

O’Jacaré : un pont entre OCaml et Java

Béatrice Carré¹, Emmanuel Chailloux², Xavier Clerc³, and Grégoire Henry⁴

¹ Sorbonne Universités, UPMC Univ Paris 06

`beatrice.carre@etu.upmc.fr`

² Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6

`Emmanuel.Chailloux@lip6.fr`

³ OCaml-Java Team

`xclerc@ocamljava.org`

⁴ OCamlPro SAS

`Gregoire.Henry@ocamlpro.com`

Résumé

L’interopérabilité des langages est une notion importante dans le développement logiciel à base de composants issus de différents langages. Ces composants peuvent communiquer de deux manières : soit à travers les mécanismes d’appels externes, soit en utilisant une bibliothèque d’exécution (*runtime*) commune. Dans le premier cas la difficulté est de faire cohabiter différents environnements d’exécution. Dans le deuxième cas les différents compilateurs ciblent le même environnement d’exécution ; cela permet de les gérer de manière uniforme tant du point de vue mémoire (représentation des données, typage, gestion mémoire) que contrôle (appels de fonctions et de méthodes, exceptions). On présente dans cet article l’outil O’Jacaré 2.0, un langage de description d’interface (IDL) permettant d’intégrer dans un même programme des composants Java et OCaml à travers leur modèle objet respectif. Cette nouvelle version de l’outil repose sur le compilateur `ocamljava 2.0α` qui traduit les programmes OCaml vers du byte-code Java tout en représentant la hiérarchie de classes Java dans le système de types d’OCaml. Cette approche conserve les bonnes propriétés de modèles d’exécution et de typage des deux mondes tout en les enrichissant mutuellement.

Mots-clés : interopérabilité, composants, IDL, JVM, Java, OCaml, typage

1 Introduction

Bien que la plupart des langages actuels s’influencent et soient multi-paradigmes, ils reposent principalement sur un certain modèle séquentiel de programmation associé à un typage dynamique ou statique. Les différences peuvent être très importantes comme par exemple entre le langage Haskell, représentant du modèle fonctionnel pur typé statiquement utilisant des monades et les langages objets à base de prototypes comme JavaScript permettant la manipulation dynamique de l’héritage. Cette influence entre paradigmes de programmation est bien illustrée sur le poster de l’histoire des langages de programmation de chez O’Reilly ¹. Elle se traduit par une inflation des traits de programmation. Par exemple Java, initialement objet et impératif, devient polymorphe paramétrique dans la version 1.5 et introduit les λ -expressions dès la version 1.8. Le langage Caml, initialement fonctionnel et impératif, introduit les modules paramétrés et un modèle objet pour devenir OCaml. Son système de types s’enrichit continuellement (variants polymorphes, types sommes extensibles, GADT, ...) pour une plus grande généricité dans un cadre statiquement et fortement typé. Cette hybridation des modèles se retrouve dans plusieurs évolutions actuelles comme l’ajout de systèmes de types plus ou moins forts pour des variations de PHP (comme Hack) et de JavaScript (comme TypeScript, PureScript, Flow) ou au niveau

1. http://cdn.oreillystatic.com/news/graphics/prog_lang_poster.pdf

des modèles de programmation comme les couples F#/C# ou Swift/Objective C ou encore l'intégration à la Scala.

Dans ce papier, et suivant cette mode d'hybridation, nous présentons des outils facilitant l'interopérabilité entre les langages Java et OCaml au niveau de leurs modèles objets : l'objectif est de permettre les appels de méthodes inter-langages ainsi que leurs redéfinitions.

Suivant l'exemple du CLR de .Net, nous avons choisi d'utiliser `ocamljava`² [3], un compilateur OCaml ciblant la machine virtuelle Java (JVM). Contrairement au compilateur F# qui cible le CLR, le compilateur `ocamljava` est pleinement compatible avec le code OCaml existant : en particulier, il conserve le modèle objet d'OCaml. Pour permettre la communication avec du code Java, le compilateur `ocamljava` étend le système de type d'OCaml avec des primitives d'appel de méthode Java, respectant la discipline de typage de Java. Ce mécanisme de communication, présenté brièvement dans la partie 2 de cet article, garantit le bon typage des appels de méthodes inter-langages et, au prix d'une grande verbosité, nous force à résoudre manuellement la surcharge. Pour faciliter l'utilisation de ces primitives de communication, nous présentons dans la partie 3 l'apport principal de ce papier : O'Jacaré³, un générateur de code basé sur un IDL *ad hoc* qui permet notamment d'encapsuler des objets Java dans des objets OCaml et de sous-classer un objet Java comme si l'on sous-classait un objet OCaml. Contrairement aux approches usuelles basées sur un IDL, telles que CORBA ou COM IDL, notre langage de définition d'interface se veut spécifique à la communication entre OCaml et Java. L'originalité de notre travail réside dans la cohabitation de deux modèles objets distincts au sein d'une même machine virtuelle—et incidemment de deux mécanismes de liaison tardives. La principale difficulté consiste à faire coopérer ces deux mécanismes pour permettre la liaison tardive inter-langages.

La suite de ce court article présente par l'exemple les caractéristiques principales d'`ocamljava`, puis celles d'O'Jacaré pour ensuite présenter une application complète, une visionneuse dvi, ainsi que les temps d'exécution pour apprécier l'efficacité obtenue.

2 Interopérabilité fonctionnelle et typage

Le compilateur `ocamljava` génère du bytecode Java à partir de sources OCaml ; il propose en outre une extension au typeur classique d'OCaml qui permet de manipuler des instances de classes Java depuis un programme écrit en OCaml. Cette extension du typeur, qui peut être vue comme une forme de FFI (*Foreign Function Interface*) sûre du point de vue de typage, repose sur un encodage des types Java en type OCaml. Cet encodage est invisible du point de vue de l'utilisateur, qui manipule des types comme `java.lang.String java_instance` pour les chaînes de caractères Java, où `java_instance` est du point de vue du typeur OCaml un type abstrait. La bibliothèque de support du compilateur `ocamljava` fournit un ensemble de primitives qui permettent de manipuler des valeurs de ce type. Par exemple la fonction `Java.make` permet de créer une instance d'une classe Java—voir les lignes 2 à 4 de l'exemple ci-dessous, qui correspond à un programme Swing de conversion de degrés Celsius en Fahrenheit. Cette fonction reçoit en premier paramètre une chaîne de caractères contenant respectivement le nom du constructeur ainsi que le type de ses arguments. De même, la fonction `Java.call` permet d'appeler une méthode—cf. lignes 18, 20 et 21. Le nouveau type `java_instance` s'intègre au mécanisme d'inférence de type d'OCaml et lorsqu'il n'y a pas d'ambiguïté il n'est pas nécessaire de préciser le type des arguments des méthodes Java, cf. ligne 19 où le type `GridLayout` du paramètre de la méthode `setLayout` a été implicitement déterminé.

2. <http://www.ocamljava.org>

3. <http://github.com/beaCarre/OJacare2/>

```

1 let str = JavaString.of_string;;
2 let frame = make "JFrame(String)" (str "Celsius Converter")
3 and text = make "JTextField()" () in
4 and label = make "JLabel(String)" (str "Celsius");;
5 let handler = proxy "ActionListener" (object
6   method actionPerformed _ =
7     try
8       let c = call "JTextField.getText()" text |> call "Double.parseDouble(_)" in
9       let f = Printf.sprintf "Celsius = %f Fahrenheit" ((c *. 1.8) +. 32.0) in
10      call "JLabel.setText(_)" label (str f)
11    with Java_exception je ->
12      let je_msg = JavaString.to_string (call "Throwable.getMessage()" je) in
13      let msg = str (Printf.sprintf "invalid float value (%s)" je_msg) in
14      let error = get "JOptionPane.ERROR_MESSAGE" () in
15      call "JOptionPane.showMessageDialog(_,_,_)"
16        frame msg (str "Celsius Converter: error") error
17    end);;
18 call "JTextField.addActionListener(_)" text handler;;
19 call "JFrame.setLayout(_)" frame (make "GridLayout(_,_,_)" 11 21 31 31);;
20 ignore (call "JFrame.add(Component)" frame text);;
21 ignore (call "JFrame.add(Component)" frame label);;
22 call "JFrame.setSize(_,_)" frame 3001 801;;
23 let exit = get "WindowConstants.EXIT_ON_CLOSE" ();;
24 call "JFrame.setDefaultCloseOperation(int)" frame exit;;
25 call "JFrame.setVisible(_)" frame true;;

```

La bibliothèque de support fournit quelques primitives de conversion entre les types de bases d'OCaml et les objets équivalents en Java—cf. par exemple la fonction `JavaString.of_string` renommé en `str` à la ligne 1 et utilisée par exemple aux lignes 2 et 4. La fonction `Java.proxy` permet d'implémenter à l'aide d'un objet OCaml une interface Java—cf. ligne 5 où un objet anonyme réalise l'interface `ActionListener`. Les langages OCaml et Java possèdent tous les deux des mécanismes de gestion des exceptions. Il est possible de lever une exception Java à l'aide de la fonction `Java.throw` et de rattraper dans du code OCaml une exception levée en Java : celle-ci est encapsulée dans une exception OCaml nommée `Java_exception`, cf. ligne 7 et 11. En conclusion, ce mécanisme de FFI permet d'effectuer depuis OCaml toutes les manipulations possibles sur des instances de classes Java, au prix d'une grande verbosité : à chaque appel de constructeur ou de méthode, il est nécessaire de préciser sa signature.

3 Interface objet et liaison tardive inter-mondes

O'Jacaré permet de pallier le problème de verbosité de la FFI de `ocamljava` en proposant une représentation différente des types Java en OCaml. L'idée fondamentale est de remplacer le type abstrait `java_instance` par des classes OCaml. En pratique, O'Jacaré repose sur le typage proposé par le compilateur `ocamljava`, et encapsule des valeurs de type `java_instance` au sein d'instance de classe OCaml. Outre que cela permet de réduire la verbosité nécessaire à la manipulation d'éléments Java depuis OCaml, cela permet également de ne présenter à l'utilisateur qu'un seul modèle objet et surtout d'étendre en OCaml des classes Java, ce que le compilateur `ocamljava` n'autorise pas. La génération de ces classes OCaml encapsulantes repose sur un IDL (*Interface Description Language*) qui sert à décrire les champs, méthodes, et constructeurs des classes Java que l'on souhaite pouvoir manipuler depuis OCaml. Ainsi, la description suivante permet de générer le code OCaml encapsulant une classe `Point` et sa sous-classe `ColoredPoint` :

```

1 class Point {
2   int x; int y;
3   [name point] <init> (int,int);
4   void moveto(int,int);
5   string toString();
6   boolean eq(Point); }
1 [callback] class ColoredPoint extends Point {
2   string getColor();
3   [name colored_point] <init> (int,int,string);
4   [name eq_colored_point] boolean eq(ColoredPoint);
5 }

```

Notre IDL utilise une syntaxe très proche de celle d'une interface Java, elle est décorée de quelques annotations entre crochets permettant de guider le générateur de code ; en particulier l'annotation à la ligne 4 (colonne de droite) permet de renommer en OCaml la méthode `eq` de la classe Java. Le langage OCaml n'ayant aucun mécanisme de surcharge, ce renommage est indispensable. À partir de cette description, l'outil O'Jacaré va produire deux classes OCaml, nommée `point` et `colored_point`. Une fois ces classes générées, il est possible d'en créer des instances puis d'appeler des méthodes sur celles-ci comme s'il s'agissait d'objets OCaml, comme le montre le programme suivant.

```

1 let p = (new point 0 0)#moveto 1 1;;
2 p#set_y 2;;
3 printf "%B\n" (p#get_x = 1 && p#get_y = 2);; (* display: true *)
4 printf "%B\n" (p#eq (new point 10 20));; (* display: false *)
5 printf "%B\n" (p#eq (new colored_point 1 2 "Green" :> jPoint)) (* display: true *)

```

L'appel de méthode en OCaml est dénoté par le caractère `#`. Les variables d'instances décrites dans l'IDL ont été remplacées par des méthodes d'accès et de modifications—cf. les appels aux méthodes `set_x` et `set_y` aux lignes 2 et 3. Le sous-typage étant rarement implicite en OCaml, il est parfois nécessaire d'ajouter une contrainte de sous-typage explicite, comme à la ligne 6 où l'objet OCaml encapsulant un objet Java de la classe `ColoredPoint` est coercé vers le type `jPoint`. Il correspond à un type objet généré par O'Jacaré et ressemblant au code suivant :

```

1 class type jPoint = object method set_x : int -> unit method get_x : unit -> int
2 method eq : jPoint -> bool method toString : unit -> string (* ... *)
3 method _get_jni_jPoint : _jni_jPoint
4 end

```

On y retrouve les interfaces des méthodes décrites dans l'IDL ainsi qu'une méthode à usage interne, `_get_jni_jPoint`, qui permet de récupérer la référence à l'objet Java encapsulé.

Liaison tardive inter-mondes En plus de simplifier la manipulation d'objets Java depuis du code OCaml, O'Jacaré fournit aussi un moyen de redéfinir les méthodes d'une classe Java en OCaml et d'étendre ainsi le mécanisme de liaison tardive par des appels inter-langages. Dans l'exemple précédent, la classe Java `ColoredPoint` contient une méthode `toString`, son implémentation n'est pas détaillée ici, mais nous supposons qu'elle fait appel à la méthode `getColor` pour obtenir la chaîne de caractères décrivant la couleur. Le programme suivant redéfinit cette méthode en OCaml puis effectue un appel à `toString` pour illustrer pour le mécanisme étendu de liaison tardive.

```

1 let cp = new colored_point 3 4 "Yellow"
2 let cb = object
3   inherit _stub_colored_point 5 6 "Blue" as super
4   method getColor () = "OCaml" ^ (super#getColor ())
5 end;;
6 print_endline (cp#to_string ());; (* display: (3,4):Yellow *)
7 print_endline (cb#to_string ());; (* display: (5,6):OCamlBlue *)

```

4 Application : une visionneuse dvi

`ojDvi` est une visionneuse dvi⁴ basée sur `mldvi`⁵. Cette dernière a été modifiée pour être interfacée selon un schéma “modèle-vue-contrôleur”. Les parties “vue” et “contrôleur” sont écrites en Java, tandis que le “modèle” est implémenté côté OCaml. Cet exemple est intéressant du point de vue performance du fait des nombreuses communications dans les deux sens entre les deux mondes lors de l'exécution. Cette application permet alors de mesurer l'avantage de n'utiliser qu'un seul runtime. Le tableau suivant nous montre le temps de cette application pour

4. DeVice-Independent

5. <http://www.pps.univ-paris-diderot.fr/~miquel/soft/mldvi-1.0.tar.gz>

la visualisation d'un nombre de pages donné du manuel d'OCaml, avec mLDvi en code natif, mLDvi en byte-code, jDvi⁶ écrite en Java, O'Jacaré 1⁷[2] et O'Jacaré 2.

	50 p.	100 p.	200 p.	300 p.	588 p.
mLDvi natif	0.2	0.35	0.7	1	2
mLDvi byte-code	0.4	0.7	1.1	1.6	2.8
ojDvi (O'Jacaré 2)	1.3	1.7	2	2.3	3
ojDvi (O'Jacaré 1)	1.3	2.2	5	8.3	15
jDvi	2	3.5	7	10	20

En terme de performances O'Jacaré se place entre Java et OCaml. Le programme Java jDvi étant différent du programme OCaml, il est impossible de tirer une conclusion sur la tendance générale de notre approche. Mais concernant les deux ojDvi, la nouvelle approche semble plus performante.

5 Conclusion

Bien que les langages de programmation s'influencent mutuellement et tendent à incorporer des éléments provenant d'autres paradigmes, ils n'en conservent pas moins des points forts et des domaines d'application privilégiés. De fait, si OCaml est nettement plus adapté que Java à l'écriture de traitements symboliques (type compilateurs), Java conserve l'avantage dans le domaine des interfaces graphiques ou des bases de données. L'interopérabilité entre (ces) deux langages est donc une question importante, tant les développements modernes reposent sur des bases de code mêlant plusieurs langages.

L'outil O'Jacaré présenté dans cet article permet de manipuler facilement des classes Java depuis un code purement OCaml. Il présente un gain important par rapport à l'utilisation du compilateur `ocamljava` seul, en permettant de faire ces manipulations par le biais d'objets OCaml, plutôt que par le biais de fonctions dédiées. Il est, en outre, possible d'écrire du code OCaml pour étendre des classes Java. Le surcoût induit par O'Jacaré est très faible ; en mémoire, c'est le coût de l'encapsulation et en temps, c'est le coût du *dispatch* de méthode OCaml.

Les pistes d'évolution du binôme O'Jacaré/`ocamljava` sont essentiellement de plusieurs natures. Une extension de l'IDL viserait à supporter à la fois les *generics* et les *lambdas*. Le générateur de code actuel privilégie l'utilisation de classe Java depuis OCaml mais il pourrait être étendu afin de faciliter l'utilisation d'une bibliothèque OCaml depuis Java. Du côté du compilateur `ocamljava` un support plus large de la plate-forme viserait à permettre le déploiement sur Android. Une meilleure intégration des deux outils (compilateur et générateur de code) permettrait éventuellement de se passer d'un IDL et de générer automatiquement le code nécessaire à la communication. Néanmoins, cette approche réintroduirait des ambiguïtés liées aux interactions entre le mécanisme de surcharge de Java et les mécanismes d'inférences de type d'OCaml, des annotations de types explicites à la manière de [1] seraient sans doute nécessaires.

Références

- [1] Alexandre Bergel. Reconciling method overloading and dynamically typed scripting languages. *Computer Languages, Systems & Structures*, 37(3) :132 – 150, 2011.
- [2] Emmanuel Chailloux and Grégoire Henry. O'jacaré, une interface objet entre objective caml et java. *L'OBJET*, 10(2-3) :75–88, 2004.
- [3] Xavier Clerc. Ocaml-java : From ocaml sources to java bytecodes. In *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, pages 71–85, 2012.

6. <http://www-sfb288.math.tu-berlin.de/jdvi/download.html>

7. version précédente basée sur OCaml natif et des communications via C et l'interface JNI de la JVM.