

Revision: 1.2

Master d'informatique 2009-2010
Spécialité STL
« Implantation de langages »
ILP – MI016
épisode ILP6

C. Queinnec

Objets, classes et envoi de message

- Préliminaires
- Syntaxe
- Interprétation
- Compilation

Caractéristiques

- classe
 - définition (globale)
 - héritage simple (champs et comportements)
 - allocation (sans initialisation) d'objets
- accès aux champs
- méthode
 - appartenance à une classe
 - moi
 - méthode héritée
 - super
- appel de méthode

Paquetage

super-paquetage `fr.upmc.ilp.ilp6` avec les paquetages usuels :

```
fr.upmc.ilp.ilp6.interfaces // 6 classes
                        .ast // 12 classes
                        .runtime // 5 classes
                        .cgen // 4 classes
```

Des ajouts à la grammaire précédente `Grammars/grammar6.rnc`
 et des programmes de tests en `Grammars/Samples/*-6.xml`

Syntaxe

On s'écarte de JavaScript qui est un langage à prototypes.

```
definitionClasse = element definitionClasse {
  attribute nom      { xsd:Name },
  attribute parent { xsd:Name } ?,
  element champs {
    element champ {
      attribute nom { xsd:Name }
    } *
  } ?,
  element methodes {
    element methode {
      attribute nom      { xsd:Name },
      element variables { variable * },
      element corps      { expression + }
    } *
  } ?
}
```

```
creationObjet = element creationObjet {
  # new Classe(arguments)
  attribute classe { xsd:Name },
  expression *
}
lectureChamp = element lectureChamp {
  # objet.champ
  attribute champ { xsd:Name },
  element cible  { expression }
}
ecritureChamp = element ecritureChamp {
  # objet.champ = expression
  attribute champ { xsd:Name },
  element cible  { expression },
  element valeur { expression }
}
```

Exemples syntaxiques

```
envoiMessage = element envoiMessage {
  # receveur.message(arguments)
  attribute message { xsd:Name },
  element receveur  { expression },
  element arguments { expression } *
}
appelSuper = element appelSuper {
  # super()
  empty
}
moi = element moi {
  # this
  empty
}
```

```
class Point { // extends Object // Ordre des definiti
  field x; field y;
  method setXlike (p) { this.x = p.x; }
  method voir () { return "x=" + double(this)/2
                  + ",y=" + this.y; }
}
class ColoredPoint extends Point {
  field color; // pas de valeur d'initialisation
  method getColor () { return this.color; }
  method voir () { return this.color + super(); }
}
function double (x) { return x + x; }
let p1 = new ColoredPoint(11, 22, "blue")
and p2 = new Point(1, 9) in
  print(p1.voir());
  p2.setXlike(p1.truc());
```

```

fr.upmc.ilp.ilp6.interfaces
  .IAST6classDefinition
  .IAST6methodDefinition
  .IAST6instantiation
  .IAST6readField
  .IAST6writeField
  .IAST6super
  .IAST6self
  .IAST6program
  .IAST6visitor

```

```

// paquetage fr.upmc.ilp.ilp6.ast
CEAST6
  CEASTclassDefinition // implante IAST6classDefi
  CEASTmethodDefinition // implante IAST6methodDej
CEASTprogram // etend ilp4.ast.CEASTpro
CEAST6expression // etend ilp4.ast.CEASTexp
  CEASTinstantiate
  CEASTreadField
  CEASTwriteField
  CEASTself
  CEASTsuper
  CEASTsend

```

Statique / dynamique

Les choix d'implantation :

- classe
 - définition (globale) **statique**
 - héritage simple (champs et comportements) **statique**
 - allocation d'objets **statique**
 - super **statique**
- ```

new Point(1, 2) // permis
o.getClass().newInstance(1, 2) // interdit

```
- pas de comportement réflexif.

- accès aux champs

**statique**

```

p.x // doit verifier que x est un champ de l'objet p
p["x"] // interdit

```

Attention : pour simplifier en ILP6, deux classes non reliées par héritage ne peuvent avoir un champ ou une méthode homonyme !  
Si  $C_1$  et  $C_2$  ont un champ ou une méthode de même nom alors  $C_1 \subseteq C_2 \vee C_2 \subseteq C_1$ .

## Modèle de données

## ● méthode

- appartenance à une classe
- moi
- méthode héritée

**dynamique**  
**dynamique**  
**statique**

```
// La methode voir de ColoredPoint
method voir () {
 return
 // this est un ColoredPoint au sens large
 this.color
 +
 // la methode voir de Point
 super();
}
```

## ● appel de méthode

**dynamique**

```
nom -> Classe // interpretation ou compilation
Classe nom
 nom superclasse
 champs herites
 methodes heritees
 champs propres (liste de noms)
 methodes propres (liste de fonctions globales)
Instance classe
 champs (nom -> valeur)
Methode fonction globale
 nom
 variables (liste)
 corps
```

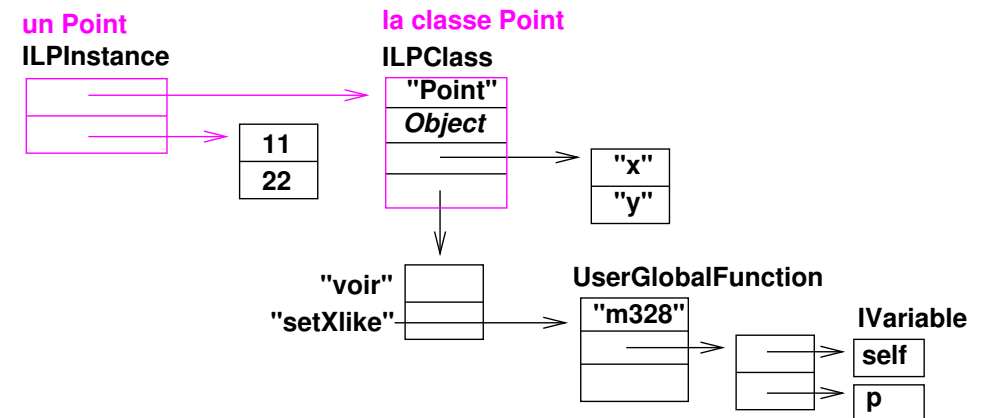
## Interprétation

Les classes sont des ressources globales donc stockées dans l'environnement global `common`.

Les classes sont représentées par des instances d'`ILPClass`

```
String className
ILPClass superClass
String[] fieldName // propres et herites
Map method // propres seulement

String getName()
String fieldName(int)
int fieldSize()
int fieldRank(fieldName)
Object send(self, message, argument [], common)
```



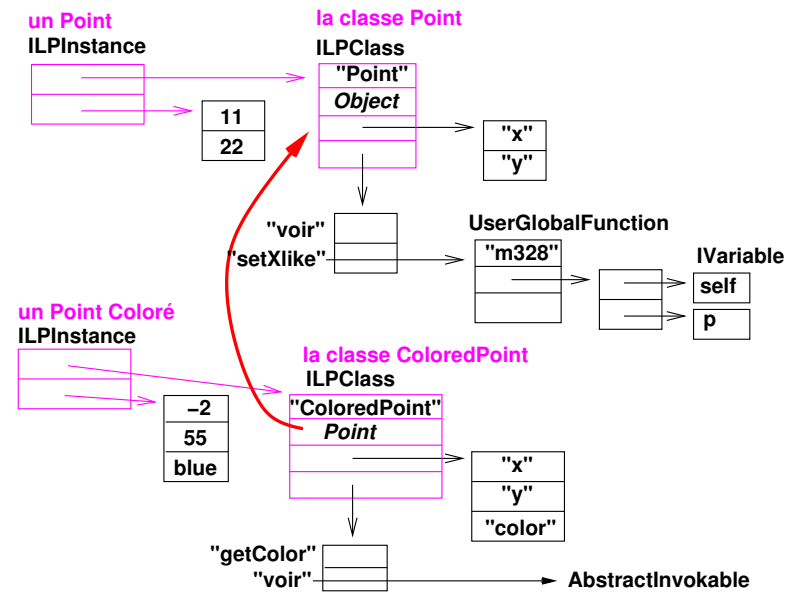
Les méthodes sont représentées par des `UserGlobalFunction` dont la première variable est `ilpSelf`. Par contre la définition des méthodes est l'apanage de `IAST6methodDefinition`.

```
o.m(a, b) ≡ m_global(o, a, b)
```

Les instances d'ILP sont représentées par des instances de `ILPInstance`

```
ILPClass clazz;
Object[] fields;

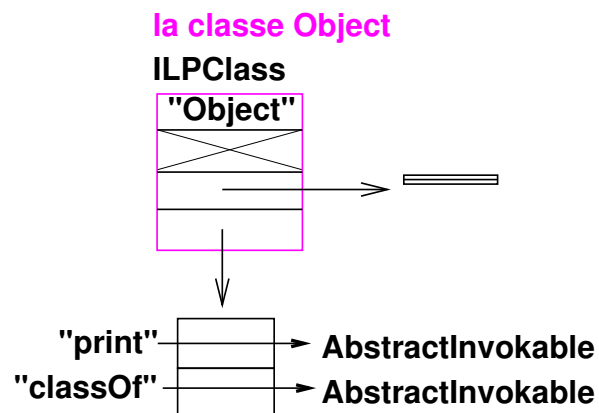
Object read(fieldName)
Object write(fieldName, object)
Object send(message, argument, common)
```



## Détails d'interprétation

La classe racine : `Object` qui pourrait avoir ses champs et méthodes propres. En fait, elle n'est pas représentée explicitement dans l'interprète.

```
print(o) ≡ o.print()
```



## Allocation

```
public class CEASTInstantiate
extends CEASTExpression {
 public Object eval (ILexicalEnvironment lexenv,
 ICommon common)
 throws EvaluationException {
 ILPClass clazz = common.findClass(className);
 Object[] value = new Object[argument.length];
 for (int i = 0 ; i<argument.length ; i++) {
 value[i] = argument[i].eval(lexenv, common);
 }
 return new ILPInstance(clazz, value);
 }
}
```

## Instance

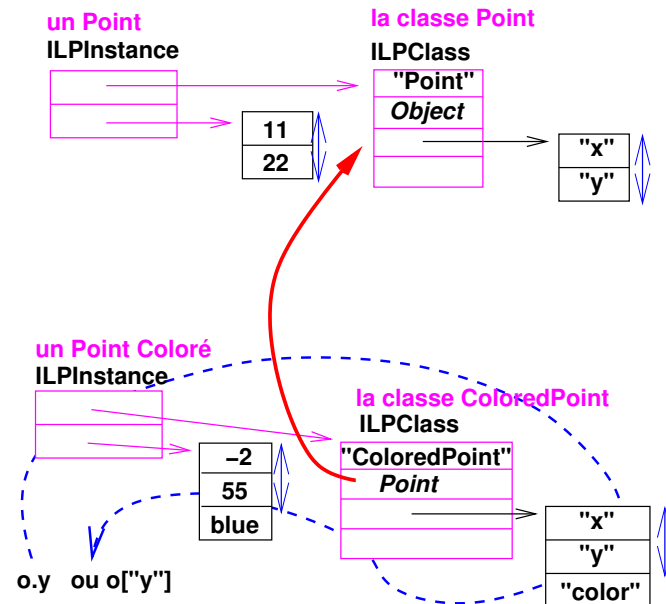
## Accès aux champs

Les noms des champs sont convertis en des décalages.

```
public class ILPInstance {
 public ILPInstance (ILPClass clazz, Object [] argument) {
 this.clazz = clazz;
 this.field = argument;
 }
 private final ILPClass clazz;
 private final Object [] field;

 public Object read (final String fieldName)
 throws EvaluationException {
 return field[clazz.fieldRank(fieldName)];
 }
 public Object write (final String fieldName, final Object value)
 throws EvaluationException {
 field[clazz.fieldRank(fieldName)] = value;
 return Boolean.FALSE;
 }
 public Object send (final String message,
 final Object [] argument,
 final ICommon common)
 throws EvaluationException {
 return clazz.send(this, message, argument, common);
 }
}
```

```
public class CEASTwriteField
extends CEASTExpression {
 public Object eval (ILexicalEnvironment lexenv,
 ICommon common)
 throws EvaluationException {
 Object target = object.eval(lexenv, common);
 Object newValue = value.eval(lexenv, common);
 if (target instanceof ILPInstance) {
 return ((ILPInstance) target)
 .write(fieldName, newValue);
 } else {
 throw new EvaluationException("Wrong class");
 }
 }
}
```

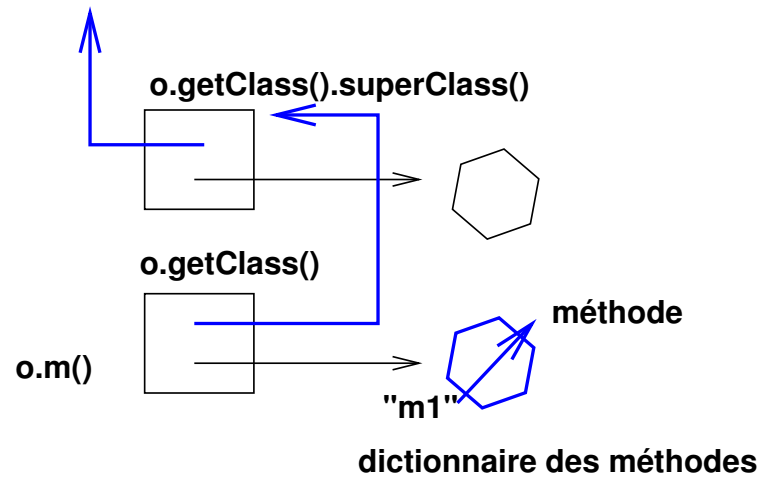


## Envoi de message

À la Smalltalk (*lookup*) :

```
public class CEASTsend extends CEASTExpression {
 public Object eval (ILexicalEnvironment lexenv,
 ICommon common)
 throws EvaluationException {
 Object target = receiver.eval(lexenv, common);
 Object [] value = new Object[argument.length];
 for (int i = 0 ; i < argument.length ; i++) {
 value[i] = argument[i].eval(lexenv, common);
 }
 if (target instanceof ILPInstance) {
 return ((ILPInstance) target)
 .send(methodName, value, common);
 } else {
 throw new EvaluationException(
 "No such method " + methodName);
 }
 }
}
```

## Classe



```
public class ILPClass {
 ...
 public String fieldName (int rank) {
 return this.fieldName[rank];
 }
 public int fieldSize () {
 return this.fieldName.length;
 }
 public int fieldRank (final String name)
 throws EvaluationException {
 for (int i = 0 ; i<fieldName.length ; i++) {
 if (fieldName[i].equals(name)) {
 return i;
 };
 }
 throw new EvaluationException(
 "No such field " + name);
 }
}
```

```
public Object send (final ILPInstance self,
 final String message,
 final Object[] argument,
 final ICommon common)
 throws EvaluationException {
 UserGlobalFunction m = (UserGlobalFunction)
 this.method.get(message);
 if (m != null) {
 final Object[] newArgument = new Object[1 + argument.length];
 newArgument[0] = self;
 for (int i = 0 ; i<argument.length ; i++) {
 newArgument[i+1] = argument[i];
 }
 return m.invoke(newArgument, common);
 } else {
 // Pas de methode propre de ce nom!
 if (superClass != null) {
 return superClass.send(self, message, argument, common);
 } else {
 // On est sur Object
 throw new EvaluationException("No such method " + message);
 }
 }
}
```

## Moi

C'est la valeur de la variable introduite en premier dans la liste des variables des méthodes.

```
// dans CEASTself
public Object eval6 (ILexicalEnvironment lexenv,
 ICommon common)
 throws EvaluationException {
 return lexenv.lookup(this);
}
```

## Super

```

public class CEASTmethodDefinition
extends CEAST6 implements IAST6methodDefinition {
public CEASTmethodDefinition (IAST4functionDefinition delegate) {
 this.methodName = delegate.getFunctionName();
 // Ajouter self en tete des variables:
 this.selfVariable = new CEASTself();
 IAST4variable[] vars = delegate.getVariables();
 IAST4variable[] varsPlusSelf = new IAST4variable[1+vars.length]
 varsPlusSelf[0] = this.selfVariable;
 for (int i=0 ; i<vars.length ; i++) {
 varsPlusSelf[i+1] = vars[i];
 }
 // toutes les fonctions sous-jacentes aux methodes ont des noms
 CEASTglobalFunctionVariable gfv =
 new CEASTglobalFunctionVariable("ilpMETHOD_" + getCounter())
 this.delegate = new CEASTfunctionDefinition(
 gfv,
 varsPlusSelf,
 delegate.getBody());
}
private final CEASTfunctionDefinition delegate;
private final IAST4variable selfVariable;
private final String methodName;

```

Différent de JavaScript et Java : `super()` désigne la valeur de la super-méthode invoquée avec les mêmes arguments que la méthode courante. Ainsi

```

class A extends B {
 method f (x) {
 return x
 + super() // en Java: super.f(x)
 }
}

```

La méthode invoquée est la méthode `f` valable en `B` : elle est connue dès la compilation car il n'y a pas de création dynamique de méthode.

```

// Dans CEASTsuper
public Object eval6 (ILexicalEnvironment lexenv,
 ICommon common)
throws EvaluationException {
 ILPmethod currentMethod = (ILPmethod) lexenv.loc
 return currentMethod.callSuper(lexenv, common);
}

```

```

// Dans ILPmethod
public static IAST4localVariable cmv =
 new CEASTlocalVariable("ilp_CurrentMethod");
protected static IAST4localVariable cmargs =
 new CEASTlocalVariable("ilp_CurrentArguments");

public void setDefiningClass (ILPClass clazz) {
 this.definingClass = clazz;
}
private ILPClass definingClass;

public ILPClass getDefiningClass () {
 return this.definingClass;
}

```



## Compilation

```

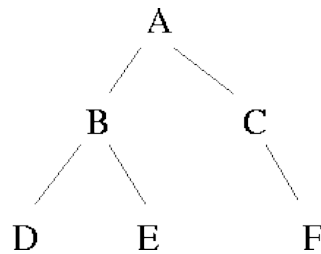
public Object callSuper (ILexicalEnvironment lexenv,
 ICommon common)
throws EvaluationException {
 Object[] arguments = (Object[]) lexenv.lookup(cmargs);
 return getDefiningClass().getSuperClass()
 .send(getMethodName(), arguments, common);
}
@Override
public Object invoke (Object[] arguments,
 ICommon common)
throws EvaluationException {
 IAST2variable[] variables = getVariables();
 if (variables.length != arguments.length) {
 String msg = "Wrong arity";
 throw new EvaluationException(msg);
 };
 ILexicalEnvironment lexenv = getEnvironment()
 .extend(cmv, this)
 .extend(cmargs, arguments);
 for (int i = 0 ; i<variables.length ; i++) {
 lexenv = lexenv.extend(variables[i], arguments[i]);
 }
 return getBody().eval(lexenv, common);
}

```

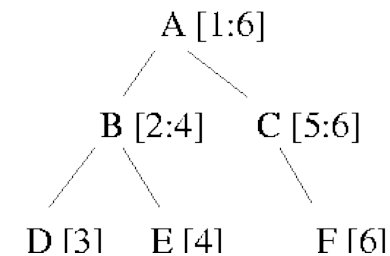
- pas d'objet en C
- pas de classe mais des `struct`
- accès aux champs statique
- déjà une solution pour `instanceof`
- pointeurs
- fonctions

## Héritage simple

`class-of` s'obtient en temps constant avec un schéma par en-tête.  
La question restante correspond à `instanceof`.

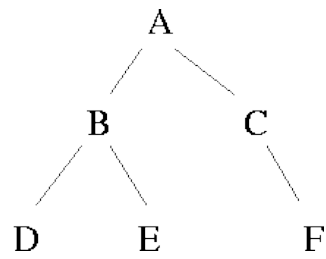


## Numérotation préfixe



`X.min ≤ o.class().min ≤ X.max`

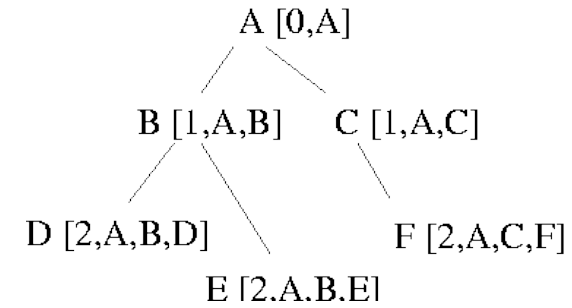
## Matrice caractéristique



|   | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | t |   |   |   |   |   |
| B | t | t |   |   |   |   |
| C | t |   | t |   |   |   |
| D | t | t |   | t |   |   |
| E | t | t |   |   | t |   |
| F | t |   | t |   |   | t |

```
matrice[o.class().numero, X.numero]
```

## Indexation en profondeur

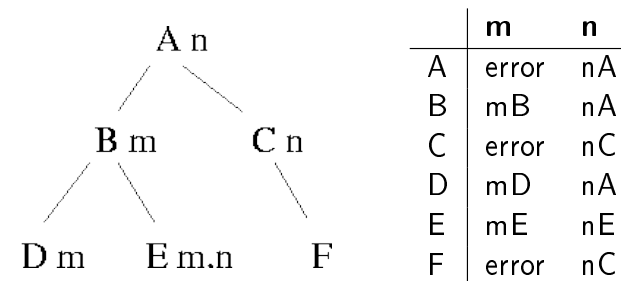


```
X.depth() <= o.class().depth()
&& o.class().supers[X.depth()] = X
```

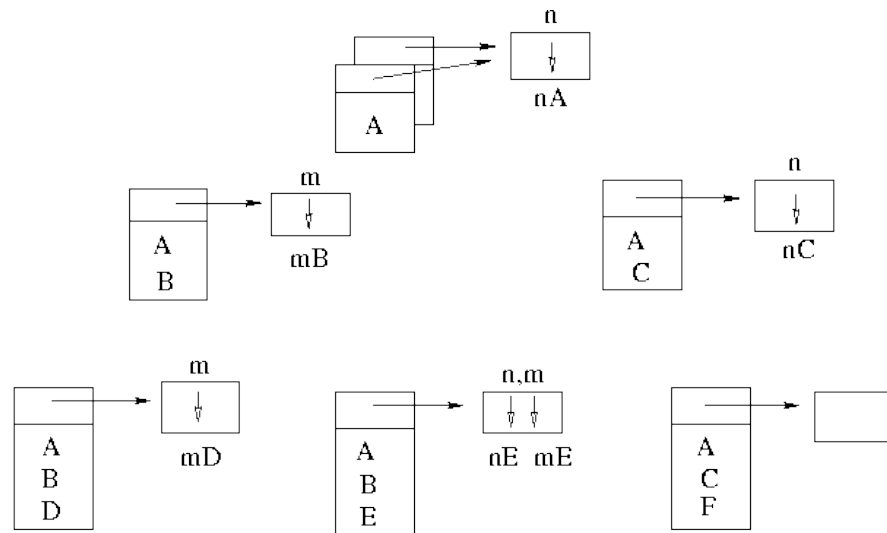
## Placement des champs

Invariant : Si  $SC < C$  alors le contenu d'un  $SC$  débute par le contenu d'un  $C$ .  
Ainsi tous les décalages sont conservés pour l'accès aux champs (dans le cas de l'héritage simple).

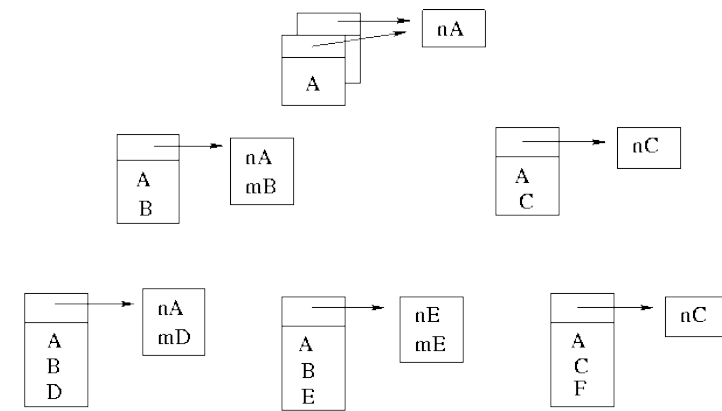
## Invocation de méthode



## À la Smalltalk

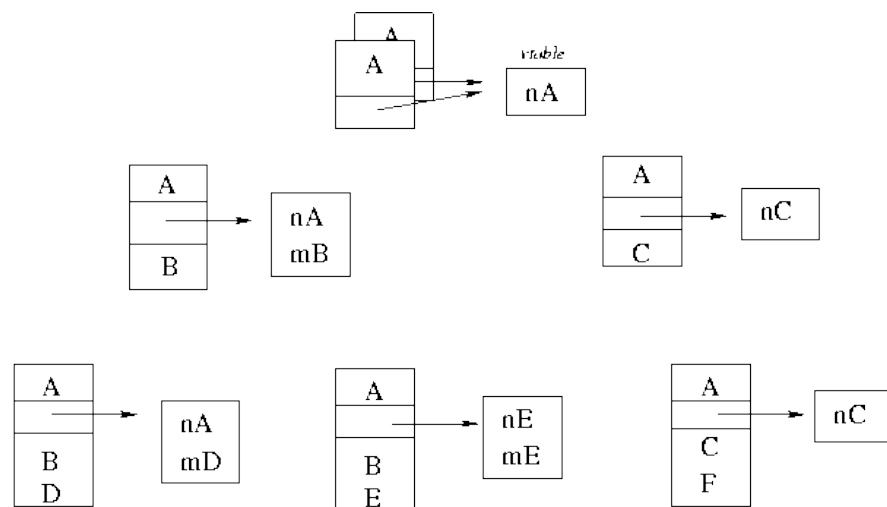


## À la Java

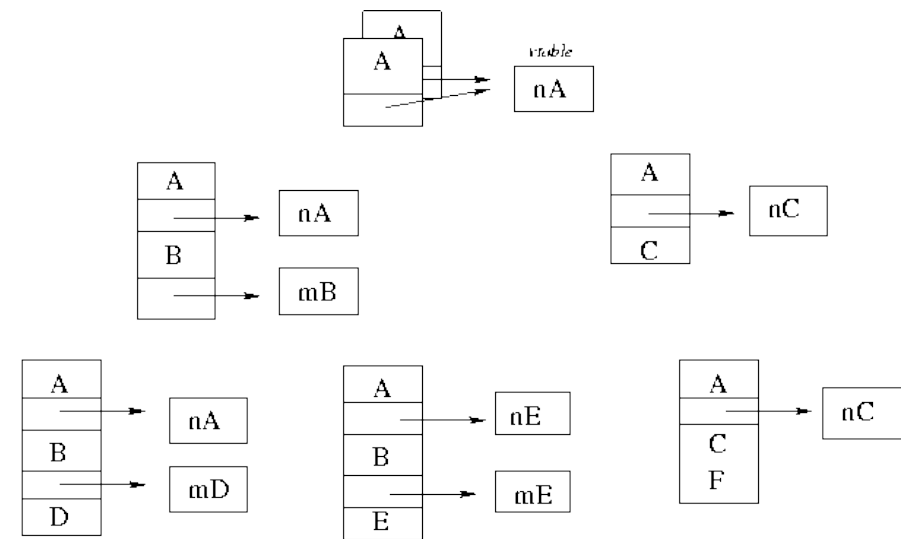


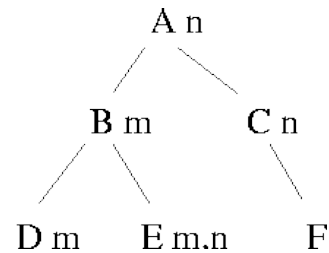
Si  $SC < C$  alors les méthodes possibles sur  $C$  forment un préfixe des méthodes possibles sur  $SC$  (en héritage simple). Possible car l'ensemble des méthodes est connu statiquement.

## À la C++



ou encore





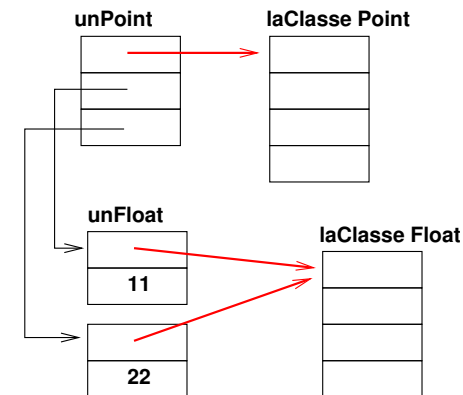
```

m : if ∈ B then (if ∈ D then mD
 elsif ∈ E then mE
 else mB)
else error
n : if ∈ C then nC elsif ∈ E then nE else nA

```

Ressource: C/ilpObj.h

Nouvelle représentation des objets en C et matérialisation du lien d'héritage



```

typedef struct ILP_Object {
 struct ILP_Class* _class;
 union {
 unsigned char asBoolean;
 int asInteger;
 double asFloat;
 struct asString {
 int _size;
 char asCharacter[1];
 } asString;
 struct asException {
 char message[ILP_EXCEPTION_BUFFER_LENGTH];
 struct ILP_Object* culprit[ILP_EXCEPTION_CULPRIT_LENGTH];
 } asException;
 struct asClass {
 struct ILP_Class* super;
 char* name;
 int fields_count;
 struct ILP_Field* last_field;
 int methods_count;
 ILP_general_function method[1];
 } asClass;
 struct asMethod {
 char* name;
 short arity;
 short index;
 } asMethod;
 struct asField {
 struct ILP_Class* defining_class;
 struct ILP_Field* previous_field;
 char* name;
 short offset;
 } asField;
 struct asInstance {
 struct ILP_Object* field[1];
 } asInstance;
 } _content;
} *ILP_Object;

```

```

#define ILP_AllocateInteger() \
 ILP_malloc(sizeof(struct ILP_Object), &ILP_object_Integer_class)

/** Constantes de classes. */

extern struct ILP_Class ILP_object_Object_class;
extern struct ILP_Class ILP_object_Class_class;
extern struct ILP_Class ILP_object_Method_class;
extern struct ILP_Class ILP_object_Field_class;
extern struct ILP_Class ILP_object_Integer_class;
extern struct ILP_Class ILP_object_Float_class;
extern struct ILP_Class ILP_object_Boolean_class;
extern struct ILP_Class ILP_object_String_class;
extern struct ILP_Class ILP_object_Exception_class;
extern struct ILP_Field ILP_object_super_field;
extern struct ILP_Field ILP_object_defining_class_field;
extern struct ILP_Method ILP_object_print_method;
extern struct ILP_Method ILP_object_classOf_method;

extern ILP_Object ILP_print (ILP_Object self);
extern ILP_Object ILP_classOf (ILP_Object self);

extern ILP_Object ILP_malloc (int size, ILP_Class class);
extern ILP_Object ILP_make_instance (ILP_Class class);
extern int /* boolean */ ILP_is_a (ILP_Object o, ILP_Class class);
extern ILP_general_function ILP_find_method (ILP_Object receiver,
 ILP_Method method,
 int argc);

```

## Hiérarchie des classes

Object

Class (super, name, fields\_count, last\_field)

Method (name, arity, index)

Field (defining\_class, previous\_field, name,

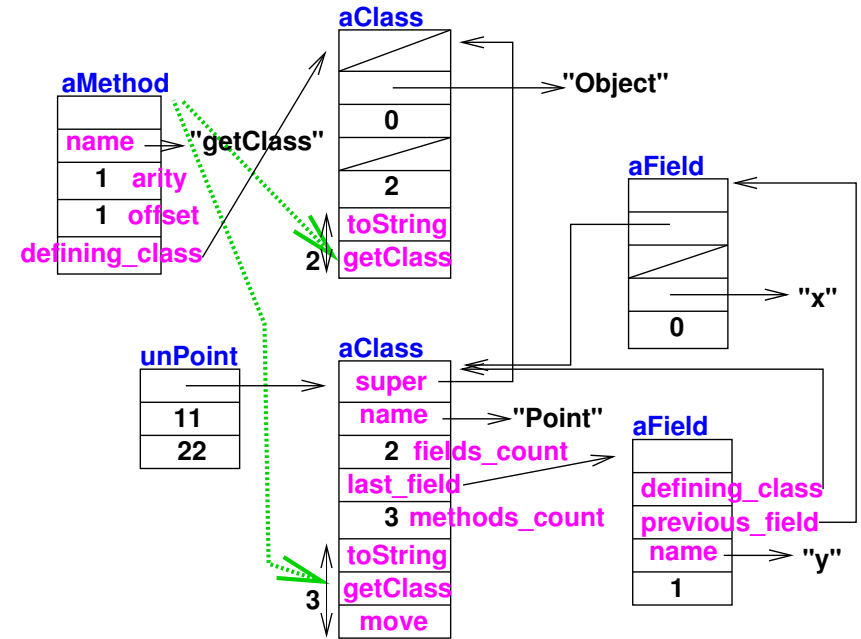
Integer

Float

Boolean

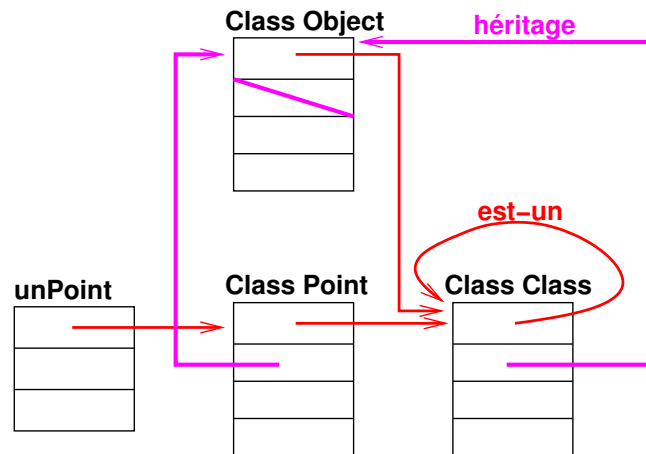
String

Exception



## Modèle ObjVlisp

## Accès aux champs



Compiler o.x

x donne (typage ou notre hypothèse additionnelle) **Point** qui donne le décalage. Efficacité ! On précalcule les décalages pour accéder aux champs mais on vérifie leur existence.

```

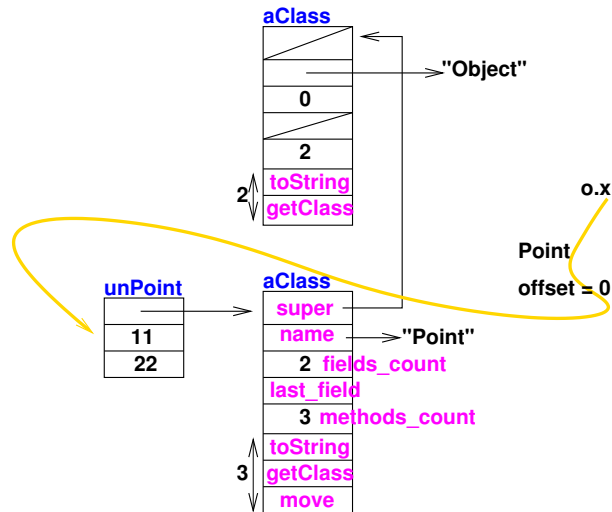
if (ILP_IsA(o, (ILP_Class)
 && ILP_object_Point_class)) {
 destination o->_content.asInstance.field[0];
} else {
 destination ILP_UnknownFieldError("x", o);
}

```

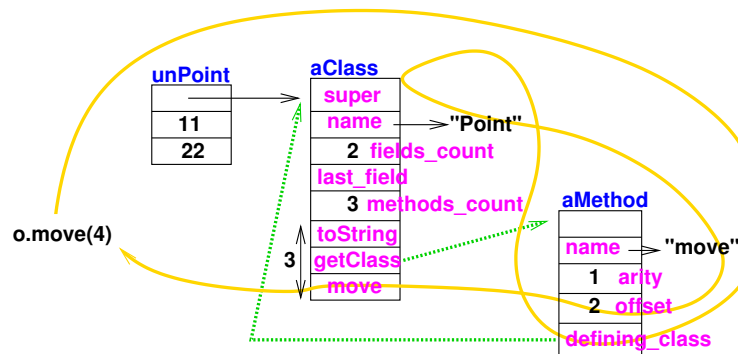
## Lien d'héritage

On peut faire plus efficace que cette recherche ascendante linéaire!

```
int /* boolean */
ILP_is_a (ILP_Object o, ILP_Class class)
{
 ILP_Class oclass = o->_class;
 if (oclass == class) {
 return 1;
 } else {
 oclass = oclass->_content.asClass.super;
 /* Object a NULL pour superclasse. */
 while (oclass) {
 if (oclass == class) {
 return 1;
 };
 oclass = oclass->_content.asClass.super;
 }
 return 0;
 }
}
```



## Envoi de méthodes



```
ilpTMP808 = ILP_find_method(
 receveur,
 &ILP_object_move_method,
 1); /* nbr args */
destination ilpTMP808(receveur, argument);
```

```
ILP_general_function
ILP_find_method (ILP_Object receiver,
 ILP_Method method,
 int argc)
{
 ILP_Class oclass = receiver->_class;
 if (! ILP_is_subclass_of(
 oclass,
 method->_content.asMethod.class_definin
 Signaler une absence de méthode
) else {
 int index = method->_content.asMethod.index;
 if (argc != method->_content.asMethod.arity)
 Signaler une erreur d'arite
 };
 return oclass->_content.asClass.method[index];
}
```

## Compilation de classe

```

ILP_GenerateClass(4);
extern struct ILP_Class4 ILP_object_Point_class;
extern struct ILP_Field ILP_object_x_field;
extern struct ILP_Field ILP_object_y_field;
extern struct ILP_Method ILP_object_longueur_method;
extern struct ILP_Method ILP_object_move_method;

```

```

struct ILP_Field ILP_object_x_field = {
 &ILP_object_Field_class,
 {(ILP_Class) & ILP_object_Point_class,
 NULL,
 "x",
 0}}
};

struct ILP_Field ILP_object_y_field = {
 &ILP_object_Field_class,
 {(ILP_Class) & ILP_object_Point_class,
 &ILP_object_x_field,
 "y",
 1}}
};

```

```

struct ILP_Class4 ILP_object_Point_class = {
 &ILP_object_Class_class,
 {(ILP_Class) & ILP_object_Object_class,
 "Point",
 2,
 &ILP_object_y_field,
 4,
 {ILP_print, /* print */
 ILP_classOf, /* classOf */
 ilpFUNC234, /* longueur */
 ilpFUNC235, /* move */
 }}}
};

```

```

struct ILP_Method ILP_object_longueur_method = {
 &ILP_object_Method_class,
 {(struct ILP_Class *) &ILP_object_Point_class,
 "longueur",
 1, /* arite */
 2 /* offset */
 }}
};

struct ILP_Method ILP_object_move_method = {
 &ILP_object_Method_class,
 {(struct ILP_Class *) &ILP_object_Point_class,
 "move",
 3, /* arite */
 3 /* offset */
 }}
};

```





## Méthodes compile6, eval6

```
// paquetage fr.upmc.ilp.ilp6.ast
CEAST6
 CEASTclassDefinition // implante IAST6classDefi
 CEASTmethodDefinition // implante IAST6methodDej
CEASTprogram // etend ilp4.ast.CEASTpro
CEAST6expression // etend ilp4.ast.CEASTexj
 CEASTinstantiate
 CEASTreadField
 CEASTwriteField
 CEASTself
 CEASTsuper
 CEASTsend
```

Toutes ces classes ont besoin de lire ou d'enrichir l'environnement des classes/méthodes. Or les `I*Environment` doivent être enrichis pour contenir cette nouvelle information en implantant :

```
public interface IClassEnvironment {
 public void addClassDefinition (IAST6classDefi
 public IAST6classDefinition findClassDefinitic
 throws RuntimeException;
}
```

Sont enrichis : `ICgenEnvironment`, `INormalizeGlobalEnvironment` et leurs implantations. Mais le code d'ILP4 ne sait pas fournir ces environnements enrichis aux nouvelles expressions d'ILP6.

```
// dans CEAST6expression
@Override
public void compile (
 StringBuffer buffer,
 ICgenLexicalEnvironment lexenv,
 fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment common,
 IDestination destination)
 throws CgenerationException {
 compile6(buffer,
 lexenv,
 CEAST6.narrowToICgenEnvironment(common),
 destination);
}
// NOTE: on aurait pu prendre le meme nom et
// laisser faire la surcharge:
public abstract void compile6 (
 StringBuffer buffer,
 ICgenLexicalEnvironment lexenv,
 ICgenEnvironment common,
 IDestination destination)
 throws CgenerationException;
```

## Visiteur

Il y a de nouvelles expressions :

```
public interface IAST6visitor extends IAST4visitor
 void visit (IAST6classDefinition classDefiniti
 void visit (CEASTinstantiate expression);
 void visit (CEASTsend expression);
 void visit (CEASTreadField expression);
 void visit (CEASTwriteField expression);
 void visit (CEASTself expression);
 void visit (CEASTsuper expression);
 // NOTE: utiliser plutot des interfaces.
}
```

On peut, dans ILP4, fournir un `IAST6visitor` à la place d'un `IAST4visitor` mais quand ILP4 visite un nœud ILP6, il doit lui fournir un `IAST6visitor`.

## Suggestions

On ne peut donc simplement écrire, dans une classe d'ILP6 :

```
//public void accept (IAST4visitor visitor) {
// FAUX
// visitor.visit(this);
// FAUX
//}
```

Il faut alors écrire :

```
public void accept (IAST6visitor visitor) {
 visitor.visit(this);
}
public void accept (IAST4visitor visitor) {
 CEAST6.narrowToIAST6visitor(visitor).visit(thi
}
```

Non nécessaire si l'on écrit son discriminant (comme en `ilp1.cgen.analyse`)

- lire le code d'ILP6
- et regarder le code C engendré