Revision: 1.11

Master d'informatique 2007-2008
Spécialité STL
« Implantation de langages »
ILP – MI016
Cours 8

C.Queinnec[a]

[a] http://www-spi.lip6.fr/~queinnec/

PLAN DU COURS 8

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 }
}
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
}
```

EXEMPLES SYNTAXIQUES

```
class Point { // extends Object // Ordre des définitions
  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());
```

INTERFACES

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

AST

```
// paquetage fr.upmc.ilp.ilp6.ast
CEAST6
  CEASTclassDefinition  // implante IAST6classDefinition
  CEASTmethodDefinition // implante IAST6methodDefinition
CEASTprogram            // étend ilp4.ast.CEASTprogram
CEAST6expression        // étend ilp4.ast.CEASTexpression
  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 vérifier 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 $C_1 \subseteq C_2 \vee C_2 \subseteq C_1$.

- méthode
  - appartenance à une classe **dynamique**
  - moi **dynamique**
  - méthode héritée **statique**

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

- appel de méthode **dynamique**

MODÈLE DE DONNÉES

nom -> Classe *interprétation ou compilation*

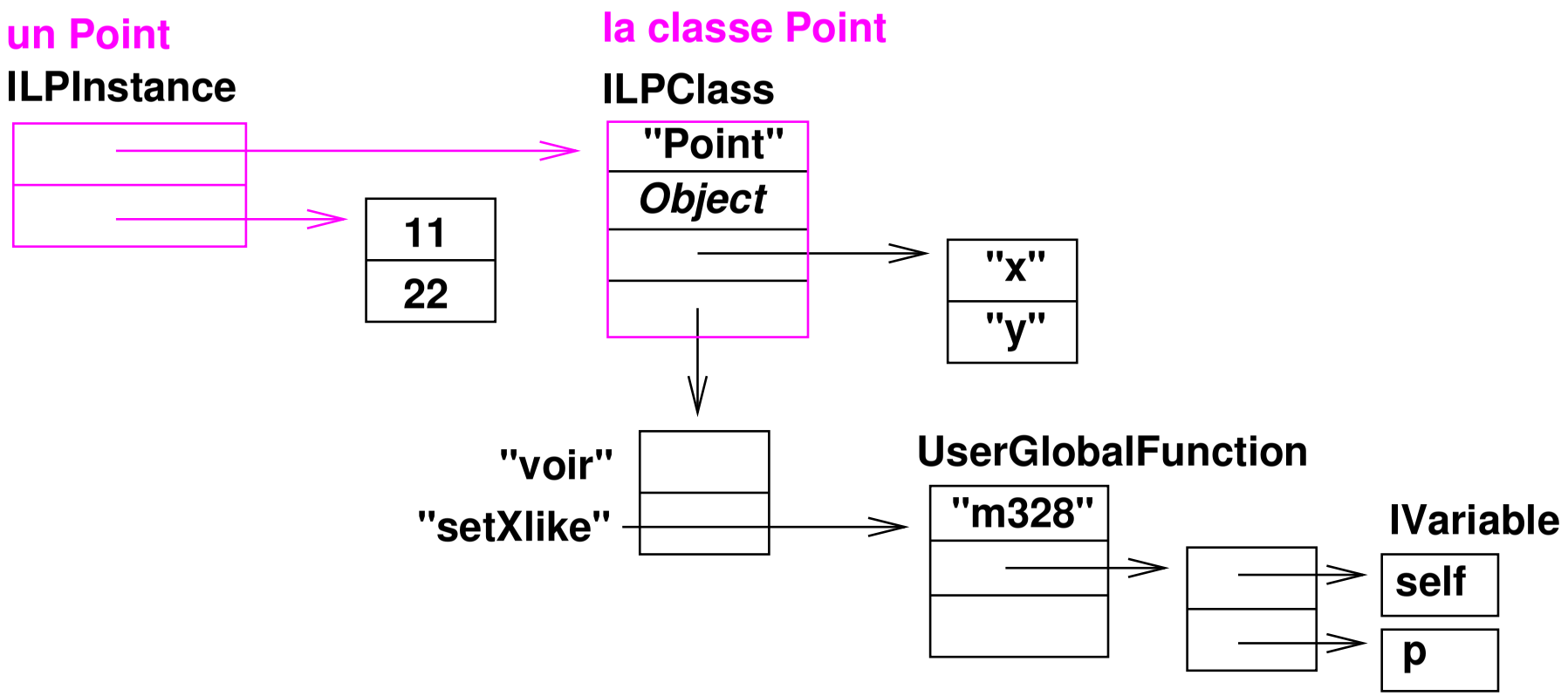
```
Classe      nom
            nom superclasse
               champs hérités
               méthodes héritées
            champs propres (liste de noms)
            méthodes propres (liste de fonctions globales)
Instance    classe
            champs (nom -> valeur)
Méthode     fonction globale
                 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 hérités
Map      method      propres seulement

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

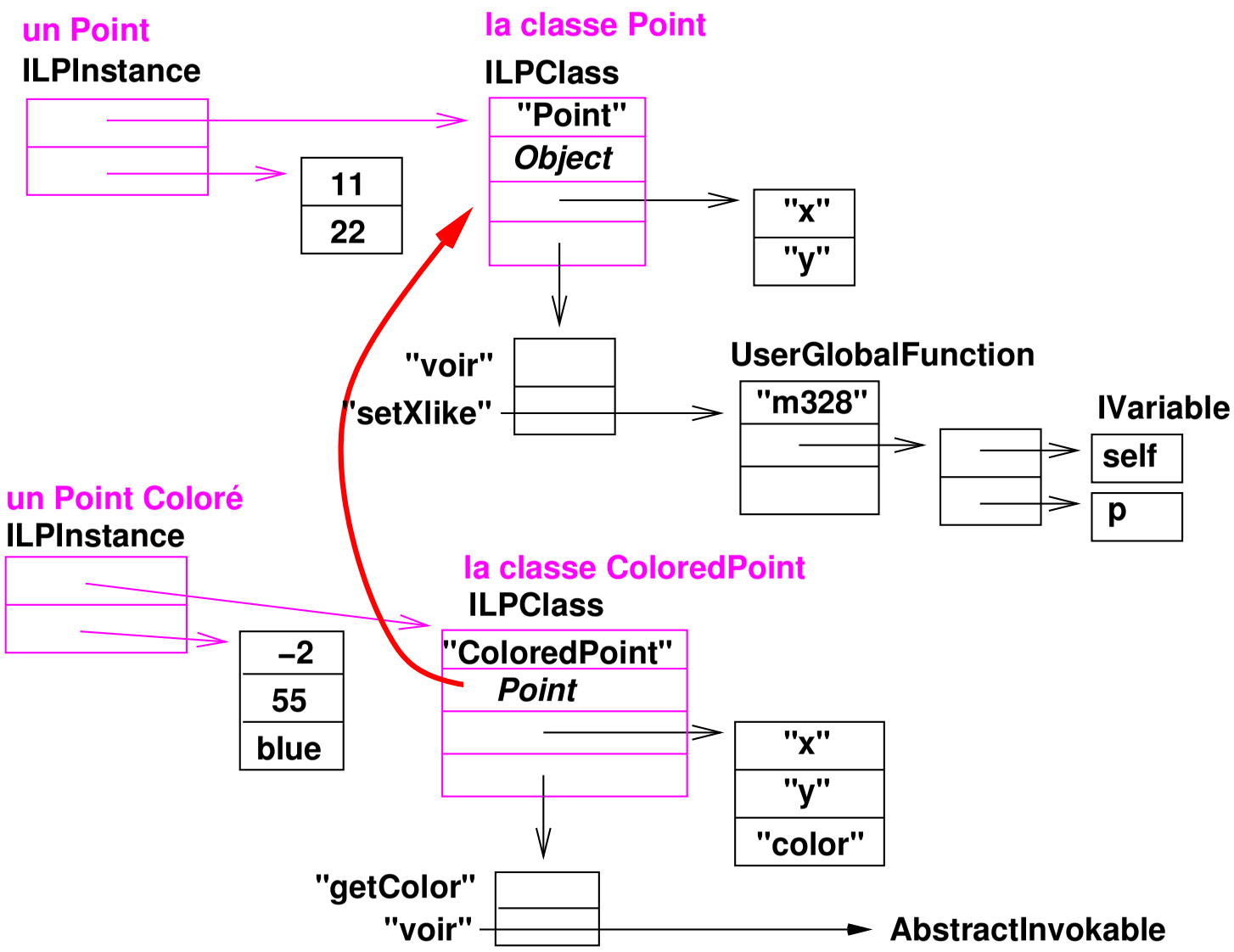

un Point
ILPInstance
11
22
la classe Point
ILPClass
"Point"
Object
"x"
"y"
"voir"
"setXlike"
UserGlobalFunction
"m328"
IVariable
self
p

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`

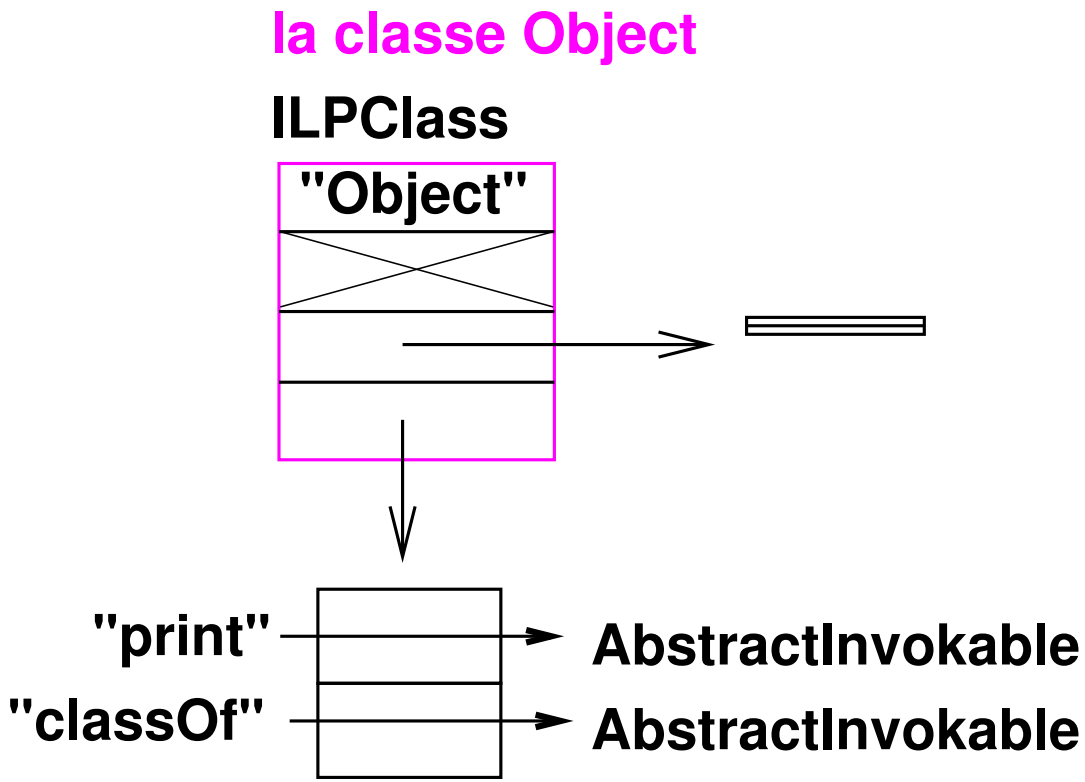
```
ILPClass clazz;
Object[] fields;

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


un Point
ILPInstance
11
22
la classe Point
ILPClass
"Point"
Object
"x"
"y"
"voir"
"setXlike"
UserGlobalFunction
"m328"
IVariable
self
p
un Point Coloré
ILPInstance
–2
55
blue
la classe ColoredPoint
ILPClass
"ColoredPoint"
Point
"x"
"y"
"color"
"getColor"
"voir"
AbstractInvokable

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 (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    final ILPClass clazz = common.findClass(className);
    final 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

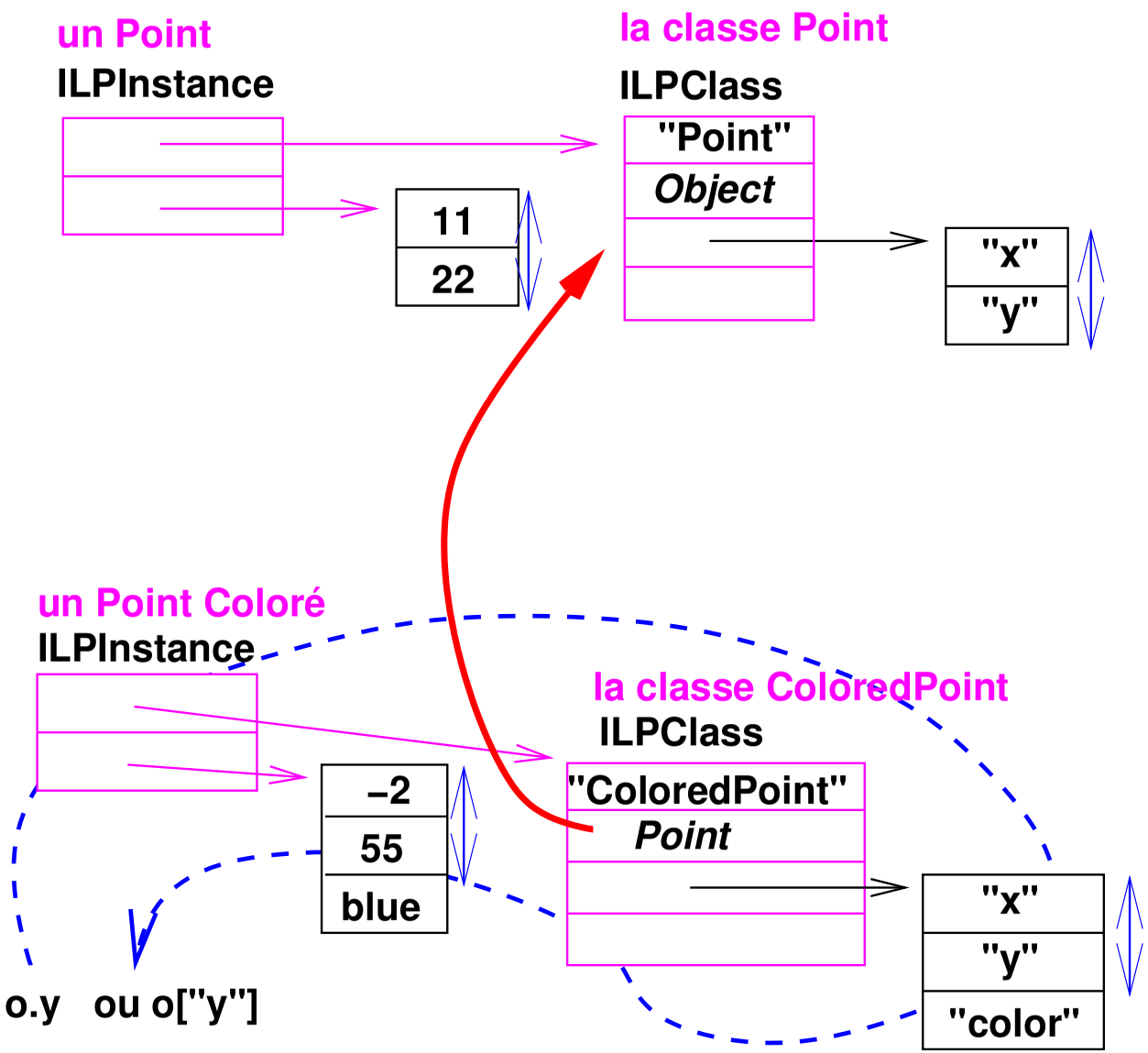
Les noms des champs sont convertis en des décalages.

```
public class ILPInstance {
  public ILPInstance (final ILPClass clazz, final 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);
  }
```

ACCÈS AUX CHAMPS

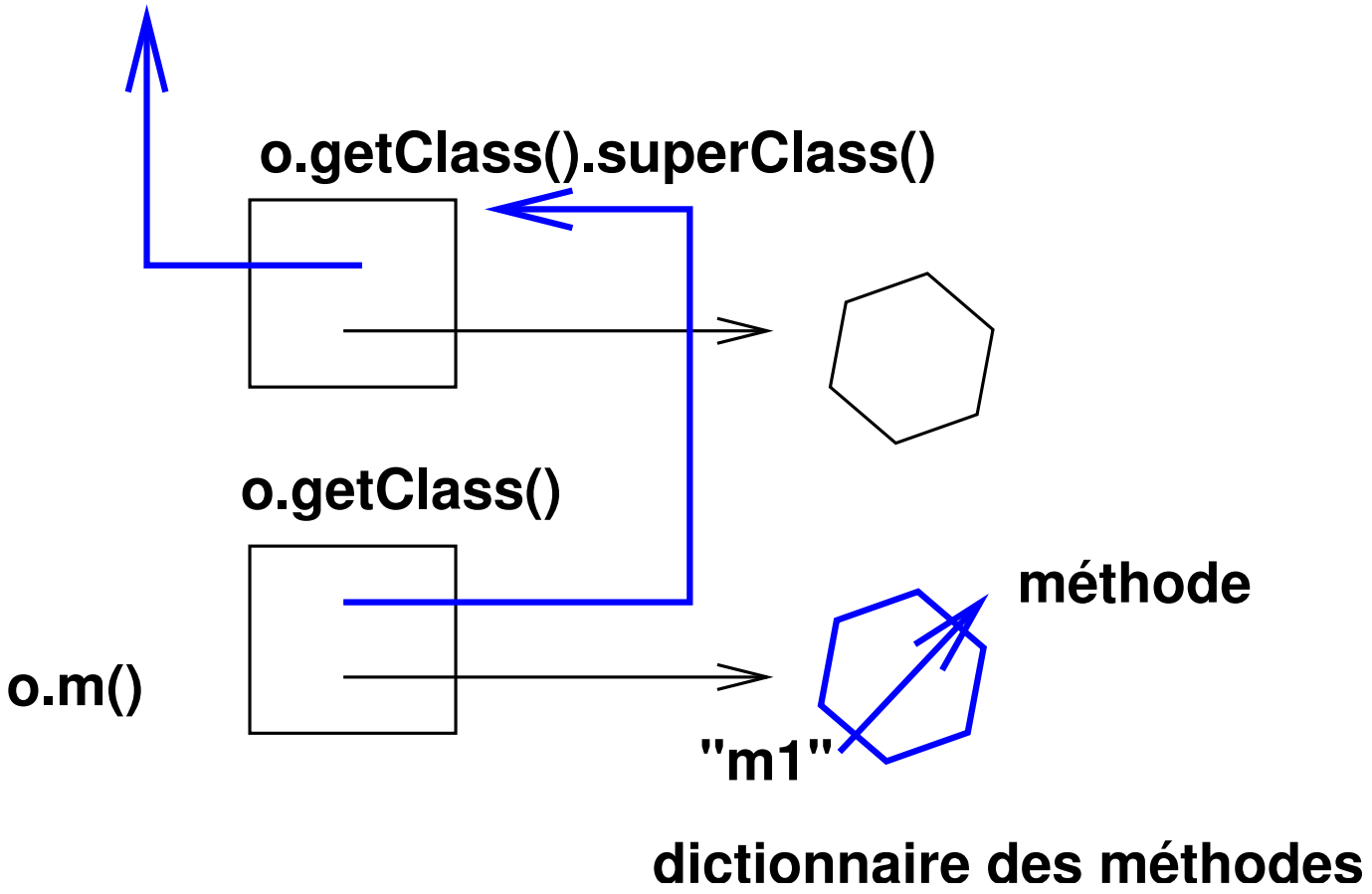
```
public class CEASTwriteField extends CEASTExpression {
  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    final Object target = object.eval(lexenv, common);
    final Object newValue = value.eval(lexenv, common);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


un Point
ILPInstance
11
22
la classe Point
ILPClass
"Point"
Object
"x"
"y"
un Point Coloré
ILPInstance
–2
55
blue
la classe ColoredPoint
ILPClass
"ColoredPoint"
Point
"x"
"y"
"color"
o.y ou o["y"]

ENVOI DE MESSAGE

À la Smalltalk (*lookup*) :

```
public class CEASTsend extends CEASTExpression {
  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    final Object target = receiver.eval(lexenv, common);
    final 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);
    }
  }
```


o.getClass().superClass()
o.getClass()
o.m()
méthode
"m1"
dictionnaire des méthodes

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 méthode 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 (final ILexicalEnvironment lexenv,
                        final ICommon common)
    throws EvaluationException {
        return lexenv.lookup(this);
    }

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 differents.
        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;
```

SUPER

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 (final ILexicalEnvironment lexenv,
                         final ICommon common)
    throws EvaluationException {
        ILPmethod currentMethod = (ILPmethod) lexenv.lookup(ILPmethod.cmv);
        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;
    }

    public Object callSuper (final ILexicalEnvironment lexenv,
                             final ICommon common )
    throws EvaluationException {
        Object[] arguments = (Object[]) lexenv.lookup(cmargs);
        return getDefiningClass().getSuperClass()
            .send(getMethodName(), arguments, common);
    }
```

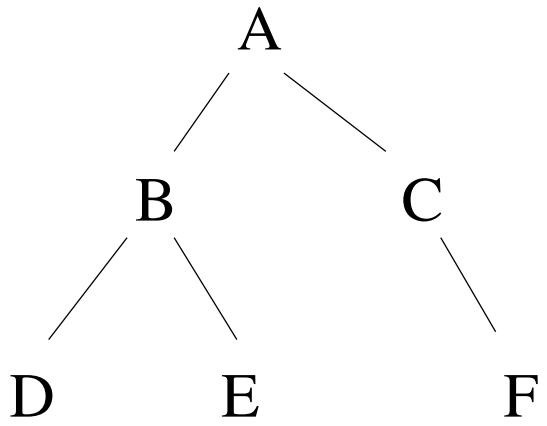
```
@Override
public Object invoke (final Object[] arguments,
                      final ICommon common)
  throws EvaluationException {
    IAST2variable[] variables = getVariables();
    if ( variables.length != arguments.length ) {
        final String msg = "Wrong arity";
        throw new EvaluationException(msg);
    };
    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);
}
```

COMPILATION

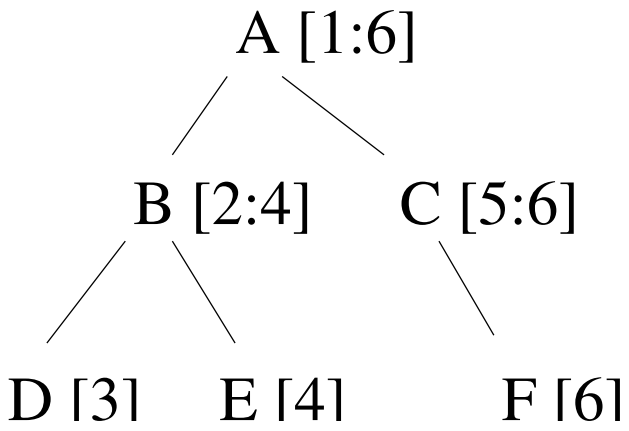
- pas d'objet en C
- pas de classe mais des `struct`
- accès aux champs statique
- déjà une solution pour `instanceof`
- pointeurs
- fonctions

HÉRITAGE SIMPLE

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

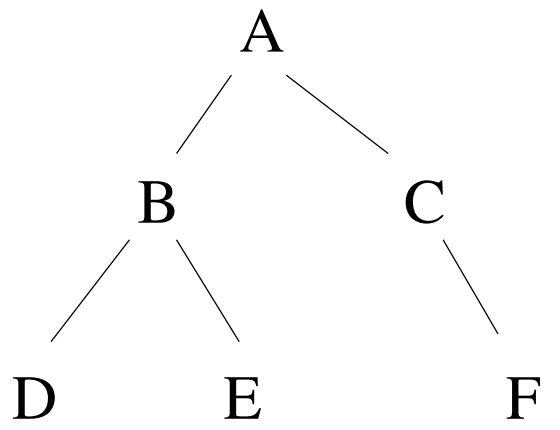


NUMÉROTATION PRÉFIXE




```
X.min ≤ o.class().min ≤ X.max
```

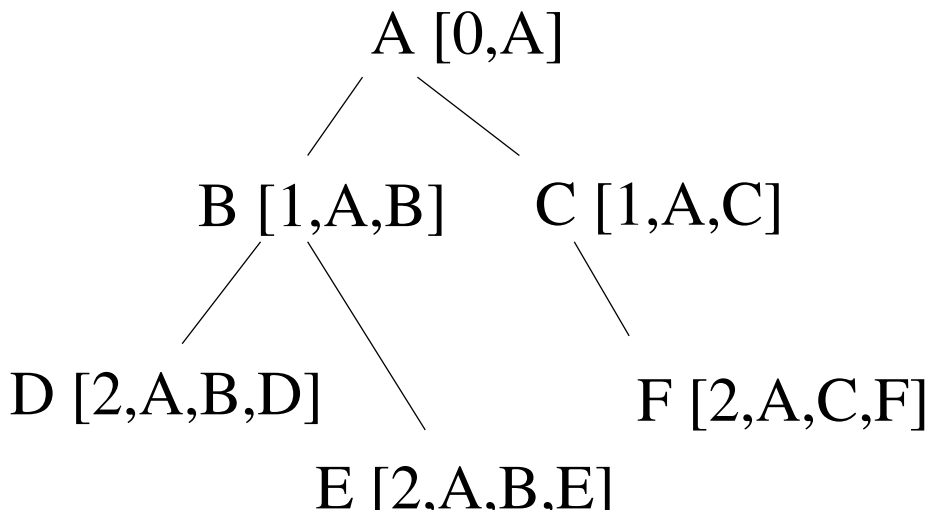
MATRICE CARACTÉRISTIQUE



	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



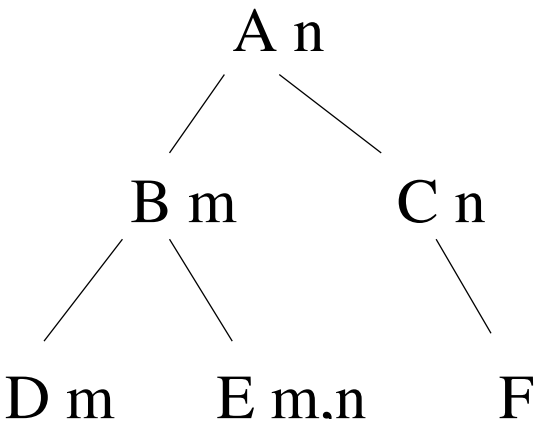

```
   X.depth() <= o.class().depth()
&& o.class().supers[X.depth()] = X
```

PLACEMENT DES CHAMPS

Invariant : Si $SC < C$ alors le contenu d'un SC débute par le contenu d'un C.

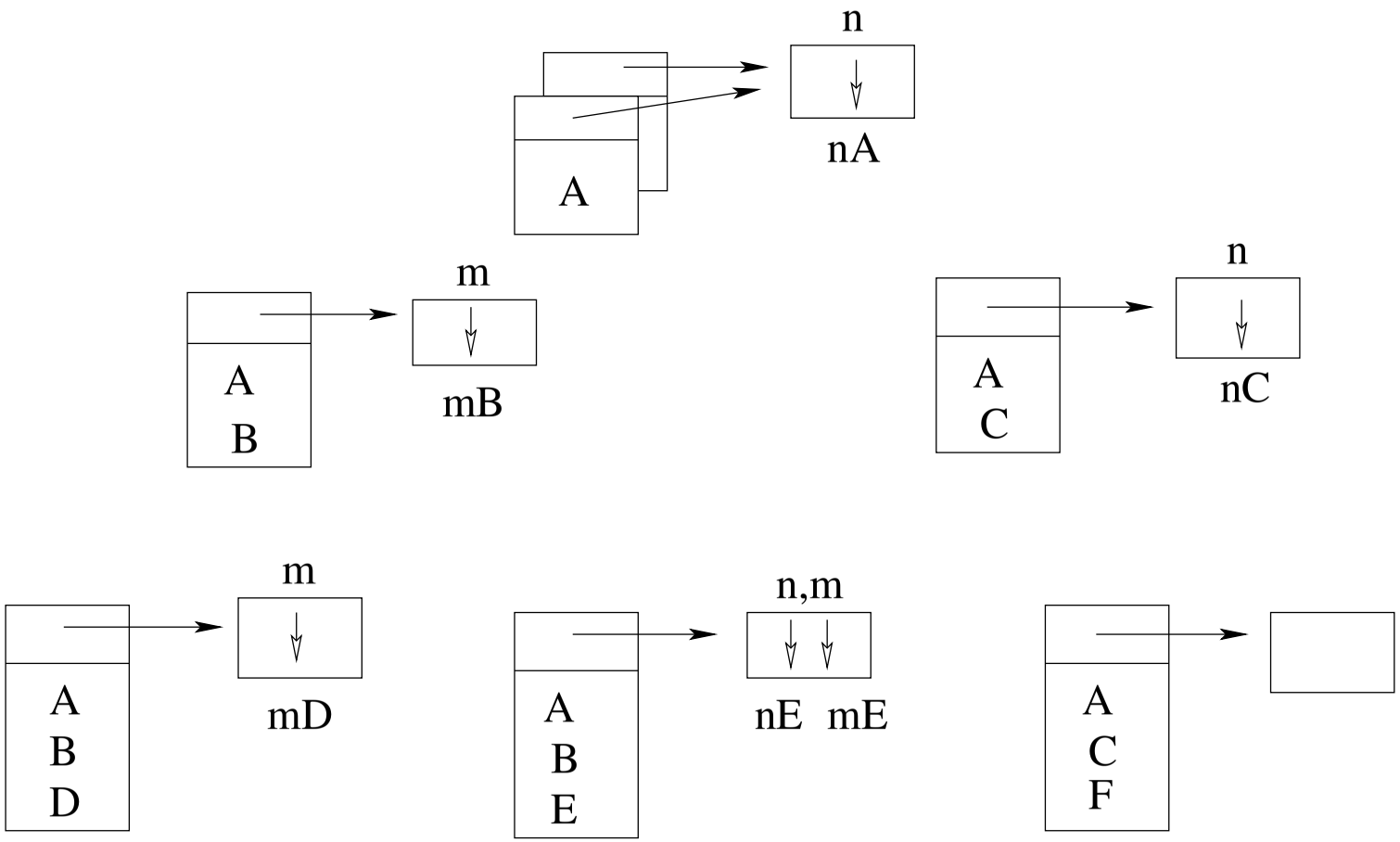
Ainsi tous les décalages sont conservés pour l'accès aux champs (dans le cas de l'héritage simple).

INVOCATION DE MÉTHODE

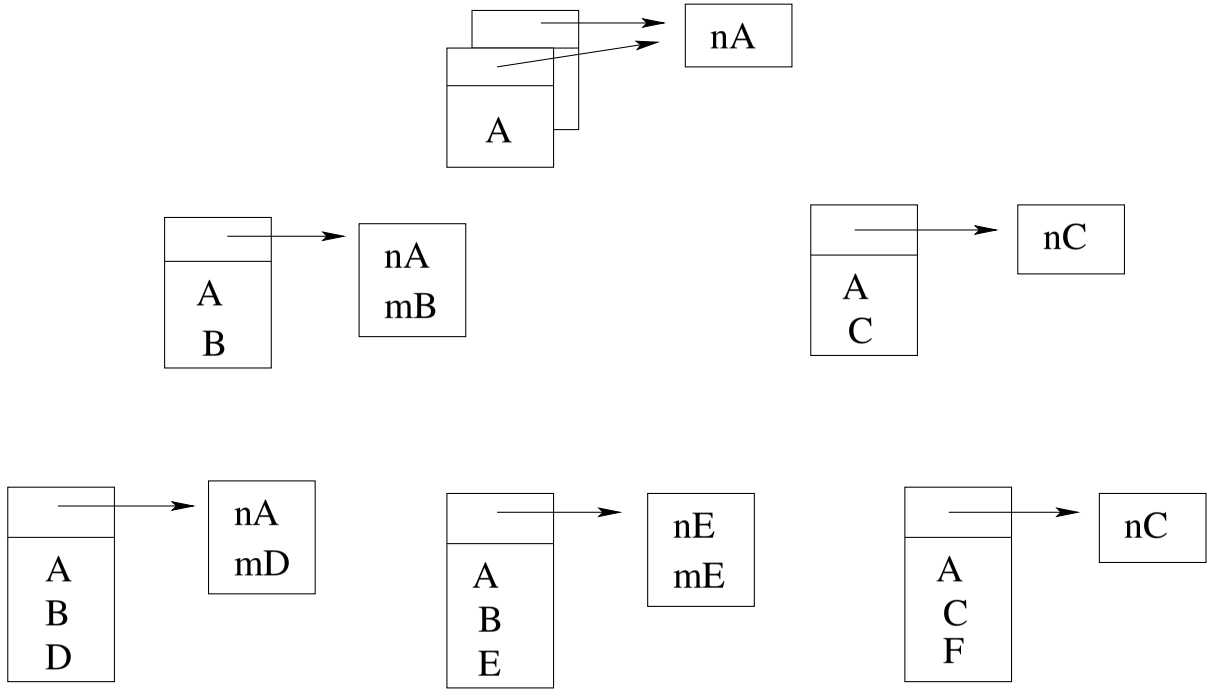


	m	**n**
A	error	nA
B	mB	nA
C	error	nC
D	mD	nA
E	mE	nE
F	error	nC

À LA SMALLTALK

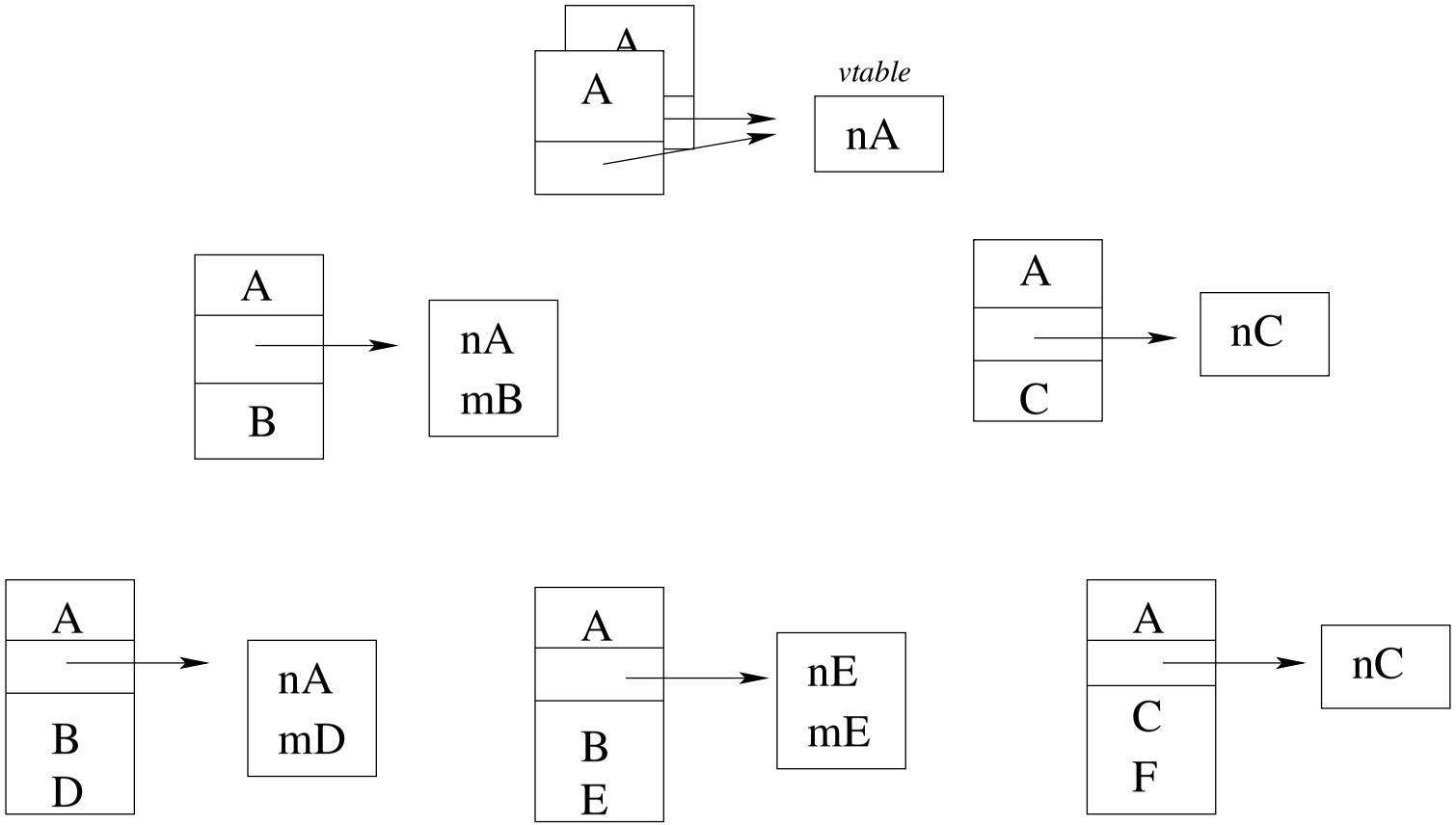


À LA JAVA

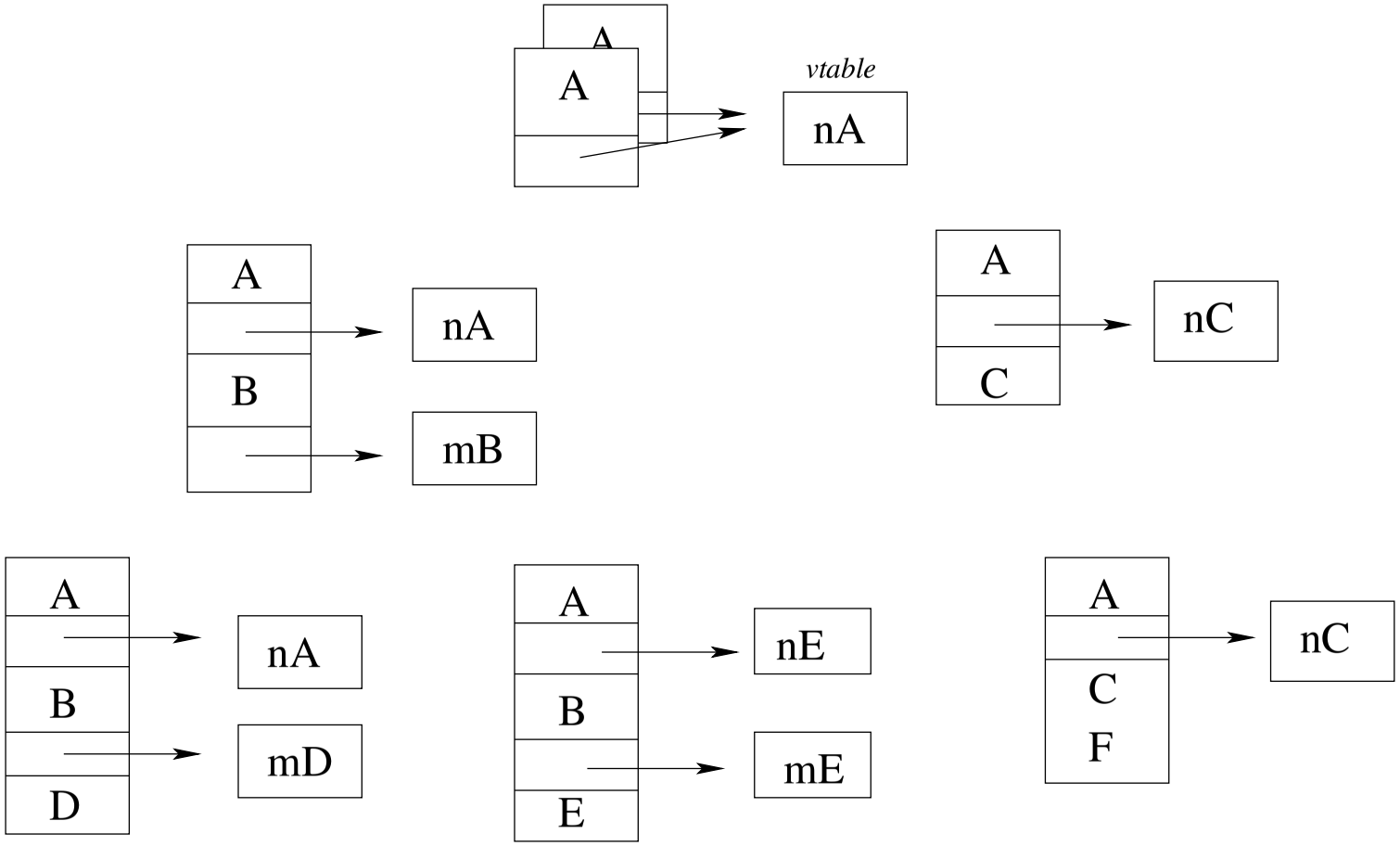


Si $SC < C$ alors les méthodes possibles sur C forment un préfixe des méthodes possibles sur SC (en héritage simple). Possible car l'ensemble des méthodes est connu statiquement.

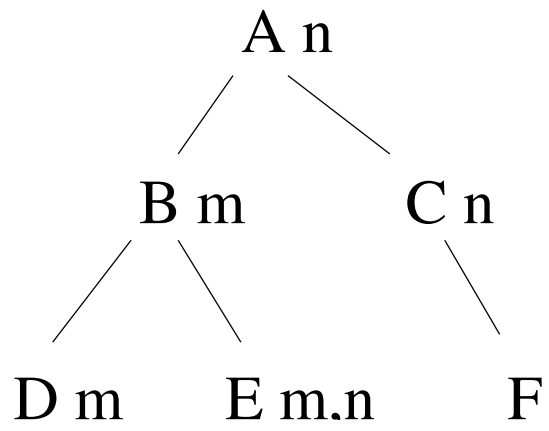
À LA C++



ou encore



ARBRE DE DÉCISION

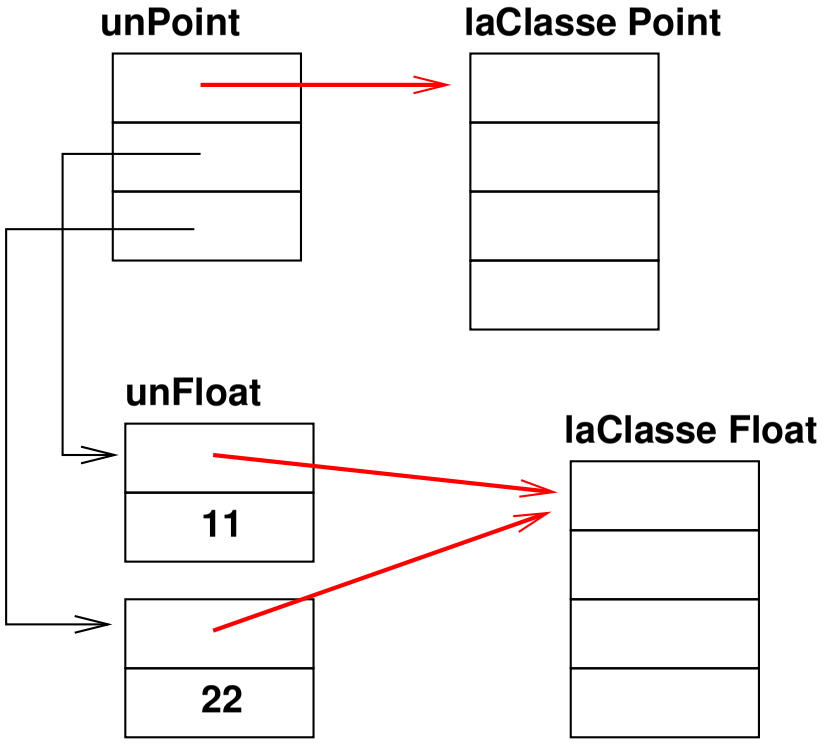



```
m : if ∈ B then (if ∈ D then mD
                 elsif ∈ E then mE
                 else mB)
    else error
n: if ∈ C then nC elsif ∈ E then nE else nA
```

COMPILATION

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;
```

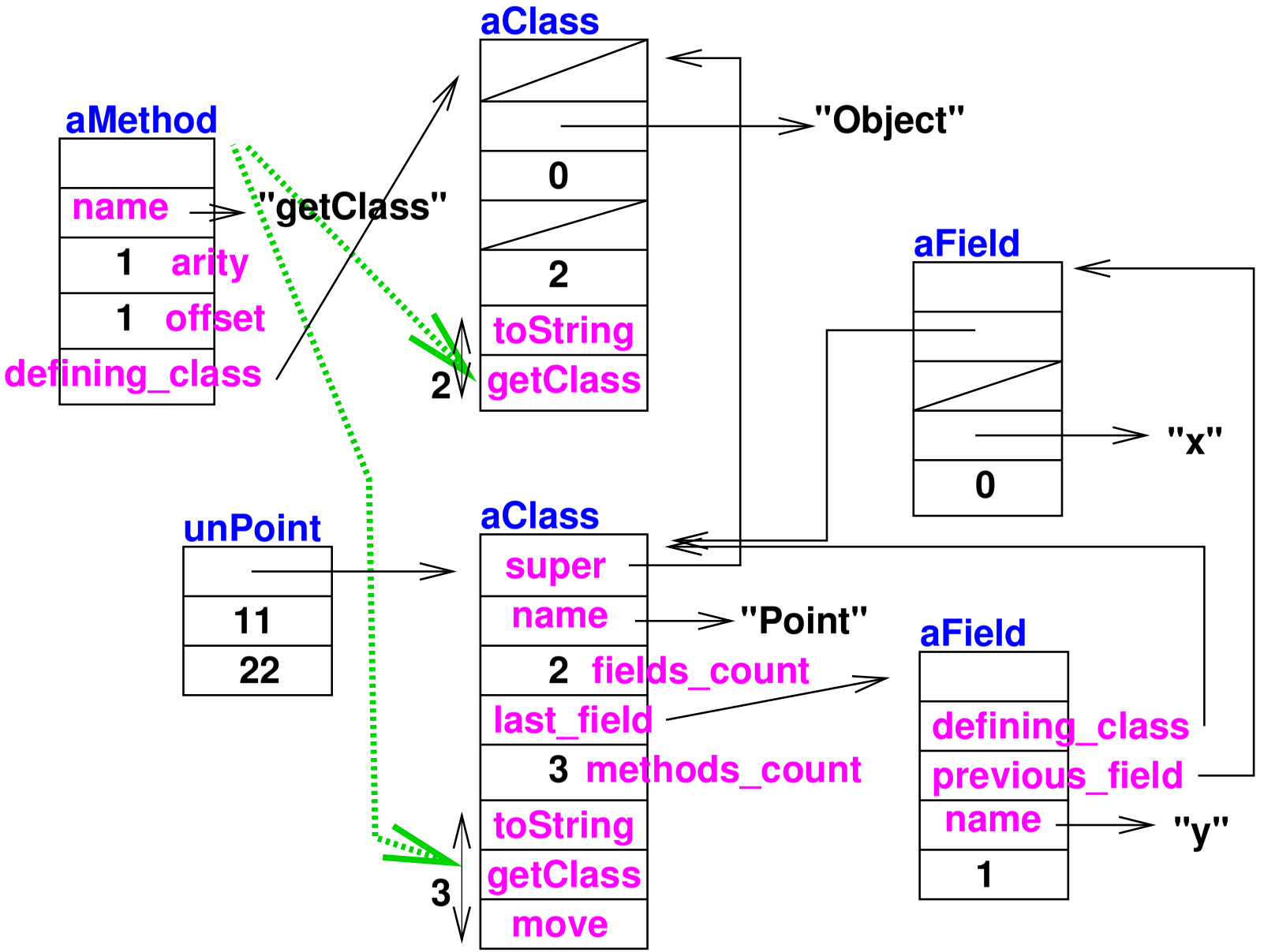
```
            ILP_general_function method[1];
        } asClass;
        struct asMethod {
            char*               name;
            short               arity;
            short               index;
        } asMethod;
        struct asField {
            struct ILP_Class*   defining_class;
            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);
```

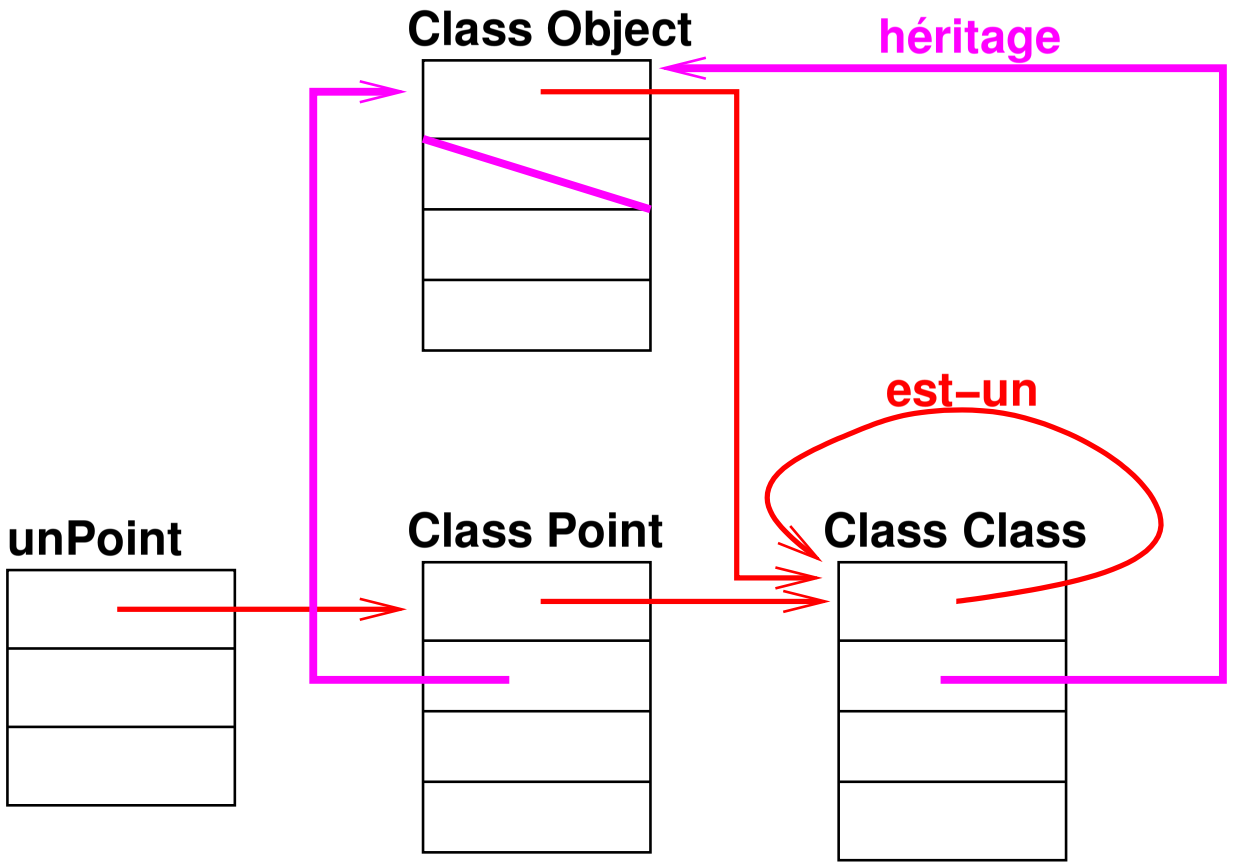
```
extern ILP_Object ILP_malloc (int size, ILP_Class class);
extern ILP_Object ILP_make_instance (ILP_Class class);
extern int /* boolean */ ILP_is_a (ILP_Object o, ILP_Class class);
extern ILP_general_function ILP_find_method (ILP_Object receiver,
                                             ILP_Method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

HIÉRARCHIE DES CLASSES

```
Object
   Class        (super, name, fields_count, last_field, methods_count, .
   Method       (name, arity, index)
   Field        (defining_class, previous_field, name, offset)
   Integer
   Float
   Boolean
   String
   Exception
```


aClass
"Object"
0
aMethod
name
"getClass"
1 arity
1 offset
defining_class
2
toString
getClass
aField
"x"
0
unPoint
11
22
aClass
super
name
"Point"
2 fields_count
last_field
3 methods_count
toString
getClass
move
3
aField
defining_class
previous_field
name
"y"
1

MODÈLE OBJVLISP



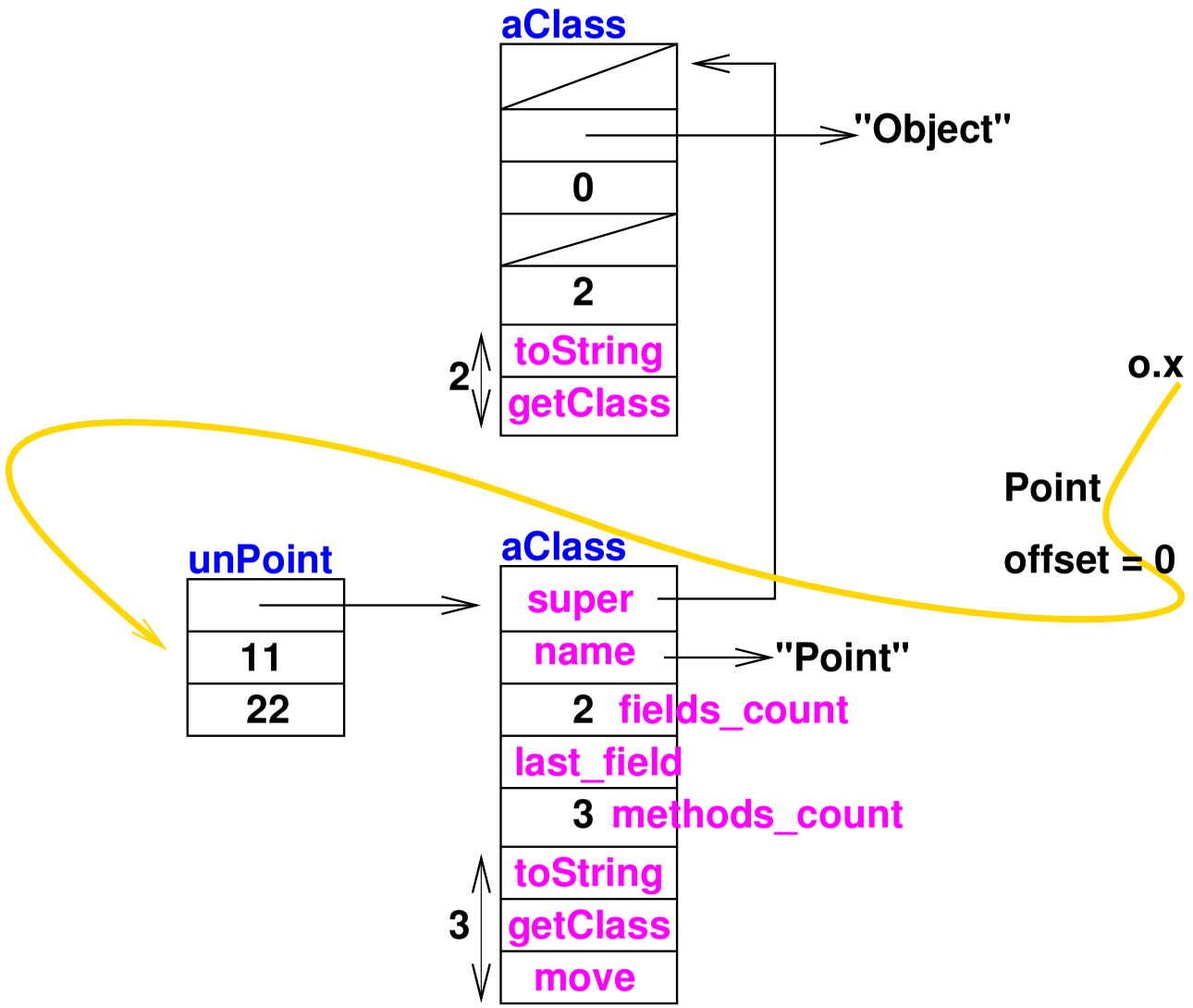
ACCÈS AUX CHAMPS

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);
}
```


aClass
"Object"
0
2
2
toString
getClass
o.x
Point
offset = 0
unPoint
aClass
11
22
super
name
"Point"
2 fields_count
last_field
3 methods_count
3
toString
getClass
move

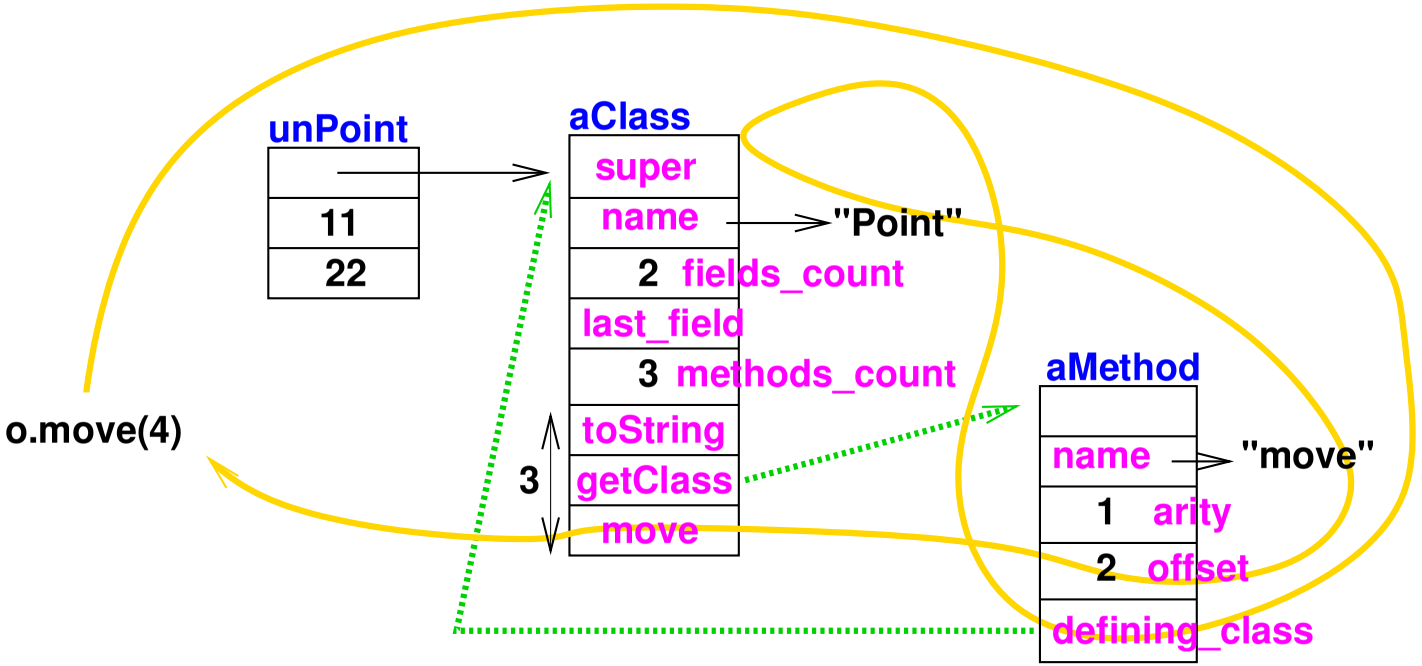
LIEN D'HÉRITAGE

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_defining) ) {
            Signaler une absence de méthode
      } else {
        int index = method->_content.asMethod.index;
        if ( argc != method->_content.asMethod.arity ) {
              Signaler une erreur d'arité
        };
        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,                          /* arité */
    2                           /* offset */
    }}
};

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

SUPER

On précalcule (statiquement) quelle est la super-méthode. La compilation d'une méthode stocke la méthode courante dans l'environnement lexical.

```
// CEASTsuper
   public void compile6 (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final IDestination destination)
    throws CgenerationException {
        IAST6methodDefinition currentMethod = null;
        ICgenLexicalEnvironment le = lexenv;
        while ( ! le.isEmpty() ) {
            if ( le instanceof CgenMethodLexicalEnvironment ) {
                CgenMethodLexicalEnvironment cmle =
                    (CgenMethodLexicalEnvironment) le;
                currentMethod = cmle.getMethodDefinition();
                break;
            } else {
                le = le.getNext();
            }
        }
```

```
        if ( currentMethod != null ) {
            destination.compile(buffer, lexenv, common);
            buffer.append("ILP_FindAndCallSuperMethod(");
            buffer.append(currentMethod.getRealArity());
            buffer.append(");");
        } else {
            final String msg = "No supermethod!";
            throw new CgenerationException(msg);
        }
    }
```

Un petit exemple :

```
/* class Point2D extends Point                                  */
/*   method print (x) { print "print@Point2D"; super() }   */
ILP_Object
ilpMETHOD_5f80_3222(ILP_Object ilp_Self, ILP_Object x)
{
  static ILP_Method ilp_CurrentMethod = &ILP_object_print_method;
  static ILP_general_function ilp_SuperMethod = ILP_print;
  ILP_Object ilp_CurrentArguments[2];
  ilp_CurrentArguments[1] = x;

  ILP_Object ilpLOCAL_3236;
  ilpLOCAL_3236 = ILP_String2ILP("print@Point2D");
  (void) ILP_print(ilpLOCAL_3236);
  return ILP_FindAndCallSuperMethod(1);
}
```

Ajout à la bibliothèque d'exécution en C :

```
#define ILP_FindAndCallSuperMethod(i) \
  (((ilp_SuperMethod != NULL) \
    ? (*ILP_find_and_call_super_method##i) \
    : (*ILP_dont_call_super_method) )( \
   ilp_Self, ilp_CurrentMethod, ilp_SuperMethod, ilp_CurrentArguments))
extern ILP_Object ILP_find_and_call_super_method1(
     ILP_Object self,
     ILP_Method current_method,
     ILP_general_function super_method,
     ILP_Object arguments[] );
#define DefineSuperMethodCaller(i) \
ILP_Object \
ILP_find_and_call_super_method##i ( \
     ILP_Object self, \
     ILP_Method current_method, \
     ILP_general_function super_method, \
     ILP_Object arguments[1] ) \
{ \
```

```
/* assert( super_method != NULL ); */ \
switch ( i ) { \
     case 0: { \
          return (*super_method)(self); \
     } \
     case 1: { \
          return (*super_method)(self, arguments[1]); \
     } \
     case 2: { \
          return (*super_method)(self, arguments[1], arguments[2]); \
     } \
     case 3: { \
          return (*super_method)(self, arguments[1],  \
                                       arguments[2],  \
                                       arguments[3]); \
     } \
     default: { \
          snprintf(ILP_the_exception._content.asException.message, \
                   ILP_EXCEPTION_BUFFER_LENGTH, \
                   "Cannot invoke supermethod %s\nCulprit: 0x%p\n", \
```

```
                        current_method->_content.asMethod.name,  \
                        (void*) self ); \
                /*DEBUG*/ \
                fprintf(stderr, ILP_the_exception._content.asException.message);\
                ILP_the_exception._content.asException.culprit[0] = self; \
                ILP_the_exception._content.asException.culprit[1] =  \
                     (ILP_Object) current_method; \
                ILP_the_exception._content.asException.culprit[2] = NULL; \
                ILP_throw((ILP_Object) &ILP_the_exception); \
                /* UNREACHED */ \
                return NULL; \
          } \
     } \
}
DefineSuperMethodCaller(0)
DefineSuperMethodCaller(1)
DefineSuperMethodCaller(2)
DefineSuperMethodCaller(3)

ILP_Object
```

```
ILP_dont_call_super_method (
      ILP_Object self,
      ILP_Method current_method,
      ILP_general_function super_method,
      ILP_Object arguments[1] )
{
      /* assert ( super_method == NULL ); */
      snprintf(ILP_the_exception._content.asException.message,
               ILP_EXCEPTION_BUFFER_LENGTH,
               "No supermethod %s\nCulprit: 0x%p\n",
               current_method->_content.asMethod.name,
               (void*) self );
      /*DEBUG*/
      fprintf(stderr, ILP_the_exception._content.asException.message);
      ILP_the_exception._content.asException.culprit[0] = self;
      ILP_the_exception._content.asException.culprit[1] =
           (ILP_Object) current_method;
      ILP_the_exception._content.asException.culprit[2] = NULL;
      ILP_throw((ILP_Object) &ILP_the_exception);
      /* UNREACHED */
```

```
    return NULL;
}
```

NOUVEAUTÉS JAVA

- extension des environnements
- extension du visiteur

MÉTHODES COMPILE6, EVAL6

```
// paquetage fr.upmc.ilp.ilp6.ast
CEAST6
  CEASTclassDefinition  // implante IAST6classDefinition
  CEASTmethodDefinition // implante IAST6methodDefinition
CEASTprogram            // étend ilp4.ast.CEASTprogram
CEAST6expression        // étend ilp4.ast.CEASTexpression
  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 (IAST6classDefinition cd);
    public IAST6classDefinition findClassDefinition (String className)
      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 (final StringBuffer buffer,
                     final ICgenLexicalEnvironment lexenv,
                     final fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment common,
                     final IDestination destination)
  throws CgenerationException {
    compile6(buffer,
             lexenv,
             CEAST6.narrowToICgenEnvironment(common),
             destination );
}
//  NOTE : on aurait pu prendre le même nom et laisser faire la surcharge :
public abstract void compile6 (final StringBuffer buffer,
                               final ICgenLexicalEnvironment lexenv, /*ilp6*/
                               final ICgenEnvironment common,
                               final IDestination destination )
throws CgenerationException;
```

VISITEUR

Il y a de nouvelles expressions :

```
public interface IAST6visitor extends IAST4visitor {
    void visit (IAST6classDefinition classDefinition);
    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.

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(this);
}
```

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

SUGGESTIONS

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