 + y1 + z;
   in 2*z1;
}

let z0 = 34 in
let x0 = 5 in
let y0 = f(1, x0+z0)
in ...
```



```
z = 2;
function f (x1 y1) {
  let z1 = x1+y1+z;
  in 2*z1;
}

let z0 = 34 in
let x0 = 5 in
let y0 = f(1, x0+z0)
in ...
```

```
z = 2;
let z0 = 34 in // α-conversion
let x0 = 5 in
let y0 = ({  // liaisons:
      let x1 = 1
      and y1 = x0 + z0 in
      // corps:
      let z1 = x1 + y1 + z;
      in 2*z1;
         }) in ...
```


Paquetages pour ILP4

Le super-paquetage `fr.upmc.ilp.ilp4` contient les paquetages habituels :

```
 LOC
 500 fr.upmc.ilp.ilp4              process et tests
 700 fr.upmc.ilp.ilp4.interfaces   interfaces divers
6000 fr.upmc.ilp.ilp4.ast          AST et analyses s
 200 fr.upmc.ilp.ilp4.runtime      bibliotheque d'i
  33 fr.upmc.ilp.ilp4.cgen         compilation vers
```

Grammaire `Grammars/grammar4.rnc`
Nouveau patron `C/templateTest3.c`
Programmes ILP4 additionnels `Grammars/Samples/*-4.xml`

Équivalence expression/instruction

Ressource: Grammars/grammar4.rnc

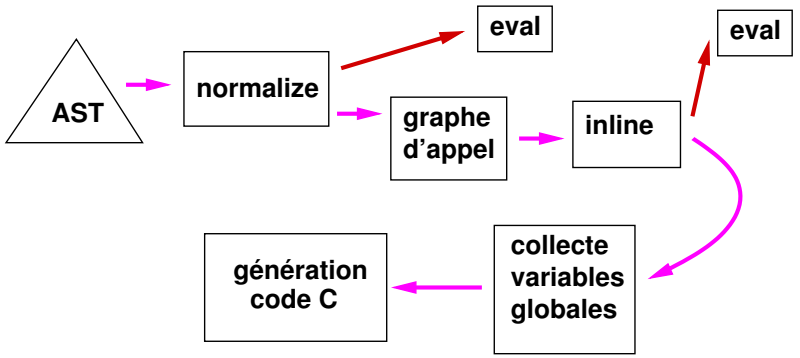
Nouvelle grammaire où toute instruction est aussi une expression. Je n'ai pas réussi à la définir incrémentiellement à partir des précédentes !

```
programme4 = element programme4 {
   definitionEtExpressions
}

definitionEtExpressions =
   definitionFonction *,
   expression +
```

```
expression =
    alternative             | sequence
  | blocUnaire              | blocLocal
  | boucle                  | try
  | affectation             | invocation
  | constante               | variable
  | operation               | invocationPrimitive
```


Passes de traitement

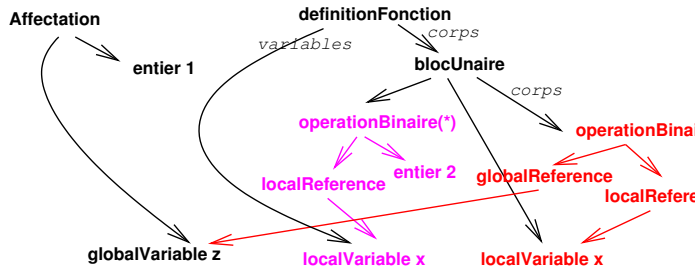
Quatre analyses statiques dont la normalisation des expressions de l'AST.



Normalisation

Partage physique des objets représentant les variables.
Taxonomie des variables locales, globales, globales fonctionnelles, prédéfinies.
Au passage, les séquences d'une seule expression sont normalisées à cette seule expression. Vérification de l'arité des appels aux fonctions globales.
Transformation IAST2 (résultant de `parse`) vers IAST4

```
z = 1;
function f(x) {
  let x = 2*x
  in z+x
}
```



Prévention des conflits de noms

- Deux références à une même variable (locale ou globale) sont représentées par le même objet en mémoire.
- Taxonomie des variables locales, globales, globales fonctionnelles, prédéfinies.
- Les séquences d'une seule expression sont normalisées à cette seule expression.
- Vérification de l'arité des appels aux fonctions globales.

L'identification des variables :

- améliore la comparaison (et notamment la vitesse de l'interprète (comme en Lisp avec la notion de symbole))
- réalise l'alpha-conversion (l'adresse est le nom).

Comparaison

Comparaison physique plutôt que structurelle :

```
// depuis LexicalEnvironment
  public Object lookup (IVariable otherVariable)
    throws EvaluationException {
    if ( variable == otherVariable ) {
      return value;
    } else {
      return next.lookup(otherVariable);
    }
  }
```


Normalisation

La normalisation est encore un parcours avec deux environnements :

- l'environnement lexical `INormalizeLexicalEnvironment`
- l'environnement global `INormalizeGlobalEnvironment`

Chaque nœud de l'AST procure des méthodes pour ces différentes passes.

```
public interface IAST4
extends IAST2 {

    IAST4 normalize (INormalizeLexicalEnvironment lexenv,
                     INormalizeGlobalEnvironment common )
    throws NormalizeException;

    void findInvokedFunctions ()
    throws FindingInvokedFunctionsException;

    Set<IAST4globalFunctionVariable> getInvokedFunctions ();

    void inline () throws InliningException;
}
```


Taxonomie des variables

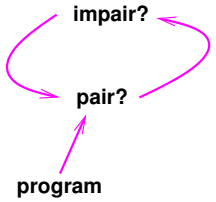
- Variables locales `CEASTlocalVariable`
- Variables globales `CEASTglobalVariable`
- Noms de fonctions globales `CEASTglobalFunctionVariable`
- Noms de fonctions prédéfinies `CEASTPredefinedVariable`

```
function foo (x) {
   ...
}
let x = 3
in x := 4;          // CEASTlocalAssignment
   g := foo(x);     // CEASTglobalAssignment
   foo := g;        // CEASTglobalVariable ???
```


locale	**globale**	**prédéfinie**
CEASTlocalVariable	CEASTglobalVariable	CEASTPredefinedVariabl
CEASTlocalAssignment	CEASTglobalAssignment	
CEASTinvocation	CEASTglobalInvocation	CEASTprimitiveInvocatio

Les variables globales fonctionnelles permettent de retrouver la définition de fonction ainsi nommée.
L'analyseur syntaxique fabrique un AST peu précis (sur les variables, les affectations et les invocations). L'analyse `normalize()` précise tout cela.

Calcul du graphe d'appels




```
function pair (n) {
   if ( n == 0 ) {
     return true;
   } else {
     return impair(n - 1);
   }
}
function impair (n) {
   if ( n == 0 ) {
     return false;
   } else {
     return pair(n - 1);
   }
```


Après calcul du graphe d'appels,

- on n'intègre que les fonctions non récursives.
- Par contre, on intégrera toutes les fonctions non récursives.

Calcul fermeture transitive

```
                    // Passe 1
pair    → impair
impair  → pair
program → pair
                    // Passe 2
pair    → impair, pair
impair  → pair, impair
program → pair, impair
                    // Passe 3: point fixe
pair    → impair, pair    → recursive
impair  → pair, impair    → recursive
program → pair, impair
```


Ressource: Java/src/fr/upmc/ilp/ilp4/ast/CEASTprogram.java

```
public void findInvokedFunctions ()
throws FindingInvokedFunctionsException {
  IAST4functionDefinition[] definitions = getFunctionDefinitions();
  for ( int i = 0 ; i<definitions.length ; i++ ) {
    definitions[i].findInvokedFunctions();
  }
  boolean shouldContinue = true;
  while ( shouldContinue ) {
    shouldContinue = false;
    for ( int i = 0 ; i<definitions.length ; i++ ) {
       IAST4functionDefinition currentFunction = definitions[i];
       for ( IAST4globalFunctionVariable gv :
                currentFunction.getInvokedFunctions()
                .toArray(IAST4GFV_EMPTY_ARRAY) ) { // !
          IAST4functionDefinition other =
               gv.getFunctionDefinition();
          shouldContinue |= currentFunction.addInvokedFunctions(
                  other.getInvokedFunctions());
       }
    }
  }
  // Savoir ce qu'invoque le programme est peu utile!
  findAndAdjoinToInvokedFunctions(getBody());
}
```

- `findInvokedFunctions` calcule un sous-ensemble des fonctions invoquées.
- `getInvokedFunctions` renvoie ce sous-ensemble.

MOCHE d'avoir ajouté un champ `invokedFunctions` à chaque expression !

Intégration

Pour tout nœud `CEASTglobalInvocation` et si la fonction invoquée n'est pas récursive : l'intégrer.
Le résultat de l'intégration est stocké dans le champ `inlined`.
L'ordre d'intégration importe peu à condition de passer partout mais seulement une seule fois.


```
public void inline () throws InliningException {
  if ( this.inlined != null ) {
     return;
  } else {
     for ( IAST4expression arg : getArguments() ) {
        arg.inline();
     }
     if ( getFunction() instanceof CEASTglobalFunctionVariable ) {
       IAST4globalFunctionVariable gv =
         (CEASTglobalFunctionVariable) getFunction();
       IAST4functionDefinition function = gv.getFunctionDefinition();
       if ( function.isRecursive() ) {
         // On n'integre pas les fonctions recursives!
         return;
       } else {
         // La fonction a toutes les qualites requises, on l'integre!
         this.inlined = new CEASTlocalBlock(
               (IAST4variable[]) function.getVariables(),
               getArguments(),
               (IAST4expression) function.getBody());
         // inlined.inline(); // deja fait quand function fut analysee.
         return;
         }
     } else {
         // La fonction est le resultat d'un calcul, on ne la connait pas!
         return;
     }
  }
}
```


Génération de code

Les fonctions non récursives sont éliminées puisqu'intégrées.
Pour les invocations intégrées, on utilise l'expression dans le champ `inlined`.

Conclusions sur intégration

- L'intégration supprime les instructions d'appel donc améliore la vitesse

Analyses statiques

Conclusions sur intégration

- L'intégration supprime les instructions d'appel donc améliore la vitesse
- mais augmente la taille du code donc diminue l'efficacité de la mémoire virtuelle.

Analyses statiques

Conclusions sur intégration

- L'intégration supprime les instructions d'appel donc améliore la vitesse
- mais augmente la taille du code donc diminue l'efficacité de la mémoire virtuelle.
- Il est possible de déplier finiment les fonctions récursives.
- On peut prendre en compte l'augmentation de taille (globale ou locale) et en faire un critère d'intégration.
- On peut ajouter des déclarations `inline` pour indiquer les intégrations utiles.

Techniques Java

- Inversion expression/instruction
  - Usage plus fin des destinations
- Nouvel analyseur syntaxique plus générique
- Délégation
- Méta-méthodes pour `inline` et annotations
- Visiteur

IAST4 et IAST2

Toutes les catégories syntaxiques ont une interface étendue IAST4*.
Elles dérivent toutes d'IAST4 (pour les nouvelles méthodes d'ILP4) et de l'interface équivalente IAST2*.

```
ilp4.IAST4
       IAST4program
       IAST4functionDefinition
       IAST4expression          // etend IAST2expressi
         IAST4instruction       // etend IAST2instruct
           IAST4while           // etend IAST2while
         IAST4variable
           IAST4localVariable
           IAST4globalVariable
             IAST4globalFunctionVariable
             IAST4predefinedVariable
```

Méthode unifiée `compile`. Du coup `compileExpression` et `compileInstruction` sont rendues obsolètes (`@Deprecated`).

En ILP4

Certaines méthodes retournent des types plus précis que suggérés dans l'interface : des instances satisfaisant IAST4* plutôt qu'IAST2*.
Certaines méthodes s'attendent à des types plus précis qu'imposés par l'interface : des instances satisfaisant IAST4* plutôt qu'IAST2*. Il faut donc les convertir et, pour aider à la mise au point, existent les méthodes statiques `CEAST.narrowToIAST4*()`.

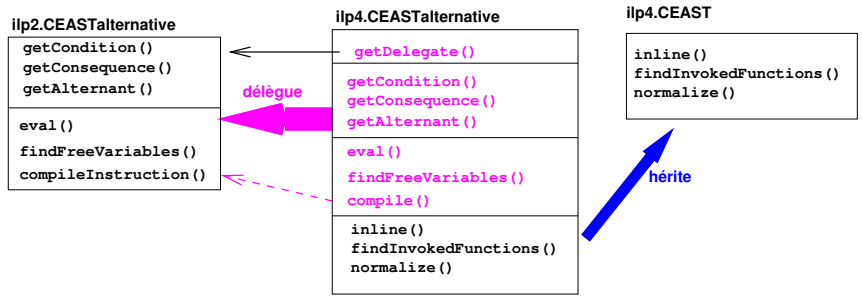
```
public interface IAST4assignment
extends IAST4expression, IAST2assignment {
  IAST4variable   getVariable(); // raffinement
  IAST4expression getValue();    // raffinement
}
```



```
public class CEASTassignment
extends fr.upmc.ilp.ilp4.CEASTexpression
implements IAST4assignment {

  public CEASTassignment (IAST4variable variable,
                          IAST4expression value) {
    ...
  }

  @ILPvariable           // annotation
  public IAST4variable getVariable () {
    return CEAST.narrowToIAST4variable(...);
  }

  @ILPexpression         // annotation
  public IAST4expression getValue() {
    return CEAST.narrowToIAST4expression(...);
  }
```


Délégation

Les nouvelles classes `ilp4.CEAST*` réalisent les interfaces `IAST4*` mais héritent d'`ilp4.CEAST` pour partager le code de certaines nouvelles méthodes, elles n'héritent donc pas d'`ilp[23].CEAST*` pourtant les comportements qui s'y trouvent (`eval`, `compile`, `findFreeVariables`) sont ceux que l'on désire.



```
public class CEASTassignment
extends CEASTexpression          // ilp4.ast.CEASTexpression
implements IAST4assignment {

  public CEASTassignment (final IAST4variable variable,
                          final IAST4expression value) {
      this.delegate =
          new fr.upmc.ilp.ilp2.ast.CEASTassignment(
                  variable, value );
  }
  private fr.upmc.ilp.ilp2.ast.CEASTassignment delegate;

  @Override
  public IAST2assignment getDelegate () {
      return this.delegate;
  }
  @ILPvariable
  public IAST4variable getVariable () {
    return CEAST.narrowToIAST4variable(getDelegate().getVariable())
  }
  @ILPexpression
  public IAST4expression getValue() {
    return CEAST.narrowToIAST4expression(getDelegate().getValue());
  }
```



```
public abstract class CEASTexpression
extends CEAST
implements IAST4expression {
  public abstract IAST2 getDelegate ();

  public Object eval (ILexicalEnvironment lexenv,
                      ICommon common)
  throws EvaluationException {
      return getDelegate().eval(lexenv, common);
  }

  public void compile (StringBuffer buffer,
                       ICgenLexicalEnvironment lexenv,
                       ICgenEnvironment common,
                       IDestination destination)
  throws CgenerationException {
      IAST2expression delegate = (IAST2expression)
          getDelegate();
      delegate.compileExpression(buffer, lexenv,
                                 common, destination);
  }
```


Nouvel analyseur par réflexion

Toutes les classes d'AST ont une méthode statique `parse(Element, IParser)` pour analyser un noeud DOM.

```
public class CEASTParser extends AbstractParser {

  public CEASTParser ()
  throws CEASTparseException {
    this.parsers = new HashMap<String, Method>();
    addParser("programme4", CEASTprogram.class);
    addParser("alternative", CEASTalternative.class)
    ...
  }
  private final HashMap<String, Method> parsers;
```



```
 /** Ajout d'une caracteristique a ILP. Lorsque l'element
  * XML nomme name est lu, la methode clazz.create(e, parser)
  * sera invoquee. */

 public void addParser (String name, Class clazz)
 throws CEASTparseException {
   try {
     Method method = clazz.getMethod("parse",
        new Class[]{ Element.class, IParser.class } );
     if ( ! Modifier.isStatic(method.getModifiers()) ) {
       final String msg = "Non static parse() method!";
       throw new CEASTparseException(msg);
     };
     parsers.put(name, method);
   } catch (SecurityException e1) {
     String msg = "Cannot access parse() method!";
     throw new CEASTparseException(msg);
   } catch (NoSuchMethodException e1) {
     String msg = "Cannot find parse() method!";
     throw new CEASTparseException(msg);
   }
 }
```

```
public IAST4 parse (final Node n)
  throws CEASTparseException {
  switch ( n.getNodeType() ) {

  case Node.DOCUMENT_NODE: {
    Document d = (Document) n;
    return this.parse(d.getDocumentElement());
  }

  case Node.ELEMENT_NODE: {
    Element e = (Element) n;
    String name = e.getTagName();

    if ( parsers.containsKey(name) ) {
        Method method = parsers.get(name);
        try {
          Object result = method.invoke(null, new Object[]{e, this});
          return CEAST.narrowToIAST4(result);
        } catch (IllegalArgumentException exc) {
          throw new CEASTparseException(exc);
        } catch (IllegalAccessException exc) {
          throw new CEASTparseException(exc);
        } catch (InvocationTargetException exc) {
            Throwable t = exc.getTargetException();
            if ( t instanceof CEASTparseException ) {
                throw (CEASTparseException) t;
            } else {
                throw new CEASTparseException(exc);
            }
        }

    } else {
      String msg = "Unknown element name: " + name;
      throw new CEASTparseException(msg);
     }
  }
  default: {
    String msg = "Unknown node type: " + n.getNodeName();
    throw new CEASTparseException(msg);
   }
  }
}
```


Analyses statiques

- La détermination des fonctions invoquées est un simple parcours de l'AST. À partir de chaque noeud, on explore les sous-arbres contenant des expressions.
- Pour `findFreeVariables()`, on ne traitait spécialement que les `CEASTreference` en remplissant un `Set<IAST2variable>`.
- Pour `inline()` et `findInvokedFunctions()`, on ne traite spécialement que les `CEASTglobalInvocation`. Pour `inline()` on renseigne le champ `inlined`. Pour `findInvokedFunctions()` on remplit un `Set<IAST4globalFunctionVariable>`.
- Pour `normalize()` on reconstruit un AST équivalent.

ILP2 — `findFreeVariables`

- Au début, `Set<IAST2variable>` vide
- Parcourir l'AST... une méthode par classe de nœud qui ne fait qu'appeler récursivement `findFreeVariables` sur les sous-expressions avec le `Set<IAST2variable>` courant
- Pour chaque `CEASTreference`, remplir `Set<IAST2variable>` avec la variable si non déjà présente
- À la fin, rendre `Set<IAST2variable>`

Environ 15 méthodes en tout.
Signature : `findFreeVariables(Set<IAST2variable>, lexenv)`

ILP4 — `findInvokedFunctions`

- Au début, pour chaque définition de fonction `Set<IAST4globalFunctionVariable>` vide
- Parcourir l'AST... une méthode sur `CEAST`, une autre sur `CEASTglobalInvocation` sinon appeler récursivement la méthode sur les sous-expressions et fusionner les résultats
- Pour chaque `CEASTglobalInvocation`, remplir `Set<IAST4globalFunctionVariable>` avec la variable nommant la fonction
- À la fin, rendre `Set<IAST4globalFunctionVariable>` via `getInvokedFunctions()`

2 méthodes en tout.
Signature : `findInvokedFunctions()`,

ILP4 — inline

- Parcourir l'AST... une méthode sur `CEAST`, une autre sur `CEASTglobalInvocation` sinon appeler récursivement la méthode sur les sous-expressions
- Pour chaque `CEASTglobalInvocation`, décorer l'AST avec la version intégrée
- À la fin, ne rien rendre

2 méthodes en tout.
Signature : `inline()`,

ILP4 — normalize

- Parcourir l'AST... une méthode par nœud normalizant ses sous-expressions et effectuant au passage la taxonomie des références, des affectations et des invocations, plus quelques normalisations (alternatives binaires, séquences d'une instruction) ou vérifications (arités des invocations)
- À la fin, rendre un nouvel AST

15 méthodes en tout.
Signature : `normalize(lexenv, common)`,

Où sont les sous-expressions ?

Dans tous ces cas, on se demande où sont les sous-expressions composant l'expression. On identifie les accesseurs menant à des sous-expressions avec l'annotation `@ILPexpression` et ceux menant à des variables avec l'annotation `@ILPvariable`. On peut donc avoir une méthode par défaut qui arpente tout noeud et ses sous-expressions.

Annotations

```
package fr.upmc.ilp.annotation;
import java.lang.annotation.*;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Target(ElementType.METHOD)
public @interface ILPexpression {

  /** Indique si la valeur obtenue par la methode
   * peut etre null. */

  boolean neverNull () default true;

  /** Indique si la valeur obtenue est en fait un
   * vecteur d'expressions. */

  boolean isArray () default false;
}
```

Malheureusement pas d'héritage pour les définitions d'annotations !

Détermination des invocations

La méthode par défaut est :

```
public void findInvokedFunctions ()
throws FindingInvokedFunctionsException {
  for ( Method m : this.getClass().getMethods() ) {
     try { // FIXME: mettre en cache cette recherche!
        ILPexpression e = m.getAnnotation(ILPexpression.class);
        if ( e != null ) {
          if ( e.isArray() ) {
            Object[] results = (Object[])
               m.invoke(this, EMPTY_ARGUMENT_ARRAY);
            for ( Object result : results ) {
               if ( e.neverNull() || result != null ) {
                 IAST4expression component =
                    CEAST.narrowToIAST4expression(result);
                 this.findAndAdjoinToInvokedFunctions(component);
               };
            }
          } else {
            Object result =
              m.invoke(this, EMPTY_ARGUMENT_ARRAY);
            if ( e.neverNull() || result != null ) {
              IAST4expression component =
                 CEAST.narrowToIAST4expression(result);
              this.findAndAdjoinToInvokedFunctions(component);
            };
          }
        }
     } catch (IllegalArgumentException e) {
       throw new FindingInvokedFunctionsException(e);
     } catch (IllegalAccessException e) {
       throw new FindingInvokedFunctionsException(e);
     } catch (InvocationTargetException e) {
       throw new FindingInvokedFunctionsException(e);
     }
  }
}
private final Set<IAST4globalFunctionVariable> invokedFunctions;
// NOTE: un tel champ par instance est dispendieux!
```


Encore plus de destination !

L'inversion expression/instruction fait que toute expression ILP peut maintenant contenir des fragments qui ne peuvent être compilés que sous forme d'instructions C. D'où problème !

En ILP2 : $\overrightarrow{alternative}^{d}$

```
if ( ILP_isEquivalentToTrue( condition ) ) {
   consequence ;
} else {
   alternant ;
}
```



```
// en ILP2
public void compileInstruction (
      StringBuffer buffer,
      ICgenLexicalEnvironment lexenv,
      ICgenEnvironment common,
      IDestination destination)
throws CgenerationException {
  buffer.append(" if ( ILP_isEquivalentToTrue( ");
  getCondition().compileExpression(buffer, lexenv,
  buffer.append(" ) ) {\n");
  getConsequent().compileInstruction(buffer, lexer
  buffer.append(";\n} else {\n");
  getAlternant().compileInstruction(buffer, lexenv
  buffer.append(";\n}");
}
```


Principes de compilation

- Les variables ILP sont compilées en variables C
- Les expressions ILP sont compilées en **instructions** C
- Les instructions ILP sont compilées en instructions C

```
                    ⟶d
               alternative

{ ILP_Object tmp;

    ⟶tmp
  condition;
  if ( ILP_isEquivalentToTrue( tmp ) ) {
       ⟶d
    consequence ;
  } else {
       ⟶d
    alternant ;
  }
}
```



```
// en ILP4
public void compile (StringBuffer buffer,
                     ICgenLexicalEnvironment lexenv,
                     ICgenEnvironment common,
                     IDestination destination)
throws CgenerationException {
  IAST4variable tmp = CEASTlocalVariable.generateVariable();
  buffer.append("{ ");
  tmp.compileDeclaration(buffer, lexenv, common);
  ICgenLexicalEnvironment bodyLexenv =
     lexenv.extend(tmp);
  getCondition().compile(buffer, lexenv, common,
                         new AssignDestination(tmp));
  buffer.append(";\n if ( ILP_isEquivalentToTrue( ");
  tmp.compile(buffer, bodyLexenv, common,
              NoDestination.create());
  buffer.append(" ) ) {\n");
  getConsequent().compile(buffer, lexenv, common, destination)
  buffer.append(";\n } else {\n");
  getAlternant().compile(buffer, lexenv, common, destination);
  buffer.append(";\n }\n}");
}
```


Visiteur

```
public interface IAST4visitor {
    Object visit (IAST4alternative iast,
                  Object data);
    Object visit (IAST4assignment iast,
                  Object data);
    ...
}
public interface IAST4visitable {
    Object accept (IAST4visitor visitor,
                   Object data);
}
```



```
public class XMLwriter implements IAST4visitor {
   public Object visit (IAST4alternative iast,
                        Object data) {
      ...
      iast.getCondition().accept(this, data);
      iast.getConsequent().accept(this, data);
      iast.getAlternant().accept(this, data);
      ...
      return data; }
   public Object visit (IAST4assignment iast,
                        Object data) {
      ...
      iast.getVariable().accept(this, data);
      iast.getValue().accept(this, data);
      ...
      return data; }
  ...
}
```

Mais la discrimination n'a plus à être écrite :

```
// dans CEASTalternative implements IAST4visitable
  public Object accept (IAST4visitor visitor,
                        Object data) {
      visitor.visit(this);
  }
// dans CEASTassignment implements IAST4visitable
  public Object accept (IAST4visitor visitor,
                        Object data) {
      visitor.visit(this);
  }
```

L'état du parcours est stocké dans le visiteur, voir exemple `XMLwriter`. La différence avec la méthode `analyze` d'ILP1 est qu'il n'y a plus besoin d'écrire le code de discrimination, on a mis à profit le mécanisme d'envoi de message pour ce faire (au prix d'une duplication de code toutefois).

Pour la prochaine fois

- Tester ILP4 sur des exemples
- Lire le code de ce grand saut technologique
- Refaire la collecte des variables globales (ou le comptage des constantes) comme un visiteur (cf. aussi `AbstractExplicitVisitor`)