Mémoire de D.E.A. :

***

Une sémantique formelle de JAVA

Marjorie RUSSO

26 juin 1997

Table des matières

Table des figures

Remerciements

En tout premier lieu, je tiens à remercier Isabelle ATTALI et Denis CAROMEL pour m'avoir prise en stage. Je souhaite aussi leur témoigner ma reconnaissance pour l'aide qu'ils m'ont apportée dans la réalisation de mon stage et la patience dont ils ont fait preuve. Leurs connaissances et leur soutien m'ont indiscutablement été d'un grand secours.
Je tiens ensuite à remercier Yves BERTOT pour m'avoir accueillie si chaleureusement au sein de son projet et pour le temps qu'il a passé à me présenter les diverses fonctionnalités de Centaur. Sa disponibilité, surtout lors du début de mon stage, et sa patience m'ont vraiment aidée à m'intégrer au sein de l'équipe de chercheurs du projet CROAP.
Je remercie également Sidi OULD-EHMETY pour sa patience et sa gentillesse, qui combinées à son expérience m'ont beaucoup aidée dans la compréhension du sujet de mon stage et dans sa réalisation.
Je remercie d'autre part Joelle DESPEYROUX dont la compagnie et le soutien moral ont été d'un grand réconfort dans les moments de doute.
Je tiens également à témoigner de ma reconnaissance envers Francis MONTAGNAC et Valerie PASCUAL qui m'ont beaucoup aidée lors de mes problèmes techniques liés à ma faible connaissance initiale de Centaur.
Je voudrais finalement remercier toute l'équipe du projet CROAP qui m'a accueillie très chaleureusement et a permis à mon stage de se dérouler dans une ambiance remarquable.

1 Introduction

Mon stage a débuté en avril. Il est d'une durée de cinq mois. Il se déroule à l'Institut National de Recherche en Informatique et Automatique de Sophia-Antipolis au sein du projet C.R.O.A.P. (Conception et Réalisation d'Outils d'Aide à la Programmation). Mon sujet de stage est **"Une sémantique formelle de Java"**.

1.1 Pourquoi étudier le langage Java?

Tout d'abord, on peut se demander pourquoi un sujet de stage spécifique au langage Java... En effet, que représente ce langage de programmation par rapport aux autres? Son grand intérêt est qu'il rassemble deux grands concepts de la programmation moderne : la programmation orientée objets et la programmation dite "concurrente" ou parallèle. Nous allons tout d'abord présenter ces deux concepts afin de voir ce qu'ils apportent à la programmation.

1.1.1 Les langages à objets

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. Ceci permet de séparer ce qu'est l'objet (l'ensemble de ses attributs) de la manière dont les autres objets interagissent avec lui (l'ensemble de ses méthodes).
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*.
On a donc isolé les deux concepts fondamentaux des objets : **l'encapsulation** et le **passage de message**. Un autre grand concept, très utile, l'**héritage**, vient les compléter. L'héritage permet de spécialiser un objet par rapport au modèle offert par sa classe en créant une *sous-classe* qui reprend le modèle de la *superclasse*, en modifie certains aspects ou en ajoute d'autres.

1.1.2 La programmation parallèle

Longtemps confinée au seul monde du calcul scientifique, l'idée de parallélisme a depuis gagné l'ensemble des champs d'application de l'informatique. Dans un même temps, le développement des réseaux de communication, tant en performances qu'en sureté de fonctionnement, a permis l'apparition de systèmes répartis à grande échelle.
Cependant, les techniques de programmation ont évolué beaucoup moins vite que les architectures des systèmes parallèles et répartis, si bien qu'aujourd'hui encore leur programmation est considérée comme difficile et délicate.
Depuis les serveurs Web qui traitent en parallèle les requêtes, jusqu'aux grands codes scientifiques tournant sur des machines massivement parallèles, en passant par les bases de données réparties des compagnies aériennes ou la simulation de trafic routier, la maîtrise de la mise en oeuvre d'activités parallèles est indispensable. Le problème de la programmation parallèle n'est pas tant de manipuler les concepts du parallélisme et de la répartition (flots d'exécution, objets distants, synchronisations) que de permettre une programmation sûre et agréable des activités concurrentes. En particulier, les programmeurs semblent souvent désorientés par le non-déterminisme inhérent à ces systèmes.

L'approche objet permet de rapprocher les systèmes parallèles classiques des systèmes répartis en encapsulant dans les deux cas la notion d'activité dans un objet, le passage de message marchant aussi bien pour des objets contenus dans le même espace d'adressage que pour des objets distribués sur plusieurs machines.
C'est pour cela que Java est aujourd'hui un langage prenant de plus en plus d'importance au niveau international. C'est à ce titre que son étude est intéressante.

1.2 Pourquoi définir formellement la sémantique de Java?

Donner une sémantique formelle à un langage de programmation permet de construire un modèle mathématique de ce langage. Son but est de servir de base à la compréhension et au raisonnement sur le comportement des programmes. Le modèle mathématique est très utile pour diverses opérations d'analyse et de vérification. De plus, sur un plan plus fondamental, il permet de définir précisément la sémantique des constructions d'un langage, ce qui peut révéler des subtilités importantes à connaître.
La sémantique formelle d'un langage permet d'effectuer des preuves sur ses propriétés. Par exemple, on peut vouloir donner une preuve de la correction du système de typage de Java. Cela est a été fait pour un sous-ensemble de Java [2]. Une telle preuve ne peut être envisagée sans une définition formelle de la sémantique du langage.
Etablir une définition formelle est primordial au sujet de Java à cause de son utilisation dans les systèmes distribués. Par exemple, il faut être certain que l'exécution d'un code récupéré par Internet ne peut causer de dommages à son propre système... Une composante importante de la recherche sur Java est donc la résolution des problèmes de sécurité.
Mon travail consiste donc en la spécification d'une sémantique formelle de Java à partir d'une description informelle [6].

Ce rapport va tout d'abord présenter les concepts et les principes du langage Java, car c'est à partir de cette étude qu'ont pu être effectués des choix au niveau de sa sémantique. Les principes de cette sémantique ainsi que les différentes possibilités que nous avons étudiées sont décrits dans une deuxième partie. Une troisième partie consiste en la définition formelle de Java. La dernière partie de ce rapport présente les spécifications qui seront effectuées durant les deux derniers mois de mon stage.

2 Etude des concepts et des principes du langage JAVA

Pour définir la sémantique formelle de Java, il a tout d'abord fallu étudier les différents concepts et principes de ce langage.

2.1 Présentation du langage Java

Java est en fait un véritable environnement de programmation orienté objets développé par Sun Microsystem. Il se compose de trois éléments :

- Un langage de programmation
- Une machine virtuelle (JVM) [9]
- Un ensemble de classes standard réparties dans différentes applications.

Dans la suite de mon rapport, Java représentera le langage de programmation, véritable but de mon étude. Java est un langage de programmation purement orienté objets.
Un programme Java est constitué d'une succession de déclarations de classes et d'interfaces.

2.1.1 Les classes

Les déclarations de classes constituent la base de tout programme Java.
Une classe définit un nouveau type d'objet et son implémentation. Elle se caractérise par :

- un nom
- des membres : attributs et méthodes
- des initialiseurs statiques
- des constructeurs

Toutes les classes sont des instances de la classe Object du package java.lang. Une classe A peut hériter d'une autre classe B grâce à la clause "extends B". De même, une classe A peut implémenter une interface I grâce à la clause "implements I".
On peut distinguer une classe particulière, la classe qui contient une méthode nommée "main". Cette méthode constitue ce qu'on appelle le programme principal en C ou en C++, ou la méthode "create" de la classe "root" en Eiffel. Il n'est pas nécessaire d'instancier cette classe particulière, le compilateur classique de Java sait la retrouver car le fichier dans lequel est écrit le programme doit avoir comme nom celui de la classe contenant la méthode "main". C'est cette méthode qui sera lancée lors de l'exécution du programme.
Java supporte le polymorphisme par héritage de classes (e.g. : si C est une classe et r une variable de type C, r peut référencer toute instance de C ou des classes dérivées de C) et par implémentation d'interface (e.g. : si I est une interface et r une variable de type I, r peut référencer toute instance de classe implémentant I). Les objets (qui sont tous alloués en mémoire dynamique) sont manipulés aux travers de variables, les références, qui sont les seuls points d'entrée sur la mémoire.

2.1.2 Les interfaces

Une déclaration d'interface introduit un nouveau type de référence dont les membres sont des méthodes constantes et abstraites. Le type ainsi défini n'a pas d'implémentation, mais des classes peuvent implémenter une interface en fournissant des implémentations à ses méthodes abstraites.
Les programmes Java peuvent utiliser les interfaces pour permettre aux classes apparentées à celles-ci de ne pas avoir à partager une superclasse abstraite commune ou à ajouter des méthodes à la classe "Object".
Une interface peut elle-même être une extension directe d'une ou plusieurs autres interfaces. Une classe peut implémenter directement une ou plusieurs interfaces ; c'est à ce niveau que l'on peut parler d'héritage multiple. Le terme correct est en fait héritage multiple d'interfaces. Cela permet aux objets de supporter plusieurs comportements communs sans avoir à partager une implémentation.
Quand une interface étend (par la clause "extends") une ou plusieurs autres interfaces, l'interface considérée hérite des méthodes et des constantes de chacune des interfaces nommées. Ces dernières sont alors appelées les "superinterfaces" directes de l'interface considérée. Une classe qui implémentera (par la clause "implements") cette interface, implémentera alors toutes ses superinterfaces.

2.1.3 Les types en Java

Java est un langage fortement typé, ce qui signifie que chaque variable ou expression a un type qui est connu au moment de la compilation. Les types du langage Java sont divisés en deux catégories :

- les types primitifs : ce sont les booléens et les types numériques (byte, short, int, long, char, float et double)
- les types par référence : ce sont les types des classes, les types des interfaces et les types des tableaux (et le type "null")

Les valeurs des types par référence sont des références à des objets.
Une variable est un emplacement de stockage. Une variable d'un type primitif contient toujours une valeur de ce type exact. Cette valeur ne peut être changée que par une opération d'assignation sur cette variable.
Les variables locales sont des variables déclarées au sein d'une méthode. Elles ne possèdent pas de valeur par défaut. Toutes les autres variables possèdent des valeurs par défaut : par exemple 0 pour un entier, 0.0 pour un réel, void pour une référence.

2.1.4 Le "security manager"

Un programme Java en train de s'exécuter peut disposer d'un "security manager", qui est une instance de la classe "SecurityManager". Les méthodes de cette classe du package java.lang ont souvent un nom débutant par "check". Elles sont invoquées par d'autres méthodes des librairies Java qui effectuent des opérations sensibles. Le "security manager" courant est retourné par l'invocation de la méthode "System.getSecurityManager();". L'invocation d'une méthode de vérification ressemblera généralement à :

```
SecurityManager security = System.getSecurityManager();
if(security!=null){
        security.checkXXXXXXX(arguments);
}
```

L'appel au "security manager" retourne simplement si l'opération souhaitée est autorisée et dans le cas contraire lance une exception.

L'attribut "inCheck" de la classe "SecurityManager" du package java.lang permet de savoir si un "security manager" est en route. Cet attribut est un booléen qui prend la valeur "true" quand un "security manager" est en train de tourner, et la valeur "false" sinon. La méthode "getInCheck" permet de tester la valeur de cet attribut.

2.1.5 Les exceptions

Une exception est un signal d'erreur lancé par la machine virtuelle de Java quand un programme Java viole une des contraintes sémantiques de Java. Quand une exception est lancée par la machine virtuelle, cela stoppe le flot d'exécution du thread ayant causé la violation et transfère le contrôle de ce point à un point qui a pu être spécifié par le programmeur. Ainsi il est possible d'effectuer un traitement d'une erreur si l'on avait prévu qu'elle puisse se produire.
Chaque exception est représentée par une instance de la classe "Throwable" ou d'une de ses sous-classes. Durant le processus de lancement d'une exception, une machine virtuelle termine de façon abrupte, une à une, chaque expression, commande, chaque invocation de méthode et de constructeur, les initialiseurs statiques et les expressions d'initialisation de champs qui ont commencé (mais sans avoir terminé) leur exécution dans le thread courant. Ce processus se poursuit jusqu'à ce que la machine virtuelle trouve une méthode écrite pour la gestion de cette exception. Si elle ne trouve pas une telle méthode, la méthode "uncaughtException" est invoquée par le groupe de threads parent du thread qui a lancé l'exception. Ce méchanisme est intégré dans le modèle de synchronisation de Java ce qui fait que les verrous sont relâchés en tant qu'instruction de synchronisation et que les méthodes "synchronized" terminent de façon abrupte.
Les exceptions sont vérifiées à la compilation et traitées à l'exécution.

2.1.6 Les packages

Les programmes Java sont organisés en ensembles de packages. Chaque package dispose de son propre ensemble de noms pour les types, ce qui aide à prévenir les conflits de noms. Un type est accessible en dehors du package qui l'a déclaré seulement si le type a été déclaré "public".
La structure des noms de packages est hiérarchique : des packages peuvent avoir des sous-packages. Un package peut être stocké dans un système de fichiers ou dans une base de données. Un programme Java dispose en général d'au moins trois packages : java.lang, java.util et java.io. Un package consiste en un certain nombre d'unités de compilation. Une unité de compilation a automatiquement accès à tous les types déclarés dans son package et importe automatiquement les types du package java.lang.
Un package peut avoir des membres appartenant à deux catégories :

- sous-packages d'un package
- types déclarés dans les unités de compilation du package

Si un package P compte un sous-package Q, le nom complet du sous-package sera : P.Q. C'est à ce niveau qu'apparaît la structure hiérarchique des packages. Les structures hiérarchiques des noms de packages permettent d'organiser les packages d'une manière conventionnelle, mais n'ont pas de signification au sein du langage Java lui-même.
Considérons le cas des packages stockés au sein d'un système de fichiers. Cette structure hiérarchique se retrouve physiquement au niveau des répertoires dans lesquels sont stockés les packages. Sous cette simple organisation de packages, une implémentation de Java pourra transformer le nom d'un package en un nom de "path", simplement en remplaçant (sous Unix) les points par des slash. Par exemple,

P.Q deviendra P/Q si Q est un sous-package de P, ce qui signifie qu'il est placé dans le sous répertoire Q du répertoire P. Dans le kit de développement Java de JAVASOFT sous Unix, cela est un peu différent : le kit fournit une variable d'environnement CLASSPATH qui spécifie un ensemble de répertoires. Ces répertoires sont alors cherchés dans l'ordre de définition des packages et des types nommés.

2.2 Exécution d'un programme Java

Le compilateur Java génère un byte code exécutable par la machine virtuelle Java (JVM). Grâce à ce mécanisme un programme Java est exécutable dans n'importe quel environnement disposant d'une JVM.
Voici les principales étapes intervenant lors de l'exécution d'un programme Java :

1. Lancement de la machine virtuelle
2. Chargement des classes et des interfaces
3. Liage des classes et des interfaces
4. Initialisation des classes et des interfaces
5. Création de nouvelles instances de classes
6. Finalisation
7. Déchargement des classes et des interfaces
8. Sortie de la machine virtuelle

Une machine virtuelle Java termine son activité lorsque l'un des deux évènements suivant se produit :

- tous les threads sont terminés
- un thread a invoqué la méthode "exit" de la classe "Runtime" ou de la classe "System" et l'opération "exit" a été autorisée par le "security manager"

2.3 Les threads

2.3.1 La création des threads

Il y a deux moyens de créer un thread :

- le premier est de déclarer une classe comme héritant de la classe "Thread". Dans ce cas, la classe créée doit avoir une méthode "run" pour surcharger celle de "Thread". Une instance d'une classe ainsi créée est un thread.
- l'autre moyen est de déclarer une classe implémentant l'interface "Runnable". Une surcharge de la méthode "run" doit aussi être décrite dans ce cas. Une instance d'une telle classe peut être utilisée pour créer un nouveau thread.

2.3.2 Les groupes de threads

Un groupe de threads est un ensemble de threads et de groupes de threads. Chaque thread appartient à exactement un groupe de threads. Chaque groupe de threads, excepté un groupe appelé le groupe de threads système, appartient à un autre groupe de threads. Les groupes de threads forment donc un arbre dont le groupe système est la racine. Les groupes de threads fournissent un moyen de gérer les threads et d'imposer des limites de sécurité s'il y a un "security manager". En

effet, ils imposent des limites à l'accès aux différents threads : un thread n'a le droit d'agir (par exemple de demander une action "stop") que sur un thread appartenant à son groupe de threads.
Quand un thread T1 crée un thread T2, le thread T2 appartient par défaut au groupe de threads dans lequel est inclus T1.
Chaque groupe de threads possède une priorité maximale qui ne peut être dépassée par aucune priorité d'un des threads de ce groupe. Chaque groupe de thread a un nom (une chaîne de caractères) pour être identifié.

2.3.3 Les priorités des threads

Chaque thread a une priorité. Quand il y a compétition au sujet d'une ressource, les threads de plus haute priorité sont, en général, exécutés en premier. Quand un thread T1 crée un autre thread T2, le nouveau thread T2 acquiert comme priorité initiale celle du thread T1.
La gestion de la priorité des threads se fait grâce à des attributs et à des méthodes de la classe Thread du package java.lang.

2.3.4 Le "Scheduling"

Si l'on utilise une machine possédant un seul processeur, tous les threads partagent la même CPU. Cela signifie que les threads "runnable" n'agissent en fait que les uns après les autres. Quand des threads "runnable" ne sont pas en train de s'exécuter du fait du "Scheduling", ils sont placés dans une queue de plannification basée sur les priorités et gérée par le système à l'exécution. Les threads de plus haute priorité s'exécutent en premier. Cependant si plusieurs threads ont la plus haute priorité, le système en choisit un de manière arbitraire.
Si un thread de priorité n devient "runnable" alors qu'un thread de priorité m inférieure à n est en train de s'exécuter, ce dernier est suspendu pour permettre au premier thread de s'exécuter.

2.4 Le mécanisme de synchronisation

2.4.1 Généralités

La synchronisation permet d'obtenir un comportement déterministe. Elle est obtenue en utilisant des "monitors". Les moniteurs sont des mécanismes de haut niveau qui permettent qu'une partie de code ne soit exécutée que par un seul thread à la fois. Ils fonctionnent à travers un mécanisme de verrous. Il y a un verrou associé à chaque objet. Seuls les threads peuvent demander le verrouillage ou le déverrouillage d'un objet. Les accès aux verrous sont réglés par des files d'attente.
La synchronisation permet l'exclusion en assurant dynamiquement qu'un seul thread à la fois puisse accéder à l'objet considéré.

2.4.2 Les verrous

Les verrous se trouvent dans la mémoire principale, commune à tous les threads. La synchronisation est implémentée par l'accès exclusif au verrou interne (appelé parfois un "mutex") qui est associé à chaque objet Java. Chaque verrou se comporte comme un compteur. Si sa valeur est non nulle à l'appel d'une méthode ou d'un bloc "synchronized" par un thread T1, cela signifie qu'un autre thread T2 possède ce verrou. Le thread T1 passe alors dans l'état "blocked" jusqu'à ce que la valeur du compteur soit nulle. En entrée, le compteur est incrémenté. Il est décrémenté à la sortie de chaque méthode ou block "synchronized", même si ce dernier s'est terminé par le lancement d'une exception.

2.4.3 L'instruction “synchronized”

L'instruction “synchronized” effectue deux actions :

1. après avoir déterminé la référence de l'objet concerné, cette commande verrouille l'objet puis exécute le corps de l'instruction
2. à la fin de l'exécution du corps de la commande, elle déverrouille l'objet.

Cette méthode peut donc bloquer le thread qui l'emploie si l'objet que le thread désire verrouiller l'est déjà par un autre thread. On peut aussi déclarer qu'une méthode est “synchronized” ; dans ce cas, cela revient à inclure le corps de la méthode dans une commande “synchronized”.

2.5 Les différents états d'un objet

Nous pouvons considérer deux grands types d'objets, les threads et ceux qui n'en sont pas, appelés ici “simples”. Il est à noter que dans la documentation de SUN, on rencontre les termes “thread” et “thread object” pour différencier le flot de contrôle de l'objet thread lui-même.

2.5.1 Les différents états d'un objet thread

A sa création, un thread T se trouve dans l'état “created”. Cela signifie qu'il n'est pas encore actif. Ce thread T passe de l'état “created” à l'état “activated” lorsqu'un autre thread T' appelle sa méthode “start”. Cela pousse T à invoquer sa méthode “run”, le thread T passe alors à l'état “alive”.

Le thread T reste alors “alive” jusqu'à ce qu'un des cinq évènement suivants se produise :

1. la méthode “run” se termine normalement
2. la méthode “run” se termine brutalement par le lancement d'une exception
3. le thread invoque la méthode “stop” sur l'objet thread
4. un autre thread invoque la méthode “stop” sur le thread T
5. un thread invoque la méthode “exit” de la classe “Runtime”

Ces cinq évènements provoquent le passage du thread de l'état courant à l'état “dead”. En passant à cet état, le thread invoque la méthode “notifyAll” sur l'objet qu'il représente.
Un thread dans l'état “alive” est dit “runnable” s'il est en train de s'exécuter normalement, ou s'il est mis en attente du fait de l'exécution du système par le “scheduler”. Un thread “runnable” peut passer dans l'état “suspended”. Dans l'état “suspended”, le thread est en attente, il n'exécute aucun travail. Un thread peut passer dans cet état lorsque lui ou un autre thread invoque la méthode “suspend”. Le thread revient dans l'état “runnable” après une opération de reprise. Cette opération a lieu lorsqu'un autre thread invoque la méthode “resume” sur ce thread. Le thread est alors dit “resumed”. La méthode “resume” n'a pas obligatoirement à être appelée par le thread qui a invoqué la méthode “suspend”. Une autre méthode a le même effet, il s'agit de la méthode “sleep” qui oblige le thread considéré à suspendre toute activité pendant un temps donné. Un thread peut aussi passer de l'état “runnable” à l'état “dormant” quand il invoque la méthode “wait”. Un thread dans l'état “runnable” peut aussi passer dans l'état “blocked” s'il est dans l'attente d'un verrouillage sur un objet, il quitte alors cet état quand il a obtenu le verrouillage sur cet objet.
Les différents états d'un thread sont résumés dans la figure 1.

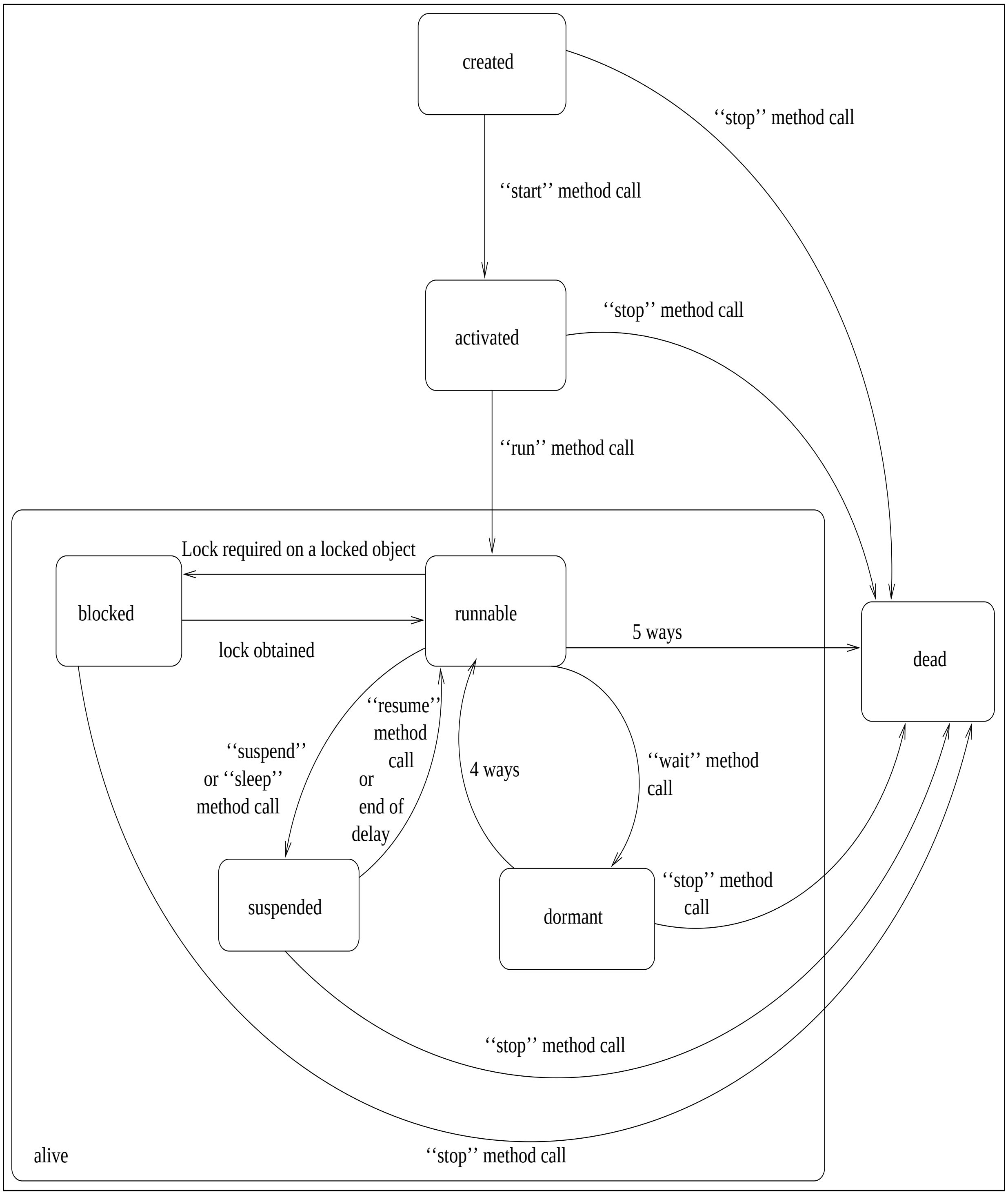

FIG. 1 – *Les différents états d'un thread.*

2.5.2 Les états d'un objet

Un objet peut être caractérisé par deux attributs :

– il peut être "reachable", "finalizer-reachable" (abrégé en "f-reachable" dans la figure 2) ou "unreachable"

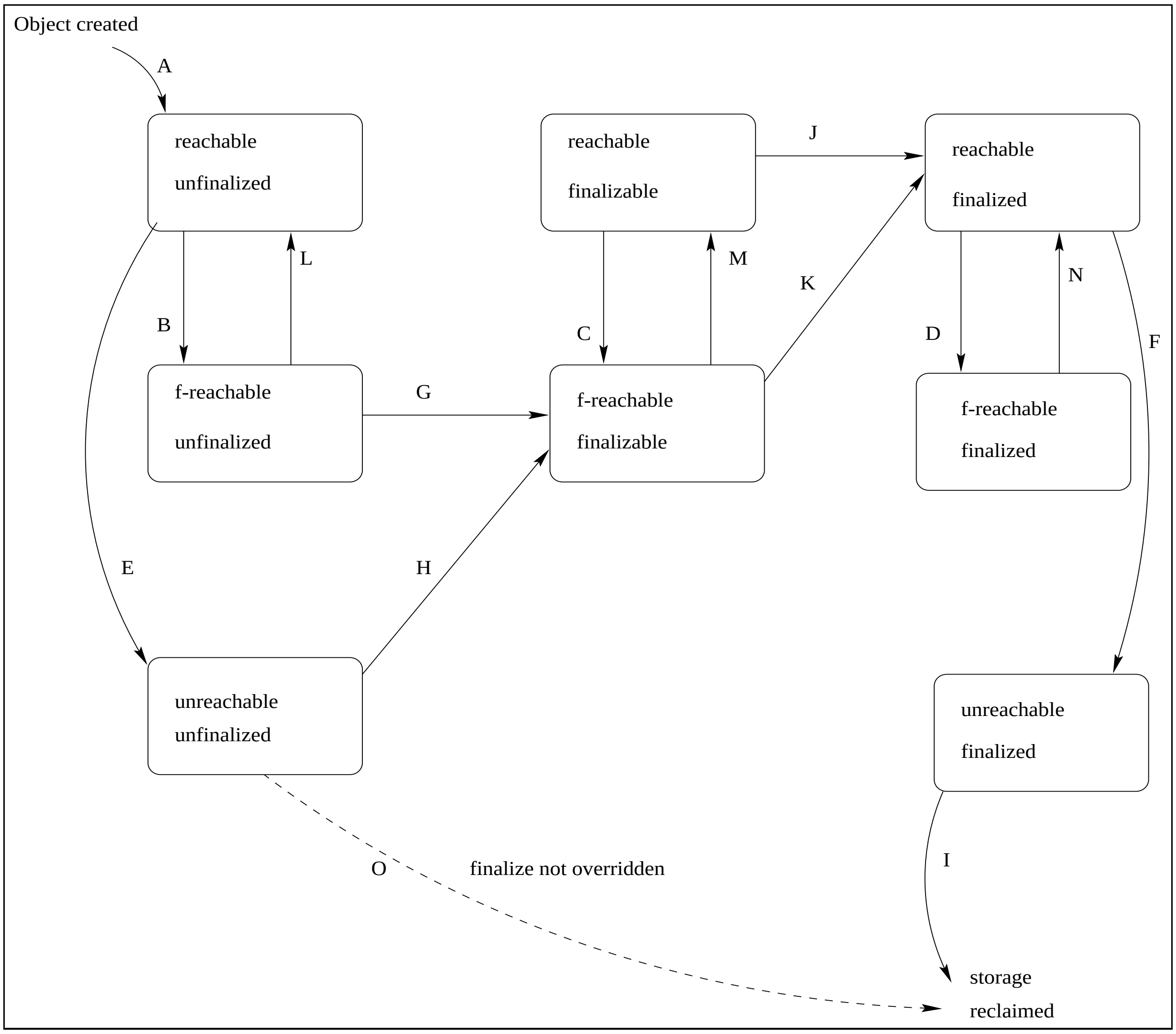

FIG. 2 – *Le cycle de vie d'un objet.*

– et d'autre part, il peut être "unfinalized", "finalizable" ou "finalized"

Un objet est dit "reachable" s'il est accessible au cours de l'exécution future d'un thread étant dans l'état "alive" . Un objet dit "finalizer-reachable" est un objet qui peut être atteint par un objet dit "finalizable" à travers une chaîne de références mais pas par un thread "alive". Un objet dit "unreachable" ne peut être atteint par aucun moyen.
Un objet "unfinalized" est un objet dont la finalisation n'a jamais été invoquée automatiquement. Un objet "finalized" est un objet dont la finalisation a été invoquée automatiquement par la machine virtuelle de Java. Un objet "finalizable" est un objet "unfinalized" mais dont la finalisation peut éventuellement être invoquée par la machine virtuelle.
Le cycle de vie d'un objet est représenté dans la figure 2.
Un objet dit "simple" est tout d'abord créé (action A). Il passe alors dans un état dit "reachable and unfinalized".
Comme les références à un objet sont effacées durant l'exécution d'un programme,

un objet qui était "reachable" peut devenir "finalizer-reachable" (actions B, C ou D), ou "unreachable" (actions E ou F).
Si la machine virtuelle détecte qu'un objet "unfinalized" est devenu "finalizer-reachable" ou "unreachable", elle peut lui donner l'étiquette de "finalizable" (actions G ou H). De plus, si l'objet était "unreachable", il devient "finalizer-reachable" (action H).
Si la machine virtuelle détecte qu'un objet "finalized" est devenu "unreachable", elle peut réclamer la place mémoire occupée par l'objet car ce dernier ne pourra jamais redevenir "reachable".
A tout instant, la machine virtuelle peut prendre un objet "finalizable", le marquer comme étant "finalized", et invoquer sa méthode "finalize" dans un thread. Dans ce cas, l'objet devient "finalized" et "reachable" (actions J ou K), et cela peut aussi faire repasser d'autres objets de l'état "finalizer-reachable" à l'état "reachable" (actions L, M ou N).
On peut remarquer qu'un objet ne peut pas être à la fois "unreachable" et "finalizable" car il doit au moins pouvoir être atteint par le thread invoquant sa finalisation, ce qui fait qu'il n'y a que huit états possibles pour un objet.
La méthode "finalize" ne peut être invoquée qu'une seule fois sur un objet donné même si celui-ci redevient "reachable". L'action O se produit quand la classe dont l'objet considéré est une instance n'a pas de méthode surchargeant la méthode "finalize" de la classe "Object" du package java.lang.

Nous voyons donc que des grands concepts apparaissent dans Java : les **objets**, les **threads** qui définissent les activités concurrentes, ainsi que les principes d'**héritage** et de **liaison dynamique**. Ces grands concepts doivent naturellement être les bases de notre définition formelle de la sémantique de Java.

3 Principes de la sémantique de Java et choix effectués

Cette partie va présenter les principes et les structures de la sémantique de Java que nous avons définis. Elle introduira d'autre part l'environnement de programmation que nous avons utilisé.

3.1 Sémantique Naturelle et Sémantique Opérationnelle Structurelle

Ce chapitre va présenter les formes de sémantiques étudiées et utilisées pour spécifier la sémantique de Java ainsi que l'environnement de programmation choisi pour cette spécification.

3.1.1 Présentation

La Sémantique Naturelle [7], étudiée en cours de DEA, est adaptée aux langages séquentiels. Cependant, Java, en étant un langage multi-threads, est un langage de programmation concurrente. Pour pouvoir prendre en compte les notions d'activités concurrentes et d'entrelacement, nous avons du utiliser la Semantique Opérationnelle Structurelle [10] en introduisant la notion de continuation sémantique. Par contre, les spécifications des points ne mettant pas en jeu ces notions, comme la spécification de l'héritage, sont effectuées en sémantique naturelle.

La sémantique naturelle consiste à écrire une spécification à l'aide d'axiomes et de règles d'inférence. Ces règles d'inférence sont similaires à celles utilisées en déduction naturelle en logique, ce qui explique le nom donné à cette sémantique. On peut parler au sujet de la sémantique naturelle de sémantique *big-step*. En effet, avec elle, l'exécution du programme consiste en un passage de l'état initial en un état final sans qu'il n'y ait d'autre étape visible.
La sémantique opérationnelle structurelle décrit la signification d'un langage de programmation en spécifiant comment il s'exécute sur une machine abstraite. Elle définit les constructions d'un langage par un ensemble de règles de réécriture sous forme d'un ensemble de règles de transition. Cela permet de dire que la sémantique opérationnelle structurelle est *small-step*. En effet, avec elle, l'exécution d'un programme consiste à partir d'un état initial, à franchir un pas de calcul pour arriver en un deuxième état, et ainsi de suite jusqu'à l'arrivée en un état final. C'est grâce à ce système de transitions que nous avons pu introduire la notion de continuation sémantique pour décrire ce qu'il reste à exécuter dans le programme à une étape donnée.

3.1.2 Un système de programmation adapté : le système Centaur

Le système Centaur [8] est un outil automatique de génération d'environnements de langages de programmation écrit par des membres du projet C.R.O.A.P. Il permet d'effectuer toutes les étapes de spécification d'un langage : spécification de sa syntaxe grâce au langage Metal, de ses règles d'affichage grâce au langage PPML et de sa sémantique grâce au langage Typol [1]. Il dispose de plus d'une interface homme-machine dans laquelle sont intégrés les outils ainsi générés.

Le langage **Metal** fourni dans l'environnement Centaur permet la description de la syntaxe sous forme de grammaire non contextuelle. Par exemple, l'affectation est décrite par :

```
<Expression> ::= <Unary> <Ass_Op> <Expression> ; (1)
```

```
   binary (<Unary>, <Ass_Op>, <Expression>) (2)
<Ass_Op> ::= "=" ; (3)
   ass () (4)
```

Ces règles indiquent que lors d'une affectation (instruction considérée comme une expression), on doit avoir successivement un terme "Unary", le symbole "=" et un terme "Expression". Les éléments "Unary" et "Expression" sont des non-terminaux qui restent à définir, alors que le symbole "=" est un terminal : son rôle est de signaler la présence d'une instruction d'affectation. L'ensemble des règles de ce type (règles (1) et (3)) forment la syntaxe concrète du langage. Le langage Metal donne aussi la possibilité d'indiquer l'arbre abstrait que l'on veut associer à une production de la grammaire non contextuelle décrivant le langage. Considérons par exemple les règles (2) et (4) ci-dessus.La règle (2) indique que l'on veut construire un arbre nommé "binary", ayant pour fils un terme sous-arbre abstrait représenté par le terme "Unary", l'opérateur abstrait d'assignation représenté par le terme "Ass_Op" qui indique seulement le type d'opérateur binaire concerné par ces règles et un sous-arbre abstrait représenté par le terme "Expression".
La définition de la syntaxe abstraite est donnée par un ensemble de règles décrivant des schémas d'arbres. Voici la syntaxe abstraite associée à l'expression Java définissant un opérateur binaire restreint au cas de l'affectation :

```
Expression ::= binary ... ;
binary -> Expression Bin_Op Expression ;
Bin_Op  ::= ass ... ;
ass  -> implemented as SINGLETON ;
```

Les règles de la syntaxe abstraite sont composées de phyla (pluriel de phylum), comme *Expression* et *Bin_ Op*, et d'opérateurs comme *binary* et *ass*. Les phyla représentent les catégories syntaxiques, (les types du langage) par exemple les instructions et les expressions, alors que les opérateurs représentent des schémas d'arbres primitifs. Il y a des opérateurs d'arité fixe ou variable. Dans ma spécification de la syntaxe abstraite de Java, les termes commençant par une majuscule sont des phyla, alors que ceux commençant par une minuscule sont des opérateurs.
Le langage **Ppml** permet de spécifier la représentation textuelle de l'arbre de syntaxe abstraite obtenu grâce au fichier Metal.
Le langage **Typol** permet de spécifier la sémantique d'un langage en évaluant l'arbre de syntaxe abstraite qui lui est asssocié. Il offre un moyen d'expression des règles sémantiques indépendamment du langage cible, il comporte des mécanismes permettant d'avoir la trace de l'instruction du programme à évaluer pendant son exécution et offre un système de vérification de type automatique. Une règle Typol classique peut se présenter sous la forme d'un axiome ou d'une règle d'inférence :

```
hypotheses |- consequent
```

```
 h1 |- c1 ... hn |- cn
-----------------------
         H |- C
```

Les types des termes sont définis à travers des jugements placés en début de programme ou de chaque chapitre. Voici par exemple le jugement définissant l'appel du prédicat lançant l'interprétation d'un programme : **judgement (string, Compilation_ Unit, OBJECT_ L, O_ IDENT → OBJECT_ L)**. Ce jugement indique que le set correspondant (ici *execute*) doit être appelé avec une chaîne de caractères, une liste d'objets, l'identificateur d'un objet et que le résultat attendu doit être de type liste d'objets.

Le système Centaur m'a paru être un système tout à fait adapté à mon travail car il permet de spécifier à la fois la syntaxe concrète et la syntaxe abstraite de Java ainsi que sa sémantique. Au-delà de ces aspects formels, il dispose aussi d'outils de visualisation, de debuggage et d'une interface homme-machine qui le rendent très utile.

3.2 Structures sémantiques

Les structures sémantiques sont des arbres de syntaxe abstraite qui ne sont pas issus directement de l'analyse des constructions du langage étudié. Ces structures (ou arbres sémantiques) sont introduites pour les besoins spécifiques de la définition sémantique du langage.

Notre sémantique doit tenir compte de la spécificité du langage Java : être un langage orienté 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 (solution la plus simple) 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 propre 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 (cf. figure 2)

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 étape de l'exécution du programme, son état est parfaitement décrit par la liste courante des objets.
Nous devons choisir une structure pour la liste d'objets qui réponde aux critères suivants :

- conformité par rapport à l'optique de SUN (le créateur de Java)
- lisibilité de la sémantique
- facilité de visualisation
- efficacité

– contrôle de l'entrelacement

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 fairons 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, 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 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 (cf figure 1) 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.

Voici la structure sémantique que nous avons retenue après cette étude : ce sera une liste d'objets dans laquelle seront rassemblés les threads (figure 3) et les objets simples (figure 4).

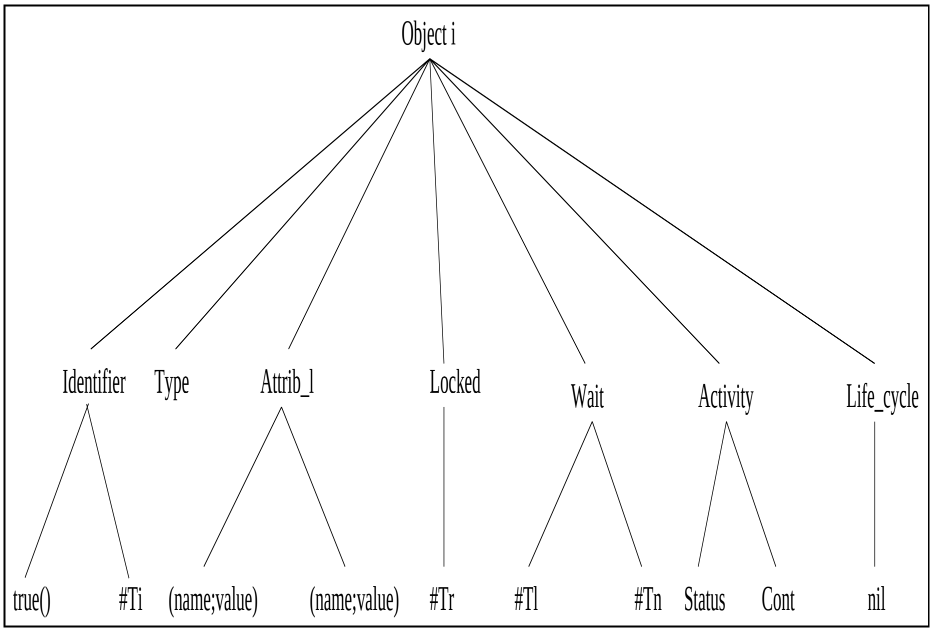

FIG. 3 – *Structure finale d'un objet thread.*

3.3 Les principales unités sémantiques de Java

Java est un langage qui permet de dissocier clairement des principes forts :

- l'héritage
- la gestion des activités concurrentes
- l'exécution des instructions au niveau de chaque thread
- 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
- 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

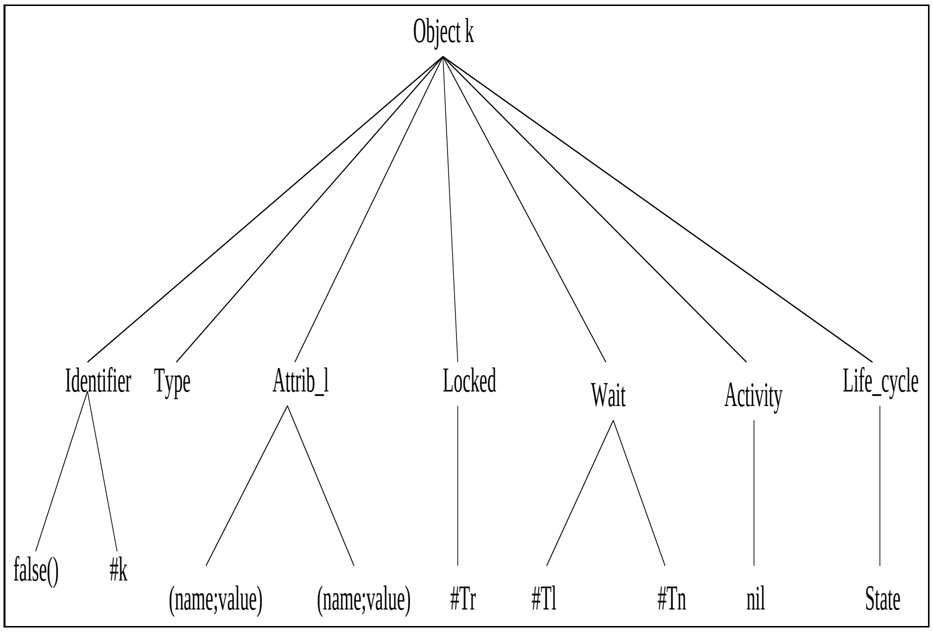

FIG. 4 – *Structure finale d'un objet "simple".*

introduira une plus grande lisibilité de notre sémantique.

Cette étude des différentes possibilités de définition de la sémantique de Java nous a permis de mettre à jour des solutions qui ne paraissaient pas toujours intuitives mais qui se sont révélées correspondre à l'esprit de Java et permettant une définition claire et assez simple de sa sémantique.

4 Définition formelle de Java

Cette partie va présenter la définition formelle de Java que nous avons écrite ainsi que l'environnement que nous avons créé afin de permettre une bonne visualisation de l'exécution d'un programme Java.

4.1 Préparation de la phase d'analyse sémantique

Cette phase a consisté en l'étude de la syntaxe de Java et en la définition formelle de la syntaxe abstraite des structures sémantiques.

Pour mon stage, je n'ai pas eu à définir la syntaxe de Java : je suis partie de la syntaxe écrite en Metal par Laurent Théry (un membre du projet C.R.O.A.P.) pour la version 1.02 de Java. Une phase de mon travail a donc été la découverte et la compréhension des grandes unités de syntaxe abstraite définies dans ce fichier.
L'unité syntaxique principale de Java est l'unité de compilation de type *Compilation_ Unit*. C'est ce qui représente l'ensemble du programme : cela constitue la racine de l'arbre de syntaxe abstraite de tout programme Java. Une unité de compilation est constituée de trois parties, chacune étant optionnelle :

1. une déclaration de package qui donne le nom entier du package auquel appartient l'unité de compilation
2. des déclarations d'importation qui permettent à des types d'autres packages d'être utilisés au moyen de leur simple nom
3. des déclarations de types de classes et de types d'interfaces

La branche principale de cet arbre est la liste des déclarations de types (c'est-à-dire des classes et des interfaces) qui est de type *Type_ Declaration_ s*. Nous avons étudié jusqu'à présent les déclarations de classes, l'étude des packages (seconde branche de l'arbre de syntaxe abstraite) et des types d'interfaces devant se faire ultérieurement.
Les déclarations de classes *class_ declaration* se composent de :

- d'une liste de modifieurs de type *Class_ Modifier_ s*
- d'un identificateur : le nom de la classe de type *Identifier*
- d'un type optionnel : le type dont hérite la classe s'il y en a un
- d'une liste optionnelle de types : la liste des interfaces que la classe implémente (s'il y en a) de type *Type_ Name_ s*
- et enfin de la liste des composantes de la classe de type *Field_ Declaration_ s*

Ces composantes sont principalement de deux types : des déclarations d'attributs *field_ variable_ declaration* et des déclarations de méthodes *method_ declaration*. Les déclarations de méthodes se composent de :

- d'une liste de modifieurs de type *Method_ Modifier_ s*
- du type du résultat
- d'un nœud de déclaration de la méthode *call_ method* qui est constitué du nom de la méthode (un identificateur) et de la liste de ses paramètres de type *Parameter_ s*
- d'une liste de noms de types
- et enfin du corps de la méthode (la liste de ses instructions) de type *Opt_ Block*

Ce sont les principales structures syntaxiques utilisées dans ma définition de la sémantique de Java avec bien sûr les structures syntaxiques des instructions de Java et de ses expressions. Dans la suite, seules seront traitées l'addition, l'affectation, la création d'objet et l'appel de méthode qui sont considérées comme des expressions en Java.

Aux différentes structures syntaxiques décrites dans le paragraphe précédent se sont rajoutées les unités de syntaxe abstraite permettant de former la structure sémantique décrivant la liste des objets. La principale unité de syntaxe abstraite est naturellement la liste des objets *object_l* qui est constituée d'objets (de type *OBJECT)*. Ces objets ont sept composantes :

1. ce qui les identifie : un nœud de type *O_ID* ayant deux fils : un terme de type booléen indique avec la valeur *true* qu'il s'agit d'un thread (un objet simple aura la valeur *false*), et le numéro de l'objet au sein de la liste, *o_ident*
2. leur type
3. la liste de leurs attibuts qui est une liste de paires *pair_l*
4. le verrou de l'objet terme de type *LOCKED*,qui prendra la valeur *locked*,opérateur ayant pour fils le numéro du thread (*o_ident*qui le verrouille, ou la valeur *empty* si l'objet n'est pas verrouillé
5. la liste des threads qui sont dans l'ensemble "wait" de l'objet *wait_l* (c'est une liste d'identificateur de threads qui peut être vide)
6. l'activité de l'objet de type *ACTIVITY*, qui prend la valeur *empty* dans le cas d'un objet simple et autrement est un opérateur *activity* ayant deux fils, le status du thread, terme de type *T_STATUS*, et sa continuation de type *CONT*
7. le statut de l'objet, c'est-à-dire son état au sein de son cycle de vie, de type *LIFE_CYCLE*

La continuation d'un objet est représentée sous la forme d'une closure *clr* comprenant le numéro de l'objet sur lequel porte la closure et une sous-closure *clr_i* comportant une liste d'instructions à exécuter *inst_l* et l'environnement d'exécution *env*. Cet environnement est un opérateur possédant quatre fils :

- du type dynamique de l'objet sur lequel porte la closure
- du nom de la méthode représentée par la liste d'instructions
- d'une liste de paires nom-valeur représentant les paramètres de la méthode
- d'une liste de paires nom-valeur représentant les variables locales à la méthode

Ce sont les principales unités de syntaxe abstraite qui vont figurer dans les règles d'analyse sémantique.
Le fichier complet de la description syntaxique de Java est disponible en annexe 1.

4.2 Conception des modules définissant la sémantique formelle de Java

Cette partie va décrire la définition sémantique de Java en respectant la modularité énoncée dans la section précédente. Cette définition a été écrite en langage Typol (cf. annexe 2). Un principe important a été suivi au cours de cette spécification : **ne pas créer de structures intermédiaires**. Cela signifie que toute recherche d'attribut, de méthode ou de type de classe s'effectue par un parcours du programme et non au moyen d'une table stockant ces informations. Cela permet d'une part d'éviter de duppliquer du code et d'autre part d'avoir à effectuer une analyse du programme préliminaire à son interprétation.

4.2.1 Sémantique du lancement de l'exécution dans le cas d'un thread unique

La sémantique du lancement de l'exécution est décrite dans deux modules. Le premier module (fichier **java_run.ty**) sert simplement d'interface en indiquant quelle est la fonction qui va lancer l'exécution. Deux appels sont possibles à ce niveau selon l'entrelacement désiré. Si l'on souhaite avoir un entrelacement très fin, le prédicat à appeler est **fine_interpret(compilation_unit)** ; dans le cas contraire, le prédicat **coarse_interpret(compilation_unit)** permet d'avoir un entrelacement moins fin. Ces deux prédicats[1] prennent comme argument l'arbre abstrait représentant le programme à exécuter. Ils retournent comme résultat la liste des objets correspondant au programme. De façon interne, ces deux prédicats appellent le module de création du thread principal **java_main_thread_create(I ⊢ compilation_unit → objl)** avec comme arguments le niveau d'entrelacement (I vaut “fine” ou “coarse”), l'arbre abstrait du programme et une liste d'objets comme résultat attendu.

La sémantique du lancement de l'exécution est spécifiée dans le second module (fichier *java_main_thread_create.ty*). Elle peut se résumer en trois étapes :

1. la création du thread principal par l'appel de :
   create_main_thread(compilation_unit(A, B, type_declaration_s [C.Cl]), Cn, object_l[] → objl)
   où A et B sont les deux premières branches de l'arbre de syntaxe abstraite du programme, C représente la déclaration de la classe se trouvant en tête de fichier, Cn est le nom de cette classe. Naturellement à cette étape, la liste d'objets est vide.

2. le lancement de l'exécution de sa continuation par l'appel du prédicat :
   execute(I, compilation_unit(A, B, type_declaration_s [C.Cl]), objl, o_ident 0 → Objl1)
   où I est le niveau d'entrelacement, objl est la liste d'objets à cette étape (elle ne contient qu'un seul objet : le thread principal). L'identificateur *o_ident 0* indique que c'est le thread de numéro zéro (c'est-à-dire le thread principal) qui va commencer à s'exécuter.

3. la fin de l'exécution

Le thread principal représente le flot initial d'exécution du programme, c'est-à-dire celui qui va exécuter la méthode “main”. C'est lui qui créera par la suite d'autres threads si cela est stipulé dans le programme.
La création du thread principal implique la création d'un objet de la liste (le premier). La sémantique de cette dernière création fait l'objet du paragraphe suivant. Ce thread particulier, transparent au programmeur qui n'en a pas réellement demandé la création est placé immédiatement dans l'état *runnable* afin qu'il puisse commencer à exécuter les instructions du programme. Sa continuation est constituée de la liste des instructions de la méthode “main”. Une fois que ce thread est créé, les règles de lancement de l'exécution sont appelées. Il existe deux règles : une règle

1. Seule le premier est disponible pour l'instant, l'entrelacement devant être spécifié durant la suite de mon stage

d'exécution dans le cas où il reste un thread dans l'état *runnable*, et une règle de terminaison de l'exécution. A ce niveau, le module de définition de la sémantique de la liste d'objets intervient en donnant des procédures de test sur l'état des threads à une étape donnée du programme. La terminaison a lieu lorsque tous les threads se trouvent dans l'état “dead”. Si tous les threads sont dans l'état “blocked”, il y a un “deadlock”.

4.2.2 Sémantique de la liste d'objets

Cette sémantique est décrite dans le fichier *java_ object_ list.ty.*
Dans ce module sont définies les opérations de création d'objets, de mise à jour de la liste d'objets et de consultation de cette liste, d'obtention d'une closure correspondante à un objet et une méthode donnés. Ce module offre aussi des fonctionnalités de test de l'état d'un thread ou d'un objet.

La création des objets
La création des objets se fait par l'appel du prédicat : **create_object(O_IDENT, Opt_Type_Name, Compilation_Unit, O_IDENT → OBJECT);**. Ce prédicat teste la nature de l'objet que l'on souhaite créer grâce à son type.

S'il s'agit d'un thread (cette vérification est spécifiée dans le paragraphe concernant l'héritage), il est créé par l'appel du prédicat : **create_thread_object(O_IDENT, Opt_Type_Name, Compilation_Unit, O_IDENT → OBJECT)** . La création proprement dite du thread dépend de son identificateur ; elle implique :

- la détermination de la liste des attributs du thread par l'appel du prédicat : **make_attribute_list(Opt_Type_Name ⊢ Compilation_Unit → PAIR_L)** qui prend comme arguments le type du thread et l'arbre abstrait du programme et qui rend la liste des attributs
- la vérification de la présence de la déclaration de la méthode “main” (si l'identificateur du thread vaut zéro ce qui implique qu'il s'agit du thread principal) ou “run” (thread normal) au sein de cette classe ou de ses ancêtres et la récupération de la déclaration de cette méthode par l'appel du prédicat : **field(Identifier, Identifier, Identifier ⊢ Compilation_Unit : Field_Declaration, Identifier, BOOL)** qui est défini dans le module de définition de l'héritage détaillé dans le paragraphe suivant
- la mise à jour de son statut (le statut du thread est mis à “runnable” dans le cas du thread principal, à “created” par le thread courant sinon)
- la création de sa closure par l'appel de la fonction : **make_closure(Opt_Type_Name, VAL_L, O_IDENT ⊢ Field_Declaration → CLR)**

S'il s'agit d'un objet simple, il est créé par l'appel du prédicat : **create_simple_object(O_IDENT, Opt_Type_Name, Compilation_Unit ⊢ OBJECT)**. Sa création implique uniquement la détermination de la liste des attributs de l'objet par l'appel de la fonction : **make_attribute_list(Cname ⊢ Sys → Attrs)**.
La fonction permettant d'obtenir la liste des attributs d'un objet est définie dans le module de définition de l'héritage.

L'obtention d'une closure

Une closure est associée à un objet et à une méthode. La première opération effectuée est l'appel du prédicat **make_bind_params(Identifier, VA_L ⊢ Parameter_s → PAIR_L)** qui permet à partir de la liste des valeurs données lors de l'appel de la méthode et de la liste des paramètres de la méthode d'obtenir la liste des paramètres liés de la méthode qui va figurer dans l'environnement de la closure.
A partir de là, on décompose la déclaration de la méthode pour pouvoir analyser la liste de ses instructions. Les instructions de déclarations de variables donnent lieu à la constitution de la liste des variables locales qui sera intégrée à l'environnement par l'appel du prédicat :
make_local_vars(PAIR_L ⊢ Local_Variable_Declaration → PAIR_L) qui met à jour la liste des variables locales avec la nouvelle déclaration. Les autres instructions forment la liste d'instructions de la closure.

Possibilités de consultation de la liste d'objets

Il est possible de connaître les caractéristiques d'un objet par l'appel du prédicat : **object(OBJECT_L, O_IDENT → O_ID, Opt_Type_Name, PAIR_L, LOCKED, WAIT_L, ACTIVITY, LIFE_CYCLE** qui à partir de la liste des objets et de l'identificateur de l'objet, par un simple parcours de liste donne les différents éléments définissant l'objet.
De même, le prédicat **some_thread_runnable(OBJECT_L → BOOL)** parcourt la liste des objets pour trouver les threads et indique par un booléen s'il reste un thread dans l'état "runnable". Le prédicat **all_threads_dead(OBJECT_L → BOOL)** indique lui par un booléen si tous les threads sont dans l'état "dead".

Possibilités de mise à jour de la liste des objets

Le module de définition de la sémantique de la liste d'objets permet naturellement de définir les différentes mises à jour possibles de la liste d'objets. Il est possible :

- d'ajouter un objet à la liste par l'appel de la fonction : **update_object_list(OBJECT_L, OBJECT → OBJECT)** qui ajoute un objet en queue de liste
- de mettre à jour le statut d'un thread par l'appel de **update_thread_status(OBJECT_L, O_IDENT, T_STATUS → OBJECT_L)** qui, à partir de la liste d'objets et de l'identificateur de l'objet concerné, récupère ses caractéristiques et met à jour son statut en renvoyant la nouvelle liste d'objets.
- de mettre à jour l'activité d'un thread par l'appel de : **update_activity_in_object_l(OBJECT_L, O_IDENT, CLR → OBJECT_L)** qui détermine l'ancienne activité de l'objet et qui remplace l'ancienne closure par la nouvelle.

4.2.3 Sémantique de l'héritage

Le module concernant l'héritage (décrit dans le fichier *java_inheritance.ty*) est surtout utilisé lorsque l'on souhaite établir :

- la liste des attributs d'un objet
- la déclaration d'une méthode

La sémantique de l'héritage de Java est relativement simple du fait de l'absence d'héritage multiple. Le concept principal est celui du parcours du graphe d'héritage d'une classe.

Parcours du graphe d'héritage

Le prédicat **ancestor(Opt_Type_Name, Opt_Type_Name ⊢ Compilation_Unit → BOOL)** permet de déterminer si la classe définie par le premier identificateur est l'ancêtre de la classe définie par le second. Le prédicat : **direct_ancestor(Opt_Type_Name ⊢ Compilation_Unit : Opt_Type_Name)** indique l'ancêtre direct de la classe représentée par le premier identificateur. Il existe les mêmes prédicats concernant les héritiers des classes.

Obtention de la liste des attributs d'un objet

La liste des attributs d'un objet est donnée par l'appel du prédicat : **make_attribute_list(Opt_Type_Name ⊢ Compilation_Unit → PAIR_L)** qui à partir du type de l'objet et de l'arbre abstrait du programme renvoie les attributs de l'objet sous la forme d'une liste de paires nom-valeur. Cette opération comprend plusieurs étapes :

1. la récupération de la déclaration de la classe grâce au prédicat : **class(Opt_Type_Name ⊢ Compilation_Unit → class_declaration)**
2. l'obtention de la liste des attributs locaux à la classe par le prédicat: **make_local_attribute_list(⊢ java : PAIR_L)**
3. l'obtention de la liste des attributs hérités par le prédicat : **make_inherited_attribute_list(Opt_Type_Name, java ⊢ Compilation_Unit → PAIR_L)** qui parcourt de façon ascendante le graphe d'héritage de la classe considérée et détermine à chaque étape la liste des attributs locaux en tenant compte des possibilités de surcharge des attributs
4. la concaténation des deux listes ainsi obtenues

Obtention de la déclaration d'une méthode ou d'un attribut

Cette opération s'effectue grâce au prédicat: **field(Identifier, Identifier, Identifier ⊢ Compilation_Unit : Field_Declaration, Identifier, BOOL)** qui recherche l'attribut ou la méthode tout d'abord au sein de la classe courante par l'appel de : **field_in_class(Identifier ⊢ CLASS : Field_Declaration, BOOL)** et qui en cas de réponse négative (si le booléen renvoie "false()") renouvelle l'opération par l'appel du prédicat **field_in_ancestors(Identifier, Identifier, java ⊢ Compilation_Unit : Field_Declaration, Identifier, BOOL)** qui parcourt le graphe d'héritage de la classe de façon ascendante en s'arrêtant dès que la réponse est positive.

Test sur les objets

Le module d'héritage permet de déterminer si un objet est un thread ou non. En effet, il sera un thread si la classe dont il constitue une instance hérite de la classe "Thread". Ce test se ramène donc à un simple parcours du graphe d'héritage. Le résultat est obtenu par l'appel de : **not_a_thread(Opt_Type_Name, Compilation_Unit → BOOL)** qui renvoie le booléen faux si l'objet est un thread.

4.2.4 Sémantique de l'exécution dans le cas d'un thread unique

Dans le cas d'un thread unique, la sémantique de l'exécution est assez facile à organiser. L'exécution d'un programme consiste en l'exécution des instructions de la closure associée à l'activité du thread principal. La première étape de l'exécution demandée par l'appel du prédicat général **execute** est l'appel du prédicat **exec_thread(string, Compilation_Unit, OBJECT_L, O_IDENT → OBJECT_L)** qui détermine les caractéristiques du thread allant exécuter ses instructions, afin de pouvoir récupérer son l'activité. Une fois que cela est fait, l'appel du prédicat **exec_activity(string, Compilation_Unit, OBJECT_L, O_IDENT, O_IDENT ⊢ ACTIVITY → OBJECT_L)** va dissocier l'activité du thread afin de déterminer la closure associée au thread considéré et va lancer finalement l'exécution de la première instruction de la liste avec l'appel du prédicat **exec_statement(string, Compilation_Unit, OBJECT_L, O_IDENT, O_IDENT ⊢ Statement, ENV → OBJECT_L, ENV)**. Le "set" *exec_statement* détermine si l'instruction est une expression ou une commande proprement dite c'est-à-dire un terme de type *STATEMENT* et appelle la fonction correspondante. Pour l'instant, toutes les instructions traitées sont des expressions. Le prédicat appelé est donc : **exec_expr(string, Compilation_Unit, OBJECT_L, O_IDENT, O_IDENT ⊢ EXPRESSION, ENV → OBJECT_L, ENV)**. Il permet de tester le type de l'expression. A ce niveau, on peut distinguer plusieurs expressions importantes :

- les expressions de calcul
- l'affectation
- les appels de méthode
- la création d'un objet

Sémantique des expressions de calcul

La sémantique de ces expressions est simple, surtout lorsque l'on ne considère qu'un seul thread. Dans le cas d'un calcul avec un opérateur binaire comme une addition, par exemple avec l'expression $\mathbf{A} + \mathbf{B}$, on doit évaluer la branche gauche de l'opérateur **A** pour obtenir une valeur, faire de même avec la branche droite **B** et ensuite demander l'évaluation de l'opérateur $+$ avec ces deux valeurs grâce à l'appel du prédicat : **eval_bin_op(⊢ VAL, Bin_Op, VAL → VAL)** qui effectue le calcul demandé et renvoie le résultat obtenu.

Sémantique de l'affectation

Prenons l'exemple de l'expression $\mathbf{A} = \mathbf{B}$. On doit tout d'abord évaluer l'expression présente dans la branche droite de l'affectation, ici **B**, pour obtenir une valeur et ensuite effectuer l'affectation proprement dite, lancée par l'appel du prédicat **assign(O_IDENT, Identifier, VAL, ENV, OBJECT_L → ENV, OBJECT_L)**. Le set *assign* va identifier l'élément auquel on doit affecter une valeur (ici **A**). Pour cela, il va tester si cet élément est un paramètre de la méthode courante, s'il s'agit d'une variable locale ou alors finalement s'il s'agit d'un attribut de l'objet auquel est rattachée la closure. Quand le set a déterminé la nature de l'élément, il lui affecte la valeur désirée.

Sémantique d'un appel de méthode

L'appel de méthode possède une sémantique beaucoup plus complexe. Voici la syntaxe abstraite le définissant :
call → Access Argument_s ;
Access ::= Name ;
Name → this super null QName ;
QName ::= component Identifier ;
component → QName Identifier ;

Un appel de méthode se présente donc sous la forme : **A.B(Params)** ce qui donne un arbre de syntaxe abstraite de la forme : **call(A.B, Params)**. La première étape d'un appel de méthode consiste donc à déterminer le nom de la méthode à

appeler et la référence de l'objet sur lequel va être appelée la méthode. Pour cela, il faut dissocier l'expression **A.B** pour obtenir la référence de l'objet sur lequel porte la méthode (ici donnée par **A**) et pour récupérer le nom de la méthode à appeler (ici **B**). C'est à ce niveau qu'intervient la récursion : l'expression **A** peut elle-même être un arbre de la forme **component(X,Y)** qu'il faudra à son tour évaluer.
La deuxième partie de la définition de la sémantique d'un appel de méthode consiste en l'évaluation de la liste des arguments de la méthode pour obtenir une liste de valeurs ; cela est fait en appelant le prédicat : **eval_arg(string, Compilation_Unit, OBJECT_L, O_IDENT, O_IDENT ⊢ Argument_s, ENV → VAL_L, ENV)** qui renvoie la liste de valeurs attendue.
La troisième phase de l'appel de méthode est l'appel du prédicat : **add_closure(string, Compilation_Unit, OBJECT_L, O_IDENT, O_IDENT, Identifier ⊢ REF, VAL_L → OBJECT_L)**. Cette fonction va successivement :

1. déterminer le type de l'objet sur lequel porte l'appel de la méthode par l'appel de : **get_object_type(REF, OBJECT_L → Opt_Type_Name)**

2. vérifier que la méthode est bien déclarée dans la classe représentant ce type ou dans un de ses ancêtres et récupérer sa déclaration par l'appel du set *field* du module de spécification de l'héritage

3. construire la closure associée à la méthode et à l'objet considérésgrâce au set *make_closure*

4. ajouter cette closure en tête de la liste des instructions de la closure courante par l'appel du prédicat **add_closure_to_closure(OBJECT_L, O_IDENT, CLR → OBJECT_L)**

5. lancer l'exécution de cette closure grâce au set *exec_activity*

Sémantique de la création d'un objet
La création d'un objet est une opération complexe qui fait appel à un grand nombre de modules.
En Java, l'instruction de création d'un objet se présente sous la forme suivante : **new A (args)**. Sa syntaxe abstraite est :
new_object → Type_Name Argument_s ;
Type_Name ::= QName Primitive_Type ;

La création de l'objet consiste en :

1. la génération de l'identificateur de l'objet par l'appel du prédicat : **gen_object_ident**

2. la création de l'objet en tant que structure sémantique par l'appel de **create_object**

3. la mise à jour de la liste des objets par l'appel du prédicat **update_object_list**

4. l'appel de la méthode associée au constructeur comme cela est expliqué dans le paragraphe précdent grâce aux sets *eval_arg* et *add_closure*

4.3 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 Java. Pour cela, il faut spécifier 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.

4.3.1 Présentation des structures sémantiques introduites précédemment

```
Object : #T0
    Type : ex_juin
    Attributes :  none
    Locked : none Wait Set : none
    Activity :  Thr. Status : Runnable
                Continuation :
                  Closure on the object : 0
                    Instruction list :  a1 = new A1() ;
                                        a2 = new A2() ;
                                        a1.foo() ;
                                        a2.foo() ;
                                        a1 = a2 ;
                                        a1.foo()
                    Environment :
                      Dynamic Type : ex_juin
                      Method Name : main
                      Binded Parameters :  none
                      Local Variables :  a1 = none ;  a2 = none
    Life_cycle : none

Object : #1
    Type : A1
    Attributes :  value = 0
    Locked : none Wait Set : none
    Activity :  none
    Life_cycle : none
```

FIG. 5 – *Représentation textuelle des objets*

De même que pour la description syntaxique de Java, je disposais au début de mon stage de la spécification Ppml correspondante. Je l'ai donc complétée afin de définir la représentation textuelle des structures sémantiques que j'ai introduites (cf. annexe 3).
La représentation textuelle de la structure sémantique d'un thread et d'un objet simple donne le résultat présenté sur la figure 5. On reconnaît la similarité des structures des threads et des objets simples avec la différenciation au niveau des identificateurs *#T0* pour le thread 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.

4.3.2 Environnement de l'interprétation d'un programme

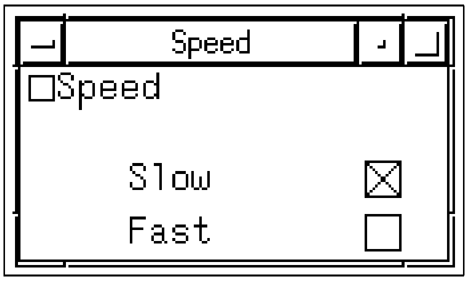

Fig. 6 – *Fenêtre de spécification de la vitesse de l'interprétation*

Afin d'avoir une présentation agréable lors de l'interprétation et surtout de pouvoir étudier le déroulement d'un programme Java, j'ai été amenée à créer différentes fenêtres.
A l'ouverture de l'éditeur contenant un programme Java, une fenêtre apparaît (figure 6). Elle permet d'établir un paramètre concernant l'interprétation du programme : la vitesse d'interprétation. Si l'on veut avoir le temps d'étudier ce qu'il se passe à chaque étape, il faut cocher la case *slow* qui introduit une pause de deux secondes entre chaque mise à jour de la liste d'objets (voir plus loin).

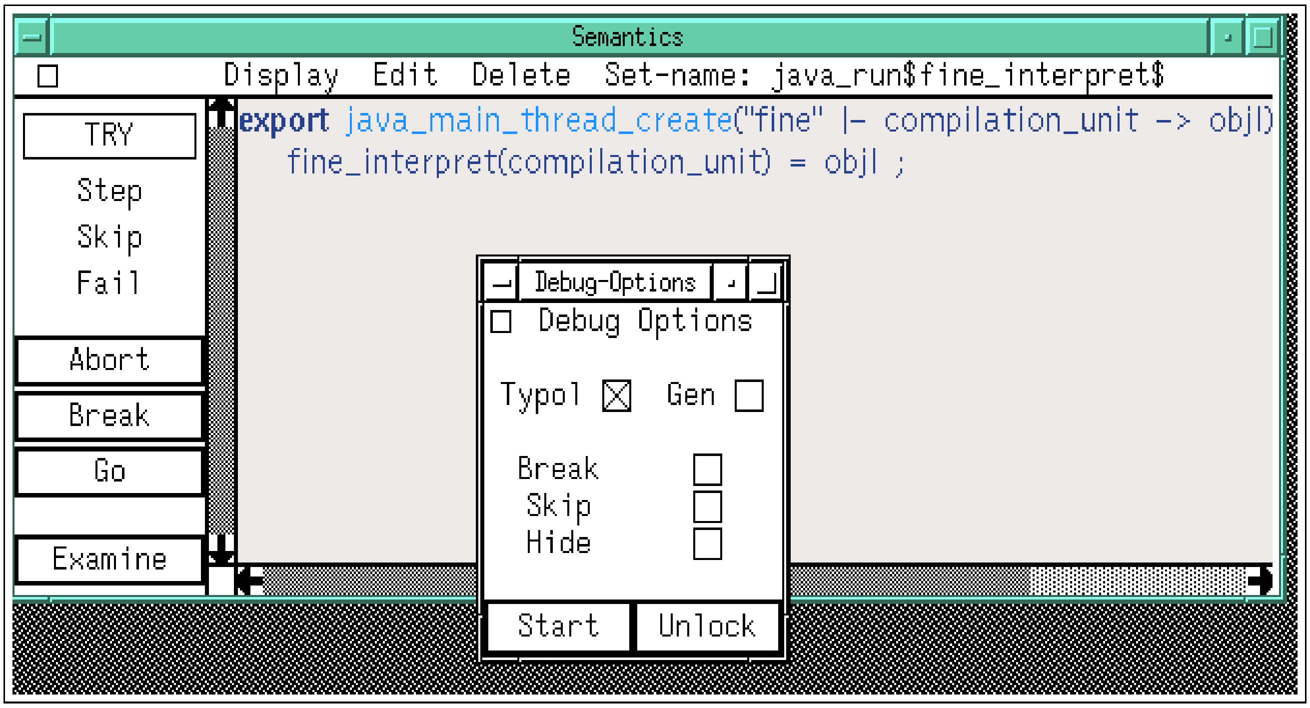

Fig. 7 – *Fenêtres de lancement de l'interprétation en mode debug*

J'ai ajouté à l'éditeur Centaur un menu Java qui se crée lorsque le fichier ouvert est un programme Java et qui permet de lancer l'interprétation du programme grâce au bouton **Interpret**. Les fenêtres *Debug-Options* et *Semantics* (figure 7) 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

ou des caches 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 :

- le bouton *Go* lance l'interprétation qui va se dérouler de manière automatique à la vitesse définie précédemment
- le bouton *Break* permet de demander une pause lorsque l'on a lancé l'interprétation par le bouton *Go*
- le bouton *Step* est utile lorque l'on souhaite dérouler l'interprétation règle par règle. Dans ce cas, les règles Typol utilisées s'affichent une á une, à chaque fois que l'on clique sur le bouton *Step*
- le bouton *Examine* permet d'afficher une autre fenêtre contenant la valeur de la variable sélectionnée dans la règle Typol courante
- le bouton *Abort* permet de stopper de façon brutale l'interprétation à l'étape courante. Dans ce cas, il faut relancer toute l'interprétation, il n'est pas possible de la reprendre à l'étape courante.

La figure 5 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 la liste des objets, il communique la nouvelle liste des objets 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. Pour plus de clarté, au niveau graphique, les parties modifiées apparaissent avec un fond coloré ce qui permet de les voir immédiatement. D'autre part, les identificateurs des objets apparaissent sur fond bleu et l'état de chaque thread est inscrit sur fond vert.
Les fichiers définissant l'environnement d'interprétation sont disponibles en annexe 4.

4.4 Exemple d'interprétation d'un programme Java

Ce chapitre présente un exemple d'utilisation de la sémantique que j'ai définie sur un programme Java réunissant les principaux concepts traités jusqu'à présent.

4.4.1 Présentation du programme

Voici tout d'abord le programme Java dont je vais présenter l'interprétation dans le paragraphe suivant :

```
class  ex_juin2 {
   public static void  main(String args[]) {
      int Sum;
      Sum = 10;
      A1 a1;
      A2 a2;
      A3 a3;
      a1 = new A1();
      a2 = new A2();
      a3 = new A3();
      a1.foo();
      a2.foo();
      a1 = a2;
```

```
        a1.foo();//dynamic binding
        a3.foo();
        a3.foo2(4, Sum);}
}
class  A1 {
    int  value;
    void  A1() {}
    void  foo() {
        value = value + 1;}
}
class  A2  extends A1 {
    void  A2() {}
    void  foo() {
        value = value + 2;}
}
class  A3  extends A2 {
    int  value2;
    void A3() {}
    void  foo2(int t, int r) {
        int val;
        val = 3;
        value2 = value + val + 10;
        value = r + 2 + t;}
}
```

Ceci forme la syntaxe concrète du programme. Sa syntaxe abstraite, c'est-à-dire l'arbre obtenu après son analyse syntaxique, est présentée en annexe 5.
Ce programme réunit les concepts suivant :

- la création du thread principal qui va exécuter la méthode *main*
- la création d'objets simples *a1, a2 et a3*
- l'affectation, par exemple *Sum=10*
- l'existence de variable locales, d'arguments et d'attributs
- l'appel de méthode sans et avec arguments par exemple *a1.foo()* et *a3.foo2(4, Sum)*
- la liaison dynamique lors du deuxième appel de la méthode *foo* sur l'objet a1
- l'héritage avec héritage d'un attribut *value* pour a2 et a3, surcharge de la méthode *foo* pour l'objet a2 et héritage de la méthode *foo* pour l'objet a3

Il forme donc un bon programme de test.

4.4.2 Interprétation du programme

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. Quand cela est fait, la fenêtre de représentation textuelle de la liste d'objets affiche le résultat présenté par la figure 8.

On retrouve la liste d'instructions de la méthode *main* au niveau de la liste d'instructions de la closure associée au thread. La première instruction est l'affectation de l'entier 10 à la variable locale Sum ce qui provoque la mise à jour de la fenêtre

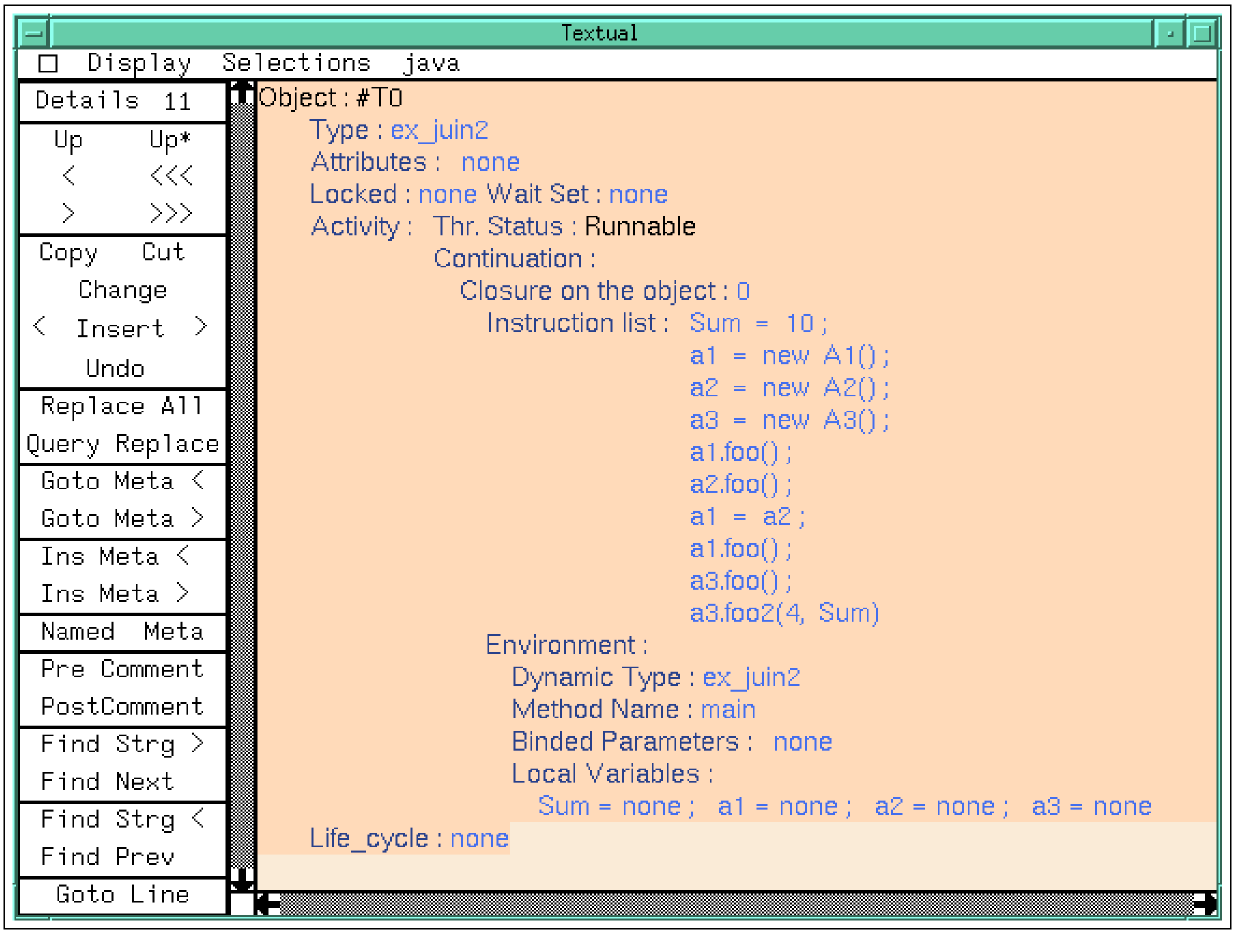

FIG. 8 – *Fenêtre de représentation textuelle de la liste d'objets après la création du thread principal.*

de représentation de la liste des objets. La valeur 10 assignée à la variable locale *Sum* apparaît sur un fond de couleur différente dans l'environnement pour indiquer que c'est à ce niveau que s'est produit une modification. D'autre part l'instruction venant d'être exécutée disparaît de la liste des instructions de la closure du thread principal.
Les trois instructions suivantes du programme entraînent la mise à jour de la fenêtre de représentation textuelle comme cela apparaît sur la figure 9. On s'aperçoit que les trois objets ont été créés avec leurs attributs hérités ou non et que les références de ces objets ont été affectées aux variables locales correspondantes dans l'environnement de la closure du thread principal.

L'instruction $a1 = a2$ effectue un changement de type : la variable *a1* qui désignait un objet de type *A1* doit maintenant désigner l'objet *a2* qui est de type *A2*. Cela est possible car la classe *A2* hérite de la classe *A1*. L'appel qui suit de la méthode *foo* sur l'objet *a1* effectue une liaison dynamique : il va faire appel à la méthode *foo* de la classe *A2* et non plus celle de la classe *A1*.
La prochaine instruction intéressante à étudier est l'appel de la méthode *foo2* avec passage des arguments. Cet appel entraîne la création d'une closure comme nouvelle instruction dans la closure du thread principal comme cela est présenté sur la figure 10. Cette figure nous permet de vérifier que l'environnement de la nouvelle

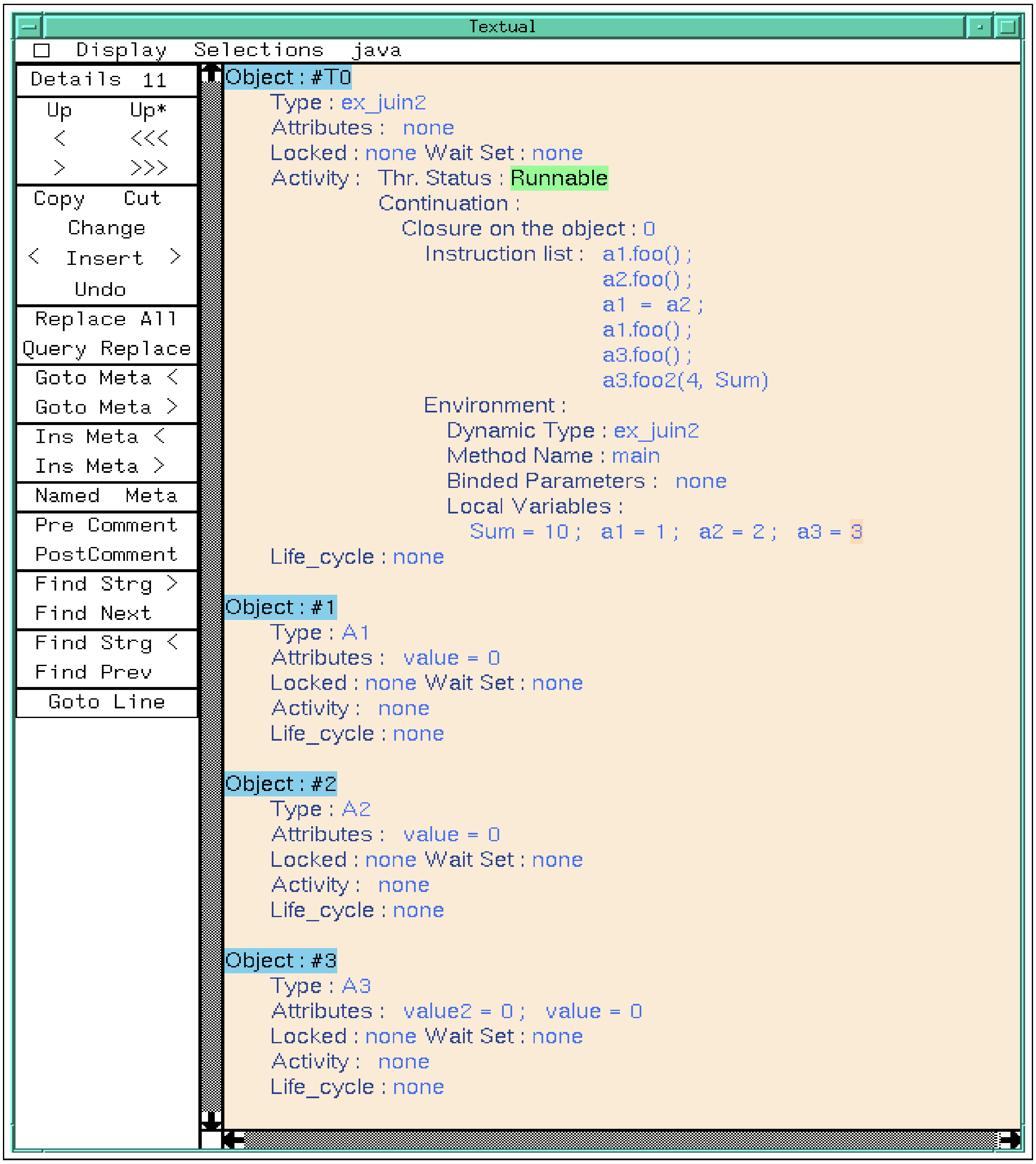

FIG. 9 – *Fenêtre de représentation textuelle de la liste d'objets après la création des trois objets.*

closure a bien pris en compte la liste des valeurs fournie lors de l'appel de la méthode ainsi que la liste des paramètres figurant dans la déclaration de celle-ci. L'état final de la liste des objets à l'issue de l'exécution du programme est décrit par la fenêtre représentée par la figure 11.

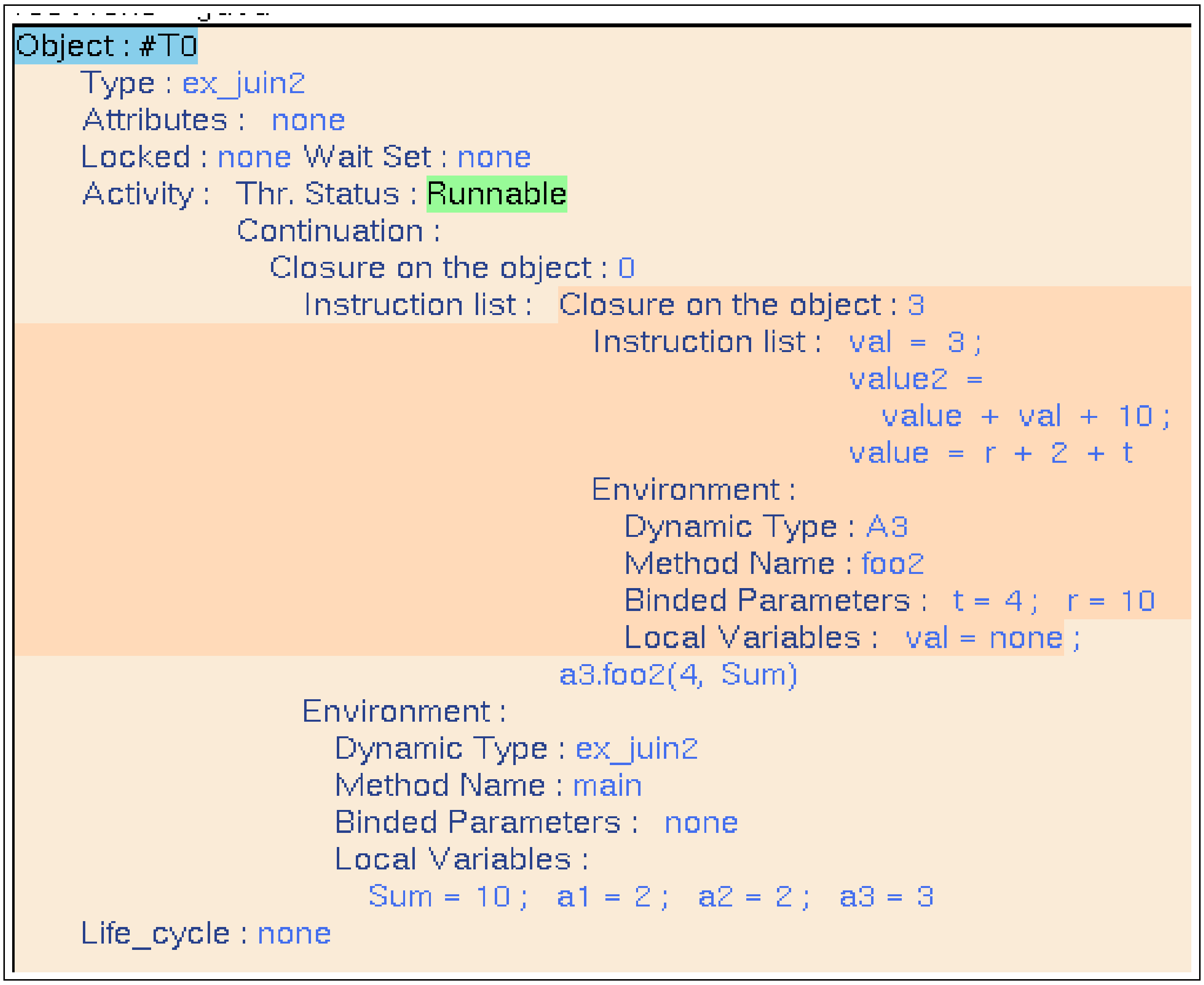

FIG. 10 – *Structure du thread principal lors de l'appel de la méthode foo2.*

La définition formelle de Java que nous venons de présenter, bien que spécifiant un grand nombre de concepts de Java et utilisant les structures sémantiques définitives, n'est bien sûr pas complète. Cependant, elle constitue une base solide sur laquelle vont pouvoir reposer les autres spécifications.

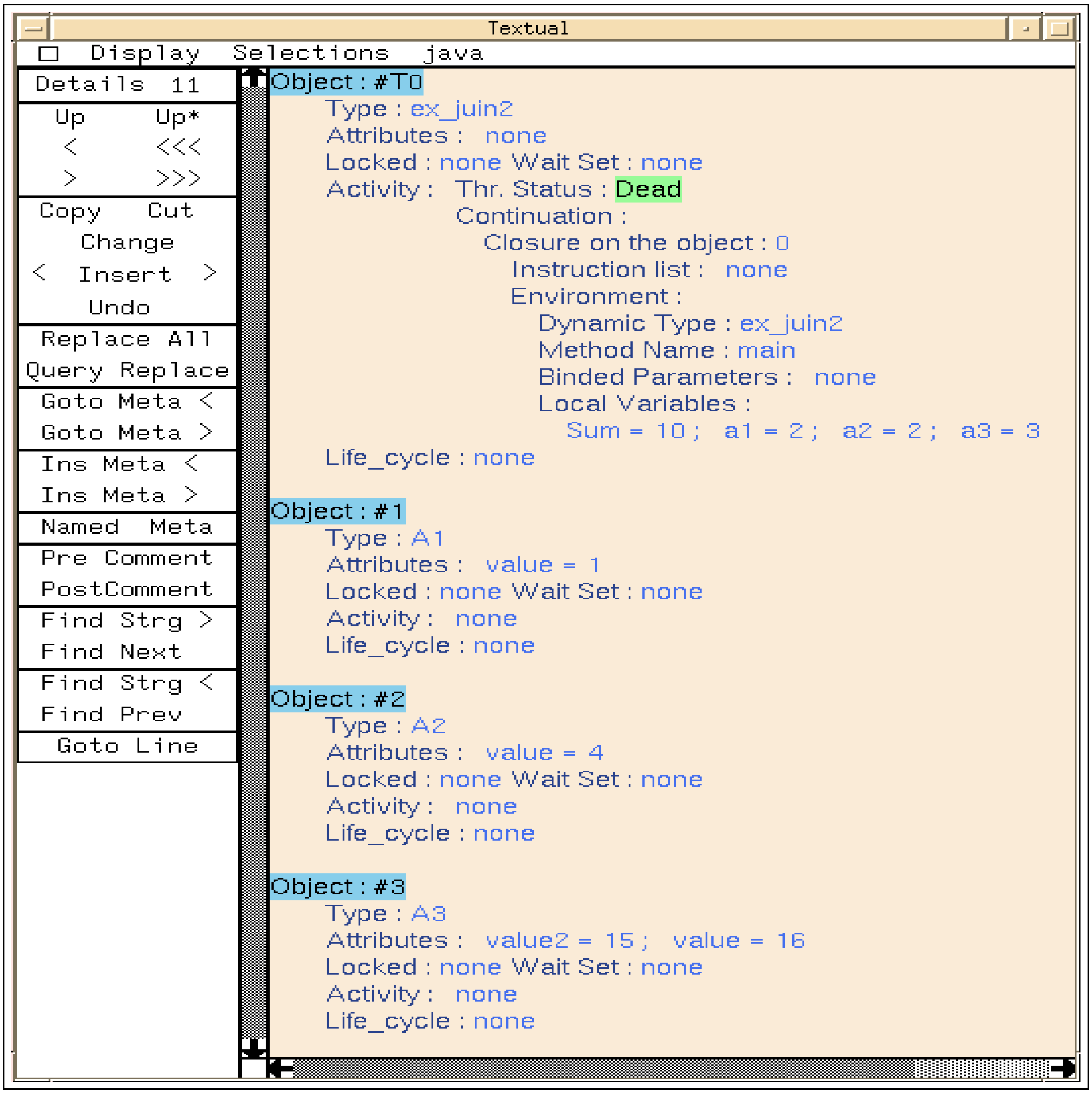

FIG. 11 – *Liste des objets à l'issue de l'interprétation du programme.*

5 Spécifications à compléter

Cette partie va présenter les spécifications qui vont être introduites durant la fin de mon stage.

5.1 Sémantique complète des instructions

Durant la première partie de mon stage, je n'ai pas eu le temps de définir la sémantique de certaines instructions du langage Java. D'autre part, il va falloir introduire la possibilité d'interruption de l'exécution d'un thread pour les besoins de l'entrelacement. Cette partie présente ce qui sera effectué.

5.1.1 Sémantique complète des expressions

Les expressions dont la sémantique reste à spécifier sont assez nombreuses, mais leur sémantique ne posera pas de problèmes. En effet, ce sont soit des opérateurs de calcul qui sont bâtis sur le modèle de l'addition déjà spécifié soit des opérateurs d'assignation et de calcul qui regrouperont deux concepts déjà spécifiés, l'assignation simple et le calcul. Leur définition sera donc simple.
En ce qui concerne l'introduction de l'entrelacement, considérons le cas d'un opérateur binaire. L'opération doit s'effectuer en trois phases :

1. calcul de l'expression de la branche gauche de l'opérateur
2. calcul de la branche droite de l'opérateur
3. application de l'opérateur aux valeurs obtenues

Une interruption doit pouvoir intervenir lors de la fin de chaque phase.
Lors d'un appel de méthode, chaque phase décrite dans la section précédente doit pouvoir être suivie d'une interruption. Cette spécification est à rajouter. Il faudra de plus spécifier la transmission d'une valeur de retour d'une méthode d'une closure à l'autre (cela sera assez simple).

5.1.2 Sémantique des autres instructions du langage Java

Les commandes de Java spécifiées jusqu'à présent formaient toutes des expressions au sein de Java. Les instructions (au sens de Java) n'ont donc pas été spécifiées au niveau sémantique. Cependant la plupart de ces instructions sont classiques dans les langages de programmation (par exemple les tests avec l'instruction *if... then... else...*, les boucles *while, for...*) et leur sémantique est bien connue.
Sur le plan de l'entrelacement, chaque calcul d'expression (par exemple le test de l'instruction *if*) doit pouvoir être suivie d'une interruption. De même, les instructions d'un bloc d'instructions seront considérées comme atomiques : chacune d'entre elles pourra être suivie d'une interruption.

Les spécifications précédentes concernant l'entrelacement étaient bien sûr valables pour un entrelacement fin. Si l'utilisateur décide qu'un entrelacement à gros grains lui convient, les interruptions n'auront lieu qu'au niveau de chaque commande de la closure.

5.2 Sémantique de l'exécution dans le cas d'une application multi-threads

La définition sémantique présentée dans la section précédente concernait le cas d'une application où le seul thread était le thread principal. Une partie très importante de la suite de mon stage va concerner le passage à une application multi-thread qui constitue un des concepts clés de Java.

5.2.1 Introduction du choix de la granularité de l'entrelacement

Tout d'abord, le lancement de l'exécution doit être modifié afin de permettre à l'utilisateur de choisir la granularité qu'il désire. Celle-ci sera ensuite transmise comme paramètre de l'exécution. C'est ce paramètre qui permettra de sélectionner le niveau de finesse de la division de chaque instruction (voir le paragraphe précédent).

5.2.2 Sémantique donnée à l'entrelacement au niveau des threads

L'entrelacement au niveau des threads ne concernera que ceux qui se trouveront dans l'état "runnable". Il sera donc nécessaire de définir un nouvel état, par exemple, "executing", pour différencier les threads pouvant s'exécuter mais ne disposant pas du processeur (les threads "runnable") du thread en train de s'exécuter (le thread "executing"). Cela permettra de visualiser immédiatement ce dernier thread au sein de la liste des objets.
Cependant, il faudra aussi définir un algorithme de choix du thread devant passer à l'état "executing" lors d'une étape de l'entrelacement. Il conviendra aussi de veiller à ce qu'un thread soit stoppé dans un état stable, c'est-à-dire un état à partir duquel il pourra reprendre normalement son activité lorsqu'il repassera dans l'état "executing".

5.2.3 Gestion de l'état des threads et des objets, définition du garbage collector

Dans ce qui a été défini jusqu'à présent, l'état des objets au sein de leur cycle de vie n'est pas pris en compte. Cela va être spécifié durant la seconde partie de stage. A cela va venir s'ajouter la gestion des états possibles d'un thread, le thread principal ne connaissant que les états "runnable" et "dead" dans le cas d'une application uni-thread. C'est à ce niveau qu'il va falloir spécifier la sémantique du garbage collector : à quel moment retirer un objet de la liste, quand appeler les méthodes de finalisation des objets ? Beaucoup de questions vont se poser.

5.2.4 Gestion de la synchronisation au niveau des objets

La gestion de la synchronisation est très importante dans une application multi-threads. En effet, c'est elle qui permet d'obtenir un certain déterminisme quant au résultat donné par un programme. Le module de la sémantique définissant la synchronisation doit spécifier :

- le choix du thread qui va obtenir le lock sur l'objet parmi les différents threads en attente de ce lock
- le choix du thread qui sera "libéré" de l'ensemble "wait" de l'objet lors de l'appel de la méthode "notify" sur cet objet
- le changement d'état d'un thread désirant obtenir un lock sur l'objet et ne pouvant l'obtenir (le thread doit passer de l'état "runnable" à l'état "blocked")

5.3 Sémantique des packages et des interfaces

La définition de la sémantique décrite dans la section précédente ne traite que les déclarations de classes. Dans la suite de mon stage, il va falloir spécifier la sémantique des déclarations d'interfaces et d'objets de type interface.
D'autre part, Java est très apprécié en tant que langage de programmation pour les packages qui lui sont attachés et qui fournissent beaucoup de classes et de méthodes utiles. Notre définition de la sémantique de Java doit donc intégrer la spécification de certains packages, ne serait-ce que pour pouvoir définir les threads. En effet, ceux-ci sont définis comme étant des instances de classes héritant de la classe *Thread* qui se trouve justement dans un package.

5.4 Sémantique de la gestion des exceptions

Un concept important de Java est la possibilité de gérer les erreurs se produisant au sein d'un programme par la levée d'exceptions. Cela doit apparaître dans notre définition de la sémantique de Java. Pour cela, il faut avoir spécifié la sémantique des instructions Java permettant de les manipuler.

Les spécifications deumeurant à définir sont nombreuses. Cela est du à l'ampleur du langage Java qui dispose d'une syntaxe très importante : sa syntaxe abstraite définit environ 143 opérateurs et 61 phyla.

6 Conclusion

Ce chapitre va tout d'abord présenter des remarques générales sur le travail effectué puis il décrira des possibilités d'élargissement du sujet traité. Enfin, il présentera des remarques plus personnelles sur le déroulement du stage.

6.1 Remarques générales

Ce stage s'est déroulé jusquà présent en trois grandes phases.
L'étude des concepts et principes de base du langage Java m'a permis d'établir les bases que devait suivre ma définition d'une sémantique de Java. Elle m'a d'autre part permis de clarifier certaines notions que je n'avais pas encore utilisées.
La deuxième phase de mon stage a consisté à étudier la définition syntaxique de Java dont je disposais et en m'inspirant du travail réalisé par Sidi Ould-Ehmety sur Eiffel [4] et Eiffel parallèle [5] à étudier les différentes possibilités de structures sémantiques afin d'établir ce qui me semblerait être la solution la plus satisfaisante et de l'inclure dans la définition de la syntaxe de Java en Metal. J'ai pour cela introduit environ 38 opérateurs et 30 phyla.
La troisième phase de mon stage a été la plus longue. Il s'agit de la spécification proprement dite d'une sémantique formelle de Java et de la définition d'un environnement pour l'interprétation des programmes Java. J'ai jusqu'à présent écrit un total d'environ 209 règles Typol. Le module spécifiant l'héritage étant le plus important au niveau du nombres de règles.
La continuation de cette phase constituera la suite de mon stage.

6.2 Possibilités de continuations du sujet

Tout d'abord, la spécification de la sémantique formelle que j'ai décrite n'étant pas terminée, la première continuation évidente du sujet est l'achèvement de cette spécification. Cette phase est déjà importante en volume de travail étant donnée la richesse du langage Java. D'autre part, il serait nécessaire d'améliorer l'environnement d'interprétation des programme par une interface graphique en utilisant les outils de visualisation adaptés aux langages parallèles orientés objets [3] développés au sein du projet C.R.O.A.P.
En ce qui concerne Java, si l'on veut pouvoir s'assurer de la sécurité du langage, il est nécessaire de spécifier le format intermédiaire et la sémantique dynamique de la machine virtuelle [9] afin de pouvoir effectuer des vérifications au niveau du bytecode. Cela pourrait constituer une ouverture vers un sujet de thèse.

D'un point de vue plus personnel, ce stage de D.E.A. a tout à fait répondu à mes attentes. En effet, il m'a permis de découvrir les différentes activités d'un chercheur : la recherche et l'étude des différents articles et livres écrits sur un sujet donné, l'étude des différents aspects d'un problème et la spécification d'une solution à ce dernier. De plus, il correspond parfaitement à mon double cursus en me permettant d'étudier un langage de programmation utilisé au sein de mon école d'ingénieurs.