Les objets de Bigloo

Manuel Serrano
Manuel.Serrano@cui.unige.ch
http://cuiwww.unige.ch/~serrano/

• *Université de Nice–Sophia-Antipolis*
• *Centre Universitaire d'Informatique, University of Geneva*
24, rue General-Dufour, CH-1211 Geneva 4, Switzerland

Résumé

Bigloo est un système implantant une variante du langage Scheme. Pour la version 1.9, nous lui avons ajouté une couche objet. Nous la présentons ici avec ses principales caractéristiques linguistiques et quelques détails de son implantation. Puisque Bigloo est un système auto-gène et que pour la nouvelle version nous l'avons entièrement réécrit avec des objets, nous avons pu évaluer l'impact du « style » objet tant dans l'écriture même du programme implantant le compilateur que dans ses performances. Nos conclusions sont que les objets n'entraînent pas des dégradations notables des performances, que leur apport en terme d' « aisance » de programmation n'est pas fondamental mais, en revanche, que les objets facilitent l'écriture de programmes extensibles et modifiables.

Introduction

Bigloo est une implantation assez libre du langage Scheme. Dès l'origine de cette réalisation, l'objectif n'a pas été de proposer une implantation stricte de Scheme tel que défini dans [7] mais plutôt d'en proposer une adaptation que nous pensons réaliste et pragmatique. Bigloo n'implante pas « tout » Scheme. Par exemple, les récursions terminales ne sont pas garanties d'être exécutées sans consommation de pile et peu d'arithmétiques sont implantées (Bigloo n'implante que les entiers et les réels). En revanche, Bigloo propose de nombreuses extensions : de l'analyse lexicale et syntaxique, du filtrage, un mécanisme d'exception simple, une interface externe et un langage de module.

La version 1.9 de Bigloo étend encore Scheme au moyen d'une couche objet très largement inspirée de MEROON [10] de C. Queinnec. C'est cette extension que nous présentons dans cet article.

Le paradigme objet détient une position hégémonique dans le monde moderne de la programmation. La majorité des programmeurs semble s'accorder à dire que ce style de programmation facilite le développement et la maintenance des programmes. Chaque communauté met des objets dans ses langages. C'est vrai pour les nouveaux langages tel que le déjà très célèbre Java [6] ainsi que pour les plus anciens qui se mettent au goût du jour tel que O'Caml [11] ou $ML_{\leq}$ [2]. L'auteur de cet article confesse une trop petite expérience dans l'utilisation de langages objet. C'est pour pallier à cette impardonnable faiblesse que nous avons entrepris d'ajouter des objets à Bigloo.

Le compilateur Bigloo est auto-gène, c'est-à-dire qu'il est écrit dans le langage qu'il implante, et qu'il est utilisé pour se compiler lui-même. Ce système est maintenant de taille respectable puisque, incluant les librairies, il compte environ 60000 lignes de Scheme et 2500 lignes de C. Faire évoluer ce programme accapare la presque totalité de notre énergie de programmeur. Bigloo est maintenant pratiquement le seul développement sérieux que nous avons le temps de faire. Ajouter des objets à Bigloo n'avait donc pour nous qu'une seule justification, réécrire le compilateur en utilisant cette nouvelle extension. Réécrire Bigloo avec des objets était pour nous le seul moyen de trancher sur les questions importantes. Est-ce que les objets facilitent vraiment la vie du programmeur? Est-ce que l'utilisation du paradigme objet n'implique pas une importante perte de performances? La comparaison entre l'ancien Bigloo (1.8) et le Bigloo moderne (1.9) nous a donné des éléments de réponse à ces questions. Nous présentons cette comparaison en conclusion de cet article.

« Langage objet » est un terme bien vague qui peut décrire des langages bien différents. Il existe plusieurs catégories de langages objets très différentes les unes des autres. Distinguons-en ici deux : le modèle de Smalltalk [5] qui est caractérisé par le fait que les méthodes sont associées aux classes et le modèle de Clos [1] où les méthodes sont associées à des fonctions génériques. Chacun de ces modèles peut être décliné en plusieurs variantes utilisant un typage statique ou dynamique, utilisant de l'héritage simple ou multiple, et utilisant du *simple* ou *multiple dispatch*. Notre choix s'est porté pour le modèle de Clos, dans sa variante utilisant de l'héritage simple et du *simple dispatch*. Dans la Section 1, nous présentons les raisons qui nous ont amenées à faire ce choix, ainsi que les principaux traits linguistiques de notre modèle objet.

Notre implantation des objets est assez standard car les techniques que nous utilisons sont, pour la plupart, bien connues. Néanmoins, notre implantation possède quelques particularités, dues en partie, au langage de modules de Bigloo. Dans la Section 2, nous présentons l'implantation des objets.

1. Les objets de Bigloo

Bigloo est construit autour de ses modules. Il nous faut les présenter en prémices avant de pouvoir exposer notre système objet.

1.1. Les modules

Les modules de Bigloo ont deux fonctions essentielles : permettre de la compilation séparée et augmenter le nombre d'erreurs potentiellement détectables par le compilateur. Ils sont simples car ils ont été conçus avec le souci d'être implantables facilement. Nous ne présentons ici des modules que la partie indispensable à la présentation ultérieure des objets.

Un module est l'unité de compilation de Bigloo. Il est physiquement incarné par un ou plusieurs fichiers. Il a la syntaxe suivante:

```
(module module-name
   (import importation+)*
   (export exportation+)*
   (static static+)*)

optional-body
```

1.1.1. Les clauses de module

Les clauses d'*importation* servent à importer des liaisons dans le module. Lors d'une importation, on précise juste l'identificateur à importer et dans quel module il se trouve.

Les clauses d'*exportation* et les clauses *statiques* jouent un rôle très proche. Elles signalent au compilateur que le module implante certaines liaisons et indiquent si les dites liaisons sont visibles depuis d'autres modules (i.e. exportées) ou pas (i.e. statiques). Ces clauses ne sont pas constituées d'identificateurs mais de prototypes. On peut ainsi exporter des variables (des liaisons modifiables) ou des fonctions (liaisons non modifiables). Les clauses statiques sont facultatives (i.e. les liaisons du module non référencées dans une clause sont statiques).

Voici en exemple le module `fib` qui importe les fonctions `fib-fx` du module `fib-fixnum` et `fib-fl` du module `fib-flonum` et qui exporte la fonction `fib`.

```
(module fib
   (import (fib-fx fib-fixnum)
           (fib-fl fib-flonum))
   (export (fib x)))

(define (fib x) (if (fixnum? x) (fib-fx x) (fib-fl x)))
```

Lors de la compilation, Bigloo effectue un certain nombre de vérifications :

- Toutes les variables et fonctions référencées doivent être définies dans le module ou mentionnées dans une clause d'importation (i.e. il ne peut y avoir de variable libre).
- Tous les identificateurs mentionnés dans les clauses statiques ou d'exportation doivent être définis dans le module avec une valeur compatible avec leur prototype.
- Les identificateurs mentionnés dans les clauses statiques ou d'exportation dont le prototype est une fonction ne doivent pas être affectés.
- Tous les appels de fonction où l'opérateur fonctionnel est connu comme étant une fonction (i.e. l'identificateur en position de fonction a un prototype de fonction) doivent avoir un nombre d'arguments compatible avec le prototype de la fonction invoquée.

1.1.2. Les corps de module

Les expressions constituant le corps d'un module peuvent être des définitions de variable ou fonction (au moyen de la forme Scheme `define`) ou bien des expressions Scheme quelconques. *Initialiser* un module est l'opération qui consiste à établir les liaisons variable/valeur et à évaluer les expressions qui ne sont pas des définitions. Les liaisons et les expressions sont exécutées dans leur ordre d'apparition dans le module.

Les graphes d'importation des modules ne sont pas limités à des arbres. C'est-à-dire que deux modules, ou plus, peuvent s'importer mutuellement. L'ordre d'initialisation des modules est alors non spécifié.

1.2. Le typage

Le langage Scheme est très fortement ancré dans une tradition de typage dynamique. Autrement dit, en Scheme, c'est lors de l'exécution que les vérifications de type sont effectuées. On a coutume de dire qu'un programme ML peut être compilé que s'il n'existe pas d'interprétation

fausse vis-à-vis des types alors qu'un programme Scheme est compilé s'il existe au moins une interprétation juste. Bigloo se place clairement dans la tradition Scheme mais contrairement aux systèmes répandus, il essaye effectivement de montrer qu'il existe au moins une interprétation juste des programmes avant de les compiler. Le résultat est que, parfois, Bigloo refuse de compiler des programmes car il détecte des erreurs de typage.

L'autre tradition de Scheme (partagée avec ML cette fois) est de ne pas écrire de types dans les programmes. Il n'y a pas moyen de déclarer qu'une variable ne contient des données que d'un type précis. Bigloo tourne le dos à cette tradition et encourage les annotations de type. Elles sont facultatives mais elles permettent d'obtenir un code produit de meilleure qualité et de détecter plus d'erreurs lors de la compilation.

Un type Bigloo est soit un type Scheme atomique (e.g. un entier Scheme, une chaîne de caractères Scheme, une liste Scheme, ...), un type externe (e.g. un entier C, une chaîne de caractères C, une structure C, ...), ou un type associé à une classe (les classes seront présentées dans la Section 1.3.1).

Les annotations de type peuvent être insérées dans les clauses d'exportation des modules, dans les définitions de variables ou dans les blocs lexicaux. La construction syntaxique est [<var-id>]`::`<type-id>. Voici en exemple, un module implantant la fonction de Fibonnacci sur les entiers.

```
(module fibo
   (export (fib::long ::long)))

(define (fib x) (if (< x 2) 1 (+ (fib (- x 1)) (fib (- x 2)))))
```

La clause `(fib::long ::long)` signifie simplement que le module implante une fonction nommée `fib` qui prend un argument de type `long` et retourne un résultat lui-même de type `long`. Il est inutile de reporter les annotations de type sur les définitions lorsqu'elles sont déjà indiquées sur une clause de module ; c'est pourquoi, ici, l'annotation `::long`[1] n'est marquée que sur l'exportation de `fib`.

Le typage de Bigloo le place vraiment en marge de Scheme. D'une part parce que, comme nous l'avons dit, la tradition n'est pas d'utiliser des annotations de type et, d'autre part, parce qu'il oblige à renoncer définitivement à la récursion terminale. En effet, lorsque le résultat d'une fonction est annotée et que Bigloo ne peut démontrer que la valeur calculée par la fonction est du type annoté, il insère alors un test dans le corps de la fonction. Ainsi la fonction:

1. Nous attirons ici l'attention du lecteur sur le fait que le type `long` dénote les entiers C et non pas les entiers Scheme pour indiquer que Bigloo manipule indifféremment les données Scheme et C.

```
(define (foo::a-type x) (bar x))
```

peut être compilée en:

```
(define (foo::a-type x)
   (let ((tmp (bar x)))
      (if (a-type? tmp)
          tmp
          (error))))
```

On voit ici pourquoi les récursions terminales ne peuvent plus être garanties.

1.3. Le modèle

Entre le modèle de Smalltalk où les méthodes sont définies dans les classes et le modèle de Clos où les méthodes sont regroupées dans des fonctions génériques, nous avons opté pour la deuxième solution. Ce choix a été motivé par la nature et l'architecture du programme, le compilateur Bigloo, que nous souhaitions réécrire avec des objets.

Le compilateur Bigloo est un programme Scheme de 40000 lignes. Il lit un programme à compiler et construit l'arbre de syntaxe abstraite représentant le programme. Cet arbre est une structure composée, dans sa version actuelle, de 23 types de nœuds différents. Il y a un nœud pour représenter les constantes, les affectations de variable, les conditionnelles, les appels de fonction, etc. Le compilateur est structuré en passes. La version actuelle contient une vingtaine de passes. Chacune d'elle peut être considérée comme une procédure modifiant l'arbre de syntaxe abstraite. Le moteur du compilateur est donc une fonction Scheme ressemblant à :

```
(define (compilateur src)
   (let ((ast (build-ast src)))
      (macro-expand! ast)        ;; 1ère passe
      (function-inline! ast)     ;; 2ème passe
      ...
      (code-generate! ast)))     ;; 20ème passe
```

Chaque passe simple est implantée dans un fichier. Les plus complexes sont éclatées en plusieurs fichiers. Pour la nouvelle version, nous avons impérativement souhaité conserver cette décomposition parce qu'elle nous semble être plus naturelle.

Cette volonté écarte donc de notre choix le modèle objet de Smalltalk car avec ce modèle, la façon naturelle d'implanter notre compilateur serait d'abandonner notre structure guidée par les passes au profit d'une structure guidée par l'arbre de syntaxe abstraite. Chaque passe du

compilateur serait éclatée dans toutes les classes implantant les nœuds de l'arbre. Il n'y aurait plus, par exemple, un fichier contenant le code de la passe `function-inline!`. Ce code devrait être éclaté dans les classes implantant les nœuds de l'arbre de syntaxe.

Cependant pour notre compilateur, le modèle de Smalltalk faciliterait l'ajout de nouveaux nœuds à l'arbre de syntaxe abstraite. Ajouter une passe avec le modèle de Smalltalk nécessite de modifier et de re-compiler tout le code déjà existant (parce qu'il faut ajouter les définitions des méthodes de la nouvelle passe) ; en revanche, ajouter un nœud n'exigerait pas de modification du code déjà présent. Ajouter un nœud revient à ajouter une nouvelle construction au langage compilé alors qu'ajouter une passe correspond bien souvent à l'implantation d'une extension du compilateur (comme une nouvelle optimisation). L'expérience nous a montré que le langage évolue bien moins vite que le compilateur lui-même. Avec le modèle de Clos, ces problèmes ne se posent pas. Il est aussi facile de rajouter des passes que des nouveaux nœuds.

1.3.1. Les déclarations de classes

Les classes sont déclarées dans les clauses d'exportation ou dans les clauses statiques d'un module. Il est donc possible de rendre visible à d'autres modules une déclaration de classe ou de limiter la portée de sa déclaration à un module. La syntaxe abrégée de la déclaration d'une classe est:

```
(class class-id[::super-class-id]
   <field>*)

<field> ::= field-id[::type-id]
  | (field-id[::type-id] <option>*)
  | (* field-id[::type-id] <option>*)

<option> ::= read-only
  | (default value)
```

Une classe ne peut hériter que d'une seule super-classe. Les classes dont la super-classe n'est pas spécifiée héritent de la seule classe de la librairie, la classe `object`. Le type associé à une sous-classe est un sous-type du type associé à la super-classe.

Un champ peut être typé (par l'annotation d'un ::type-id), ne pas être modifiable (`read-only`) et avoir une valeur par défaut (`default`). Les champs dont la clause commence par un `*` sont dits indexés. C'est-à-dire que ces champs ne contiennent pas une valeur mais une série de valeurs. Les champs indexés (que nous tenons directement de MEROON) pourraient, par exemple, être utilisés pour implanter les chaînes de

caractères.

Voici de possibles déclarations pour les classiques `point` et `point-3d`:

```
(module les-petits-points
   (export (class point
              (x::double read-only (default 0.0))
              (y::double read-only (default 0.0)))
           (class point-3d::point
              (z::double read-only (default 0.0)))))
```

1.3.2. Les fonctions génériques

Les déclarations de fonctions génériques sont des déclarations de fonctions, annotées du mot clé `generic`. Elles peuvent dont être exportées, ce qui signifie qu'elles peuvent être utilisées depuis d'autres modules et que des méthodes peuvent être ajoutées à ces fonctions depuis d'autres modules. Elles peuvent être statiques et donc pas visibles depuis d'autres modules. Dans ce cas, aucune méthode ne peut-être ajoutée à celles introduites par le module définissant la fonction générique.

La syntaxe de définition d'une fonction générique est très proche de celle de la définition d'une fonction:

```
(define-generic (fun-id[::type-id] arg-id[::class-id] ...)
   optional-body)
```

Les fonctions génériques doivent avoir au moins un argument puisque c'est leur premier argument qui est utilisé pour réaliser la liaison dynamique des méthodes (*dynamic dispatch*). Cet argument peut être typé et dans ce cas, le type T doit être associé à une classe et il est impossible d'ajouter à la fonction générique des méthodes dont le premier argument n'est pas un sous-type de T. Les fonctions génériques peuvent avoir un corps. Ce corps est évalué si lors d'un envoi de message, aucune méthode n'est définie pour le type du premier argument. Si cette situation se produit alors que la fonction générique n'a pas de corps, une erreur est signalée. Voici un exemple de définition de fonction générique dont l'argument discriminant n'est pas typé :

```
(define-generic (display-point p . port)
   (if (number? p) (display p port) (error)))
```

1.3.3. Les méthodes

Les méthodes sont déclarées au moyen de la forme syntaxique :

```
(define-method (fun-id[::type-id] arg-id[::class-id] ...)
   body)
```

C'est une erreur de définir une méthode s'il n'existe pas de fonction générique définie avec un prototype compatible (type des arguments et du résultat en relation de sous-typage avec ceux de la fonction générique). Voici un exemple de méthode :

```
(define-method (display-point p::point . port)
   (with-access::point p (x y)
      (print port "{" x ", " y "}")))
```

Il est possible d'invoquer à partir de la méthode la méthode qui aurait été utilisée si la méthode n'avait pas été définie (i.e. d'invoquer la méthode associée à la super-classe de l'objet). Cela se fait au moyen de la forme syntaxique `(call-next-method)`. Les arguments de la nouvelle méthode appelée sont les mêmes que les arguments de la méthode qui utilise le `call-next-method`. Cette forme n'est accessible que dans la définition d'une méthode.

1.3.4. Les instances

Lorsqu'une classe `class-id` est déclarée, Bigloo génère automatiquement un prédicat `class-id?`, un allocateur `instantiate::class-id`, un copieur `duplicate::class-id`, des accesseurs (pour un champ `x` non indexé, l'accesseur se nomme `class-id-x` ; si le champ est indexé, l'accesseur se nomme alors `class-id-x-ref`), des modificateurs (pour un champ `x` modifiable, le modificateur se nomme `class-id-x-set!`) et, enfin, une forme d'accès abrégée `with-access::class-id` qui permet d'accéder et de modifier les champs en les référençant seulement par leur nom. Voici un exemple d'allocation et d'accès à une instance :

```
(let ((p (instantiate::point-3d (y -3.4) (x 1.0))))
  ;; La valeur d'initialisation d'un champ peut être omise de la liste
  ;; des arguments si ce champ a une valeur par défaut comme c'est
  ;; le cas ici pour le champ z des point-3d.
   (with-access::point-3d p (x y z)
      (sqrt (+ (sqr x) (sqr y) (sqr z)))))
```

1.3.5. Introspection

Bigloo propose des fonctionnalités d'introspection pour les objets. Schématiquement, il est possible de déterminer lors de l'exécution le nom des champs et la classe de tout objet. Il est aussi possible d'avoir accès à la liste de ses accesseurs et de ses modificateurs. Grâce à cette possibilité, la bibliothèque d'exécution standard de Bigloo dispose d'un écrivain « raisonnable » d'objets et implante le prédicat `equal?` pour les objets.

1.3.6. Sérialisation

Les objets de Bigloo peuvent être sérialisés et dé-sérialisés au moyen des fonctions de librairie `obj->string` et `string->obj`. Cette sérialisation/dé-sérialisation respecte le partage présent dans chaque objet. La complexité de la sérialisation est en $\mathcal{O}(2n)$ où n est la taille de l'objet à sérialiser. La dé-sérialisation a la même complexité. Lorsqu'un objet est dé-sérialisé, une vérification est effectuée pour s'assurer que la classe de cet objet actuellement existant dans l'exécution est bien la même que celle qui existait lorsque l'objet a été sérialisé.

2. L'implantation des objets

Les modules et leur implantation conditionnent très fortement l'implantation de la couche objet. Nous devons faire ici une présentation de leur implantation.

2.1. L'implantation des modules

La conception des modules les rend simples à implanter. En particulier, pour compiler un module, il n'est pas nécessaire que les modules importés soient préalablement compilés. Bigloo n'utilise les clauses de module que pour connaître les liaisons exportées. Il suffit donc que le fichier source du module existe et qu'il contienne, au moins, sa déclaration de module.

2.1.1. L'initialisation des modules

Exécuter un programme, signifie pour Bigloo, initialiser tous les modules qui le composent. Comme les graphes d'importation des modules ne sont pas restreints à des arbres, il n'existe pas d'ordre garantissant que tout module est initialisé avant qu'une de ses liaisons ne soit utilisée. Les modules sont initialisé de la façon suivante. Il existe un module particulier (nommé pour l'exposé `main`), indiqué par le programmeur, qui joue le rôle de premier module à initialiser. Bigloo commence son exécution par marquer `main` comme initialisé, initialise tous les modules importés par `main` et exécute le corps de `main`. Voici le pseudo-code réalisant l'initialisation des modules :

```
(initialize-module*! (list main))

(define (initialize-module*! modules)
   (if (null? modules)
       ;; l'exécution se termine ici
       (exit 0))
       (begin
          ;; on initialise le premier module
          (initialize-module! (car modules))
          ;; on passe au suivant...
          (initialize-module*! (cdr modules))))
```

Initialiser un module signifie, le marquer, initialiser les modules qu'il importe et évaluer son corps.

```
(define (initialize-module! a-module)
   (if (not (initialized? a-module))
       (begin
          ;; on marque le module comme étant initialisé
          (mark-initialized! a-module)
          ;; on initialise tous les modules qu'il importe
          (initialize-module*! (imported-modules a-module))
          ;; on exécute son corps
          (evaluate-body! a-module))))
```

La procédure d'initialisation permet de garantir que si le graphe d'importation d'un programme est un arbre, alors l'initialisation des modules de ce programme sera faite selon une descente en profondeur de cet arbre.

2.1.2. Cohérence entre module importé et module important

Bigloo garantit que lorsqu'un module `mod-1` importe un module `mod-2`, et que `mod-1` est compilé avant `mod-2`, les modifications qui peuvent intervenir sur `mod-2` avant sa compilation le laisse compatible avec la forme qu'il avait lors de la compilation de `mod-1`. Ceci est assuré par un système de nombre identifiant (*checksum*). Lorsque `mod-1` est compilé, un nombre identifiant `num-2` est calculé à partir de ses clauses d'exportation. Lorsque `mod-2` est compilé à son tour, un nouveau nombre identifiant est calculé `num-2'`. Lors de l'initialisation des modules, Bigloo s'assure que `num-2` est égal à `num-2'`.

2.2. L'implantation des classes

Bigloo n'offre ni MOP (*meta object protocol* [8]), ni méta-classe et ses classes ne sont même pas des objets ! Pour nous épargner les problèmes classiques d'initialisation du système d'exécution des objets, nous avons

fait des classes une structure primitive de Bigloo. Voici présentée, en pseudo code C, cette structure:

```
struct a_class {
   symbol_t name;         // Son nom
   long     depth;        // Sa profondeur dans l'arbre d'héritage
   class_t  super;        // Son unique directe super-classe
   class_t  sub[];        // Ses sous-classes
   class_t  ancestors[];  // Ses super-classes (même indirectes)
   class_t *(*alloc)();   // Son allocateur d'instances
   long     checksum;     // Son checksum identificateur
   field_t  field[];      // Ses informations de réflection
} class_t;
```

La fonction de librairie **add-class!** qui prend en argument un nom de classe, une super-classe, un allocateur d'instances, un *checksum* et une liste de champs alloue une structure de type **class_t** et l'ajoute à la liste globale des classes utilisées dans l'exécution.

Lorsqu'un module est initialisé, s'il contient des déclarations de classes, il les déclare au moyen de **add-class!** avant d'évaluer son corps. Lors de l'exécution, il existe une variable globale pour chaque classe qui porte le même nom que la classe. Ainsi les déclarations des classes **point** et **point-3d** de nos exemples précédents ressemblent à :

```
(define point
   (add-class! 'point object allocate-point 244543 '(x y)))
(define point-3d
   (add-class! 'point-3d point allocate-point-3d 720461 '(z)))
```

La variable globale qui contient la structure de classe n'est exportée que si la classe déclarée est elle même exportée.

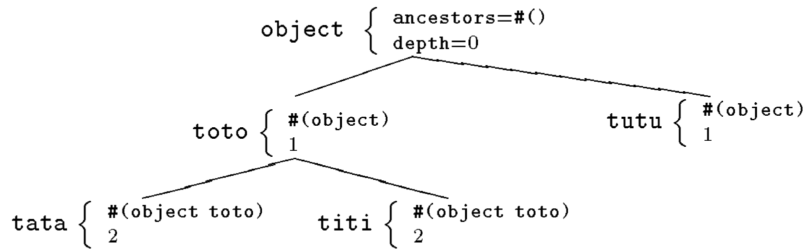
2.2.1. Les prédicats de classe

Nous savons grâce à Cohen [4] et grâce à l'implantation de Modula-3 [3] que tester l'appartenance à une classe en sous-typage simple peut être effectué en temps constant.

Pour l'algorithme de Cohen, chaque classe possède un vecteur **ancestors** de toutes ses super-classes (directe et indirectes) et sa profondeur **depth** dans l'arbre d'héritage. Considérons les classes suivantes :

```
(class toto::object)
(class tata::toto)
(class titi::toto)
(class tutu::object)
```

Le graphe des classes annotées de leur champ **ancestors** et **depth** est :



Un objet `obj`, instance d'une classe `cla`, est aussi une instance d'une autre classe `a-class` si une de ces deux conditions est vérifiée :

- `a-class` est `cla`.
- `a-class` n'est pas une classe plus « profonde » que `cla` et `a-class` est la classe présente dans le vecteur d'ancêtres de `cla` à la position qui est la profondeur de `a-class`.

Tester qu'un objet est, par exemple, une instance de `toto` revient donc à tester si cet objet est une instance directe de `toto` ou si `toto` est égale à la classe qui est à l'indice `1` (profondeur de `toto` dans l'arbre d'héritage) dans le vecteur d'ancêtres de la classe directe de l'objet. Le pseudo-code Scheme implantant le test de Cohen pourrait être :

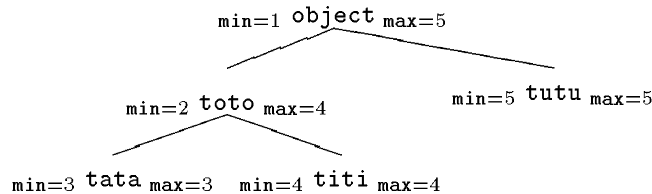
```
(define (cohen-is-a? obj a-class)
   (if (and (object? obj) (class? a-class))
       (let ((o-class (object-class obj)))
          (if (eq? o-class a-class)
              #t
              (let ((direct-depth (class-depth o-class))
                    (depth        (class-depth a-class)))
                 (if (<fx depth direct-depth)
                     (let ((anc (class-ancestors o-class)))
                        (eq? (vector-ref anc depth) a-class))
                        #f))))
       #f))
```

L'intérêt de cette technique est qu'elle fonctionne même si le système permet l'ajout dynamique de classes car ajouter une classe ne change pas les vecteurs d'ancêtres des classes déjà installées.

La méthode utilisée dans Modula-3 utilise une numérotation de l'arbre d'héritage. Les classes sont numérotées lors d'un parcours en profondeur à main gauche de l'arbre d'héritage. Chaque classe est annotée de deux numéros : son numéro `min` (lors de la descente de l'arbre ; chaque fois qu'une nouvelle classe est atteinte, un compteur global est

incrémenté et la nouvelle valeur de ce compteur est le numéro de cette classe atteinte) et le numéro le plus élevé **max** de ses sous-classes. Tester l'appartenance revient alors à tester si le numéro de la classe de l'objet testé est compris dans l'intervalle `[min..max]` de la classe testée.

L'arbre d'héritage et une numérotation selon ce schéma de nos classes données en exemple pourraient être :



Tester qu'un objet est, par exemple, une instance de la classe **toto** revient donc à tester si le **min** de sa classe est compris dans l'intervalle `[3..4]`. Le pseudo-code Scheme implantant cette technique pourrait être :

```
(define (modula3-is-a? obj a-class)
   (if (and (object? obj) (class? a-class))
       (let ((o-class (object-class obj)))
          (if (eq? o-class a-class)
              #t
              (let ((o-class-min (class-min o-class))
                    (o-class-max (class-max o-class))
                    (class-num   (class-min a-class)))
                 (and (>= class-num o-class-min)
                      (<= class-num o-class-max)))))
       #f))
```

Comme on peut le constater, la technique de Modula-3 est un peu plus efficace que celle de Cohen parce qu'elle permet d'éviter un accès mémoire et parce qu'elle permet l'économie du vecteur d'ancêtres (**ancestors**) dans les classes. Sa contrainte est qu'elle exige une numérotation complète de l'arbre d'héritage. Chaque ajout d'une nouvelle classe peut nécessiter une renumérotation de tout l'arbre d'héritage. Comme nous l'avons montré dans la Section 2.2, dans Bigloo, les classes sont ajoutées au fur et à mesure de l'initialisation des modules. Il n'y a pas, à proprement parler, un temps de déclaration des classes et un temps d'exécution. Les deux sont imbriqués. Lors de l'exécution, tout se passe comme si les classes étaient ajoutées dynamiquement. Si nous avions choisi la méthode de Modula-3, nous aurions du renuméroter l'arbre des classes après les déclarations de classes de chaque module. Cette solution est peut-être viable mais nous ne l'avons pas essayé. Nous avons opté pour la méthode de Cohen.

2.3. L'implantation des instances

Bigloo conserve des informations de type sur les nœuds de l'arbre de syntaxe abstraite tout au long de la compilation, jusqu'à la génération de code. Dans sa version actuelle, le compilateur produit du C typé. Une passe de compilation introduit des tests de type et des coercicions qui permettent d'utiliser une représentation mixte [9] des données dans la bibliothèque d'exécution. Il en va de même pour les objets. Les objets de Bigloo sont compilés vers des structures C typées. Voici un exemple de compilation de classe :

```
(class une-classe (x::long read-only) y::char)
```

est compilée en :

```
typedef struct une_classe_125 {
   header_t header;    // Le numéro de la classe dont les objets sont
                       // des instances.
   obj_t    widening;  // Un champ utilisé pour la sérialisation
   long     x;         // Le 1er champ
   char     y;         // Le 2ème champ
} *une_classe_125_t;
```

2.3.1. L'implantation des fonctions d'accès

Puisque notre couche objet utilise de l'héritage simple, l'accès aux champs des instances est toujours très efficace ; il ne nécessite qu'un accès mémoire. Bigloo implante les fonctions d'accès et de modification des champs au moyen de fonctions *inline*, introduites par la construction `define-inline`, et la construction `pragma`. Les fonctions *inline* se distinguent des fonctions traditionnelles par le fait que lorsqu'elles sont invoquées, l'appel fonctionnel est remplacé, par le compilateur, par une copie du corps de la fonction appelée. La forme `pragma` permet d'insérer dans le code source Bigloo du code C. Cette forme est un peu l'équivalent Scheme des `#asm` de certains compilateurs C.

Voici les fonctions générées automatiquement par Bigloo pour accéder et modifier le champ `y` des instances de `une-classe` :

```
(define-inline (une-classe-y::char obj::une-classe)
   (pragma::char "(((une_classe_125_t)($1))->y)" obj))

(define-inline (une-classe-y-set!::obj obj::une-classe val::char)
   (pragma::obj
    "((((une_classe_125_t)($1))->y) = ((char)($2)), BUNSPEC)"
    obj
    val))
```

2.4. L'implantation des fonctions génériques

Bigloo utilise de l'héritage simple et du *dispatch* simple, c'est-à-dire que seul le type du premier argument des méthodes est examiné pour faire la liaison dynamique. Les fonctions génériques sont des sortes de paquets de méthodes. Elles sont représentées dans Bigloo par des fermetures dont le corps sera vu ultérieurement et dont l'environnement contient deux variables. L'une des variables est liée à une fonction qui est le corps par défaut de la fonction générique. L'autre variable est liée à une table de méthodes. Cette table a exactement pour taille le nombre de classes définies. Nous ne faisons pas de compaction des tables de méthodes. C'est peut-être quelque chose qu'il faudra que nous envisagions à l'avenir mais comme Bigloo n'offre ni classe ni fonction générique en librairie et que ce n'est pas un système « tout objet » il nous semble que la taille des tables de méthodes reste suffisamment petite. Par exemple, le compilateur Bigloo utilise environ 100 classes et 75 fonctions génériques. Sur une architecture 64 bits, l'espace total occupé par les tables de méthodes est donc de 75*100*8 octets, soit environ 60k, ce qui, de nos jours, n'est que broutille.

2.4.1. L'initialisation des fonctions génériques

Comme nous l'avons vu dans la Section 2.2, les classes déclarées dans un module sont initialisées avant que les expressions constituant le corps du module ne soient évaluées. Il en va de même avec les fonctions génériques. Lors de l'initialisation d'un module, Bigloo initialise les classes, initialise les fonctions génériques au moyen de la fonction de la librairie `add-generic!`, initialise les méthodes au moyen de la fonction de la librairie `add-method!`, puis évalue les expressions du module.

La fonction `add-generic!` alloue les vecteurs de méthodes pour les fonctions génériques. La fonction `add-method!` insère une nouvelle méthode (i.e. une simple fonction Scheme) dans la table des méthodes d'une fonction générique. Lorsqu'une nouvelle classe est initialisée (ce qui peut se produire lors de l'initialisation ultérieure d'un autre module), toutes les fonctions génériques sont ré-ajustées pour que leur table de méthodes ait une entrée correcte pour la nouvelle classe.

Étudions en guise d'exemple la compilation de ce petit module :

```
(module les-petits-points
   (export (class point
              (x::double read-only (default 0.0))
              (y::double read-only (default 0.0)))
           (class point-3d::point
              (z::double read-only (default 0.0)))
           (generic show ::point)))

(define-generic (show p::point))
(define-method (show p::point) (display "point"))
(define-method (show p::point-3d) (display "point-3d"))
```

Bigloo remplace alors les formes `define-generic` et `define-method` de la façon suivante :

```
(define show
   (let ((method-table #unspecified)
         (default-body #unspecified))
      (lambda (p) generic-body)))

(define (show-default p) (error))
(define (show-point p) (display "point"))
(define (show-point-3d p) (display "point-3d"))

;; La déclaration de la fonction générique va allouer un vecteur
;; de méthodes et mettre à jour les deux variables libres.
;; method-table et default-body
(add-generic! show show-default)

;; Une déclaration de méthode insère la fonction implantant le corps de
;; la méthode dans la table des méthodes de la fonction générique.
(add-method! show point show-point)
(add-method! show point-3d show-point-3d)
```

Il apparaît clairement, qu'avec ce schéma de compilation, des méthodes peuvent être ajoutées à une fonction générique même si elles sont déclarées dans un autre module que celui de la fonction générique. Il suffit en effet que la fonction générique soit exportée pour que la fermeture qui lui est associée par Bigloo le soit aussi et donc que des `add-method!` la concernant puissent être compilées.

2.4.2. La liaison dynamique (*dynamic dispatch*)

Étudions maintenant les liaisons dynamiques. Cela a, jusqu'à présent, été laissé en suspens puisque nous n'avons pas donné le corps des fonctions génériques. Lors d'une invocation de méthode, c'est la fonction générique à laquelle la méthode est associée qui est invoquée ; la fonction générique examine le type du premier argument, accède à son vecteur de

méthodes pour trouver la méthode à évaluer. Les classes sont numérotées de façon croissante. Les numéros de classes sont utilisés pour accéder aux vecteurs de méthodes. Afin de rendre un peu plus efficace l'invocation de méthode, les instances n'incluent pas un pointeur vers leur classe mais directement le numéro de leur classe. L'invocation de méthode coûte ainsi 3 accès mémoire (un pour accéder au numéro de la classe de l'objet, un deuxième pour accéder au vecteur de méthodes et un troisième pour accéder à la méthode dans le vecteur), un test, un appel de fonction connue (l'appel à la fonction générique) et un appel inconnu, l'appel à la méthode. Voici donné en Scheme le corps de la fonction générique `show` :

```
(define show
   (let ((method-table #unspecified)
         (default-body (lambda (p) (error))))
      ;; La variable method-table sera
      ;; modifiée par la fonction add-generic!
       (lambda (p)
         (let* ((class-num (object-class-num p))
                (method (vector-ref method-table class-num)))
            (if (procedure? method)
                (method p)
                (default-body p))))))
```

Il faut noter ici que `show` a été déclarée comme prenant en argument des objets de type `point` ; il n'est donc pas utile de tester, dans la fonction générique, que l'argument est une instance de `object`. Si aucune information de type n'avait été indiquée lors de la déclaration de `show`, cette fonction générique aurait compté un test supplémentaire.

Les méthodes sont de simples fermetures Scheme. Ces fermetures ont un environnement vide (i.e. elles n'ont pas de variables libres) sauf si elles utilisent la forme `call-next-method`. Si tel est le cas, la méthode est compilée dans un environnement où une variable `call-next-method` est liée à une sorte de pseudo fonction générique. Nous ne donnons pas ici le code de cette nouvelle fonction car elle ressemble très fortement aux fonctions génériques standard sauf, qu'elle « déroule » le vecteur de méthodes jusqu'à trouver la méthode la plus proche des super-classes.

2.4.3. La liaison dynamique optimisée

Nous avons présenté dans la Section 2.4.2 l'implantation standard de la liaison dynamique. Lorsque Bigloo optimise les programmes, il utilise un schéma différent. Avec ce nouveau schéma, les méthodes qui sont déclarées dans le même module sont insérées à l'intérieur du code des fonctions génériques. Ces méthodes ne sont plus référencées alors dans le vecteur de méthodes par un pointeur de code mais par un numéro. Un `case` Scheme permet ensuite de trouver la méthode à exécuter. Voici le

résultat de la compilation de **show** optimisée :

```
(define show
   (let ((method-table '#(#unspecified #unspecified 0 1 ...))
         ;; Ici le vecteur est partiellement calculé lors de la compilation.
         ;; si l'on suppose que les classes point et
         ;; point-3d ont les numéros 2 et 3, alors le vecteur
         ;; de méthodes est initialisé comme indiqué.
          (default-body (lambda (p) (error))))
      (lambda (p)
         (let* ((class-num (object-class-num p))
                (method (vector-ref method-table class-num)))
            (case method
               ((0) 'point)
               ((1) 'point-3d)
               (else
                (if (procedure? method)
                    ;; On teste si c'est une fonction car des méthodes
                    ;; ont pu être ajoutées depuis d'autres modules.
                     (method p)
                     (default-body p))))))))
```

Avec ce schéma de compilation, il n'y a plus de **add-method!** pour les méthodes définies dans le même module que leur fonction générique. Comme d'autres modules peuvent enrichir la fonction générique, il faut tester, lorsque le **case** échoue, si **method** est une fonction (i.e. une méthode déclarée pour la fonction générique dans un autre module). Si la fonction générique n'est pas exportée, ce test est inutile.

Cette technique de compilation ne permet pas une liaison dynamique plus efficace. Elle remplace simplement l'appel de fonction calculé des méthodes par un saut indexé, ce qui est presque équivalent. Son intérêt réside dans l'économie de code et des données qu'elle permet. Il n'est plus ici utile de compiler des fermetures pour les méthodes car leur corps est intégré directement dans la fonction générique et il n'y a plus de structures de données pour représenter ces fermetures.

Ce schéma de compilation complique légèrement la compilation des formes `call-next-method`. Nous ne présentons pas ici cette compilation car le micmac effectué est sans grand intérêt.

3. Conclusion

Nous présentons dans cette conclusion quelques réflexions informelles que nous a inspiré le passage aux objets pour l'écriture du compilateur de Bigloo.

3.1. Bigloo1.8 vs. Bigloo1.9

Les deux compilateurs de Bigloo1.8 et Bigloo1.9 sont très comparables. À quelques détails près, ils implantent les mêmes passes de compilation et utilisent les mêmes optimisations. Le langage de Bigloo1.9 est comme nous l'avons présenté dans cet article plus étendu que celui de Bigloo1.8. Le nouveau compilateur est donc un peu plus complexe car il contient, en plus, le compilateur de la couche objet. Malgré ce code supplémentaire, les deux compilateurs sont restés de taille très proche : 36000 lignes (29500 sans commentaires) pour Bigloo1.8 et 40000 lignes pour Bigloo1.9 (30000 sans commentaires). Les objets semblent donc permettre un code plus compact. Ce résultat doit être considéré avec précaution car, pour Bigloo1.9, nous n'avons que réimplanté un programme déjà existant : il est donc un peu mieux écrit que Bigloo1.8.

Nous pensons que le principal apport de la technologie objet est le sous-typage. C'est à notre sens ce qui permet économie et réutilisation du code. En revanche, nous ne pensons pas que le *dynamic dispatch* représente une avancée majeure. Pour un langage n'offrant pas de structures de contrôles évoluées, comme c'est par exemple le cas de C, l'ajout du *dispatch* aide vraiment le programmeur (dans ce sens C++ apporte vraiment *un* plus à C). En revanche, pour un langage offrant du filtrage, l'intérêt du *dispatch* est beaucoup plus faible tant ces deux constructions sont proches.

Le couple fonction générique/méthode possède l'avantage de pouvoir être éclaté sur plusieurs fichiers (les méthodes peuvent être déclarées dans d'autres fichiers que leur fonction générique) et d'être extensible sans modification du code déjà existant. En revanche, il est un peu plus fastidieux à utiliser. Examinons les deux styles dans un fragment de code implantant une descente dans un arbre :

```
 1: (define-generic (walker node::node))
 2: (define-method (walker node::leaf)
 3:    (with-access::leaf node (value)
 4:       value))
 5: (define-method (walker node::binary)
 6:    (with-access::binary node (left right)
 7:       (cons (walker left) (walker right))))
 8: (define-method (walker node::ternary)
 9:    (with-access::ternary node (left middle right)
10:        (list (walker left)
11:              (walker middle)
12:              (walker right))))
```

Le même programme avec des structures et du filtrage :

```
 1:  (define (walker node::node)
 2:     (match-case node
 3:        (#{leaf ?value}
 4:         value)
 5:        (#{binary ?left ?right}
 6:         (cons (walker left)
 7:               (walker right)))
 8:        (#{ternary ?left ?middle ?right}
 9:         (list (walker left)
10:               (walker middle)
11:               (walker right)))))
```

Le code utilisant du filtrage (**match-case**) est plus court et il présente l'avantage qu'il est facile de lui faire précéder un prélude (il suffirait dans notre exemple d'ajouter du code entre les lignes *1* et *2*). Bigloo ne possède actuellement pas d'outils de mise au point. Nous en sommes donc parfois réduis, lorsque les choses vont vraiment mal, à insérer du code de mise au point dans le source. Cela est facile à faire avec du filtrage mais plus difficile avec du *dispatch* car ce code de mise au point doit être dupliqué dans toutes les méthodes.

3.1.1. Les performances

La taille de Bigloo1.8 (comprenant le compilateur et l'interprète) est de 1.5 Mega sur Sparc pour 1.8 Mega pour Bigloo1.9. Il y a donc un accroissement d'environ 20%. On peut penser que cet accroissement de la taille de l'exécutable est normal puisque le langage compilé est plus vaste. Ce n'est pas totalement vrai puisque les deux compilateurs ont le même nombre de lignes de code source. Néanmoins, on peut affirmer que le passage aux objets n'a pas entraîné une explosion de la taille des codes compilés.

Pour mesurer les performances du compilateur en terme de vitesse d'exécution, nous nous sommes limités à une unique mesure comparative. Il n'est pas très juste de mesurer le temps passé par Bigloo1.8 pour compiler un fichier et de le comparer au temps de Bigloo1.9 car la nouvelle version applique plusieurs fois certaines passes. C'est par exemple le cas de l'intégration fonctionnelle qui est appliquée en début de la compilation et en fin, après l'insertion des tests de type. Il est donc normal que Bigloo1.9 soit un peu plus lent. Nous avons mesuré le temps passé par Bigloo pour compiler son lecteur. C'est un assez bon test puisque c'est un des plus longs fichiers à compiler. Le lecteur contient une grammaire lexicale qui est compilée vers un automate déterministe qui est lui même expansé vers une fonction Scheme. Sur une Sparc 5 avec 64 Mega de mémoire, la compilation du lecteur prend 10.3 s avec la version 1.8 et 11.8 s avec la version 1.9. La nouvelle version est donc un

peu plus lente d'environ 15%. Il est difficile d'établir la cause de cette perte de performance. D'une part, les applications multiples de passes ralentissent le compilateur, d'autre part, puisqu'il est auto-compilé, grâce à ces multiples applications, le compilateur résultant devrait être plus performant. Il est très difficile de savoir si le fait d'appliquer deux fois une passe suffit à améliorer le compilateur produit d'un facteur qui lui permet de récupérer le temps passé dans la deuxième application de la passe. Nous n'essayerons pas de savoir à quoi correspondent exactement les 15% perdus. Nous nous contenterons seulement de remarquer que même si 15% ne sont pas négligeables, les performances « générales » du compilateur ont été préservées.

3.2. Récapitulatif

Nous avons présenté la couche objet que nous avons ajoutée à Bigloo. Nous avons présenté certaines de ses caractéristiques linguistiques. Nous avons ensuite montré comment les objets sont implantés. Nous avons présenté la méthode utilisée pour implanter les prédicats de classe et pour réaliser le *dispatch* des méthodes. En conclusion, nous avons livré quelques remarques inspirées par la réécriture du compilateur de Bigloo en utilisant les objets présentés dans l'article. En particulier, nous avons montré que le code source du compilateur est resté de la même taille alors que le langage compilé est devenu plus vaste, que la taille de l'exécutable résultat de l'auto-compilation du compilateur a augmenté d'environ 20% et que les performances globales du compilateur semblent s'être légèrement dégradées (d'un peu moins de 15%).

Références

[1] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. **Common lisp object system specification**. In *special issue*, number 23 in SIGPLAN Notices, September 1988.

[2] F. Bourdoncle and S. Merz. **Type Checking Higher-Order Polymorphic Multi-Methods**. In *Symposium on Principles of Programming Languages*, pages 40–53, 1997.

[3] L. Cardelli and al. **Modula-3 Report (revised)**. Technical Report 52, DEC Systems Research Center, Palo-Alto, CA, USA, November 1989.

[4] N. Cohen. **Type-extension type tests can be performed in constant time**. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, 1992.

[5] A. Goldberg and D. Robson. **Smalltalk-80: The Language and Its Implementation**. Addison-Wesley, 1983.

[6] J. Gosling, B. Joy, and G. Steele. **The Java™ Language Specification**. Addison-Wesley, 1996.

[7] IEEE Std 1178-1990. **IEEE Standard for the Scheme Programming Language**. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[8] G. Kicsales, J. des Rivières, and D. Bobrow. **the Art of the Metaobject Protocol**. MIT Press, Cambridge, Mass., USA, 1992.

[9] X. Leroy. **Unboxed objects and polymorphic typing**. In *Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, January 1992.

[10] C. Queinnec. **Designing** MEROON **V3**. In *Workshop on Object-Oriented Programming in Lisp*, 1993.

[11] D. Rémy and J. Vouillon. **Obective ML: A simple object-oriented extension of ML**. In *Symposium on Principles of Programming Languages*, pages 40–53, 1997.