Les objets

MU4IN501 – DLP : Développement d'un langage de programmation
Master STL, Sorbonne Université

Antoine Miné

Année 2020–2021

Cours 8 & 9

Plan général

Développement d'un langage de programmation : interprète et compilateur.

Par étapes, avec ajout progressif de fonctionnalités.

- **ILP1** : langage de base

  constantes (entiers, flottants, booléens, chaînes),
  opérateurs ($+$, $-$, $\times$, ...),
  appels de primitives (`print`, ...),
  blocs locaux (`let x = ... in ...`),
  alternatives (`if ... then ... else`)

- **ILP2** : ajout des boucles, affectations, fonctions globales
- **ILP3** : ajout des exception et des fonctions de première classe
- **ILP4** : ajout des classes et des objets

Les objets

Principes généraux

Objets : structures regroupant

- des données : les champs (ou attributs, ou propriétés) ;
- du code : les méthodes.

Encapsulation :

- les méthodes d'un objet ont un accès privilégié à ses champs ;
- l'accès aux champs des autres objets est contrôlé (voire interdit) ;

$\Longrightarrow$ abstraction, indépendance entre interface et implantation.

Réutilisation :

- des objets différents partagent leur implantation (classes)
  ou dérivent leur implantation d'autres objets (héritage) ;
- polymorphisme : une implantation est réutilisable dans plusieurs contextes.

Variétés d'implantation

La programmation orientée objet est intégrée à de nombreux langages, avec une grande diversité d'implantations :

- langages à objets purs (tout est objet) ou avec des types primitifs ;
- contrôle d'accès aux champs ;
- langages avec ou sans classes ;
- héritage simple ou multiple ;
- liaison statique ou tardive (dynamique) ;
- liste des champs et méthodes fixée à la création de l'objet ou extensible après création ;
- introspection ;
- typage statique ou dynamique ;
- interfaces ;
- gestion automatique de la mémoire (ramasse-miettes) ou manuelle.

Sous-typage et héritage

Principe de substitution de Liskov (Barbara Liskov, prix Turing 2008).

Relation d'ordre partiel entre les types d'objets.

`A <: B` : `A` est un sous-type de `B`

Tout objet de type `A` peut être utilisé
dans un context où un objet de type `B` est attendu.

toute propriété prouvable sur `B` est aussi vraie sur `A`

Notion sémantique de haut niveau avec différentes implantations :

- sous-typage nominal : par héritage explicite
  (`A <: B` si `class A extends B`)
- sous-typage structurel : implicite, découvert par le compilateur
  (`A <: B` si `A` implante un sur-ensemble des méthodes de `B`)

Difficultés : comment le sous-typage interagit avec les autres constructions du langage ?

- covariance ou contravariance ?
- tableaux : si `A <: B`, a-t-on bien `A[] <: B[]` ?
- fonctions (covariance des arguments, contravariance de la valeur de retour) ;
- génériques : `class C<Data>`.
  variance fixée par le programmeur `<A extends B>` ou `<A super B>`

Exemples de langages à objets

Exemple : Smalltalk

Smalltalk : langage orienté objet pur pionnier.

Conçu dans les années 1970s, diffusé à partir de 1980.

- tout est un objet (y compris les constantes, les classes, etc.) ;
- tout objet est une instance d'une classe ;
- toute classe hérite d'une unique classe parent (héritage simple) ;
- la recherche de messages (méthodes) suit la chaîne d'héritage ;
  (erreur à l'exécution si aucun message de ce nom n'existe dans la classe)
- une méthode n'accède qu'aux champs de l'instance courante ;
- typage dynamique ;
- introspection ;
- environnement de programmation très dynamique.

Objective-C : modèle objet de Smalltalk au-dessus du C. (NeXT, Apple)

Berceau des *Design Patterns*.

Source d'inspiration principale pour ILP !

Exemple : C++

C++ : C avec classes (1983) (inspiré plus par Simula-67 que par Smalltalk).

Langage riche, dynamique et complexe
mais ayant vocation à être aussi efficace que le C (voir plus).
(pas de gestion automatique de mémoire, *run-time* assez léger, peu de vérification dynamique)

Nombreux choix laissés au programmeur :

- appel de méthode avec au choix liaison statique ou tardive ;
- héritage multiple, au choix avec ou sans partage des classes de base.

méthodes virtuelles

```
class A {
   void a() { ... }
   virtual void b () { ... }
   void c() { a(); b(); }
}
class B : public A {
   void a() { ... }
   void b() { ... }
}
```

classes de base virtuelles

```
class A {
    int a;
}
class B : public virtual A { ...
}
class C : public virtual A { ...
}
class D : public B, public C { ...
}
```

Exemple : Java

Java (1995) : langage orienté-objet inspiré par Objective-C
mais non encombré par des soucis de compatibilité.

- types primitifs, mais encapsulables dans des objets ;
  (`int` vs. `Integer`)
- héritage simple ;
- séparation syntaxique entre interface et implantation.

Popularise hors du monde académique :

- la gestion automatique de la mémoire, grâce au ramasse-miettes ;
- la compilation en *code-octet* pour une machine virtuelle portable ;
- la sûreté du langage (typage statique et vérifications dynamiques).

Exemple : JavaScript

JavaScript : développé en 1996 par Netscape pour son navigateur comme un langage de script pour les pages WEB (exécuté sur le client).

Trait distinctif : langage sans classe.

- héritage par prototype ;
  résolution en remontant la chaîne des prototypes, modifiable à souhait
  exemple d'utilisation : stocker les champs et méthodes communs à toutes les instances
- ajout dynamique de champs (propriétés) et de méthodes
  un objet est une table d'association modifiable
- un constructeur est une fonction.
  initialise les champs et méthodes spécifiques à l'instance

exemple

```
var Person = function (firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastname;
}
var moi = new Person("X","Y");
moi.age = 42;
Person.prototype.name() { this.firstName + " " + this.lastName; }
```

Exemple : Eiffel

Eiffel : ajoute les contrats à la programmation objet ;
encore plus d'information statique que le typage ;
$\Longrightarrow$ permet de s'assurer de la correction des programmes.

exemple

```
class STACK [ELEMENT] is
    invariant
        count <= capacity

    put (x : ELEMENT) is
        require
            count < capacity
        do
            -- implantation ...
        ensure
            count = old count + 1
end
```

- invariants d'objets (maintenus, sauf peut-être à l'intérieur des méthodes) ;
- pré et post-conditions de méthodes.

Exemple : OCaml

OCaml : ajoute l'orienté-objet au langage fonctionnel fortement typé ML.

Trait distinctif : séparation entre héritage et typage.

- sous-typage structurel (`A <: B` possible même si `A` n'hérite pas de `B`) ;
- possibilité de créer des objets sans classe ;
- les classes permettent l'héritage des champs et des méthodes ;
- typage statique, avec inférence (limitée) de type.

exemple

```
let create_person firstname surname = object
    val mutable _name = firstname ^ " " ^ surname
    method name = _name
    method set_name name = _name <- name
end
class person firstname surname = object
    method name = firstname ^ " " ^ surname
end
let a = create_person "A" B"
let b = new person "A" "B"
([ (a :> person); b] : person list)
```

Les objets dans ILP

ILP4 ajoute les objets, avec les choix suivants :

- système de classes (tout objet est une instance d'une classe) ;
- types primitifs séparés des classes (mais représentation partagée avec les objets) ;
- les classes sont globales (déclarées en début de programme) ;
- champs ;
- méthodes ;
- objet courant `self` (≃`this`, accès aux champs et méthodes) ;
- pas de contrôle d'accès (champs et méthodes publiques) ;
- héritage simple (des champs et méthodes) ;
- accès à la classe parent `super` (appel de méthodes) ;
- pas de méthode constructeur ;
  (à chaque création d'instance, la valeur d'initialisation des champs est passée en argument)
- pas d'ajout dynamique de champs et de méthodes (cf. TME) ;
- pas d'introspection (pas d'accès aux objets « classe »).

Exemple ILP

exemple

```
class Point {
  var x,y;
  method hash() (self.x);
};

class ColoredPoint extends Point {
  var color;
  method hash() (super + self.color);
};

let o = new Point(1,2) in o.hash();
```

- `class` : déclaration de classe
- `var x` : déclaration de champ
- `method m(...) ...` : déclaration de méthode
- `self.x` : lecture d'un champ ;
- `extends` : héritage, ajout d'un champ et redéfinition d'une méthode ;
- `super` : appel à la méthode courante de la classe parent ;
- `new` : création d'un objet ;
- `o.hash()` : appel de méthode.

Syntaxe des objets en ILP

Syntaxe concrète : les déclarations de classes (1/2)

ILPgrammar4.g4

```
prog returns [com.paracamplus.ilp4.interfaces.IASTprogram node]
    : (defs+=globalDef ';'?)*  (exprs+=expr ';'?) * EOF
    ;

globalDef returns [com.paracamplus.ilp1.interfaces.IASTdeclaration node]
    : def=globalFunDef # GlobalFunctionDefinition
    | def=classDef # ClassDefinition
    ;

classDef returns [com.paracamplus.ilp4.interfaces.IASTclassDefinition node]
    : 'class' name=IDENT ('extends' parent=IDENT)? '{'
        ('var' fields+=IDENT (',' fields+=IDENT)* ';')*
        (methods+=methodDef ';'?)*
      '}'
    ;
```

Un ensemble de classes globales, ayant chacune :

- le mot-clé `class` ;
- un nom de classe ;
- `extends` et le nom de la classe parent (optionnel ; `object` si absent)
- `var` puis une liste de noms de champ (éventuellement vide) ;

Syntaxe concrète : les déclarations de classes (2/2)

ILPgrammar4.g4

```
classDef returns [com.paracamplus.ilp4.interfaces.IASTclassDefinition node]
    : 'class' name=IDENT ('extends' parent=IDENT)? '{'
        ('var' fields+=IDENT (',' fields+=IDENT)* ';')*
        (methods+=methodDef ';'?)*
      '}'
    ;

methodDef returns [com.paracamplus.ilp4.interfaces.IASTmethodDefinition node]
    : 'method' name=IDENT '(' vars+=IDENT? (',' vars+=IDENT)* ')' body=expr
    ;
```

- une liste (éventuellement vide) de méthodes, ayant chacune :
  - le mot-clé `method` ;
  - un nom de méthode ;
  - une liste d'arguments entre parenthèses (sans l'argument self, implicite) ;
  - un corps (expression) ;

  $\Longrightarrow$ très similaire à une déclaration de fonction.

Attention à l'utilisation des accolades `{ }`,
et à l'ordre : champs puis méthodes.

Syntaxe concrète : opérations sur les objets (1/2)

ILPgrammar4.g4

```
expr returns [com.paracamplus.ilp1.interfaces.IASTexpression node]
    | 'self'  # Self
    | 'super' # Super
    | obj=expr '.' field=IDENT '=' val=expr # WriteField
    | obj=expr '.' field=IDENT # ReadField
    ...
    ;
```

- Expressions utilisables uniquement dans les méthodes :
  - `super` : accès à la méthode de la classe parent ;
  - `self` : accès à l'objet sur lequel la méthode est appelée.
- Lecture ou écriture du champ `field` de l'objet `obj` :
  - similaire à une lecture / affectation de variable
    mais `obj` est une expression arbitraire
    (on peut écrire `(f()).x = 12`)
  - le nom du champ est une chaîne constante
    (on ne peut pas écrire `obj.(f()) = 12`)
  - il faut toujours préciser un objet cible ; par exemple `self`.

Syntaxe concrète : opérations sur les objets (2/2)

ILPgrammar4.g4

```
expr returns [com.paracamplus.ilp1.interfaces.IASTexpression node]
    ...
    | 'new' className=IDENT  '(' args+=expr? (',' args+=expr)* ')' # New
    | obj=expr '.' field=IDENT '(' args+=expr? (',' args+=expr)* ')' # Send
    ;
```

- `new` : création d'un nouvel objet étant données :
  - sa classe (chaîne constante) ;
  - les valeurs initiales des champs (expressions arbitraires)
    (suivant l'ordre de déclaration dans la classe, en commençant par les champs hérités).
- Appel de méthode, étant donnés :
  - `obj` : l'objet (expression arbitraire) ;
  - `field` : le nom de la méthode (chaîne constante) ;
  - les arguments de la méthode (expressions arbitraires).

Interfaces de l'AST

Interfaces d'AST dans `com.paracamplus.ilp4.interfaces` :

- `IASTself`
- `IASTsuper`
- `IASTfieldRead`
- `IASTfieldWrite`
- `IASTinstantiation`
- `IASTsend`
- `IASTclassDefinition`
- `IASTmethodDefinition`

Et les classes les réalisant dans `com.paracamplus.ilp4.ast` :
`ASTself`, `ASTsuper`, `ASTfieldRead`, `ASTfieldWrite`, etc.

Les interfaces `IAST` reflètent les règles et attributs de la grammaire `.g4`.
Les implantations `AST` sont pour la plus part de simples conteneurs.
$\Longrightarrow$ on ne détaille que `IASTmethodDefinition` et `IASTclassDefinition`, plus intéressants.

AST : définition de méthode

IASTmethodDefinition.java

```
public interface IASTmethodDefinition
extends IASTfunctionDefinition
{
    String getMethodName();
    String getDefiningClassName();
}
```

Hérite d'une définition de fonction :

- `getFunctionVariable()` : nom de la méthode ;
  (sous forme de `IASTvariable`)
- `getVariables()` : arguments formels de la méthode ;
- `getBody` : corps de la méthode.

On y ajoute :

- `getMethodName` :
  en fait identique à `getFunctionVariable().getName()` ;
- `getDefiningClassName` :
  nom de la classe où la méthode est définie.

AST : définition de classe

IASTclassDefinition.java

```
public interface IASTclassDefinition
extends IASTdeclaration, Inamed
{
    String getSuperClassName();
    String[] getProperFieldNames();
    IASTmethodDefinition[] getProperMethodDefinitions();
    default String[] getProperMethodNames() {
        return Arrays.stream(getProperMethodDefinitions())
                     .map((md) -> md.getMethodName())
                     .toArray(String[]::new);
    }
}
```

- `getProper...` : les champs et méthodes déclarées dans la classe sans compter ceux hérités ;
- en Java 8, les interfaces peuvent avoir des définitions de méthodes ! ce sont les méthodes `default` ;
- `getProperMethodNames` extrait le nom des méthodes de `getProperMethodDefinitions`, en utilisant les lambdas de Java 8.

Aspects statiques, aspects dynamiques

Objets et classes

La classe d'un objet est fixée à la création.

Certaines propriétés d'un objet sont déterminées par sa classe :

- l'ensemble des champs ;
- l'ensemble des méthodes ;
- le code associé à chaque méthode ;

$\Longrightarrow$ l'objet garde un pointeur vers une structure de classe : dynamique ;
les structures de classe sont fixées à la compilation : statique.

D'autres propriétés évoluent après la création :

- la valeur des champs ;

$\Longrightarrow$ chaque objet a une table de champs : dynamique.

Structure de classe : exemple

```
class Point {
  var x,y;
  method toString() { code 1 };
  method getPos() { code 2 };
};
```

```
class ColorPoint extends Point {
  var color;
  method toString() { code 3 };
  method getColor() { code 4 };
};
```

structure de classe `Point`
étend `object`
champs : `x`, `y`
méthode `toString` : *code 1*
méthode `getPos` : *code 2*

structure de classe `ColorPoint`
étend `Point`
champs : `x`, `y`, `color`
méthode `toString` : *code 3*
méthode `getPos` : *code 2*
méthode `getColor` : *code 4*

- nom de la classe ;
- pointeur sur la classe parent (`object` pointe sur lui-même) ;
- table des champs propres et hérités ;
- table des méthodes propres, héritées ou redéfinies, et code associé.

Attention : l'ordre dans les tables sera important !

Accès aux champs

Les **champs d'un objet** sont stockés dans un tableau T.

Opération statique :

- associer un indice fixe i_f (dans $0 \dots n-1$) à chaque champ f.

À l'exécution :

- retrouver la classe de l'objet ;
- vérifier que la classe possède le champ f ;
  Sinon, l'indice est invalide, voire dépasse du tableau !
  Nous verrons plusieurs méthodes pour effectuer ce test.
- accès direct à $T[i_f]$
  $\Longrightarrow$ très efficace !

objet de classe `ColorPoint`
pointeur vers la structure de classe `ColorPoint`
T[0] : valeur du champ `x`
T[1] : valeur du champ `y`
T[2] : valeur du champ `color`

Accès aux champs : limitation

Limitation :

Un nom de champ a un indice unique dans tout le programme.
$\Longrightarrow$ deux classes ne peuvent pas déclarer des champs de même nom !

interdit

```
class A {
  var x,y;
};
class B {
  var y,z;
};
a.y; // quel indice ?
```

Deux classes partagent un nom de champ uniquement si l'une dérive de l'autre ; le champ est alors hérité, pas redéclaré.

Java n'a pas ce problème car les références aux objets sont typés !
`y` dans `A` et `y` dans `B` peuvent avoir des indices différents.

Résolution de méthode

Les **méthodes d'une classe** sont stockées dans un tableau M.
Un tableau par classe, pas un tableau par objet !

Opérations statiques :

- associer un indice fixe i_m à chaque nom de méthode m ;
- créer une table pour chaque classe associant à i_m :
  - le code de la méthode (hérité, nouveau ou redéfini) ;
  - l'arité (nombre d'arguments).

À l'exécution :

- retrouver la classe de l'objet ;
- vérifier que la classe possède la méthode m ;
  identique au problème des champs d'objets
- vérifier l'arité de la méthode ;
- appeler la méthode, en passant l'objet en premier argument (caché).

Cas particulier de `super` :

- appel uniquement possible dans une méthode ;
- la classe est connue statiquement, donc la classe parent également !
  $\Longrightarrow$ résolution statique sans faire appel aux tables.

Interprète

Interfaces ajoutées

Dans `com.paracamplus.ilp4.interpreter.interfaces` :

- `IInstance` : instance d'objet
  (sait lire, écrire les champs, appeler les méthodes, retrouver la classe)
- `IClass` : classe
  (table des indices de champs, table de méthodes)
- `IMethod` : méthode
  (`Invocable` + information de classe)
- `IClassEnvironment` : table des classes, par nom
- `ISuperCallLexicalEnvironment`, `ISuperCallInformation` : tout ce qu'il faut pour appeler `super`
  (`IMethod` + arguments de la méthode courante, implicite dans `super`)

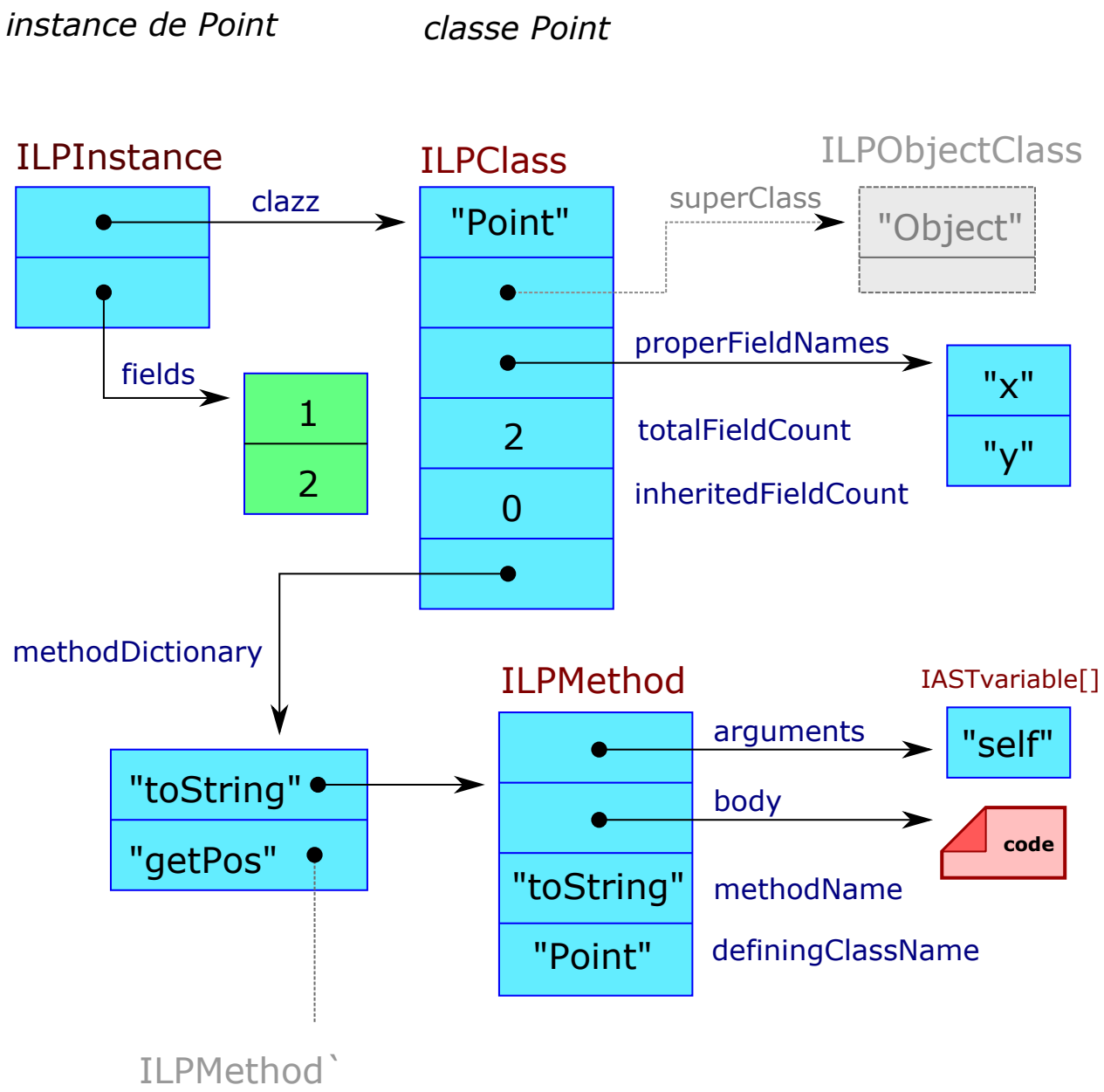
Classes ajoutées

Dans `com.paracamplus.ilp4.interpreter` :

- implantations de `IClass` :
  - `ILPAbstractClass` : implantation commune
  - `ILPClass` : classes utilisateur
  - `ILPObjectClass` : classe `object` prédéfinie
- `ILPInstance` implante `IInstance`
- `ILPMethod` implante `IMethod`
- `ClassEnvironment` implante `IClassEnvironment`
- implantations de `ISuperCallLexicalEnvironment` :
  - `SuperCallEmptyLexicalEnvironment` : liste vide
  - `SuperCallLexicalEnvironment` : nœud de variable
  - `SuperCallInformationLexicalEnvironment` : nœud `super`
- `SuperCallInformation` implante `ISuperCallInformation`

Certains noms de classe Java sont réservés, comme `Class` et `Method`, d'où le préfixe `ILP`.

Exemple : instance de classe `Point`



Instances : interface

IInstance.java

```
public interface IInstance
{
    IClass classOf();
    Object read (String fieldName) throws EvaluationException;
    Object write(String fieldName, Object value) throws ...;
    Object send (Interpreter interpreter,
                 String message,
                 Object[] arguments) throws ...;
}
```

- l'information de classe ;
- toutes les actions possibles sur un objet :
  - lire un champ : `read`
  - modifier un champ : `write`
  - exécuter une méthode : `send`.

  Les noms de champs et méthodes sont des chaînes.

Instances : état et construction

ILPInstance.java (début)

```
public class ILPInstance implements IInstance
{
    private final IClass clazz;
    private final Object[] fields;

    public ILPInstance (IClass clazz, Object[] fields)
    throws EvaluationException
    {
        this.clazz = clazz;
        this.fields = fields;
        if (fields.length != clazz.getTotalFieldCount())
            throw new EvaluationException(...);
    }
```

Construction : définition des attributs

- classe `clazz` de l'objet créé ;
- vérification du nombre d'arguments du constructeur ;
- table de champs `fields` initialisée à la table d'arguments du constructeur.

Instances : opérations

ILPInstance.java (suite)

```
public Object read(String fieldName) throws ...
{
    int offset = classOf().getOffset(fieldName);
    return fields[offset];
}

// idem pour write

public Object send(Interpreter interpreter, String msg, Object[] args)
throws ...
{
    return classOf().send(interpreter, this, msg, args);
}
```

Les fonctions importantes sont déléguées à la classe :

- conversion d'un champ de nom `fieldName` en indice `offset` ;
- appel de méthode par son nom `msg`.

Classes : interface

IClass.java

```
public interface IClass
{
    String getName();

    IClass getSuperClass() throws EvaluationException;

    String[] getProperFieldNames();
    String[] getTotalFieldNames();
    int getTotalFieldCount();
    int getOffset(String fieldName) throws EvaluationException;

    Map<String,IMethod> getMethodDictionary();
    Object send(Interpreter interpreter, IInstance receiver,
                String message, Object[] arguments) throws ...
}
```

Accès à la définition des champs et des méthodes, au nom et au parent.
Informations communes à toutes les instances d'une même classe.

Classes : état

ILPAbstractClass.java (début)

```
public abstract class ILPAbstractClass implements IClass, Inamed
{
    private final String className;
    private final IClass superClass;
    private final String[] properFieldNames;
    private final int totalFieldCount;
    private final int inheritedFieldCount;
    private final Map<String, IMethod> methodDictionary;
}
```

`ILPAbstractClass` est une classe abstraite.
En réalité elle possède toute l'implantation nécessaire...

`ILPAbstractClass` est parent de deux implantations :

- `ILPClass` : classes du programme (identique à `ILPAbstractClass`) ;
- `ILPObjectClass` : classe `Object`, prédéfinie.

Classes : construction

ILPAbstractClass.java (suite)

```
public ILPAbstractClass (IClassEnvironment classEnvironment,
                         String className, String superClassName,
                         String[] fieldNames, IMethod[] methods) throws ...
{
    this.className = className;
    if ("Object".equals(className)) this.superClass = this;
    else this.superClass = classEnvironment.getILPClass(superClassName);

    this.properFieldNames = fieldNames;
    this.inheritedFieldCount = superClass.getTotalFieldCount();
    this.totalFieldCount = inheritedFieldCount + properFieldNames.length;

    this.methodDictionary = new HashMap<>();
    Map<String,IMethod> superDictionary = superClass.getMethodDictionary();
    for (IMethod method : superDictionary.values()) {
        methodDictionary.put(method.getName(), method);
    }
    for (IMethod method : methods) {
        methodDictionary.put(method.getName(), method);
        method.setDefiningClass(this);
    }

    classEnvironment.addILPClass(this);
}
```

- `classEnvironment` associe à un nom de classe sa structure `IClass`
- `object` est son propore parent

Classes : recherche de champ

ILPAbstractClass.java (suite)

```
public int getOffset(String fieldName)
throws EvaluationException
{
    String[] properFieldNames = getProperFieldNames();
    for (int i = 0; i < properFieldNames.length; i++) {
        String properFieldName = properFieldNames[i];
        if (properFieldName.equals(fieldName)) {
            return getSuperClass().getTotalFieldCount() + i;
        }
    }
    return getSuperClass().getOffset(fieldName);
}
```

Associe un indice (démarrant à 0) à chaque nom de champ

- identique à l'indice dans la classe parent pour les champs hérités ;
- au-delà des indices hérités pour les nouveaux champs ;

par recherche récursive le long de la chaîne d'héritage.
$\Longrightarrow$ compatibilité ascendante

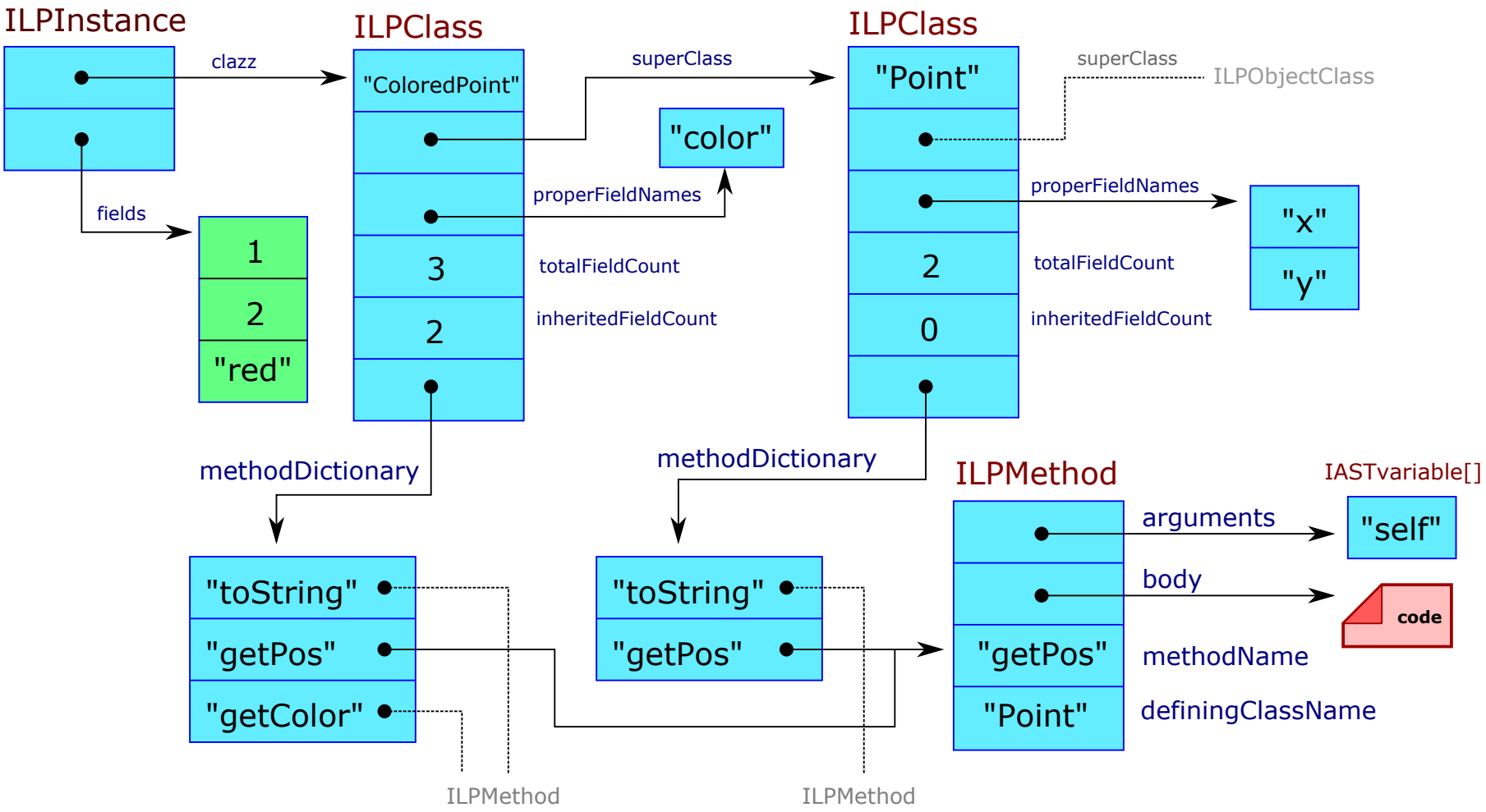
Classes : appel de méthode

ILPAbstractClass.java (suite)

```
public Object send(Interpreter interpreter, IInstance receiver,
                   String message, Object[] arguments) throws ...
{
  IMethod method = getMethodDictionary().get(message);
  if (method == null) throw new EvaluationException("Does not understand " + message);

  if (arguments.length != method.getMethodArity())
      throw new EvaluationException("Wrong arity for " + message);

  List<Object> functionArguments = new Vector<>();
  functionArguments.add(receiver);
  for (Object argument : arguments) {
      functionArguments.add(argument);
  }

  return method.apply(interpreter, functionArguments.toArray());
}
```

- recherche de la méthode dans le dictionnaire (rempli par le constructeur) ;
- vérification du nombre d'arguments ;
- `self` est passé en premier argument de la méthode ;
- l'appel lui-même est délégué à un objet `IMethod`.

Exemple : instance de classe `ColoredPoint`



La classe `Object`

ILPObjectClass.java

```
public class ILPObjectClass extends ILPAbstractClass
{
  ILPObjectClass (IClassEnvironment classEnvironment) throws ... {
      super(classEnvironment, "Object", null, new String[0], new IMethod[0]);
  }
  public IClass getSuperClass() throws ... {
      throw new EvaluationException("Object has no super class");
  }
  private final static String[] properFieldNames = new String[0];
  public int getTotalFieldCount() {
      return 0;
  }
  public int getOffset(String fieldName) throws ... {
      throw new EvaluationException("Object has no field " + fieldName);
  }
}
```

- pas de classe parent ;
- pas de champ ;
- des méthodes, mais elles seront ajoutées par `ClassEnvironment`.

Environnement de classes : interface

IClassEnvironment.java

```
public interface IClassEnvironment
{
  IClass getILPClass(String name) throws EvaluationException;
  void addILPClass(IClass clazz);
}
```

Les classes sont des ressources globales :

- déclarées au début du programme ILP ;
- valables pour toute la durée du programme ;
- stockées directement dans l'objet `IASTprogram` ;

$\Longrightarrow$ l'interprète stocke les classes dans une table globale avant d'exécuter le programme.

Environnement de classes : implantation

ClassEnvironment.java

```
public class ClassEnvironment implements IClassEnvironment
{
  private final Map<String, IClass> clazzes;

  public ClassEnvironment (Writer out) throws ...{
      this.clazzes = new HashMap<>();
      IClass objectClass = new ILPObjectClass(this);
      initializeClassEnvironment(objectClass, out);
  }
  protected void initializeClassEnvironment(IClass clazz, Writer out) {
      addPrimitiveAsMethod(clazz, new Print(out));
      addPrimitiveAsMethod(clazz, new Newline(out));
  }
  protected void addPrimitiveAsMethod(final IClass clazz,
                                      final IPrimitive primitive) {
      IMethod method = new IMethod() {  /* ... */ };
      clazz.getMethodDictionary().put(primitive.getName(), method);
  }
}
```

- création de la classe `Object` racine (prédéfinie) ;
- remplissage avec deux méthodes : `Print` et `Newline` ;
- il s'agit en fait de primitives bien connues (`IPrimitive`) enrobées dans des méthodes (`IMethod`).

Gestion de super

exemple ILP

```
class A
{
  method m(a,b) { ... };
};
class B extends A
{
  method m(a,b) { super; };
};
let o = new B () in o.m(1,2);
```

Cas particulier de super :

- super ne prend pas d'argument ; il réutilise ceux de la méthode courante ;
- l'appel « normal » à o.m dans B doit se souvenir des valeurs des arguments ;
- et appeler la méthode du parent avec ces arguments : o.m(1,2) de A.

C'est le rôle de ISuperCallInformation.

Information de méthode parent

ISuperCallInformation.java

```
public interface ISuperCallInformation
{
    Object[] getArguments();
    IMethod getSuperMethod() throws EvaluationException;
}
```

SuperCallInformation.java

```
class SuperCallInformation implements ISuperCallInformation
{
  private final Object[] arguments;
  private final IMethod method;
  public IMethod getSuperMethod() throws EvaluationException {
      IClass definingClass = method.getDefiningClass();
      if ("Object".equals(definingClass.getName()))
          throw new EvaluationException("Cannot invoke super()");
      else
          return definingClass.getSuperClass()
                 .getMethodDictionary().get(method.getName());
  }
}
```

- `arguments` et `method` sont spécifiés par le constructeur ;
- `getArguments` retourne `arguments` ;
- `getSuperMethod` demande au parent de la classe définissant `method` de trouver la méthode du même nom.

Nouvel environnent lexical (1/2)

L'information `SuperCallInformation` doit être propagée dans l'interprète (depuis un nœud appel de méthode, jusqu'à un nœud `super`).

Le visiteur d'interprétation a un seul argument,
il est déjà utilisé pour l'environnement lexical.

$\Longrightarrow$ création d'une classe combinant les deux informations :

- l'environnement lexical : variable $\rightarrow$ valeur ;
- un objet `SuperCallInformation`.

Nouvel environnent lexical (2/2)

ISuperCallLexicalEnvironment.java

```
public interface ISuperCallLexicalEnvironment
extends ILexicalEnvironment
{
  ISuperCallLexicalEnvironment extend(ISuperCallInformation isci);
  ISuperCallInformation getSuperCallInformation()
          throws EvaluationException;
}
```

Similaire à `ILexicalEnvironment` : liste chaînée avec un cas vide (`SuperCallEmptyLexicalEnvironment`) et un cas cellule (`SuperCallLexicalEnvironment`).

La liste chaînée passée à l'interprète mélange deux types de cellules :

- les fonctions classiques utilisent des `ILexicalEnvironment` ;
- les méthodes utilisent des `ISuperCallLexicalEnvironment` ;

$\Longrightarrow$ conversion de classe nécessaire dans l'itérateur pour les distinguer.

Méthodes : état et construction

ILPMethod.java

```
public class ILPMethod extends Function implements IMethod, Inamed
{
  private final String methodName;
  private final String definingClassName;
  public ILPMethod(String methodName, String definingClassName,
                   IASTvariable[] variables, IASTexpression body)
  {
      super(variables, body, new SuperCallEmptyLexicalEnvironment());
      this.methodName = methodName;
      this.definingClassName = definingClassName;
  }
```

- hérite de `Function` : arguments (`IASTvariable[]`) + corps (`IASTexpression`) + environnement lexical ;
- l'environnement lexical est maintenant de type `ISuperCallLexicalEnvironment` ;
- ajoute l'information permettant de retrouver la classe : `definingClassName`.

Méthodes : appel

ILPMethod.java

```
public Object apply(Interpreter interpreter, Object[] argument)
throws EvaluationException
{
    if (arguments.length != getArity())
        throw new EvaluationException("Wrong arity");
    ILexicalEnvironment lexenv2 = getClosedEnvironment();
    ISuperCallInformation isci =
        new SuperCallInformation(arguments, this);
    lexenv2 = ((ISuperCallLexicalEnvironment)lexenv2).extend(isci);
    IASTvariable[] variables = getVariables();
    for (int i=0 ; i<arguments.length ; i++)
        lexenv2 = lexenv2.extend(variables[i], arguments[i]);
    return getBody().accept(interpreter, lexenv2);
}
```

Identique à la méthode `apply` de `Function`
excepté l'utilisation d'un `ISuperCallInformation` pour ajouter l'information de méthode parent.

Interprète : initialisation

Interpreter.java (début)

```
public Object visit(IASTprogram iast, ILexicalEnvironment lexenv)
{
    for (IASTclassDefinition cd : iast.getClassDefinitions())
        this.visit(cd, lexenv);
     ...
}

public IClass visit(IASTclassDefinition iast, ILexicalEnvironment lexenv)
{
    List<IMethod> methods = new Vector<>();
    for (IASTmethodDefinition md : iast.getProperMethodDefinitions()) {
        IMethod m = visit(md, lexenv);
        methods.add(m);
    }
    IClass clazz = new ILPClass( ... );
    return clazz;
}

public IMethod visit(IASTmethodDefinition iast, ILexicalEnvironment lexenv)
{
    IMethod method = new ILPMethod(...);
    return method;
}
```

Début de l'interprétation : création des classes et méthodes.

Interprète : création d'un objet

Interpreter.java (suite)

```
public Object visit(IASTinstantiation iast, ILexicalEnvironment lexenv)
{
    IClass clazz = getClassEnvironment().getILPClass(iast.getClassName());

    List<Object> args = new Vector<Object>();
    for (IASTexpression arg : iast.getArguments()) {
        Object value = arg.accept(this, lexenv);
        args.add(value);
    }

    return new ILPInstance(clazz, args.toArray());
}
```

- recherche de la classe dans l'ensemble des classes du programme ;
- évaluation des arguments ;
- construction d'une instance.

Interprète : accès à un champ

Interpreter.java (suite)

```
public Object visit(IASTfieldWrite iast, ILexicalEnvironment lexenv)
{
    String fieldName = iast.getFieldName();
    Object target = iast.getTarget().accept(this, lexenv);
    Object value = iast.getValue().accept(this, lexenv);
    if (target instanceof ILPInstance)
        return ((ILPInstance) target).write(fieldName, value);
    else
        throw new EvaluationException("Not an ILP instance " + target);
}
```

- évaluation de la cible `target` à modifier ;
- évaluation de la nouvelle valeur `value` du champ ;
- vérification que la cible est une instance de classe ;
- délégation de la modification du champ à l'instance, par `write`.

`IASTfieldRead` is similaire.

Interprète : self et super

Interpreter.java (fin)

```
public Object visit(IASTself iast, ILexicalEnvironment lexenv)
{
    return lexenv.getValue(iast);
}

public Object visit(IASTsuper iast, ILexicalEnvironment lexenv)
{
    ISuperCallInformation isci =
        ((ISuperCallLexicalEnvironment) lexenv).getSuperCallInformation();
    IMethod supermethod = isci.getSuperMethod();
    return supermethod.apply(this, isci.getArguments());
}
```

- IASTself est une variable locale (extends IASTvariable),
  de nom self,
  sa valeur est dans l'environnement lexical ;
  (ajouté automatique par le visiteur ANTLR, et lors des appels de méthode)
- super cherche la méthode parent dans l'environnement lexical
  et l'appelle.

Coût des objets dans l'interprète ILP

- lecture et écriture de champ :
  - recherche de l'indice du champ,
    d'abord dans la classe `properFieldNames`
    puis en remontant la hiérarchie de classes par `superClass`
    $\Longrightarrow$ coût proportionnel au nombre de champs
    $+$ profondeur de la hiérarchie ;
  - accès dans le tableau de champs `fields` de l'instance ;
    $\Longrightarrow$ coût constant.

- appel de méthode :
  table d'association `methodDictionary` locale à chaque classe ;
  $\Longrightarrow$ coût constant.

L'accès aux champs pourrait être amélioré ! (cf. compilateur)

Note : l'indice associé à un nom de champ est fixé statiquement,
c'est la vérification que le champ existe qui est coûteux.
– Java évite ce coût grâce au typage statique (coût nul à l'exécution).
– Python évite ce coût grâce à une table de hachage (coût constant).

Compilation des objets

Représentation en C

Principes

Le C n'a pas de notion d'objet, nous utiliserons :

- des structures et tableaux (pour les champs) ;
- des pointeurs de fonctions (pour les méthodes).

Nous utiliserons :

- une structure par instance, classe, champ, méthode ;
- un tableau de valeurs de champs pour chaque instance ;
- un tableau de méthodes pour chaque classe.

Mécanisme classique pour simuler la programmation objet en C.

Si `A` hérite de `B` :

- le tableau de champs / méthodes de `B` est un préfixe de celui de `A` ;
- les champs / méthodes supplémentaires sont ajoutés à la fin.

Le coût d'accès à un champ ou une méthode est constant, donc très efficace ;
mais on garde un coût élevé pour la vérification que la classe de l'objet possède le champ ou la méthode demandée...

Représentation des valeurs : ILP_Object

ilp.h

```
typedef struct ILP_Object {
   struct ILP_Class*  _class;
   union {
      int             asInteger;
      struct asString {
         int       _size;
         char      asCharacter[];
      } asString;
      struct asInstance {
           struct ILP_Object*  field[];
      } asInstance;
      // ldots
   } _content;
} *ILP_Object;
```

- Représentation universelle pour les valeurs : types primitifs et instances.
- Réification : les classes, champs, méthodes, sont aussi des `ILP_Object` !
- Toute valeur a un type, qui est encodé par une classe `_class`.

Représentation des classes

ILP_Object dans ilp.h

```
struct asClass_ {
    struct ILP_Class*     super;
    char*                 name;
    int                   fields_count;
    struct ILP_Field*     last_field;
    int                   methods_count;
    ILP_general_function  method[];
} asClass;
```

`ILP_Class` = `ILP_Object` réduit à une classe (même champs que `asClass`).

- `super` pointe vers la classe parent (`NULL` pour `Object`) ;
- `fields_count` : nombre total de champs (utile pour allouer les instances) ;
- `last_field` pointe sur une liste chaînée des noms de champs ;
  (champs propres et hérités, la liste est partagée avec le parent)
- `method` est un tableau de `methods_count` corps de méthode.
  (méthodes propres et héritées)

Représentation des champs

ILP_Object dans ilp.h

```
struct asField_ {
    struct ILP_Class*   defining_class;
    struct ILP_Field*   previous_field;
    char*               name;
    short               offset;
} asField;
```

Chaque nom de champ a une unique structure `ILP_Field` partagée par toutes les classes ayant ce champ.

Les noms de champ forment une liste chaînée.

- `defining_class` : classe (unique) où le champ est déclaré ;
- `previous_field` : champ suivant de la classe ;
- `name` : nom du champ ;
- `offset` : indice où est stockée la valeur du champ dans une instance.

Représentation des méthodes

ILP_Object dans ilp.h

```
struct asMethod_ {
    struct ILP_Class*   class_defining;
    char*               name;
    short               arity;
    short               index;
} asMethod;
```

Comme les noms de champ, chaque nom de méthode a une unique structure, indiquant en particulier :

- `defining_class` : classe où la méthode est déclarée pour la première fois ;
  la structure est réutilisée par les classes qui en héritent
- `arity` : le nombre d'arguments, en comptant `self` ;
- `index` : l'indice dans la table des méthodes de la classe.

Classes, champs, méthodes, instances prédéfinies

ilp.h

```
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_Closure_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_Field  ILP_object_value_field;

extern struct ILP_Method ILP_object_print_method;
extern struct ILP_Method ILP_object_classOf_method;
```

- classe racine : `Object` ;
- classes structurelles : méta-classe, classe des méthodes, des champs ;
- classes des types primitifs : `Integer`, `Float`, ...
- champs et méthodes de ces classes : `super`, `print`, ...

Classe d'un type primitif : les entiers `Integer`

ilp.c

```
struct ILP_Class ILP_object_Integer_class =
{
     &ILP_object_Class_class,
     { { &ILP_object_Object_class,
         "Integer",
         0,
         NULL,
         2,
         { ILP_print,
           ILP_classOf }
     } }
};
```

- c'est une valeur de type `Class` (donc `ILP_object_Class_class`) ;
- la classe parent de `Integer` est `Object` (donc `ILP_object_Object_class`) ;
- pas de champ (donc `0` et `NULL`) ;
- deux méthodes, d'implantation `ILP_print` et `ILP_classOf`.

Classe racine : Object

ilp.c

```
struct ILP_Class ILP_object_Object_class =
{
     &ILP_object_Class_class,
     { { NULL,
         "Object",
         0,
         NULL,
         2,
         { ILP_print,
           ILP_classOf }
     } }
};
```

Très similaire à la classe Integer, mais pas de classe parent (NULL).

La méta-classe : Class

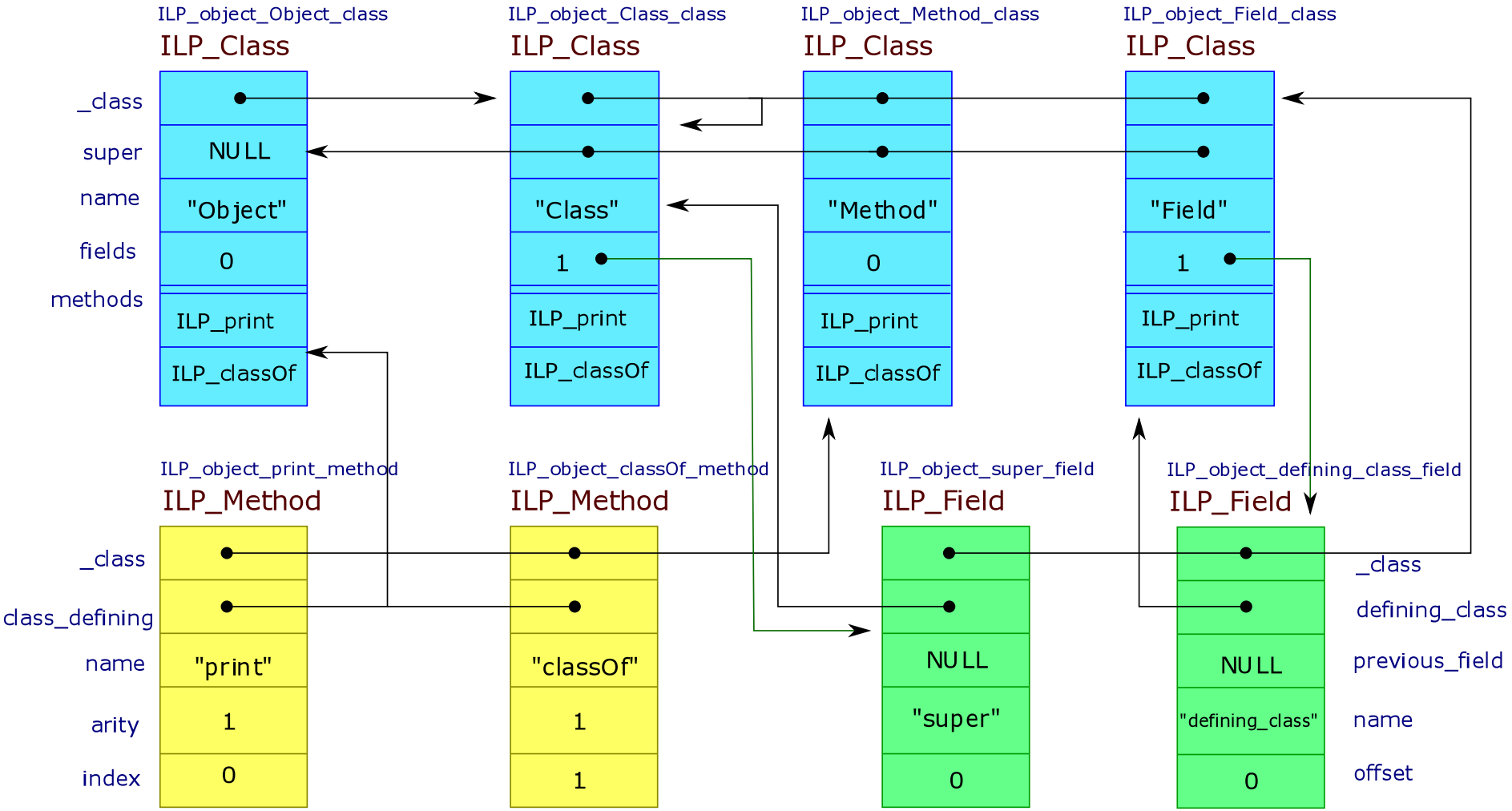
ilp.c

```
struct ILP_Class ILP_object_Class_class = {
   &ILP_object_Class_class,
   { { &ILP_object_Object_class,
       "Class",
       1, &ILP_object_super_field,
       2, { ILP_print, ILP_classOf }
   } }
};

struct ILP_Field ILP_object_super_field = {
   &ILP_object_Field_class,
   { { &ILP_object_Class_class,  NULL, "super", 0 } }
};
```

La classe des valeurs de type classe...
c'est bien sûr une classe (`ILP_object_Class_class`)
dérivant directement de la classe `Object` (`ILP_object_Object_class`).

La classe parent est le premier champ de la structure `asClass` ;
nous pouvons la voir comme l'unique champ, `super`, d'un objet.

Illustration



Structures à générer pour une classe utilisateur

Dans le fichier C généré, le compilateur ajoute une variable globale constante pour chaque classe, champ, méthode du programme ILP :

- pour chaque classe *C* ayant *i* méthodes,
  une structure :
  `struct ILP_Class`*i*` ILP_Object_`*C*`_class`
  avec un champ tableau `method` de taille *i*
- pour chaque nom de méthode *m* de *C*
  qui n'existe pas dans son parent,
  une structure :
  `struct ILP_Method ILP_object_`*m*`_method`
- pour chaque nom de champ *f* de *C*
  qui n'existe pas dans son parent,
  une structure :
  `struct ILP_Field ILP_object_`*f*`_field`

Macro-instruction de génération de types de classes

ilp.h

```
#define ILP_GenerateClass(i)                                    \
typedef struct ILP_Class##i {                                   \
     struct ILP_Class* _class;                                  \
     union {                                                    \
          struct asClass_##i {                                  \
               struct ILP_Class*     super;                     \
               char*                 name;                      \
               int                   fields_count;              \
               struct ILP_Field*     last_field;                \
               int                   methods_count;             \
               ILP_general_function  method[i];                 \
          } asClass;                                            \
     } _content;                                                \
} *ILP_Class##i
```

Par exemple, `ILP_GenerateClass(2)` génère un type `ILP_Class2` avec un tableau `method` à 2 cases...

Macro-instruction, pour fixer la taille des tableaux statiques.

Pour les instances, le problème ne se posera pas : nous utiliserons l'allocation dynamique, `malloc`.

Exemple de classe utilisateur (1/4)

code ILP

```
class Point
{
    var x,y;
    method toString() { code 1 };
    method getPos()   { code 2 };
};

class ColorPoint extends Point
{
    var color;
    method toString() { code 3 };
    method getColor() { code 4 };
};
```

Exemple de classe utilisateur (2/4)

structures générées (début)

```
ILP_GenerateClass(4);
ILP_GenerateClass(5);

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_Class5 ILP_object_ColoredPoint_class;
extern struct ILP_Field  ILP_object_color_field;

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_Field ILP_object_color_field = {
    &ILP_object_Field_class,
    {{(ILP_Class) &ILP_object_ColoredPoint_class, &ILP_object_y_field, "color", 2}}
};
```

Définition des types de classe `ILP_GenerateClass`.

Déclaration avant définition des variables (permet les structures récursives en C).

Définition et initialisation des objets « champ », en liste chaînée.

Exemple de classe utilisateur (3/4)

structures générées (suite)

```
struct ILP_Class4 ILP_object_Point_class = {
    &ILP_object_Class_class,
    {{(ILP_Class) &ILP_object_Object_class,
      "Point",
      2, &ILP_object_y_field,
      4, {ILPm_print, ILPm_classOf, ilp__toString_1, ilp__getPos_2}
    }}
};

struct ILP_Class5 ILP_object_ColoredPoint_class = {
    &ILP_object_Class_class,
    {{(ILP_Class) &ILP_object_Point_class,
      "ColoredPoint",
      3, &ILP_object_color_field,
      5, {ILPm_print, ILPm_classOf, ilp__toString_3, ilp__getPos_2, ilp__getColor_4}
    }}
};
```

Définition des objets « classe ».

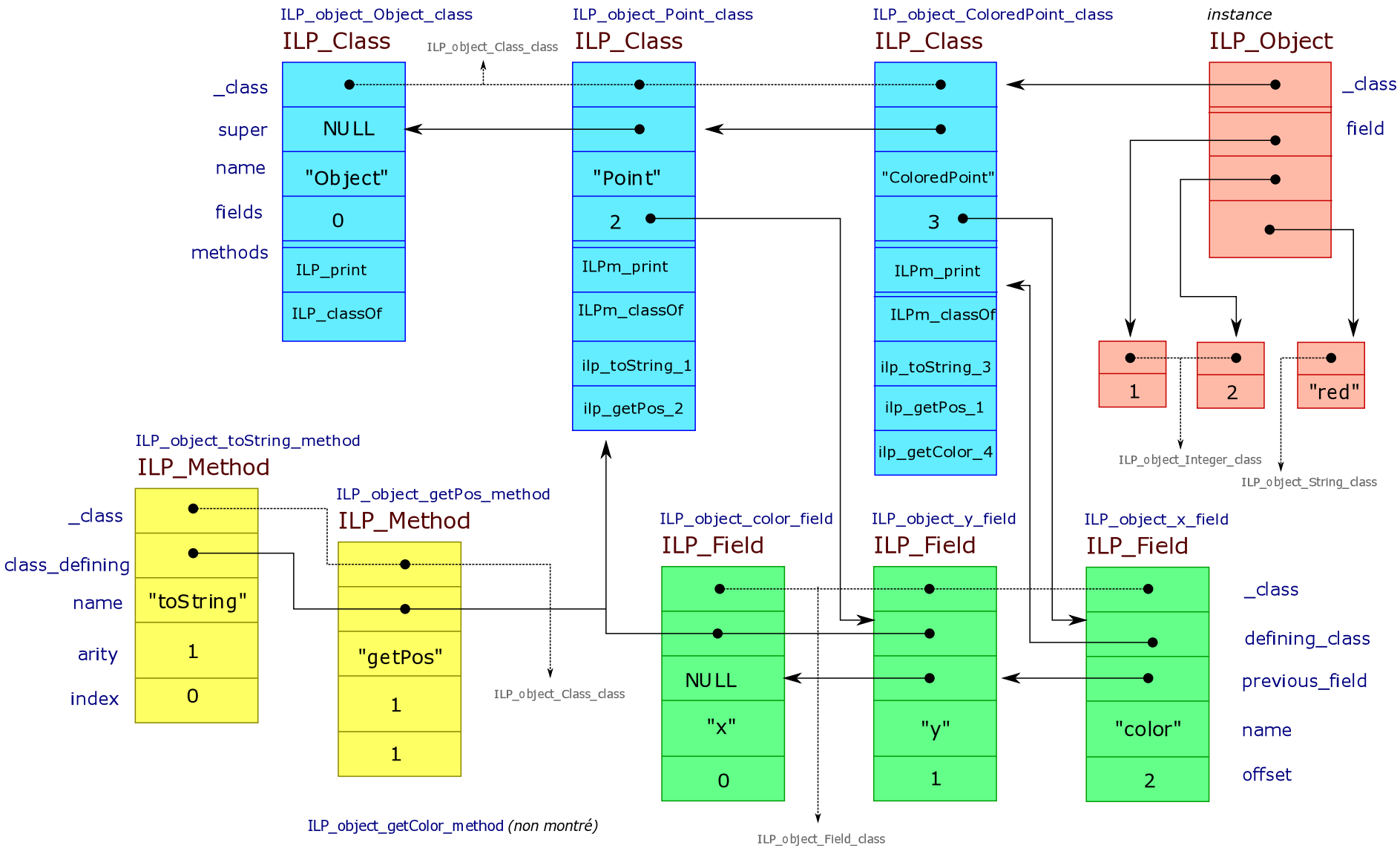
Exemple de classe utilisateur (4/4)

structures générées (suite)

```
struct ILP_Method ILP_object_toString_method = {
    &ILP_object_Method_class,
    {{(struct ILP_Class *)&ILP_object_Point_class,
      "toString",
      1, /* arité */
      2  /* offset */
    }}
};
struct ILP_Method ILP_object_getPos_method = {
    &ILP_object_Method_class,
    {{(struct ILP_Class *)&ILP_object_Point_class,
      "getPos",
      1, /* arité */
      3  /* offset */
    }}
};
struct ILP_Method ILP_object_getColor_method = {
    &ILP_object_Method_class,
    {{(struct ILP_Class *)&ILP_object_ColoredPoint_class,
      "getColor",
      1, /* arité */
      4  /* offset */
    }}
};
```

Définition des objets « méthode ».

Illustration



Opérations sur les objets

Allocation : cas primitif

ilp.h, ilp.c

```
ILP_Object ILP_malloc (int size, ILP_Class class) {
    ILP_Object result = ILP_MALLOC(size);
    if ( result == NULL ) return ILP_die("Memory exhaustion");
    result->_class = class;
    return result;
}

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

ILP_Object ILP_make_integer (int d) {
    ILP_Object result = ILP_AllocateInteger();
    result->_content.asInteger = d;
    return result;
}
```

- `ILP_malloc` : alloue l'instance et renseigne sa classe ;
- `ILP_AllocateInteger` : version spécialisée au type `Integer` ;
- `ILP_make_integer` : alloue un entier et initialise sa valeur ;
- alloue juste la place nécessaire à une structure `ILP_Object`.
  ne pas confondre `ILP_Object`, qui est un pointeur, avec `struct ILP_Object` qui est une structure avec de la place dans `asInteger` pour stocker un entier

Allocation : cas non-primitif

ilp.h, ilp.c

```
#define ILP_MakeInstance(c) \
    ILP_make_instance((ILP_Class) &ILP_object_##c##_class)

ILP_Object ILP_make_instance (ILP_Class class) {
    int size = sizeof(ILP_Class);
    size += sizeof(ILP_Object) * class->_content.asClass.fields_count;
    return ILP_malloc(size, class);
}
```

Alloue exactement assez d'espace pour stocker :

- une référence sur la classe du type (renseignée par `ILP_malloc`) ;
- la valeur de chaque champ (il y en a `fields_count`).

Attention : comme `ILP_Object`, `ILP_Class` est un pointeur !
(pas une structure, ne pas confondre avec `struct ILP_Class`)

Allocation : code généré

code généré

```
ILP_Object ilptmp960;
{
  ILP_Object ilptmp961;
  ILP_Object ilptmp962;
  ILP_Object ilptmp963;
  ilptmp962 = ILP_Integer2ILP(11);
  ilptmp963 = ILP_Integer2ILP(22);
  ilptmp961 = ILP_MakeInstance(Point);
  ilptmp961->_content.asInstance.field[0] = ilptmp962;
  ilptmp961->_content.asInstance.field[1] = ilptmp963;
  ilptmp960 = ilptmp961;
}
// ilptmp960 est ici initialisé à new Point(11,12)
```

Code généré pour `new Point(11,22)`.

- allocation de l'instance avec `ILP_MakeInstance` ;
- initialisation des champs `asInstance.field[]`.

Test de type : cas primitif

ilp.h

```
#define ILP_isInteger(o) \
    ((o)->_class == &ILP_object_Integer_class)

#define ILP_CheckIfInteger(o)                    \
    if ( ! ILP_isInteger(o) ) {                  \
        ILP_domain_error("Not an integer", o);   \
    }
```

- `ILP_isInteger` retourne faux si `o` n'est pas un entier ;
- `ILP_CheckIfInteger` signale une exception si `o` n'est pas entier.

Il suffit de regarder l'information de classe `_class`.

Chaque classe est codée par une unique variable globale C constante ;
$\Longrightarrow$ une comparaison d'adresse `==` suffit !

Accès à un champ : code généré

code généré

```
ILP_Object ilptmp806;
ILP_Object ilptmp807;
ilptmp807 = ...
if (ILP_IsA(ilptmp807, Point)) {
    ilptmp806 = ilptmp807->_content.asInstance.field[0];
} else {
    ilptmp806 = ILP_UnknownFieldError("x", ilptmp807);
}
```

Code généré pour une lecture `o.x` :

- calcul de `o`, dans `ilptmp807` (omis) ;
- vérification dynamique que `o` a bien un champ nommé `x` ;
  $\Longrightarrow$ `ILP_IsA` vérifie que `o` est de la classe `Point` ou dérivée.
  (voir transparent suivant)
- lecture du champ dans la table de champs de l'instance,
  à l'indice fixe 0.
  (l'indice de `x` est pré-calculé statiquement par le compilateur)

Test de type : instanceof

ilp.h, ilp.c

```
#define ILP_IsA(o,c) \
    ILP_is_a(o, (ILP_Class)(&ILP_object_##c##_class))

int 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;
        while ( oclass ) {  /* Object's superclass is NULL */
            if ( oclass == class ) return 1;
            oclass = oclass->_content.asClass.super;
        }
        return 0;
    }
}
```

Teste si o est de la classe class, ou d'une classe dérivée de class
$\Longrightarrow$ il faut remonter la chaîne des classes parents de o.

Test de type : `classof`, utilisé par `ILP_print`

Tester si un objet `o` est exactement de la classe `class` se fait simplement par : `o->_class == class`.

Application : fonction `print` générique.

ilp.c

```
ILP_Object ILP_print(ILP_Object self)
{
    if ( self->_class == &ILP_object_Integer_class ) {
        fprintf(stdout, "%d", self->_content.asInteger);
    } else if ( self->_class == &ILP_object_Float_class ) {
        fprintf(stdout, "%12.5g", self->_content.asFloat);
    } else /* nombreux cas omis ... */ {
        fprintf(stdout, "<%s", self->_class->_content.asClass.name);
        ILP_print_fields(self, self->_class->_content.asClass.last_field);
        fprintf(stdout, ">");
    }
}
```

- cas primitif : affichage direct de la valeur ;
- cas objet : affichage du nom de classe et de la valeur des champs

  `ILP_print_fields` appelle `ILP_print` sur chaque champ
  $\Longrightarrow$ affichage récursif de la structure complète.

Recherche de méthode

ilp.c

```
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) ) {
        // erreur : pas de méthode
    }
    if ( argc != method->_content.asMethod.arity ) {
        // erreur : arité incorrecte
    }
    int index = method->_content.asMethod.index;
    return oclass->_content.asClass.method[index];
}
```

Retourne un pointeur vers le code correspondant à un appel de méthode.

- partie coûteuse : vérifier que l'objet est compatible avec la méthode ; la classe de `receiver` doit dériver de la classe définissant la méthode ;
- partie efficace : récupérer un pointeur dans la table de méthodes ;
- comme pour tout appel, vérification du nombre d'arguments.

Test de sous-typage, utilisé par ILP_find_method

ilp.c

```
int ILP_is_subclass_of (ILP_Class oclass, ILP_Class otherclass)
{
    if ( oclass == otherclass ) return 1;
    else {
        oclass = oclass->_content.asClass.super;
        while ( oclass ) { /* Object's superclass is NULL */
            if ( oclass == otherclass ) return 1;
            oclass = oclass->_content.asClass.super;
        }
        return 0;
    }
}
```

ILP_is_subclass_of, utilisé par la recherche de méthodes
très similaire à ILP_is_a, utilisé par la recherche de champs...

Appel de méthode : code généré

code généré

```
ILP_Object ilptmp1;
{
  ILP_general_function ilptmp2;
  ILP_Object ilptmp3;
  ilptmp3 = o;
  ilptmp2 = ILP_find_method(ilptmp3, &ILP_object_print_method, 1);
  ilptmp1 = ilptmp2(NULL, ilptmp3);
}
```

Code généré pour `o.print()` :

- la méthode `print` est représentée par l'objet `ILP_object_print_method`, généré par le compilateur ;
- recherche de fonction C implantant `print` pour la classe de `o` ;
- appel, en passant `o` en argument.

  (rappel : le premier argument, est la clôture, inutile ici, donc on passe `NULL`)

Implantation des méthodes prédéfinies

Tout objet a une méthode `print` prédéfinie.

Nous avons vu la fonction d'affichage générique `ILP_print`,
mais elle n'a pas la signature d'une méthode !

$\Longrightarrow$ la méthode prédéfinie `ILPm_print` enrobe la fonction `ILP_print`.

ilp.c

```
ILP_Object ILPm_print (ILP_Closure useless, ILP_Object self)
{
    return ILP_print(self);
}
```

Rappel : les tables de méthodes générées pour nos classes référencent `ILPm_print` et `ILPm_classOf` !

Méthode utilisateur : code généré

code généré

```
ILP_Object ilp__m_2(ILP_Closure ilp_useless, ILP_Object self3, ILP_Object arg4)
{
    static ILP_Method ilp_CurrentMethod = &ILP_object_m1_method;
    static ILP_general_function ilp_SuperMethod = ilp__m_1;
    ILP_Object ilp_CurrentArguments[2];
    ilp_CurrentArguments[0] = self3;
    ilp_CurrentArguments[1] = arg4;
    // corps de la méthode
}
```

En-tête du code généré pour une méthode `m` à un argument `arg` :

- génère une fonction C `ilp__m_2` ;
- le `_2` distingue cette implantation de celle du parent (`ilp__m_1`) ;
- `self` est passé en argument, et est accessible au corps de la méthode.

Pour faciliter un éventuel appel à `super` :

- `ilp_SuperMethod` pointe sur la méthode `m` du parent.
  (connu statiquement à la compilateur)
- les arguments sont stockés dans `ilp_CurrentArguments`.

Appel de la méthode parent : super (1/2)

ilp.h

```
#define ILP_FindAndCallSuperMethod()                                    \
  (((ilp_SuperMethod != NULL)                                           \
    ? (*ILP_find_and_call_super_method)                                 \
    : (*ILP_dont_call_super_method) )( ilp_CurrentMethod, ilp_SuperMethod, \
                                       ilp_CurrentArguments))
```

super est simplement compilé en ILP_FindAndCallSuperMethod qui utilise les informations stockées en tête de méthode : ilp_CurrentMethod, ilp_SuperMethod, ilp_CurrentArguments.

Cas normal : appel de ILP_find_and_call_super_method.

Cas particulier : pas de méthode parent : ilp_SuperMethod == NULL
$\Longrightarrow$ appel de ILP_dont_call_super_method qui part en erreur.

Appel de la méthode parent : super (2/2)

ilp.c

```
ILP_Object ILP_find_and_call_super_method (ILP_Method current_method,
                                           ILP_general_function super_method,
                                           ILP_Object arguments[1])
{
    ILP_Object self = arguments[0];
    int arity = current_method->_content.asMethod.arity;
    switch ( arity ) {
    case 0: {
        return (*super_method)(NULL, self);
    }
    case 1: {
        return (*super_method)(NULL, self, arguments[1]);
    }
    ...
}
```

Simple appel par pointeur de fonction ;
en gérant tous les cas de nombre d'arguments (similaire à `ILP_invoke`).

Compilateur

Rappel : les étapes de compilation

Peu de choses à dire sur la partie Java.

Les grandes étapes n'ont pas changé par rapport à ILP1–3 :

- extension de la structure d'ASTC ;
  (transparent suivant)
- `Normalizer` : normalisation de l'AST en ASTC ;
  (pas de traitement spécifique ; il s'agit surtout d'appeler le visiteur sur les sous-expressions)
- `FreeVariableCollector`, `GlobalVariableCollector` :
  analyses statiques des variables libres et globales ;
  (idem : pas de traitement spécifique)
- `Compiler` : génération de code ;
  (long mais facile, une fois compris la forme du code généré)

$\Longrightarrow$ à lire chez soi.

Extension de l'ASTC

Dans `com.paracamplus.ilp4.compiler.interfaces`,
des nouvelles interfaces de nœuds ASTC :

- `IASTCprogram`
  ajoute à ILP3 une méthode `getClassDefinitions`
- `IASTCclassDefinition`
  associe un indice à chaque champ et méthode de la classe
- `IASTCmethodDefinition`
  information sur la méthode parent : `findSuperMethod`
- `IASTCclassRelated`
  ajoute `getDefiningClass`, réutilisé dans plusieurs nœuds ASTC
- `IASTCfieldRead`, `IASTCfieldWrite`
  nœud IAST d'ILP4, plus `getDefiningClass` pour le champ utilisé
- `IASTCinstantiation`
  nœud IAST d'ILP4, plus `getDefiningClass` pour la classe instanciée

et également l'extension du visiteur d'ASTC (`IASTCvisitor`, `IASTCvisitable`).

À lire chez soi : les implantations `ASTCprogram`, `ASTCclassDefinition`, `ASTCmethodDefinition`, `ASTCfieldRead`, `ASTCfieldWrite`, `ASTCinstantiation`.

Efficacité du code généré

Lectures et écritures de champ, appels de méthode :

- vérification de type dynamique, de type `instanceof`,
  coût linéaire en la profondeur de la hiérarchie ;
- puis coût constant d'accès à un tableau de champs ou méthodes.
  grâce au précalcul des indices par le compilateur

L'accès aux champs est plus rapide que pour l'interprète.
(ne dépend plus du nombre de champs)

Mais on pourrait améliorer le coût associé à la vérification de type !

Autres implantations des objets

Accès au champ : typage statique et héritage simple

Accès au champ dans un langage à la Java :

- comme en ILP, le champ est à un indice constant dans un tableau ;
- contrairement à ILP, la vérification de classe est statique ;

$\Longrightarrow$ coût constant à l'exécution.

Si le compilateur accepte l'expression `o.f`, alors :

- `o` dérive d'une classe `C` déclarant le champ `f` ;
- les instances de toutes les classes `D` héritant de `C` stockent la valeur de `f` au même indice que `C` ;
  avec l'héritage simple, si `D <: C`, la liste des champs de `D` commence par la liste des champs de `C`...

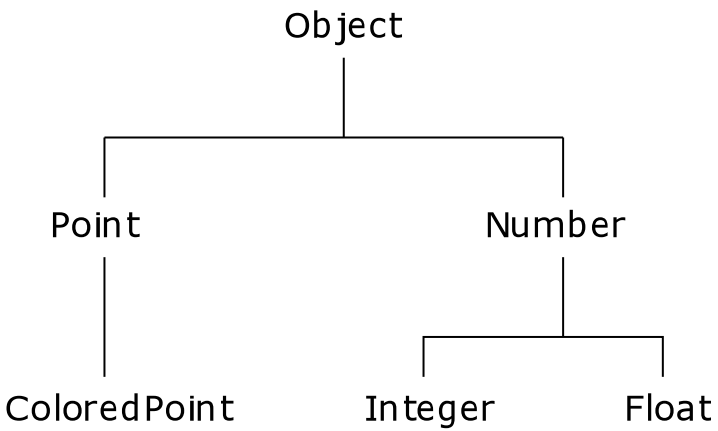
$\Longrightarrow$ une méthode compilée pour une instance de `C`, fonctionne avec toute instance d'une classe héritant de `C`.

Contrairement à ILP, deux classes indépendantes peuvent déclarer des champs de même nom.
éventuellement stockés à des indices différents ; l'information de type statique sert à les distinguer

Test `instanceof` efficace : matrice

Comme ILP, Java permet un test dynamique de classe, avec `instanceof`.

Exemple d'implantation : précalcul du résultat dans une matrice.

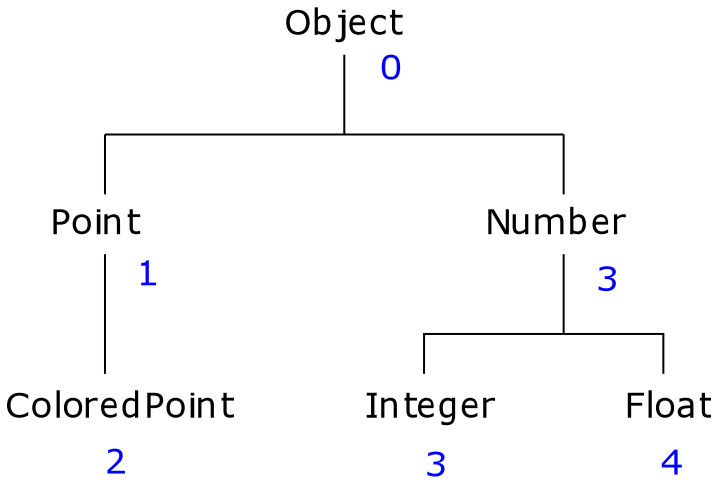


	Point	Col.	Num.	Int.	Float
Point	T				
Col.	T	T			
Num.			T		
Int.			T	T	
Float			T		T

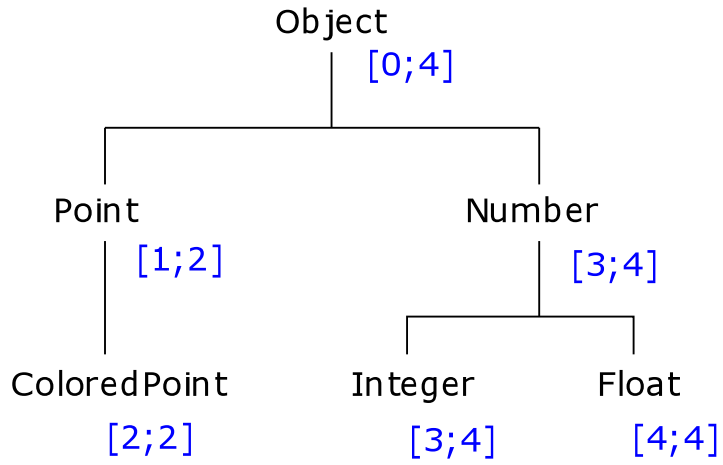
`o instanceof C` $\iff$ M[`classof o`][`C`]

Coût constant à l'exécution.
Mais peu efficace en mémoire.

Test `instanceof` efficace : arbre d'intervalles



indice $i(\texttt{C})$



intervalle d'indices $[i(\texttt{C}); m(\texttt{C})]$

Méthode alternative, compacte en mémoire :

- numéroter l'arbre des classes par un parcours en profondeur $i(\texttt{C})$;
- calculer l'intervalle d'indice pour chaque sous-arbre $[i(\texttt{C}); m(\texttt{C})]$ où $m(\texttt{C}) = \max\{i(\texttt{D}) \mid \texttt{D} <: \texttt{C}\}]$;

$\texttt{o instanceof C} \iff i(\texttt{classof o}) \in [i(\texttt{C}); m(\texttt{C})]$.

Interfaces et recherche de méthode

Les interfaces de Java proposent une forme limitée d'héritage multiple ;

- aucun changement à la recherche de champ ;
- mais rend l'appel de méthode plus complexe.

example Java

```
class A extends C implements I, J { void m() { ... } }
A obj1 = new A();  obj1.m();
C obj2 = (C)obj1;  obj2.m();
I obj3 = (I)obj1;  obj3.m();
J obj4 = (J)obj1;  obj4.m();
```

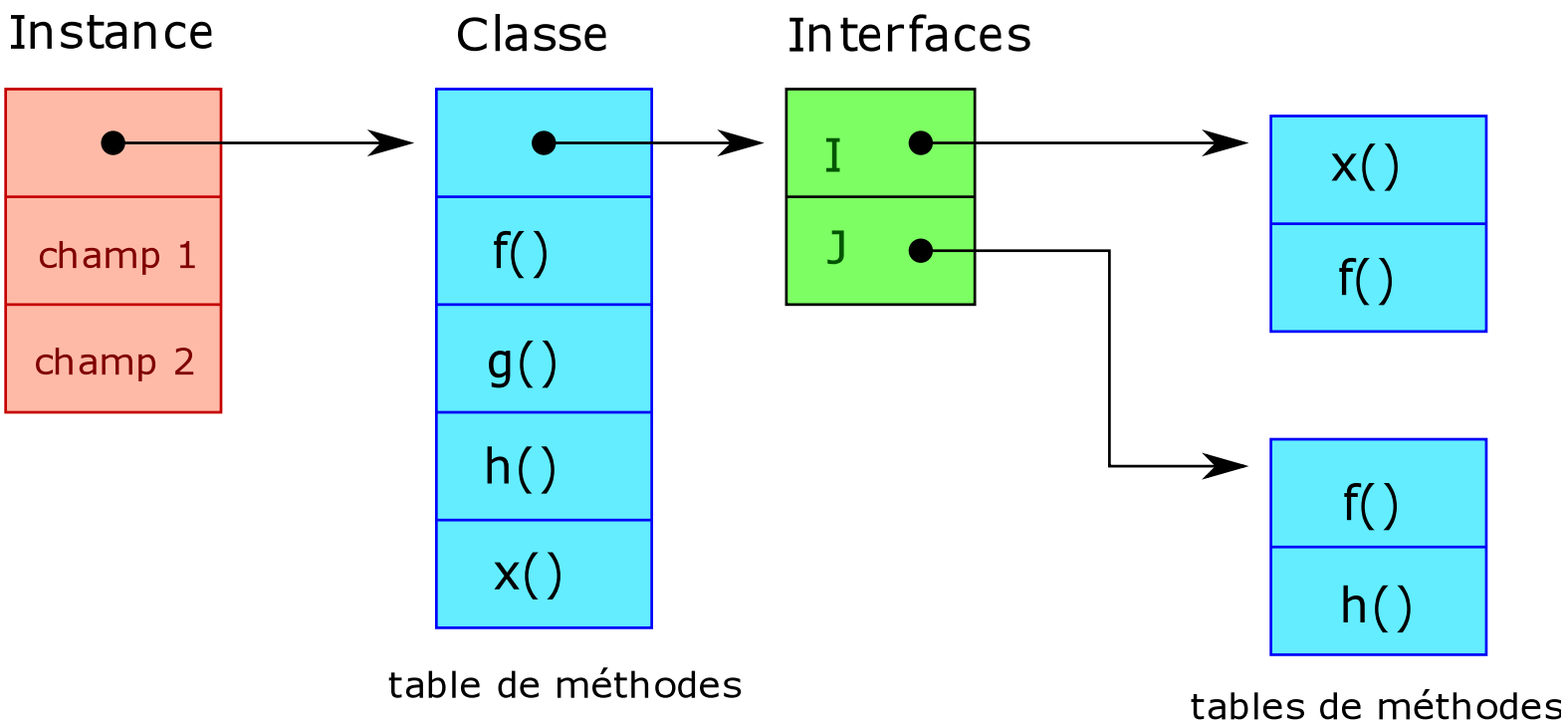
La méthode `m` de `A` peut aussi être appelée par un objet de type `C`, `I` ou `J`.

La classe a donc une table de méthodes supplémentaire par interface implantée.
L'appel de méthode par interface doit d'abord retrouver la bonne table. . .

$\Longrightarrow$ solutions : recherche linéaire, tables de hachage, caches, etc.

Le test `instanceof interface` est également plus complexe. . .
(l'approche par arbre d'intervalles ne fonctionne plus, à cause de l'héritage multiple)

Interfaces et recherche de méthode : illustration



Héritage multiple non virtuel, tremplin

C++ offre un support complet pour l'héritage multiple ;
$\Longrightarrow$ rend l'accès aux champs plus complexe.

Cas simple : héritage non virtuel.

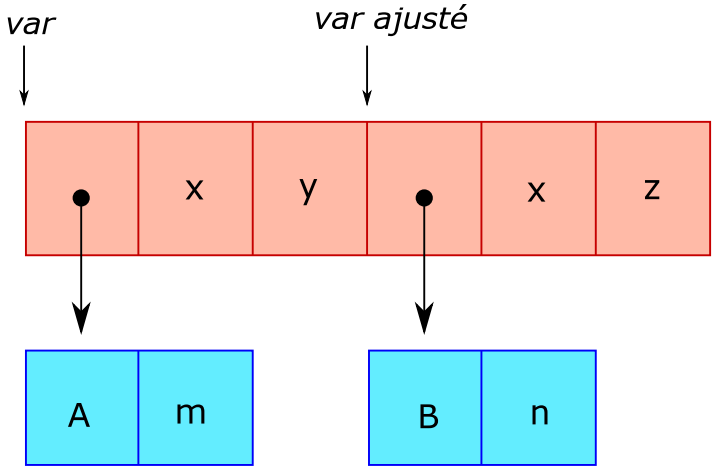
example C++

```
class A { int x, y; void m(); };
class B { int x, z; void n(); };
class C : public A, public B {  };
C var;
var.m(); var.n();
```

Encodage d'un objet de classe `C` :
on concatène le codage de `A` et de `B` !

`var.m()` peut être appelée directement.

`var.n()` nécessite un ajustement de `var`
pour pointer sur un objet de classe `B`.
(une méthode « tremplin » d'ajustement est synthétisée)



Classes de base virtuelles

Cas complexe : héritage virtuel.

Les occurrences multiples d'une classe de base sont partagées.

$\Longrightarrow$ impossible de stocker tous les champs dans un seul bloc contigu ; des blocs séparés sont nécessaires, liés par des pointeurs.

example C++

```
class A { int x; };
class B : public virtual A { int y; }
class C : public virtual A { int z; }
class D : public B, public C { ... }
B obj1; obj1.x;
D obj2; obj2.x;  ((B)obj2).x;  ((C)obj2).x;
```