`O'Jacaré`, une interface objet entre Objective Caml et Java

Emmanuel Chailloux — Grégoire Henry

Equipe Preuves, Programmes et Systèmes (UMR 7126),
Université Pierre et Marie Curie (Paris VI),
175, rue du Chevaleret, F-75013 Paris
{Emmanuel.Chailloux,Gregoire.Henry}@pps.jussieu.fr

RÉSUMÉ. On présente dans cet article un nouveau générateur de code, appelé `O'Jacaré`*, pour faciliter l'interopérabilité entre Java et Objective Caml à travers leur modèle objet respectif.* `O'Jacaré` *définit un* IDL (Interface Definition Language) *simple permettant la description des classes et des interfaces afin de communiquer d'Objective Caml vers Java. Pour les communications de Java vers Objective Caml on ajoute un mécanisme de rappel. L'implantation repose sur les interfaces de bas niveau avec C de chaque langage (*Java Native Interface *pour Java et* `external` *pour Objective Caml) et utilise une version étendue de la bibliothèque* `camljava`*.* `O'Jacaré` *engendre toutes les classes encapsulantes nécessaires et vérifie le typage statique des deux mondes. Bien que l'*IDL *soit une intersection de ces deux modèles objet,* `O'Jacaré` *permet de combiner les caractéristiques des deux.*

ABSTRACT. We present in this paper a new code generator, called `O'Jacaré`*, to interoperate between Java and Objective Caml through their object model.* `O'Jacaré` *defines a basic* IDL (Interface Definition Language) *for classes and interfaces description to communicate from Objective Caml to Java. For communications from Java to Objective Caml we add a callback mechanism. The implementation is based on each language low-level interfaces (*Java Native Interface *for Java and* `external` *for Objective Caml) and uses an extended version of the* `camljava` *library.* `O'Jacaré` *generates all needed wrapper classes and enables static type-checking in both worlds. Although the* IDL *is an intersection from these two object models,* `O'Jacaré` *allows to combine all features from both.*

MOTS-CLÉS : interopérabilité, IDL*, Objective Caml, Java.*

KEYWORDS: interoperability, IDL*, Objective Caml, Java.*

RSTI - L'objet – 10/2004. LMO'04, pages 75 à 88

1. Introduction

L'assemblage de bibliothèques et/ou de composants pour la construction d'applications multi-langages est devenu une nécessité comme le rappelle l'introduction de [FIN 99] : « *Programming languages that do not support a foreign-language interface die a slow, lingering death - good languages die more slowly than bad ones, but they all die in the end.* »

L'interfaçage entre langages pose les difficultés d'implantation suivantes : protocoles d'appel compatibles des fonctions et des méthodes ; cohérence des types d'un langage à l'autre ; copie ou partage des valeurs d'un monde à l'autre, en particulier la convention de passage de paramètres (`in`, `out`, `inout`) ; ruptures de calcul (exceptions) transmises d'un monde à l'autre ; gestion automatique de la mémoire (*GC*). De plus certains traits de programmation d'un langage ne se retrouvent pas forcément dans un autre langage et sont difficilement traduisibles. Cela apparaît tant au typage (selon les classes de polymorphisme) que pour les valeurs manipulées (comme les valeurs fonctionnelles) et le contrôle de l'exécution (exceptions, continuations).

Les interfaces entre langages varient selon le rôle donné à chacun des langages suivant un des trois schémas suivants. Le premier schéma réduit la bibliothèque d'appels extérieur au minimum ; l'encapsulation et le décodage des valeurs est effectuée par du code supplémentaire écrit à la main. Le second intègre dans un des langages les interfaces des modules/paquetages de l'autre langage ; le code d'interfaçage peut alors être engendré automatiquement. Le dernier schéma utilise un langage extérieur de définition d'interfaces (*Interface Definition Language* ou *IDL*). La compilation de ces définitions produit le code des interfaces.

Dans cet article nous décrivons l'ensemble des outils, appelé `O'Jacaré`[1], pour l'interfaçage de Java et d'O'Caml (surnom pour Objective Caml). `O'Jacaré` définit un *IDL* simple et construit un générateur de code d'interfaces. O'Caml est un langage fonctionnel/impératif statiquement typé muni d'une extension objet. Cette couche objet permet de voir du côté O'Caml les bibliothèques Java comme une nouvelle hiérarchie de classes. Pour que Java puisse manipuler les instances de classe O'Caml on introduit un mécanisme de rappel (*callback*) autorisant l'appel de méthode O'Caml en Java. Comme les deux modèles objets diffèrent fortement, notre *IDL* n'intégrera pas toutes les fonctionnalités des deux langages (surcharge, héritage multiple, classes paramétrées, type de `self`, ...), mais concernera l'intersection des deux modèles. On obtient néanmoins des programmes possédant la richesse de ces deux modèles.

On compare les principales caractéristiques des modèles objets des deux langages pour ensuite décrire notre *IDL* en l'illustrant par un premier exemple simple. On détaille ensuite l'implantation d'`O'Jacaré` pour apprécier deux exemples du modèle objet ainsi obtenu. On présente alors une mesure de performance sur l'application `ojDvi`, une visionneuse `dvi`. Au final on commente les travaux connexes à ce travail en précisant ses futurs prolongements. Enfin une courte conclusion clôt cet article.

1. `http://www.pps.jussieu.fr/~henry/ojacare`

2. Comparaison des modèles objets

Objective Caml [LER 02] comme son nom l'indique possède une extension objet qui apporte un langage de classes [REM 98] au noyau fonctionnel/impératif tout en s'intégrant au système de types. Une déclaration de classe construit un nouveau type « objet » et une fonction de construction d'instances de ce type. Un type « objet » correspond à un ensemble de méthodes associées à leurs types. Par exemple le type suivant peut être inféré pour les instances de classes possédant des méthodes `moveto` et `toString` :

$$< \text{moveto} : (\text{int} * \text{int}) \rightarrow \text{unit} ; \text{toString} : \text{unit} \rightarrow \text{string} >.$$

Les seules méthodes que l'on peut appeler sur un objet sont celles visibles dans son type. Le typage statique garantit alors que l'objet receveur possède bien une méthode du bon nom et du bon type. L'exemple suivant est correct si la classe `point` définit (ou hérite) une méthode `moveto` prenant une paire d'entiers comme argument. Les objets s'intègrent dans le modèle fonctionnel typé par les types objets ouverts (`<. .>`). La fonction `f` suivante acceptera comme argument n'importe quel objet dont le type contient une méthode `moveto` dont l'argument est une paire d'entiers.

appel de méthode	style fonctionnel-objet
let p = **new** point(1,1) ; ; p#moveto(10,2) ; ;	# **let** f o = o # moveto (10,20) ; ; **val** f : < moveto : int * int –> 'a ; .. > –> 'a

Les classes peuvent être en relation d'héritage et en relation de sous-typage. Ces deux relations sont bien distinctes [CHA 00]. Les déclarations de classes acceptent l'héritage multiple et les classes paramétrées. Par contre la surcharge de méthode n'est pas acceptée. Du point de vue de l'exécution la liaison est toujours tardive (*late binding*).

Le modèle du langage Java [GOS 00] est bien connu et ne sera pas décrit ici. Par contre on compare ses principales caractéristiques avec celles d'O'Caml dans le tableau suivant :

caractéristiques	Java	O'Caml	caractéristiques	Java	O'Caml
classes	√	√	sous-typage	√	√
accès champs	1	2	héritage ≡ sous-typage ?	oui	non
liaison tardive	√	√	surcharge	√	5
liaison précoce	√	3	héritage multiple	6	√
typage statique	√	√	classes paramétrées	7	√
typage dynamique	√	4	paquetages/modules	8	8
			classes mutuellement récursives	√	9

Un √ indique que le langage supporte pleinement un trait de programmation sinon une note numérotée ci-dessous en détaille les spécificités.

1) selon la nature des attributs de visibilité les variables de classes ou d'instances sont accessibles ;

2) uniquement *via* un appel de méthode ;

3) les méthodes statiques sont des fonctions globales d'un module et les variables de classes des déclaration globales ;

4) pas de *downcast* en O'Caml seulement dans l'extension coca-ml [CHA 02] ;

5) pas de surcharge en O'Caml mais le type de `self` peut apparaître dans le type d'une méthode qui pourra être redéfinie (*overriding*) dans une sous-classe ;

6) pas d'héritage multiple pour les classes Java, seulement pour les interfaces ;

7) pas de classes paramétrées actuellement en Java, seulement dans les extensions Pizza [ODE 97] et Generic Java [BRA 98] ; ce trait est en voie d'intégration et de test pour la version 1.5 de Java ;

8) les modules simples d'O'Caml correspondent aux parties publiques des paquetages Java ; la notion de modules paramétrés est inexistante en Java ;

9) avec une limitation car les modules ne sont pas mutuellement récursifs .

L'intersection des deux modèles objet correspond à un langage structuré en classes, dont l'appel de méthode est toujours en liaison tardive. Les relations d'héritage et de sous-typage sont confondues. Du point de vue des types, il n'y a pas de surcharge, de plus le type de l'instance ne peut pas apparaître dans le type d'une méthode. L'héritage est simple et il n'y a pas de classes paramétrées. C'est effectivement un modèle objet simple pour un langage. C'est une qualité pour une langage d'interface : cela facilitera la communication entre les classes Java et O'Caml.

3. Description de l'*IDL*

`O'Jacaré` définit un *IDL* pour la construction des interfaces entre O'Caml et Java. Ce langage s'inspire fortement de la syntaxe Java et bien qu'il privilégie le sens d'appel d'O'Caml vers Java, un mécanisme de rappel autorise la redéfinition d'une méthode Java par une méthode O'Caml.

3.1. *Principes*

Notre motivation principale étant la simplicité d'emploi, l'*IDL* défini n'accepte que les déclarations de classe, de classe abstraite et d'interface (voir figure 1) dans un espace de noms. Celles-ci correspondent respectivement à une classe, une classe abstraite et une interface Java, et à une classe et des classes abstraites O'Caml. L'héritage suit celui de Java : simple pour les classes mais multiple pour les interfaces. Du point de vue du programmeur un appel de O'Caml vers Java se fait donc par un appel de méthode, et le sens inverse par un appel à une méthode redéfinie en O'Caml. Les variables d'instance ou de classe Java sont accessibles par des méthodes engendrées automatiquement depuis O'Caml. Seules les valeurs des types de base sont donc passées (ou retournées) par copie, dans tous les autres cas (objet, tableau) les valeurs sont transmises par référence assurant ainsi le partage. On distingue dans l'*IDL* le type de base `string` d'un objet de classe `java.lang.String`. Les exceptions sont propagées

d'un monde à l'autre. Le code engendré à partir d'un fichier interface est sûr du point de vue du typage par une vérification des classes Java au chargement du programme.

3.2. *Syntaxe*

La syntaxe du langage d'interface est donnée à la figure 1. On utilise les symboles $\langle \ldots \rangle$ pour les règles optionnelles et un corps gras pour les mots-clés.

classes et interfaces	arguments, attributs, types
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* ⟨ **implements** *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** | *type_attribute* *type_attribute* ::= *basetype* | *object* | *type_attribute* **array** *type* ::= *basetype* | *object* | *basetype* [ ] *basetype* ::= **void** | **boolean** | **byte** | **char** | **short** | **int** | **long** | **float** | **double** | **string** *object* ::= *qname* *qname* ::= *name* ⟨. *name*⟩* *name* ::= ident

Figure 1. *Grammaire de l'*IDL

Les espaces de noms sont indiqués par le mot clé **package**. Les constructeurs sont nommés **<init>**, en référence à la signature des fonctions d'initialisation de bas niveau en Java. Les attributs de classe, de méthodes ou de variables d'instance concernent les alias de nom, les conversions explicites de types et la qualification pour le mécanisme de `callback`. Les types manipulés sont les classes, les tableaux, et les types de base auxquels a été ajouté `string`.

3.3. *Exemple d'utilisation*

Dans cet exemple on cherche à utiliser depuis O'Caml les classes Java `Point` et `PointColore` en sachant que la méthode `toString` de la classe `PointColore` retourne la concaténation d'un appel à la méthode `toString` de `super` et d'un appel à la méthode `getColor` sur `this`.

La compilation du fichier `p.idl`, figure 2, engendre d'une part un fichier `p.ml` contenant le code d'interfaçage et d'autre part un fichier `PointColoreStub.java` pour la classe `PointColore` qui possède l'attribut `callback`. On distingue les classes engendrées en présence de l'attribut `callback` pour préciser le mécanisme d'appel

```
package mypack;

class Point {
  int x; int y;

  [name default_point] <init> ();
  [name point] <init> (int,int);
  void moveto(int,int);
  void rmoveto(int,int);
  string toString();
  void display();
  double distance();
  boolean eq(Point);
}
```

```
[callback] class PointColore extends Point {
  [name default_point_colore] <init> ();
  [name point_colore] <init> (int,int,string);

  string getColor();
  void setColor(string);
  [name eq_pc] boolean eq(PointColore)
}

class Nuage {
  [name empty_nuage] <init> ();
  void addPoint(Point);
  string toString();
}
```

Figure 2. *Les inévitables classes* `Point` *et* `PointColore`

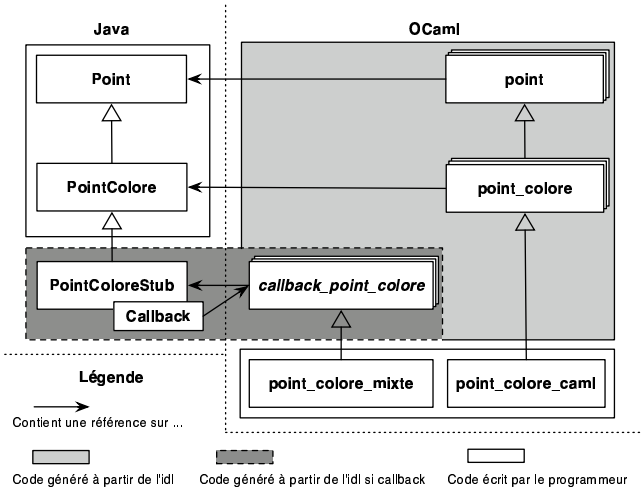

Figure 3. *Relations entre classes*

des méthodes Java (*virtual* ou non) dans le but de limiter la communication systématique entre les deux *runtimes* à chaque appel de méthode (voir figure 3).

Le programme `test_p.ml` de la figure 4 montre l'utilisation des classes `Point` et `PointColore` à partir d'O'Caml. L'opérateur de contrainte de type, noté `:>`, permet de considérer le type d'un objet comme un surtype au sens de la relation de sous-typage.

Les trois premiers affichages appellent la méthode `display` de l'objet Java. Bien que la classe `point_colore_caml` redéfinisse aussi la méthode `getColor`, l'appel de `display` n'affiche pas l'information supplémentaire fournie par cette nouvelle méthode dans la mesure où l'appel à `display` reste dans le monde Java.

programme `test_p.ml`

```
open P;;

let p = new point 1 1;;
p#display();;                                   (* 1 *)
let pc = new point_colore 1 3 "bleu";;
pc#display();;                                  (* 2 *)

class point_colore_caml x y c =
  object
    inherit point_colore x y c as super
    method getColor () =
      "[Caml"^(super#getColor ())^"]"
  end;;

let pcc = new point_colore_caml 1 3 "bleu";;
pcc#display();;                                 (* 3 *)
```

```
class point_colore_mixte x y c =
  object
    inherit callback_point_colore x y c
      as super
    method getColor () =
      "[Caml"^(super#getColor ())^"]"
  end;;

let pcm =
  new point_colore_mixte 1 3 "bleu";;
pcm#display();;                                 (* 4 *)

let l = [(pcc :> point); (pcm :> point)];;
List.iter (fun x -> x#display()) l;;            (* 5 *)
```

trace d'exécution

```
(1,1)
(1,3):bleu
(1,3):bleu
```

```
(1,3):[Camlbleu]
(1,3):bleu(1,3):[Camlbleu]
```

Figure 4. *Exécution d'un programme Java-O'Caml*

Les deux derniers affichages appellent bien la méthode `getColor` redéfinie dans la classe concrète `point_colore_mixte` construite en héritant de la classe `callback_point_colore`. Sans cette redéfinition, la méthode `getColor` héritée appelle directement la méthode `getColor` de la classe `PointColore` de Java.

La construction de la liste `l`, de type `point list`, montre le mélange de style fonctionnel et objet d'O'Caml y compris pour des objets Java encapsulés.

4. Implantation

Cette implantation utilise l'interface de bas niveau `camljava` [2] pour la communication entre Java et O'Caml. La compilation d'un fichier `.idl` construit les classes nécessaires pour une communication sûre à travers cette bibliothèque de base.

4.1. *La bibliothèque* `camljava`

L'interface `camljava` fait communiquer O'Caml et Java à travers les interfaces avec C de chaque langage : *Java Native Interface* (*JNI* [GOR 98]) pour Java et *external* [LER 02] pour O'Caml. Les méthodes Java (d'instance ou de classe) sont accessibles depuis O'Caml en deux temps : recherche d'une méthode par son nom et sa signature sur un objet instance de `Class`, puis appel (virtuel ou non) de cette méthode, sur un objet quelconque pour les méthodes d'instance, en lui passant un tableau d'ar-

2. `http://cristal.inria.fr/~xleroy/software.html`

guments. Les appels inverses suivent le même schéma en simplifiant la recherche des méthodes qui sont seulement identifiées par leur nom de par l'absence de surcharge en O'Caml. Cette séparation ne garantit pas, par exemple, le passage du bon nombre et du bon type des arguments. La bibliothèque `camljava` automatise le transfert des exceptions d'un monde à l'autre et assure le passage des arguments : par valeur pour les types de base, et par référence sur les objets.

Du point de vue mémoire tout objet Java alloué en O'Caml est une racine du *GC* de Java. Quand O'Caml ne l'utilise plus, la racine Java est supprimée mais l'objet peut être conservé par le *GC* de Java s'il est référencé par un autre objet encore vivant. Réciproquement les objets O'Caml passés à Java sont eux aussi considérés comme des racines du *GC* d'O'Caml le temps qu'ils résident en Java. On retrouve donc les problèmes inhérents aux compteurs de références pour les structures circulaires.

La bibliothèque `camljava` a été étendue pour assurer que les communications entre les deux mondes s'effectuent toujours dans le *thread* principal. On s'aperçoit que `camljava` offre des possibilités de programmation riches mais dangereuses d'où la nécessité d'engendrer automatiquement les codes d'encapsulation et d'intégration au système de types.

4.2. *Génération de code*

On cherche à faire correspondre une classe Java décrite dans un fichier *IDL* à une classe O'Caml. Comme toute instance de classe Java est considérée du type abstrait `Jni.obj` en O'Caml, nous allons construire autour de ces objets Java de véritables objets O'Caml. Pour faciliter la représentation de la hiérarchie de classes Java, nous introduisons une classe principale, appelée `top`, dont toute classe O'Caml engendrée héritera.

La figure 5 décrit les différents classes et types engendrés en O'Caml et Java par la compilation d'une déclaration dans l'*IDL* selon la présence ou non de l'attribut `callback`.

classe	interface
- 1 type objet t - 1 classe encapsulante W de type t - 1 à n classes (C_i), sous-classes de W (1 par constructeur) - 1 fonction `instanceof` pour ce type - 1 fonction de *cast* pour ce type	- 1 type objet t - 1 classe encapsulante W de type t - 1 fonction `instanceof` pour ce type - 1 fonction de *cast* pour ce type
avec `callback`	
- 1 classe souche (*stub*) - 1 à n classes abstraites (1 par constructeur) dont toutes les méthodes sont concrètes - 1 sous-classe Java	- 1 classe Java implantant l'interface - 1 classe abstraite dont toutes les méthodes sont abstraites

Figure 5. *Compilation des déclarations*

Dans l'exemple du `PointColore` défini à la figure 3 on engendrera donc un type objet `jPointColore` contenant toutes les méthodes décrites dans l'*IDL* et deux classes « utilisateur » `point_colore` et `default_point_colore` héritant d'une même classe encapsulante. La déclaration de cette classe vérifie au chargement l'existence de la classe Java et les signatures de ses méthodes. Ces trois classes ont le type `jPointColore`.

```
class point_colore _p0 _p1 _p2 =
  let java_obj = _alloc_jPointColore () in
  let _ = _init_point_colore java_obj _p0 _p1 _p2 in
  (object (self) inherit _wrapper_jPointColore java_obj end : jPointColore)
```

Figure 6. *Exemple de code engendré*

En présence de l'attribut `callback` une nouvelle classe dite « souche » est définie en O'Caml pour encapsuler l'objet Java dont la classe a redéfini les méthodes de son ancêtre comme des appels vers la souche O'Caml. L'utilisateur peut-alors définir ses propres comportements en héritant d'une classe « utilisateur » (héritière directe de la classe souche), en sachant que le comportement par défaut de ces classes « utilisateur » est de propager l'appel vers la classe ancêtre Java.

Pour la figure 7 qui illustre ce mécanisme, les noms des classes de définition des méthodes sont indiquées en abrégé (par exemple *pc* pour `PointColore`).

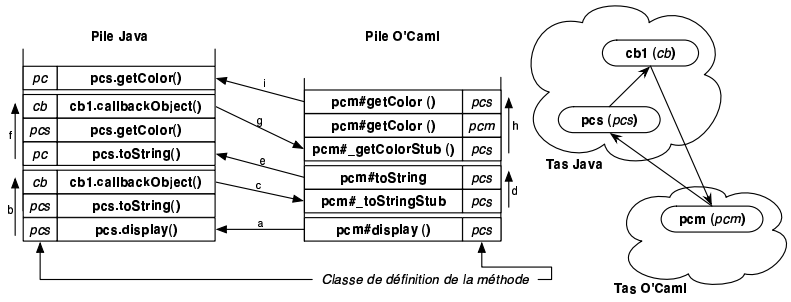

Figure 7. *Détail d'un appel à display sur un* `point_colore_mixte`

La création d'un objet de classe `point_colore_mixte` s'effectue en 4 étapes :
– allocation d'un objet Java `pcs` de classe `PointColoreStub`,
– allocation de l'objet O'Caml `pcm`
– initialisation de l'objet O'Caml `pcm` avec la référence sur `pcs`
– initialisation de l'objet Java `pcs` qui se détaille :
  - appel à l'initialisateur de `PointColore`
  - allocation d'un objet Java `cb1` de classe `Callback`
  - initialisation de l'objet `cb` avec une référence sur `pcm`

Certes la liaison tardive entre deux machines virtuelles provoquent quelques indirections supplémentaires. Pour cela les classes « utilisateur » correspondant à une souche `callback` sont déclarées abstraites et ne peuvent pas être utilisées par inadvertance.

4.3. *Génération automatique de fichiers* `.idl`

Comme le nombre de classes Java et de modules O'Caml est important il peut sembler fastidieux de devoir écrire manuellement les différents fichiers `idl` correspondants. Pour cela on engendre automatiquement les fichiers de classes Java par un programme Java qui explore dynamiquement les classes Java. En sens inverse l'interface d'un module simple O'Caml permet de construire son fichier `.idl`. Néanmoins pour éviter d'utiliser une convention de nommage pour la surcharge de méthodes nous préférons utiliser des fichiers `.idl` construits manuellement.

5. Extension des modèles objets

`O'Jacaré` nous autorise à manipuler en partie les deux modèles objets. On illustre ces nouvelles possibilités par un héritage multiple en O'Caml de classes Java et en effectuant une vérification dynamique de type (*downcast*) en O'Caml.

5.1. *Héritage multiple de classe Java*

L'exemple suivant est repris de [CHA 00]. On définit deux hiérarchies de classes en Java : les objets graphiques et les objets géométriques. Chaque hiérarchie possède une classe `Rectangle`. Le programme O'Caml suivant crée une classe héritant de ces deux classes Java.

fichier `q.idl`	programme O'Caml

package mypack ;
 	
class Point {
[**name** point] <**init**> (**int**, **int**) ;
}
class RectangleGr {
[**name** rect_graph] <**init**>(Point, Point) ;
string toString() ;
}
class RectangleGeo {
[**name** rect_geo] <**init**>(Point, Point) ;
double compute_area() ;
}
	open Q ; ;
	
	class rect_geo_graph p1 p2 =
	object
	inherit rect_geo p1 p2 **as** super_geo
	inherit rect_graph p1 p2 **as** super_graph
	end ; ;
	
	let p1 = **new** point 10 10 ; ;
	let p2 = **new** point 20 20 ; ;
	let rgg = **new** rect_geo_graph p1 p2 ; ;
	Printf.printf "area=%g\n" (rgg#compute_area ()) ; ;
	Printf.printf "toString=%s\n" (rgg#toString ()) ; ;

Figure 8. *Exemple d'héritage multiple*

5.2. Downcast *d'objet O'Caml*

O'Caml n'autorise aucune opération de typage dynamique, néanmoins dans une interface avec Java cela est nécessaire au moins pour les objets provenant d'un calcul du côté Java. Dans l'exemple de la figure 4 on construisait une liste `l` de `point` or ceux-ci correspondaient à des points colorés. On autorise alors l'utilisation des fonctions de contraintes de type de `top` vers le type O'Caml d'une classe Java. Ces fonctions déclenchent une exception si le type n'est pas compatible.

```
let l = [(pcc :> point); (pcm :> point)];;
let lc = Lisp.map (fun x -> jPointColore_of_top (x :> top)) l;;
```

On garde le côté explicite des contraintes de type propre à O'Caml.

6. Une visionneuse `dvi : ojDvi`

`ojDvi` est une visionneuse `dvi`[3](voir figure 9) construite à partir de `mlDvi`[4] écrite en O'Caml que l'on a modifiée pour l'interfacer dans un schéma Modèle-Vue-Contrôleur. Les parties « Vue » et « Contrôleur », réalisées en Java, ont été séparées comme décrit à la figure 9.

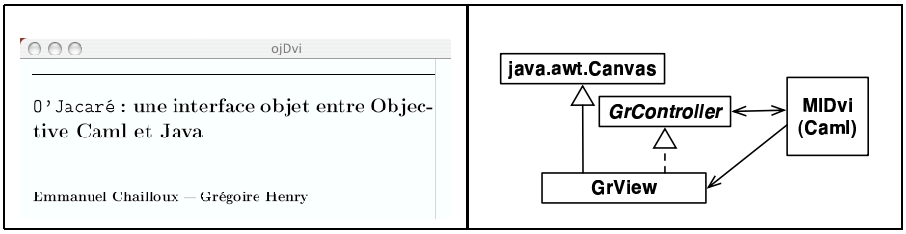

Figure 9. *Capture d'écran et organisation d'*`ojDvi`

L'objet `GrControler` contient une file d'attente d'événements, interrogée régulièrement par le « Modèle » et `GrView` est un `Canvas AWT`. Cette séparation permet notamment une fois le programme compilé, de modifier l'interface en ne touchant qu'aux fichiers Java et sans repasser par une phase d'édition de liens ; le programme engendré vérifie à l'initialisation la disponibilité des méthodes définies dans l'*IDL* (figure 10). L'interface `MlDvi`, déclarée `callback`, permet d'utiliser depuis Java un objet implanté entièrement en O'Caml : c'est le point d'accès au « Modèle ». Cet objet sera transmis à Java lors de l'appel au `main` Java. La classe `Image`, n'est utile que pour introduire un type.

3. DeVice Independent.
4. `http://pauillac.inria.fr/~miquel/mldvi-1.0.tar.gz`

```
package java.awt;
class Image { }

package mypack;
[name ml_dvi, callback] interface MlDvi {
  void run(string, GrView, GrControler);
}

class GrView {
  static final int transp;
  int width;
  int height;
  void init(int,int);
  void clear();
  void close();
  void setColor(int);
  void fillRect(int, int,int, int);
  void drawImage(java.awt.Image,int, int);
  java.awt.Image makeImage(int[],int,int);
}
```

```
class DviFrame {
  static void main(MlDvi, string[]);
}

class CamlEvent {
  static final int KEY_PRESSED;
  static final int BUTTON_DOWN;
  static final int BUTTON_UP;
  static final int MOUSE_MOTION;
  final int mouse_x;
  final int mouse_y;
  final boolean button;
  final boolean keypressed;
  final char key;
}

interface GrControler {
  CamlEvent waitBlockingEvent(int);
  CamlEvent pollNextEvent(int);
}
```

Figure 10. *Interface Vue et Contrôleur pour* `ojDvi`

Cette application nous permet une première évaluation des performances d'une application `O'Jacaré`. Pour cela on choisit de comparer `ojDvi` avec l'application d'origine `mlDvi` et `jDvi`[5] une application équivalente réalisée entièrement en Java. La figure 11 donne les temps de ces applications pour la visualisation d'un nombre de pages donné du manuel O'Caml. Les tests sont réalisés sur une architecture Intel sous le système Linux et le `jdk` de SUN. Les temps indiqués correspondent à la somme des temps *user* et *system* de la commande `time` de Linux.

	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

Figure 11. *Mesure de performances des visionneuses*

Le rapport d'efficacité entre `jDvi` et `mlDvi` est de l'ordre d'un facteur 10 qui reste constant en augmentant le nombre de pages affichées. `ojDvi` se place naturellement entre ces deux extrêmes en profitant de la rapidité de calcul d'O'Caml tout en utilisant l'affichage graphique de Java et en communiquant entre les deux *runtimes*. Ce test n'est pas suffisant pour dégager une tendance générale sur notre approche parce que les programmes Java et O'Caml sont différents. Néanmoins il montre qu'une conception prenant en compte les risques de surcoût dus à la communication entre les deux *runtimes* permet d'obtenir une efficacité acceptable.

5. `http://www-sfb288.math.tu-berlin.de/jdvi/index.html`

7. Travaux connexes et futurs

Le fait de vouloir enrichir le modèle objet de Java n'est pas récent. Odersky et Wadler introduisent du polymorphisme paramétrique à Java dans leur système [ODE 97]. Ils ajoutaient ainsi à Java un style de programmation proche d'O'Caml. L'évolution de Pizza vers Generic Java [BRA 98] reprend ce nouveau style en introduisant un polymorphisme borné. Cette volonté de généricité se retrouve bien entendu du côté de C#. Le langage intermédiaire ILX [SYM 01] se veut plus général que MSIL[6] pour faciliter les compilateurs de langages à la ML mais aussi pour introduire des *generics* en C#.

Du côté des langages fonctionnels la définition d'extensions objets comme O'Caml autorise grâce à la liaison tardive de nouvelles structurations logicielles [BOU 99]. Néanmoins le besoin d'une véritable hiérarchie de classes est réel et une ouverture du dogme du typage statique [CHA 02] devient nécessaire pour la réalisation de certains modèles de conception (*Design Patterns*).

Les interfaces externes des langages fonctionnels sont passées de liens de bas niveau avec C et C++ [DAV 94], [FIN 98] à des *IDL* pour interfacer les composants COM (H/Direct [FIN 99] et CamlIDL[7]). Toutes ces interfaces doivent gérer l'interaction des différents gestionnaires de mémoire, d'exceptions et de *threads* le plus souvent en passant par C. Une tendance actuelle est de changer d'assembleur portable pour produire directement du code-octet soit pour la machine Java comme MLj [BEN 99], soit pour .NET avec SML.NET[8]. L'intérêt est alors de n'avoir qu'une seule plate-forme d'exécution.

L'expérience d'`O'Jacaré` a montré l'intérêt du mélange de Java et O'Caml, cela tant du point de vue modèle objet que des facilités de diffusion (portabilité). L'efficacité est au rendez-vous comme le montre l'application `ojDvi` qui a d'honnêtes performances (dues au compilateur natif O'Caml utilisé pour la partie algorithmique). D'autre part les objets Java sont manipulables dans le *toplevel* O'Caml ce qui facilite le développement incrémental.

Les difficultés rencontrées pour `O'Jacaré` viennent principalement de la bibliothèque d'exécution. Le multi-threading n'est pas traité par la bibliothèque de bas niveau `camljava` (il y a qu'un seul contexte d'exécution Java global). De plus les objets circulaires mixtes sont toujours conservés. Pour la simplification de la gestion des deux *runtimes* une solution est alors de compiler O'Caml vers un code-octet commun avec le langage à interfacer (JVM ou MSIL). Nous allons adapter `O'Jacaré` pour interfacer O'Caml avec C# en utilisant le compilateur expérimental `ocamil`[9] pour .NET.

---

6. Microsoft Intermediate Language
7. `http://caml.inria.fr/camlidl`
8. `http://research.microsoft.com/projects/sml.net/`
9. `http://www.pps.jussieu.fr/~montela/ocamil`

8. Conclusion

A la différence de MLj et SML.NET qui introduisent respectivement le modèle objet de Java et de C# dans des dialectes ML, notre outil `O'Jacaré` conserve les particularités de l'extension objet d'O'Caml. Le résultat nous semble plus enrichissant en fusionnant aux polymorphismes paramétrique et de rangées d'O'Caml le typage dynamique de Java. Bien que limité au niveau de la surcharge, `O'Jacaré` est simple d'emploi et suffisamment expressif comme le montre notre visionneuse `ojDvi`. Nous espérons que de futures applications profiterons de cette complémentarité entre les modèles à la Java ou C# et O'Caml.

9. Bibliographie

[BEN 99] BENTON N., KENNEDY A., « Interlanguage Working Without Tears : Blending SML with Java », *International Conference on Functional Programming*, 1999.

[BOU 99] BOULMÉ S., « Modules, Objets et Calcul Formel », *Journées Francophones des Langages Applicatifs*, Inria, 1999.

[BRA 98] BRACHA G., ODERSKY M., STOURAMINE D., WADLER P., « Making the future safe from the past : Adding Genericity to the Java Programming Language », *International Conference on Object-Oriented Programming System, Languages and Applications*, 1998.

[CHA 00] CHAILLOUX E., MANOURY P., PAGANO B., *Développement d'Applications avec Objective Caml*, O'Reilly, 2000, (`http://www.oreilly.fr/catalogue/ocaml.html`).

[CHA 02] CHAILLOUX E., « Dynamic Object Typing in Objective Caml », *International Lisp Conference*, 2002.

[DAV 94] DAVIS H., PARQUIER P., SÉNIAK N., « Sweet harmony : The talk/c++ connection », *Lisp and Functional Programming*, 1994.

[FIN 98] FINNE S., LEIJEN D., MEIJER E., JONES S. P., « H/Direct : A Binary Foreign Language Interface for Haskell », *International Conference on Functional Programming*, 1998.

[FIN 99] FINNE S., LEIJEN D., MEIJER E., JONES S. L. P., « Calling Hell From Heaven and Heaven From Hell », *International Conference on Functional Programming*, 1999.

[GOR 98] GORDON R., *Essential JNI : Java Native Interface*, Prentice Hall, 1998.

[GOS 00] GOSLING J., JOY B., STEELE G., BRACHA G., *The Java Language Specification Second Edition*, Addison-Wesley, Boston, Mass., 2000.

[LER 02] LEROY X., « The Objective Caml system release 3.06 : Documentation and user's manual », rapport, 2002, Inria, (`http://caml.inria.fr`).

[ODE 97] ODERSKY M., WADLER P., « Pizza into Java : Translating Theory into Practice », *Symposium on Principles of Programming Languages (POPL'97)*, ACM Press, New York (NY), USA, 1997, p. 146–159.

[REM 98] REMY D., VOUILLON J., « Objective ML : An Effective Object-Oriented Extension to ML », *Theory and Practice of Object Systems*, vol. 4, n° 1, 1998, p. 27-50.

[SYM 01] SYME D., « ILX : Extending the .NET Common IL for Functional Language Interoperability », *Electronic Notes in Theoretical Computer Science*, vol. 59, n° 1, 2001.