Chapitre 3

Une Sémantique Formelle de Java

Le chapitre précédent a décrit les aspects principaux de Java, ainsi que les difficultés liées à la concurrence, à la synchronisation et aux objets partagés (comme le `CubbyHole` de l'exemple décrit en Section 2.1.3.2). Cette section s'attache à décrire et expliciter notre spécification sémantique de Java. Celle-ci s'appuie sur la description informelle de Java donnée par Sun dans le livre `The Java Language Specification` [76]. Il est important de noter que les règles Typol présentées dans ce chapitre afin d'expliciter certains aspects de notre spécification sont exactement celles utilisées, sans aucune modification ou simplification.

La sémantique dynamique permet de décrire l'exécution d'un programme Java, contrairement à une sémantique statique qui donne des informations au niveau du typage. L'exécution d'un programme Java construit et fait évoluer un ensemble d'objets ; nous avons donc considéré que le résultat de cette exécution était l'état final de cet ensemble d'objets. Notre spécification suppose d'autre part que le programme Java donné en entrée sous la forme d'un arbre de syntaxe abstraite est syntaxiquement correct (ce qui est assuré par le parseur), et correctement typé. Ce dernier point pourrait être vérifié par un vérificateur de typage (ou *type-checker*), basé sur la spécification d'une sémantique statique du langage Java, et qui pourrait être couplé à notre sémantique dynamique. Un tel vérificateur permettrait d'autre part de disposer d'informations sur les types statiques des variables, capitales au moment de la recherche de la méthode à appliquer, comme le type statique de la variable représentant l'objet sur lequel la méthode est appelée, ou le type statique des arguments.
Au lieu de construire des structures intermédiaires (tables de symboles ou mémoires d'un compilateur traditionnel), nous avons choisi d'utiliser directement la syntaxe abstraite du programme pour y chercher les informations nécessaires à son exécution. Nous ne construisons donc pas de tables de symboles pour mettre en mémoire le type des variables, ou le source d'une méthode. Cela évite un par-

cours du source du programme préalable à toute interprétation. A la place, nous parcourons directement l'arbre de syntaxe abstraite du source du programme en recherchant au sein de cet arbre les informations désirées.
Cette absence de table des symboles possède toutefois un inconvénient. En effet, cela nous oblige à parcourir le source du programme lors de chaque recherche, même si elle concerne le même attribut ou la même méthode plusieurs fois de suite, comme dans le cas d'un appel récursif de méthode, où l'on a besoin du source de la même méthode lors de chaque appel. Une table des symboles permettrait dans ce cas précis de n'effectuer cette recherche qu'une seule fois et de garder ensuite son résultat en mémoire.

Une première section décrit une syntaxe abstraite de Java ainsi que les structures sémantiques introduites en détaillant les différents choix effectués. Une deuxième section présente une vue d'ensemble de notre spécification sémantique de Java. La spécification des aspects objets de Java est étudiée au sein de la troisième section. Une quatrième section porte sur les différents mécanismes des exceptions alors que les aspects concurrents sont traités dans la cinquième section. Un sixième section présente alors une piste pour l'intégration du typage statique à travers une étude du masquage des attributs. Enfin, la dernière section permet de situer notre spécification par rapport aux différents travaux connexes.

3.1 Syntaxe abstraite et structures sémantiques

Cette section présente les bases de notre spécification sémantique, c'est-à-dire la définition d'une syntaxe abstraite pour les différents constructeurs du langage Java, ainsi que celle des différentes structures sémantiques introduites.

3.1.1 Spécifications syntaxiques

La spécification des aspects syntaxiques de Java inclut une syntaxe concrète et une syntaxe abstraite. Nous ne nous attachons ici qu'à la description de la syntaxe abstraite qui sert de base à notre spécification sémantique.

La syntaxe abstraite que nous avons définie est constituée de 128 opérateurs et 56 types. Elle couvre la totalité du langage Java dans sa version 1.2, sans se limiter aux aspects inclus au sein de notre sous-ensemble de Java. Nous avons choisi de coller aux unités syntaxiques décrites dans [76] afin de rendre notre spécification lisible pour le plus grand nombre. De plus, la description syntaxique fournie dans [76] était particulièrement adaptée au langage Metal dans lequel les spécifications syntaxiques sont tournées vers les outils d'édition structurée du

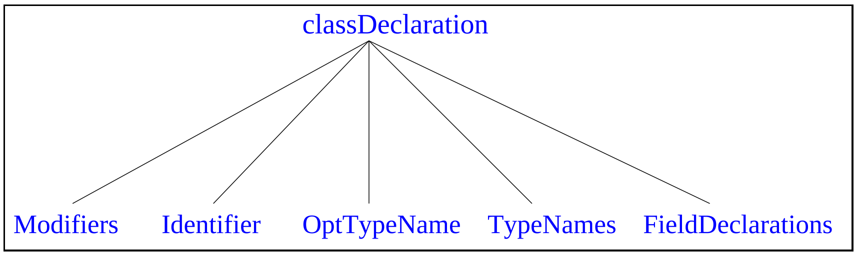

FIG. 3.1 – *Arbre de syntaxe abstraite d'une déclaration de classe.*

système Centaur.
Ce paragraphe présente la syntaxe abstraite des principaux composants d'un programme Java, afin de permettre une meilleure compréhension des règles sémantiques décrites dans les prochaines sections. C'est sous cette forme que les programmes Java sont conservés en mémoire au sein de Centaur, et c'est elle qui est la base de notre sémantique formelle.
Dans cette section, les noms de type (ou phylum) commenceront par des majuscules, alors que les noms des opérateurs commencent par une minuscule.

L'unité racine de la syntaxe abstraite d'un programme Java est l'opérateur `compilationUnit` dont le principal fils est composé d'une liste de déclarations de types (classes ou interfaces). Java étant un langage à objets, la classe est le composant essentiel de tout programme Java. La Figure 3.1 présente l'arbre de syntaxe abstraite associé à une déclaration de classe. Les éléments les plus importants sont : `Identifier` qui est le nom de la classe, `OptTypeName` qui représente le nom de la super-classe si elle existe, et `FieldDeclarations` qui est une liste de déclarations de membres de la classe.

Comme nous l'avons vu dans la Section 2.1.1, les membres d'une classe peuvent être :
- des attributs,
- des constructeurs,
- des méthodes.

L'arbre de syntaxe abstraite de la déclaration des attributs est représenté dans la Figure 3.2. Prenons l'exemple de la déclaration d'attribut suivante (sous forme de syntaxe concrète) :

```
public int longueur = 0;
```

La syntaxe abstraite correspondante est :

```
fieldVariableDeclaration(modifiers[public()], int(),
variableDeclarators[ intializedVariable(identifier ‘‘longueur’’, integer 0))]])])
```

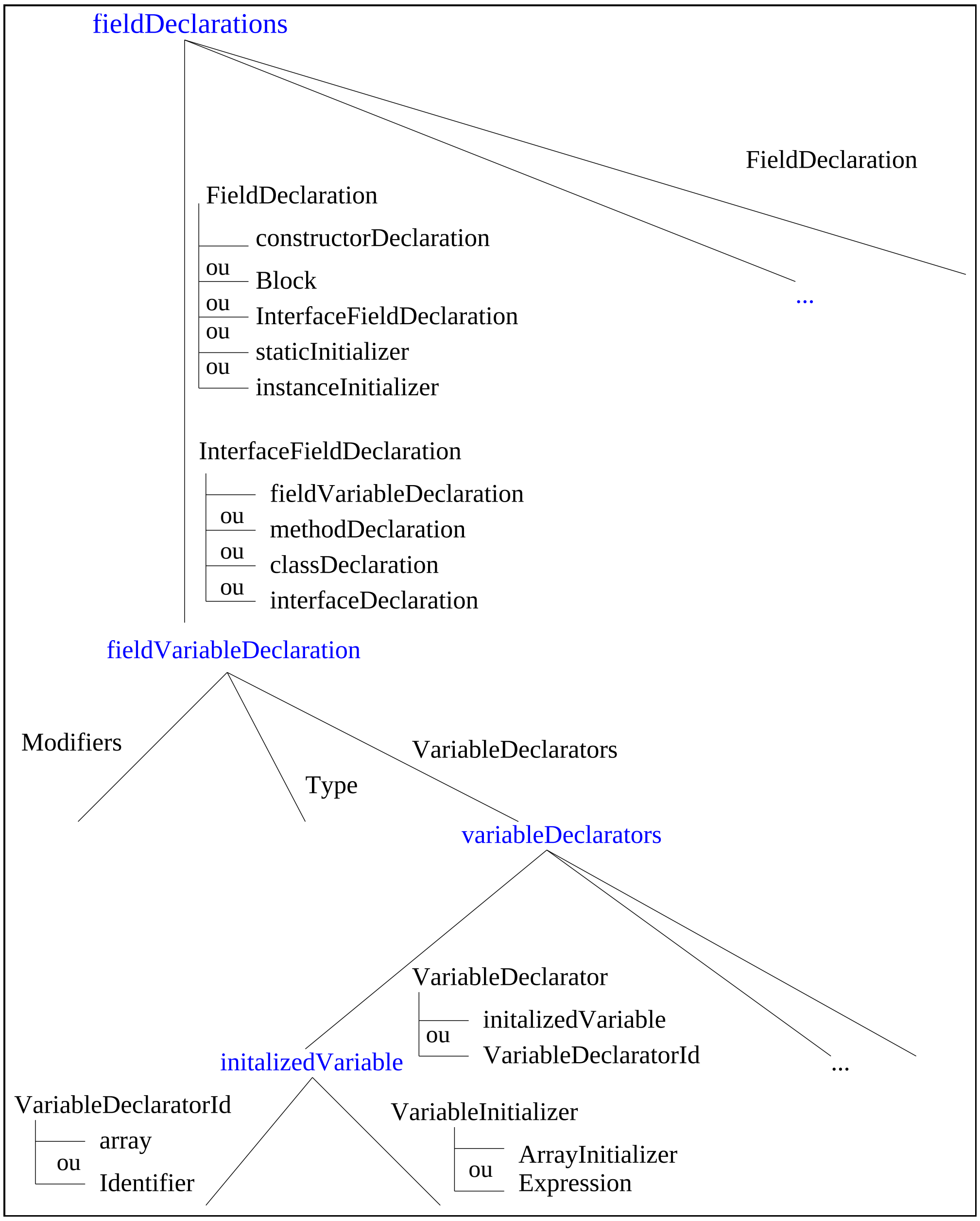

FIG. 3.2 – *Arbre de syntaxe abstraite d'une déclaration d'attribut.*

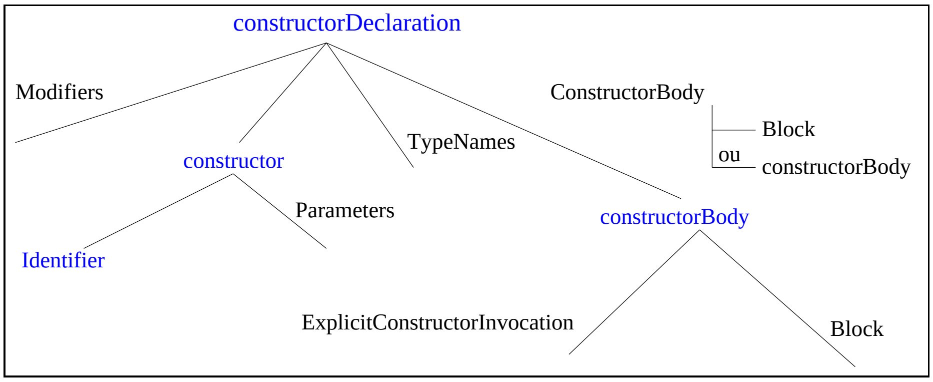

FIG. 3.3 – *Arbre de syntaxe abstraite d'une déclaration de constructeur.*

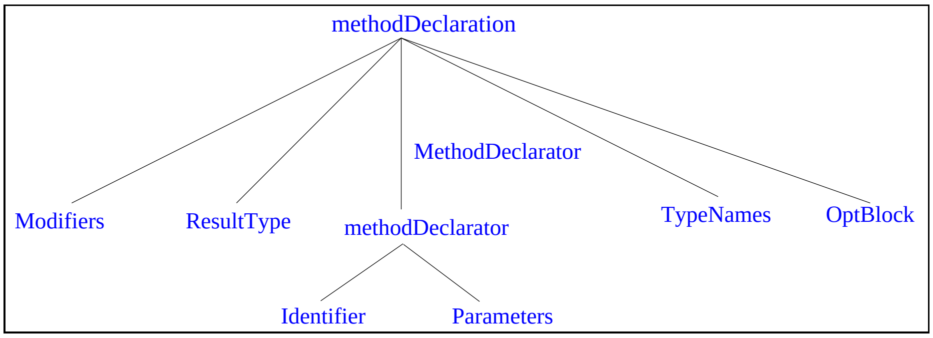

FIG. 3.4 – *Arbre de syntaxe abstraite d'une déclaration de méthode.*

L'arbre de syntaxe abstraite présenté en Figure 3.2 est complexe, bien plus que ne l'est la syntaxe concrète d'une déclaration d'attribut. Cela est dû à la présence de sous-typage représenté par les différentes alternatives proposées pour un type donné. Par exemple, le type *Interface Declaration* est un sous-type du type *InterfaceFieldDeclaration*, qui est lui-même un sous-type du type *FieldDeclaration*.

L'arbre de syntaxe abstraite d'un constructeur est présenté dans la Figure 3.3. Au sein de cet arbre on remarque notamment le nom du constructeur (`Identifier`) et ses paramètres (`Parameters`) tous deux fils de l'opérateur `constructor`, ainsi que la liste d'instructions du constructeur dans `ContructorBody`. Les deux possibilités d'arbres du type `ConstructorBody` tiennent compte de la possibilité de la présence d'un appel explicite d'un autre constructeur (de la même classe ou de la classe ancêtre) en tête de la liste des instructions.

La Figure 3.4 présente l'arbre de syntaxe abstraite d'une déclaration de méthode. Nous pouvons remarquer que cet arbre de syntaxe abstraite est assez proche de celui d'un constructeur, si ce n'est la présence d'un type de la valeur de retour `ResultType`

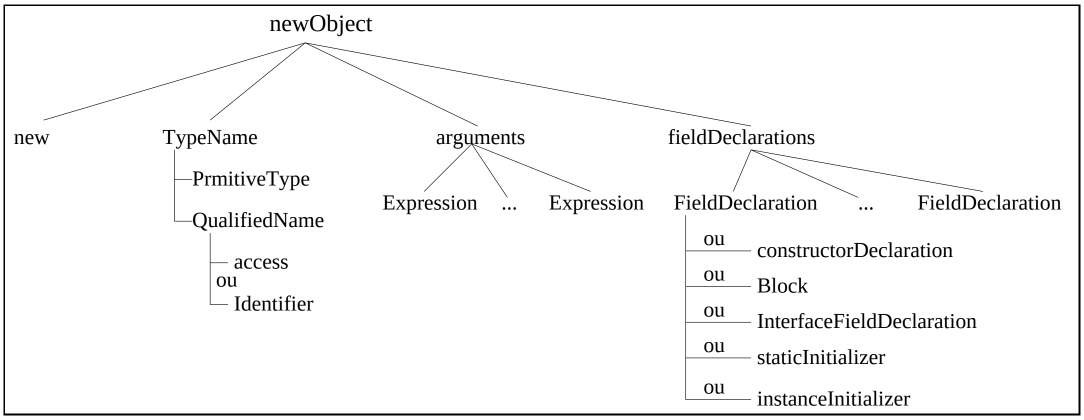

FIG. 3.5 – *Arbre de syntaxe abstraite d'une création d'objet.*

(qui est soit un type du programme soit le mot-clé *void*).

Chaque méthode comprend ainsi une liste d'instructions Java dont nous présentons ici un exemple de syntaxe abstraite.
Considérons la création d'un nouvel objet, dont la Figure 3.5 représente l'arbre de syntaxe abstraite la définissant.
Voici un exemple (sous forme concrète) d'expression de création d'objet :

```
new Producer(cubbyhole);
```

La syntaxe abstraite de cette expression est :

```
newObject(new(), identifier ‘‘Producer’’, arguments[identifier
        ‘‘cubbyhole’’], fieldDeclarations[])
```

L'intégralité de cette syntaxe abstraite est disponible en Annexe 1 (Section 6.2.2.3). Cette syntaxe abstraite des constructeurs du langage Java doit être complétée avec la définition des structures sémantiques et de leur syntaxe abstraite.

3.1.2 Structures sémantiques

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

Ce paragraphe explicite la définition des différentes structures sémantiques que nous avons été amenés à introduire, ainsi qu'à la spécification de leur syntaxe abstraite. Ces structures sémantiques portent à la fois sur les objets et les threads, et sur les continuations nécessaires à la simulation de la concurrence sous forme d'entrelacement.

3.1.2.1 Objets et threads

Java étant un langage à objets, la première structure sémantique importante doit 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 est un entier (solution la plus simple) qui sert de référence lorsque l'on souhaite utiliser l'objet ou simplement le désigner. L'identificateur de la classe est son nom.
Cette liste de caractéristiques 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 indique, dans le cas où l'objet est verrouillé, par quel thread et combien de fois il l'est.
- Un ensemble *wait* de threads qui vont vouloir le verrouiller à une certaine étape future de l'exécution.

Pour représenter les différents objets créés au cours d'un programme, la structure sémantique la plus simple est un ensemble d'objets (qui est implanté sous forme de liste). En effet, à chaque étape de l'exécution du programme, l'état de l'exécution est parfaitement décrit par l'ensemble courant des objets.
Nous devons choisir une structure pour cet ensemble d'objets qui réponde aux critères suivants :
- Conformité par rapport à l'optique de SUN selon laquelle les threads sont avant tout des objets comme les autres,
- Lisibilité de la sémantique, facilitant ainsi sa mise au point et sa compréhension par un lecteur non averti.
- Facilité de visualisation : la structure de liste choisie pour représenter l'ensemble des objets permet d'afficher directement cet ensemble sous forme textuelle à l'aide d'un formatage direct, sans aucune transformation de la structure par l'environnement.
- Efficacité : dans le cadre de notre sémantique qui est directement exécutable au sein du système Centaur, il est nécessaire pour que notre sémantique soit utilisable qu'elle soit efficace, permettant ainsi d'interpréter un programme Java en un temps raisonnable.
- Contrôle de l'entrelacement : au sein de notre sémantique, nous n'avons pas introduit de possibilité de contrôle sur l'entrelacement par l'utilisateur, mais cela serait facilement réalisable. Cela a d'ailleurs déjà été fait dans le cadre de la spécification d'Eiffel// [9].

Nous devons également trouver une structure sémantique pour représenterles threads. 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'objets 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 affichons la liste des objets à l'écran, nous faisons seulement apparaître l'identificateur, en introduisant une différence textuelle : un thread a un identificateur du type $\#Tnum$, alors que l'identificateur d'un objet "simple" est du type $\#num$.

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

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 qui sera utile au moment de l'ordonnancement 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 d'exécution) constitue une closure qui définit l'activité future du thread considéré. Chaque thread possède donc une closure représentant son activité future. A ce niveau

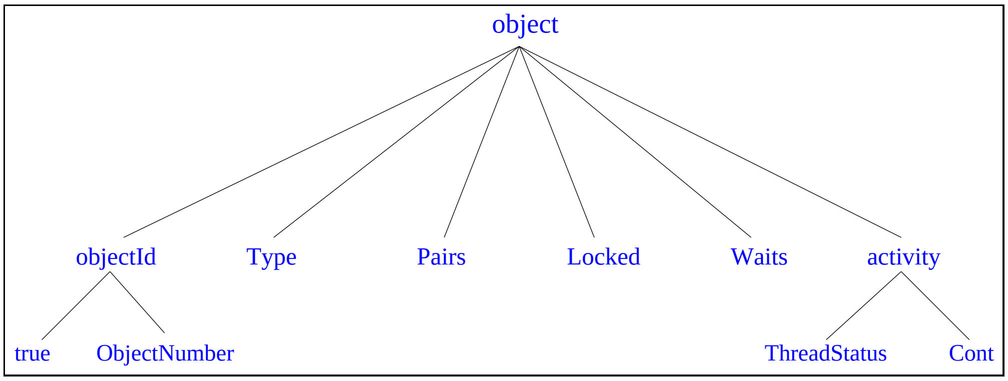

FIG. 3.6 – *Structure sémantique d'un thread.*

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 : il s'agit d'un ensemble (implanté sous forme de liste) dans lequel seront rassemblés les threads et les objets simples.
La Figure 3.6 présente la structure sémantique correspondant à un thread. Dans le cas d'un objet simple, la seule différence par rapport à la structure sémantique d'un thread est que le champ *activity* prend la valeur *empty*. L'activité d'un thread est composée de son statut (*executing* ou *blocked* par exemple) et d'une continuation (ce que le thread doit exécuter). Cette continuation est constituée :

- de l'identifiant du thread
- du nom de la méthode courante
- d'un environnement d'exécution (paramètres et variables locales) sous forme de deux listes de paires (nomVariable, valeurVariable)
- d'une liste d'instructions représentant ce que le thread doit exécuter lors des prochains pas d'exécution. Cette liste contient à la fois des instructions du langage Java et des closures correspondantes aux appels de méthodes.

Comme on l'a vu plus haut, durant son exécution, un programme Java crée, utilise et met à jour des objets et des threads. Mais il faut aussi tenir compte des classes

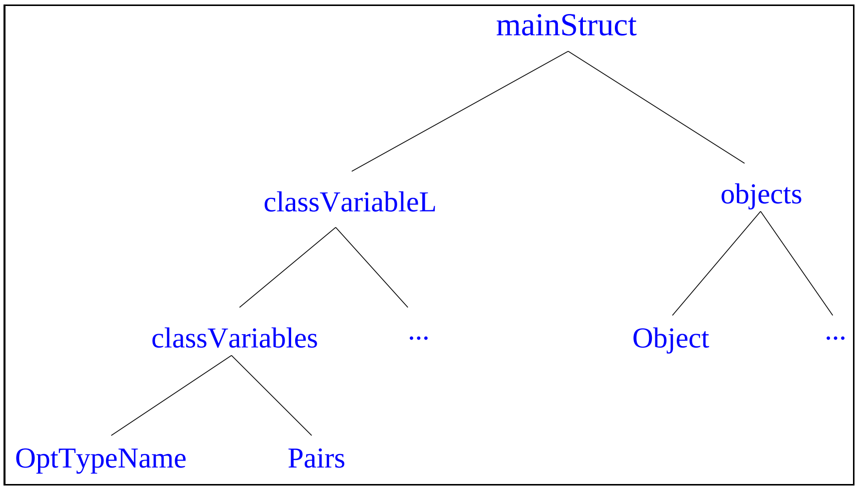

FIG. 3.7 – *Arbre de syntaxe abstraite de la structure sémantique principale.*

possédant des variables de classe ; ces dernières ne sont pas modélisées par la structure sémantique d'un objet, il faut donc les représenter par une nouvelle structure. La méthode la plus simple est de construire un ensemble de classes munies chacune de leur liste de variables de classe, en ne faisant figurer dans cette liste que les classes possédant des variables de classe (pour des raisons évidentes d'efficacité).
Afin de simplifier les règles sémantiques, nous avons décidé de réunir cette nouvelle structure avec celle des objets sous la forme d'un arbre dont les deux fils à la racine sont l'ensemble des classes possédant des variables de classes, et l'ensemble des objets. Cela nous permet ainsi de n'introduire qu'un seul paramètre dans les règles Typol ne manipulant pas ces deux structures, et de ne transmettre qu'un argument à l'environnement. La Figure 3.7 présente l'arbre de syntaxe abstraite obtenu. On y retrouve d'une part une liste de classes possédant des variables de classe sous forme de paires (nom de la classe-liste des variables de classe), et d'autre part la liste des objets.

3.1.2.2 Continuations

Des continuations, comme l'explique la Section 2.2.2.2, sont des constructeurs de syntaxe abstraite que nous sommes amenés à introduire afin de pouvoir simuler l'entrelacement entre threads lors de l'exécution de programmes Java concurrents. En effet, si les constructeurs propres au langage Java suffisent en sémantique big-step, ils sont largement insuffisants en sémantique small-step, où chaque instruction du langage Java est divisée en sous-instructions atomiques. Ce paragraphe présente un exemple de continuations introduites, afin d'expliciter notre propos.

Considérons l'exemple d'une affectation :

```
Variable = ExpressionGauche + ExpressionDroite
```

dont la syntaxe abstraite est

```
binaryAssign( Variable, assignment(), binary( ExpressionGauche, plus(),
                          ExpressionDroite))
```

La Figure 3.8 résume les différentes étapes de l'évaluation de cette affectation. D'après les spécifications informelles de Java, la première étape de cette affectation est l'évaluation de :

`ExpressionGauche + ExpressionDroite`

(notée 3 sur la Figure 3.8), suivie de l'affectation proprement dite (notée 4). Cela représente déjà deux étapes distinctes lors de l'évaluation de cette expression. De plus, l'évaluation de la somme :

`ExpressionGauche + ExpressionDroite`

se décompose elle aussi en trois étapes : calcul de `ExpressionGauche` (étape 1), calcul de `ExpressionDroite` (étape 2), suivi de l'évaluation de la somme proprement dite (étape 3). D'autre part, `ExpressionGauche` et `ExpressionDroite` peuvent elles aussi être des expressions complexes nécessitant plusieurs étapes d'évaluation, par exemple si ce sont des appels de méthode.

D'après notre politique d'ordonnancement (définie à la section 2.1.3.2), il y a interruption du calcul après chaque étape élémentaire afin d'introduire de l'entrelacement. Il faut donc à chaque reprise de l'exécution de cette affectation savoir exactement à quelle étape du calcul nous nous trouvons. De plus, suite à la décomposition du calcul, on peut se retrouver à une étape donnée avec une valeur définissant une référence (par exemple `#T4`), qui n'est pas prévue dans les constructeurs du langage Java. C'est pour cela que nous avons introduit des continuations, et donc que nous avons spécifié leur syntaxe abstraite.
Revenons à l'exemple ci-dessus de l'affectation. Pour pouvoir évaluer cette expression, nous avons défini principalement un opérateur *semAssign* et un type *SemExpression*. L'opérateur *semAssign* permet de traiter par exemple le cas de l'expression `Variable = #4` qui affecte une valeur référence à une variable, ou encore le cas de `#3.frere = #6`, qui affecte la valeur référence `#6` à l'attribut `frere` de l'objet `#3`.
Le type *SemExpression* est un type englobant à la fois le type *Expression* qui appartient au langage Java, et le type *Ref* (entre autres). Cela permet de représenter les valeurs références comme `#T3` par exemple.
Explicitons cette transformation sur l'exemple suivant :

Simon.frere = Pascal

La Figure 3.9 présente l'évaluation de cette expression à la fois sous forme de syntaxe concrète (pour être clair), et de syntaxe abstraite (pour observer l'introduction des continuations). Les continuations ont été introduites en particulier pour gérer les nouveaux éléments des structures sémantiques étudiées dans le paragraphe précédent, par exemple l'opérateur `objectId` qui représente la référence d'un objet (e.g. `#3`), le booléen indiquant s'il s'agit d'un thread (valeur *true()*) ou non.

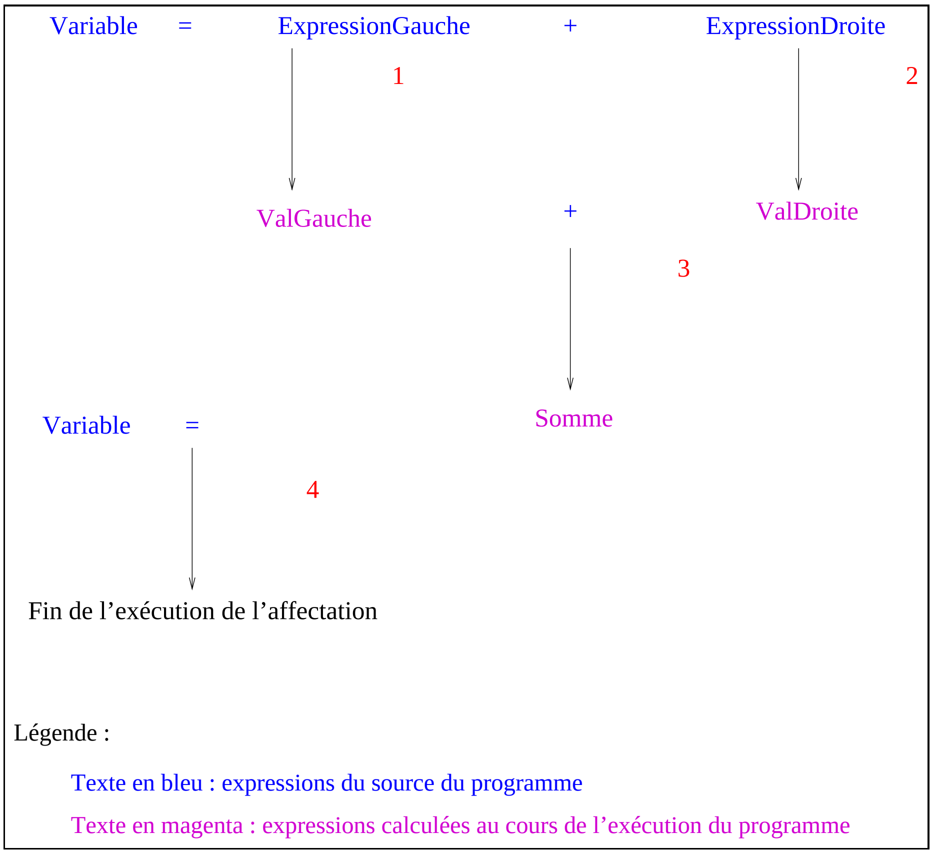

FIG. 3.8 – *Etapes de l'évaluation d'une affectation.*

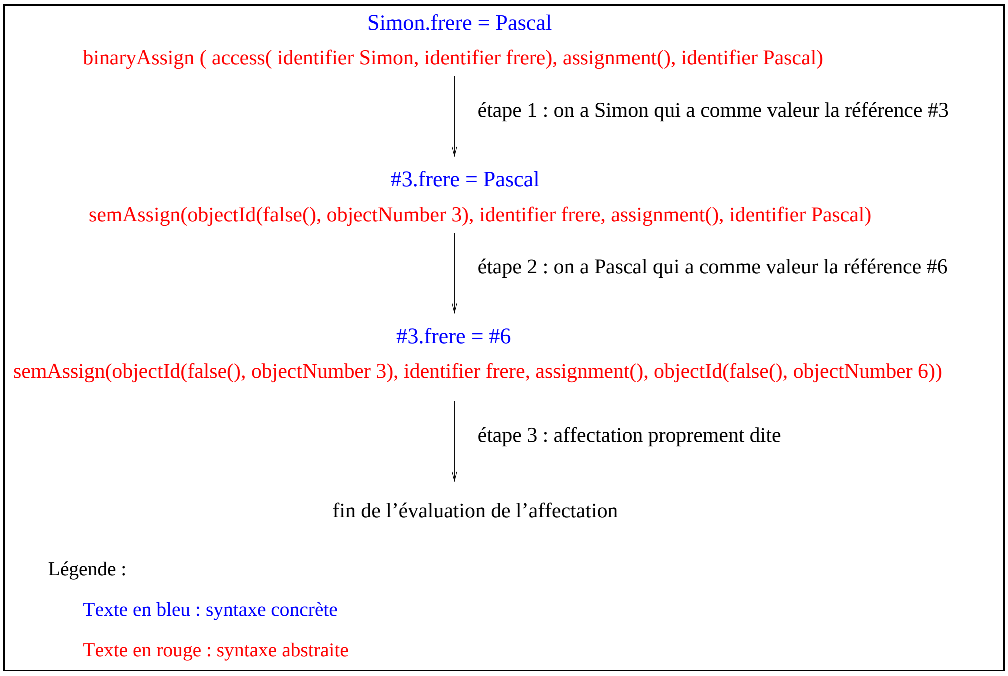

FIG. 3.9 – *Etapes de l'évaluation d'une affectation en syntaxe concrète et abstraite.*

L'intégralité de la syntaxe abstraite des structures sémantiques et des continuations introduites pour spécifier une sémantique de Java est disponible en Annexe 2.

3.2 Organisation générale

L'objectif de cette section est de donner une vue d'ensemble de notre spécification d'une sémantique formelle dynamique de Java. Les prochaines sections présentent de façon approfondie les principaux aspects de cette définition.

Cette section présente les différents modules dont est composée notre sémantique, et le système de transitions que nous avons développé.

3.2.1 Une spécification modulaire

Ce paragraphe présente les différents modules constituant notre spécification d'une sémantique formelle dynamique de Java.
Certains modules sont écrits en Sémantique Naturelle, principalement en ce qui concerne les aspects objets, par exemple la création d'un nouvel objet dont les différentes phases

(création, recherche des attributs, de l'activité s'il s'agit d'un thread) sont considérées comme une étape atomique au sein de notre sémantique. Les autres modules sont écrits dans un style opérationnel utilisant la Sémantique Naturelle (possibilité offerte par Typol), particulièrement en ce qui concerne les aspects liés à la concurrence comme l'exécution des instructions ou l'évaluation des expressions. Le style opérationnel nous permet ainsi d'introduire de l'entrelacement au niveau de l'exécution des threads et donc de simuler la concurrence.

Nous avons opté pour une spécification modulaire, nous rapprochant en cela de l'approche suivie par E. Börger et W. Schulte dans [31]. Cette approche nous a paru naturelle. En effet, Java est un langage qui permet de dissocier clairement des aspects forts :

- Les objets
- L'héritage
- La gestion des activités concurrentes
- L'exécution des instructions au niveau de chaque thread
- L'évaluation des expressions
- La présence de bibliothèques (ou API).

Il est important que la sémantique que nous définissons conserve cette modularité. En effet, cela facilite la mise au point de cette spécification, et introduit une plus grande lisibilité de notre sémantique. De plus, cela nous permet de plus de la développer de manière incrémentale, c'est-à-dire d'ajouter la spécification de nouveaux aspects du langage (comme par exemple des API), sans pour autant modifier les modules déjà définis (si ce n'est l'ajout d'une règle introduisant le nouveau module).
Elle doit de plus intégrer de nouveaux aspects :

- Le lancement de l'exécution
- La gestion de la structure sémantique introduite : la liste des objets
- La gestion de la communication avec l'environnement (voir le chapitre 4).

Ces différents aspects forment autant de modules au sein de notre définition d'une sémantique de Java.

Notre spécification se compose d'environ 800 règles Typol organisées en modules présentés dans la Figure 3.10. La partie gauche de la Figure présente les modules principaux, alors que la partie de droite regroupe les modules qui sont utilisés par les modules principaux.
Ces modules ont tous une fonction bien particulière :

- Modules "principaux" :
  - *java_program_interpretation.ty* : crée le thread principal (celui qui exécute la méthode *main* et qui est déclaré de façon implicite), lance l'interprétation du programme en modélisant le système de transitions (voir paragraphe suivant pour plus de détails), et achève l'interprétation.

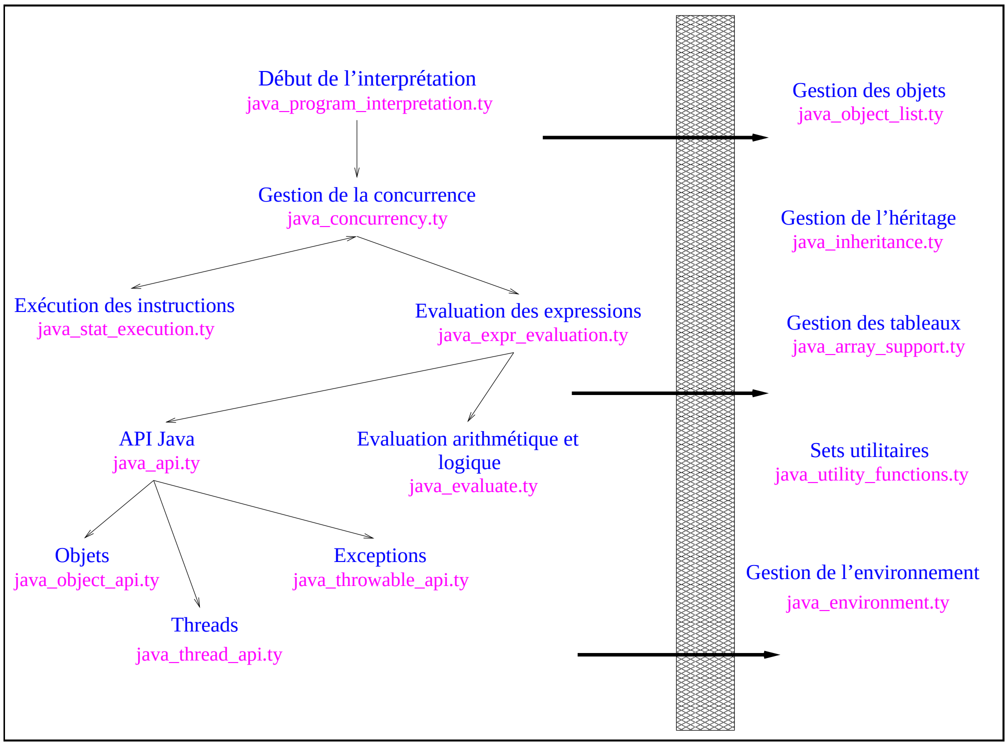

FIG. 3.10 – *Modules de la spécification sémantique de Java.*

  - *java_concurrency.ty* : détermine quel thread va s'exécuter au prochain pas d'exécution, et lance l'exécution d'une instruction atomique de ce thread.
  - *java_api.ty* : se charge de détecter quelle API est appelée et fait appel au module correspondant. Nous détaillons au sein de ce chapitre la spécification des méthodes des API liées à la concurrence et à la synchronisation.
- Modules "utilitaires" :
  - *java_object_list.ty* : crée les différents objets et threads, et gère les différents accès à la liste des objets que ce soit pour lecture ou pour modification.
  - *java_inheritance.ty* : traite les différents aspects liés à l'héritage, comme par exemple la détermination des attributs d'un objet, ou de la méthode appelée.
  - *java_environment.ty* : gère la transmission des différentes informations à l'environnement, par exemple pour mettre les différentes vues à jour après une modification de la liste des objets.

La spécification des principaux modules présentés ici est décrite dans les prochaines sections de ce chapitre.

3.2.2 Des commentaires structurés pour les règles Typol

Dans la suite de ce chapitre, nous employons au sein des sets et des règles Typol des commentaires suivant une syntaxe précise, dont voici la légende :

- `-SC` (*Set Comment*) représente un commentaire qui concerne le set Typol dans son ensemble, son utilisation et son action.
- `-JC` (*Judgment Comment*) représente un commentaire sur le jugement principal du set.
- `-iJC` (*internal Judgment Comment*) représente un commentaire sur un jugement interne au set, c'est-à-dire qui ne sera pas utilisé à l'extérieur du set.
- `-RC` (*Rule Comment*) représente un commentaire sur la règle figurant juste après le commentaire.
- `-CS` (*Concrete Syntax*) donne la syntaxe abstraite correspondant au sujet de la règle où il figure.
- `-PC` (*Premisse Comment*) indique un commentaire sur une prémisse d'une règle.
- `-c` (*comment*) représente un commentaire sur un aspect précis de la règle où il figure.
- `-BSC` (*Begin Structured Comment*) pour indiquer un commentaire portant sur un ensemble de règles (commentaire structuré), par exemple l'ensemble des règles définissant l'affectation. Cela permet d'écrire un commentaire global sur l'exécution de l'instruction concernée, et de pouvoir par la suite commenter chaque phase de son exécution en commentant chaque règle.
- `-ESC` (*End Structured Comment*) pour indiquer la fin d'un commentaire structuré.

Cette syntaxe permet de reconnaître au premier coup d'œil le style de commentaire que l'on est en train de lire et à quel élément (set, jugement, règle ou prémisse) il se rapporte.

D'autre part, on peut imaginer comme perspective de permettre la génération automatique de documentation à partir de la spécification grâce à ces commentaires.

3.2.3 Système de transitions

La Figure 3.11 présente un set Typol qui spécifie notre système de transitions. Les transitions entre les configurations (ou états) sont spécifiées avec des règles qui décrivent un pas d'exécution d'un thread donné. Ces transitions sont de la forme :

$$< ObjL0, ClVarL0 > \rightarrow < ObjL1, ClVarL1 >$$

où chaque configuration est composée d'un ensemble d'objets (*ObjL*) et d'un ensemble de classes avec leurs variables de classes (*ClVarL*).
Cette transition s'interprète comme suit :
Un système dans une configuration initiale $< ObjL0, ClVarL0 >$ *effectue un pas d'exécution et change sa configuration en* $< ObjL1, ClVarL1 >$.
L'exécution est alors modélisée par une séquence de transitions.

La configuration initiale est composée :

- D'un singleton : le thread principal, qui va exécuter la méthode *main*
- D'un singleton formé du nom de la classe contenant la méthode *main* (classe principale) et de ses variables de classe, ou d'un ensemble vide si la classe principale ne possède pas de variable de classe.

La configuration finale (lorsque l'interprétation du programme sera achevée) sera composée :

- D'un ensemble d'objets : c'est-à-dire de tous les objets créés au cours de l'exécution. Dans le cas d'une exécution "normale", sans levée non récupérée d'exception, tous les threads de cet ensemble seront dans le statut *dead*.
- D'un ensemble de noms de classes et des variables de classe correspondantes.

Ces éléments définissant les configurations (l'ensemble des objets, et l'ensemble des classes avec leurs variables de classes) sont donc présents au sein de la plupart de nos règles, en hypothèse pour la configuration initiale, et en conclusion pour la configuration finale de la règle.

Nous montrons dans la Figure 3.11 comment ces transitions sont exprimées en Typol.
La première règle (règle appelée *InitialisationOfTheInterpreter*) du set *interpretProgram* initialise l'exécution du programme : elle crée le thread principal (celui qui va exécuter la méthode *main*) qui devient *executing* et appelle la règle récursive (la règle appelée *RecursiveRule_InterpretOneStep*). Il est important de noter que le sujet de cette règle est constitué de l'opérateur `compilationUnit` qui est en fait l'arbre de syntaxe abstraite correspondant au texte du programme à interpréter. Ce programme est alors stocké dans une variable globale, ce qui permettra de le réutiliser au cours de l'interprétation, par exemple quand on recherchera le source d'une méthode, sans pour autant le faire figurer au niveau des hypothèses au sein de chaque séquent.

```
set interpretProgram is
--SC: Loads the classes (retrieves the class variables), creates the "main"
--SC: thread (the thread which executes the "main" method) and launches the
--SC: program interpretation.
--
--JC: |- Compilation Unit (i.e. the program abstract syntax tree)
--JC: -> Final list that contains all the objects created during this
--JC: interpretation, Final class variable list.
judgement |- CompilationUnit -> Objects, ClassVariablesL;
--iJC: Objects, Class variables
--iJC: -> updated object list, updated class variable list
judgement (Objects, ClassVariablesL -> Objects, ClassVariablesL);

initialisationOfTheInterpreter:
    loadClasses(TDecL -> ClVarL1)  &  getMainThread(|- TDecL -> Thr)  &
    setFirstExecutingThread(objectId(true(), objectNumber 0))  &
    (objects[Thr], ClVarL1 -> ObjL, ClVarL1_1)
    ----------------
    |- compilationUnit(PackName, ImportL, TDecL) -> ObjL, ClVarL1_1 ;
            provided
                ENVgetUserAction()  &  initialize()  &
                setCompilationUnit(compilationUnit(PackName, ImportL, TDecL))
                &
                setNewCompilationUnit(compilationUnit(
                                        PackName, ImportL, TDecL));

RecursiveRule_InterpretOneStep:
    getTheNewExecutingThread(ClVarL1 |- ObjL1 -> ObjL1_1, OThId)  &
    executeThread(ObjL1_1, ClVarL1, OThId -> ObjL1_2, ClVarL1_1)  &
    (ObjL1_2, ClVarL1_1 -> ObjL1_3, ClVarL1_2)
    ----------------
    (ObjL1, ClVarL1 -> ObjL1_3, ClVarL1_2) ;
            provided ENVgetUserAction();

DeadLock:
    --RC : No thread can execute itself but there is at least one alive thread.
    --RC : Deadlock.
    areAllThreadsDead(ObjL -> false())
    ----------------
    (ObjL, ClVarL -> ObjL, ClVarL) ;
            do deadlock()  &  ENVfreeEnvironment();

InterpretationEnd:
    --RC : All threads are dead => end of the program interpretation.
    areAllThreadsDead(ObjL -> true())
    ----------------
    (ObjL, ClVarL -> ObjL, ClVarL) ;
            do ENVfreeEnvironment();
end interpretProgram;
```

Fig. 3.11 – *Notre système de transitions en Typol.*

Dans cette deuxième règle, le set *getTheNewExecutingThread* détermine de manière déterministe un thread à exécuter, l'appel du set *executeThread* effectue alors un pas d'exécution du thread précédemment choisi, et enfin le set courant (dont le nom est dans ce cas implicite) est appelé récursivement.
Les deux dernières règles correspondent à la fin de l'exécution, soit parce qu'il y a *deadlock*, c'est-à-dire qu'aucun thread ne peut plus s'exécuter, mais qu'au moins un des threads n'a pas fini son exécution, soit parce qu'il n'y a plus de thread à exécuter, la configuration considérée est alors la configuration finale (elle est renvoyée sans être modifiée).

La modularité de notre spécification permet comme on l'a vu dans cette section de préserver les différents concepts de Java, et favorise par ailleurs une bonne compréhension de notre sémantique. La présentation du système de transitions a posé les bases de notre spécification sémantique dont les principaux aspects sont développés dans les sections suivantes.

3.3 Objets, héritage et appels de méthodes

Cette section s'intéresse plus particulièrement à la spécification sémantique des aspects reliés à l'orientation objet. Les aspects concurrents seront traités dans la section suivante.
Même si cette section ne concerne pas la concurrence, il est important de garder à l'esprit certains impératifs dus à cette dernière. En effet, même si notre spécification est modulaire du point de vue de l'ajout de la spécification de nouvelles notions, cette modularité ne permet pas de passer d'un style sémantique (comme la Sémantique Naturelle) à un autre (comme la Sémantique Opérationnelle). Ce passage demande une modification complète des règles : pour passer d'une sémantique *big-step* à une sémantique *small-step*, il est nécessaire de fractionner chaque exécution d'instruction. De même, l'interprétation d'instructions “simples” du langage Java peut donner lieu à la création de continuations afin de permettre une simulation de la concurrence par un entrelacement des threads. Il faut donc prévoir de spécifier le langage directement en sémantique *small-step*, même si dans un premier temps, les aspects totalement concurrents ne sont pas définis.

Les caractéristiques principales des langages à objets sont la présence d'objets, la notion d'héritage entre classes et les appels de méthodes. Ces différents aspects sont successivement étudiés au sein de cette section.

3.3.1 Héritage

Comme cela est présenté dans la Section 3.2.1, un module de notre spécification est consacré à la gestion de l'héritage. Plus précisément, nous nous intéressons au parcours du graphe d'héritage afin de déterminer principalement :

- Les attributs d'un objet,

– La méthode à appeler et son code source.

La recherche de la hiérarchie d'héritage entre les différentes classes du programme se fait directement à partir du source de celui-ci (sous forme d'un arbre de syntaxe abstraite). En effet, nous avons choisi de ne pas stocker de structures intermédiaires (par exemple une table des symboles) afin d'éviter d'avoir à effectuer une passe "préparatoire" sur le programme avant de pouvoir lancer véritablement son interprétation. Nous ne construisons donc pas ce graphe mais le reconstituons en parcourant le code du programme. Cependant, cela présente un inconvénient, cela nous oblige à refaire parfois plusieurs fois la même recherche, surtout lors d'un appel récursif de méthode. Dans ce dernier cas, notre sémantique effectue cette recherche à chaque appel, étant donnée l'absence de structure mise en mémoire.
La condition d'arrêt lors de toute remontée dans le graphe d'héritage est soit l'obtention de la structure recherchée (par exemple une classe, ou une méthode), soit l'arrivée au niveau de la classe *Object* qui constitue la racine de tout graphe d'héritage. Dans ce cas, la recherche continue au sein des différentes API. Si elle échoue encore, cela signifie qu'il y a une erreur au niveau du programme car l'attribut ou la méthode recherché n'a pas été trouvé. Ce cas n'est pas sensé se produire car nous prenons comme hypothèse que les programmes à interpréter sont corrects du point de vue de la sémantique statique.
Cette recherche est liée au typage statique : selon le type statique de la variable référençant un objet, cela peut modifier la recherche d'un attribut de cet objet. Notre sémantique ne prend pas en compte cet aspect statique : nous avons en effet choisi de nous concentrer sur les aspects purement dynamiques du langage, en particulier sur la concurrence. Certains mécanismes liés au typage statique des variables sont d'ailleurs assez peu utilisés comme le masquage des attributs (ce dernier point est cependant étudié dans la Section 3.6). Par contre nous avons spécifié la surcharge des attributs et des méthodes.

La Figure 3.12 présente le principal set (appelé `getFieldDeclaration`) permettant de retrouver un membre donné d'une classe à partir de la liste des objets (*ObjL*), du nom de la classe (*ClName*) dans laquelle on va le chercher, d'une liste éventuelle d'arguments (*ArgL*) (dans le cas d'une recherche de méthode), et bien entendu du nom du membre (*FName*) à retrouver. Ce set comporte deux règles, qui représentent en fait les deux cas pouvant se produire : soit le membre cherché se trouve dans la classe courante (première règle), soit il faut le chercher dans l'ancêtre de cette classe (deuxième règle).
Ces deux règles présentent des points communs. En effet, les deux premières prémisses sont quasiment identiques. Cela est du à une caractéristique de Typol : on ne doit pas tenir compte de l'ordre des règles dans leur écriture. C'est à nous d'assurer l'exclusivité entre les règles Typol d'un même set. Cela possède un inconvénient majeur : la deuxième règle doit vérifier qu'elle s'applique bien en exécutant les deux prémisses de la première règle (en attendant un échec) alors qu'elles viennent de conduire à un échec lors de la tentative d'application de la première règle.
Cela complique l'écriture des règles si l'on souhaite éviter cet écueil (qui se traduit par un allongement de la durée d'interprétation d'un programme). Cela est dû à l'absence

```
set getFieldDeclaration is
--SC: Retrieves the field declaration for the given field. Returns false if
--SC: it is not found.
--
--JC: Objects, Current class name where the fields is sought, Field arguments
--JC: (useful to find the right method or constructor declaration),
--JC: True (for a constructor declaration)/ False (for an attribute or method
--JC: declaration)
--JC: |- Field name -> Field declaration, Name of the class that owns this field (can be
--JC: a different name from the given one. In this case, it is
--JC: an ancestor), True if found/False otherwise.
judgement
    Objects, Identifier, Vals, Bool |- Identifier -> FieldDeclaration, Identifier, Bool;

FieldInTheGivenClass:
    getClassDeclaration(|- ClName -> ClDec)  &
    getFieldDeclarationInClass(ObjL, FName, ArgL, TF |- ClDec -> FDec, true())
    ----------------
    ObjL, ClName, ArgL, TF |- FName -> FDec, ClName, true() ;

FieldInAnAncestorNotInTheGivenClass:
    getClassDeclaration(|- ClName1 -> ClDec)  &
    getFieldDeclarationInClass(ObjL, FName, ArgL, TF |- ClDec -> _, false())  &
    getDirectAncestor(|- ClName1 -> Ancestor)  &
    getFieldDeclarationInAncestors(ObjL, Ancestor, ArgL, TF
                                            |- FName -> FDec, ClName2, true())
    ----------------
    ObjL, ClName1, ArgL, TF |- FName -> FDec, ClName2, true() ;
end getFieldDeclaration;
```

FIG. 3.12 – *Gestion de l'héritage lors de la recherche d'un membre d'une classe.*

du phénomène de coupure qui interdirait de tenter d'appliquer la seconde règle si la première s'appliquait.

La deuxième règle est la plus intéressante du point de vue de l'héritage. C'est donc elle que nous allons analyser ici. Tout d'abord, la première prémisse appelle un set (`getClassDeclaration`) permettant d'extraire la déclaration de la classe (au sein de laquelle le membre va être recherché) de l'arbre de syntaxe abstraite. Puis on utilise le set `getFieldDeclarationInClass`, qui effectue la recherche proprement dite.
Le set `getFieldDeclarationInClass` analyse la déclaration de la classe donnée en parcourant la liste de ses membres et de ses constructeurs afin de retrouver le membre cherché. La condition d'arrêt de cette recherche est soit son succès, soit l'arrivée à la fin de la liste des membres et des constructeurs. Le *false()* que l'on trouve en conclusion de cette seconde prémisse permet d'indiquer qu'en fait le membre recherché n'a pas été trouvé : si la conclusion renvoyée par l'appel de ce set vaut *true()*, la règle conduira directement à un échec. On remonte alors dans le graphe d'héritage pour trouver l'ancêtre de la classe courante (à l'aide du set `getDirectAncestor`), et on poursuit la recherche dans la classe trouvée ainsi que dans ces ancêtres si cela est nécessaire (set `getFieldDeclarationInAncestors`).

Il est intéressant de signaler que, contrairement à d'autres langages comme Eiffel, la notion d'héritage en Java est simplifiée par l'absence d'héritage multiple. Cela permet d'éviter de parcourir de multiples branches du graphe d'héritage lorsque l'on recherche successivement tous les ancêtres d'une classe donnée par exemple. Cependant, les règles détaillées dans cette section seraient très facilement utilisables dans le cas d'un langage avec héritage multiple : il suffirait d'ajouter des règles permettant de parcourir les différentes branches de l'arbre d'héritage, et d'utiliser au sein de chacune les sets déjà définis.
En ce qui concerne les interfaces, dans le cadre de notre sémantique dynamique nous ne nous intéressons qu'aux déclarations concrètes d'éléments (attributs ou méthodes) afin d'obtenir, surtout dans le cas des méthodes, le code correspondant, et non aux déclarations abstraites. Les recherches de ces éléments ne se font donc qu'au sein du graphe d'héritage des classes.

3.3.2 Création des objets

La création d'un objet est un élément essentiel dans le cadre des langages à objets. Avant même le début de l'exécution des instructions du programme, elle intervient dans le cadre de la création du thread principal, qui est créé de manière implicite pour exécuter la méthode *main*.
En Java, l'instruction de création d'un objet s'écrit *new ClassName()*, et on la retrouve le plus souvent dans la forme *newObject = new ClassName()* qui permet d'associer une variable (*newObject*) à l'objet ainsi créé. Il s'agit d'une expression complexe qui s'exécute en fait en deux étapes successives (chacune pouvant prendre plusieurs pas d'exécution

```
NewObjectCreation:
    --RC: Creates an instance of the given class and adds this instance to the
    --RC: object list that contains all the objects created by the current
    --RC: interpreted program. The next execution step of the current
    --RC: executing thread will be the constructor call.
    --
    --CS: new Thread();
    generateANumberForTheNewObject(|- ObjL1 -> i)  &
    createObject(OThId |- i, TVClName -> Obj, ObjId)  &
    appendtree(ObjL1, objects[Obj], ObjL1_1)
    ----------------
    ObjL1, ClVarL, Env, OThId, _, _
        |- newObject(new(), TVClName, ArgL, fieldDeclarations[]) ->
            ObjL1_1, ClVarL, Env, ObjId,
            insts[constructorCall1(TVClName, ArgL, ObjId)], runnable() ;
```

FIG. 3.13 – *Création d'un objet (règle générale).*

du thread courant) :

1. Création de l'objet (expression *new ClassName()*) : elle s'effectue en trois sous-étapes
   (a) Détermination de la référence interne de l'objet à créer : c'est le nombre d'objets présents dans la liste d'objets avant sa création.
   (b) Création proprement dite de l'objet
   (c) Appel ascendant des constructeurs.
2. Affectation de la référence de l'objet créé (qui pour l'instant n'est pas accessible) à la variable `newObject` (expression *newObject* = ...).

La première étape (1) est spécifiée par la règle Typol présentée en Figure 3.13. Les deux premières sous-étapes (a et b) sont représentées par les deux premières prémisses de la règle (appels des sets `generateANumberForTheNewObject` et `createObject`), la troisième prémisse est l'appel d'une primitive Prolog (*appendtree*) permettant d'ajouter le nouvel objet à la liste des objets déjà créés. La troisième sous-étape (c) est indiquée sous forme de continuation par la partie grisée de la règle. Cet élément de la conclusion de la règle représente la liste des instructions à ajouter en tête de l'activité du thread courant. Elle indique que lorsque le thread courant s'exécutera à nouveau, sa première action sera l'appel `constructorCall1(TVClName, ArgL, ObjId)` du constructeur correspondant à la classe `TVClName` dont l'objet créé `ObjId` est une instance. On parle d'appel ascendant, car cet appel va remonter dans le graphe d'héritage de cette classe afin de commencer par appeler le constructeur de la racine du graphe d'héritage pour redescendre successivement jusqu'à la classe considérée.
La deuxième étape (2) est exécutée après les appels successifs aux constructeurs. Il s'agit d'une simple affectation.

```
SimpleObjectCreation_GetAttributeList:
   makeAttributeList(|- ClName -> AttrL)
   ----------------
   (ObjId, ClName -> object(ObjId, ClName, AttrL, unlocked(), waits[], empty())) ;
```

FIG. 3.14 – *Création d'un objet simple.*

Revenons à la création proprement dite de l'objet (étape 1, sous-étape b). Celle-ci s'exprime en sémantique naturelle par la règle Typol présentée en Figure 3.14, ce qui signifie que cette création s'effectue de façon atomique, en un seul pas d'exécution.
Nous avons été amenés à considérer trois catégories d'objets :

- Les objets simples : qui ne sont pas des threads.
- Les threads : qui sont des objets ayant une activité propre décrite dans la méthode *run* de la classe dont ils sont une instance.
- Le thread principal : qui est créé de façon implicite afin d'exécuter la méthode *main*.

La règle présentée ne porte que sur la création d'un objet simple. Le cas des threads sera traité en Section 3.5.

Une phase de la création est commune à ces trois catégories d'objets, il s'agit de la détermination de l'ensemble des attributs de l'objet. Cette phase (qui est l'unique phase lors de la création d'un objet simple comme on le voit sur le Figure 3.14), présentée en Figure 3.15, fait appel à un parcours du graphe d'héritage afin de déterminer l'ensemble complet des attributs (hérités et non hérités). Le set `makeAttributeList` contient deux règles, la première traite les objets instances d'une classe du programme, la deuxième quant à elle gère le cas des objets instances d'une classe d'une bibliothèque. Considérons la première règle. Le sujet de la règle est le nom de la classe dont l'objet est une instance. La première étape consiste à récupérer la déclaration de celle-ci (appel du set `getClassDeclaration` qui parcourt l'arbre de syntaxe abstraite du programme à la recherche de la bonne classe). L'ensemble des attributs locaux à cette classe est alors déterminé lors de l'appel du set `getLocalAttributeList`. Le set `getDirectAncestor` permet d'obtenir l'ancêtre direct (ou super-classe) de la classe courante par remontée du graphe d'héritage. Le set `makeInheritedAttributeList` se charge alors de déterminer l'ensemble total des attributs de l'objet à créer en parcourant le graphe d'héritage. L'ensemble des attributs locaux à une classe est obtenu en analysant tous les composants (membres et constructeurs) de la classe. Plusieurs cas peuvent se présenter :

- déclaration de méthode ou de constructeur : passage au composant suivant
- déclaration d'une variable de classe (mot-clé *static*) : passage au composant suivant
- déclaration d'un attribut : création de la structure sémantique correspondante avec comme valeur initiale :
  - attribut sans valeur d'initialisation : la connaissance de son type permet

```
set makeAttributeList is
--SC: Determines the attribute list correponding to a given type (useful for
--SC: an object creation).
--
--JC: |- Class name of the new instance
--JC: -> Initialized Attribute list
judgement |- OptTypeName -> Pairs;

DeclarationObject_InProgramClasses:
   getClassDeclaration(ClName1 -> ClDec)  &
   getLocalAttributeList(|- ClDec -> LocAttrL)  &
   getDirectAncestor(|- ClName1 -> ClName2)  &
   makeInheritedAttributeList(LocAttrL |- ClName2 -> AttrL)
   ----------------
   |- ClName1 -> AttrL ;

DeclarationObject_inAPI:
   getAPIObjectAttributes(|- ClName -> AttrL)
   ----------------
   |- ClName -> AttrL ;
end makeAttributeList;
```

FIG. 3.15 – *Obtention de l'ensemble des attributs d'un objet.*

d'attribuer une valeur par défaut

– attribut avec valeur d'initialisation : celle-ci prend la place de la valeur par défaut.

Cette phase d'obtention de l'ensemble des attributs est commune à tous les objets.

Un tel mécanisme de spécification de la création des objets, même s'il est spécifique au langage Java, est facilement adaptable à d'autres langages à objets en ce qui concerne l'obtention des attributs de l'objet, et concurrents en ce qui concerne la création des threads. Si l'on considère le langage Eiffel// [9], celui-ci possède aussi un objet actif principal exécutant la routine *create* et des objets actifs autres, ainsi que des objets inactifs. On reconnaît là les trois types d'objets que nous distinguons dans notre spécification du langage Java.

3.3.3 Appels de méthodes

Java étant un langage à objets, les appels de méthodes constituent un point clé des programmes après celui de la création proprement dite des objets.
L'exécution d'un appel de méthode (qui est considéré comme une expression en Java) peut être fractionnée en quatre étapes (chacune pouvant représenter plusieurs pas d'exécution du thread courant) :

1. Détermination de l'objet sur lequel la méthode est appelée

2. Détermination de la valeur des arguments (ils sont évalués les uns après les autres de gauche à droite)
3. Détermination de la méthode à appeler et construction de la closure correspondante
4. Exécution proprement dite de la méthode.

Cette section va étudier successivement la spécification de ces quatre étapes (nous nous restreignons ici aux méthodes d'instance, qui porte sur un objet par opposition aux méthodes de classe).

3.3.3.1 Détermination de l'objet

La détermination de l'objet sur lequel la méthode est appelée peut paraître simple si l'on considère les appels de base comme `obj.MethodName(ArgL);` où `obj` est une variable référençant un objet. Par contre, cela peut s'avérer complexe dans le cas d'un appel de méthode construit sur le modèle :

```
obj.MethodName1(ArgL1).MethodName2(ArgL2)...MethodNameN(ArgLN);
```

Nous avons donc été amenés à écrire des règles générales permettant de traiter ces différents cas. Le mécanisme principal en est le suivant : on remplace l'expression :

```
obj.MethodName1(ArgL1).MethodName2(ArgL2)...MethodNameN(ArgLN);
```

par la liste d'expressions :

```
obj.MethodName1(ArgL1);
7RETURN.MethodName2(ArgL2)...MethodNameN(ArgLN);
```

Dans cette liste d'expressions, `7RETURN` représente une variable interne à notre sémantique (une variable Java ne pouvant commencer par un chiffre, nous sommes sûrs de ne pas créer de conflits) utilisée pour représenter la valeur de retour d'une méthode. La première expression :

```
obj.MethodName1(ArgL1);
```

réalise l'appel de la méthode `MethodName1` sur l'objet `obj`. La valeur de retour de cette méthode sera affectée de façon interne à notre sémantique à la variable `7RETURN`. Cette variable sera à son tour évaluée lors de l'évaluation de l'expression :

```
7RETURN.MethodName2(ArgL2)...MethodNameN(ArgLN);
```

afin de déterminer sur quel objet doit être appelée la méthode `MethodName2`.
L'expression :

```
obj.MethodName1(ArgL1).MethodName2(ArgL2)...MethodNameN(ArgLN);
```

et la liste d'expressions :

```
obj.MethodName1(ArgL1);
7RETURN.MethodName2(ArgL2)...MethodNameN(ArgLN);
```

sont donc équivalentes du point de vue de l'exécution.

Ce mécanisme est mis en oeuvre par la règle Typol présentée en Figure 3.16.
Cette règle s'applique lorsque le sujet est de la forme :

```
call(access(call(Expr1, ArgL1), MethodName), ArgL2)
```

```
MethodCall_TargetEvaluationWithAnotherMethodCall:
    --RC: Before executing the given method, the target object represented
    --RC: here by a method call must be retrieved. Therefore this method call
    --RC: is evaluated. Moreover, a special variable called 7RETURN is added to
    --RC: the local variable list in order to handle the value returned by
    --RC: this method.
    --
    --CS: myObject.bar().setAttrValue(var2 - var1);
    evaluateExpression(ObjL1, ClVarL1, Env, OThId, ObjId, Mode
                           |- call(Expr1, ArgL1) ->
                                  ObjL1_1, ClVarL1_1, env(ParamL1, LocVarL1), _,
                                  InstL1, ThStatus)   &  diff(InstL1, insts[])   &
    appendtree(InstL1, insts[call(access(identifier "7RETURN", MethName), ArgL2)],
                 InstL1_1)   &
    addReturnVariableInLocalVariableList(LocVarL1 -> LocVarL1_1)
    ----------------
    ObjL1, ClVarL1, Env, OThId, ObjId, Mode
        |- call(access(call(Expr1, ArgL1), MethName), ArgL2) ->
             ObjL1_1, ClVarL1_1, env(ParamL1, LocVarL1_1), identifier "7RETURN",
             InstL1_1, ThStatus ;
```

FIG. 3.16 – *Règle générale des appels de méthodes.*

c'est-à-dire possède une syntaxe concrète de la forme :

`Expr(ArgL1).MethodName(ArgL2).`

La première prémisse de la règle figurant dans la Figure 3.16 permet d'effectuer un pas d'évaluation de l'expression `call(Expr1, ArgL1)`, la troisième de concaténer la liste d'instructions restant à exécuter obtenue lors de cette évaluation et l'instruction :

`call(access(identifier “7RETURN”, MethodName), ArgL2)`

et la dernière de vérifier que l'on a bien ajouté la variable `7RETURN` à la liste des variables locales. Le prochain pas d'exécution du thread courant permettra d'effectuer un pas d'évaluation de l'expression `call(Expr1, ArgL1)`, et cela continuera jusqu'à l'obtention d'une valeur qui sera affectée à la variable `7RETURN`. Le prochain pas d'interprétation sera alors l'exécution de l'expression :

`call(access(objectId(Bool, objectNumber Nb), MethodName), ArgL2)`

où `objectId(Bool, objectNumber Nb)` représente la référence affectée à la variable `7RETURN` (le booléen indique si l'objet est un thread, valeur *true()* ou non, Nb indique le nom interne de l'objet).

Notre spécification comporte plusieurs autres règles permettant de traiter les autres cas de l'appel de méthode, c'est-à-dire :

- Les appels de méthodes sans indication d'objet (par exemple `MName(...);`). Dans ce cas, la méthode est appelée sur l'objet courant (celui sur lequel a été appelée la méthode que l'on est en train d'interpréter).
- Les appels de méthodes de classes.
- Les appels de la forme `Expr.MName(...)` où `Expr` n'est pas un autre appel de méthode.

```
NonStaticMethodCallWithArguments_FirstArgumentEvaluation:
   --RC: One evaluation step of the first argument is made.
   --
   --CS: #4.myMethod(Obj2.Attr2, 3+6*9, Obj1.getVal());
   evaluateArgument(ObjL1, ClVarL1, Env1, OThId, ObjId1, Mode1
                         |- ArgL1 ->
                                ObjL1_1, ClVarL1_1, Env1_1, ArgL1_1, InstL1)  &
   diff(ArgL1, ArgL1_1)  &
   appendtree(InstL1, insts[methodCall2(Mode2, TVObjId, MethName, ArgL1_1)],
              InstL1_1)
   ----------------
   ObjL1, ClVarL1, Env1, OThId, ObjId1, Mode1
      |- methodCall1(Mode2, TVObjId, MethName, ArgL1) ->
           ObjL1_1, ClVarL1_1, Env1_1, empty(), InstL1_1, runnable() ;
           provided diff(ArgL1, arguments[]);
```

FIG. 3.17 – *Evaluation du premier argument lors d'un appel de méthode.*

– Les appels de méthode conduisant à la levée d'une exception (voir section 3.4), par exemple quand une méthode est appelée sur un objet n'ayant pas été créé.

Pour des raisons évidentes de concision, nous ne les détaillons pas toutes ici. Le lecteur peut se référer à l'Annexe 3 pour étudier l'intégralité des règles spécifiant une sémantique des appels de méthode.

3.3.3.2 Evaluation des arguments

Après avoir déterminé sur quel objet la méthode allait être appelée, l'expression à interpréter est de la forme : `#4.MethName(ArgL1);` (en considérant la syntaxe concrète pour plus de clarté). La syntaxe abstraite de cette expression est :

`methodcall1(Mode2, TVObjectID, MethName, ArgL1)`

où la variable `TVObjectId` représente la référence de l'objet sur lequel la méthode est appelée, le préfixe `TV` indiquant qu'il s'agit d'une variable globale dont le type est fixe (ici, il s'agit d'un type référence). Ce mécanisme de variable globale typée est une des caractéristiques du formalisme Typol. Cette variable typée permet à cette règle de ne s'appliquer que lorsque l'unification entre le type de la valeur et celui de la variable est possible du point de vue du typage. Il est intéressant de remarquer que là aussi, une continuation a été introduite, il s'agit de l'opérateur *methodcall1* afin de pouvoir y intégrer une valeur référence.

La deuxième étape de l'interprétation de l'appel de méthode consiste alors à évaluer les arguments, les uns après les autres, de gauche à droite. La première étape de cette évaluation est spécifiée par la règle représentée par la Figure 3.17. Le sujet de cette règle, qui apparaît en grisé, est exactement l'arbre de syntaxe abstraite décrit précédemment. Cette règle permet d'effectuer un pas d'évaluation du premier argument, et d'introduire une nouvelle continuation, l'opérateur `methodcall2` qui représente les ap-

```
set getMethodClosure is
--SC: Gets the closure corresponding to a method call.
--
--JC: Objects, Class variables
--JC: |- Method name for which the closure is wanted, Object (on which the
--JC: method is called) reference, method parameter values
--JC: -> Closure for the given method call.
judgement Objects |- Identifier, Ref, Vals -> Clr;

getObjectType(ObjId |- ObjL -> ClName)  &
getFieldDeclaration(ObjL, ClName, ValueL, false()
                        |- MethName -> MethDec, _, true())  &
makeClosure(ClName, ValueL, ObjId |- MethDec -> Clr)
----------------
ObjL |- MethName, ObjId, ValueL -> Clr ;
end getMethodClosure;
```

FIG. 3.18 – *Obtention de la closure correspondante à un appel de méthode.*

pels de méthode dont certains arguments ont été évalués et peuvent donc comprendre des valeurs références.

3.3.3.3 Recherche de la méthode à appeler

Une fois que tous les arguments auront été évalués, l'on passe à la troisième étape de la spécification des appels de méthode : la recherche de la méthode à appeler, et la construction de la closure correspondante.

Le set le plus important à considérer est alors le set `getMethodClosure` présenté en Figure 3.18. Ce set permet tout d'abord de rechercher le type de l'objet sur lequel est appelée la méthode, pour ensuite récupérer le source de cette méthode, à l'aide de ce type et de la liste des arguments donnée lors de l'appel, grâce au set `getFieldDeclaration`. La liste des arguments est nécessaire afin d'établir la signature de la méthode appelée. Cette signature sera comparée à celle des méthodes des différentes classes lors de la remontée dans le graphe d'héritage à partir de la classe dont l'objet sur lequel la méthode est appelée est une instance. La troisième prémisse de l'unique règle de ce set permet de construire la closure correspondante au source de la méthode obtenu par la prémisse précédente.

C'est le set `makeClosure` qui a la charge de la construction de cette closure. La Figure 3.19 présente la principale règle de ce set. Cette règle s'applique dans le cas des méthodes non *synchronized* (voir section 3.5.3 pour avoir plus de détails sur la synchronisation), comme le montre la première prémisse de cette règle. Le sujet de cette règle est l'arbre de syntaxe abstraite représentant la déclaration de la méthode à partir

```
NonSynchronizedMethod:
   --RC: Gets the closure for this non synchronized method
   isThisModifierInTheList(synchronized() |- ModL -> false())  &
   getMethodNameAndParameters(TVMethodDecl -> MethName, ParamL)  &
   makeBindedParamList(ParamValueL |- ParamL -> BindParamL)  &
   getModeAndTargetClass(ClName |- ModL -> _, TargetClName)  &
   getInstructionsAndLocalVariables(insts[], locs[pairs[]]
                                   |- MBlock => InstL, LocVarL)
   ----------------
   ClName, ParamValueL, ObjId
      |- methodDeclaration(ModL, _, TVMethodDecl, _, MBlock) ->
            clr(ObjId, TargetClName, MethName, env(BindParamL, LocVarL), InstL) ;
```

FIG. 3.19 – *Construction de la closure.*

Méthode setSides de la classe Square	Closure associée :

void setSides(int s){
setWidth(s);
}
	Closure on the object : X
	Method name : setSides
	Environment :
	Bound parameters : s = Y
	Local variables : none
	Instruction list :
	setWidth(s);

FIG. 3.20 – *Exemple de closure associée à une méthode.*

duquel va être définie la closure.
Cette règle permet tout d'abord de déterminer le nom de la méthode ainsi que ses paramètres (deuxième prémisse), puis effectue le liage entre ces paramètres et les valeurs des arguments (troisième prémisse). La dernière prémisse de cette règle permet d'analyser le bloc de la méthode afin de déterminer la liste des variables locales et les instructions à exécuter. Ces données vont former d'une part l'environnement d'exécution de la méthode (variables locales et paramètres), ainsi que l'activité proprement dite que le thread courant aura à exécuter.
La Figure 3.20 illustre ce mécanisme en montrant la closure obtenue après analyse de l'arbre de syntaxe abstraite associé à la méthode `setSides` de la classe `Square` du programme présenté en Figure 2.1. La partie gauche de cette figure permet de visualiser le source de la méthode, alors que la partie droite montre la closure construite à partir de ce source. Nous présentons ici les données sous forme de syntaxe concrète afin de faciliter la lecture de la figure. Nous utilisons les lettres majuscules `X` et `Y` dans la closure à la place des données qui sont fournies par l'appel de méthode proprement dit : référence de l'objet sur lequel est appelée la méthode (`X`), et valeur donnée en argument lors de l'appel (`Y`).
Lors du prochain pas d'exécution du thread courant, ce dernier va analyser cette closure afin de déterminer quelle instruction exécuter; dans l'exemple de la Figure 3.20, la

prochaine instruction exécutée sera l'appel de méthode `setWidth`.

Les appels de méthode étant un point crucial des langages à objets, leur spécification est complexe. Nous nous sommes efforcés dans cette section de développer la spécification du cas le plus général, qui est aussi le plus courant (par rapport aux appels de méthodes de classe), mais le lecteur intéressé pourra retrouver l'ensemble des règles concernant les appels de méthode au sein du set `evaluateExpression` du fichier *java_expr_evaluation.ty* en Annexe 3.

3.4 Les exceptions

Comme on l'a vu dans le chapitre précédent, les exceptions sont un mécanisme du langage Java permettant de transférer le contrôle de l'exécution du thread courant du point où une contrainte sémantique (comme un pointeur non initialisé ou une division par zéro) a été violée, ou lors d'une instruction directe du programmeur, à un point qui peut être spécifié par le programmeur. Ce point de récupération de l'exception dépend de son type. En effet, toute exception est un objet, instance de la classe Java *Throwable*, ou d'une de ses sous-classes.

Nous modélisons au sein de notre sémantique les principales exceptions de la classe *java.lang.RuntimeException* et de ses sous-classes :

- *NullPointerException* : utilisation d'une variable de type référence non initialisée.
- *ArithmeticException* : exceptions portant sur l'évaluation arithmétique (division par zéro).
- *IllegalThreadStateException* : appel de méthode alors que le thread n'est pas dans le bon statut.
- *IllegalMonitorStateException* : appel de méthode alors que le thread ne possède pas le lock de l'objet sur lequel la méthode est appelée.

Nous ne traitons pas des exceptions appelées *Error* qui sont liées à des problèmes physiques (mémoire ou autre) et non directement à la sémantique du langage Java.

Cette section présente tout d'abord des généralités sur les mécanismes des instructions et la structure sémantique qui leur est associée au sein de notre spécification du langage Java. Les deux autres paragraphes de cette section décrivent une sémantique de deux instructions permettant de manipuler les exceptions en Java, c'est-à-dire les instructions *try...catch...finally...* et *throw*.

3.4.1 Mécanisme général et structures sémantiques

Il existe deux catégories d'exceptions : les exceptions levées automatiquement et les exceptions levées explicitement par l'appel de l'instruction *throw*. Dans les deux cas, le mécanisme déclenché est identique. Ses principales étapes sont :

1. Récupérer le type de l'exception.

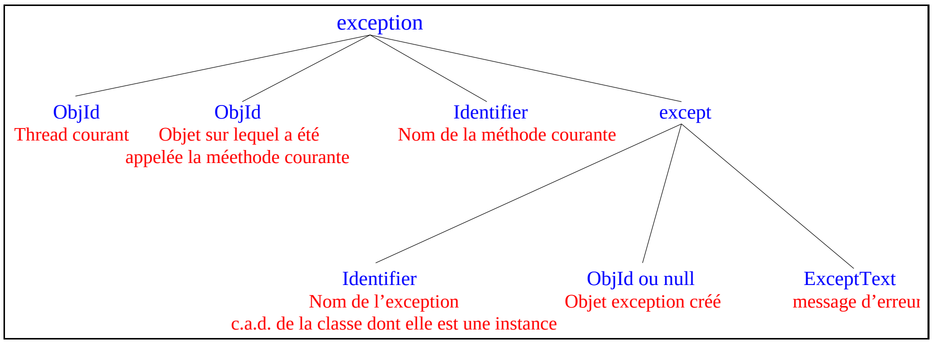

FIG. 3.21 – *Structure sémantique d'une instruction exception.*

2. Relâcher les verrous en cours s'il y en a (au sein du bloc courant).
3. Si on est dans un bloc *try*, rechercher un bloc *catch* convenable du point de vue du type de l'exception :
   – S'il y a un *catch* convenable, on exécute le bloc *catch* puis le bloc *finally*.
   – S'il n'y en a pas, on exécute le bloc *finally* et on propage l'exception.
4. Effacer la suite des instructions du bloc courant.

Afin de spécifier ce mécanisme, nous avons introduit dans notre spécification une structure sémantique modélisant une instruction de lancement d'une exception. L'arbre de syntaxe abstraite de cette structure sémantique est présenté en Figure 3.21. Cette structure sémantique permet de connaître le contexte d'exécution au moment où l'exception a été lancée : elle indique le thread courant, la méthode qui était en train d'être exécutée et sur quel objet elle l'était. D'autre part, cette structure sémantique comporte des informations sur l'exception elle-même (sous-arbre *except*) : son type, c'est-à-dire la classe dont l'objet exception créé est une instance, la référence de cet objet, ainsi qu'un message d'erreur qui pourra aider le programmeur.

Quand une exception est levée, la première étape est de créer l'objet exception qui lui est associé. Cette création est identique à celle d'un objet simple, que ce soit au niveau de la création proprement dite ou au niveau des appels ascendants des constructeurs associés. Prenons l'exemple d'une tentative d'affectation d'une expression à l'attribut d'un objet référencé par *null* (c'est-à-dire sans référence), comme cela est présenté en Figure 3.22.

Le fait de tenter d'accéder à l'objet représenté par *null* constitue une violation de la sémantique de Java, et déclenche par l'appel du set `createExceptionObject` la levée d'une exception . Ce set va non seulement créer l'objet représentant l'exception, mais aussi permettre d'obtenir la liste d'instructions à placer en tête des instructions à exécuter par le thread courant lors de son prochain pas d'exécution. Cette liste se compose d'une part de l'appel du constructeur associé à la classe dont l'objet exception est une instance, et d'autre part de l'instruction de levée de l'exception qui est présentée plus haut.

```
ComplexAssignExpressionViaNull:
    --RC: Attempt to assign via a null reference to an object.
    --RC: Throw an exception.
    --
    --CS: null().Attr = Var1 * Var2;
    createExceptionObject(ObjL1_1, OThId
                                |- identifier "NullPointerException",
                                   exceptText "Use of a pointer with a null value."
                                       ->
                                       _, ObjL1_2, InstL)
    ----------------
    ObjL1, ClVarL1, Env1, OThId, ObjId1, Mode
        |- semAssign(null(), _, _, _) ->
               ObjL1_2, ClVarL1_1, Env1_1, empty(), InstL, runnable() ;
```

FIG. 3.22 – *Exemple de levée d'une exception suite à une violation d'une contrainte sémantique.*

Lorsqu'une exception est levée, deux possibilités se présentent. Soit rien n'est prévu pour la récupérer, et l'exécution du thread courant est arrêtée après relâchement des locks qu'il possède, soit elle est récupérée par une clause *catch* comme nous allons l'étudier dans le paragraphe suivant.

3.4.2 L'instruction *try...catch...finally...*

Une instruction Java est directement associée aux exceptions. Il s'agit de l'instruction *try* ... *catch* ... *finally* ... qui se présente comme suit :

- Bloc *try* : bloc d'instructions contenant le code que l'on veut exécuter.
- Blocs *catch* : chaque bloc *catch* prend comme paramètre un type d'exception et une variable référence. Il contient d'autre part un bloc d'instructions.
- Bloc *finally* : bloc d'instructions qui sera exécuté qu'il y ait levée d'une exception ou non.

Sa sémantique informelle est la suivante. Lors de l'exécution du bloc *try*, deux cas se présentent :

- Aucune exception est levée. Dans ce cas, le bloc *finally* est exécuté directement à la sortie du bloc *try*.
- Une exception est levée au sein du bloc *try*. Dans ce cas, il faut chercher si le type de l'exception levée est compatible avec le type du paramètre de l'un des blocs *catch*. Là encore, deux possibilités :
  - Un tel bloc est trouvé. Ce bloc (le premier qui convient dans la liste des blocs *catch*) est exécuté, le paramètre du bloc se voyant affecter la référence de l'objet exception venant d'être créé. Le bloc *finally* est exécuté à la suite de ce bloc *catch*.

- Aucun bloc catch n'a un paramètre dont le type est compatible avec celui de l'exception levée. Dans ce cas, le bloc *finally* est exécuté. Deux possibilité peuvent apparaître à ce niveau :
  - Ce bloc s'exécute sans levée d'une autre exception et l'on remonte dans la hiérarchie d'inclusion des blocs jusqu'à ce qu'un bloc *catch* compatible avec le type de l'objet exception créé au sein du bloc *try* soit trouvé.
  - L'exécution du bloc *finally* lève une autre exception. Dans ce cas, l'on remonte dans la hiérarchie d'inclusion des blocs jusqu'à ce qu'un bloc *catch* avec un paramètre ayant un type compatible avec celui de l'objet exception créé au sein du bloc *finally* soit trouvé. L'exception levée au sein du bloc *try* est alors oubliée.

Dans ce qui précède, nous appelons remontée dans la hiérarchie d'inclusion des blocs le fait de passer du bloc courant au bloc dans lequel il est directement inclus. Par exemple si une exception est levée au sein d'une méthode sans être récupérée par une clause *catch* au sein de celle-ci, l'exécution se poursuit en propageant l'exception au sein du bloc dans lequel a eu lieu l'appel de la méthode courante. A chaque propagation, les locks acquis au sein du bloc dont on abandonne l'exécution sont relâchés avant cette propagation.
D'autre part, on dit que le type `TC` du bloc *catch* est compatible avec celui `TE` de l'objet exception lorsque la classe `TE` est soit la classe `TC`, soit une de ses sous-classes.

Comme nous pouvons le voir, la sémantique informelle de cette instruction est assez complexe. Cette complexité se retrouve au sein de notre sémantique. En effet, le traitement des différents cas a nécessité de nombreuses règles, notamment dans le cas de la levée d'une exception au sein du bloc *try*, et de la propagation de celle-ci au sein de la hiérarchie d'inclusion. Nous ne pouvons détailler ici les différentes règles concernées, mais elles sont toutes disponibles en Annexe 3, principalement dans le fichier `java_stat_execution.ty`.

3.4.3 L'instruction *throw*

Comme nous l'avons vu précédemment, une exception peut être levée soit à cause d'une erreur de programmation entraînant une violation de la sémantique du langage, soit de manière explicite par une instruction du programmeur. Cela est possible par l'utilisation de l'instruction *throw*. Les règles Typol spécifiant une sémantique de cette instruction sont présentées en Figure 3.23. Nous avons été amenés à utiliser trois règles afin de permettre l'entrelacement au niveau de l'exécution des différents threads. Les deux dernières règles permettent l'évaluation de l'expression donnée en argument de l'instruction *throw* dont la syntaxe abstraite est représentée par l'opérateur *throw*, puis par l'opérateur *semThrow*, ce dernier permettant de traiter la fin de l'évaluation, c'est-à-dire lorsque l'argument de l'instruction est la référence d'un objet exception. La première règle s'applique lorsque l'évaluation de l'argument est terminée, ce dernier étant alors la référence d'un objet exception.

```
ThrowExpression_EvaluatedExpression:
   --BSC: Throw Statement
   --
   --CS: throw #4
   getObjectType(TVObjId |- ObjL -> ClName)
   ----------------
   ObjL, ClVarL, OThId
      |- clr(ObjId, Mode, MethName, Env, insts[semThrow(TVObjId).InstL]) ->
           ObjL, ClVarL,
           clr(
              ObjId, Mode, MethName, Env,
              insts[except(ClName, TVObjId, exceptText "thrown exception").InstL]),
           runnable() ;

ThrowExpression_EvaluateExpression:
   --CS: throw Expr;
   evaluateExpression(ObjL1, ClVarL1, Env1, OThId, ObjId, Mode
                     |- Expr1 ->
                          ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL2,
                          ThStatus) &
   appendtree(InstL2, insts[semThrow(Expr1_1).InstL1], InstL3)
   ----------------
   ObjL1, ClVarL1, OThId
      |- clr(ObjId, Mode, MethName, Env1, insts[semThrow(Expr1).InstL1]) ->
           ObjL1_1, ClVarL1_1, clr(ObjId, Mode, MethName, Env1_1, InstL3),
           ThStatus ;

ThrowExpression_EvaluateExpression:
   --CS: throw Expr;
   --
   --ESC: Throw Statement
   evaluateExpression(ObjL1, ClVarL1, Env1, OThId, ObjId, Mode
                     |- Expr1 ->
                          ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL2,
                          ThStatus) &
   appendtree(InstL2, insts[semThrow(Expr1_1).InstL1], InstL3)
   ----------------
   ObjL1, ClVarL1, OThId
      |- clr(ObjId, Mode, MethName, Env1, insts[throw(Expr1).InstL1]) ->
           ObjL1_1, ClVarL1_1, clr(ObjId, Mode, MethName, Env1_1, InstL3),
           ThStatus ;
```

FIG. 3.23 – *Sémantique de l'instruction throw.*

```
NonCaughtExceptionInRunMethod:
    --RC: As the exception is uncaught, the thread execution
    --RC: is aborted (empty instruction list) and the exception
    --RC: is sent to the environment.
    ObjL, ClVarL, OThId
        |- clr(
                OThId, Mode, identifier "run", Env,
                insts[
                    exception(
                        OThId, ObjId1, MethName, except(ClName, ObjId2, ExceptText)).._]) ->
                ObjL, ClVarL, clr(OThId, Mode, identifier "run", Env, insts[]), runnable() ;
                do
                    ENVsendException(exception(
                                            OThId, ObjId1, MethName,
                                            except(ClName, ObjId2, ExceptText)));
```

FIG. 3.24 – *Fin de l'exécution d'un thread suite à une exception non récupérée.*

La Figure 3.24 présente la règle Typol qui termine l'exécution du thread courant lorsque son exécution a levé une exception qui n'a pas été récupérée. Cette règle s'applique dans tous les cas, quelle que soit la raison de la levée de l'exception. L'interprétation de l'exception qui est en tête de la liste des instructions de la méthode *run* du thread courant permet de comprendre qu'elle n'a pas été récupérée et ne peut plus l'être (car elle n'est pas incluse dans un bloc *try*). La liste des instructions restant à exécuter est donc rendue vide, l'exécution du thread courant va alors être stoppée. Il est intéressant de noter la notification vers l'environnement (appel du set `ENVsendException`), qui va permettre de renseigner le programmeur sur l'exception qui a été levée, comme nous allons le voir dans le prochain chapitre (chapitre 4). Naturellement, la même règle existe avec la méthode *main*, permettant ainsi de traiter le cas de la levée d'une exception au cours de l'exécution du thread principal.

Nous avons vu dans la section 3.3 que, même si cette dernière portait sur les éléments liés à l'orientation objets du langage Java, comme la création d'objets ou les appels de méthodes, elle se devait de prendre en compte les contraintes amenées par les aspects liés à la concurrence, notamment au niveau de la granularité fine de la sémantique permettant un entrelacement au niveau de l'exécution des threads et donc une simulation de la concurrence. Cette remarque est encore plus forte en ce qui concerne les exceptions. En effet, si dans un programme avec un unique thread la levée d'une exception entraîne l'arrêt de l'exécution du programme si l'exception n'est pas récupérée, dans le cas d'un programme créant plusieurs threads, cela entraîne non seulement la fin de l'exécution du thread ayant levé l'exception, et mais aussi l'arrêt de l'exécution du programme et donc des autres threads.

```
ThreadCreation:
   makeAttributeList(|- ClName1 -> AttrL)  &
   getFieldDeclaration(objects[], ClName1, vals[], false()
                         |- identifier "run" -> FieldDecl, ClName2, true())  &
   makeClosure(ClName2, vals[], OThId1 |- FieldDecl -> Clr)
   ----------------
   (OThId1, ClName1, OThId2
      ->
      object(
         OThId1, ClName1, AttrL, unlocked(), waits[],
         activity(created(OThId2), Clr))) ;
           provided diff(OThId1, objectId(true(), objectNumber 0));
```

FIG. 3.25 – *Création d'un thread.*

3.5 Multithreading et concurrence

Cette section décrit la spécification sémantique des différentes caractéristiques du langage Java liées à la concurrence. La concurrence, sur une architecture mono-processeur, est simulée grâce à l'introduction d'entrelacement au niveau de l'exécution des différents threads. Nous avons choisi de spécifier une sémantique avec une granularité fine afin d'introduire cet entrelacement de la manière la plus réaliste possible.
Cette section présente tout d'abord la spécification de la création des threads. Une deuxième partie discute les choix effectués au niveau de l'ordonnancement. Une troisième partie permet d'étudier la formalisation du mécanisme de synchronisation, alors qu'une dernière partie décrit une sémantique des méthodes manipulant les locks sur les objets.

3.5.1 Création des threads

Nous avons étudié dans les sections précédentes la structure sémantique d'un thread (Section 3.1.2.1), ainsi que les mécanismes de création d'un objet simple (Section 3.3.2). Comme nous l'avons vu, la principale différence entre un thread et un objet simple est la présence d'une activité au sein du premier.
La première phase de création d'un objet est l'obtention de la liste de ses attributs. Cependant, si l'objet est un thread, il faut de plus construire son activité, en recherchant sa méthode *run* (ou sa méthode *main* dans le cas du thread principal).

Considérons l'exemple d'un thread autre que le thread principal. Sa règle de création est présentée par la Figure 3.25. Cette règle possède comme sujet l'identificateur du thread à créer (`OThId1`), son type (`ClName1`), ainsi que l'identificateur du thread qui le crée (`OThId2`). Comme indiqué précédemment, la première action est de déterminer l'ensemble des attributs du thread à créer. La deuxième prémisse de la règle Typol présentée en Figure 3.25 permet de déterminer le corps de la méthode *run* du thread. La troisième prémisse construit la continuation du thread à partir du source de la

```
GetTheNewExecutingThread_EntryPoint:
   canAThreadBeInTheExecutingStatus(|- ObjL1 -> true(), LastThreadNb)  &
   getExecutingThread(OThId1)  &
   OThId1, LastThreadNb |- ObjL1 -> OThId2, ObjL1_1
   ----------------
   ClVarL |- ObjL1 -> ObjL1_1, OThId2 ;
            do
               ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL)  &
               ENVsendCurrentExecutingThread(ObjL1_1, OThId2)  &
               setNewExecutingThread(OThId2);
```

FIG. 3.26 – *Règle principale de l'ordonnanceur.*

méthode *run*. Ce mécanisme est détaillé dans la Section 3.3.3 qui traite de la spécification sémantique des appels de méthode.
La conclusion de la règle permet de créer l'arbre de syntaxe abstraite représentant le thread à partir des éléments obtenus dans les conclusions des prémisses. L'état du thread ainsi créé est alors *created(O2ThId)*, précisant par la référence *OThId2* quel thread l'a créé. Un tel thread, bien que pourvu d'une activité, n'est pas encore actif. Il ne pourra commencer son exécution propre qu'une fois que sa méthode *start* aura été appelée par un autre thread.

3.5.2 Ordonnancement

La simulation de la concurrence au niveau de l'exécution des threads nous a conduit à introduire de l'entrelacement : après chaque pas atomique d'exécution, le thread courant se stoppe afin de permettre à un autre thread de s'exécuter.
C'est à l'ordonnanceur que revient le choix du thread qui va s'exécuter à un instant donné. Au sein de notre sémantique, nous avons spécifié un ordonnancement déterministe s'assurant que chaque thread va avoir l'occasion de s'exécuter. La Figure 3.26 présente la règle principale du *set* permettant de spécifier l'ordonnanceur.
La sémantique informelle de cet ordonnanceur est la suivante. Tout d'abord, la liste des objets est parcourue à partir du dernier thread exécuté (qui a été dans le statut *executing*) à la recherche du prochain thread pouvant s'exécuter (voir Figure 3.27). Si aucun thread valable n'a été trouvé avant la fin de la liste, la recherche reprend au début de celle-ci (voir Figure 3.28).
Voici les différents cas possibles quand on analyse un objet donné :

- C'est un objet simple, on passe à l'objet suivant.
- C'est un thread, il faut analyser son statut. Différents cas se présentent alors.
  1. Son statut est *runnable*, il devient alors le prochain thread à exécuter, et son statut va devenir *executing*.
  2. Son statut est *blocked* sur un objet qui n'est plus verrouillé. Dans ce cas, il devient le prochain thread à s'exécuter et son statut va devenir *executing*.

```
LookForTheNextExecutingThread_StartAfterTheCurrentThread:
   inf(NbThr, NbLastThr)  &
   ObjL1 |- ObjL1, objectId(true(), objectNumber NbThr) => OThId, ObjL1_1
   ----------------
   objectId(true(), objectNumber NbThr), objectNumber NbLastThr
      |- ObjL1 -> OThId, ObjL1_1 ;
```

FIG. 3.27 – *Recherche à partir du dernier thread executing.*

```
LookForTheNextExecutingThread_StartByTheFirstObject:
   supeq(NbThr, NbLastThr)  &  ObjL1 |- ObjL1 => OThId, ObjL1_1
   ----------------
   objectId(true(), objectNumber NbThr), objectNumber NbLastThr
      |- ObjL1 -> OThId, ObjL1_1 ;
```

FIG. 3.28 – *Recherche à partir du début de la liste.*

3. Son statut ne correspond pas aux deux cas précédents. Dans ce cas, on passe à l'objet suivant au sein de la liste.

La Figure 3.29 présente la règle Typol spécifiant une sémantique du cas considéré en 2. La prémisse (appel du set `isTheObjectLocked`) de cette règle permet de vérifier que le thread dont le statut courant est *blocked* (sinon la règle ne s'applique pas) est en attente d'un lock qui est libre. Le nouveau statut du thread devient alors *executing*.
Il est bon de noter les deux clauses (précédées du mot clé *do*) permettant de mettre à jour l'environnement. La première est utilisée pour effacer le noeud correspondant au lock sur l'objet dans la vue graphique de notre environnement (voir Section 4.2.2) si plus aucun thread n'est dans le statut *blocked* ou *dormant* en attente d'un lock sur l'objet considéré dans cette règle. La seconde met à jour notre liste classant les threads et les objets en fonction de leur statut (voir Section 4.2.3.1).

```
HasFoundTheNextExecutingThread_Blocked:
   isTheObjectLocked(ObjId |- ObjL1 -> false())
   ----------------
   ObjL1 |- objects[object(OThId, Type, AttrL, Lock, WaitSet, activity(blocked(ObjId), Clr)).ObjL2] =>
               OThId,
               objects[object(OThId, Type, AttrL, Lock, WaitSet, activity(executing(), Clr)).ObjL2] ;
            do
               ENVupdateEnvIfNonDormantOrBlockedThreadOnObject(ObjId, ObjL1, 1)  &
               ENVupdateThreadStatusList(OThId, blocked(ObjId), executing());
```

FIG. 3.29 – *Règle permettant à un thread blocked de devenir executing.*

Si aucun thread ne peut devenir *executing*, tous ont fini leur exécution et dans ce cas, l'exécution du programme est terminée; sinon nous détectons et signalons un deadlock.

3.5.3 Formalisation de la synchronisation

La concurrence entre les threads a développé de nouveaux phénomènes comme les problèmes de cohérence au niveau des attributs des objets partagés. En effet, si deux threads exécutent, l'un une action de lecture et l'autre une action d'écriture sur un même attribut d'un objet sur lequel ils possèdent tout deux une référence, le résultat global dépend de l'exécution, et plus précisément de l'entrelacement et l'ordonnancement.
Le mécanisme de synchronisation a été introduit afin de lever cet indéterminisme en permettant de protéger l'accès aux objets partagés. A cette fin, un lock est associé à chaque objet, et à chaque étape de l'exécution, ce lock ne peut être possédé que par un unique thread. Au concept de lock est lié celui de synchronisation qui permet d'affirmer qu'une portion de code (par exemple une méthode manipulant un attribut d'un objet) ne pourra être exécutée que par un thread à la fois. Cela reprend le concept de *section critique* bien connu dans la littérature.

En Java, le seul moyen pour un thread d'acquérir un lock sur un objet, est soit d'entrer dans un bloc déclaré *synchronized*, soit d'appeler une méthode déclarée *synchronized* sur cet objet. Nous avons choisi (comme cela est fait dans la plupart des autres formalisations de Java présentées en Section 3.7.1) de considérer qu'une méthode *synchronized* est équivalente à une méthode "normale" dont le corps est inclus dans un bloc *synchronized*. L'exécution d'un bloc *synchronized* peut se résumer de la façon suivante :

1. Calcul de la référence de l'objet que l'on veut verrouiller (cette référence est donnée sous forme d'une expression en argument de la clause *synchronized*).
2. Tentative de verrouillage de l'objet : le statut du thread courant devient *blocked* si l'objet est déjà verrouillé par un autre thread.
3. Exécution du corps du bloc *synchronized* dès que le lock est acquis.
4. Libération du verrou à la sortie du bloc.

Notre sémantique reflète ce comportement : les Figures 3.30, 3.31, 3.33 et 3.34 montrent les règles Typol correspondant à chaque étape de la spécification informelle ci-dessus.
La Figure 3.30 présente deux règles Typol : la seconde concerne l'évaluation de l'expression donnée en entrée du bloc pour obtenir la référence de l'objet à verrouiller. Elle s'applique tant que l'expression à évaluer n'est pas une valeur référence. Lorsque l'expression à évaluer a été ramenée à une valeur référence de l'objet dont le thread désire acquérir le lock, la première s'applique. Elle permet la transition entre la première et la deuxième étape de l'exécution du bloc *synchronized*.
La Figure 3.31 présente la règle déterminant si le thread courant peut acquérir le lock sur l'objet dont la référence vient d'être obtenue. L'appel du set `getObjectLockInformation` permet d'obtenir les informations relatives au lock de l'objet : quel thread le possède, et combien de fois l'a-t-il acquis (zéro si l'objet n'est pas verrouillé, n locks, avec n

```
SynchronizedStatement_HasTheObjectReference:
    --CS: synchronized(#8) Statement
    ObjL1, ClVarL, OThId
        |- clr(ObjId, Mode, MethName, Env, insts[synchroLock(TVObjId, Statement).InstL]) ->
               ObjL1_1, ClVarL_1, Clr, ThStatus
    ----------------
    ObjL1, ClVarL, OThId
        |- clr(ObjId, Mode, MethName, Env, insts[synchroBegin(TVObjId, Statement).InstL]) ->
               ObjL1_1, ClVarL_1, Clr, ThStatus ;

SynchronizedStatement_EvaluateExpressionToGetTheObjectReference:
    --CS: synchronized(Obj.getObj2()) Statement
    evaluateExpression(ObjL1, ClVarL1, Env1, OThId, ObjId, Mode
                          |- Expr1 -> ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL2, ThStatus)  &
    appendtree(InstL2, insts[synchroBegin(Expr1_1, Statement).InstL1], InstL3)
    ----------------
    ObjL1, ClVarL1, OThId
        |- clr(ObjId, Mode, MethName, Env1, insts[synchroBegin(Expr1, Statement).InstL1]) ->
               ObjL1_1, ClVarL1_1, clr(ObjId, Mode, MethName, Env1_1, InstL3), ThStatus ;
```

FIG. 3.30 – *Détermination de la référence de l'objet.*

```
SynchronizedStatement_GetTheLockOrBeBlocked:
    --CS: synchronized(#8) Statement
    getObjectLockInformation(ObjId2 |- ObjL1 -> OThId2, nbLocks)  &
    getTheLockOrBeBlocked(OThId1, ObjId2, Statement, ObjL1, ClVarL
                              |- OThId2, nbLocks -> ObjL1_1, Inst, ThStatus)
    ----------------
    ObjL1, ClVarL, OThId1
        |- clr(ObjId1, Mode, MethName, Env, insts[synchroLock(ObjId2, Statement).InstL]) ->
               ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, insts[Inst.InstL]), ThStatus ;
```

FIG. 3.31 – *Acquisition d'un verrou sur l'objet ou blocage du thread courant.*

```
set getTheLockOrBeBlocked is
--SC: Locks the given object if it is not already locked.
--SC: If the given object is already locked by
--SC: 1] by the current thread, the number of locks on this object is
--SC:     increased by one.
--SC: 2] by an other thread, the current existing thread is blocked to
--SC:     wait the unlock of the object
--
--JC: Current executing thread reference, Reference of the object to lock,
--JC: Statement, Objects, Class variables (only used for the environment)
--JC: |- Thread that demands the lock on the given object,
--JC: Number of existing locks on the given object
--JC: -> modified object list, Instruction,
--JC: New current executing thread status
judgement
    ObjectId, ObjectId, Statement, Objects, ClassVariablesL
        |- ObjectId, integer -> Objects, Inst, ThreadStatus;

UnlockedObject:
    setLock(ObjId, OThId, integer 1 |- ObjL1 -> ObjL1_1)
    ----------------
    OThId, ObjId, Statement, ObjL1, ClVarL
        |- _, 0 -> ObjL1_1, synchroExecution(ObjId, Statement), runnable() ;
                do ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL);

ObjectLockedByAnotherThread:
    OThId1, ObjId, Statement, ObjL1, _
        |- OThId2, nbLocks -> ObjL1, synchroLock(ObjId, Statement), blocked(ObjId) ;
                provided diff(nbLocks, 0)  &  diff(OThId1, OThId2);

ObjectLockedByTheCurrentThread:
    plus(nbLocks1, 1, nbLocks1_1)  &
    setLock(ObjId, OThId, integer nbLocks1_1 |- ObjL1 -> ObjL1_1)
    ----------------
    OThId, ObjId, Statement, ObjL1, ClVarL
        |- OThId, nbLocks1 -> ObjL1_1, synchroExecution(ObjId, Statement), runnable() ;
                provided diff(nbLocks, 0);
                do ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL);
end getTheLockOrBeBlocked;
```

Fig. 3.32 – *Set permettant au thread d'acquérir le lock ou d'être blocked.*

```
SynchronizedStatement_BeginBlockExecution:
   appendtree(insts[Statement, synchroEnd(ObjId2)], InstL1, InstL2)
   ----------------
   ObjL, ClVarL, _
      |- clr(ObjId1, Mode, MethName, Env, insts[synchroExecution(ObjId2, Statement).InstL1]) ->
            ObjL, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL2), runnable() ;
```

FIG. 3.33 – *Début de l'exécution du bloc synchronized.*

supérieur à zéro, sinon).
Le set `getTheLockOrBeBlocked` représenté en Figure 3.32 analyse ces valeurs. Les trois règles de ce set spécifient les cas suivants :

- L'objet considéré n'est pas verrouillé (valeur 0). Dans ce cas, le thread courant acquiert le lock.
- L'objet considéré est verrouillé par un thread autre que le thread courant. Dans ce cas, le prochain statut du thread courant sera *blocked* en attente d'un lock sur l'objet considéré.
- L'objet considéré est déjà verrouillé par le thread courant. Dans ce cas, le nombre de locks possédés par le thread courant est augmenté de un. Cela peut se produire si un thread appelle successivement plusieurs méthodes *synchronized* sur cet objet, notamment dans le cas de l'appel récursif d'une méthode *synchronized*.

La règle présentée en Figure 3.33 s'applique lorsque le thread courant a pu obtenir un lock sur l'objet considéré. Elle exprime le début de l'exécution du bloc *synchronized*.
Les deux règles Typol de la Figure 3.34 portent sur la fin de l'exécution d'un bloc *synchronized*. La première traite le cas où l'objet est verrouillé une seule fois par le thread courant : dans ce cas, l'objet n'est plus verrouillé. La seconde traite le cas où l'objet est verrouillé plusieurs fois par le thread courant : dans ce cas, on décrémente juste le compteur de verrous, et l'objet reste verrouillé.
Ces règles permettent d'une part de libérer un lock sur l'objet considéré et de déterminer son nouveau statut (verrouillé ou non), et d'autre part de mettre à jour l'environnement en tenant compte des ces modifications (par les clauses *do* suivant la conclusion de ces règles).

3.5.4 Manipulation des verrous

La synchronisation, si elle permet un certain déterminisme lors de l'accès à un attribut d'un même objet par différents threads, introduit de nouveaux problèmes. En effet, un thread peut rester bloqué au sein d'un bloc *synchronized* dans l'attente d'un évènement qui ne pourra se produire que lorsqu'un autre thread aura verrouillé l'objet considéré. Ce phénomène peut conduire à un dead-lock.
C'est pour éviter cela qu'ont été introduites au sein des bibliothèques de "base" de Java certaines méthodes permettant de bloquer un thread au sein d'un bloc *synchronized* tout en relâchant le lock qu'il possédait (méthode *wait*), puis de l'autoriser à acquérir

```
EndSynchronizedStatementWithOneLock_RemoveLock:
   getObjectLockInformation(ObjId2 |- ObjL1 -> OThId, 1)   &
   releaseObjectLock(ObjId2 |- ObjL1 -> ObjL1_1)
   ----------------
   ObjL1, ClVarL, OThId
      |- clr(ObjId1, Mode, MethName, Env, insts[synchroEnd(ObjId2).InstL]) ->
            ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL), runnable() ;
            do
               ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL)   &
               ENVupdateEnvIfNonDormantOrBlockedThreadOnObject(ObjId2, ObjL1_1, 0)   &
               ENVchangeObjectStatusInStatusList(ObjId2);

EndSynchronizedStatementWithSeveralLocks_RemoveOneLock:
   --ESC: End Synchronized
   --
   getObjectLockInformation(ObjId2 |- ObjL1 -> OThId, nbLocks1)   &
   diff(integer nbLocks1, integer 1)   &   sub(nbLocks1, 1, nbLocks1_1)   &
   setLock(ObjId2, OThId, integer nbLocks1_1 |- ObjL1 -> ObjL1_1)
   ----------------
   ObjL1, ClVarL, OThId
      |- clr(ObjId1, Mode, MethName, Env, insts[synchroEnd(ObjId2).InstL]) ->
            ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, InstL), runnable() ;
            do ENVsendObjectAndClassVariableLists(ObjL1_1, ClVarL);
```

FIG. 3.34 – *Suppression d'un verrou sur l'objet.*

```
Wait:
    --RC: Makes the current thread "dormant", adds it to the wait set of the current
    --RC: object and releases the locks on the current object. The current thread is
    --RC: disabled for thread scheduling until another thread call the notifyAll method
    --RC: on the current object (or the notify method).
    --
    --CS: ObjId.wait();
    --PC: Checks that the object 'ObjId' is locked by the current executing
    --PC: thread 'OThId' and determines how many times the object is locked
    --PC: (nb times).
    --PC: Adds the thread identifier 'OThId' to the wait set of the object 'ObjId'.
    --PC: Releases the nb locks on the object 'ObjId'.
    getObjectLockInformation(ObjId |- ObjL1 -> OThId, nb)  &  diff(nb, 0)  &
    addThreadToTheObjectWaitSet(OThId, ObjId |- ObjL1 -> ObjL1_1)  &
    releaseObjectLock(ObjId |- ObjL1_1 -> ObjL1_2)
    ----------------
    ObjL1, ClVarL, OThId, ObjId
        |- identifier "wait", vals[]
            ->
            ObjL1_2, ClVarL, insts[synchroWaitEndRelock(ObjId, integer nb)],
            dormant(ObjId, integer nb) ;
            do ENVchangeObjectStatusInStatusList(ObjId);
```

FIG. 3.35 – *Sémantique de la méthode wait.*

de nouveau ce lock lorsque par exemple un évènement se sera produit (méthode *notify*).

La Figure 3.35 présente une sémantique de la méthode *wait*. Cette méthode est appelée par un thread possédant un lock sur un objet et qui doit attendre qu'un certain évènement se produise avant de continuer l'exécution du bloc *synchronized*. Cela lui permet de passer dans le statut *dormant* tout en relâchant ce lock, et donc d'autoriser un autre thread à l'acquérir.
La première prémisse de la règle Typol appelle le set *getObjectLockInformation* afin de vérifier que le thread courant possède bien un lock sur l'objet considéré (celui sur lequel la méthode *wait* a été appelée), et de connaître le nombre de ces locks. Si cet objet est verrouillé par un autre thread ou s'il n'est pas verrouillé, ce qui démontre une erreur de programmation, une exception est levée ; ce cas fait l'objet d'une seconde règle Typol (voir Figure 3.36).
La deuxième prémisse de la règle de la Figure 3.35 (règle traitant du cas général) ajoute le thread courant à l'ensemble *wait* de l'objet. Cet ensemble réunit les threads ayant appelé la méthode *wait* sur cet objet et qui donc souhaiteront, quand ils quitteront le statut *dormant*, acquérir de nouveau un ou des locks sur cet objet (selon son statut initial déterminé par la première prémisse) comme nous allons le voir par la suite avec la méthode *notify*. La troisième prémisse relâche quant à elle le ou les locks que détenait le thread courant sur l'objet considéré.
Il est important de noter qu'une instruction spéciale est ajoutée en tête de la liste des

```
Wait_Exception:
   --RC: An exception is thrown because the current object is not locked by
   --RC: the current thread.
   --
   --CS : ObjId1.wait();
   getObjectLockInformation(ObjId1 |- ObjL1 -> _, 0)   &
   createExceptionObject(ObjL1, OThId1
                            |- identifier "IllegalMonitorStateException",
                               exceptText
                                  "Method 'wait' called on a non locked object."
                                  ->
                                  _, ObjL1_1, InstL)
   -----------------
   ObjL1, ClVarL, OThId1, ObjId1
      |- identifier "wait", vals[] -> ObjL1_1, ClVarL, InstL, runnable() ;
```

FIG. 3.36 – *Sémantique de la méthode wait lors de la levée d'une exception.*

instructions que le thread aura à exécuter quand il quittera le statut *dormant*. Il s'agit de l'instruction *synchroWaitEndRelock(ObjId, integer nb)*, qui indique que la première action du thread lorsqu'il s'exécutera sera de tenter d'acquérir `nb` locks sur l'objet considéré dont la référence est `ObjId`.

La méthode *notify*, dont la spécification en Typol est décrite par la Figure 3.37, constitue en quelque sorte le pendant de la méthode *wait*. En effet, cette méthode permet à un thread détenant un lock sur un objet de libérer un (méthode *notify*) ou tous les threads (méthode *notifyAll*) appartenant à l'ensemble *wait* de cet objet. Les threads ainsi libérés deviennent alors *blocked* et vont entrer en compétition afin d'acquérir à nouveau le ou les locks qu'ils possédaient sur l'objet considéré avant d'appeler la méthode *wait*.
Nous ne présentons ici que la règle concernant la méthode *notify*, la règle portant sur la méthode *notifyAll* étant similaire à cette dernière si ce n'est l'appel au set *makeBlockedAllDormantThread* à la place du set *makeBlockedTheOldestDormantThread*. De même, il existe aussi des règles traitant, comme dans le cas de la règle présentée en Figure 3.36, des exceptions levées dans le cas où le thread courant ne possède pas de lock sur l'objet considéré.

Il est important de noter que dans le cas de la méthode *notify*, nous avons choisi de sélectionner de façon déterministe le thread à retirer de l'ensemble *wait* de l'objet considéré en choisissant celui qui y avait été ajouté en premier. Ce choix peut varier selon les sémantiques formelles de Java, sa référence informelle ne spécifiant pas la politique à suivre. On pourrait par exemple envisager de retirer de l'ensemble *wait* le dernier thread à y avoir été ajouté, ou un thread déterminé au hasard.

```
Notify:
   --RC: Wakes up a single thread that is waiting on this object monitor.
   --
   --CS: ObjId.notify();
   --PC: A check is done to be sure that the given object is locked by the
   --PC: current executing thread The oldest dormant thread of the wait set
   --PC: is taken, its status becomes blocked on 'ObjId' instead of
   --PC: dormant=> waiting for nb locks on 'ObjId'.
   getObjectLockInformation(ObjId |- ObjL1 -> OThId, nb)  &  diff(nb, 0)  &
   makeBlockedTheOldestDormantThread(ObjId |- ObjL1 -> ObjL1_1)
   ----------------
   ObjL1, ClVarL, OThId, ObjId
      |- identifier "notify", vals[] -> ObjL1_1, ClVarL, insts[], runnable() ;
```

FIG. 3.37 – *Sémantique de la méthode notify.*

Comme nous l'avons vu dans cette section, les aspects liés à la concurrence forment une part très importante de notre spécification en définissant non seulement l'exécution des programmes, mais aussi ses éléments principaux, les threads. Ces aspects touchent aussi la majorité des règles de notre spécification suite à l'introduction de l'entrelacement qui a induit une division de l'exécution en instructions atomiques.

3.6 Typage statique : masquage des attributs, une piste

Comme nous l'avons vu dans la Section 2.1.4, notre spécification formelle de Java n'intègre pas le mécanisme de masquage des attributs. Cependant nous avons étudié les problèmes posés par ce dernier, et ses conséquences au niveau de notre spécification formelle et de notre représentation graphique.

3.6.1 Présentation

Prenons le programme présenté en Figure 3.38. Ce programme comporte trois classes : `SequentialEx`, `Shape` et `Square`, cette dernière héritant de la classe `Shape`.
Dans cet exemple, l'instruction `(3)` est une affectation portant sur des variables (respectivement `shape` et `square`) référençant des objets de types différents (respectivement de type `Shape` et `Square`). Après cette instruction, la variable `shape`, déclarée comme étant de type `Shape`, référence un objet de type `Square`. On dit alors que le *type statique* de la variable `shape` est `Shape`, alors que son *type dynamique* est `Square`. Contrairement au type statique qui est connu à la compilation, le type dynamique d'une variable n'est connu que lors de son exécution.

```java
class SequentialEx {
    public static void main(String[] args) {
        int i;
        boolean b;
        Square square = new Square();
        Shape shape = new Shape();
        /*recuperation de la valeur de nbVisibleSides --> false (1)
        */
        b = square.nbVisibleSides;
        /*recuperation de la valeur de nbVisibleSides --> 0 (2)
        */
        i = shape.nbVisibleSides;
        /*
        affectation polymorphe (3)*/
        shape = square;
        /*recuperation de la valeur de nbVisibleSides --> false (4)
        */
        b = square.nbVisibleSides;
        /*recuperation de la valeur de nbVisibleSides --> 0 (5)
        */
        i = shape.nbVisibleSides;
    }
}

class Shape {
    int nbVisibleSides = 0;
    Shape() {
    }
}

class Square extends Shape {
    boolean nbVisibleSides = false;
    int nbSides = 2;
    Square() {
    }
}
```

FIG. 3.38 – *Programme avec masquage des attributs d'une classe.*

On peut donc se demander quel sera le résultat de l'instruction (5), et plus précisément quelle sera la valeur retournée par l'expression `shape.nbVisibleSides`. Deux possibilités sémantiques seraient envisageables :

1. la valeur de retour est `0` , c'est-à-dire la valeur de l'attribut au sein de la classe `Shape`, si l'on considère une liaison statique
2. la valeur de retour est `false`, c'est-à-dire la valeur de l'attribut au sein de la classe `Square`, si l'on considère une liaison dynamique.

En Java (contrairement à d'autres langages comme Eiffel [1]), la règle est la suivante : *la liaison au niveau des attributs se fait de façon statique* afin d'éviter des problèmes de typage. La valeur retournée sera donc `0`.
Cela illustre le fait que l'instance de la classe `Square` contient en fait l'attribut entier `nbVisibleSides` déclaré dans la classe `Shape` même si celui-ci est masqué par l'attribut booléen de même nom déclaré dans la classe `Square`. La variable `shape` ayant comme type statique `Shape`, elle accède directement à l'attribut déclaré dans la classe `Shape`, alors que toute variable de type statique `Square` accédera à l'attribut booléen déclaré dans la classe `Square`.

3.6.2 Conséquences sur la sémantique formelle

Si notre sémantique veut prendre en compte cet aspect, la structure sémantique d'un objet, et plus précisément de la liste des attributs d'un objet doit être améliorée. Soient `Obj` un objet instance de la classe `C`, et `C'` l'ensemble des classes ancêtres de la classe `C`. La liste des attributs d'un objet serait alors représentée par la structure suivante (visible sous forme graphique dans la Figure 3.39):

- Liste des attributs propres au type `C` sous forme de paires noms-valeurs
- Liste des classes appartenant à l'ensemble `C'` avec pour chacune d'entre elles :
  - la liste des attributs spécifiques à cette classe
  - la liste des attributs communs à cette classe et ses ancêtres.

Il faut de plus modifier la structure sémantique des variables : la nouvelle structure doit permettre de conserver non seulement le nom de la variable, mais aussi son type statique, son type dynamique étant donné par le type de la valeur de cette variable. La Figure 3.40 présente cette nouvelle structure sémantique.

3.7 Travaux connexes et discussion

Cette section permet, en étudiant tout d'abord les principaux travaux portant sur la définition d'une sémantique formelle du langage Java ou plus précisément d'un de ses sous-ensembles, et en les comparant à notre spécification, de dégager des similarités au niveaux des techniques et modèles employés, mais aussi certaines différences qui font l'originalité de notre travail de spécification.

1. En Eiffel, la sémantique est dynamique.

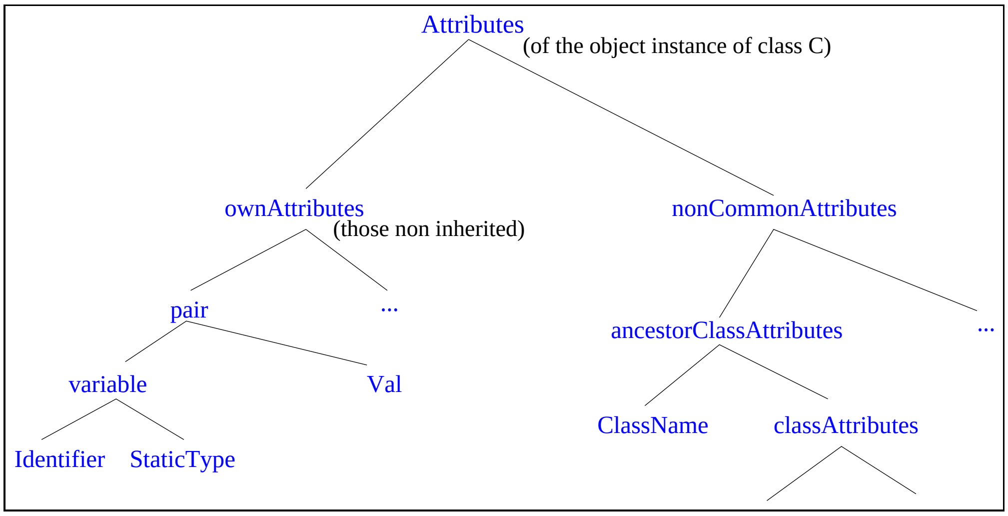

FIG. 3.39 – *Possibilité de structure sémantique pour la liste d'attributs d'un objet instance de la classe* `C`*.*

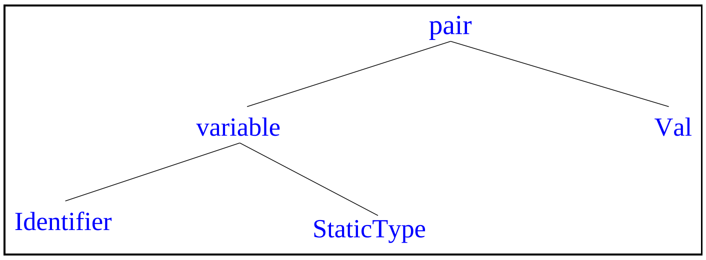

FIG. 3.40 – *Possibilité de structure sémantique pour une paire (variable; valeur).*

3.7.1 Sémantiques formelles de Java

La spécification d'une sémantique formelle de Java est un domaine actif de recherche, sur lequel de nombreuses équipes travaillent. Comme le montre cette section, si les méthodes et les sous-ensembles de Java traités diffèrent, le but principal de ces travaux est d'effectuer des preuves de propriétés.

Sémantiques de la machine virtuelle Java (JVM)

Une première approche dans la spécification d'une sémantique formelle de Java est de s'intéresser au bytecode de la machine virtuelle.

Cette approche a été notamment suivie par Qian [143] qui a spécifié une sémantique d'un sous-ensemble des instructions de la JVM portant sur les objets, les méthodes et les instructions de base du langage. Il décrit les comportements à l'exécution dans l'espace mémoire par des transitions entre états, et les contraintes structurelles sur les instructions à l'aide d'un système de typage statique.
Börger et Schulte [30] définissent de façon modulaire une sémantique du bytecode afin d'effectuer des preuves de correction sur la compilation Java.
Jensen, Le Metayer et Thorn [94] ont formalisé les mécanismes de chargement dynamique des classes dans la JVM afin d'étudier des propriétés de sécurité de Java.
Cette approche a l'avantage d'être au plus proche de l'interpréteur Java, et de porter sur un ensemble d'instructions de base moins large que les instructions du code source, mais si l'on se place au niveau du programmeur, le bytecode est trop bas niveau. C'est pourquoi, comme nous allons le voir dans la section suivante, l'approche la plus suivie est la spécification d'une sémantique du code source Java.

Sémantiques du source Java

La principale approche suivie par les équipes de recherche (et que nous avons adoptée) consiste en la spécification d'une sémantique formelle du source Java, ou plus précisément d'un sous-ensemble de Java.

Drossopoulou et Eisenbach (voir [61, 62, 63, 64]), Syme ([162]), et Nipkow et Oheimb ([127]) spécifient une sémantique de différents sous-ensembles de Java afin de prouver la sûreté des types au sein de ces sous-ensembles. Drossopoulou et Eisenbach travaillent dans les quatre articles cités sur un large sous-ensemble séquentiel de Java (incluant les types primitifs, les classes, l'héritage, les variables et méthodes d'instance, les interfaces, le masquage des variables d'instance, la liaison dynamique, les tableaux et les exceptions) en définissant une sémantique opérationnelle de ce sous-ensemble, et prouvent que l'exécution des programmes préserve le typage au moyen du *Subject Reduction Theorem*.
Nipkow et Oheimb [127] définissent et prouvent des propriétés sur leur sous-ensemble de Java appelé “Java Light” à l'aide du prouveur de théorèmes Isabelle/HOL.
Dans [72], Flatt, Krishnamurthi et Felleisen spécifient un sous-ensemble de Java appelé

"ClassicJava" à travers la définition de sa syntaxe et de règles de typage et d'évaluation. Ils définissent alors une sémantique opérationnelle à travers un système de réécriture contextuelle, et vérifient la sûreté du typage.
Dans [138], Müller et Poetzsch-Heffter présentent une logique proche du style de Hoare ([103]) pour un noyau séquentiel de Java. Ils spécifient alors une sémantique opérationnelle de ce sous-ensemble, et prouvent la sûreté de cette sémantique en la formalisant avec des règles de typage dans une logique d'ordre supérieur.
Börger et al [31, 29] cherchent quant à eux à spécifier une sémantique formelle de l'ensemble du langage Java en incluant les exceptions et surtout la concurrence. Pour cela, ils ont choisi d'utiliser une modélisation dirigée par l'exécution, en utilisant des machines à états abstraits (ou ASM [150]). Les auteurs construisent successivement cinq modèles, chacun étant une extension du précédent : ils distinguent les modèles impératif, procédural, à objets, avec exceptions et enfin concurrent. Un de leurs buts est de formaliser la sémantique de Java afin de pouvoir utiliser cette formalisation pour effectuer des preuves de propriétés du langage et de programmes Java.

3.7.2 Discussion

La plupart des spécifications formelles de Java ne portent que sur un sous-ensemble séquentiel de ce langage, et s'intéressent principalement au typage. Notre travail porte quant à lui sur un sous-ensemble plus large, prenant en compte la concurrence et les aspects liés à la synchronisation qui apportent une complexité supplémentaire à la spécification formelle du langage.

Parmi les travaux portant sur la définition d'une sémantique dynamique du source du langage, les spécifications de Börger et al [29] paraissent les plus proches des notres. Ils souhaitent en effet définir une sémantique formelle de l'ensemble du langage, en essayant de refléter le plus directement possible les spécifications informelles du livre The Java Language Specification ([76]). Ces auteurs poursuivent principalement les mêmes buts que nous, c'est-à-dire définir une base permettant une meilleure compréhension du langage, la standardisation et la clarification des composants critiques de Java, et surtout utiliser cette définition afin de réaliser des preuves de propriétés du langage et des programmes Java (voir chapitre 5).
Contrairement à notre approche qui est dirigée par la syntaxe (la sémantique du langage est définie à partir de la spécification de la syntaxe abstraite des constructeurs de Java), ils ont choisi une approche dirigée par l'exécution utilisant des machines à états abstraits (ASM). Le lecteur intéressé pourra par contre retrouver dans les travaux de Börger et Schulte des simplifications que nous avons aussi introduites dans notre spécification. Par exemple, lors des déclarations des variables locales, une déclaration dans laquelle est incluse une initialisation sera décomposée d'une part en une déclaration simple, et d'autre part en une instruction d'affectation permettant d'affecter à la variable considérée l'expression donnée pour son initialisation.

3.8 Conclusion

Les spécifications sémantiques décrites dans ce chapitre ont été exprimées directement en Typol. L'un des intérêts de ce langage est de permettre d'écrire des spécifications modulaires, rendant ainsi la sémantique plus lisible. De plus, nous avons dû non seulement utiliser Typol de façon classique en écrivant des règles en Sémantique Naturelle, mais aussi écrire des spécifications en Sémantique Opérationnelle Structurelle afin de décrire les différents aspects liés à la concurrence.
Cette utilisation conjointe des deux sémantiques est réalisable simplement au sein du langage Typol, mais est en opposition avec la volonté de fournir une sémantique modulaire. En effet, il n'est pas possible de décrire d'une part une sémantique des instructions "simples" du langage Java, de la notion de création d'objets ou d'appels de méthodes en ignorant la présence de multithreading, puis d'ajouter un module traitant des aspects concurrents sans pour autant modifier les autres modules. Cela est dû à la nécessité de simulation de la concurrence par l'introduction d'entrelacement au niveau de l'exécution des différents threads. Les règles qui pourraient être écrites en Sémantique Naturelle comme celles de l'évaluation des expressions si l'on considérait uniquement un langage séquentiel doivent être transformées afin de suivre la granularité fine imposée par la simulation de l'entrelacement. Cela a entraîné une augmentation assez conséquente du nombre de règles de notre spécification.
L'approche des ASM suivie par Börger et al. possède dans ce cadre-là l'avantage de résoudre ce problème : ils définissent cinq sous-ensembles orthogonaux du langage formalisés par cinq ASM, chacun étant une extension incrémentale du précédent. L'exécution de tout programme Java par une machine ASM est alors modélisée par la composition de l'exécution de cinq sous-machines ASM.

D'un autre côté, Typol et le système Centaur permettent de dériver un interpréteur du langage en compilant directement les fichiers Typol de spécifications formelles en des fichiers Prolog interprétables. Cela nous a conduit à développer Jitan, un environnement d'exécution de Java basé sur notre spécification sémantique.
Cet environnement fait l'objet du prochain chapitre.