L'interopérabilité entre OCaml et Java

Béatrice Carre
beatrice.carre@etu.upmc.fr

encadrants : Emmanuel Chailloux, Xavier Clerc et Grégoire Henry

25 avril 2014

Table des matières

Introduction

Il arrive d'avoir envie de réutiliser des structures écrites dans un certain langage sans avoir à les réécrire complètement. Il peut arriver de vouloir d'utiliser l'efficacité et l'élégance du style fonctionnel d'Ocaml pour les calculs d'un programme mais aussi la portabilité du style objet de Java et la diversité de son API.

C'est pourquoi l'interopérabilité est un problème intéressant. L'interopérabilité engenre beaucoup de questions sur la gestion de plusieurs éléments : le cohérence des types d'un langage à l'autre, la copie ou partage des valeurs d'un monde à l'autre, le passage des exceptions, la gestion automatique de la mémoire (GC), et celle des caractéristiques de programmation pas forcément gérées dans les deux langages.

Deux études ont déjà été réalisées pour l'interopérabilité entre Ocaml et Java à travers leur modèle object respectif :

O'Jacaré conserve les runtimes des deux langages (GC, Exceptions, ...) et les fait communiquer avec l'aide de CamlJava, TODO.

OCaml-java 2.0 utilise un seul runtime, celui de Java, en compilant les programmes OCaml en byte-code Java. TODO.

L'idée est de profiter des deux approches, l'accès direct à toute l'API Java grâce à OCaml-Java, et l'accès à d'autres classes définies par le programmeur intéressé, en générant le code nécessaire à cet accès grâce à O'Jacaré, et le tout en gardant qu'un seul runtime, la JVM.

Après l'étude des deux outils, le projet consiste à engendrer pour ocaml-java les fichiers d'encapsulation d'O'Jacaré. Ce portage est réalisé en OCaml étant donné qu'il reprend ce qui a déjà été développé pour O'Jacaré.

Dans ce rapport, nous décrivons le schéma global avec ses avantages du générateur de code d'O'Jacaré, et celui du compilateur d'Ocaml-Java, pour détailler ensuite les modifications apportées à la génération d'interfaces, adaptée pour une encapsulation utilisable par le compilateur d'OCaml-Java. Pour finir, un exemple d'application accompagnée d'un test de performance sera présenté.

1 L'interopérabilité entre OCaml et Java

1.1 O'Jacaré, un générateur de code d'interface

O'Jacaré génère le code nécessaire à l'encapsulation des classes définies dans un IDL, pour permettre aux interface avec C de chacunes de comminuquer.

1.1.1 Principe global

O'Jacaré génère des fichier permettant à CamlJava de faire communiquer les deux mondes. CamlJava est une interface bas-niveau basée sur les interfaces de chaque langage avec C : la JNI (Java Native Interface) et external.

O'Jacaré permet à OCaml (resp. Java) d'accéder aux classes Java (resp. OCaml) grâce aux capsules (resp. stub) générées, qui contrôlent la communication à travers l'interface Camljava.

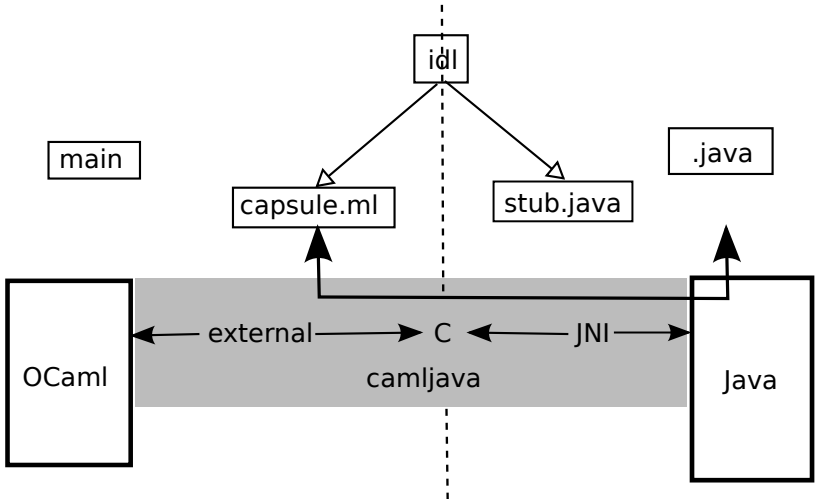

FIGURE 1 – Schéma global de la communication par Camljava

La génération de code se fait en plusieurs passes :
- analyse lexicale et analyse syntaxique de l' idl donnant un AST.
- vérification des types de l'AST, donnant un nouvel arbre nommé CAST (Checked AST).
- la génération des fichiers stub java nécessaires pour un appel callback
- la génération à partir du CAST des classes encapsulantes dans un fichier .ml
- la génération à partir du CAST du module .mli adapté

Ces différentes étapes seront présentées plus en profondeur.

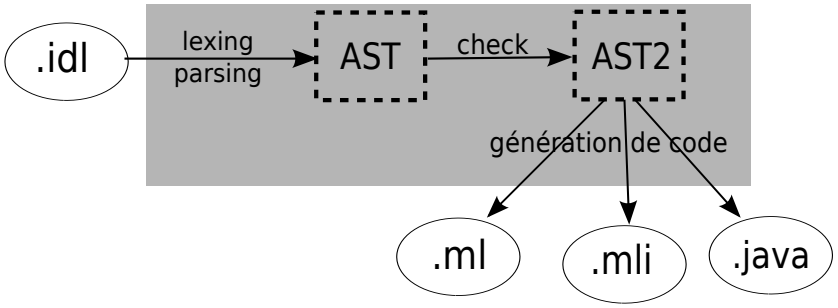

FIGURE 2 – Schéma global de la génération d'O'Jacaré

1.1.2 La syntaxe de l'idl

L'IDL est défini pour la construction des interfaces entreO'Caml et Java, il est donc défini en s'approchant de l'intersection des deux mondes objet (TODO : A completer : definir cette intersection)

La syntaxe du langage d'interface est donné en annexe, en utilisant la notation BNF.

Les symboles < et > encadrent des règles optionnelles, les terminaux sont en bleu, et les non-terminaux sont en italique.

1.1.3 Analyse lexical, syntaxique et sémantique

La première phase est celle d'analyse lexicale et syntaxique, séparant l'idl en lexèmes et construisant l'AST, défini en annexe par Idl.file, dont la structure est définie en annexe.

Vient ensuite la phase d'analyse sémantique, analysant l'AST obtenue par la phase précédente, vérifiant si le programme est correct, et

construisant une liste de CIdl.clazz, restructurant chaque classe ou interface définie dans l'idl. Le module Cidl définit le nouvel AST nommé CAST allant être manipulé dans les passes de génération de code. Il est décrit en annexe.

1.1.4 génération stub_file

pour callback : génération java

1.1.5 génération des classes encapsulantes

Type jni
Class type
Cast JNI (up et down)
Fonction d'allocation
Capsule / souche
Downcast utilisateur (_downcast, _instance_of)
Tableaux
Fonction d'initialisation Classe de construction
fonctions / methodes static

1.1.6 compilation par camljava

interfacage C, 2 runtimes, destiné à Jni,

1.2 OCaml-Java et la compilation de code OCaml vers du bytecode Java

intro TODO presentation

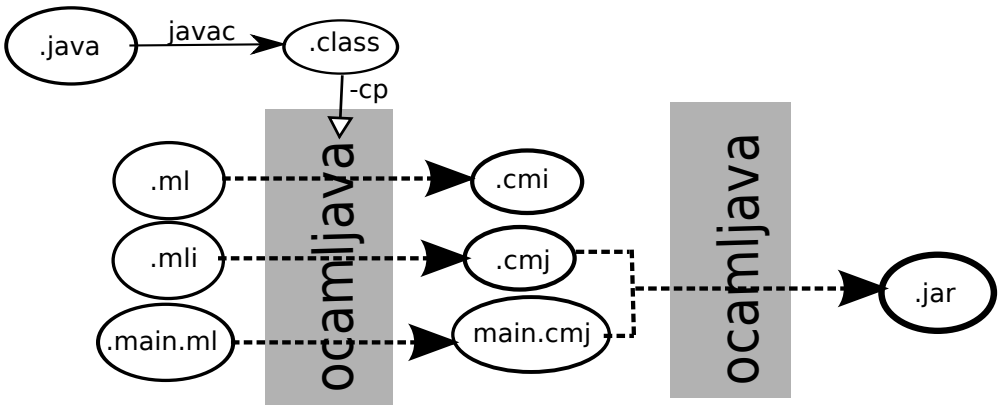

FIGURE 3 – Schéma global du compilateur d'OCaml-Java

1.2.1 Accès à du code Java

Description des types manipulés par OCaml-Java permettant un accès au monde de Java depuis celui d'OCaml.

type Java	description et exemple
java_constructor	signature d'un constructeur "java.lang.Object()"
java_method	signature d'une méthode "java.lang.String.lastIndexOf(string) :int"
java_field_get	signature d'un attribut "mypack.Point.x :int"
java_field_set	signature d'un attribut "mypack.Point.x :int"
java_type	classe, interface ou type Array "java.lang.String"
java_proxy	type d'une interface "java.lang.Comparable"

Description du module Java

```
1 make : 'a java_constructor -> 'a (* cree une instance d'objet *)
2 call : 'a java_method -> 'a (* appelle d'une methode *)
3 get : 'a java_field_get -> 'a (* acces a un champ, statique ou non *)
4 set : 'a java_field_set -> 'a (* modification du champ non statique
5 d'une instance d'objet *)
6 is_null : 'a java_instance -> bool (* teste si une instance est egale a
      null*)
7 instanceof : 'a java_type -> 'b java_instance -> bool
```

```
8 cast : 'a java_type -> 'b java_instance -> 'a
9 proxy : 'a java_proxy -> 'a
```

Une exception est définie dans le module pour permettre d'attraper les exceptions du côté OCaml :

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

1.3 profiter des deux approches

O'Jacaré construit les classes encapsulantes de classes java définies par l'utilisateur, et permet ainsi l'accès aux méthodes (d'instance ou de classe) Java en OCaml en passant par l'interface de bas niveau CamlJava. Cette interface de bas-niveau

OCaml-Java permet l'accès à toute l'API Java depuis OCaml TODO

Solution problèmes : TODO - gestion de la surchage (absente en OCaml) -> renommage obligatoire dans IDL. - 2 runtimes vs 1 seul (-> résoud pd de communication GC et exceptions) - gestion du typage (statype en OCaml vs dynamique en Java) -

2 Portage d'O'Jacaré pour OCaml-Java

2.1

2.2 schémas de compilation actuels d'O'Jacaré

Nous considérons un environnement contenant les variables suivantes, initialisées à leur valuer par défaut :

ρ = "" : le nom du **package** où trouver les classes définies.
Λ = "" : le **nom de la classe** courant.
γ = false : si la déclaration est une **interface**.
θ = false : si l'élément porte l'attribut **callback**.
α = false : si l'élément est déclaré **abstract**.
δ = "JniHierarchy.top" : la classe dont **extends** la classe courante.
Δ = [] : les interfaces qu'**implements** la classe courante.

package

$[\![package\ qname\ ;\ decl^*]\!] \longrightarrow$
$[\![decl^*]\!]_{\rho=qname}$

decl interface

$[\![interface\ name]\!] \longrightarrow$
$[\![class\ name]\!]_{\rho,\Lambda=name,\gamma=true}$

decl class

$[\![[callabck]class\ name]\!]_{\rho,\Lambda,\gamma,\theta,\alpha} \longrightarrow$
$[\![class\ name]\!]_{\rho,\Lambda=name,\gamma,\theta=true,\alpha}$

decl class

$[\![class\ CLASS\ extends\ E\ implements\ I1, I2...\{$
$attr1; attr2; ...;$
$m1; m2; ...;$
$init1; init2; ...;$
$\}]\!]_{\rho=PACK,\ \Lambda,\ \gamma,\ \theta=false,\ \alpha=false,\ \delta,\ \Delta} \longrightarrow$

$\Lambda = CLASS$
$\alpha = false$
$\delta = E$
$\Delta = [I1, I2, I3]$

```
let clazz = Jni.find_class PACK/CLASS

(** type jni.obj t *)
"type _jni_jCLASS = Jni.obj"

(** classe encapsulante *)
"class type jCLASS = object
   inherit E
   inherits jI1
   inherits jI2 ...
   method _get_jni_jCLASS : _jni_jCLASS
   end"

(** upcast jni *)
"let __jni_obj_of_jni_jCLASS (java_obj : _jni_jCLASS) =
   (Obj.magic : _jni_jCLASS -> Jni.obj) java_obj"
(** downcast jni *)
"let __jni_jCLASS_of_jni_obj =
   fun (java_obj : Jni.obj) ->
     Jni.is_instance_of java_obj clazz"

(* allocation *)
if not γ then
"let _alloc_jCLASS =
     fun () -> (Jni.alloc_object clazz : _jni_jCLASS)"

(* capsule wrapper *)
"class _capsule_jCLASS = fun (jni_ref : _jni_jCLASS) ->
    object (self)
      method _get_jni_jCLASS = jni_ref
      method _get_jni_jE = jni_ref
      method _get_jni_jI1 = jni_ref
      method _get_jni_jI2 = jni_ref
      inherit JniHierarchy.top jni_ref
    end"

(* downcast utilisateur *)
"let jCLASS_of_top (o : TOP) : jCLASS =
    new _capsule_jCLASS (__jni_jCLASS_of_jni_obj o#_get_jniobj)"
(* instance_of *)
```

```
"let _instance_of_jCLASS =
    in fun (o : TOP) -> Jni.is_instance_of o#_get_jniobj clazz"

(* tableaux *)
"let _new_jArray_jCLASS size =
    let java_obj = Jni.new_object_array size (Jni.find_class \"PACK/CLASS
        \")
    in
      new JniArray._Array Jni.get_object_array_element Jni.
        set_object_array_element (fun jniobj -> new _capsule_jCLASS jniobj)
        (fun obj -> obj#_get_jni_jCLASS) java_obj"
"let jArray_init_jCLASS size f =
    let a = _new_jArray_jCLASS size
    in (for i = 0 to pred size do a#set i (f i) done; a)"

(* inits *)
```

$[\![[name\ init1]< init >(arg*);...\}]\!]_{\rho,\ \Lambda,\ \gamma,\ \theta,\ \alpha,\ \delta,\ \Delta}$
$[\![[name\ init2]< init >(arg*);...\}]\!]_{\rho,\ \Lambda,\ \gamma,\ \theta,\ \alpha,\ \delta,\ \Delta}$
...

```
(* fonctions et  methodes statiques*)

    (*TODO*)
```

Ce tableau représente le résultat des fonctions str, jni_type, getJni, cast sur les types lors des générations des constructeurs ou des méthodes.

TYPE	str	jni_type	getJni	cast
void	V			
boolean	Z	Jni.Boolean _pi	_pi	_pi
byte	B	Jni.Byte _pi	_pi	_pi
char	C	Jni.Char _pi	_pi	_pi
short	S	Jni.Short _pi	_pi	_pi
int	I	Jni.Camlint _pi	_pi	_pi
long	J	Jni.Long _pi	_pi	_pi
float	F	Jni.Float _pi	_pi	_pi
double	D	Jni.Double _pi	_pi	_pi
string	LJava/lang/String;	Jni.Obj _pi	Jni.string_to_java _pi	_pi
pack/Obj	Lpack/Obj;	Jni.Obj _pi	_pi#_get_jni_jname	(_pi : jObj)

inits

$[\![[name\ INIT]< init >(A0, A1, ...)]\!] \longrightarrow$

```
"let _init_INIT =
  let id = Jni.get_methodID clazz \"<init>\"
             \"("(toStr A0)(toStr A1)...")V\"
  in
    fun (java_obj : _jni_jCLASS) "(cast A0) (cast A1) ..." -> ...
      let _p1 = "(getJni A1)" in
```

```
      let _p0 = "(getJni A0)" in
      Jni.call_nonvirtual_void_method java_obj clazz id
          [| "(jni\_type A0)"; "(jni\_type A1)"; ... |]

class INIT _p0 _p1 ... =
  let java_obj = _alloc_jCLASS ()
  in let _ = _init_INIT java_obj _p0 _p1 ...
    in object (self) inherit _capsule_jCLASS java_obj
end"
```

attributs

$[\![TYPE\ ATTR;]\!] \longrightarrow$

```
...
(* type class *)
"class type jCLASS =
  ...
   method set_ATTR : (j)TYPE -> unit
   method get_ATTR : unit -> (j)TYPE
   ... "
(* capsule *)
"class _capsule_jCLASS =
   let __fid_ATTR = try Jni.get_fieldID clazz \"ATTR\" "(toStr TYPE)" in
   ...
   fun (jni_ref : _jni_jCLASS) ->
     object (self)
        method set_ATTR =
           fun "(castArg TYPE)" ->
              let _p = "(getJni TYPE)"
              in Jni.set_object_field jni_ref __fid_ATTR _p
        method get_ATTR =
        fun () ->
           (new _capsule_jCLASS (Jni.get_object_field jni_ref __fid_ATTR) :
           jCLASS)
        ...
   "
```

methodes

TYPE	str	jni_type	getJni	cast
void	" "||V			
boolean	Z	Jni.Boolean _pi	_pi	_pi
byte	B	Jni.Byte _pi	_pi	_pi
char	C	Jni.Char _pi	_pi	_pi
short	S	Jni.Short _pi	_pi	_pi
int	I	Jni.Camlint _pi	_pi	_pi
long	J	Jni.Long _pi	_pi	_pi
float	F	Jni.Float _pi	_pi	_pi
double	D	Jni.Double _pi	_pi	_pi
string	LJava/lang/String;	Jni.Obj _pi	Jni.string_to_java _pi	_pi
pack/Obj	Lpack/Obj;	Jni.Obj _pi	_pi#_get_jni_jname	(_pi : jObj)

$[\![TYPEMETH(ARG1, ARG2, ...)]\!] \longrightarrow$

```
...
(* type class *)
"class type jCLASS =
    ...
    method METH : ARG1 -> ARG2 -> ... -> TYPE
    ... "
(* capsule *)
"class _capsule_jCLASS =
    let __mid_METH = Jni.get_methodID clazz "mETH"
          \"("(toStr ARG1)(toStr ARG2)...")"(toStr TYPE)"\"
    ...
    in
    object (self)
"(*TODO*)"       (*method METHObj1Obj2 =
          fun (_p0 : jObj1) ->
            let _p0 = _p0#_get_jni_jObj1
            ...
              in
              (new _capsule_jObj2
                (Jni.call_"Object"_method jni_ref __mid_mETHObj1Obj2
                [| Jni.Obj _p0 |]) : jObj2)
       *)
       method METH =
          fun "(cast A0) (cast A1) ..." ->
            let _p2 = "(getJni ARG2)" in
            let _p1 = "(getJni ARG1)" in
            let _p0 = "(getJni ARG0)"
            in
              Jni.call_"(aJniType TYPE)"_method jni_ref __mid_METH
                [| "(jni\_type ARG0)"; "(jni\_type ARG1)"; ... |]
```

//TODO : retour Obj dans methode array callback

2.3 Génération de code pour Ocaml-Java

Le type top manipulé sera le type d'instance objet de Ocaml-Java :

```
1 type top = java'lang'Object java_instance;;
```

Exception :

```
1 exception Null_object of string
```

class

Le schéma de compilation de base pour une classe est largement allégé. En effet,la fonction downcast jni est inutile, puisqu'on a la fonction Java.cast, effectuant tout le travail.

De même, l'upcast ->

TODO : voir http ://www.pps.univ-paris-diderot.fr/ henry/ojacare/doc/ojacare006.html. (** cast JNI, exporté pour préparer la fonction 'import' *)

L'allocation n'est pas non plus nécessaire, OCaml-Java gérant tout ça côté Java.

La capsule est aussi très simplifiée, les tests d'existance des méthodes classes etc est aussi géré par COaml-Java.

$[\![class\ CLASS\ extends\ E\ implements\ I1, I2...\{$
$attr1; attr2; ...;$
$m1; m2; ...;$
$init1; init2; ...;$
$\}]\!]_{\rho,CB} \longrightarrow$

```
(** type 'a java_instance*)
"type _jni_jCLASS = PACK'CLASS java_instance;;"

(** classe encapsulante *)
"class type jCLASS =
   object inherit E
   inherits jI1
   inherits jI2 ...
   method _get_jni_jCLASS : _jni_jCLASS
   end"

(* capsule wrapper *)
"class _capsule_jCLASS =
  fun (jni_ref : _jni_jCLASS) ->
     let _ =
        if Java.is_null jni_ref
        then raise (Null_object "mypack/Point")
        else ()
     in

     object (self)
      (* method _get_jni_jCLASS = jni_ref
       method _get_jni_jE = jni_ref
       method _get_jni_jI1 = jni_ref
       method _get_jni_jI2 = jni_ref*)
       inherit JniHierarchy.top jni_ref
     end"

(* downcast utilisateur *)
"let jCLASS_of_top (o : TOP) : jCLASS =
    new _capsule_jCLASS (__jni_jCLASS_of_jni_obj o#_get_jniobj)"
(* instance_of *)
"let _instance_of_jCLASS =
    in fun (o : TOP) -> Jni.is_instance_of o#_get_jniobj clazz"
```

methodes

Tableau représentant les équivalents en OCaml des types Java manipulés par OCaml-Java. La troisième colonne représente les types manipulés par les programmes OCaml écrit par l'utilisateur du nouvel outil. Le problème est donc de convertir du deuxième

au troisième type pour la manipulation côté OCaml et du troisième au second lors d'un appel à une fonction du module Java (un appel, un constructeur ou autre).

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

Tableau associant pour chaque types de l'IDL les fonctions utiles aux schémas de compilation manipulant ceux-ci, comme explicité ci-dessus.

TYPE IDL	to_oj_Type ARGi	to_ml_type ARGi	fcast
void			
boolean			_pi
byte			_pi
char	TODO	TODO	_pi
short			
int	Int32.of_int	Int32.to_int	_pi
long	Int64.of_int	Int64.to_int	_pi
float			_pi
double			_pi
string	JavaString.of_string	JavaString.to_string	_pi
pack/Obj	_pi#_get_jni_jObj	(new _capsule_jObj ... : jObj)	(_pi : jObj)

$[\![\ RTYPE\ \ METH\ (TARG1,\ TARG2, ...)]\!]_{TODO} \longrightarrow$

```
...
(* type class *)
"class type jCLASS =
   ...
   method METH : "(ml_type TARG1) -> (ml_type TARG2)" -> ... ->(ml_type
      RTYPE)
   ... "
(* capsule *)
"class _capsule_jCLASS =
   object (self)
      method METH =
         fun "(fcast TARG0) (fcast TARG1) ..." ->
           let _p1 = "(to_oj_type TARG1)" _p1 in
           let _p0 = "(to_oj_type TARG0)" _p0
```

```
            in"
              (to_ml_type RTYPE)
              "Java.call \"PACK.CLASS.METH("(javaType TARG1),(javaType TARG2
                  ),...):(javaType RTYPE)"\" jni_ref _p0 _p1 ..."
```

inits

$[\![[name\ INIT] < init > (TARG0,\ TARG1, ...)]\!] \longrightarrow$

```
"class INIT _p0 _p1 ... =
  let _p1 = "(to_oj_type TARG1)"  in
  let _p0 = "(to_oj_type TARG2)" in
  let java_obj = Java.make \"PACK.CLASS("(javaType
          TARG0),(javaType TARG1),...")\" _p0 _p1
  in
  object (self)
     inherit _capsule_jCLASS java_obj
  end;;"
```

attributs

$[\![TYPE\ \ ATTR;]\!] \longrightarrow$

```
...
(* type class *)
"class type jCLASS =
  ...
   method set_ATTR : "(ml_type TYPE)" -> unit
   method get_ATTR : unit -> "(ml_type TYPE)"
   ... "
(* capsule *)
"class _capsule_jCLASS =
   ...
   fun (jni_ref : _jni_jCLASS) ->
     object (self)
     ...
        method set_ATTR =
           fun "(fcast TYPE)" ->
              let _p = "(to_oj_type TYPE)" _p
              in Java.set \"PACK.CLASS.ATTR:TYPE\" jni_ref _p
        method get_ATTR =
        fun () ->
           "(to_ml_type TYPE)" (Java.get \"PACK.CLASS.ATTR:TYPE\" jni_ref)
        ...
   "
```

3 Application et performance

Conclusion

Bibliographie, références

[1] CHAILLOUX E., MANOURY P., PAGANO B., *Développement d'applications avec Objective Caml*, O'Reilly , 2000, (`http://www.oreilly.fr/catalogue/ocaml.html`)

[2] CHAILLOUX E., HENRY G., *O'Jacaré, une interface objet entre Objective Caml et Java*, 2004,

[3] CLERC X., *OCaml-Java : Typing Java Accesses from OCaml Programs*, Trends in Functional Programming, Lecture Notes in Computer Science Volume 7829, 2013, lien

[4] CLERC X., *OCaml-Java : OCaml on the JVM*, Trends in Functional Programming, 2012,lien

[5] CLERC X., *OCaml-Java :OCaml-Java : from OCaml sources to Java bytecodes* , Trends in Functional Programming, 2012,lien

[6] Leroy X., *The camljava project*, (`http://forge.ocamlcore.org/projects/camljava/`)

[7] CLERC X.,*OCaml-java : module Java* `http://ocamljava.x9c.fr/preview/javalib/index.html`

[8] CamlP4 (* todo *)

Annexe

BNF

```
class

file ::= package <package>*
       | decl <decl>*

package ::= package qname ; decl <decl>*

decl ::= class
       |interface

class ::= <[attributes]> <abstract> class name
          < extends qname >
          < implements qname <, qname>* >
          { <class_elt ;>* }
class_elt ::= <[ attributes ]> <static> <final> type name
            | <[ attributes ]> <static> <abstract> type name (<args>)
            | [ attributes ] <init> (<args>)

interface ::= <[ attributes ]> interface name
              < extends qname <, qname>* >
             { <interface_elt;>* }
interface_elt ::=
     <[ attributes ]> type name
   | <[ attributes ]> type name (<args>)

args ::= arg <, arg>*
arg ::= <[ attributes ]> type <name>

attributes ::=  attribute <, attribute>*
attribute ::= name ident
            | callback
            | array

type ::= basetype
       | object
       | basetype [ ]
basetype ::= void
           | boolean
           | byte
           | char
           | short
           | int
           | long
           | float
           | double
           | string
object := qname
qname ::= name<.name>*
name ::= ident
```

Module CIdl, structure manipulée par O'Jacaré à partir de l'IDL

```
(**  module CIdl  *)
type typ =
  | Cvoid
  | Cboolean (** boolean -> bool *)
  | Cchar (** char -> char *)
  | Cbyte (** byte -> int *)
  | Cshort (** short -> int *)
  | Ccamlint (** int -> int<31> *)
  | Cint (** int -> int32 *)
  | Clong (** long -> int64 *)
  | Cfloat (** float -> float *)
  | Cdouble (** double -> float *)
  | Ccallback of Ident.clazz
  | Cobject of object_type (** object -> ... *)
and object_type =
  | Cname of Ident.clazz (** ... -> object *)
  | Cstring (** ... -> string *)
  | Cjavaarray of typ (** ... -> t jArray *)
  | Carray of typ (** ... -> t array *)
  | Ctop

type clazz = {
    cc_abstract: bool;
    cc_callback: bool;
    cc_ident: Ident.clazz;
    cc_extend: clazz option; (* None = top *)
    cc_implements: clazz list;
    cc_all_inherited: clazz list; (* tout jusque top ... (et avec les
        interfaces) sauf elle-meme. *)
    cc_inits: init list;
    cc_methods: mmethod list; (* methodes + champs *)
    cc_public_methods: mmethod list; (* methodes declarees + celles
        heritees *)
    cc_static_methods: mmethod list;
  }
and mmethod_desc =
  | Cmethod of bool * typ * typ list (* abstract, rtype, args *)
  | Cget of typ
  | Cset of typ
and mmethod = {
    cm_class: Ident.clazz;
    cm_ident: Ident.mmethod;
    cm_desc: mmethod_desc;
  }
and init = {
    cmi_ident: Ident.mmethod;
    cmi_class: Ident.clazz;
    cmi_args: typ list;
  }
type file = clazz list
```

module Ident

```
(* module Ident  *)
(* le type des identifiants de classe de l'IDL *)
type clazz = {
    ic_id: int;
    ic_interface: bool;
    ic_java_package: string list;
    ic_java_name: string;
    ic_ml_name: string;
    ic_ml_name_location: Loc.t;
    ic_ml_name_kind: ml_kind;
  }
type mmethod = {
    im_java_name: string;
    im_ml_id: int; (** entier unique pour une nom ml *)
    im_ml_name: string;
    im_ml_name_location:Loc.t;
    im_ml_name_kind: ml_kind;
  }
```

phases de

Type jni
    MlClass.make_jni_type
Class type
    MlClass.make_class_type
Cast JNI
    MlClass.make_jniupcast
    MlClass.make_jnidowncast
Fonction d'allocation
    MlClass.make_alloc
    MlClass.make_alloc_stub
Capsule / souche
    MlClass.make_wrapper
Downcast utilisateur
    MlClass.make_downcast
    MlClass.make_instance_of
Tableaux
    MlClass.make_array
Fonction d'initialisation
    MlClass.make_fun
Classe de construction
    MlClass.make_class
fonctions / methodes static
    MlClass.make_static