Interopérabilité entre OCaml et Java

Béatrice Carré

13 mai 2014

Objectif

Definition

L'interopérabilité entre deux langages : est la capacité d'un programme écrit dans un certain langage d'utiliser un programme dans un autre langage.

- Pour quoi faire ?
  Tirer parti des spécificités de chaque langage
- Caractéristiques d'une interopérabilité efficace :
  - accès simple ou implicite à l'autre langage
  - gestion mémoire et des exceptions
  - bonne gestion des caractéristiques différentes

Comparaison des deux mondes

L'interopérabilité se fait sur le modèle objet de chacun

caractéristiques	*Java*	*OCaml*
accès champs	selon la visibilité	via appels de méthode
var./méth. statiques	✓	fonct./décl. globales
typage dynamique	✓	×
surcharge	✓	×
héritage multiple	pour les interfaces	✓
modules paramétrés	×	✓

⇒ Il faut réduire les possibilités d'un outil à l'intersection des deux mondes

O'Jacaré : schéma global

Pour un accès à des classes Java :
O'Jacaré génère leurs classes encapsulantes à partir d'un fichier dans lequel sont décrites les classes qu'on veut manipuler.

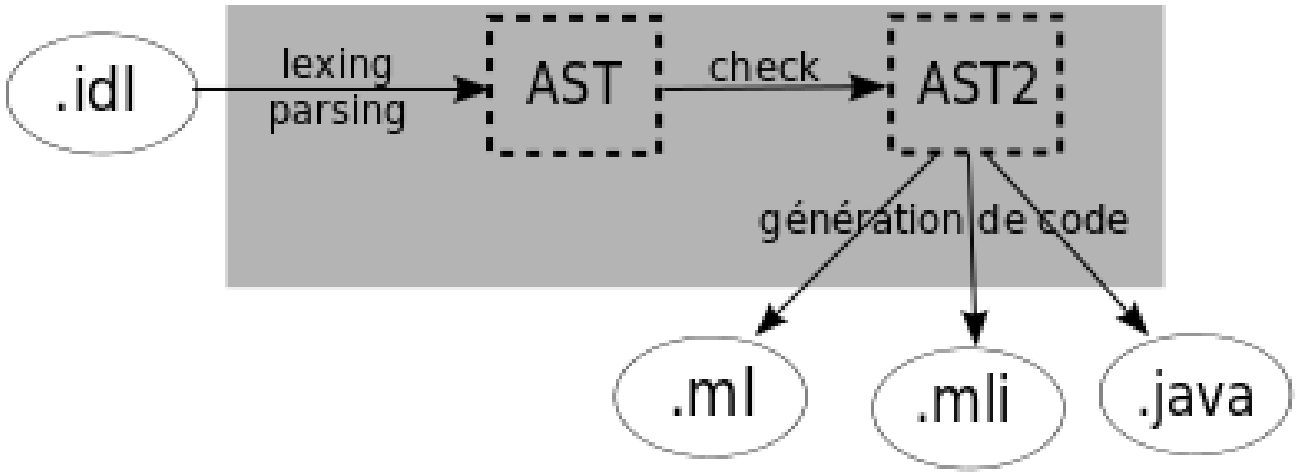
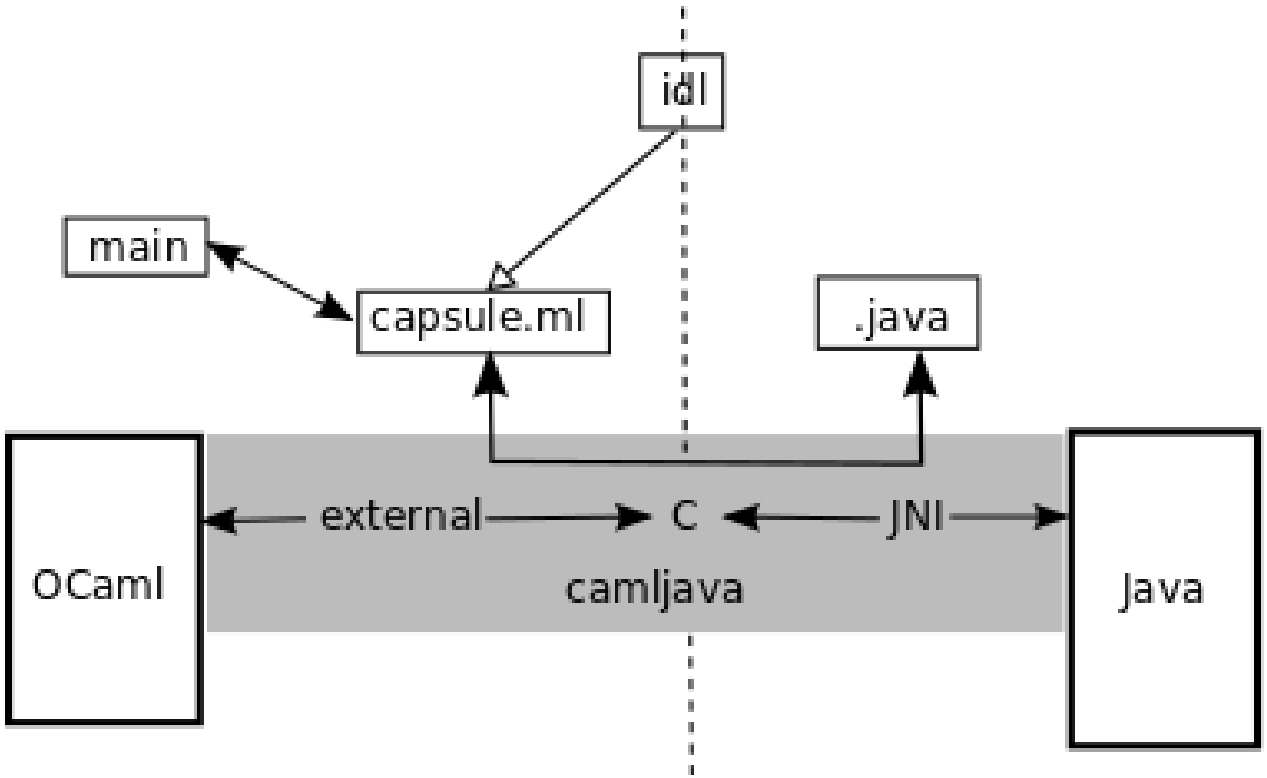


Figure: La génération de code d'O'Jacaré

O'Jacaré : schéma global

- La capsule générée permet à l'utilisateur de faire des appels transparents à des méthodes Java.
- Camljava gère la communication entre les deux mondes :
  - Recherche des classes par nom et des méthodes par signature
  - Conversion des types de base est assurée



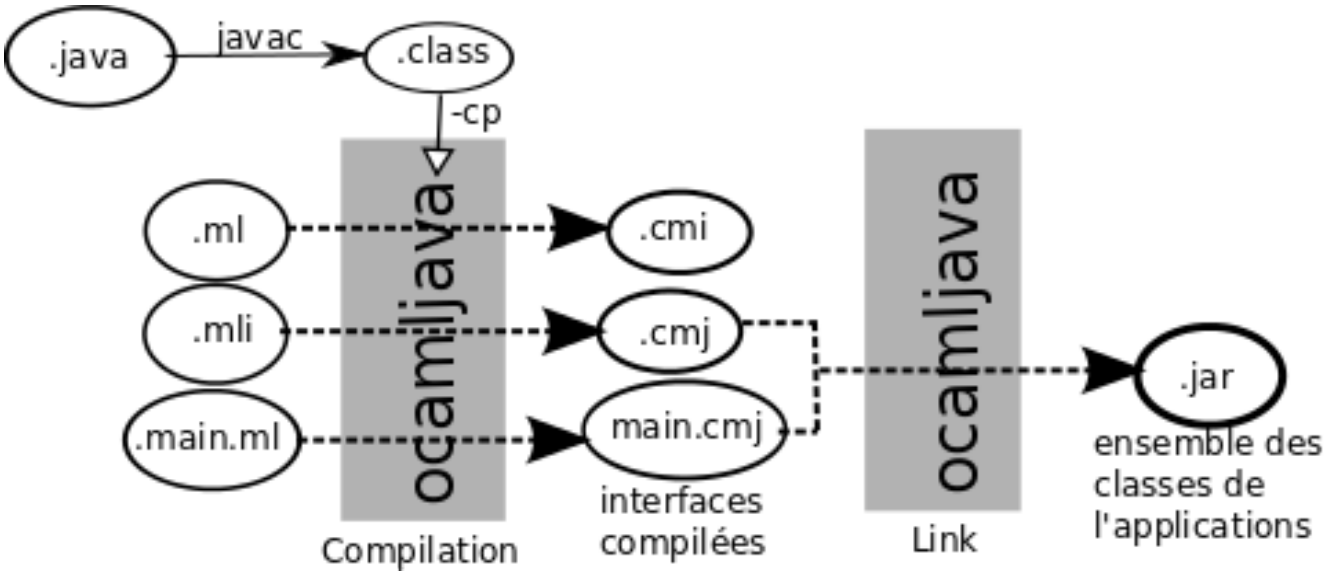
Code Java

```
1  package mypack;
2  public class Point {
3     int x;
4     int y;
5     public Point() {
6        this.x = 0;
7        this.y = 0;
8     }
9     public Point(int x, int y) {
10       this.x = x;
11       this.y = y;
12    }
13    public void moveto(int x, int y){
14       this.x = x;
15       this.y = y;
16    }
17    public String toString() {
18       return "("+x+","+y+")";
19    }
20    public double distance() {
21       return Math.sqrt
                          t Hts, y);
```

O'Jacaré : exemple d'idl

```
package mypack;
class Point {
  int x;
  int y;
  [name default_point] <init> ();
  [name point] <init> (int,int);
  void moveto(int,int);
  string toString();
  boolean eq(Point);
}
interface Colored {
  string getColor();
  void setColor(string);
}
[callback] class ColoredPoint extends Point implements Colored {
  [name default_colored_point] <init> ();
  [name colored_point] <init> (int,int,string);
  [name eq_colored_point] boolean eq(ColoredPoint);
}
```

OCaml-Java : schéma global



Compilation vers du bytecode Java

$\Rightarrow$ un seul runtime Pas de problème de gestion mémoire, de communications .

OCaml-Java : l'accès au monde Java

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

Accès aux bibliothèques Java et à du code utilisateur grâce à ce module.

```
let color = JavaString.of_string "bleu"
and x = Int32.of_int 1
and y = Int32.of_int 2 in
let p = Java.make "mypack.ColoredPoint(int,int,java.lang.String)" s y color
in
   Java.call "mypack.Point.eq(mypack.Point):boolean" p p2
```

Fusion des deux approches

O'Jacaré+CamlJava	*OCaml-Java*	*O'Jacaré+OCaml-Java*
appels transparents	via module Java	appels transparents
2 runtime	1 runtime	1 runtime
Pas d'allocation		
vérifications accès dans la capsule (avant accès Java)	vérifications accès par le compilateur (côté Java)	vérifications accès par le compilateur (côté Java)

Adaptation d'Ojacaré

TYPE IDL	type Java (java_type)	type OCaml pour OCaml-Java (oj_type t)	type OCaml (ml_type t)
void	void	unit	unit
boolean	boolean	bool	bool
byte	byte	int	int
char	char	int	char
double	double	float	float
float	float	float	float
int	int	int32	int
long	long	int64	int
short	short	int	int
string	java.lang.String	java'lang'String java_instance	string
pack/Obj	pack.Obj	pack'Obj java_instance	jObj

Figure: Les types dans OCamlJava

Comparaison de génération

```
let _init_point =
  let clazz = Jni.find_class "mypack/Point" in
  let id =
    try Jni.get_methodID clazz "<init>" "(II)V"
    with
    | _ ->
        failwith
          "Unknown constructor from IDL in class \"mypack.Point\" : \"Point
              (int,int)\"."
  in
    fun (java_obj : _jni_jPoint) _p0 _p1 ->
      let _p1 = _p1 in
      let _p0 = _p0
      in
        Jni.call_nonvirtual_void_method java_obj clazz id
          [| Jni.Camlint _p0; Jni.Camlint _p1 |];;
class point _p0 _p1 =
  let java_obj = _alloc_jPoint ()
  in let _ = _init_point java_obj _p0 _p1
    in object (self) inherit _capsule_jPoint java_obj end;;
```

```
class point _p0 _p1 =
  let _p1 = Int32.of_int _p1
  in let _p0 = Int32.of_int _p0
    in let java_obj = Java.make "mypack.Point(int,int)" _p0 _p1
      in object (self) inherit _capsule_jPoint java_obj end;;
```

Figure: La génération du constructeur de Point

Conclusion

Ce nouvel outil a apporté des nouvelles possibilités d'un côté, avec une simplicité d'utilisation :

- Accès simple à l'API Java
- Accès utilisateur transparent grâce aux classes encapsulantes
- 1 seul runtime -> Gestion mémoire simplifiée et sûre
- Code généré simplifié ($\sim$ 5 fois moins)
-

caractéristiques	*O'Jacaré + OCaml-Java*
accès champs	selon la visibilité + via appels de méthode
var./méth. statiques	fonctions/décl. globales
héritage≡sous-typage ?	×
surcharge	×
héritage multiple	✓