MPIL (LI332) : COURS 10

Modèles de Programmation et Interopérabilité des Langages

Emmanuel Chailloux

Interopérabilité OCaml et Java

Plan du cours :

- motivations
- comparaison OCaml et Java
- JNI et bibliothèque camljava
- 1ère approche : 2 bibliothèques d'exécution avec O'Jacaré
  - O'Jacaré, un IDL simple
    - IDL + générateur de code
    - appel Java depuis OCaml
    - appel OCaml depuis Java
  - exemples d'utilisation
    - *Design patterns*
    - voir .net
  - limitations
- 2ème approche : 1 runtime avec ocaml-java
  - types Java en OCaml
  - création d'objets et appel de méthodes
  - accès facile à toute l'API Java

Motivations

1. augmenter les caractéristiques des langages : Java et OCaml
   - faciliter l'emploi de bibliothèques
2. sans sacrifier la sûreté : typage statique, GC
   - tout en conservant une efficacité raisonable
3. sans dénaturer son langage de prédilection

Caractéristiques d'OCaml

- Langage fonctionnel, exceptions, extension impérative
- Types de données de haut niveau + filtrage de motifs,
- Polymorphisme + typage *implicite*,
  - statiquement typé,
  - inférence de types,
  - types polymorphes (choix du type le plus général)
- Différents styles de programmation (dans un cadre commun de typage).
  - Programmation orientée objet (structuration par classes),
  - Modules paramétrés (style SML)
  - *Plus récemment* : labels et variants polymorphes.

Exemples d'inférence de types

- type fonctionnel :

```
let compose f g = fun x -> f (g x);;
```

$(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$

- type fonctionnel sur les listes :

`List.map` : $(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`

- type objet et fonctionnel :

```
let toStringNL o = o#toString() ^"\n";;
```

`< toString : unit` $\to$ `string ; .. >` $\to$ `string`

Présentation du langage Java

- Historique : 1994, Java SUN, James Gosling (Emacs, NeWS)
- Influences : C, C++, Lisp, ML, Modula3
- Caractéristiques :
  - langage à classes,
  - syntaxe à la C,
  - typé statiquement,
  - à sous-typage nominal,
  - intégrant polymorphisme objet, de surcharge, paramétrique et borné,
  - à chargement dynamique de classes
  - avec un mécanisme d'exceptions
  - avec un mécanisme d'introspection et de sérialisation

Comparaison Java / OCaml (1)

caractéristiques	Java	OCaml	caractéristiques	Java	OCaml
classes	√	√	sous-typage	√	√
accès champs	√		héritage ≡ sous-typage ?	oui	non
liaison tardive	√	√	surcharge	√	
liaison précoce	√		héritage multiple		√
typage statique	√	√	classes paramétrées	√	√
dynamique	√		paquetages/modules	√	√
			classes mutuellement récursives	√	√

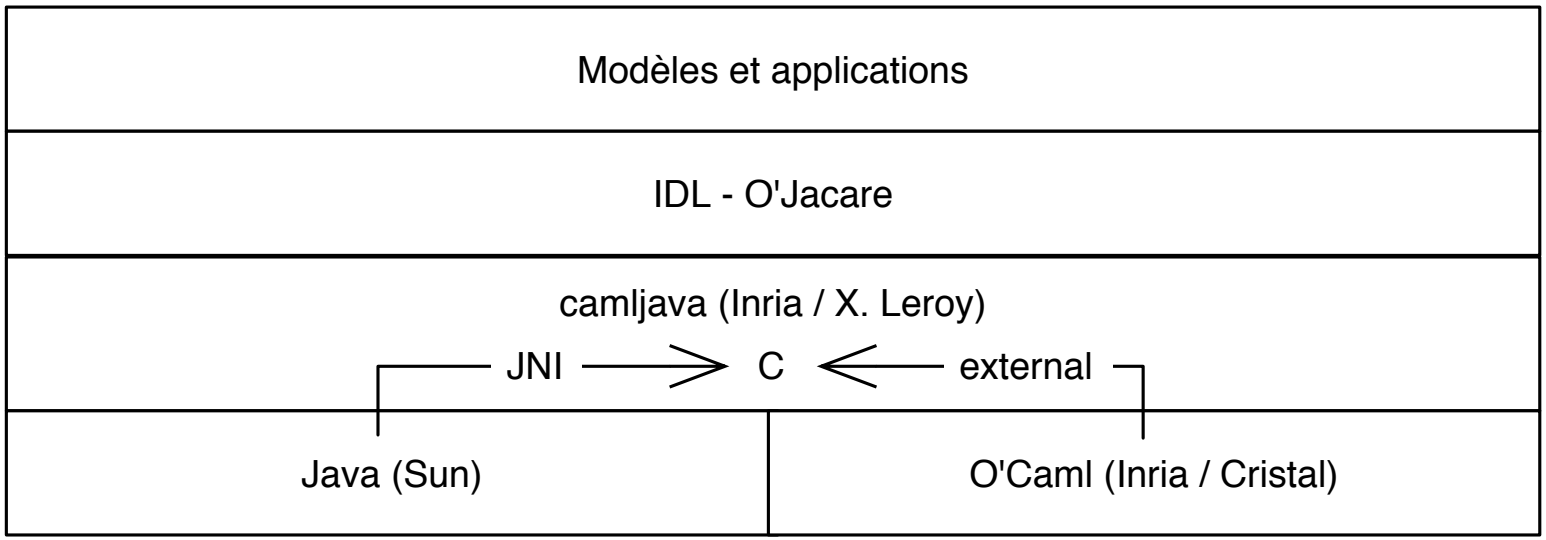
OCaml n'est pas un langage objet, mais a une extension objet

- une déclaration de classe définit un type objet et un constructeur ;
- type objet = noms et types des méthodes.

Comparaison Java/OCaml (2)

- accès champs :
  - en Java : selon la nature des attributs de visibilité les variables de classes ou d'instances sont accessibles ;
  - en OCaml : uniquement *via* un appel de méthode ;
- liaison statique : en OCaml les méthodes statiques sont des fonctions globales d'un module et les variables de classes des déclaration globales ;
- typage dynamique : pas de *downcast* en OCaml
- surcharge : pas de surcharge en OCaml mais le type de `self` peut apparaître dans le type d'une méthode qui pourra être redéfinie (*overriding*) dans une sous-classe ;
- pas d'héritage multiple pour les classes Java, seulement pour les interfaces ;
- les modules simples d'OCaml correspondent aux parties publiques des paquetages Java ; la notion de modules paramétrés est inexistante en Java ;

Architecture de l'interface



- point d'entrée d'un programme en OCaml
- compilateur OCaml : byte-code ou natif
- compilateur Java : byte-code

- Java Native Interface (JNI) : interface de Java avec C
- external : interface d'OCaml avec C

JNI et bibliothèque `camljava`

Initialisation de la JVM

Manipulation des classes et introspection

- Recherche par nom qualifié
- Identifiant d'une méthode par sa signature

Appel de méthodes

- Appel avec un tableau d'arguments
- Assure la conversion des type de base

Type objet unique `jobject` (C), `Jni.obj` (OCaml)

Exception Encapsulation

GC Met en racine les objets avant de passer une référence

Callback Définit quelques classes Java utiles.

O'Jacare, un *IDL* simple - 1/2

Faire correspondre 1 objet Java à 1 objet OCaml

À l'intersection des deux modèles

- définition de classe, classe abstraite et interface
- héritage simple pour les classes
- héritage multiple pour les interfaces
- pas de surcharge
  (mais un mécanisme d'alias de nom)
- pas de classe paramétrée

Sites

`http://www.pps.univ-paris-diderot.fr/~henry/ojacare`
`https://sites.google.com/site/keigoattic`

O'Jacare, un générateur de code - 2/2

Passage des arguments

- par partage pour les objets (ex : java.lang.String)
- par recopie pour les types de base (ex : int, string)

Typage La cohérence des type de l'IDL est vérifiée :

- à la compilation avec OCaml (et en partie avec Java)
- au chargement avec Java (introspection)

En conclusion partielle :

- privilégie le sens d'appel Java depuis OCaml
- sens inverse via un *callback*

Le fichier d'IDL est une description simplifiée de classe Java

Encapsulation et typage : Interface Colorée

IDL : `point.idl`

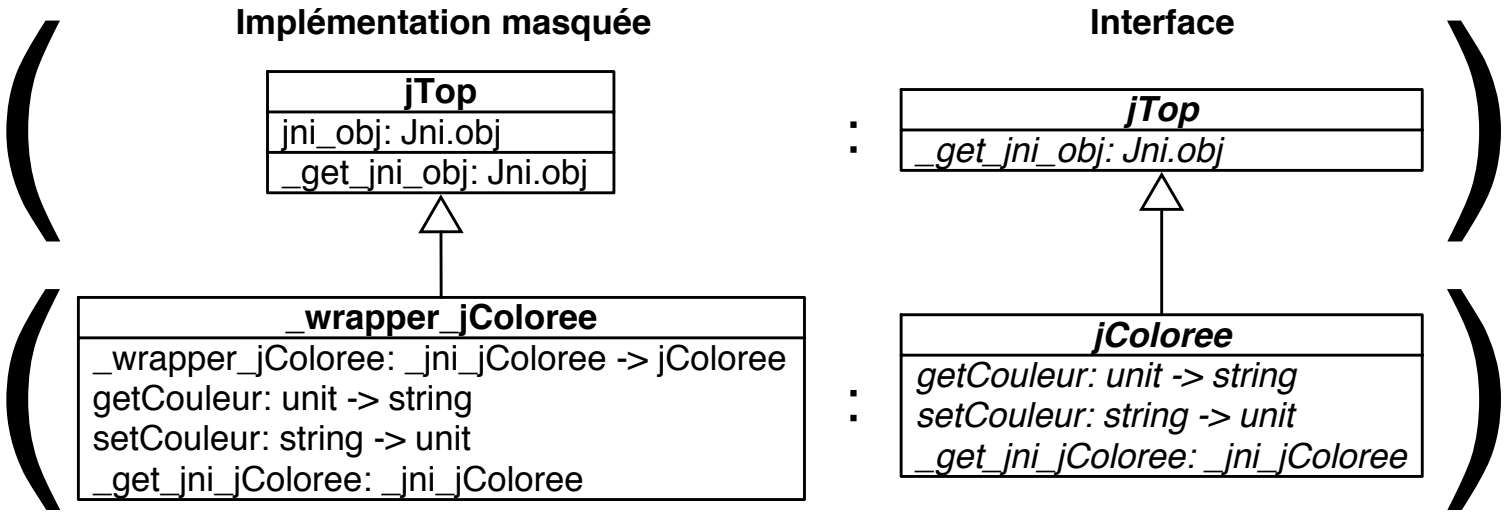
```
1  interface Colored {
2
3    string getColor();
4    void setColor(string);
5
6  }
```

OCaml : `point.ml`

Type abstrait `_jni_jColoree`

Type objet `jColoree`

Capsule `_wrapper_jColoree`



Constructeur : Classe Point

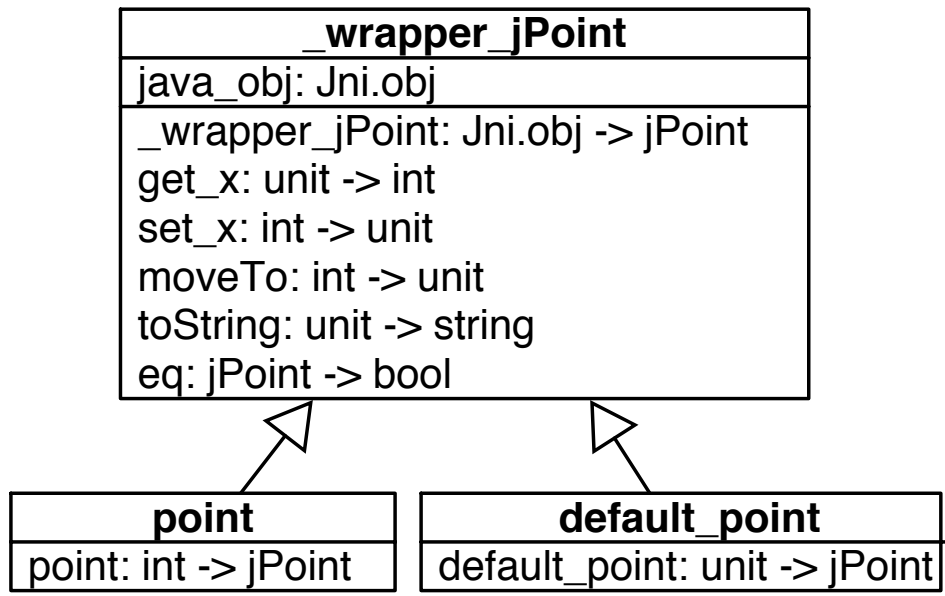
IDL : `point.idl`

```
1  class Point {
2   int x;
3   int y;
4   [name default_point] <init> ();
5   [name point] <init> (int,int);
6   void moveto(int,int);
7   void rmoveto(int,int);
8   string toString();
9   void display();
10  double distance();
11  boolean eq(Point);
12  static void main(string [] arg);
13 }
```

OCaml : `point.ml`

Types et capsule

Classes utilisateurs `default_point`, `point`

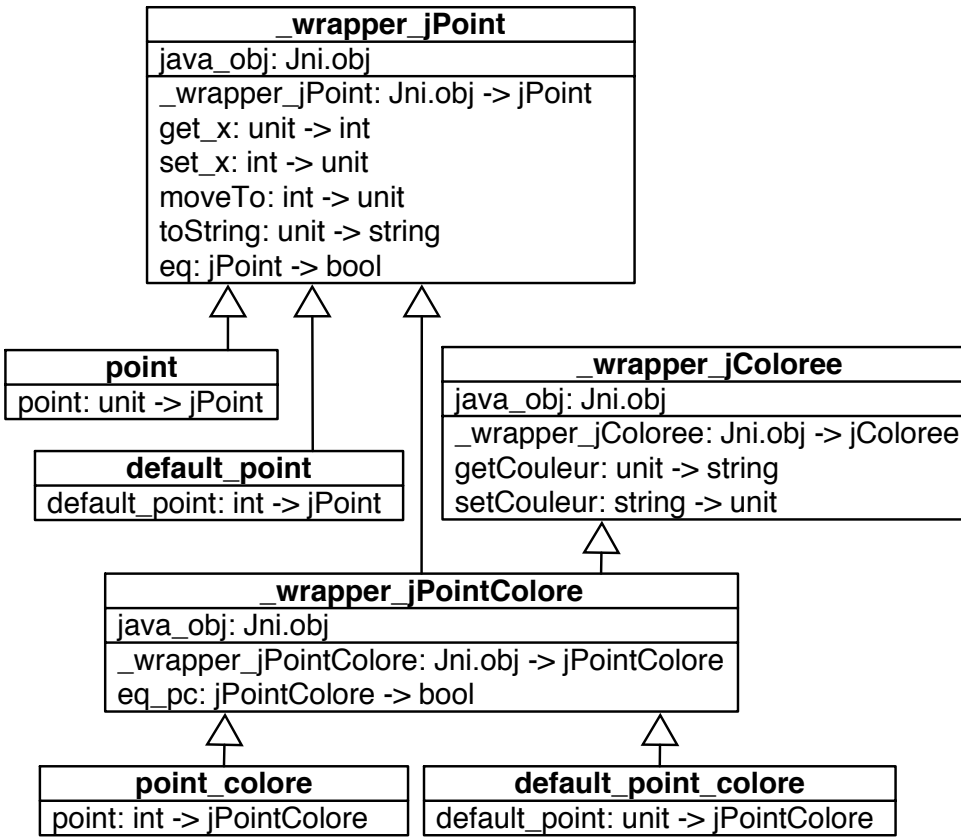


Héritage : Classe PointColore

IDL : point.idl

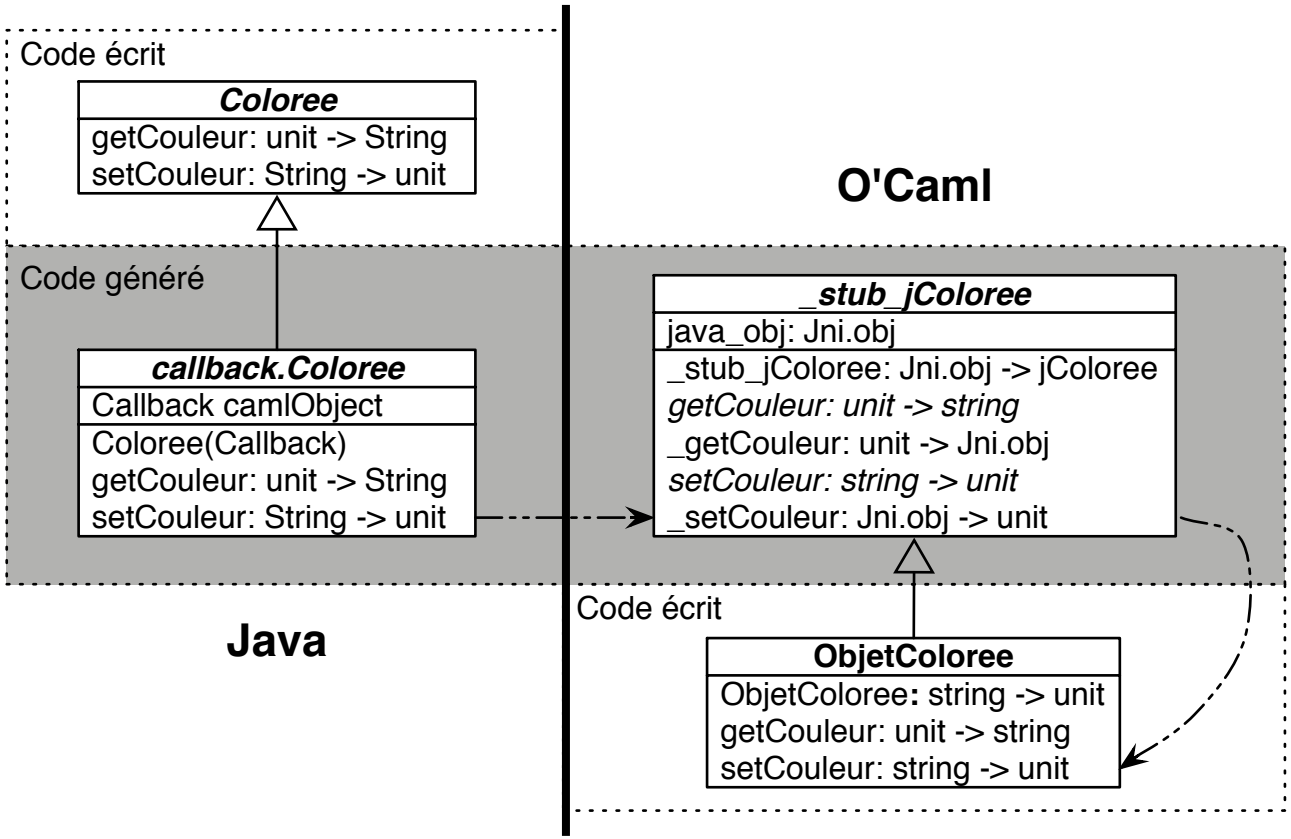
```
1  [callback] class ↩
       ColoredPoint extends ↩
       Point implements ↩
       Colored {
2   [name ↩
        default_colored_point↩
        ] <init> ();
3   [name colored_point] <↩
        init> (int,int,string↩
        );
4   [name eq_colored_point] ↩
        boolean eq(↩
        ColoredPoint);
5    static void main(string ↩
        [] arg);
6  }
```

OCaml : point.ml



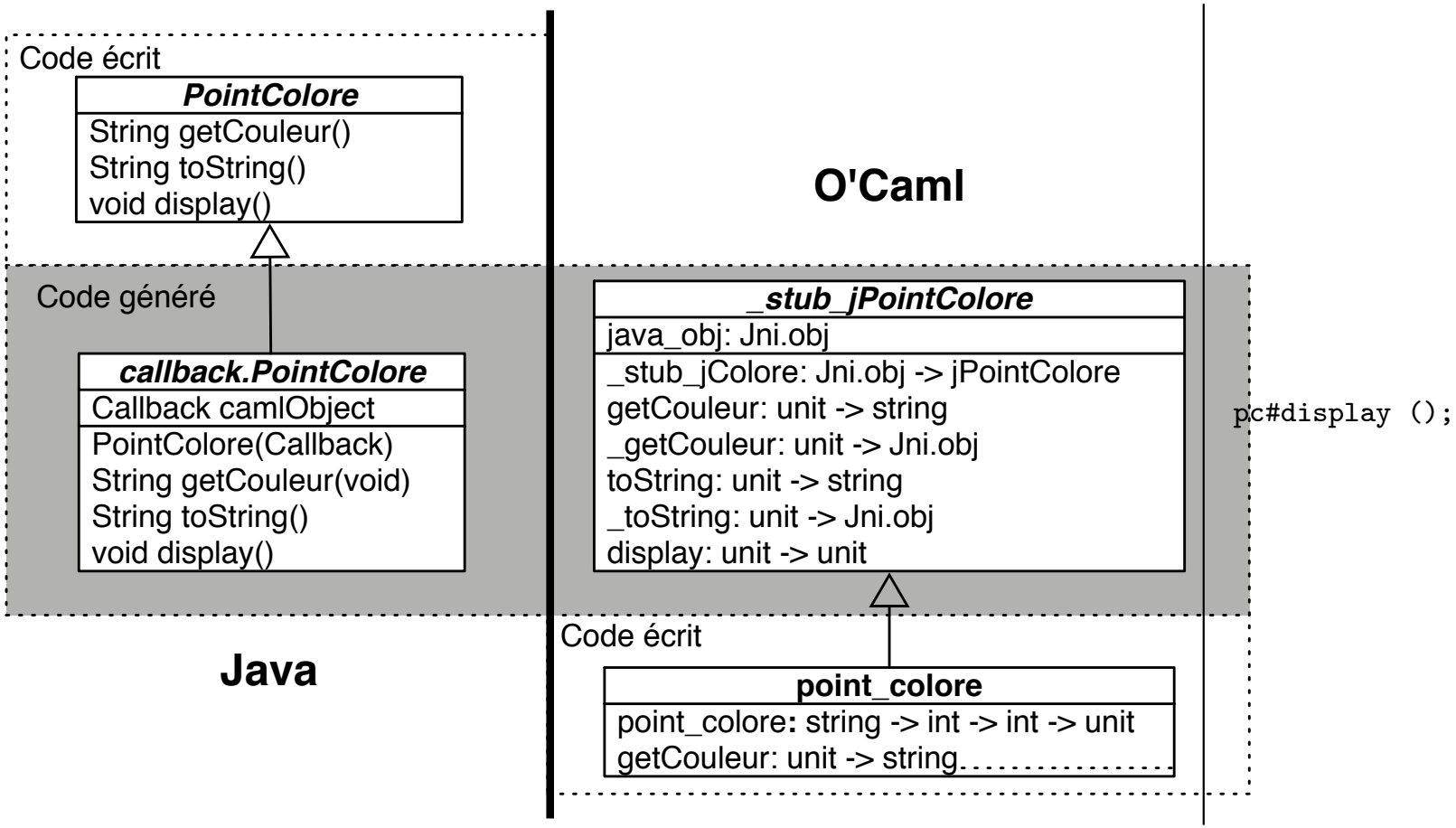
Attention : pas de liaison tardive entre les deux mondes ici !

Attribut *callback* et Interface Java



Une interface Java déclarée `callback` dans l'IDL, sera implémentée en OCaml.

Attribut *callback* et Classe Java



Exemple - Visiteur en OCaml 1/2

Modèle de formule logique en Java : visiteurML.idl

```
1  abstract class Formule {
2    abstract void accepte(Visiteur v);
3  }
4
5  class Constante extends Formule {
6    [name constante] <init>(boolean cst);
7    boolean valeur();
8    void accepte(Visiteur v);
9  }
10 // Variable, OpBin ...
11
12 interface Visiteur {
13   [name visite_cst] void visite(Constante c); // Variable,↩
         OpBin, ...
14 }
15
16 [callback] interface VisiteurML implements Visiteur {
17   string get_res();
18 }
```

Exemple - Visiteur en OCaml 2/2

Modèle de formule logique en Java : visiteur.ml

```
1  class visiteur_ml =
2    object (self)
3      inherit _stub_jVisiteurML ()
4      val buf = Buffer.create 80
5
6      method visite_cst cst =
7        Buffer.add_string buf (if cst#valeur () then "true" ↩
             else "false")
8
9      method visite_non non =
10       let sf = non#sous_formule () in
11       Buffer.add_string buf "!(";
12       sf#accepte (self :> jVisiteur);
13       Buffer.add_string buf ")"
14
15     (* ... *)
16     method get_res () = (* ... *)
17
18   end
```

Exemple - Visiteur en Java 1/3

Modèle de formule logique en OCaml : visiteurJava.idl

```
1  [callback] interface SyntAbs {
2    void accepte(Visiteur v);
3  }
4
5  [callback] interface MainML {
6      SyntAbs creerConstante(boolean valeur);
7      SyntAbs creerNon(SyntAbs sous_formule); // Variable, ↩
           OpBin, ...
8  }
9
10 interface Visiteur {
11     void visite_cst(boolean valeur);
12     void visite_non(SyntAbs sous_formule); // Variable, ↩
           OpBin, ...
13 }
14
15 interface MainJava {
16     static void main(String [] argv, MainML mainMl);
17 }
```

Exemple - Visiteur en Java 2/3

Modèle de formule logique en OCaml : MainJava.java

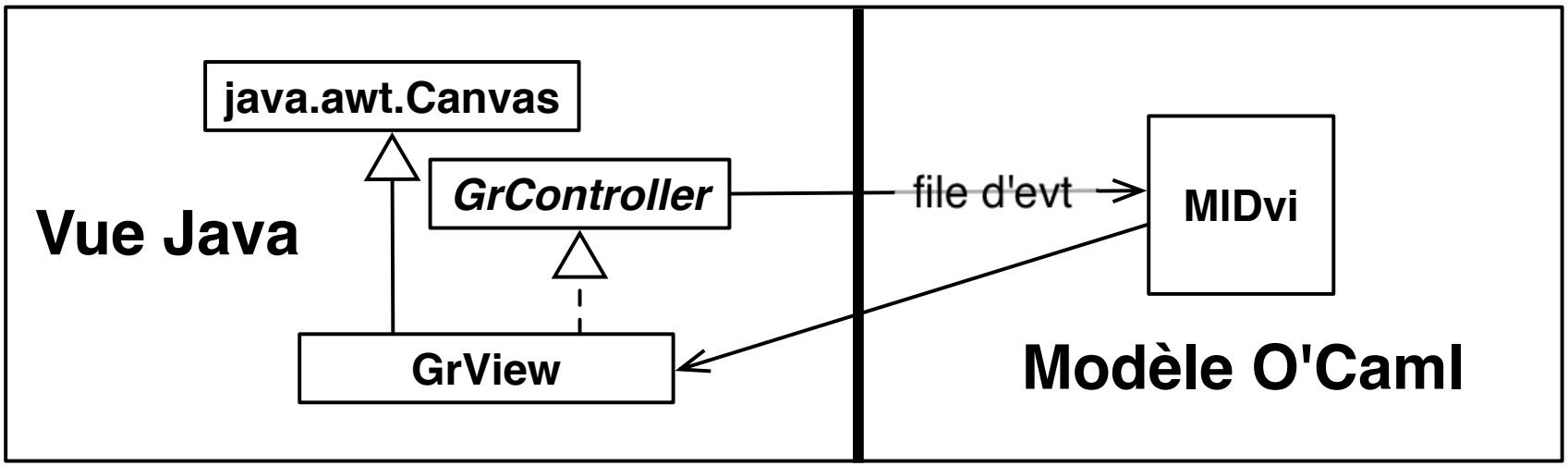
```
1  public class VisiteurToString {
2      public String res;
3      public VisiteurToString() { res = ""; }
4      void visite_cst(boolean b) { res += b; }
5      void visite_non(SyntAbs sf) {
6        res += "!("; sf.accepte(this); res += ")";
7      }
8      // ...
9  }
10
11 public class MainJava {
12   public static void main(String[] argv, MainML mainML) {
13     SyntAbs f = mainML.creerNon(mainML.creerConstante(true↩
           ));
14     VisiteurToString v = new VisiteurToString();
15     f.accepte(v);
16   }
17 }
```

Exemple - Visiteur en Java 3/3

Modèle de formule logique en OCaml : syntAbs.ml

```
1  type syntAbs =
2    | Cst of bool | Var of string
3    | Non of syntAbs
4    | Et of syntAbs * syntAbs | Ou of syntAbs * syntAbs
5
6  class syntAbs t =
7    object (self)
8      inherit VisiteurJava._stub_jSyntAbs ()
9      val synt_abs = t
10     method accepte v =
11       match synt_abs with
12       | Cst b -> v#visite_cst b
13       | Var nom -> v#visite_var nom
14       | Non sf -> v#visite_non (new formule sf :> jFormule↩
             )
15       (* ... *)
16   end
```

Exemples - MVC : visionneuse DVI



`jDvi` tout en Java - AWT

`mlDvi` tout en OCaml - X11 (Graphics)

`ojDvi` mixte : OCaml - AWT

	1p	8p	50p	100p	200p	300p
`jDvi`	3,4	6,0	11,7	16,0	26,5	35,5
`ojDvi`	2,0	3,0	6,9	10,0	18,0	25,0
`mlDvi`	0,3	0,6	1,1	1,6	2,4	3,0

JDK de Sun sous Linux

Enrichir les modèles

- Héritage multiple sur des classes Java
- Faire du *downcasting* d'objets Java en OCaml.

Héritage multiple de classes Java

fichier : `rect.idl`

```
1  package mypack;
2  class Point {
3   [name point]
4   <init> (int,int);
5  }
6
7  class GraphRectangle {
8   [name graph_rect]
9    <init> (Point, Point)↩
         ;
10  string toString();
11 }
12
13 class GeomRectangle {
14  [name geom_rect]
15  <init> (Point, Point);
16 }
```

programme OCaml

```
1  open Rect;;
2  class geom_graph_rect p1 p2 ↩
       =
3  object
4    inherit geo_rect p1 p2 as ↩
         super_1
5    inherit graph_rect p1 p2 ↩
         as super_2
6  end;;
7
8  let p1 = new point 10 10;;
9  let p2 = new point 20 20;;
10 let ggr =
11  new geom_graph_rect p1 p2;;
12 Printf.printf "area=%g\n"
13               (ggr#aera())
14 Printf.printf "toStr=%s\n"
15               (ggr#toString())
```

Downcasting d'objets Java en OCaml

```
let l = [ (ml_cp :> jPoint; (wml_cp :> jPoint)];;
val l = jPoint list = <obj>
let lc = List.map
          (fun x -> jColoredPoint_of_top (x :> top))
          l;;
val lc = jColoredPoint list
```

- La hiérarchie de classes OCaml engendrée a comme racine la `top`,
- `O'Jacaré` définit les fonctions de conversion de types de `top` vers les classes descendantes.

Discussion et travaux futurs

- Limitations

  Threading : communication limitée au thread principal
  GC : les objets `callback` ne sont jamais désalloués
  main Java : pas d'instrospection en OCaml.

- Améliorations

  finalisation : pour les objets circulaires mixtes

- Applications
  - interfaces graphiques Java
  - parsers OCaml
- Autre plate-forme `.net`
  - OCaml.NET : projet ocamil
  - O'Jacaré pour `C#`

Projet ocaml-java

- s'interfacer facilement avec tout Java en ciblant Java pour le compilateur de byte-code
- Xavier Clerc (Inria)
- distribution : http ://ocamljava.x9c.fr/preview/
- article : OCaml-Java : Typing Java Accesses from OCaml Programs (IFL 2013)

Environnement

différents outils

- compilation
  - ocamljava ex.ml produit un .jar
- exécution : par un interprète de la machine virtuelle Java
  - java -jar ex.jar
- le module Java

```
1 let make_frame ttl w h lbl act =
2 let open Java in
3 let f = make "JFrame(String)" ttl in
4 let b = make "JButton(String)" lbl in
5 let p = call "JFrame.getContentPane()" f in
6 call "JFrame.setSize(_,_)" f w h;
7 call "JButton.addAciontListener(_)" b act;
8 call "JFrame.add(Component)" p b;
9 f
```

Difficultés

- manipuler les types Java en Ocaml
- convertir les valeurs des types de base d'un monde à l'autre
- construire des objets Java et les exécuter en OCaml
- récupérer les exceptions Java en OCaml
- avoir accès à l'ensemble des API Java
- pouvoir appeler du code OCaml en Java

Codage des types Java

Utilisation des types fantômes et des variants polymorphes.

- types fantômes

```
1 type ('a, 'b) resource = { value : 'a }
```

- variants polymorphes

```
1 type flags = [ `Public; `Private; `Synchronized ]
2 type class_flags = [ `Public; `Private ]
```

- mixte

```
1 val make_ro : 'a -> ('a, [ `Read ]) resource
2 val make_rw : 'a -> ('a, [ `Read | `Write ]) resource
3 val read : ('a, [> `Read]) resource -> 'a
4 val write : ('a, [> `Write]) resource -> 'a -> unit
```

Types de base

type Java	type OCaml	remarques
boolean	bool	
byte	int	entier 63 bits
char	int	entier 63 bits
double	float	double précision
float	float	double précision
int	int32	
long	int64	
short	int	entier 63 bits
pack.Class	_'a java instance	

Espace de noms et hiérarchie de classes

- notation quotée : `java'lang'Object`
  `['java'lang'Object ] java_instance`

exemple:

```
1  val make_frame :
2    java'lang'String java_instance ->
3    int32 ->
4    int32 ->
5    java'lang'String java_instance ->
6    java'awt'event'ActionListener java_instance ->
7    javax'swing'JFrame java_instance
```

Les types Java

java_constructor	signature d'un constructeur "java.lang.Object()"
java_method	signature d'une méthode "java.lang.Object.wait() :void"
java_field_get	signature d'un attribut "java.lang.Thread.MAX PRIORITY :int"
java_field_set	signaturd d'un attribut "java.lang.Thread.MAX PRIORITY :int"
java_type	classe, interface ou type Array "java.lang.String"
java_proxy	type d'une interface "java.lang.Comparable"

On utilise la même ruse que `format` (printf) : par exemple le `java_constructeur` est décrit par une chaîne de caractères mais ce n'est pas une chaîne de caractères mais bien un constructeur Java.

Création et manipulation d'instance Java

module Java :

```
make : 'a java_constructor -> 'a
call : 'a java_method -> 'a
get : 'a java_field_get -> 'a
set : 'a java_field_set -> 'a
instanceof : 'a java_type -> 'b java_instance -> bool
cast : 'a java_type -> 'b java_instance -> 'a
proxy : 'a java_proxy -> 'a
```

```
let obj = Java.make "java.lang.Object()"
let itg = Java.make "java.lang.Integer(int)" 123l
```

1er exemple

Création d'instance et appel de méthode:

```
1  let main () =
2    let p = Java.make "Point()" in
3      Java.call "Point.rmoveto(_,_)" p 2 3;
4      let r = Java.call "Point.distance():float" p in
5      r ;;
6
7  main();;
```

Exceptions

Les exceptions Java sont récupérées en OCaml.

```
exception Java_exception of java'lang'Throwable ↩
    java_instance
```

Exemple :

```
open Package'java'io

let f ... =
  try
    ...
  with
  | Java_exception t when Java.instanceof "IOException" t ↩
      -> Java.cast "IOException" t
  | Java_exception _ ->
    (* any other Java exception *)
    ...
  | _ ->
    (* any other OCaml exception *)
    ...
```

Exemple (1)

```
1  open Package'java'awt
2  open Package'java'awt'event
3  open Package'javax'swing
4
5  let () =
6     let str = JavaString.of_string in
7     let open Java in
8     let title = str "Celsius Converter" in
9     let frame = make "JFrame(String)" title in
10    let temp_text = make "JTextField()" () in
11    let celsius_label = make "JLabel(String)" (str "Celsius↩
          ") in
12    let convert_button = make "JButton(String)" (str "↩
          Convert") in
13    let farenheit_label = make "JLabel(String)" (str "↩
          Farenheit") in
```

Exemple (2)

```
1   let handler = proxy "ActionListener" (object
2     method actionPerformed _ =
3       try
4         let c = call "JTextField.getText()" temp_text in
5         let c = call "Double.parseDouble(_)" c in
6         let f = (c *. 1.8) +. 32.0 in
7         let f = Printf.sprintf "%f Farenheit" f in
8         call "JLabel.setText(_)" farenheit_label (str f)
9       with Java_exception je ->
10        let je_msg = call "Throwable.getMessage()" je in
11        let je_msg = JavaString.to_string je_msg in
12        let msg = str (Printf.sprintf "invalid float ↩
              value (%s)" je_msg) in
13        let error = get "JOptionPane.ERROR_MESSAGE" () in
14        call "JOptionPane.showMessageDialog(_,_,_,_)"
15          frame msg title error
16  end) in
```

Exemple (3)

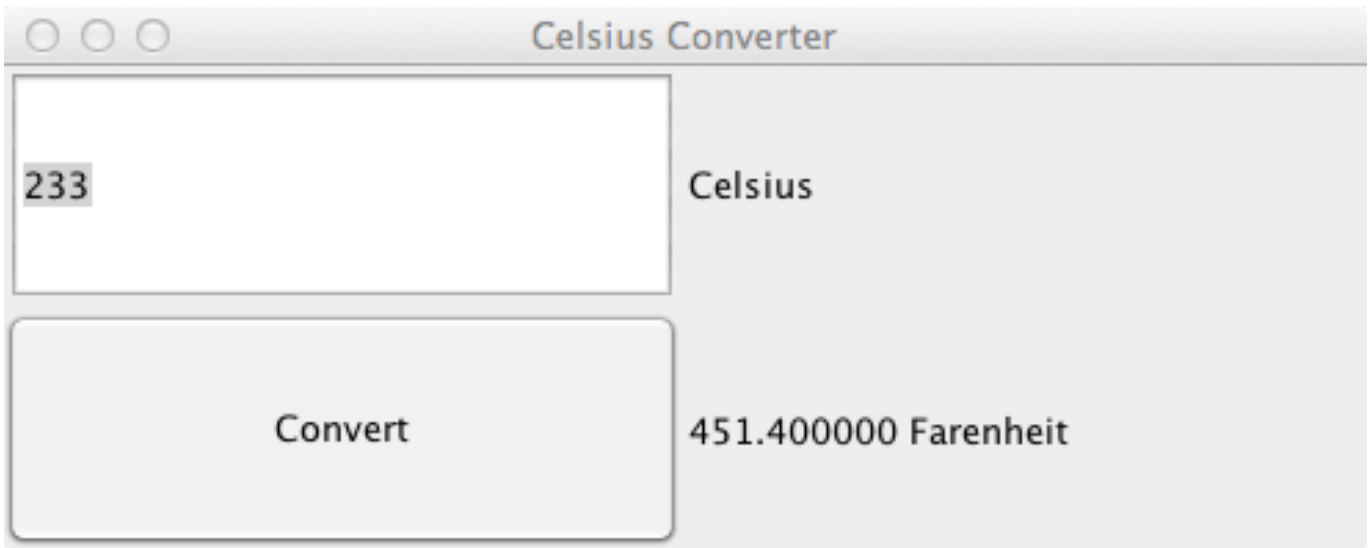
```
1   let () = call "JButton.addActionListener(_)" ↩
        convert_button handler in
2   let layout = make "GridLayout(_,_,_,_)" 2l 2l 3l 3l in
3   call "JFrame.setLayout(_)" frame layout;
4   ignore (call "JFrame.add(Component)" frame temp_text);
5   ignore (call "JFrame.add(Component)" frame ↩
        celsius_label);
6   ignore (call "JFrame.add(Component)" frame ↩
        convert_button);
7   ignore (call "JFrame.add(Component)" frame ↩
        farenheit_label);
8   call "JFrame.setSize(_,_)" frame 300l 80l;
9   let exit = get "WindowConstants.EXIT_ON_CLOSE" () in
10  call "JFrame.setDefaultCloseOperation(int)" frame exit;
11  call "JFrame.setVisible(_)" frame true
```

Exemple (4)

compilation et lancement :

```
ocamljava -c source.ml
ocamljava -o myprog.jar source.cmj

java -jar myprog.jar
```



Appel à du code OCaml de Java

En direct :

```
1  import javax.script.*;
2  import org.ocamljava.runtime.support.scripting.↩
       OCamlContext;
3
4  public final class SimpleInterpreted {
5
6      private static final String SCRIPT =
7          "external get_binding : string -> 'a = \"↩
               script_get_binding\";;\n" +
8          "let n = Int32.to_int (get_binding \"repetitions↩
               \");;\n" +
9          "let s : string = get_binding \"message\";;\n" +
10         "for i = 1 to n do print_endline s done;;\n";
```

(suite)

```
1
2       public static void main(final String[] args) throws ↩
            Throwable {
3           final ScriptEngineManager manager = new ↩
                ScriptEngineManager();
4           final ScriptEngine engine = manager.↩
                getEngineByName("OCaml");
5           final ScriptContext ctxt = new OCamlContext();
6           ctxt.getBindings(ScriptContext.ENGINE_SCOPE).put("↩
                repetitions", 3);
7           ctxt.getBindings(ScriptContext.ENGINE_SCOPE).put("↩
                message", "hello!!!");
8           engine.eval(SCRIPT, ctxt);
9       }
10  }
```

Conclusion

- O'Jacaré
  - nécessite un IDL
  - ne permet pas de charger l'ensemble des API Java
  - difficultés des 2 runtimes : GC, threads, ...
  - sous-classage entre mondes
- ocaml-java
  - 1 seul runtime
  - meilleure intégration (types, classes, ...)
  - nombreuses extensions : donc les API de concurrence
  - pas de sous-classage entre les mondes
- intégration des deux mondes ?