Programmation multi-niveaux dans Scala

Damien Pous, **[dpous@ens-lyon.fr]**

Stage de maîtrise, Juin, Juillet et début Août 2003

au Laboratoire des Méthodes de Programmation (Lamp),
École Polytechnique Fédérale de Lausanne (Epfl),
sous la direction du Professeur **Martin Odersky**.

Table des matières

1 Introduction

Au laboratoire des méthodes de programmation de l'École polytechnique fédérale de Lausanne, on étudie différents concepts afin faciliter la programmation.

Actuellement, une bonne partie du laboratoire travaille sur un langage de programmation : SCALA (Scalable Langage). Il s'agit d'un langage orienté objet, dont les fonctions sont des valeurs de première classe, possédant un système de type puissant, inféré localement. Il se veut le plus général possible : tout langage spécialisé devrait idéalement pouvoir être écrit sous forme d'une librairie SCALA ; cela est facilité par une compatibilité très forte avec *Java*[1] : toute classe Java peut être héritée par une classe SCALA.

Cependant, cette généricité a un coût, de telles librairies étant en quelque sorte des micro-compilateurs, transformant à l'exécution des constructions propres à un langage donné en constructions du langage SCALA.

En 1997, Walid Taha a proprement défini dans sa thèse [TS97] le concept de *programmation multi-niveaux* (Multi-Stage Programming), dont l'idée de base est de fournir des outils efficaces pour manipuler des *programmes* à l'exécution, afin de pouvoir factoriser le code, tout en l'optimisant. On peut alors écrire une fonction prenant certains arguments en entrée, et fournissant en sortie une nouvelle fonction, optimisée pour ces arguments.

L'outil standard d'analyse syntaxique *Yacc* peut être vu comme de la programmation à deux niveaux : on fournit la grammaire, et *Yacc* construit du code efficace pour parser cette grammaire. Un langage pour la programmation à plusieurs niveaux doit en quelque sorte fournir les outils nécessaires à l'écriture de "fonctions" telles que `yacc`.

Mon travail au sein de l'équipe consistait donc à introduire cette notion dans SCALA. Ceci devrait permettre d'une part de dériver des implémentations efficaces de langages spécialisés à partir de SCALA ; mais aussi de se rendre compte des modifications à apporter au travail de Walid Taha pour passer d'un langage fonctionnel pur[2] à un langage essentiellement objet, tant au niveau du typage qu'au niveau de l'implémentation.

[1] *Java* étant l'équivalent Lausannois de l'*OCaml* Français

[2] dans sa thèse, Walid Taha étend le lambda calcul fortement typé. Deux implémentations en ont résulté : *MetaML* [TS], extension de SML et *MetaOCaml* [CTHL], extension d'OCaml

2 Programmation multi-niveaux

2.1 Intuitivement

Quel programmeur ne s'est jamais demandé s'il valait mieux généraliser une fonction afin de la rendre facilement réutilisable, ou au contraire, la spécialiser afin de la rendre plus efficace ?

La programmation multi-niveaux donne une réponse : généraliser la fonction, mais de manière à ce que le système puisse la spécialiser en une fonction efficace, à l'exécution. De cette manière, on paie certes le coût de la spécialisation, mais ce coût devient négligeable si la fonction obtenue est ensuite intensément utilisée.

C'est de cette manière que Walid Taha introduit la programmation multi-niveaux : le programmeur “annote”[3] son code de manière à indiquer quelles réductions peuvent être faites une fois pour toutes. Il s'agit en quelque sorte de donner la possibilité au programmeur d'intervenir sur l'ordre d'évaluation ; afin de combler les lacunes des stratégies usuelles d'appels par nom ou par valeur.

Pour ce faire, trois annotations sont introduites :
- <.> pour les expressions comme pour les types, par exemple
  <fun x -> x> : <'a -> 'a>
  qui se lit “code de la fonction identité, de type code d'une fonction 'a -> 'a”.
- ~. seulement pour les expressions, lu “splice”. Cette annotation ne peut se trouver qu'entre des piquants (code), et possède en quelque sorte le type <'a> -> 'a. Cette annotation permet de forcer l'évaluation d'une portion de code, et de l'insérer en ce point.
- run . de type <'a> -> 'a. Cet opérateur permet de compiler du code et d'en obtenir l'objet qu'il représente.

2.2 Par l'exemple

Voici un exemple de session MetaML :

```
# let a = <2+3>;;
val a: <int> = <2+3>
# run a;;
- : int = 5
# let b = <1+~a>;;
val b: <int> = <1+2+3>
# let bb = <(+) ~a>;;
val bb: <int->int> = <(+) (1+2+3)>
# let c = <<~c+4>>;;
val c: <<int>> = <<~c+4>>
# run c;;
- : <int> = <1+2+3+4>
```

[3]Certains systèmes sont étudiés pour annoter automatiquement le code (*BTA* : Binding-Time Analysis [GJ96, BW94, PS94, Ber]), le fait que le programmeur ait à annoter lui-même le code pourrait donc sembler caduc. Cependant rien ne semble pouvoir remplacer la connaissance du programmeur ; et toute recherche d'algorithme générique commence par une étude manuelle de cas particuliers...

2.2.1 La fonction puissance

Considérons la fonction puissance naïve (la même idée s'applique à l'algorithme *efficace*…) :

```
(* p: int -> int -> int *)
let rec p n =
  fun x -> match n with
  | 0 -> 1
  | n -> x * (p (n-1) x)
```

Intuitivement, on a envie de pouvoir fixer `n` et de dérouler la récursion une seule fois : le type de `p` deviendrait "réellement" $int \to (int \to int)$.

On commence par annoter le type. On veut une fonction qui prenne un entier en argument, et qui renvoie le code d'une fonction sur les entiers :

```
(* p1: int -> <int -> int> *)
```

Puis on annote le corps pour obtenir le type désiré :

```
let rec p n =
  <fun x -> ~match n with
   | 0 -> <1>
   | n -> <x * (~(p (n-1)) x)>
  >
```

Lors de l'évaluation de `(p 2)`, la stratégie d'évaluation force le déroulage de la récursion, jusqu'à obtenir `<fun x -> x*x*1>`[4] , ce qui correspond bien à une spécialisation optimisée de notre fonction puissance. Il ne reste alors plus qu'à appeler `run` pour compiler cette fonction :

```
# let p3_c = p1 3;;
val p3_c: <int -> int> = <fun x -> x*x*x*1>
# let p3 = run p3_c;;
val p3: int -> int = <fun>
# p3 5;;
- : int = 125
```

2.2.2 Le produit scalaire

Un second exemple de programmation multi-niveaux, considérons le produit scalaire de deux vecteurs, représentés par deux listes :

```
(* axpb: int -> int -> int *)
let axpb a x b =
  match a with
  | 0 -> b
  | 1 -> x+b
  | a -> a*x+b
(* iprod: int list -> int list -> int *)
```

[4] on obtient en fait `<fun x->x*((fun x->x*((fun x->1) x)) x)>`, mais une optimisation relativement simple permet l'élimination des applications inutiles

```
let rec iprod a =
  fun b -> match a with
  | [] -> 0
  | x::q -> let y::p = b in axpb x y (iprod q p)
```

on souhaite pouvoir fixer le premier vecteur, d'où :

```
(* axpb1: int -> <int> -> <int> -> <int> *)
let axpb1 a x b =
  match a with
  | 0 -> b
  | 1 -> <~x + ~b>
  | a -> <a * ~x + ~b>
(* iprod: int list -> <int list -> int> *)
let rec iprod1 a =
  <fun b -> ~match a with
   | [] -> <0>
   | x::q -> <let y::p = b in ~(axpb1 x <y> <~(iprod1 q) p>)>
  >
```

on peut alors utiliser cette fonction :

```
# let iprod103_c = iprod1 [1; 0; 3];;
val iprod103_c: <int list -> int> =
  <fun b -> let y::p = b in
             y + (let y::p = p in
                    (let y:p = p in
                       3 * y + 0
                    )
                 )
  >
# (run iprod103_c) [2; 3; 3];;
- : int = 11
```

L'utilisation d'un tel produit scalaire pour construire le produit de matrices peut avoir des conséquences impressionnantes en termes de temps de calcul lorsque la première matrice est creuse.

On pourrait aussi reprendre cette idée pour implémenter directement le produit matriciel, ce serait très efficace dans le domaine de la synthèse d'images où les matrices utilisées dans une scène sont fixes, et souvent creuses (translations).

2.3 Formellement

On commence par définir la notion de *niveau*, différence entre le nombre de *piquants* et le nombre de *splices* entourant syntaxiquement une expression. Dans l'expression $\langle\langle\tilde{}a\rangle\rangle$ la variable a apparaît au niveau 1.

Outre le fait que les opérateurs `run .` et `~.` doivent toujours être appliqués à des valeurs de type code, le système de type doit vérifier que toute variable liée à un certain niveau n'est pas utilisée à un niveau moindre (considérer $\langle\lambda x.\tilde{}x\rangle$) tout en permettant l'utilisation dans des niveaux plus élevés ($\lambda a.\langle a\rangle$ ou l'addition dans $\langle\lambda x.x+1\rangle$), cette dernière propriété étant appelée *persistance à travers les niveaux* (cross-stage persistence).

2.3.1 Lambda calcul multi-niveaux : λ-M

Partant du lambda calcul fortement typé, sur les entiers, on aboutit à la syntaxe des expressions et des types suivante :

$$e \ := \ i \mid x \mid e\ e \mid \lambda x.e \mid \langle e \rangle \mid \tilde{}e \mid \text{run } e$$

$$\tau \ := \ int \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle$$

Il faut retenir le niveau auquel une variable est liée, les environnements de typage prennent donc la forme :

$$\Gamma \ := \ [] \mid \Gamma, x : \tau^n \qquad (n \text{ entier})$$

On modifie alors les règles de typage du lambda calcul, afin qu'elles prennent en compte le niveau : "$\Gamma \vdash^n x : \tau$" se lit "*x est de type τ au niveau n et sous l'environnement* Γ".[5] Les règles, détaillées figure 1 sont alors assez naturelles, on remarquera toutefois la quantification universelle sur l'entier naturel m dans la règle Var.

$$(\textsc{Int}) \quad \frac{}{\Gamma \ \vdash^n \ i : int} \qquad\qquad \frac{x : \tau^{n-m} \in \Gamma}{\Gamma \ \vdash^n \ x : \tau} \quad (\textsc{Var})$$

$$(\textsc{Lam}) \quad \frac{\Gamma, x : \tau'^n \ \vdash^n \ e : \tau}{\Gamma \ \vdash^n \ \lambda x.e : \tau' \to \tau} \qquad\qquad \frac{\Gamma \ \vdash^n \ e_2 : \tau' \qquad \Gamma \ \vdash^n \ e_1 : \tau' \to \tau}{\Gamma \ \vdash^n \ e_1\ e_2 : \tau} \quad (\textsc{App})$$

$$(\textsc{Code}) \quad \frac{\Gamma \ \vdash^{n+} \ e : \tau}{\Gamma \ \vdash^n \ \langle t \rangle : \langle \tau \rangle} \qquad\qquad \frac{\Gamma \ \vdash^n \ e : \langle \tau \rangle}{\Gamma \ \vdash^{n+} \ \tilde{}e : \tau} \quad (\textsc{Spl})$$

$$(\textsc{Run}) \quad \frac{\Gamma \ \vdash^n \ e : \langle \tau \rangle}{\Gamma \ \vdash^n \ rune : \tau}$$

Fig. 1 – Système de type de λ-M

Les quatre premières règles sont les règles de typage standard du lambda calcul, généralisées à tout niveau n. Les trois dernières permettent de contraindre l'utilisation des trois annotations introduites.

Une sémantique à grandes étapes en appel par valeur pour ce calcul est donnée figure 2.

Les réductions Int, Lam et App sont celles du lambda calcul, toutes celles situées à droite traduisent l'aspect figé de toute expression de niveau non nul. Run avant d'être appliquée, force la réduction de tout splice de niveau un, via Spl+, puis *libère* le code ainsi construit.

Autrement dit, parmi les relations $(\overset{n}{\hookrightarrow})_n$ ainsi définies, le cas $n = 0$ représente la sémantique de l'évaluation proprement dite tandis les cas $n > 0$ représentent les opérations de *construction* du code aux niveaux non nuls : les

[5] insistons sur le fait que le niveau ne fait pas partie du type, mais uniquement du jugement de typage

$$\text{(Int)}\quad \frac{}{i \overset{n}{\hookrightarrow} i} \qquad\qquad \frac{}{x \overset{n+}{\hookrightarrow} x}\quad \text{(Var+)}$$

$$\text{(Lam)}\quad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e} \qquad\qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\lambda x.e_1 \overset{n+}{\hookrightarrow} \lambda x.e_2}\quad \text{(Lam+)}$$

$$\text{(App)}\quad \frac{\begin{array}{c} e_1 \overset{0}{\hookrightarrow} \lambda x.e \\ e_2 \overset{0}{\hookrightarrow} e_3 \\ e[x := e_3] \overset{0}{\hookrightarrow} e_4 \end{array}}{e_1\ e_2 \overset{0}{\hookrightarrow} e_4} \qquad\qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_3 \qquad e_2 \overset{n+}{\hookrightarrow} e_4}{e_1\ e_2 \overset{n+}{\hookrightarrow} e_3\ e_4}\quad \text{(App+)}$$

$$\text{(Run)}\quad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle \qquad e_2 \overset{0}{\hookrightarrow} e_3}{\text{run } e_1 \overset{0}{\hookrightarrow} e_3} \qquad\qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\text{run } e_1 \overset{n+}{\hookrightarrow} \text{run } e_2}\quad \text{(Run+)}$$

$$\text{(Spl+)}\quad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle}{\tilde{}e_1 \overset{1}{\hookrightarrow} e_2} \qquad\qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\tilde{}e_1 \overset{n++}{\hookrightarrow} \tilde{}e_2}\quad \text{(Spl++)}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\langle e_1 \rangle \overset{n}{\hookrightarrow} \langle e_2 \rangle}\quad \text{(Code)}$$

FIG. 2 – Sémantique en appel par valeur pour λ-M

β-réductions n'interviennent qu'au niveau 0 (règles APP et APP+) et les splices de niveau 1 sont substitués une fois leur contenu réduit (*construit*) avec la règle SPL.

Il est important de remarquer que cette sémantique manipule des termes ouverts : le terme clôt $\langle \lambda x.\tilde{}\langle x \rangle \rangle$ se réduit en $\langle \lambda x.x \rangle$, mais fait intervenir la dérivation $\langle x \rangle \overset{n}{\hookrightarrow} \langle x \rangle$ (BRK), réduction du terme ouvert $\langle x \rangle$.

Faisons aussi le lien avec l'idée de la modification de l'ordre d'évaluation : bien que cette sémantique corresponde à l'appel par valeur, le point essentiel est qu'elle permet les réductions *sous lambda abstraction* aux niveaux non nuls, via la règle LAM+. Dans le terme $\langle \lambda x.(\tilde{}(f\ y)\ x) \rangle$ le β-redex $(f\ y)$ sera réduit avant même que x soit liée. Il ne s'agit donc pas seulement d'une question d'évaluation paresseuse, et cette sémantique ne peut pas être implémentée simplement par une compilation vers le lambda-calcul, en faisant apparaître des continuations pour retarder certaines réductions.

2.3.2 Le problème de run

Malheureusement, ce calcul n'est pas *satisfaisant*, car il existe des termes typables, dont la sémantique n'est pas définie, comme par exemple

$$\langle \lambda x.\tilde{}\text{run } \langle x \rangle \rangle : \langle \langle 'a \rangle \to 'a \rangle$$

En effet, le splice force l'évaluation de run $\langle x \rangle$ alors que x n'est pas encore liée. Ce problème provient du fait que l'on manipule des termes ouverts, et que les règles de typage permettent à run de retirer le type *code* d'une expression, sans en abaisser le niveau (règle RUN).

Walid Taha fournit une première solution en renforçant le système de type, de manière à abaisser le niveau lors de l'application de run . Cependant, le système en résultant est trop contraignant, car il empêche toute abstraction de run (λx.run x n'est pas typable) et par là même toute utilisation raisonnable du calcul (en ML, `let a = <5> in run a` correspond au terme $((\lambda a.\text{run } a) \ \langle 5 \rangle))$. Il ajoute donc des règles de typage permettant de relâcher cette contrainte dans les cas *suffisamment sûrs* (variables liées au top-level). Il semble que dans son implémentation (MetaML) il utilise le premier système de type, laissant au programmeur l'entière responsabilité d'utiliser run à bon escient.

Sa seconde solution consiste à introduire de nouvelles annotations pour les expressions et les types, afin que le programmeur puisse définir explicitement l'ensemble des variables libres d'une portion de code. Le calcul en résultant (λ^{BN}) est nettement plus complexe, mais cette fois cohérent. Son inconvénient majeur est la lourdeur des annotations que le programmeur doit introduire.

3 νObj, Scala

Scala [Ode03b, Ode03a] est un langage conçu au Lamp dans l'idée d'unifier programmation fonctionnelle et orientée objet. En effet, si le premier style est idéal pour l'abstraction, la vérification et, grâce aux types algébriques et au filtrage de motifs pour la manipulation de données structurées (XML) ; rien ne semble pouvoir égaler le second en ce qui concerne la réutilisation de code. La base théorique de ce langage de programmation provient d'un calcul également défini au Lamp : νObj. Ce calcul est particulier du fait qu'il utilise des *types dépendant* : les membres des objets peuvent être des types.

3.1 νObj

Je ne donne ici que l'essentiel de ce calcul, pour plus de précisions, veuillez consulter [OCRZ].

La base de l'unification objet / fonctionnel est la mise en correspondance suivante :

Objet	=	Module		
Type d'objet	=	Signature		
Classe	=	Méthode	=	Foncteur

(en Scala, il reste cependant une distinction syntaxique entre classes et méthodes)

Le tableau 3 et les règles de typage de la figure 4 donnent formellement les grandes lignes du calcul, en voici une explication (très) sommaire :

- un objet est constitué d'un ensemble de définitions, soit de termes, soit de types.
- les méthodes (définitions de termes) ne prennent pas de paramètres.
- une définition de type dans un objet correspond exactement à une définition de type dans un module ML.
- le type $p.\mathbf{type}$ est le singleton $\{p\}$
- les déclaration de types abstraits ($<:$) jouent le même rôle que les variables de type ML (ou encore les déclarations de types abstraits dans les arguments des foncteurs ML), à ceci près qu'elles permettent d'imposer une borne inférieure sur ces types.
- les deux réductions de base du calcul sont la *sélection* ($p.l$) et la composition par *mixin* ($\&_S$)

Syntaxe :

x, y, z	Nom		
l, m, n	Label de terme	L, M, N	Label de type
s, t, u ::=	Terme	S, T, U ::=	Type
x	Variable	$p.\mathbf{type}$	Singleton
$t.l$	Sélection	$T \bullet L$	Sélection de type
$\nu x \leftarrow t \ ; u$	Instanciation	$\{x \mid \overline{D}\}$	Enregistrement
$[x : S \mid \overline{d}]$	Template	$[x : S \mid \overline{D}]$	Classe
$t \ \&_S \ u$	Composition	$T \ \& \ U$	Type composé
d ::=	Définition	D ::=	Déclaration
$l = t$	terme	$l : T$	terme
$L \preceq T$	type	$L \preceq: T$	type
p ::=	Chemin	$\preceq:$ ::=	Liaison de type
$x \mid p.l$		$=$	alias
v ::=	Valeur	$\prec$	type frais
$x \mid [x : S \mid \overline{d}]$		$<:$	type abstrait
		$\preceq$ ::=	Liaison concrète
		$= \mid \prec$	

Équivalence structurelle : α-renommage des variables x liées, ainsi que

(extrude) $\quad e[[\nu x \leftarrow t \ ; u]] \equiv \nu x \leftarrow t \ ; e[[u]]$
$\quad$ **si** $x \notin fn(e), bn(e) \cap fn(x, t) = \emptyset$

Réduction :

(select) $\quad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \ ; e[[x.l]] \hookrightarrow \nu x \leftarrow [x : S \mid \overline{d}, l = v] \ ; e[[v]]$
$\quad$ **si** $bn(e) \cap fn(x, v) = \emptyset$

(mixin) $\quad [x : S_1 \mid \overline{d}_1] \ \&_S \ [x : S_2 \mid \overline{d}_2] \hookrightarrow [x : S \mid \overline{d}_1 \uplus \overline{d}_2]$

Contexte d'évaluation :

$e \ ::= \ [[]] \mid e.l \mid e \ \&_S \ t \mid t \ \&_S \ e \mid \nu x \leftarrow t \ ; e \mid \nu x \leftarrow e \ ; t \mid \nu x \leftarrow [x : S \mid \overline{d}, m = e] \ ; t$

FIG. 3 – Le calcul νObj

$$\text{(Var)}\quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad\qquad \frac{\Gamma \vdash t : T, \quad T \ni (l : U)}{\Gamma \vdash t.l : U}\quad \text{(Sel)}$$

$$\text{(VarPath)}\quad \frac{\Gamma \vdash x : R}{\Gamma \vdash x : x.\mathbf{type}} \qquad\qquad \frac{\Gamma \vdash t : p.\mathbf{type}, \quad t.l : R}{\Gamma \vdash t.l : p.l.\mathbf{type}}\quad \text{(SelPath)}$$

$$\text{(Sub)}\quad \frac{\Gamma \vdash t : T, \quad T \leq U}{\Gamma \vdash t : U}$$

$$\text{(\&)}\quad \frac{\Gamma \vdash t_i : [x : S_i \mid \overline{D}_i] \qquad \Gamma \vdash S \text{ wf}, \quad S \leq S_i \qquad (i = 1, 2)}{\Gamma \vdash t_1 \,\&_S\, t_2 : [x : S \mid \overline{D}_1 \uplus \overline{D}_2]}$$

$$\text{(Class)}\quad \frac{\begin{array}{c}\Gamma \vdash S \text{ wf} \qquad \Gamma, x : S \vdash \overline{D} \text{ wf}, \quad t_i : T_i \\ t_i \textit{ contractive in } x \qquad (i \in 1..n)\end{array}}{\Gamma \vdash [x : S \mid \overline{D}, l_i = t_i{}^{i \in 1..n}] : [x : S \mid \overline{D}, l_i : T_i{}^{i \in 1..n}]}$$

$$\text{(New)}\quad \frac{\begin{array}{l}\Gamma \vdash t : [x : S \mid \overline{D}], \quad S \prec \{x \mid \overline{D}\} \\ \Gamma, x : S \vdash u : U \qquad x \notin \text{fn}(U)\end{array}}{\Gamma \vdash (\nu x \leftarrow t \;;\; u) : U}$$

FIG. 4 – Système de type de νObj

Le fait le plus surprenant lorsque l'on sait que νObj se veut être la base théorique d'un langage fonctionnel, est que ce calcul ne contient ni fonctions, ni types paramétrés, ni polymorphisme !

Que l'on se rassure, System $F_{<:}$ peut être encodé dans νObj à l'aide des transformations *context-free* suivantes :

– types :

$$\begin{array}{lcl}
\langle\!\langle \forall X <: S.T \rangle\!\rangle & = & \{\mathit{val} : [\ X : \{\mathit{Arg} <: \langle\!\langle S \rangle\!\rangle\} \\
& & \qquad \mid \mathit{fun} : [\mathit{res} : \langle\!\langle T \rangle\!\rangle]\ ]\} \\
\langle\!\langle T \to U \rangle\!\rangle & = & \{\mathit{val} : [\ x : \{\mathit{arg} : \langle\!\langle T \rangle\!\rangle\} \\
& & \mid \mathit{fun} : [\mathit{res} : \langle\!\langle U \rangle\!\rangle]\ ]\} \quad (x \text{ frais}) \\
\langle\!\langle X \rangle\!\rangle & = & X.\mathit{Arg} \\
\langle\!\langle \top \rangle\!\rangle & = & \{\,\}
\end{array}$$

– termes :

$$\begin{array}{lcl}
\langle\!\langle \lambda x : T.t \rangle\!\rangle & = & \text{new } [\ \mathit{val} = [\ x : \{\mathit{arg} : \langle\!\langle T \rangle\!\rangle\} \\
& & \qquad \mid \mathit{fun} = [\mathit{res} = \langle\!\langle t \rangle\!\rangle]\ ]\ ] \\
\langle\!\langle t\ u \rangle\!\rangle & = & \nu\, x \leftarrow \langle\!\langle t \rangle\!\rangle.\mathit{val}\ \&\ [\mathit{arg} = \langle\!\langle u \rangle\!\rangle]\ ; \\
& & \nu\, y \leftarrow x.\mathit{fun}\ ; \\
& & y.\mathit{res} \\
\langle\!\langle \Lambda X <: S.t \rangle\!\rangle & = & \text{new } [\ \mathit{val} = [\ X : \{\mathit{Arg} <: \langle\!\langle S \rangle\!\rangle\} \\
& & \qquad \mid \mathit{fun} = [\mathit{res} = \langle\!\langle t \rangle\!\rangle]\ ]\ ] \\
\langle\!\langle t[T] \rangle\!\rangle & = & \nu\, x \leftarrow \langle\!\langle t \rangle\!\rangle.\mathit{val}\ \&\ [\mathit{Arg} = \langle\!\langle T \rangle\!\rangle]\ ; \\
& & \nu\, y \leftarrow x.\mathit{fun}\ ; \\
& & y.\mathit{res} \\
\langle\!\langle x \rangle\!\rangle & = & x.\mathit{arg}
\end{array}$$

– environnements :

$$\begin{array}{lcl} \langle\!\langle x : T \rangle\!\rangle & = & x : \{\mathrm{arg} : \langle\!\langle T \rangle\!\rangle\} \\ \langle\!\langle X <: T \rangle\!\rangle & = & X : \{Arg <: \langle\!\langle T \rangle\!\rangle\} \\ \langle\!\langle \epsilon \rangle\!\rangle & = & \epsilon \\ \langle\!\langle \Gamma, \Sigma \rangle\!\rangle & = & \langle\!\langle \Gamma \rangle\!\rangle, \langle\!\langle \Sigma \rangle\!\rangle \end{array}$$

Les points essentiels sont bien sûr l'abstraction et l'application (de types comme de valeurs), que l'on simule avec une classe abstraite à laquelle il manque l'argument de la fonction, et une composition par mixin pour fournir cet argument.

3.2 SCALA

SCALA se veut, lui, autant fonctionnel qu'orienté objet ! Il étend donc νObj avec des méthodes paramétrées, des fonctions de premier ordre, les types algébriques, le polymorphisme...

Étant orienté objet, toute valeur est un objet, mais le caractère fonctionnel implique que toute fonction est une valeur, donc un objet. En fait une fonction est un objet possédant une méthode `apply`.

Le pendant objet d'un type algébrique étant une hiérarchie de classes c'est de cette manière qu'ils sont représentés, permettant ainsi aussi bien les types sommes ouverts (dynamiques) que fermés (statiques).

Afin de permettre aux librairie de définir des *sous-syntaxes*[6] adaptées à leur besoins, SCALA possède une syntaxe très souple, mais aussi très complexe (et instable...), je ne la détaillerai donc pas, l'exemple sur les listes présenté figure 5 devrait suffire à en donner l'essence.

On notera les points suivants :

- les classes ne possèdent pas de constructeur,
- il existe quatre types de définition :
  - **val** pour les champs, initialisés à la création d'une instance de la classe. L'ensemble de ces définitions définit donc grossièrement le constructeur de la classe.
  - **def** pour les méthodes, réévaluées à chaque appel (le mot clef **def** peut par ailleurs être utilisé pour définir des arguments de méthode *paresseux*).
  - **type** pour les types, abstraits ou alias.
  - **class** pour les classes, qui peuvent être imbriquées, tout comme les fonctions.
- une méthode n'est pas une fonction (ni un objet).
- l'inférence de types est locale.
- la valeur **null** existe :(
- le filtrage de motifs est étendu aux expressions régulières (ce qui simplifie énormément le traitement de données de type XML).
- les boucles (for, while...) sont, à ce jour, du sucre syntaxique autour des fonctionnelles `map`, `fold` et `iter`. Il en est de même des fonctions, transformées en classes anonymes dès l'analyse syntaxique.
- toute classe JAVA peut être instanciée ou héritée.

[6] voir l'appel de `map` trois lignes avant la fin de la figure 5...

```
object Lyst {
  trait Lyst[+a] {
    def ::[b >: a](x: b): Lyst[b] =
      Lyst.::(x, this) ;
    def length: Int = match {
      case Nyl ⇒ 0 ;
      case _ :: q ⇒ q.length + 1 ;
    }
    def map[b](f: a⇒b): Lyst[b] = match {
      case Nyl ⇒ Nyl ;
      case x :: q ⇒ f(x) :: q.map(f) ;
    }
    type t <: a ;
  }
  case object Nyl extends Lyst[All] { type t = All ; }
  case class ::[a, b >: a](x: b, q: Lyst[a]) extends Lyst[b] { type t = b ; }
  val p = 2 :: 3 :: Nyl ;
  val q = 5.1 :: p ;
  def main(args: Array[String]): Unit = {
    System.out.println(q) ;                        // "Lyst(5.1, 2, 3)"
    System.out.println(q map (x⇒x*x)) ;            // "Lyst(26.01, 4.0, 9.0)"
    System.out.println(3.3 as p.t) ;               // "3"
    System.out.println(3.3 as q.t) ;               // "3.3"
  }
}
```

FIG. 5 – Exemple de code SCALA

3.3 SOCOS : le compilateur

SOCOS COmpiles Scala. Environ 70 000 lignes de PICO[7], sans compter le code généré automatiquement pour les opérations de base sur la représentation intermédiaire...

Après les phases standard d'analyses lexicale, syntaxique et sémantique, la compilation consiste en une série de phases visant à transformer le code SCALA en code JAVA. Enfin, différents *backends* peuvent convertir les classes obtenues en *bytecode* `JVM`, `MSIL` ou `.NET`.

Voici une rapide description de ces transformations :
- **UnCurry** : toutes les méthodes sont décurryfiées.
- **TransMatch** : transformation des filtrages de motifs en code *standard*.
- **LambdaLift** : toutes les classes internes à une méthode sont déplacées à l'extérieur de cette méthode, en rajoutant des paramètres pour relier toute variable éventuellement *libérée*.
- **ExplicitOuterClasses** : ajout d'un paramètre vers la classe englobante de toute classe imbriquée, et transformation des accès aux champs et méthodes de la classe englobante via ce paramètre.
- **AddAccessors** : des accesseurs (méthodes *get* et/ou *set*) sont ajoutés

[7] JAVA + types algébrique + filtrage de motifs

pour chaque définition de valeur (**val**) et pour chaque paramètre de la classe.

- **AddInterfaces** : la composition par mixin nécessite la connaissance des interfaces des classes mises en jeu, cette phase remplace donc toute classe par une interface et une classe l'implémentant.
- **ExpandMixins** : expansion *textuelle* des mixins.
- **Erasure** : conversion des types Scala en types Java (effacement de tous les paramètres de type, $List[A]$ devient $List$).
- **AddConstructors** : ajout d'un constructeur pour chaque classe, initialisant tous les champs (**val**).

4 νObj-M, Scala-M

4.1 Extension du modèle théorique : νObj-M

L'extension multi-niveaux de νObj peut se faire de la même manière que dans le cadre du lambda calcul : on introduit l'opérateur $\langle . \rangle$, ainsi que deux *noms*, ˜ et run . Ces deux noms doivent être vu comme deux méthodes d'un objet de type $\langle 'a \rangle$[8]

$$\text{(Var)}\quad \frac{x : T^{n-m} \in \Gamma}{\Gamma \vdash^n x : T} \qquad\qquad \frac{\Gamma \vdash^n t : T,\quad T \ni (l : U)}{\Gamma \vdash^n t.l : U}\quad \text{(Sel)}$$

$$\text{(VarPath)}\quad \frac{\Gamma \vdash^n x : R}{\Gamma \vdash^n x : x.\mathbf{type}} \qquad\qquad \frac{\Gamma \vdash^n t : p.\mathbf{type},\quad t.l : R}{\Gamma \vdash^n t.l : p.l.\mathbf{type}}\quad \text{(SelPath)}$$

$$\text{(Sub)}\quad \frac{\Gamma \vdash^n t : T,\quad T \leq U}{\Gamma \vdash^n t : U}$$

$$\text{(\&)}\quad \frac{\Gamma \vdash^n t_i : [x : S_i |\ \overline{D}_i] \qquad \Gamma \vdash^n S \text{ wf},\quad S \leq S_i \qquad (i = 1, 2)}{\Gamma \vdash^n t_1 \&_S t_2 : [x : S \mid \overline{D}_1 \uplus \overline{D}_2]}$$

$$\text{(Class)}\quad \frac{\begin{array}{c}\Gamma \vdash^n S\ \mathit{wf} \qquad \Gamma, x : S^n \vdash^n \overline{D}\ \mathit{wf},\quad t_i : T_i \\ t_i\ \mathit{contractive\ in}\ x \qquad (i \in 1..n)\end{array}}{\Gamma \vdash^n [x : S |\ \overline{D}, l_i = t_i{}^{i \in 1..n}] : [x : S |\ \overline{D}, l_i : T_i{}^{i \in 1..n}]}$$

$$\text{(New)}\quad \frac{\begin{array}{l}\Gamma \vdash^n t : [x : S |\ \overline{D}],\quad S \prec \{x |\ \overline{D}\} \\ \Gamma, x : S^n \vdash^n u : U \qquad x \notin \mathrm{fn}(U)\end{array}}{\Gamma \vdash^n (\nu x \leftarrow t\ ;\ u) : U}$$

$$\text{(Code)}\quad \frac{\Gamma \vdash^{n+} t : T}{\Gamma \vdash^n \langle t \rangle : \langle T \rangle} \qquad\qquad \frac{\Gamma \vdash^n t : \langle T \rangle}{\Gamma \vdash^{n+} t.\tilde{} : T}\quad \text{(Spl)}$$

$$\text{(Run)}\quad \frac{\Gamma \vdash^n t : \langle T \rangle}{\Gamma \vdash^n t.run : T}$$

Fig. 6 – Système de type de νObj-M

L'extension du système de type est présentée figure 6, les méthodes run et ˜ ne doivent être appliquées que sur des objets de type code, et on doit toujours vérifier que les variables ne sont utilisées qu'aux niveaux supérieurs ou égaux à leur niveau de liaison. Attention cependant, on a ici deux types de liaison :

- liaison d'un objet à sa création, règle New,

[8]la syntaxe de νObj étant orientée objet, introduire ces deux opérations de manière *fonctionnelle* n'aurait pas été judicieux

– liaison de la référence vers soi-même (*this*), lors de la construction d'une classe, règle CLASS.

Étant donné que l'on ne donne l'accès qu'au type et au constructeur d'un *objet code*, sa classe est "cachée", et ne peut donc pas être héritée. En particulier les méthodes ˜ et run ne seront jamais surchargées (ce qui permettrait de passer outre le système de type).

Pour la sémantique du calcul, donnée figure 7, cela reste aussi similaire à λ-M : réductions de νObj (*select* et *mixin*) au niveau zéro, et reconstruction au niveau un.

Pour ce faire, on utilise une notion de contexte d'évaluation plus précise, afin de contrôler le niveau de l'expression à réduire : x apparaît forcément au niveau n dans $e^n[[x]]$.

Équivalence structurelle : α-renommage des variables x liées, ainsi que

$$\text{(extrude)} \quad e[[\nu x \leftarrow t\ ;\ u]] \ \equiv\ \nu x \leftarrow t\ ;\ e[[u]] \quad \textbf{si } x \notin fn(e), bn(e) \cap fn(x,t) = \emptyset$$

Réduction :

$$\text{(select)} \quad \nu x \leftarrow [x : S |\ \overline{d}, l = v]\ ;\ e[[x.l]] \ \hookrightarrow\ \nu x \leftarrow [x : S |\ \overline{d}, l = v]\ ;\ e[[v]] \quad \textbf{si } bn(e) \cap fn(x,v) = \emptyset, \ \textbf{et } l \notin \{run, \tilde{}\}$$

$$\text{(mixin)} \quad [x : S_1 |\ \overline{d}_1] \ \&_S\ [x : S_2 |\ \overline{d}_2] \ \hookrightarrow\ [x : S |\ \overline{d}_1 \uplus \overline{d}_2]$$

$$\text{(run)} \quad e[[\langle x \rangle.run]] \ \hookrightarrow\ e[[x]] \quad \textbf{si } \tilde{}\ \text{n'apparaît pas au niveau 0 dans } x$$

$$\text{(splice)} \quad e^1[[\langle x \rangle.\tilde{}]] \ \hookrightarrow\ e^1[[x]]$$

Contexte d'évaluation, au niveau n :

$$\begin{aligned} e^{n+} \ ::=\ & e^n.l\ (l \neq \tilde{})\ |\ e^n\ \&_S\ t\ |\ t\ \&_S\ e^n\ | \\ & \nu x \leftarrow t\ ;\ e^n\ |\ \nu x \leftarrow e^n\ ;\ t\ |\ \nu x \leftarrow [x : S\ |\ \overline{d}, m = e^n]\ ;\ t\ | \\ & \langle e^{n-} \rangle\ |\ e^{n+}.\tilde{} \\ e = e^0 \ ::=\ & [[]]\ |\ \langle e^{-1} \rangle\ |\ e^1.\tilde{}\ |\ e^0.l\ (l \neq \tilde{})\ |\ e^0\ \&_S\ t\ |\ t\ \&_S\ e^0\ | \\ & \nu x \leftarrow t\ ;\ e^0\ |\ \nu x \leftarrow e^0\ ;\ t\ |\ \nu x \leftarrow [x : S\ |\ \overline{d}, m = e^0]\ ;\ t \end{aligned}$$

FIG. 7 – Sémantique pour νObj-M

Bien entendu, run pose les mêmes problèmes qu'auparavant, puisqu'il abaisse le niveau d'une expression, lors de l'évaluation, sans contrôle depuis le système de type.

L'introduction des annotations supplémentaires comme pour λ^{BN} semble possible, bien que je n'ai pas eu le temps de m'y attaquer.

4.2 Extension de Scala : Scala-M

Après avoir étudié la théorie, penchons-nous sur la pratique...

4.2.1 Syntaxe, la classe `Code`

La représentation intermédiaire utilisée par le compilateur étant vraiment conséquente, y ajouter le moins de nœuds possible m'a semblé primordial. Au niveau de la syntaxe, j'ai donc simplement ajouté une nouvelle classe à la librairie standard Scala, comme suit :

```
final class Code[+A](src : A) {
    def run = src ;
    def splice = src ;
}
```

Fig. 8 – La classe `Code`

Cette classe permet d'une part de définir simultanément les trois annotations, sans avoir à modifier le parseur ni introduire de nouveau nœud ; mais elle fournit aussi un outil très utile : si le compilateur n'est pas modifié (ou que les modifications sont désactivées), tout programme multi-niveaux correct compile, et renvoie à l'exécution les "bons" résultats[9]. En effet, vu qu'il ne s'agit que d'annotations, cette implémentation *idiote* de la classe `Code` suffit à les ignorer. Il est donc facile de vérifier que le travail effectué dans le compilateur pour obtenir réellement une exécution multi-niveaux est correct.

Pour illustrer la syntaxe ainsi définie, voici la version Scala de la fonction puissance présentée en 2.2.1 :

```
import scala.meta._ ;
object power {
    def pow(n: Int): Code[Double⇒Double] =
        new Code(x ⇒
            (if (n==0) new Code(1)
             else new Code(x * (pow(n-1).splice)(x))
            ).splice
        ) ;
    val pow3 = pow(3).run ;
    def main(args: Array[String]): Unit =
        System.out.println(pow3(5.0)) ;
}
```

Fig. 9 – Fonction puissance en Scala-M

À l'intérieur du compilateur, on peut ensuite identifier les symboles correspondant au constructeur de la classe `Code`, ainsi qu'aux méthodes `run` et `splice`.

[9] tant que les effets de bords ne jouent pas...

4.2.2 Extension du système de type

Un autre gros avantage de cette manière d'introduire les annotations, se trouve au niveau du système de type. En effet, le système actuel s'occupe déjà de vérifier que `run` et `splice` sont toujours appelés sur du code, et que ce code a le bon type (c'est un type polymorphe). Il ne reste donc qu'à vérifier que les variables sont liées et utilisées correctement vis à vis des niveaux.

Cette vérification est un simple parcours de l'arbre, avec calcul du niveau courant, annotation des symboles définis, et vérification des symboles utilisés.

Bien que les définitions mutuellement récursives soient autorisées, et que l'ordre d'apparition des champs et méthodes d'une classe n'ait pas d'importance ; une seule passe est suffisante : le symbole à vérifier dans ces cas là n'est pas celui du champ ou de la méthode, mais celui de la classe courante (`this`), que l'on a forcément déjà annoté.

4.2.3 Première implémentation

Le lambda calcul n'est pas assez souple pour permettre la *réduction sous lambda abstraction*, SCALA si ! En effet, les champs **val** d'un objet sont évalués à sa création, et donc, avant toute utilisation de cet objet. L'idée de la première implémentation est donc de transformer les splices contenus par un objet code, en **val**s.

La fonction puissance donnée figure 9 sera alors transformée en :

```
def pow(n: Int) = new Code$0(n) ;
class Code$0(n : Int) {
    private val splice0 = if (n==0) new Code$1()
                          else new Code$2() ;
    def run = x => splice0.run ;
    class Code$1() { def run = 1 } ;
    class Code$2() {
        private val splice0 = pow(n-1) ;
        def run = x * (splice0.run)(x) ;
    }
}
```

FIG. 10 – Transformation des *splices* en **val**, 1

Ainsi, lors de l'appel `pow(2)`, la récursion est déroulée en une *liste d'objets* possédant une méthode `run`, l'appel de cette méthode sur l'objet en tête de la liste fournissant la fonction "puissance deux".

Cependant, le code précédent n'est pas correct : la variable `x` est libre dans la méthode `Code$2.run`. On doit donc calculer pour chaque splice, l'ensemble des variables potentiellement libres, et les donner en paramètres à la méthode `run` correspondante[10]. On obtient alors le code donné figure 11.

[10]en fait, il faut aussi définir un type pour de tels objets, afin qu'ils puissent être unifiés (**if** ($n == 0$) **new** Code$1() **else new** Code$2())

```
def pow(n: Int) = new Code$0(n);
class Code$0(n : Int) {
  private val splice0 = if (n==0) new Code$1()
                        else new Code$2();
  def run = x ⇒ splice0.run;
  class Code$1() { def run(x : Double) = 1 };
  class Code$2() {
    private val splice0 = pow(n-1);
    def run(x : Double) = x * (splice0.run)(x);
  }
}
```

FIG. 11 – Transformation des *splices* en **val**, 2

Mais ce n'est toujours pas correct : considérons la fonction `h` (figure 12), qui introduit *pour le plaisir* une variable libre à chaque pas de la récursion. Le problème est que l'appel récursif à `h` est fait sur une valeur de type `Code$1`, et non pas `Code[Int]`.

```
def h(n: Int, z: Code[Int]): Code[Int] =
  if (n==0) z
  else new Code((x ⇒ h(n-1, new Code(x+z.splice)).splice)(n));
```

devient avec la transformation de la figure 11

```
def h(n: Int, z: Code[Int]): Code[Int] =
  if (n==0) z else new Code$0(n, z);
class Code$0(n: Int, z: Code[Int]) {
  val splice0 = h(n-1, new Code$1(z));          // Erreur de typage
  def run = x ⇒ (splice0.run(x))(n);
  class Code$1(z: Code[Int]) {
    val splice0 = z;
    def run(x: Int) = x+splice0.run;
  }
}
```

FIG. 12 – Problème de la fonction `h`

En fait, il faudrait que la fonction `h` soit en quelque sorte typée
"**def** h[A ≪ Code[Int]](n: Int, z: A): A"
où la relation "≪" permettrait de rajouter des variables libres à un type `Code[.]`

L'idée est alors de définir un *environnement d'évaluation*, passé en paramètre de toutes les méthodes `run` :

```
def pow(n: Int) = new Code$0(n);
class Code$0(n : Int) {
    private val splice0 = if (n==0) new Code$1()
                          else new Code$2();
    def run(e: E) = x ⇒ splice0.run(e ⊕ x);
    class Code$1() { def run(e: E^{x: Double}) = 1 };
    class Code$2() {
        private val splice0 = pow(n-1);
        def run(e: E^{x: Double}) = e(x) * (splice0.run(e))(e(x));
    }
}
```

FIG. 13 – Transformation des *splices* en **val**, 3

Future work...

Cet environnement peut être une simple table de hachage (et quelques *cast*...), mais je pense qu'il est possible d'utiliser un objet SCALA, contenant les définitions nécessaires, et que l'on étend au fur et à mesure. L'avantage de cette seconde solution serait de maintenir un typage fort, certainement proche du typage de λ^{BN}. La classe `Code` prendrait un second paramètre de type, indiquant sous quel environnement on peut appeler `run`, et la fonction `h` pourrait être proprement typée :

```
final class Code[+A, -E](src : A) {
    def run(e : E) = src ;
    def splice = src ;
}

def h[E](n: Int, z: Code[Int, E]): Code[Int, E] ;
```

$$
\begin{array}{rcl}
e \oplus x & ::= & \textbf{new}\ e\ \{\ \textbf{def}\ x' = x\ \} \\
\mathrm{E}^{x:T} & ::= & \mathrm{E}\ \textbf{with}\ \{\ \textbf{def}\ x' : T\ \} \\
e(x) & ::= & e.x'
\end{array}
$$

FIG. 14 – Possibilité de typage *à la* λ^{BN}

L'extension d'environnement ($\oplus$) serait alors un *refinment*, et l'accès à une valeur ($e(.)$), une sélection de champ. De cette manière le type de l'environnement décrirait l'ensemble des valeurs qu'il contient, un peu à la manière d'un type enregistrement, *extensible*.

La figure 15 donne la fonction `h` telle qu'elle serait transformée.

```
def h[E](n: Int, z: Code[Int, E]): Code[Int, E] =
    if (n==0) z else new Code$0(n, z) ;
class Code$0[E](n: Int, z: Code[Int, E]) {
    val splice0: Code[Int, E with {val x': Int}] =
        h(n-1, new Code$1(z)) ;
    def run(e: E) = x ⇒ (splice0.run(new e {val x'=x}))(n) ;
    class Code$1[E](z: Code[Int, E]) {
        val splice0: Code[Int, E] = z ;
        def run(e: E with {val x': Int}) = e.x'+splice0.run(e) ;
    }
}
```

FIG. 15 – Problème de la fonction `h`

Il reste cependant encore deux problèmes que je n'ai pas encore pu/su résoudre :

- SCALA n'autorise pas l'extension d'objets par refinment (**new** e {**val** x'=x}), mais seulement les extensions de classe.
- Il faut pouvoir générer des noms de champ (x') frais : ici, la récursion ne fait que surcharger le champ, et fausse donc le résultat.

Critiques

En transformant le code à l'aide des possibilités fournies par les **val**s, on obtient donc une solution plus ou moins bien typée selon le modèle d'environnement utilisé.

Par contre, le code ainsi obtenu n'a pas forcément gagné en efficacité : c'est une grosse liste chaînée où l'on ne cesse d'étendre et d'utiliser des environnements ! Sans ces derniers, on pourrait encore espérer que la machine virtuelle puisse faire de l'*inlining* à l'exécution ; mais ils sont bel et bien nécessaires...

Cependant, cette solution reste intéressante car elle se contente des outils déjà présents dans SCALA et ne nécessite qu'une phase relativement simple dans le compilateur, afin de réaliser la transformation. De plus, une étude plus approfondie de la solution avec environnements typés devrait pouvoir conduire vers un système de type *complet*, éventuellement adaptable à toute autre implémentation plus efficace ; sans que le programmeur ait à utiliser des annotations trop contraignantes, puisqu'elles seraient en quelque sorte automatiquement insérées.

4.2.4 Implémentation efficace

Vue sous un œil moins théorique, la programmation multi-niveaux, c'est aussi de la génération de code à l'exécution. C'est dans cet esprit que j'ai écrit cette deuxième implémentation : les arbres correspondant aux portions du code source de niveau non nul sont *sérialisés*, afin d'être reconstruits, puis compilés à l'exécution.

Bien que la base théorique soit solide, inutile de dire qu'il s'agit là plus de bricolage et de *hacking* que de programmation propre et vérifiable...

On introduit les trois nouveaux symboles comme précédemment, mais dans une classe JAVA car une partie du travail sera fait directement dans cette classe[11] : ce sont les instances de cette classe qui contiendront les arbres, feront les substitutions nécessaires, et lanceront des compilations à l'exécution.

La phase `TransMeta`

La première étape consiste à transformer le code de manière à ce que les splices correspondant à une portion de code soient évalués avant la construction de ce dernier. Tout comme au 4.2.3, on les avait transformés en **val**s, on va ici les passer en argument au constructeur de l'objet code[12], comme illustré figure 16. Les *trous* ($\bullet_{(n,i)}$) sont des nouveaux nœuds introduits dans la représentation intermédiaire ; n indiquant le niveau du code auquel *appartient* le trou, et i son indice dans le tableau passé en argument du constructeur.

$$\begin{array}{ll} & \textbf{new}\ \mathrm{Code}(3 + \textbf{new}\ \mathrm{Code}(2+1).\mathrm{splice})\,; \\ \longrightarrow & \quad \textbf{new}\ \mathrm{Code}(3 + \bullet_{(0,0)},\ [\textbf{new}\ \mathrm{Code}(2+1)])\,; \end{array}$$

$$\begin{array}{ll} & \textbf{new}\ \mathrm{Code}(\textbf{new}\ \mathrm{Code}(3 + \mathrm{f(x)}.\mathrm{splice} + \mathrm{g(y)}.\mathrm{splice}.\mathrm{splice}))\,; \\ \longrightarrow & \quad \textbf{new}\ \mathrm{Code}(\textbf{new}\ \mathrm{Code}(3 + \bullet_{(1,0)} + \bullet_{(1,1)},\ [\mathrm{f(x)},\ \bullet_{(0,0)}]),\ [\mathrm{g(y)}])\,; \end{array}$$

FIG. 16 – Déplacement des splices en paramètres du constructeur

Cette transformation est effectuée juste avant la phase `LambdaLift`, tant que la structure de l'arbre correspond toujours vraiment au code source. En effet, cette phase déplace certaines définitions de classes, et pourrait donc *endommager* les niveaux...

Cependant, le travail de `LambdaLift` est très utile, puisqu'il permet d'isoler les variables libres d'une classe, et de les lui passer en paramètres. Afin de pouvoir en profiter, on empaquette donc le corps de chaque objet code dans une nouvelle classe. Ainsi, toutes les variables libres apparaîtrons en tant que paramètres de cette classe, et les sous-classes éventuellement déplacées resterons dans le corps de cette classe, conservant ainsi correctement les niveaux.

La transformation complète, notée $\widetilde{\quad}^{n}$, est décrite relativement formellement figure 17. Les contextes (e[[.]]) utilisés sont similaires à ceux utilisés pour la sémantique de νObj-M, figure 7 : ils imposent que l'insertion soit effectuée au même niveau que l'ensemble.

[11] Afin de bénéficier du typage SCALA, on annote le code JAVA avec des commentaires spéciaux, pour que le compilateur puisse récupérer les types forts

[12] pour des raisons techniques, on ne peut pas définir plusieurs constructeurs (utilisateur / interne). En pratique, on invalide le premier, et on définit une méthode d'initialisation pour

$$f[[\textbf{new}\ \mathrm{Code}(e[[Spl_i.\mathrm{splice}]])]]$$
$$\overset{\sim}{\longrightarrow}{}^{n}$$
$$f[[\textbf{new}\ \mathrm{Code}(\{\ \textbf{class}\ \mathrm{wrp}()\ \{\ \textbf{def}\ \mathrm{src} = \widetilde{e}^{n+}[[\bullet_{(n,i)}]]\ \};\ \textbf{new}\ \mathrm{wrp}()\ \},\ \left[\widetilde{Spl_i}^{n}\right])]]$$

FIG. 17 – Transformation `TransMeta`

La phase `MetaSerialize`

Après la phase `LambdaLift`, les objets code ont la forme suivante :

$\textbf{new}\ \mathrm{Code}(\textbf{new}\ \mathrm{wrp}(x_j),\ [s_i])$;
...
$\textbf{class}\ \mathrm{wrp}(p_j : T_j)\ \{\ \textbf{def}\ \mathrm{src} = B\,;\overline{B}\ \}$;

où
- B est le corps de la portion de code,
- $\overline{B}$ est l'ensemble des définitions de classe déplacées,
- s_i sont les splices de la portion de code,
- p_j sont les variables de niveau inférieur ou égal au code courant, apparaissant libres dans $B \cup \overline{B}$,
- x_j sont les valeurs de ces variables dans le contexte global.

Pour tous les objets code de niveau zéro[13], on peut alors sérialiser le sous-arbre correspondant à la classe, retirer ce sous-arbre, et donner au constructeur la sérialisation et les valeurs x_j :

$\textbf{new}\ \mathrm{Code}(\ll\textbf{class}\ \mathrm{wrp}(p_j : T_j)\ \{\ \textbf{def}\ \mathrm{src} = B\,;\overline{B}\ \}\gg,\ [x_j],\ [s_i])$;

La classe `Code` ainsi instanciée a alors tout ce qu'il lui faut : l'arbre, les objets liés à cet arbre, et les portions de code manquant pour combler les trous (splices).

Il reste à savoir où placer cette phase...

Étant donné que la compilation de la partie sérialisée du code s'arrête à la phase MetaSerialize, c'est là qu'elle devrait reprendre, à l'exécution. L'idéal serait donc qu'elle vienne le plus tard possible, juste avant le *backend*, afin de minimiser le coût de la compilation à l'exécution.

Malheureusement, ce n'est pas possible actuellement, car le processus de sérialisation ne permet pas le stockage de la table des symboles, du fait de leur caractère unique. Or, c'est la phase d'analyse qui s'occupe de créer les symboles et de les attacher aux nœuds de l'arbre, les phases suivantes ayant l'interdiction formelle de manipuler directement les noms... Dans tous les cas, il faudra donc repasser par la case départ pour reconstruire la table des symboles.

De plus, les types utilisés à la compilation sont fortement dépendant des symboles, et on doit donc les *oublier* aussi. L'inférence de type étant seulement locale, il faut donc, avant de sérialiser, reconvertir certains types sous forme

le second.

[13] sérialiser le niveau zéro suffit : les objets code de niveaux non nuls et les trous seront tous sérialisés, car ils se trouvent forcément dans le corps d'un code de niveau zéro.

d'arbres (les types tels que le programmeur les écrit). Cette opération n'est pas possible dans le cas général, car certains types générés par l'analyse ne correspondent à aucun type *utilisateur* (types surchargés...). Cependant il s'agit typiquement des types que l'analyse sait inférer à partir du reste. Cette transformation est réalisée dans la fonction `mkTycon` de la phase `MetaSerialize`.

Or, le passage par la phase d'analyse est assez contraignant : une fois modifiés par `ExplicitOuterClasses`, certains sous-arbres ne sont plus *corrects* et sont rejetés. Cela est principalement dû au fait qu'après cette phase (qui rajoute un paramètre à chaque classe vers sa classe englobante) les classes sont complètement closes au niveau des termes, mais pas au niveau des types ; il est donc impossible de manipuler la classe sans son contexte. Or dès que l'on compile à l'exécution une classe sérialisée, c'est toujours hors de son contexte.

On doit donc placer la phase juste avant `ExplicitOuterClasses`, bien que cela interdise l'utilisation dans une portion de code, des méthodes définies dans les classes englobantes.

Cette disposition des phases pose aussi un autre problème : les méthodes curryfiées ne peuvent pas non plus être utilisées dans du code. En effet, les appels à ces méthodes sont décurryfiés lors de la première compilation, et ne sont donc plus valides lors de la seconde phase d'analyse.

Le tableau 1 récapitule l'ensemble des phases du compilateur, les phases concernant la programmation multi-niveaux étant marquées d'un M, et celles qui sont ignorées lors des compilations à l'exécution d'une croix.

×		`Parser`
		`Analyzer`
		`RefCheck`
×	M	*MetaCheck*
×		`UnCurry`
×		`TransMatch`
×	M	*TransMeta*
×		`LambdaLift`
	M	*MetaSerialize*
		`ExplicitOuterClasses`
		`AddAccessors`
		`AddInterfaces`
		`ExpandMixins`
		`Erasure`
		`AddConstructors`
		BACKEND

TAB. 1 – Phases du compilateur (multi-niveaux)

À l'exécution : la classe `Code`

Les transformations précédentes permettent la réduction des splices de premier niveau, avant la création des objets les englobant. Voyons désormais ce qui doit être fait à la création de ces objets.

Tout d'abord, le code est dé-sérialisé, puis le *contenu* des splices (s_i) est substitué aux trous de niveau nul ($\bullet_{(0,i)}$). Remarquons que les s_i, qui sont des objets code, ont déjà fait leur travail de dé-sérialisation/substitution, puisqu'ils ont été construits *avant* ; on peut donc parfaitement utiliser les arbres qu'ils contiennent. Les niveaux des autres trous sont aussi decrémentés. En effet, lorsque l'arbre sera compilé, tous les niveaux auront diminué d'un.

Attention toutefois, cette substitution implique aussi l'ajout des paramètres $s_i.p_j$ aux paramètres p_j de la classe (et l'ajout des valeurs $s_i.x_j$ aux valeurs x_j liées à la classe) : en introduisant un sous-arbre s_i, on introduit aussi toutes ses variables libres !

Cette extension de l'ensemble des variables libres doit bien entendu se faire sous α-renommage, afin d'éviter de malencontreuses captures...

Après avoir implémenté cette substitution, j'ai rapidement obtenu des portions de code comprenant des dizaines de paramètres pour ne passer en fait que deux valeurs distinctes. En effet, les variables libres dans une portion de code, le sont souvent aussi dans la suivante ; et à chaque substitution, on duplique son paramètre. Afin de limiter ce phénomène, on ne rajoute en pratique que les paramètres $s_i.p_j$ correspondant à une valeur $s_i.x_j$ n'étant pas présente dans les valeurs x_j déjà passées. Les autres paramètres sont alors renommés avant la substitution, de manière à ce qu'ils référencent la même valeur.

Le code ainsi préparé peut alors être compilé, dès que le programmeur appelle sa méthode `run`. Cette méthode[14] consiste à :

- invoquer le compilateur sur une unité contenant l'arbre, dans un mode spécial qui saute les phases marquées d'une croix sur le tableau 1.
- récupérer la classe ainsi construite (*Class.forName(String)*).
- récupérer son constructeur (*Class.getConstructors()*).
- vérifier et modifier éventuellement les types des valeurs x_j par rapport à la signature du constructeur, car il peut y avoir des problèmes d'*(un)boxing* ou de conversion entre les types primitifs Scala et Java.
- instancier la classe, avec les valeurs x_j (*Constructor.newInstance(Object[])*).
- récupérer la méthode `src` de l'objet (*Class.getMethod(String, Class[])*)
- appeler cette méthode sur l'objet Scala encapsulé. (*Method.invoke(Object, Object[]*).
- convertir éventuellement ce dernier (même problèmes que précédemment).

Cela semble tenir du miracle, mais ça fonctionne bel et bien !

Derniers détails sur cette classe :

- la méthode `toString()` est redéfinie pour qu'elle affiche l'arbre qu'elle contient, permettant ainsi au programmeur de vérifier ce qu'il se passe.
- un cache est introduit, afin de n'effectuer la compilation qu'au premier appel de `run`.
- on s'arrange pour que cette classe puisse aussi être utilisée comme la classe Scala présentée figure 8 : cela permet de compiler et d'exécuter du code multi-niveaux de manière *sûre*, en ignorant les phases `TransMeta` et `MetaSerialize` lors de la première compilation.

[14] ou plus exactement ce fouillis de hacks Java...

5 Conclusions

Du côté théorique, le travail de Walid Taha s'étend très bien au cas du langage objet νObj. On retrouve les même difficultés (typage de `run`) et l'étude d'une première implémentation du système semble pouvoir apporter une solution plus légère que les annotations supplémentaires de Walid Taha. Il pourrait être intéressant, au passage, de regarder si une telle solution serait utilisable dans le cadre du lambda-calcul.

De plus, on a pu définir une transformation relativement simple, permettant d'obtenir la sémantique désirée, tout en restant dans SCALA, chose qui ne semble aussi simple dans le cadre du lambda-calcul (ou plus généralement, des systèmes ML).

En pratique cette transformation n'est cependant pas satisfaisante, car pas spécialement efficace. La seconde implémentation, efficace cette fois, manque malheureusement actuellement de stabilité : figer la compilation à un stade donné, puis la reprendre à l'exécution n'est pas aussi simple qu'il pourrait sembler.

Actuellement l'équipe travaille sur la compilation séparée, et pourrait finir par trouver un moyen de stocker les arbres entièrement. On devrait alors pouvoir facilement déplacer `MetaSerialize`, et obtenir ainsi quelque chose non seulement plus robuste mais aussi d'encore plus efficace, les compilations à l'exécution étant alors réduites au strict minimum.

Remerciements

Je tiens à remercier toute l'équipe, qui m'a très chaleureusement accueilli : Michel Schinz, Martin Odersky, Stéphane Micheloud, Philippe Altherr, Matthias Zenger, Nikolay Mihaylov, Burak Emir, Sebastian Maneth, Johannes Börgstrom, Rachele Fuzzati, Sébastien Briais, Daniel Bünzli, Erik Stenman, Vincent Cremet, Simon Kramer, Uwe Nestmann, Yvette Dubuis...

Et plus particulièrement, Michel pour sa patience, ses bouquins et son aide précieuse pour la compréhension du compilateur, Martin pour toutes ses réponses à mes questions de typage, Stéphane pour la climatisation, Burak pour la musique, Sebastian pour les *whitecaps*, Johannes, Paolo et Lorenzo pour la jongle et les barbecues, et Yvette pour le logement.

Références

[Ber] Peter Bertelsen. Binding-time analysis for a jvm core language.

[Blu] Matthias Blume. Refining hygienic macros for modules and separate compilation.

[BMTS99] Z. Benaissa, E. Moggi, W. Taha, and T. Sheard. Logical modalities and multi-stage programming, 1999.

[BW94] L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 61–71, 1994.

[Cis] Antonio Cisternino. Multi-stage and meta-programming support in stringly typed execution engines.

[Cli91] William Clinger. Macros in scheme. *Lisp Pointers*, IV(4) :25–28, - 1991.

[CTHL] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. A bytecode-compiled, type-safe, multi-stage language.

[FM] GianLuigi Ferrari and Eugenio Moggi. Metaklaim : A type safe multi-stage language for global computing.

[FMP01] Gianluigi Ferrari, Eugenio Moggi, and Rosario Pugliese. MetaKlaim : Meta-programming for global computing. *Lecture Notes in Computer Science*, 2196 :183– ? ?, 2001.

[GJ96] Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, volume 1181, pages 261–272. Springer-Verlag, 1996.

[OCRZ] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types.

[Ode03a] Martin Odersky. Report on the programming language scala, 2003.

[Ode03b] Martin Odersky. Scala by examples, 2003.

[OZ01] Martin Odersky and Christoph Zenger. Nested types. In *Workshop on Foundations of Object-Oriented Languages (FOOL 8)*, 2001.

[PS94] Jens Palsberg and Michael I. Schwartzbach. Binding-time analysis : abstract interpretation versus type inference. In *ICCL'94*, pages 289–298, 1994.

[PTS02] E. Pasalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages, 2002.

[Rtc] Survey January Rtcg. Run-time code generation and ml.

[SD] Tim Sheard and Iavor S. Diatchki. Staging algebraic datatypes.

[She] T. Sheard. Using metaml : a multi-stage programming language.

[TBS98] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming : Axiomatization and type safety. *Lecture Notes in Computer Science*, 1443 :918– ? ?, 1998.

[TS] W. Taha and T. Sheard. Metaml : Multi-stage programming with explicit annotations.

[TS97] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York : ACM, 1997.

[WLPD98] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es) : ? ?– ? ?, 1998.

[Yan] Zhe Yang. A survey on multi-stage programming languages.