Vers une Sémantique Formelle de JAVA

Isabelle Attali, Denis Caromel* et Marjorie Russo
INRIA Sophia Antipolis - BP 93
06902 Sophia Antipolis - France
(33) 93 65 79 10
First.Last@sophia.inria.fr
*SLOOP - CNRS - Universite de Nice Sophia Antipolis - INRIA

Abstract: Les principales caractéristiques du langage Java sont l'orientation objets et le multi-threading. Cet article présente une sémantique formelle d'une partie de ce langage. Pour décrire la sémantique des concepts d'objets et d'héritage, nous avons utilisé une sémantique de type *big-step*. Cependant, pour la sémantique des aspects parallèles de Java, nous sommes passés à une sémantique *small-step* qui est plus appropriée pour décrire les actions concurrentes des threads et l'entrelacement. A partir de cette sémantique, nous générons automatiquement un interprète du langage.

1 Introduction

Le langage Java [5] rassemble deux grands concepts de la programmation moderne : la programmation à objets et la programmation dite "concurrente" ou parallèle. La caractéristique principale de l'approche objets est que toutes les entités intervenant dans l'univers modélisé sont des *objets*, constitués d'*attributs* qui définissent son état interne et de *méthodes* qui agissent sur ces attributs. La propriété d'*encapsulation* de l'état interne d'un objet assure que, pour un objet donné, seules les méthodes propres à cet objet peuvent accéder à ses attributs. Ainsi, les différents objets constituant un système interagissent entre eux uniquement par *invocation de méthode*, ce qui revient à dire que les objets ne communiquent que par *envoi de message*.
Cette approche permet d'autre part de rapprocher les systèmes parallèles classiques des systèmes répartis en encapsulant dans les deux cas la notion d'activité dans un objet. En effet, des objets contenus dans le même espace d'adressage ou distribués sur plusieurs machines communiquent tous par envoi de messages.
Notre objectif est de définir formellement la sémantique du langage Java, en utilisant le système Centaur [6] et le formalisme Typol [1]. Nous nous sommes pour cela inspirés du travail effectué sur Eiffel [3] et Eiffel// [2].

Dans cet article, nous présentons les principales unités syntaxiques de Java que nous avons retenues pour notre spécification sémantique. Nous détaillons alors les structures sémantiques choisies, l'organisation générale des modules de notre spécification ainsi que des règles Typol définissant l'affectation. Nous décrivons ensuite l'environnement d'interprétation que nous avons conçu. Nous illustrons finalement la sémantique et l'environnement avec des exemples d'interprétation de programmes Java.

Journées du GDR Programmation.
12, 13, 14 novembre 1997.
Rennes.

2 Structures syntaxiques

Notre spécification décrit actuellement les expressions :

- $A = B$ où B représente une expression autre que l'affectation et la création d'un objet
- $A\ Op\ B$ où Op est un opérateur binaire
- $A = new\ B\ (\ Args)$: expression de création d'un objet
- $A.B\ (\ Args)$: appel de la méthode B avec les arguments Args sur l'objet dont la référence est donnée par A (A pouvant être un autre appel de méthode)

Nous traitons aussi les instructions suivantes du langage :

- *return* A : instruction de retour d'une valeur à la fin de l'exécution d'une méthode
- *if* (A) B *else* C : test classique où A est une expression booléenne
- *while* (A) B : boucle de type "tant que A est vrai faire B"
- les blocks

3 Structures sémantiques

Notre sémantique doit tenir compte de la spécificité du langage Java : être un langage à objets. Cette section va détailler les choix que nous avons du effectuer au niveau des structures sémantiques de notre spécification ainsi que les grandes unités constituant celle-ci. Un dernier paragraphe présentera les principales règles Typol définissant l'affectation.

3.1 Structure des objets

La première structure sémantique importante doit donc être un objet. Nous pouvons cependant nous demander quels sont les éléments définissant un objet, c'est-à-dire de quelles branches sera formé l'arbre abstrait représentant un objet. Les informations devant figurer dans ce sous-arbre abstrait sont au minimum :

- un identificateur permettant de le reconnaître au sein des autres objets
- un identificateur de la classe dont il constitue une instance afin de connaître son type
- l'ensemble de ses attributs

L'identificateur d'un objet sera un entier qui servira de référence lorsque l'on souhaitera utiliser l'objet ou simplement l'indiquer. L'identificateur de la classe sera son nom.
Cette liste n'est pas spécifique au langage Java mais pourrait être reprise pour un autre langage à objets. Il faut de plus introduire des éléments propres au langage Java :

- un attribut "locked" qui indiquera, dans le cas où l'objet est verrouillé, par quel thread il l'est
- un ensemble "wait" de threads qui vont vouloir le reverrouiller à une certaine étape
- l'état courant de l'objet

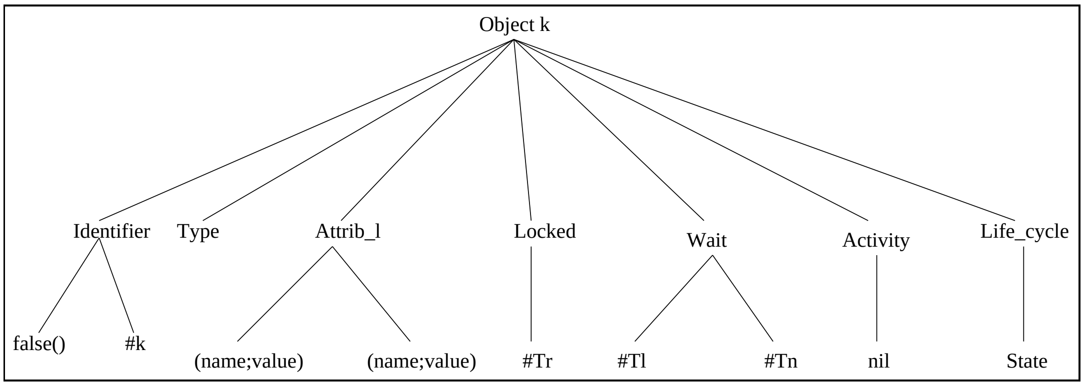

Figure 1: Structure d'un objet "simple".

Un objet "simple" (qui n'est pas un thread) aura donc la structure représentée sur la Figure 1. Pour représenter les différents objets créés au cours d'un programme, la structure sémantique la plus simple est une liste d'objets. En effet, à chaque pas d'exécution, l'état du programme est parfaitement décrit par la liste courante des objets.

Il faut aussi ne pas oublier l'existence des threads... Nous devons trouver une structure sémantique pour les représenter. Plusieurs possibilités existent. Notre première alternative a été : faut-il considérer des structures séparées pour les threads et les objets "simples" ou doit-on les réunir au sein d'une même structure ?

Dans l'optique des fondateurs de Java, une chose est très importante : *un thread est avant tout un objet*. Notre structure doit donc mettre ce fait en évidence. C'est pour cela que nous avons décidé de représenter les threads et les objets "simples" dans la même structure. Nous différencierons simplement ces deux types d'objet par leur identificateur. Celui-ci sera un nœud ayant deux fils : un fils sera un booléen prenant la valeur "true" dans le cas d'un thread et "false" sinon, le second fils étant le véritable identificateur de l'objet au sein de la liste. Sur le plan visuel, lorsque nous afficherons la liste des objets à l'écran, nous ferons seulement apparaître l'identificateur, en introduisant une différence textuelle : un thread aura un identificateur du type "#Tnum", alors que l'identificateur d'un objet "simple" sera du type "#num".

D'autre part, on peut se demander comment représenter les activités des threads au sein de cette structure sémantique ? Une nouvelle alternative se présente à nous :

1. On peut représenter la structure comme un arbre sémantique ayant deux fils : un premier fils représentant les activités des threads sous forme d'une liste de paires identificateur-activité et un second fils représentant la liste de tous les objets en considérant les threads comme des objets "simples".

2. On peut aussi garder la structure sémantique décrite dans le paragraphe précédent en ajoutant au niveau des objets un élément définissant les différentes activités.

Ces deux propositions ont l'avantage de montrer de manière claire que les threads sont avant tout des objets. Cependant, la première proposition possède un important défaut : la structure obtenue peut être totalement déséquilibrée. En effet considérons le cas d'un programme Java n'utilisant qu'un thread mais créant successivement un grand nombre d'objets simples ; l'arbre sémantique obtenu aura alors une première branche menant à un seul élément alors que la seconde branche mènera à une liste d'un grand nombre d'objets. Par contre, la deuxième structure a l'avantage d'être équilibrée. De plus si l'on considère la solution évidente concernant les activités, qui est de

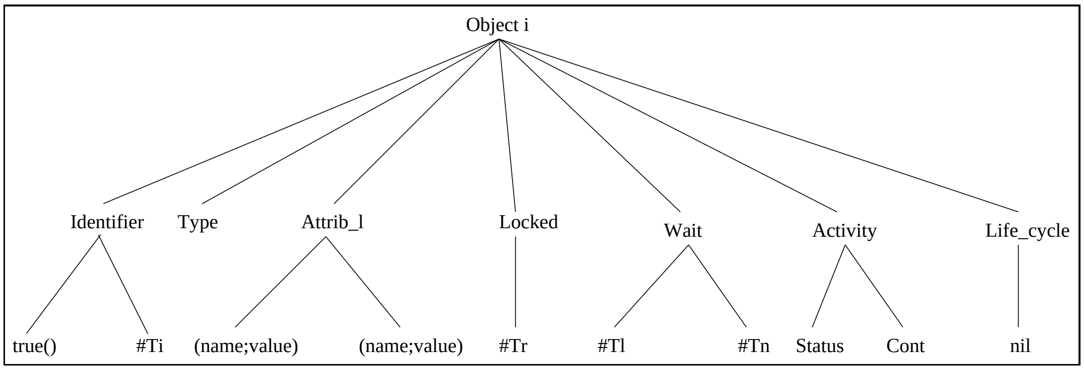

Figure 2: Structure d'un objet thread.

rajouter à chaque objet thread un constituant représentant son activité, la lecture de la structure sémantique sera simplifiée : elle ne nécessitera pas de se reporter à des références pour connaître les autres éléments définissant le thread considéré.
Il reste à définir précisément ce qu'est l'activité d'un thread. Cette activité doit rassembler d'une part l'état courant du thread qui sera utile au moment du scheduling et de l'entrelacement (un thread n'étant pas dans l'état "runnable" ne peut s'exécuter) et d'autre part les instructions à exécuter ainsi que l'environnement dans lequel elles doivent être exécutées. Ce dernier point (les instructions et leur environnement) constituera une closure qui définira l'activité future du thread considéré.
Chaque thread aura donc une closure représentant son activité future. A ce niveau se pose encore une question : doit-on avoir une liste de closures (une pour chaque appel de méthode) ou doit-on considérer que la closure constituant un appel de méthode est en fait une instruction comme une autre pour le thread considéré ? Nous avons choisi cette dernière solution qui semble être plus en accord avec l'optique selon laquelle un appel de méthode est une instruction simple du langage Java. Cela évite d'autre part d'avoir à ajouter des instructions pour la communication d'une closure à la suivante.
Par contre, il est nécessaire de garder une trace de l'objet sur lequel la méthode a été appelée. Pour cela, nous avons défini une closure comme étant rattachée à un objet : l'identificateur de l'objet sur lequel s'applique la closure sera placé comme premier argument de la closure. La structure sémantique que nous avons retenue pour les threads est représentée Figure 2.

3.2 Les principales unités de notre sémantique de Java

Nous avons noté que des principes forts apparaissent au sein du langage Java : *l'héritage*, la *gestion des activités concurrentes*, *l'exécution des instructions au niveau de chaque thread* et *l'évaluation des expressions*. Il est important que la sémantique que nous définissons conserve cette modularité. Elle doit de plus intégrer de nouveaux aspects : le *lancement de l'exécution* et la *gestion de la structure sémantique introduite* : la liste des objets. Ces différents aspects vont former autant de différents modules au sein de notre définition de la sémantique de Java. D'une part, cela permettra de conserver l'aspect modulaire du langage et d'autre part, au niveau de la programmation, cela introduira une plus grande lisibilité de notre sémantique.
Lors de notre spécification, nous avons suivi un principe important : *ne pas créer de structures intermédiaires*. Cela signifie que toute recherche d'attribut, de méthode ou de type s'effectue par un parcours de l'arbre de syntaxe abstraite représentant le programme et non au moyen d'une table

gardant en mémoire ces informations. Cela permet d'une part d'éviter de dupliquer du code et d'autre part d'avoir à effectuer une analyse préliminaire du programme avant son interprétation.

3.3 Règles Typol spécifiant l'affectation

Ce paragraphe va présenter les principales règles Typol permettant de spécifier la sémantique de l'affectation (dans les cas autres que la création d'un nouvel objet).
Voici tout d'abord la règle s'appliquant à l'exemple suivant : *A = 15.*

$$\frac{assign(Oi2, Name, Val, env, Objl, cvar \rightarrow Env1, Objl1, cvar1)}{I, Comp, Objl, Oi1, Oi2, cvar \vdash binary(Name, ass(), Val) \rightarrow Objl1, Env1, Val, Val, inst\_l[], cvar1}$$

Dans celle-ci, *Comp* représente l'arbre de syntaxe abstraite du programme, *Objl* la liste d'objets, *Oi1* l'identifiant du thread en train de s'exécuter et *Oi2* l'identifiant de l'objet sur lequel on exécute la méthode courante. Le prédicat *assign* réalise l'affectation proprement dite de la valeur *Val* à la variable *Name*.

Cependant, le plus souvent, l'instruction à exécuter n'est pas simplement l'affectation d'une valeur à une variable, mais l'affectation du résultat de l'évaluation d'une expression à une variable. Par exemple, l'exécution de l'instruction $A = B$. Dans ce cas, il faut utiliser la règle suivante :

$$\frac{eval(I, Comp, Objl, Oi1, oi2, cvar \vdash Exp, env \rightarrow Objl1, Env1, V, exp, cvar1)}{\begin{array}{l} I, Comp, Objl, Oi1, Oi2, cvar \vdash binary(Name, ass(), Exp) \\ \rightarrow Objl1, Env1, V, V, inst\_l[binary(Name, ass(), exp)], cvar1 \end{array}}$$

L'affectation $A = B$ est alors remplacée par l'affectation $A = B'$ où *B'* est le résultat d'une étape d'évaluation de *B*. L'instruction ainsi obtenue continuera à être exécutée lorsque le thread courant redeviendra *executing*. A cette étape-là, les deux règles seront toujours valables selon que *B'* sera une valeur ou une expression.
L'existence de ces deux règles permet de spécifier l'entrelacement au niveau de l'affectation.

4 Définition d'un environnement pour l'interprétation des programmes

Au-delà de la définition de la syntaxe et de la sémantique de java, il est important de disposer d'un environnement attractif et utile pour l'interprétation des programmes. Pour cela, en nous appuyant sur l'exemple d'Eiffel//[4], nous avons spécifié en Ppml la représentation des structures abstraites du langage et en Lisp l'environnement d'interprétation, c'est-à-dire par exemple la fenêtre que l'on souhaite créer pour faire figurer le résultat.

4.1 Présentation des structures sémantiques introduites précédemment

La représentation textuelle de la structure sémantique d'un thread et d'un objet simple donne le résultat présenté sur la Figure 3. On reconnait la similarité des structures des threads et des objets simples avec la différenciation au niveau des identificateurs *#T0* pour le thread principal (thread créé de manière transparente pour l'utilisateur et qui va exécuter la méthode "main") et *#1* pour l'objet 1 qui est un objet simple. Les parties fixes des structures comme les termes *Type* ou *Attributes* et les caractéristiques des objets sont présentées différemment par souci de lisibilité.

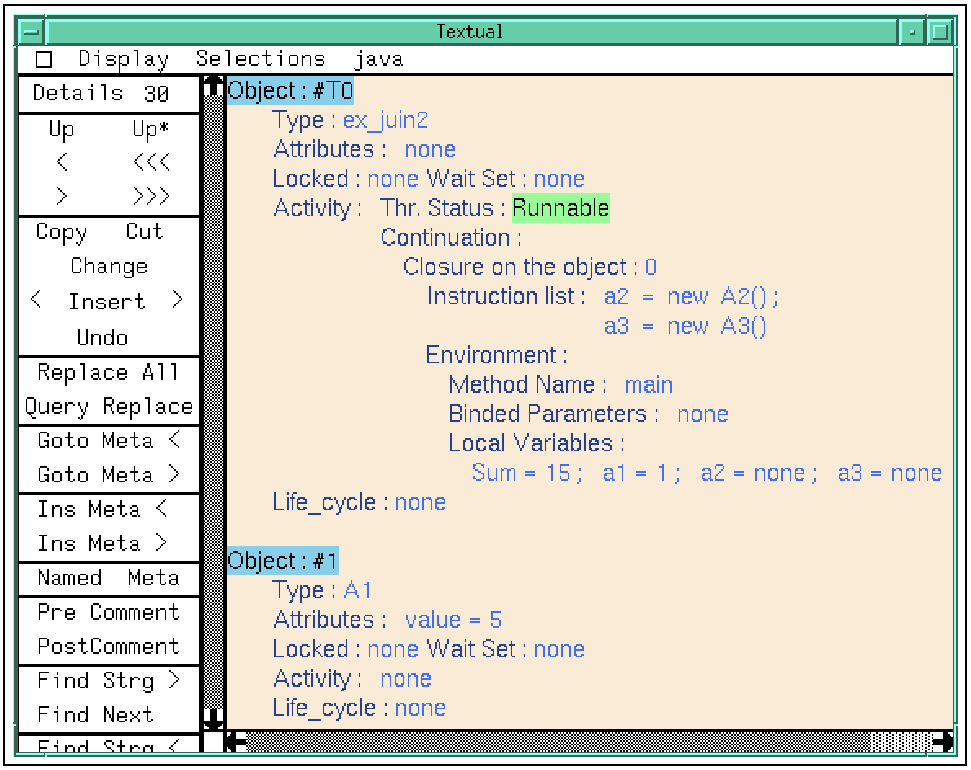

Figure 3: Représentation textuelle des objets.

4.2 Environnement de l'interprétation d'un programme

Afin de pouvoir étudier le déroulement d'un programme Java, nous avons été amenés à créer différentes fenêtres. A l'ouverture de l'éditeur Centaur contenant un programme Java, une fenêtre permettant d'établir la vitesse d'interprétation apparaît. Si l'on veut avoir le temps d'étudier ce qu'il se passe à chaque pas d'exécution, il faut cocher la case *slow* qui introduit une pause de deux secondes entre chaque mise à jour de la liste d'objets (ce mode est sélectionné par défaut).
Nous avons aussi ajouté à l'éditeur Centaur un menu Java qui est créé lorsque le fichier ouvert est un programme Java et qui permet de lancer l'interprétation du programme grâce au bouton **Interpret**. Des fenêtres *Debug-Options* et *Semantics* apparaissent quand les spécifications sémantiques ont été compilées en mode "debug". La fenêtre *Debug-Options* permet d'indiquer si l'on a fait figurer des temps d'arrêt dans les fichiers de définition sémantique pour que l'interpréteur en tienne compte ou non. La fenêtre *Semantics* permet de sélectionner le déroulement de l'interprétation : exécution automatique ou règle par règle.
La Figure 3 du paragraphe précédent apparaît quand l'interprétation est lancée. Elle indique l'état courant de la liste des objets. La mise à jour de cette fenêtre se fait de façon incrémentale : quand une règle Typol de la sémantique modifie la structure de cette liste, la nouvelle liste est communiquée à l'environnement qui effectue une comparaison entre la liste qu'il affichait jusqu'à présent et la nouvelle liste. Seules les parties modifiées sont mises à jour ce qui évite d'avoir à réafficher entièrement la liste d'objets.

5 Exemple simple d'interprétation d'un programme Java

Ce paragraphe présente un exemple simple d'utilisation de la sémantique que nous avons définie. Voici tout d'abord le programme Java que nous allons exécuter :

```
class  ex_gdr {
   public static void  main(String args[]) {
      A1 a1;
      A2 a2;
      a1 = new A1();
      a2 = new A2();
      a1.foo();
      a2.foo(); }}
class  A1 {
   int  value;
   A1() { value = 5; }
   void  foo() { value = value + 1; }}
class  A2  extends A1 {
   int  value2;
   A2() { value = value + 2; }
   void  foo() {
      if (value < 2)  value = value * 3;
      else  value = value + 10; }}
```

Ce programme réunit les concepts suivants :

- la création du thread principal qui va exécuter la méthode *main*
- la création d'objets simples *a1*, *a2*

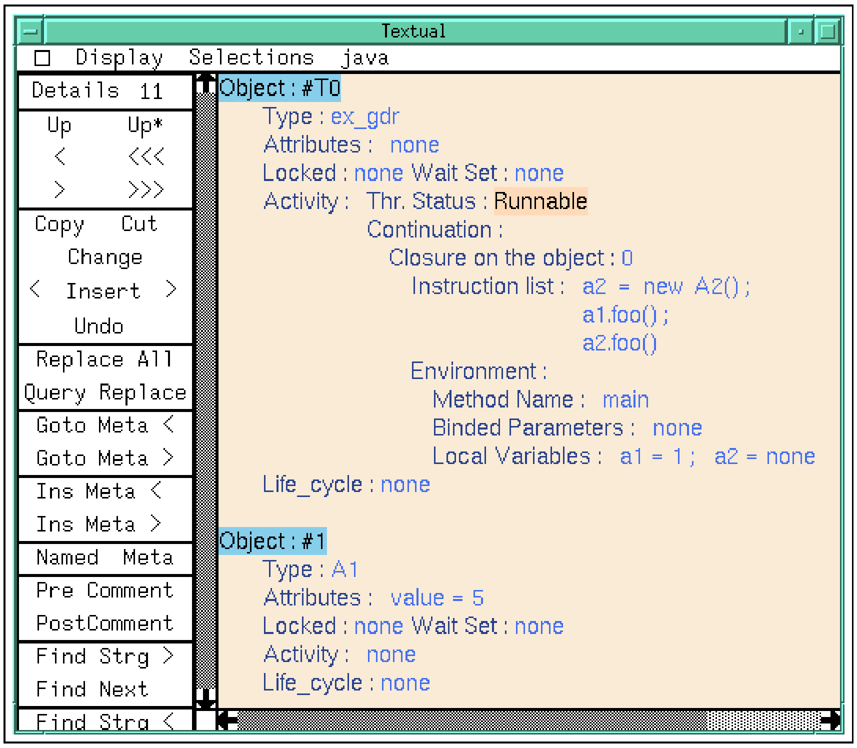

Figure 4: Représentation textuelle des objets après la création de a1.

- l'affectation *value = 5*
- l'appel de méthode comme *a1.foo()*
- l'héritage à travers l'héritage d'un attribut *value* pour a2, le masquage de la méthode *foo* pour l'objet a2
- le test de type *if ... then ... else ...*

Quand on lance l'interprétation du programme (comme cela est expliqué dans le paragraphe sur l'environnement de l'interprétation), la liste d'objets est vide. La première phase du programme est donc la création du thread principal. La première instruction que le thread principal doit exécuter est celle de création de l'objet *a1*. Cela entraîne une modification de la fenêtre avec l'apparition du nouvel objet ainsi que la disparition de cette instruction de l'activité du thread principal. On obtient donc la fenêtre présentée en Figure 4. L'appel de la méthode *foo* sur l'objet a2 (de même que sur l'objet a1) entraîne l'apparition de la closure associée à cette méthode au sein de la closure définissant l'activité du thread principal. C'est à ce niveau qu'il y a eu masquage de la méthode *foo* de la classe A1 par celle de la classe A2, car ces deux méthodes ont la même signature. La Figure 5 présente la fenêtre d'affichage obtenue à ce niveau de l'exécution restreinte au thread principal.

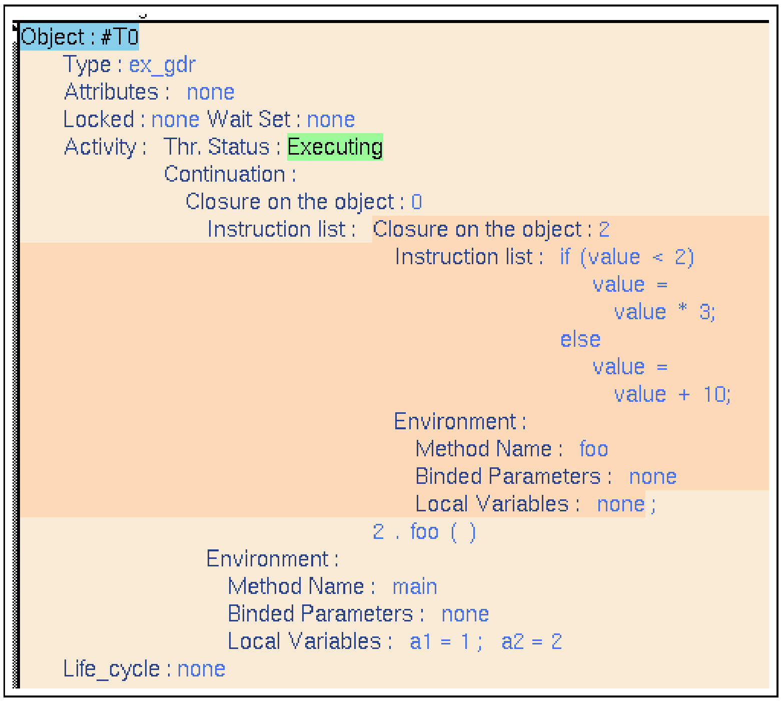

Figure 5: Visualisation du thread principal lors de l'appel d'une méthode.

6 Exemple de multithreading

Nous avons pu observer sur les différentes figures du paragraphe précédent que l'état du thread principal est soit *runnable* soit *executing*. Ces deux états sont nécessaires lors de l'exécution concurrente de plusieurs threads sur une machine mono-processeur : dans ce cas on introduit un entrelacement des exécutions pour simuler le parallélisme. Un thread pouvant s'exécuter est alors soit dans l'état *executing*, c'est-à-dire qu'il est en train de s'exécuter, soit dans l'état *runnable*, c'est-à-dire qu'il attend de pouvoir s'exécuter : un autre thread est dans l'état *executing*. Ce paragraphe va présenter un exemple de base du multithreading. Le programme d'exemple est le suivant :

```
class  simple_threads {
   public static void  main(String args[]) {
      TA1 a1;
      TA2 a2;
      a1 = new TA1();
      a2 = new TA2();
      a1.start();
      a2.start(); }}
class  TA1  extends Thread {
   int  value1;
   TA1() { value1 = 5; }
   void  run() { value1 = value1 + value1 * 15; }}
class  TA2  extends Thread {
   int  value2;
   TA2() { value2 = 100; }
   void  run() { value2 = value2 - value2 * 5; }}
```

Ce programme réunit les concepts suivants : création de deux threads par le thread principal (créé automatiquement au début de l'interprétation du programme) et exécution concurrente des trois threads ainsi obtenus.
Au début de l'exécution, seul le thread principal s'exécute. Puis il crée successivement deux threads et lance l'exécution du premier. A ce moment-là, le thread principal et le premier thread créé peuvent s'exécuter, mais un seul d'entre eux est réellement en train de s'exécuter, le thread *#T1* qui se trouve dans l'état *executing* (l'autre est dans l'état *runnable*) ; le troisième thread est juste dans l'état *created* comme cela apparaît sur la Figure 6. Dans la Figure 7, l'exécution du programme s'est poursuivie : le thread principal a terminé son activité, il est dans l'état *dead*, le deuxième thread attend de pouvoir poursuivre son activité, il est dans l'état *runnable* alors que le troisième est dans l'état *executing* : il s'exécute.
En comparant les instructions des activités des deux threads créés dans les Figures 6 et 7, on s'aperçoit qu'une instruction comme l'affectation est exécutée en plusieurs étapes, car elle n'est pas atomique (cf paragraphe 3.3).

7 Conclusion

Notre spécification permet pour l'instant de créer plusieurs threads, de les faire s'exécuter de façon concurrente sur les instructions simples du langage. Nous avons d'autre part créé une nouvelle structure sémantique permettant de représenter les variables de classe sous la forme d'une liste de paires *Nom de la classe - Liste de paires (nom de la variable ; valeur)*. Cela va nous permettre de spécifier les interactions qui existent au niveau des threads : bloquage en attente d'un verrou sur un objet, suspension d'un thread par un autre thread...
Cette spécification devra bien entendu respecter l'atomicité définie par les créateurs du langage. Nous allons aussi introduire un contrôle de la granularité de l'entrelacement.
A terme, notre objectif est, outre de disposer d'une sémantique formellement spécifiée, d'être capable de prouver certaines propriétés du langage ou de programmes Java.

```
Object : #T0
    Type : simple_threads
    Attributes :  none
    Locked : none Wait Set : none
    Activity :  Thr. Status : Runnable
                Continuation :
                  Closure on the object : 0
                    Instruction list :  a2.start()
                    Environment :
                      Method Name :  main
                      Binded Parameters :  none
                      Local Variables :  a1 = 1 ;  a2 = 2
    Life_cycle : none

Object : #T1
    Type : TA1
    Attributes :  value1 = 5
    Locked : none Wait Set : none
    Activity :  Thr. Status : Executing
                Continuation :
                  Closure on the object : 1
                    Instruction list :  value1 = value1 + value1 * 15
                    Environment :
                      Method Name :  run
                      Binded Parameters :  none
                      Local Variables :  none
    Life_cycle : none

Object : #T2
    Type : TA2
    Attributes :  value2 = 100
    Locked : none Wait Set : none
    Activity :  Thr. Status : Created by : #T0
                Continuation :
                  Closure on the object : 2
                    Instruction list :  value2 = value2 - value2 * 5
                    Environment :
                      Method Name :  run
                      Binded Parameters :  none
                      Local Variables :  none
    Life_cycle : none
```

Figure 6: Exécution concurrente de deux threads.

```
Object : #T0
    Type : simple_threads
    Attributes :   none
    Locked : none Wait Set : none
    Activity :   Thr. Status : Dead
                 Continuation :
                   Closure on the object : 0
                     Instruction list :   none
                     Environment :
                       Method Name :   main
                       Binded Parameters :   none
                       Local Variables :   a1 = 1 ;   a2 = 2
    Life_cycle : none

Object : #T1
    Type : TA1
    Attributes :   value1 = 5
    Locked : none Wait Set : none
    Activity :   Thr. Status : Runnable
                 Continuation :
                   Closure on the object : 1
                     Instruction list :   value1  =  80
                     Environment :
                       Method Name :   run
                       Binded Parameters :   none
                       Local Variables :   none
    Life_cycle : none

Object : #T2
    Type : TA2
    Attributes :   value2 = 100
    Locked : none Wait Set : none
    Activity :   Thr. Status : Executing
                 Continuation :
                   Closure on the object : 2
                     Instruction list :   value2  =  100  -  value2  *  5
                     Environment :
                       Method Name :   run
                       Binded Parameters :   none
                       Local Variables :   none
    Life_cycle : none
```

Figure 7: Fin de l'exécution d'un thread.

References

[1] T. Despeyroux. Typol: A Formalism to Implement Natural Semantics. Research Report 94, INRIA, 1988.

[2] I. Attali & D. Caromel & S. O. Ehmety. An Operational Semantics for the Eiffel// Language. research report 2732, I.N.R.I.A., 1995.

[3] I. Attali & D. Caromel & S. O. Ehmety. A Natural Semantics for Eiffel Dynamic Binding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(5), 1996.

[4] I. Attali & D. Caromel & S. O. Ehmety and S. Lippi. Semantic-based Visualization for Parallel Object-oriented Programming. In *Proceedings of Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'96), October 1996*, 1996.

[5] B. Joy J. Gosling and G. Steele. *The Java Language Specification.* Addison–Wesley, 1996.

[6] P. Borras & D. Clément & T. Despeyroux & J. Incerpi & G. Kahn & B. Lang and V. Pascual. Centaur : the System. In *SIGSOFT'88, Third Annual Symposium on Software Development Environments, Boston, Nov. 1988*, 1987.