Intégration de SMALLTALK dans le projet AGLAE.

Jean-Pierre BRIOT (*, **)

Emmanuel St JAMES (*)

LITP (*)
5 place Jussieu 75005 PARIS

IRCAM (**)
31 rue Saint-Merri 75004 PARIS

Résumé:

Ce papier présente le projet AGLAE, visant à proposer un ensemble de langages applicatifs (LISP, LOGO, SMALLTALK ainsi qu'un système de calcul formel) autour d'un noyau commun, la machine META. La machine virtuelle META et l'évaluateur L seront décrits ici, puis le modèle proposé par l'interprète SMALLTALK implanté sur META.

Mots-clés:

AGLAE, SMALLTALK, META, L, méta-récursivité, lecteur, compilation, objets, classes, métaclasses, sélecteurs, méthodes, hiérarchie, ...

1. INTRODUCTION

Le projet **AGLAE** (Atelier de Génie Logiciel Applicatif pour l'Enseignement) est un projet de recherche et d'enseignement dans le domaine des langages applicatifs. Il s'agit d'un travail mené au sein du LITP (Laboratoire Informatique Théorique et Programmation) à l'université Paris VI, et commandé par le CNDP (Centre National de Documentation Pédagogique) afin d'équiper de manière homogène toutes les écoles, collèges, lycées et universités en langages de programmation de ce type.

Les langages de programmation en question sont **VLISP**, **LOGO**, **SMALLTALK** et un système de calcul formel baptisé **FORMULE**, les machines à équiper sont pourvues soit d'INTEL 8086, soit de Zilog 80, soit de Motorola 6809; d'autres machines seront toutefois concernées par la suite.
Cette machine ainsi que l'interprète SMALLTALK ou plutôt son prototype vont être présentés succintement maintenant.

2. La machine META

L'idée centrale du projet est non seulement de concevoir une machine virtuelle munie d'assembleurs pour les machines réelles, mais aussi de réaliser un évaluateur commun à tous ces langages applicatifs, l'implantation des langages se réduisant alors à l'écriture du lecteur/compilateur et du scribe/décompilateur, lesquels seront bien sûr des programmes exécutés par l'évaluateur. De manière générale, pour chaque langage, la plus grande partie de celui-ci sera écrit en lui-même. On verra qu'en SMALLTALK, les classes résidentes sont spécifiées à partir d'elles mêmes.

2.1 L'évaluateur L

La machine virtuelle est d'une conception totalement originale puisque entièrement fondée sur la méta-circularité (ou mieux *meta-récursivité*, terme qui nous paraît plus précis que celui inventé par Reynolds[Reynolds72a]): l'évaluateur universel (baptisé **L**) est capable d'interpréter le texte même qui a servi à son assemblage. Cette contrainte permet

- de s'affranchir totalement de choix d'implémentation de trop bas niveau, rapidemment obsolètes confrontés à l'évolution des processeurs;
- du point de vue pédagogique, de fournir dans le manuel de référence le texte d'un interprète méta-récursif qui n'est pas une description sommaire et imprécise de l'interprète réel, mais exactement celui-ci.

Enfin l'écriture de l'interprète y est aisée: lorsqu'un langage ne fait pas la distinction entre programmes et données, il est évident que le processeur le plus adapté à son interprétation est le langage lui-même! Cette machine virtuelle se nomme **META**.

Le tableau suivant résume les différentes composantes du projet:

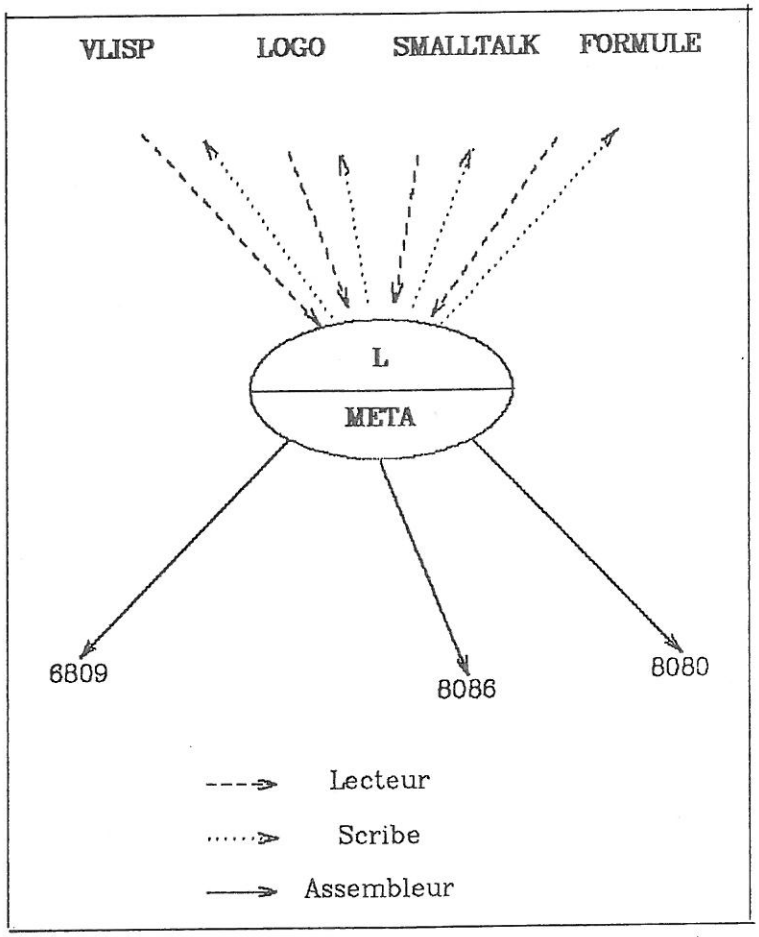

Figure 1. La machine META et son environnement

2.2 Rapports entre L et les langages de haut niveau

L'évaluateur "L" est un interprète d'un langage strictement applicatif, pour visualiser les choses on fera référence à Lisp pur.

Les problèmes déjà évoqués de passage de la représentation interne à la représentation externe et réciproquement sont purement syntaxiques et se résolvent assez facilement en utilisant "L" lui-même. Par la même occasion, l'utilisateur a ainsi accès à tout ce qui concerne la syntaxe et peut ainsi la modifier, rediriger les flux d'entrée ou de sortie, etc...
En revanche, tous ces langages admettent une composante non applicative qu'il convient de traiter spécialement.
Pour LISP, il s'agit de la séquentialité, permise par la fonction PROGN, dont l'abus par certains lui a finalement permis d'avoir le privilège d'être implicite dans presque toutes les constructions du langage.
Pour LOGO, le problème devient plus difficile car la syntaxe ne permet pas toujours de connaître l'arité des fonctions à leur seule position dans un texte. La solution est un séquenceur qui s'auto-modifie: un prototype a déjà été réalisé[Krief84a].

Pour SMALLTALK, il s'agit de la transmission de message que nous allons décrire maintenant.

3. PRESENTATION DE META SMALLTALK

Nous allons présenter succintement les caractéristiques de l'interprète SMALLTALK, en nous attachant surtout au modèle proposé et à ses caractéristiques par rapport aux autres interprètes SMALLTALK existants.

3.1 Introduction

Tout d'abord nous tenons à signaler que ce projet a grandement bénéficié des différents modèles d'interprètes SMALLTALK en LISP, à savoir les modèles SMALLTALK-72[Cointe81a], SMALLTALK-76[Cointe82a, Cointe83a] et Objvlisp[Cointe84a] de Pierre Cointe implémentés en VLISP[Chailloux78a, Greussay82a] et le Micro-Smalltalk d'Isabelle Borne[Borne84a] implémenté en Le_Lisp[Chailloux83a].

Le système META est un système de petite taille, surtout si on le compare au système SMALLTALK-80[Goldberg83a]. C'est l'occasion d'essayer de clarifier le langage en restreignant les diverses structures proposées, à la fois dans les domaines sémantiques et syntaxiques. Actuellement l'interprète est à l'état de prototype en cours de développement, dont le langage d'implémentation et le code produit est le langage Le_Lisp Vax[Chailloux84a].

3.2 Modèle

Comme on l'a vu précédemment, l'interprète repose sur un lecteur/compilateur et son réciproque le scribe/décompilateur. A titre indicatif voilà le code LISP produit à partir de la méthode *fac* définie pour la classe *number*.

```
number under| fac stand|
  [self = 0 then| 1 else| [self * [self - 1] fac]] ! `
```

produit la forme LISP suivante:

```
(lambda () (if (equal self 0) 1 (* self (react (- self 1) 'fac))))
```

La fonction *react* est chargée du traitement du message pour son receveur. Plus précisément de l'installation temporaire de l'environnement de la fermeture, de la recherche de la méthode dans la hiérarchie, et de son application fonctionnelle aux arguments, sans oublier la réécriture des champs dans la fermeture à sa sortie. Cette fonction prototype n'est que provisoire et sera remplacée ensuite par un traitement direct dans l'évaluateur par une clause spécifique.

3.3 Structures

3.3.1 Structures de données

L'interprète ne considère qu'une structure de données: les objets, certains d'entre eux, ne suivront pas le protocole normal de création, et seront instanciés à partir de la classe qui les type, et cela implicitement au niveau du lecteur. On parlera alors d'*objets primitifs*, qui ne seront pas créés explicitement par l'utilisateur. Ce sous-ensemble comprendra les variables, identificateurs permettant d'adresser les objets, et les objets de bas niveau (ou encore *rock-bottom objects*[1]) seuls à pouvoir s'auto-dénoter. Ces derniers comprendront les nombres (instances de la classe *number*), les chaînes de caractères (classe *string*) et enfin les listes (classe *list*), considérées comme seules données à la différence de LISP, et utilisées pour structurer les données du langage (par exemple pour spécifier la liste des champs d'une classe).
On remarque l'absence de la notion de bloc présentée dans SMALLTALK-80[Goldberg83a], par souci de simplification.

3.3.2 Structures de contrôle

L'unique structure de contrôle du langage est la transmission de messages. Nous introduisons la notion de *message dégénéré* pour alléger la syntaxe dans certains cas d'espèce.
Ainsi les messages de sélecteur implicite *value*, ou de receveur implicite (envoyé à un objet quelconque):

```
# a <- 3 !
> 3
# a !                          , valeur d'une variable
> 3
# end !                        , sans retour possible !
```

3.4 Syntaxe

Une transmission, i.e. un envoi de message à un objet, est encadrée par des crochets ouvrant et fermant, sauf au top-level où la transmission se termine par un ! (do-it!). Le premier élément sera le receveur du message. Les structures syntaxiques composées seront la séquence (*transmissions séparées par des* ".") et la cascade (*messages séparés par des* ";").
L'ordre de priorité est l'ordre habituel, mais ce sont les crochets et non les parenthèses comme en SMALLTALK-80 qui sont aptes à le modifier ou le confirmer en précisant explicitement les messages, les parenthèses n'ayant donc que le rôle décrit plus haut.

1. terminologie employée par Lieberman dans[Lieberman81a].

3.5 Une première session

Mais avant de continuer plus loin l'étude de l'interprète, laissons le s'auto-décrire:

```
# 1 + 2 !                                  , un peu d'arithmétique
> 3
# n <- 7; is !
> number
# l <- (1 ((2) 3)) + (4) !                 , passons aux listes
> (1 ((2) 3) 4)                            , une première application
# l nextl !                                , du concept de généricité
> 1
# l !                                      , message dégénéré:
> (((2) 3) 4)                              , sélecteur value implicite
# (+ 2 3) !                                , valeur d'un rock-bottom:
> (+ 2 3)                                  , lui-même
# (+ 2 3) eval !                           , appel explicite de l'évaluateur L
> 5
# "small" + "talk" !                       , place aux chaînes
> smalltalk
# fac load !
> fac.stk                                  , indispensable ...
# 3 fac !
> 6
# n foc !                                  , une faute de frappe,
** foc unknown method of number
> stop!
# n fac !                                  , vite rectifiée
> 5040
```

4. OBJETS ET CLASSES

4.1 Objets

Les objets du langage sont anonymes, ils peuvent être dénotés par des identificateurs. Les identificateurs ne sont pas typés et peuvent donc adresser n'importe quel objet, un pointeur étant l'unique moyen d'adresser un objet, excepté pour les objets primitifs qui, on l'a vu, peuvent s'auto-dénoter. Ce choix[2] permet d'éviter des recours à des *funcall* pour distinguer un objet d'un identificateur l'adressant[3] , ainsi qu'une uniformité de manipulation pour les classes autant que pour les instances terminales..

2. par opposition au choix d'implémentation des objets par des atomes (nommés) de Objvlisp par exemple.

3. problème, classique en LISP, d'atteindre une valeur fonctionnelle placée en CVAL.

Pour des raisons de confort de l'utilisateur, on donne néanmoins la possibilité à une classe d'être nommée, c'est à dire la possibilité de répondre au message *name* et *is* pour ses instances, par le message *name|* lui donnant la connaissance d'un identificateur qui l'adressera.

```
# foo <- class new !                    , création d'une classe anonyme
> *a_class*
# bar <- foo new !                      , et son instanciation produit
> *t_instance*                          , une instance terminale anonyme
# bar is !
> *anonymous*
# foo name| foo. bar print. bar is !    , plus d'anonymat
*a_foo*
> foo
```

La création de la classe *compteur* (comme celle de toute autre classe) n'introduit pas de syntaxe spécifique, elle repose en effet sur une cascade de messages à la classe créée par la transmission [*class new*].

```
class new name| compteur;
  fields| (unites);
  under| init stand| ["encore des taxes a payer!" print. self raz];
  under| raz stand| [unites <- 0];
  under| incr stand| [unites <- unites + 1];
  under| facture stand| [unites prin. " unites a payer" print. "avant 15 jours"] !
```

Voici maintenant un exemple de session en compagnie de cette classe:

```
# compteur load !
> compteur.stk
# edf <- compteur new !                    , menaces sur le budget
encore des taxes a payer!
> *a_compteur*
# compteur selectors !
 facture incr raz init
> done!
# 5 repeat| [edf incr]. edf facture !       , accelerendo
5 unites a payer
> avant 15 jours
# gdf pretty !
** gdf undefined variable
> stop!
# edf pretty !                             , structure et caractéristiques
I am a compteur
()
()
(unites)
(5)
> done!
```

4.2 **Classes**

L'interprète se propose ici de simplifier et clarifier le modèle des classes présent dans le système SMALLTALK-80. Dans la plupart des langages orientés objets basés sur le concept de classe[4] , on retrouve une dichotomie de l'univers entre deux catégories d'objets[5] :

- les générateurs,
  appelées *classes*, définissant un type de données et ayant le pouvoir de s'instancier

- et les *instances terminales*[6] ,

4. Les langages d'*acteurs* tels que ACT 1 [Cointe84a, Cointe84b, Lieberman81a, Theriault81a] et ACT 2 [Theriault83a], mais également le langage FORMES[Cointe84c] ne considèrent pas le concept de classe, l'instanciation a alors une sémantique complètement différente, celle d'une *copie différentielle*[Barthes53a, Cointe84d].

5. Remarquons qu'existe également un autre modèle où *tous* les objets sont des générateurs potentiels[Briot83a], les distinctions qui suivent n'ont alors plus de raison d'être, et une seule classe génératrice suffit à exprimer la sémantique de l'instanciation.

6. souvent appelées *instances* par abus de langage.

n'ayant pas ce pouvoir.

De fait la première catégorie se subdivise elle même en deux types:

- les *méta-classes*,
  générateurs d'objets non terminaux, i.e. des classes.

- et les *classes simples*;
  générant des instances terminales.

A ces deux types correspondent deux sémantiques différentes du message d'instanciation *new*.
Nous nommerons donc **metaclass** le générateur du premier des types, et **class** celui du deuxième. **class** est donc instance de **metaclass**, cette dernière étant son propre générateur. Nous verrons d'ailleurs par la suite que l'interprète s'auto-génère à partir de cette classe.

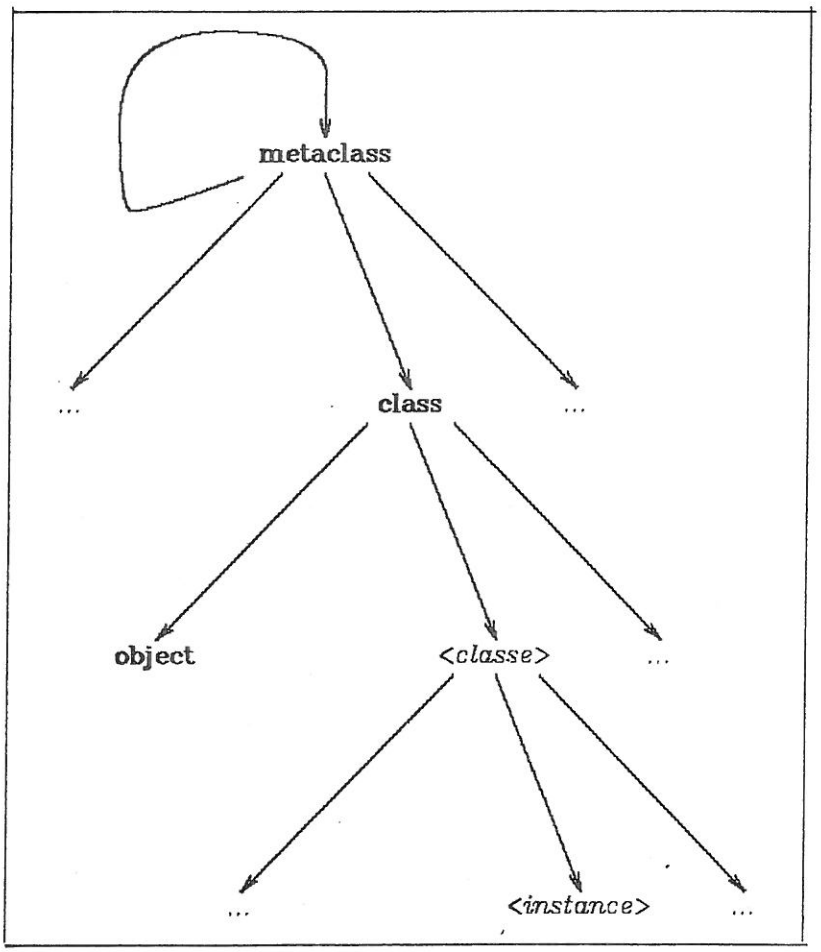

Figure 2. L'arbre d'instanciation

4.3 Hiérarchie

class est déclarée sous-classe de **metaclass**, une sémantique différente du sélecteur *new* les distinguant principalement.
La classe **object** est la racine de cette hiérarchie, elle incarne les comportements les plus généraux et est la sous-classe par défaut.

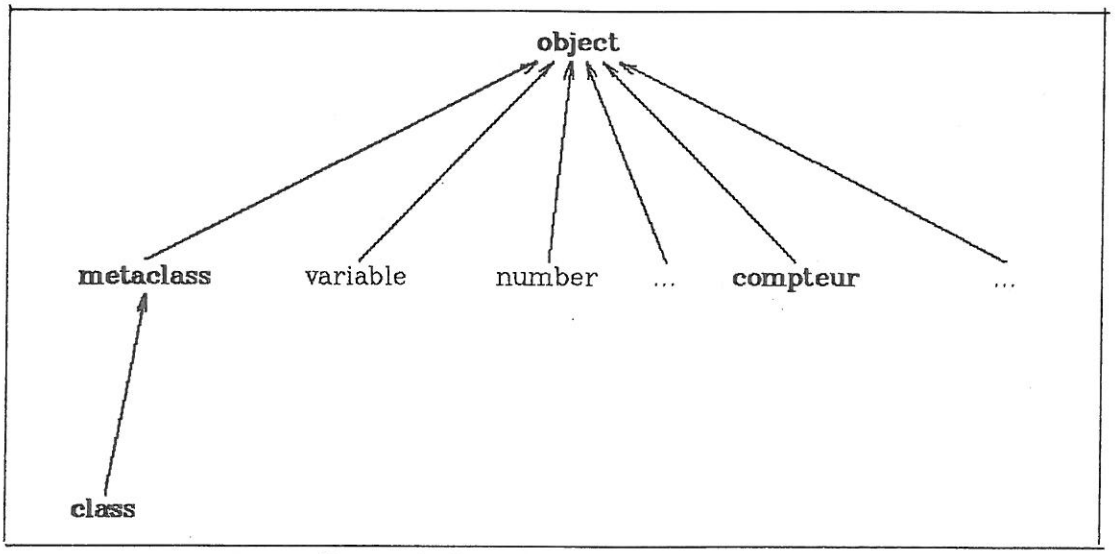

Figure 3. L'arbre hiérarchique

4.3.1 Héritage multiple

Dans ce premier prototype l'héritage est simple, mais deviendra plus tard multiple.
L'algorithme de recherche le plus simple définit un parcours de l'arbre[7] hiérarchique en profondeur et préfixe à gauche. Dans le cas d'occurences multiples d'un même sélecteur dans des branches distinctes[8] seule la première trouvée est ramenée, ce qui n'est pas suffisant pour traiter les problèmes de compositions de classes (voir à ce sujet[Borning79a]).
Nous ne concevons donc une réalisation de l'héritage multiple qu'à travers la définition d'un ensemble de moyens pour gérer cette exploration du graphe hiérarchique. Un système tel que les flavors de la machine Lisp[Moon80a], très sophistiqué mais volumineux et d'emploi parfois délicat est une réalisation de référence. Une telle réalisation n'est pourtant pas envisageable dans notre cas pour des raisons de taille et de simplicité d'emploi. La direction choisie est d'élaborer un ensemble minimal d'opérateurs simples permettant de poser des contraintes sur le parcours: contrainte de niveau (pseudo-objet *super*), d'élimination de chemins (tel que le propose LOOPS[Bobrow83a]), d'appels par sélecteur *absolu* (en spécifiant la classe)[Borning82a], ...

7. On suppose alors déjà que ce n'est pas un graphe quelconque.

8. ce que Cannon appelle la *non-orthogonalité*[Cannon7?a].