Myriam Fourati

VERS UNE SÉMANTIQUE STATIQUE FORMELLE POUR JAVA

Mémoire
présenté
à la Faculté des études supérieures
de l'Université Laval
pour l'obtention
du grade de maître ès sciences (M.Sc.)

Département d'informatique
FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL

JUILLET 1999

À mon mari,
À ma mère et à mon père,
À ma belle-mère et à mon beau-père.

Remerciements

Au terme de ce mémoire, j'aimerais témoigner ma reconnaissance à toutes les personnes qui ont contribué à l'aboutissement de ce travail.

M. Mourad Debbabi qui a dirigé mon travail de maîtrise. Il a développé en moi la passion pour l'informatique théorique en général et les théories sémantiques en particulier. Je tiens à le remercier bien sincèrement pour ses conseils judicieux, ses suggestions intéressantes et ses remarques pertinentes sans lesquelles ce mémoire n'aurait pas pu aboutir.

Je remercie également M. Hakim Lounis mon codirecteur de recherche qui m'a accueilli au Centre de Recherche Informatique de Montréal dans lequel j'ai trouvé de bonnes conditions de travail.

Je désire également remercier M. Jules Desharnais qui a accepté d'évaluer ce mémoire.

Un remerciement particulier est adressé à mon époux M. Imed Sabri Cherif qui a toujours su m'écouter, m'encourager et me soutenir dans les moments les plus difficiles. Sans son amour et ses encouragements, je n'aurais jamais pu accomplir mes rêves.

Je remercie également Mlle Mouetsie Molière et Mme Marta Mitrovic pour leur profonde amitié, leur soutien moral et les discussions techniques fructueuses que nous avons eues.

Mes remerciements vont également à M. Stéphane Doyon et M. Béchir Ktari, mes collègues du groupe LSFM, qui m'ont aidée à démystifier quelques concepts subtils dans le domaine des langages de programmation.

Ma reconnaissance va également à tout le personnel du département d'informatique qui, dans le sourire et la bonne humeur, m'a facilité les tâches administratives et m'a aidée à mener à bien et à terme ce travail. Je tiens à remercier particulièrement Mme Lynda Goulet pour sa grande disponibilité.

Enfin, toute ma gratitude va à toute ma famille et à mes amis. Ils m'ont tous soutenue et encouragée dans les moments difficiles. Mes remerciements sont adressés particulièrement à mes parents qui m'ont donné tout leur amour et leur attention.

Résumé

L'objectif principal de ce mémoire est d'étudier les fondements théoriques du langage Java. Pour ce faire, nous examinons l'état de l'art en matière de sémantique formelle de Java. En conséquence, nous discutons de la complétude et de la correction des propos avancés dans la littérature. En outre, nous rapportons une brève évaluation de la spécification officielle du langage Java au niveau du typage et nous montrons la subtilité de sa sémantique. Dans ce mémoire, nous contribuons en donnant une définition d'une sémantique statique réaliste qui couvre un très grand sous-ensemble de Java jusque là non formalisé. À travers cette définition, nous montrons les difficultés techniques sous-jacentes à la mise au point de cette sémantique tout en discutant des traitements sémantiques adéquats pour répondre à ces difficultés.

Table des matières

i

Remerciements ii

Résumé iii

Table des matières iv

Liste des tableaux vii

1 Introduction 1
- 1.1 Motivations et contexte . . . 1
- 1.2 Objectifs et contributions . . . 3
- 1.3 Structure du mémoire . . . 3

2 Revue de la littérature 4
- 2.1 Introduction . . . 4
- 2.2 État de l'art . . . 5
- 2.3 BALI . . . 6
- 2.4 Notations . . . 7
- 2.5 Notions de base de Isabelle/HOL . . . 7
- 2.6 Syntaxe abstraite . . . 8
  - 2.6.1 Programmes . . . 8
  - 2.6.2 Instructions . . . 10
  - 2.6.3 Expressions . . . 11
  - 2.6.4 Valeurs . . . 11
- 2.7 Tables de recherche . . . 12
- 2.8 Sémantique statique . . . 15
  - 2.8.1 Types . . . 15
  - 2.8.2 Relations de types . . . 16
  - 2.8.3 Règles de typage . . . 18
- 2.9 Bonne formation des déclarations . . . 25
  - 2.9.1 Bonne formation des champs . . . 25
  - 2.9.2 Bonne formation des méthodes . . . 25
  - 2.9.3 Bonne formation des interfaces . . . 26
  - 2.9.4 Bonne formation des classes . . . 26

2.9.5 Bonne formation d'un programme 27
2.10 Sémantique opérationnelle 27
2.10.1 Notion d'état 28
2.10.2 Format des règles d'évaluation 31
2.10.3 Règles d'évaluation opérationnelle 32
2.11 Cohérence de la sémantique statique vis-à-vis de l'opérationnelle 39
2.12 Conclusion 41

3 Évaluation de la spécification et des théories sémantiques 42
3.1 Spécification du langage Java 43
3.1.1 Évolution du langage 43
3.1.2 Quelques erreurs découvertes dans la spécification 45
3.1.3 Ambiguïté de la spécification 47
3.2 Subtilité de la sémantique du langage 48
3.2.1 Héritage de plusieurs méthodes avec la même signature 48
3.2.2 Utilisation d'un champ qui n'est pas encore déclaré 50
3.2.3 Coercition 50
3.2.4 Accessibilité 50
3.3 Fondements théoriques de Java 53
3.3.1 Quelques difficultés dans la formalisation de Java 53
3.3.2 Conformité de l'état de l'art avec la spécification de Java 56
3.3.3 Exceptions 56
3.3.4 Création d'un tableau 57
3.3.5 Instruction d'expression 58
3.4 Conclusion 58

4 Sémantique statique pour Java 59
4.1 Syntaxe abstraite 59
4.1.1 Mots-clés 60
4.1.2 Grammaire du langage Java 61
4.2 Algèbre de Types 62
4.3 Quelques notations 68
4.4 Environnement statique 68
4.5 Objets sémantiques et abréviations 69
4.6 Validité des types 71
4.7 Relations de types 71
4.7.1 Relation de sous-classe 71
4.7.2 Relation de sous-interface 71
4.7.3 Relation d'implantation 73
4.7.4 Relation de conversion implicite entre les types primitifs 74
4.7.5 Relation de conversion implicite entre les types de références . . 74
4.7.6 Relation de coercition 75
4.8 Visibilité 76
4.9 Bonne formation de l'environnement statique 83
4.9.1 Bonne formation des champs des classes 83

4.9.2 Bonne formation des méthodes des classes 84
4.9.3 Bonne formation des constructeurs 84
4.9.4 Bonne formation des classes 85
4.9.5 Bonne formation des champs des interfaces 89
4.9.6 Bonne formation des méthodes des interfaces 89
4.9.7 Bonne formation des interfaces 89
4.9.8 Bonne formation de l'environnement global 90
4.10 Autres objets sémantiques 91
4.11 Contexte 95
4.12 Règles de typage de la sémantique statique 95
4.12.1 Règles de typage des déclarations 96
4.12.2 Règles de typage des blocs 105
4.12.3 Règles de typage des instructions 109
4.12.4 Règles de typage des expressions 116
4.13 Conclusion 122

5 Conclusion **127**

Bibliographie **129**

Liste des tableaux

2.1 Syntaxe d'un programme BALI. 9
2.2 Noms de BALI. 10
2.3 Instructions de BALI. 11
2.4 Expressions de BALI. 11
2.5 Valeurs de BALI. 12
2.6 Fonctions pour la manipulation des tables de recherche (1). 13
2.7 Fonctions pour la manipulation des tables de recherche (2). 13
2.8 Types de BALI. 15
2.9 Fermetures transitives des relations $\Gamma \vdash \_ \prec_i^1 \_$, $\Gamma \vdash \_ \prec_c^1 \_$ et $\Gamma \vdash \_ \rightsquigarrow^1 \_$. 17
2.10 Relation de conversion implicite. 18
2.11 Relation de coercition de types. 19
2.12 Règles de typage des instructions de BALI. 20
2.13 Règles de typage des expressions de BALI. 22
2.14 Fonctions de projection sur les objets 30
2.15 Fonctions de construction et de mise à jour de l'état 30
2.16 Règles d'évaluation des instructions 33
2.17 Règles d'évaluation des expressions : partie 1 36
2.18 Règles d'évaluation des expressions : partie 2 37

4.1 Mots clés 60
4.2 Syntaxe abstraite de Java : partie 1 63
4.3 Syntaxe abstraite de Java : partie 2 64
4.4 Syntaxe abstraite de Java : partie 3 65
4.5 Syntaxe abstraite de Java : partie 4 66
4.6 Syntaxe abstraite de Java : partie 5 67
4.7 Types syntaxiques de Java 67
4.8 Types sémantiques 68
4.9 Environnement statique 70
4.10 Validité des types 72
4.11 Relation de sous-classe 72
4.12 Relation de sous-interface 72
4.13 Relation d'implantation 73
4.14 Relations réflexives de sous-classe et de sous-interface 73
4.15 Relation de conversion implicite entre les types primitifs 74

4.16 Relation de conversion implicite entre les types de références 75
4.17 Relation de conversion implicite 75
4.18 Relation de conversion par coercition 77
4.19 Règles de typage des déclarations : partie 1 97
4.20 Règles de typage des déclarations : partie 2 98
4.21 Règles de typage des déclarations : partie 3 99
4.22 Règles de typage des déclarations : partie 4 100
4.23 Règles de typage des déclarations : partie 5 101
4.24 Règles de typage des déclarations : partie 6 103
4.25 Règles de typage des déclarations : partie 7 104
4.26 Règles de typage des déclarations : partie 8 106
4.27 Règles de typage des déclarations : partie 9 107
4.28 Règles de typage des déclarations de blocs 108
4.29 Règles de typage des déclarations des variables locales 109
4.30 Règles de typage des instructions : partie 1 113
4.31 Règles de typage des instructions : partie 2 114
4.32 Règles de typage des instructions : partie 3 115
4.33 Règles de typage des expressions : partie 1 123
4.34 Règles de typage des expressions : partie 2 124
4.35 Règles de typage des expressions : partie 3 125
4.36 Règles de typage des expressions : partie 4 126

Chapitre 1

Introduction

1.1 Motivations et contexte

L'invention du microprocesseur a déclenché un mouvement technologique révolutionnaire. Cette technologie est si petite qu'elle est difficilement visible et si puissante qu'elle fera naître l'espoir de développer des appareils électroniques intelligents. Afin de maîtriser la complexité de ce genre d'appareils, *Sun Microsystems* a élaboré un projet de recherche nommé «Green». Celui-ci a donné naissance au langage «Oak», créé par James Golsing en 1991 et destiné à écrire des programmes pour contrôler des appareils intelligents. Toutefois, ces appareils sophistiqués n'ont pas apparu aussi rapidement que *Sun Microsystems* l'avait prédit. Par chance, le succès du Web en 1993 a conduit à adapter ce langage à Internet en prenant le nom définitif de Java en mai 1995. Dans cette version, Java a été renforcé et d'autres niveaux de sécurité ont été ajoutés.

Depuis son apparition, le langage Java ne cesse de connaître une ascension fulgurante. Il est considéré comme un nouveau genre de révolution dans la communauté intéressée aux langages de programmation. Il est probablement le langage le plus répandu dans l'histoire de l'informatique moderne. Cette popularité est motivée par les caractéristiques intéressantes que combine le langage Java. En effet, Java est un langage multiparadigmes qui unifie quatre styles de programmation : impératif, orienté objet, distribué et parallèle. Il capitalise les connaissances en langages informatiques, comme Eiffel, SmallTalk, Objective C, ML et Lisp. Les concepteurs du langage Java ont examiné plusieurs aspects des langages usuels C et C++. Le but était de déterminer les caractéristiques qui pourraient être éliminées dans le contexte de la programmation orientée objet moderne et ce pour rendre Java simple et sécurisé. En outre, Java offre la possibilité d'animer des pages Web et peut être utilisé pour créer des applications performantes, notamment dans le domaine du client-serveur. Cependant, la plus importante innovation du langage Java est sa capacité de permettre la transmission d'un code compilé à travers le réseau. Ceci a rendu possible la réalisation des célèbres applications, appelées *applets*, qui peuvent migrer d'une machine à une autre et être exécutées sur n'importe quelle plate-forme qui inclut un interpréteur de code intermédiaire de Java. En effet, afin de concilier la diversité des systèmes d'exploitation, un programme Java est compilé en un format intermédiaire, appelé *bytecode*, indépendant des architectures

matérielles. Ce code, ainsi obtenu, est transmis sur le réseau et est interprété par un moteur appelé la machine virtuelle de Java, encore connu sous l'abréviation de MVJ.

Toutes les particularités du langage sus-dénommées ont contribué à son expansion et à sa popularité. En effet, Java est considéré comme un langage universel qui offre un paradigme révolutionnaire de distribution, de maintenance et d'exécution de programmes. Il est perçu comme la solution pour tous les problèmes opiniâtres dans le domaine du client-serveur et du Word Wide Web. En outre, Java commence à toucher notre vie quotidienne puisque les travaux sur le langage visent à ce qu'il puisse être exécuté sur toute machine ayant un microprocesseur, comme les serveurs, téléphones mobiles, cartes à puces ou récepteurs de poche. Ceci nous conduit à nous préoccuper davantage de la sécurité qu'offre le langage.

La transmission du code sur Internet et son exécution soulèvent de sérieux problèmes en matière de sécurité des informations, notamment la confidentialité et l'intégrité. Java y répond par un renforcement des mécanismes de sécurité à plusieurs niveaux. Entre autres, il offre une plus grande sûreté de programmation, comme la vérification renforcée au niveau du typage, la vérification de code importé par la machine virtuelle et l'élimination des possibilités de réécriture dans la mémoire et donc la corruption des données et ce, par la définition de son modèle de pointeurs. En outre, le langage possède un Gestionnaire de Sécurité qui assure la responsabilité du respect des propriétés de sécurité exigées par l'utilisateur. Le langage est donc supposé être sûr. Cependant, ce propos est sujet à des interrogations et à des débats, d'autant plus que nous ne disposons pas encore de définition formelle de la sémantique du langage Java en entier.

Récemment, plusieurs chercheurs se sont penchés avec beaucoup d'intérêt sur l'étude du langage Java et sur l'élaboration d'un cadre formel pour celui-ci. Cet intérêt n'est pas uniquement motivé par sa popularité. En effet, et comme nous l'illustrerons dans les prochains chapitres, Java est doté d'une sémantique peu habituelle et très subtile. De plus, Java est destiné à être utilisé dans les systèmes qui requièrent un très haut niveau de sécurité. En outre, le fait de permettre la mobilité du code constitue une gageure pour les langages de programmation en matière de sécurité. Nous pensons que ces facteurs justifient le besoin d'avoir des fondements théoriques solides pour Java. D'autant plus que certaines études ont révélé l'existence de failles de sûreté et de sécurité dans le système de type de Java [9, 24].

Le langage Java est, certes, innovateur, mais encore immature et instable. Plusieurs modifications ont été apportées à la description du langage [12, 14] et il existe également des erreurs inédites. En outre, la spécification du langage Java n'est qu'une description textuelle et informelle qui est ambiguë, incomplète et parfois non conforme à l'implantation. Une étude formelle du langage au niveau du typage ne peut qu'être avantageuse pour s'assurer de la sûreté du typage en Java, et par conséquent permettra de consolider ou ébranler la confiance des gens dans le langage.

L'intérêt porté au langage Java a conduit à l'apparition de plusieurs théories sémantiques aussi bien au niveau langage qu'au niveau de la machine virtuelle. Tous ces travaux ont du mérite : ils ont chacun à leur tour aidé à la compréhension du langage Java. Toutefois, une formalisation n'est utile que si elle contribue réellement à raisonner sur la sécurité du langage Java. Une telle sémantique doit traiter des caractéristiques du

langage qui touchent à la sécurité, comme les modificateurs. Nous pensons que toutes les formalisations données dans la littérature ne respectent pas ce critère.

1.2 Objectifs et contributions

Le but principal de ce mémoire est de proposer une définition d'une sémantique statique et formelle réaliste qui couvre presque tout le langage Java. À travers cette définition, nous montrons les difficultés techniques sous-jacentes à la formalisation de Java tout en discutant des traitements sémantiques adéquats pour répondre à ces difficultés. Pour ce faire, nous avons examiné l'état de l'art en matière de fondements théoriques de Java qui ont été proposés dans la littérature. Ceci nous a aidés à développer une sémantique statique qui couvre un grand ensemble de Java jusqu'à maintenant non formalisé. Dans ce mémoire, nous présentons un travail représentatif de la littérature, ensuite nous exhibons une évaluation de la spécification officielle du langage afin de souligner son ambiguïté et nous exposons la subtilité de sa sémantique. Nous discutons également de la complétude et de la cohérence de quelques-unes des propositions avancées dans la littérature par rapport à la spécification et nous montrons les difficultés qu'engendre la mise au point d'une théorie sémantique qui couvre tout le langage Java.

En résumé, les principales contributions de ce mémoire sont :

- Une évaluation d'une partie de la littérature ainsi qu'une exposition des difficultés sous-jacentes à la formalisation de tout le langage Java.
- Une clarification de la description informelle du langage. En effet, nous estimons que notre travail sera d'une grande aide pour la compréhension de la sémantique du langage Java et par la même occasion facilitera son traitement. En outre, le fait d'avoir un système de type formel pour Java lèvera toute ambiguïté pour la compréhension de la description du langage puisque les textes informels sont toujours sujets à différentes interprétations.
- Une théorie sémantique statique réaliste pour presque tout le langage Java.

1.3 Structure du mémoire

Ce document est structuré en cinq chapitres. Outre le présent qui sert d'introduction, le chapitre 2 présente l'état de l'art en matière de sémantique formelle de Java. Dans le chapitre 3, nous évaluons les travaux présentés dans le chapitre précédent et nous procédons à une évaluation de la spécification officielle de Java. Le chapitre 4 est dédié à la présentation d'une sémantique statique qui couvre presque tous les aspects du langage Java. Finalement, une conclusion sur ce travail ainsi que les travaux futurs envisagés sont présentés dans le chapitre 5.

Chapitre 2

Revue de la littérature

2.1 Introduction

Grand nombre de chercheurs se sont penchés avec beaucoup d'intérêt sur l'étude de la formalisation du langage Java, en particulier dans le domaine de la preuve de la sûreté de typage de celui-ci. Ceci a donné naissance à plusieurs théories sémantiques dans la littérature aussi bien au niveau du langage qu'au niveau de la machine virtuelle. Tous ces travaux sont intéressants. Chacun a permis de lever certaines ambiguïtés et subtilités sur le langage. Toutefois, la majorité des travaux est axée sur le même noyau syntaxique de Java modulo quelques différences mineures.

Afin de juger de la pertinence de notre travail et de le situer par rapport à l'état de l'art, nous estimons nécessaire de présenter les différents travaux liés à la sémantique de Java. Le but de ce chapitre est de passer en revue certains de ces travaux. Plusieurs d'entre eux nous ont beaucoup aidé à développer à notre tour une sémantique statique pour Java. Nous nous intéressons en particulier aux contributions de Tobias Nipkow et David von Oheimb [17, 18]. Dans le cadre d'un projet nommé BALI, qui consiste à formaliser plusieurs aspects du langage Java, ils ont donné une syntaxe abstraite, une sémantique statique, une sémantique opérationnelle ainsi qu'une preuve de cohérence de la sémantique statique vis-à-vis de la sémantique opérationnelle d'un sous-ensemble de Java et ce, en utilisant le démonstrateur de théorèmes Isabelle/HOL. Nous avons choisi de présenter ce travail puisque nous pensons qu'il est le plus récent, le plus complet et le plus représentatif de la littérature dans le domaine de la formalisation du langage Java.

Ce chapitre comporte 11 sections. Un survol de l'état de l'art en matière de théories sémantiques de Java est donné dans la section 2.2. La section 2.3 présente le langage BALI d'une façon informelle. Les notations utilisées dans ce chapitre sont décrites dans la section 2.4. Dans la section 2.5, nous donnons les notions de base de Isabelle/HOL. La syntaxe abstraite de BALI est donnée dans la section 2.6. La section 2.7 introduit les tables de recherche qui seront utilisées dans la formalisation. Dans la section 2.8, nous présentons les conditions de la bonne formation des déclarations d'un programme BALI. La section 2.9 est dédiée à la présentation de la sémantique statique de BALI et la section 2.10 à sa sémantique opérationnelle. Dans la section 2.11, nous présentons

le théorème de correction du typage. En conclusion, nous dégageons les avantages de cette étude et nous présentons quelques critiques dans la section 2.12.

2.2 État de l'art

Il existe de nombreuses études liées à la sémantique formelle de Java. Le but de cette section est de passer en revue certains de ces travaux. Nous visons par cela à fournir au lecteur une idée générale sur les formalisations existantes du langage Java.

Au niveau du langage, Sophia Drossopoulou et Susan Eisenbach [6, 7, 8] ont fourni une sémantique statique et une autre opérationnelle ainsi qu'une preuve de correction du typage d'un sous-ensemble de Java qui inclut les types primitifs, les classes, l'héritage, les variables et les méthodes d'instance, les interfaces excluant leurs champs, la création d'objet Java, les tableaux et le traitement de quelques exceptions. Toutefois, plusieurs restrictions et simplifications ont été rapportées à ces constructions. Dans ces travaux, des langages intermédiaires ont été utilisés dans les différentes étapes de la formalisation. Le langage original extrait de Java est nommé Java_s. Ensuite l'évaluation statique de celui-ci y ajoute des annotations de types afin d'y inclure des informations concernant les accès aux champs et les appels aux méthodes. Le langage résultant est alors nommé Java_{se}. Enfin, ce langage est modifié en un autre nommé Java_r et ce, afin de pouvoir décrire l'exécution des programmes. Dans ces travaux, une sémantique à réduction *à petits pas* est adoptée. Cette sémantique décrit toutes les étapes du calcul du résultat.

En rapport avec les travaux sus-dénommés, Don Syme [28, 29] s'est basé sur les résultats fournis par Drossopoulou et Eisenbach afin de corriger quelques failles dans le système formel proposé, de le spécifier et de le vérifier avec le démonstrateur de théorèmes déclaratif DECLARE, basé sur la logique d'ordre supérieur. Toutefois, Syme a apporté moins de simplifications aux constructions du langage Java original. En effet, contrairement à [6, 7, 8], il considère que la classe mère `Object` ainsi que les tableaux possèdent des méthodes. Il a également adopté une approche différente dans la formalisation, comme la représentation de l'environnement statique de typage par des tables (fonctions partielles finies) plutôt que par des listes de déclarations. Nous avons trouvé ce travail très intéressant ; il est complet, clair, concis et assez détaillé. Cependant, une grande partie de la formalisation est écrite dans DECLARE et nécessite donc que le lecteur ait une connaissance de cet outil.

Parallèlement aux travaux de Syme, Tobias Nipkow et David von Oheimb [17, 18] ont développé une théorie sémantique formelle pour un sous-ensemble de Java très similaire à celui de Drossopoulou, Eisenbach et Syme. Cette formalisation sera détaillée dans les prochaines sections.

Isabelle Attali, Denis Caromel et Marjorie Russo [13] ont également proposé une sémantique opérationnelle structurelle d'un autre sous-ensemble de Java incluant le parallélisme en utilisant le système Centaur et en adoptant le formalisme de Typol.

Plus nombreux sont les travaux réalisés sur le vérificateur du code intermédiaire de la machine virtuelle de Java et sur l'élaboration d'un vérificateur fiable et correct. Parmi les théoriciens qui se sont intéressés à ce domaine, nous citons Martin Abadi

et Raymie Stata [26] qui ont développé des règles de typage pour les sous-routines du code intermédiaire. Un autre travail complémentaire au précédent est celui de Stephen N. Freund et John C. Mitchell [9] qui ont formalisé un autre sous-ensemble du byte code et ont prouvé qu'il satisfait certaines contraintes de typage. Leur formalisme a également permis d'identifier une faille inédite dans le vérificateur JDK de Sun. Zhenyu Qian [20] a également développé un système de types d'un large fragment du JVML. Enfin, Stéphane Doyon et Mourad Debbabi [5] se sont intéressés à la spécification du vérificateur et ont trouvé des failles dans celui-ci.

Il existe également d'autres études faites sur le JVML que nous n'avons pas mentionnées puisque nous nous intéressons en particulier au langage Java de haut niveau. Cependant, dans le langage Java, contrairement à d'autres langages de programmation comme C++, les systèmes de types de Java et de la JVML sont très proches.

Le reste de ce chapitre est dédié à la présentation de tous les aspects de la formalisation de BALI, de sa syntaxe abstraite jusqu'à la preuve de correction du typage.

2.3 BALI

Le langage BALI est un sous-ensemble de Java qui a été formalisé par Tobias Nipkow et David von Oheimb dans [18]. Ce travail est une version révisée et étendue de [17] où ce langage s'appelait Java_{light}. Dans la dernière version, plusieurs erreurs ont été corrigées et d'autres caractéristiques du langage Java ont été ajoutées, comme les exceptions.

BALI est un langage impératif et orienté objet qui comprend les classes, les interfaces, les méthodes et les variables d'instance, l'héritage avec la sur-définition des méthodes d'instance et le masquage des variables d'instance, l'appel de méthodes, quelques types primitifs, la création d'objets Java, les tableaux, la levée et le traitement de quelques exceptions, le traitement du pointeur **null** ainsi que quelques conversions de types. Cependant, certaines constructions du langage ont été simplifiées et plusieurs aspects ont été ignorés. Parmi les caractéristiques qui n'ont pas été considérées nous citons le parallélisme, le paquetage, les constructeurs, les variables et les méthodes de classe, les modificateurs d'accès, l'initialisation, le mot clé **super**, les champs des interfaces, le traitement des instructions inaccessibles, quelques types primitifs et les règles de portée des variables. Par ailleurs, le langage BALI comporte quelques généralisations par rapport à la spécification du langage Java telle que décrite dans [11], généralisations que nous présentons ci-dessous :

- Le type du résultat d'une méthode qui sur-définit une autre méthode peut ne pas être identique au type de retour de cette dernière. Il suffit qu'il puisse être converti en celui-ci.
- Les méthodes qui possèdent la même signature et qui sont héritées par la même classe ou interface ne doivent pas obligatoirement avoir le même type de retour.
- Lors de la recherche dynamique des méthodes, il n'y a pas de vérification du type du résultat.
- Le type d'une expression d'affectation est déterminé par le côté droit de l'expression et non par son côté gauche.

Ces généralisations n'ont pas nuit à la preuve de correction du typage, cependant nous pensons que la formalisation du langage BALI augmentée par les caractéristiques omises de Java ne pourrait pas contenir ces généralisations.

2.4 Notations

Les conventions typographiques suivantes sont utilisées dans le reste de ce chapitre : les mots réservés de BALI comme **catch** apparaissent en caractère télétype, les noms des constantes logiques comme cfield sont présentées en police de caractère sans sérif, tandis que les non-terminaux comme *expr* et les méta-variables comme v sont en italique. Tout au long de la formalisation, nous utilisons le terme *ssi* qui est une abréviation de si et seulement si.

Dans ce qui suit, nous présentons la formalisation telle que donnée dans [18] même si nous ne sommes pas d'accord sur certains points, puisque le chapitre suivant sera consacré à l'évaluation et de ce travail et de la spécification du langage Java. Cependant, nous ajoutons plusieurs explications afin de faciliter la compréhension de la formalisation au lecteur. Nous supposons que celui-ci est familiarisé avec la spécification du langage Java. Dans le cas contraire, nous le renvoyons à [11].

2.5 Notions de base de Isabelle/HOL

Avant de présenter la formalisation du langage BALI, nous jugeons nécessaire d'introduire quelques notions de base de l'environnement Isabelle/HOL [16, 25] qui est une instanciation du démonstrateur de théorèmes générique Isabelle [1, 15, 19]. Toutes les définitions que nous donnons ci-dessous ont été prises dans [18].

- Le symbole «$\longrightarrow$» est celui de l'implication.
- Les constantes logiques sont déclarées en séparant leur nom et leur définition par «::».
- Les définitions non-récursives sont écrites en utilisant le symbole «$\stackrel{\text{def}}{=}$».
- Les prédicats sont des fonctions qui retournent une valeur booléenne. Le style curryfié est utilisé dans l'application d'une fonction à un argument.
- L'expression $\varepsilon x.\ P\ x$ dénote une certaine valeur x qui satisfait le prédicat P si elle existe, et une valeur arbitraire sinon.
- La syntaxe des types est celle du langage ML. Cependant, la flèche utilisée comme un constructeur de types de fonctions est notée «$\Rightarrow$». Un type de données est défini en énumérant ses constructeurs, incluant les types de ses paramètres, en les séparant par le symbole «|».
- Les deux types de base sont *bool* et *int*. Les types polymorphes sont $\alpha \times \beta$, $(\alpha)set$ et $(\alpha)list$ pour tout type α.
- L'opérateur " permet d'appliquer une fonction à tous les éléments d'un ensemble :

$$f\text{“}S \stackrel{\text{def}}{=} \{y.\exists x \in S.y = fx\}$$

- Le type polymorphe $\alpha \times \beta$ est fourni avec deux fonctions de projection **fst** et **snd**. Les tuples en HOL sont des paires imbriquées à droite : $(a, b, c) = (a, (b, c))$.
- Étant donné que le type $(\alpha)list$ est défini via une constante générique [], dénotant la liste vide, et un générateur «♯» (le «cons»), il peut être introduit par la déclaration de type suivante :

  $$(\alpha)list \quad = \quad [] \mid \alpha\sharp(\alpha)list$$

  L'opérateur de concaténation de listes est représenté par le symbole «@». Deux fonctions sont fournies avec le type *list*. La première permet d'appliquer une fonction à tous les éléments d'une liste :

  $$\mathsf{map} \quad :: \quad (\alpha \Rightarrow \beta) \Rightarrow (\alpha)list \Rightarrow (\beta)list$$

  et la deuxième permet de convertir une liste d'éléments polymorphes en un ensemble comportant ces mêmes éléments :

  $$\mathsf{set} \quad :: \quad (\alpha)list \Rightarrow (\alpha)set$$

- Le type $(\alpha)option$ est utilisé souvent dans ce chapitre ; il est défini comme suit :

  $$(\alpha)option \quad = \quad \mathsf{None} \mid \mathsf{Some}\ \alpha$$

  Ce type est fourni avec la fonction suivante :

  $$\mathsf{the} \quad :: \quad (\alpha)option \Rightarrow \alpha$$

  telle que $\mathsf{the}\,(\mathsf{Some}\ x) = x$ et $\mathsf{the}\,\mathsf{None} = \mathsf{arbitrary}$, avec **arbitrary** comme une valeur inconnue définie comme suit : $\varepsilon x.\mathsf{False}$.
- Il existe une fonction qui permet de convertir une valeur de type $(\alpha)option$ en un ensemble :

  $$\mathsf{o2s} \quad :: \quad (\alpha)option \Rightarrow (\alpha)set$$

  telle que $\mathsf{o2s}\,(\mathsf{Some}\ x) = \{x\}$ et $\mathsf{o2s}\,\mathsf{None} = \{\}$.

2.6 Syntaxe abstraite

Dans cette section, nous présentons la syntaxe abstraite du langage BALI. Elle inclut la syntaxe des programmes, des instructions, des expressions et des valeurs.

2.6.1 Programmes

Les règles grammaticales d'un programme BALI n'ont pas été fournies dans [18]. Cependant, les auteurs ont fourni une représentation intermédiaire, que nous désignons par syntaxe, qui définit un programme BALI comme une paire de listes qui contiennent les déclarations des classes et des interfaces du programme. Toutefois, la transformation d'un programme BALI original en des listes n'a pas été spécifiée dans ce travail.

prog	=	*(idecl)list* × *(cdecl)list*	déclaration d'un programme BALI
iface	=	*(tname)list* × *(sig* × *mhead)list*	
idecl	=	*tname* × *iface*	déclaration d'une interface BALI
class	=	*(tname)option* × *(tname)list* × *fdecl(list)* × *(mdecl)list*	
cdecl	=	*tname* × *class*	déclaration d'une classe BALI
field	=	*ty*	type du champ
fdecl	=	*ename* × *field*	déclaration d'un champ
sig	=	*mname* × *ty*	nom de la méthode et type du paramètre
mhead	=	*ename* × *ty*	nom du paramètre et type du résultat
lvar	=	*ename* × *ty*	nom et type d'une variable locale
mbody	=	*(lvar)list* × *stmt* × *expr*	variables locales, instructions et expression de retour
methd	=	*mhead* × *mbody*	méthode d'une classe
mdecl	=	*sig* × *methd*	

TAB. 2.1: Syntaxe d'un programme BALI.

La syntaxe d'un programme BALI est donnée par le tableau 2.1. Chaque déclaration est composée du nom de l'entité déclarée et de sa définition. Les noms *tname*, *mname* et *ename* correspondent aux noms de types, de méthodes et des expressions (les identificateurs des champs et des variables) d'un programme BALI respectivement. Le nom *xname* représente un nom d'une classe d'exception système. Les types *tname0* et *ename0* sont des types de HOL et représentent respectivement un nom de type et un nom d'une expression qui est défini par l'utilisateur. La structure des noms est donnée dans le tableau 2.2. La syntaxe d'un programme BALI exige quelques explications que nous donnons ci-dessous :

- La définition d'une interface (*iface*) est composée d'une liste de noms de ses super-interfaces et d'une liste de déclarations de ses méthodes. Si l'interface ne possède pas de super-interfaces, alors la liste est tout simplement vide.
- La définition d'une classe (*class*) comprend le nom de sa super-classe, une liste de déclarations de ses super-interfaces ainsi que deux listes de déclarations de champs et de méthodes. Le type de la super-classe est *(tname)option*. En effet, la classe mère `Object` ne possède pas de super-classe. Dans ce cas, la super-classe sera représentée par le type **None**.
- La déclaration d'un champ d'une classe (*fdecl*) est composée du nom du champ et de son type (*ty*). Les types seront présentés dans la section 2.8.1.
- La déclaration d'une méthode (*sig* × *mhead* pour les méthodes des interfaces et

xname	= **Throwable** | **NullPointer**	les exceptions système
	| **OutOfMemory** | **ClassCast**	
	| **NegArrSize** | **IndOutBound**	
	| **ArrStore**	
tname	= **Object**	sommet de la hiérarchie des classes
	| SXcpt *xname*	nom d'une exception du système
	| TName *tname*0	nom d'une interface ou d'une classe
ename	= **this**	nom spécial attribué au pointeur **this**
	| EName *ename*0	nom d'une expression

TAB. 2.2: Noms de BALI.

mdecl pour les méthodes des classes) est composée d'une signature (nom de la méthode et le type de son paramètre), suivie de (*mhead*) qui est constitué du nom du paramètre de la méthode et du type de sa valeur de retour. S'il s'agit d'une déclaration d'une méthode à l'intérieur d'une classe, alors cette déclaration est suivie du corps de la méthode, d'instructions (*stmt*) et de l'expression de retour de la méthode (*expr*). Pour des raisons de simplicité, les auteurs ont considéré que chaque méthode possède exactement un paramètre. L'existence de plusieurs paramètres peut être simulée par un seul paramètre objet ayant plusieurs champs. En outre, les auteurs exigent qu'il existe une seule expression de retour et qu'elle soit la dernière expression dans le programme.

2.6.2 Instructions

Les instructions de BALI sont présentées dans le tableau 2.3. Seules les instructions élémentaires sont prises en compte. Les différentes variantes des conditionnelles et des boucles ainsi que les instructions de branchement n'ont pas été considérées. En Java, il existe une instruction spéciale appelée *instruction d'expression*. Cette instruction est exécutée en évaluant l'expression correspondante et dans le cas où celle-ci rend une valeur, elle est ignorée. Par exemple, les affectations et les appels de méthodes sont des expressions puisqu'elles retournent des valeurs. Cependant, ils peuvent être considérés comme des instructions en ignorant ces valeurs. Cette instruction spéciale fait partie des instructions de BALI. La conversion d'une expression en une instruction se fait via Expr. Par cette conversion, les auteurs permettent à toutes les expressions d'être des instructions ce qui n'est pas le cas dans le langage Java. ceci permet de simplifier la formalisation. L'instruction SKIP représente l'instruction vide. Elle ne fait rien et son exécution se complète toujours normalement. Pour des raisons de simplicité, l'instruction `try _ catch _ finally _` est divisée en deux instructions élémentaires, soient `try _ catch _` et `_ finally _`. En plus, les auteurs supposent que l'instruction `try _ catch _` possède exactement une clause `catch`. Ils disent que plusieurs clauses `catch` peuvent être représentées par des instructions `if _ else _` en cascade en appliquant l'opérateur `instanceof` qui permet de choisir le bloc `catch` à exécuter.

stmt	= **SKIP**	
	| **Expr** *expr*	
	| *stmt*; *stmt*	
	| **if** (*expr*) *stmt* **else** *stmt*	
	| **while** (*expr*) *stmt*	
	| **throw** (*expr*)	
	| **try**(*stmt*) **catch**(*tname ename*) *stmt*	
	| *stmt* **finally** *stmt*	

TAB. 2.3: Instructions de BALI.

expr	= **new** *tname*	création d'une instance de classe
	| **new** *ty*[*expr*]	création d'un tableau
	| (*ty*)*expr*	coercition de type
	| *expr* **instanceof** *ref_ty*	opérateur de comparaison de types
	| Lit *val*	littéral
	| *ename*	accès à une variable locale/paramètre
	| *ename* := *expr*	affectation à une variable locale/ paramètre
	| {*ref_ty*}*expr*.*ename*	accès à un champ
	| {*ref_ty*}*expr*.*ename* := *expr*	affectation à un champ
	| *expr*[*expr*]	accès à un tableau
	| *expr*[*expr*] := *expr*	affectation à un tableau
	| *expr*.*mname*({*ty*}*expr*)	appel de méthode

TAB. 2.4: Expressions de BALI.

2.6.3 Expressions

En ce qui concerne les expressions BALI, elles sont présentées dans le tableau 2.4. Le formalisme adopté ne traite pas les opérateurs unaires et binaires. L'expression `this` est considérée comme une variable locale à laquelle aucune valeur ne doit être affectée. Les auteurs prétendent que la construction **super** peut être considérée comme une expression `this` qui a été convertie explicitement, par une coercition, en le type de la super-classe de la classe courante. La création de tableaux multidimensionnels peut être simulée par la création de tableaux emboîtés.

Les termes entre accolades {...} sont appelés *annotations de types*. Ils ne font pas partie du langage, mais ils servent d'informations auxiliaires (fournies par le vérificateur de types) qui s'avèrent importantes pour les liaisons statiques des champs et pour résoudre la surcharge des méthodes.

2.6.4 Valeurs

Les valeurs BALI sont présentées dans le tableau 2.5. Le type *loc* représente des adresses abstraites non nulles des objets. Les définitions suivantes introduisent des destructeurs pour *val* :

val	=	Unit	valeur du type void
	|	Bool *bool*	
	|	Intg *int*	
	|	Null	
	|	Addr *loc*	

TAB. 2.5: Valeurs de BALI.

$$\begin{aligned}
&\mathsf{the_Bool} :: \mathit{val} \Rightarrow \mathit{bool}\\
&\mathsf{the_Intg} :: \mathit{val} \Rightarrow \mathit{int}\\
&\mathsf{the_Addr} :: \mathit{val} \Rightarrow \mathit{loc}
\end{aligned}$$

$$\begin{aligned}
\mathsf{the_Bool}\ (\mathsf{Bool}\ b) &= b\\
\mathsf{the_Intg}\ (\mathsf{Intg}\ i) &= i\\
\mathsf{the_Addr}\ (\mathsf{Addr}\ a) &= a
\end{aligned}$$

La représentation d'un programme BALI par des listes joue un rôle important dans la vérification de la bonne formation des programmes. Entre autres, elle rend possible la vérification de l'unicité des déclarations. Ceci est réalisé grâce à la fonction unique qui sera utilsée plus loin dans ce chapitre. Elle est définie comme suit :

$$\mathsf{unique} :: (\alpha \times \beta)\mathit{list} \Rightarrow \mathit{bool}$$

$$\mathsf{unique}\ t \stackrel{\mathsf{def}}{=} \forall (x_1, y_1) \in \mathsf{set}\ t.\ \forall (x_2, y_2) \in \mathsf{set}\ t.\ x_1 = x_2 \rightarrow y_1 = y_2$$

2.7 Tables de recherche

Afin de permettre une recherche efficace des entités déclarées dans un programme BALI, les listes des déclarations sont transformées en des tables abstraites. Ces tables sont réalisées en HOL à l'aide de fonctions partielles qui font correspondre des valeurs à des noms. Ces tables sont définies comme suit :

$$(\alpha, \beta)\mathit{table} = \alpha \Rightarrow (\beta)\mathit{option}$$

Elles sont fournies avec des fonctions qui sont présentées dans le tableau 2.6. Nous donnons ci-dessous quelques explications relatives à ces fonctions ainsi que les types de ces dernières :

- La fonction **empty** retourne une table vide qui ne possède aucune association :

  $$\mathsf{empty} :: (\alpha, \beta)\mathit{table}$$

- La fonction $\_[\_ \mapsto \_]$ permet de mettre à jour une table par l'association $[x \mapsto y]$. Étant donné que le type d'une table est $\alpha \Rightarrow (\beta)\mathit{option}$, la valeur y, associée à x, devrait être de type $(\beta)\mathit{option}$. Ceci est assuré par Some y :

  $$\_[\_ \mapsto \_] :: (\alpha, \beta)\mathit{table} \Rightarrow \alpha \Rightarrow \beta \Rightarrow (\alpha, \beta)\mathit{table}$$

- La fonction $\_ \oplus \_$ permet d'augmenter une table par une autre :

  $$\_ \oplus \_ :: (\alpha, \beta)\mathit{table} \Rightarrow (\alpha, \beta)\mathit{table} \Rightarrow (\alpha, \beta)\mathit{table}$$

empty	$\stackrel{\text{def}}{=}$	$\lambda k.$None
$t[x \mapsto y]$	$\stackrel{\text{def}}{=}$	$\lambda k.$ if $k = x$ then Some y else $t\ k$
$s \oplus t$	$\stackrel{\text{def}}{=}$	$\lambda k.$ case $t\ k$ of None $\longrightarrow$ $s\ k$ | Some $x \longrightarrow$ Some x
table_of []	=	empty
table_of $((k,x)\#t)$	=	(table_of t)$[k \mapsto x]$
t hiding s entails R	$\stackrel{\text{def}}{=}$	$\forall\, k\ x\ y.\ t\ k = \text{Some}\ x \longrightarrow s\ k = \text{Some}\ y \longrightarrow R\ x\ y$

TAB. 2.6: Fonctions pour la manipulation des tables de recherche (1).

Un_tables ts	$\stackrel{\text{def}}{=}$	$\lambda k.\ \bigcup t \in ts.\ t\ k$
$s \oplus\oplus t$	$\stackrel{\text{def}}{=}$	$\lambda k.$ if $t\ k = \{\}$ then $s\ k$ else $t\ k$
t hidings s entails R	$\stackrel{\text{def}}{=}$	$\forall k.\ \forall x \in t\ k.\ \forall y \in s\ k.\ R\ x\ y$

TAB. 2.7: Fonctions pour la manipulation des tables de recherche (2).

- La fonction récursive **table_of** permet de convertir une liste en une table :

  table_of :: $(\alpha \times \beta)list \Rightarrow (\alpha,\beta)table$

- Le prédicat _ **hiding** _ **entails** _ permet de vérifier si les entrées de deux tables sont reliées par un certain prédicat R :

  _ hiding _ entails _ :: $(\alpha,\beta)table \Rightarrow (\alpha,\gamma)table \Rightarrow (\beta \Rightarrow \gamma \Rightarrow bool) \Rightarrow bool$

La définition d'une table, donnée ci-dessus, associe à toute valeur de type α une seule valeur de type $(\beta)option$. Cette table est appelée une table de recherche à entrée simple. Ceci par opposition à une table à entrées multiples dont l'application à une valeur de type α peut rendre plus d'une valeur de type $(\beta)option$. Pour les besoins de la formalisation, les tables à entrées multiples sont requises. Ces dernières sont définies comme suit :

$(\alpha,\beta)tables = \alpha \Rightarrow (\beta)set$

Ces tables sont données avec les fonctions du tableau 2.7. Ci-dessous, nous commentons et nous donnons les types de ces fonctions :

- L'opérateur _ $\oplus\oplus$ _ permet d'augmenter une table à entrées multiples par une autre table à entrées multiples en adoptant la sur-défintion :

  _ $\oplus\oplus$ _ :: $(\alpha,\beta)tables \Rightarrow (\alpha,\beta)tables \Rightarrow (\alpha,\beta)tables$

- La fonction **Un_tables** permet de transformer un ensemble de tables à entrées multiples en une seule table à entrées multiples :

  Un_tables :: $((\alpha,\beta)tables)set \Rightarrow (\alpha,\beta)tables$

- Le prédicat _ **hidings** _ **entails** _ est le même que celui présenté plus haut, sauf que celui-ci s'applique aux tables à entrées multiples :

$$\_ \text{ hidings } \_ \text{ entails } \_ \quad :: \quad (\alpha, \beta)\mathit{tables} \Rightarrow (\alpha, \gamma)\mathit{tables} \Rightarrow (\beta \Rightarrow \gamma \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$$

Dans le reste de ce chapitre, toutes les tables seront appelées tout simplement des tables de recherche.

Les tables de recherche, ainsi définies, permettent de réaliser plusieurs applications sur un programme BALI. À titre d'exemple, les fonctions suivantes permettent de transformer un programme BALI en des tables indexées par les noms des interfaces et des classes :

$$\begin{aligned} &\text{iface} \quad :: \quad \mathit{prog} \Rightarrow (\mathit{tname}, \mathit{iface})\mathit{table} \quad \stackrel{\text{def}}{=} \quad \text{table\_of} \circ \text{fst} \\ &\text{class} \quad :: \quad \mathit{prog} \Rightarrow (\mathit{tname}, \mathit{class})\mathit{table} \quad \stackrel{\text{def}}{=} \quad \text{table\_of} \circ \text{snd} \end{aligned}$$

La projection **fst** sur un programme Γ permet d'extraire une liste de déclarations des interfaces (*idecl list*), tandis que la projection **snd** extrait une liste de déclarations des classes du programme (*cdecl list*). Ensuite, l'application de la combinaison de fonctions table_of ∘ snd permet de transformer ces listes en des tables dont le domaine[1] est l'ensemble des noms des classes et des interfaces.

Il existe également d'autres fonctions intéressantes qui utilisent ces tables et qui permettent de traverser la hiérarchie des types d'un programme pour collecter les méthodes et les champs d'une classe ou d'une interface dans des tables :

$$\begin{aligned} &\text{imethds} \quad :: \quad \mathit{prog} \Rightarrow \mathit{tname} \Rightarrow (\mathit{sig}, \mathit{ref\_ty} \times \mathit{mhead})\mathit{tables} \\ &\text{cmethd} \quad :: \quad \mathit{prog} \Rightarrow \mathit{tname} \Rightarrow (\mathit{sig}, \mathit{ref\_ty} \times \mathit{methd})\mathit{table} \\ &\text{fields} \quad :: \quad \mathit{prog} \Rightarrow \mathit{tname} \Rightarrow ((\mathit{ename} \times \mathit{ref\_ty}) \times \mathit{field})\mathit{list} \end{aligned}$$

Les méthodes des classes et des interfaces déclarées dans un programme Γ sont stockées dans des tables. Le domaine de ces dernières est l'ensemble des signatures de ces méthodes. Leur co-domaine est un ensemble composé d'un type de références *ref_ty* qui correspond au type de la classe ou de l'interface qui définit la méthode et de la définition de celle-ci. La fonction **imethds** retourne une table de déclarations des méthodes à entrées multiples permettant l'héritage, par les interfaces, de plusieurs méthodes ayant la même signature. En ce qui concerne les champs des classes, ils sont stockés dans une liste. Chaque champ est représenté dans la liste par son nom *ename*, la classe qui le définit *ref_ty* et par son type *field*.

La hiérarchie des classes et des interfaces peut être cyclique. Dans ce cas, une définition récursive des fonctions ci-dessus ne serait pas possible. Cependant, un programme bien formé est sûrement acyclique. Par conséquent, ces fonctions seront présentées sous forme d'équations récursives qui s'assurent de la bonne formation du programme et ce, par le prédicat **wf_prog** qui sera introduit dans la section 2.9 :

$$\begin{aligned} &\text{wf\_prog } \Gamma \wedge \text{iface } \Gamma\ I = \text{Some}(is, ms) \longrightarrow \\ &\text{imethds } \Gamma\ I = \text{Un\_tables}((\lambda J.\ \text{imethds } \Gamma\ J)\text{``set } is) \oplus\oplus \\ &\qquad\qquad (\text{o2s} \circ \text{table\_of}(\text{map } (\lambda(s, mh).\ (s, \text{IfaceT } I, mh))\ ms)) \end{aligned}$$

[1] Ici nous pouvons utiliser les termes domaine et co-domaine puisque les tables sont des fonctions.

prim_ty	=	**void**
	|	**boolean**
	|	**int**
ref_ty	=	NullT
	|	IfaceT *tname*
	|	ClassT *tname*
	|	ArrayT *ty*
ty	=	PrimT *prim_ty*
	|	RefT *ref_ty*

TAB. 2.8: Types de BALI.

$$\text{wf\_prog}\ \Gamma \wedge \text{class}\ \Gamma\ C = \text{Some}(sc, si, fs, ms) \longrightarrow$$
$$\text{cmethd}\ \Gamma\ C = (\text{case}\ sc\ \text{of None} \Rightarrow \text{empty} \mid \text{Some}\ D \Rightarrow \text{cmethd}\ \Gamma\ D) \oplus \text{table\_of}(\text{map}\ (\lambda(s, m).\ (s, (\text{ClassT}\ C, m)))\ ms))$$

$$\text{wf\_prog}\ \Gamma \wedge \text{class}\ \Gamma\ C = \text{Some}(sc, si, fs, ms) \longrightarrow$$
$$\text{fields}\ \Gamma\ C = \text{map}\ (\lambda(fn, ft).\ ((fn, \text{ClassT}\ C), ft))\ fs\ @\ (\text{case}\ sc\ \text{of None} \Rightarrow [] \mid \text{Some}\ D \Rightarrow \text{fields}\ \Gamma\ D)$$

Ces trois équations possèdent la même structure. La table contenant les membres d'une classe ou d'une interface est construite récursivement à partir des tables des super-interfaces et/ou de la super-classe, si elles existent, afin de modéliser l'héritage. Ensuite, les membres déclarés dans la classe ou l'interface courante sont ajoutés à cette table en adoptant la sur-définition. Tous les membres ainsi collectés dans la table reçoivent un nouveau libellé, soit la classe ou l'interface qui les définit.

2.8 Sémantique statique

Cette section est consacrée à la présentation de la sémantique statique du langage BALI. Nous commençons par introduire les types de BALI. Ensuite, nous présentons les relations de types. Enfin, nous décrivons les règles de typage des instructions et des expressions.

2.8.1 Types

Les types de BALI sont des valeurs du type de données *ty*. Ils sont ou bien des types de références ou bien des types primitifs. Les types de BALI sont présentés dans le tableau 2.8 et ils sont commentés ci-dessous :

- Un type de données *ty* représente tous les types de BALI, il est constitué de types primitifs *prim_ty* et de types de références *ref_ty*.
- Un type primitif est ou bien un type de base *int* ou *bool* ou bien le type **void**. Celui-ci représente le type de résultat des méthodes n'ayant pas de valeur de retour.

- Un type de références *ref_ty* est ou bien la référence nulle, le type d'une interface, le type d'une classe ou encore le type d'un tableau. Le type NullT est le type de l'expression `null`.

Afin d'augmenter la lisibilité de la formalisation, les auteurs ont défini des abréviations pour les types : NT est l'abréviation de RefT NullT, Iface *I* représente RefT(Iface *I*), Class *C* correspond à RefT(Class *C*) et *T*[] est l'abréviation de RefT(ArrayT *T*).

Pour s'assurer que les types utilisés dans un programme BALI sont valides, reconnus par le programme, les auteurs ont défini un prédicat récursif nommé is_type défini formellement comme suit :

$$\begin{aligned}
&\mathsf{is_type} :: \mathit{prog} \Rightarrow \mathit{ty} \Rightarrow \mathit{bool} \\
&\mathsf{is_type}\ \Gamma\ (\mathsf{PrimT}\ _) = \mathsf{True} \\
&\mathsf{is_type}\ \Gamma\ \mathsf{NT} = \mathsf{True} \\
&\mathsf{is_type}\ \Gamma\ (\mathsf{Iface}\ I) = \mathsf{is_iface}\ \Gamma\ I \\
&\mathsf{is_type}\ \Gamma\ (\mathsf{Class}\ C) = \mathsf{is_class}\ \Gamma\ C \\
&\mathsf{is_type}\ \Gamma\ (T[]) = \mathsf{is_type}\ \Gamma\ T
\end{aligned}$$

Ce prédicat stipule que tous les types primitifs ainsi que la référence nulle sont considérés comme des types valides dans un programme. En plus, un type d'interface ou d'une classe n'est considéré comme un type valide du programme que s'il existe une déclaration de ce type dans celui-ci. Ceci peut être vérifié grâce aux deux prédicats suivants :

$$\begin{aligned}
&\mathsf{is_iface} :: \mathit{prog} \Rightarrow \mathit{tname} \Rightarrow \mathit{bool} \\
&\mathsf{is_class} :: \mathit{prog} \Rightarrow \mathit{tname} \Rightarrow \mathit{bool} \\
&\mathsf{is_iface}\ \Gamma\ \mathit{tn} \stackrel{\mathsf{def}}{=} \mathsf{iface}\ \Gamma\ \mathit{tn} \neq \mathsf{None} \\
&\mathsf{is_class}\ \Gamma\ \mathit{tn} \stackrel{\mathsf{def}}{=} \mathsf{class}\ \Gamma\ \mathit{tn} \neq \mathsf{None}
\end{aligned}$$

Enfin, un tableau *T*[] est un type valide dans un programme Γ ssi *T* est un type valide dans ce même programme.

Nous tenons à préciser que *T*[] est le type d'un tableau dont les éléments sont de type *T* et que les noms des interfaces et des classes sont des types.

Toute variable de type *ty* possède une valeur par défaut obtenue grâce à la fonction définie formellement comme suit :

$$\begin{aligned}
&\mathsf{default_val} :: \mathit{ty} \Rightarrow \mathit{val} \\
&\mathsf{default_val}\ (\mathsf{PrimT}\ \texttt{boolean}) = \mathsf{Bool\ False} \\
&\mathsf{default_val}\ (\mathsf{PrimT}\ \texttt{void}) = \mathsf{Unit} \\
&\mathsf{default_val}\ (\mathsf{PrimT}\ \texttt{int}) = \mathsf{Intg\ 0} \\
&\mathsf{default_val}\ (\mathsf{RefT}\ r) = \mathsf{Null}
\end{aligned}$$

2.8.2 Relations de types

Les relations entre les types sont déduites à partir de la hiérarchie des classes et des interfaces d'un programme Γ donné. Ces relations sont détaillées ci-dessous.

$$\frac{\Gamma \vdash I \prec_i^1 K}{\Gamma \vdash I \prec_i K} \qquad \frac{\Gamma \vdash I \prec_i J;\ \Gamma \vdash J \prec_i K}{\Gamma \vdash I \prec_i K}$$

$$\frac{\Gamma \vdash C \prec_c^1 E}{\Gamma \vdash C \prec_c E} \qquad \frac{\Gamma \vdash C \prec_c D;\ \Gamma \vdash D \prec_c E}{\Gamma \vdash C \prec_c E}$$

$$\frac{\Gamma \vdash C \rightsquigarrow^1 J}{\Gamma \vdash C \rightsquigarrow J} \qquad \frac{\Gamma \vdash C \rightsquigarrow^1 I;\ \Gamma \vdash I \prec_i J}{\Gamma \vdash C \rightsquigarrow J} \qquad \frac{\Gamma \vdash C \prec_c^1 D;\ \Gamma \vdash D \rightsquigarrow J}{\Gamma \vdash C \rightsquigarrow J}$$

TAB. 2.9: Fermetures transitives des relations $\Gamma \vdash \_ \prec_i^1 \_$, $\Gamma \vdash \_ \prec_c^1 \_$ et $\Gamma \vdash \_ \rightsquigarrow^1 \_$.

Relations de sous-classe, de sous-interface et d'implantation

Les relations de sous-interface directe ($\_ \vdash \_ \prec_i^1 \_$), de sous-classe directe ($\_ \vdash \_ \prec_c^1 \_$) ainsi que d'implantation directe ($\_ \vdash \_ \rightsquigarrow^1 \_$) sont de type $prog \times tname \times tname \Rightarrow bool$ et elles sont définies comme suit :

$$\Gamma \vdash I \prec_i^1 J \stackrel{\text{def}}{=} \mathsf{is\_iface}\ \Gamma\ I \wedge \mathsf{is\_iface}\ \Gamma\ J \wedge J \in \mathsf{set}(\mathsf{fst}(\mathsf{the}(\mathsf{iface}\ \Gamma\ I)))$$
$$\Gamma \vdash C \prec_c^1 D \stackrel{\text{def}}{=} \mathsf{is\_class}\ \Gamma\ C \wedge \mathsf{is\_class}\ \Gamma\ D \wedge \mathsf{Some}\ D = \mathsf{fst}(\mathsf{the}(\mathsf{class}\ \Gamma\ C))$$
$$\Gamma \vdash C \rightsquigarrow^1 I \stackrel{\text{def}}{=} \mathsf{is\_class}\ \Gamma\ C \wedge \mathsf{is\_iface}\ \Gamma\ I \wedge I \in \mathsf{set}(\mathsf{fst}(\mathsf{snd}(\mathsf{the}(\mathsf{class}\ \Gamma\ C))))$$

La relation $\Gamma \vdash I \prec_i^1 J$ indique que l'interface I est une sous-interface directe de l'interface J. C'est-à-dire que J figure parmi l'ensemble des super-interfaces dans la déclaration de l'interface I. De même, la relation $\Gamma \vdash C \prec_c^1 D$ indique que la classe C est une sous-classe directe de la classe D. Ceci veut dire que cette dernière existe en tant que super-classe dans la déclaration de C. Enfin, la relation $\Gamma \vdash C \rightsquigarrow^1 I$ stipule que la déclaration de la classe C fournit une implantation à l'interface I. Pour un Γ donné. on définit, dans le tableau 2.9, la fermeture transitive (et non réflexives) de $\Gamma \vdash \_ \prec_i^1 \_$, $\Gamma \vdash \_ \prec_c^1 \_$ et $\Gamma \vdash \_ \rightsquigarrow^1 \_$.

Relation de conversion implicite

La relation de conversion implicite : $\Gamma \vdash S \preceq T$ signifie que S est un sous-type syntaxique de T. C'est-à-dire qu'en présence de n'importe quel contexte d'expression, le type S peut remplacer le type T sans aucune conversion explicite. La relation de conversion implicite est définie dans le tableau 2.10. Dans [18], uniquement les conversions impliquant les types de références sont considérées.

Relation de coercition

Pour permettre la coercition de types, les auteurs ont introduit une relation de coercition : $\Gamma \vdash S \preceq_? T$, qui signifie qu'une valeur de type S peut être convertie explicitement en une valeur de type T. Cette relation est présentée dans le tableau 2.11. Nous donnons ci-dessous quelques explications relatives à ces règles.

- Afin de comprendre la règle (CL-CL$_C$), nous rappelons la spécification de Java : pour qu'un objet instance d'une classe de type D puisse être converti par une coercition en une classe de type C, il suffit que ces deux classes soient reliées.

(IDENT)	$\dfrac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T}$	(NL-RF$_I$)	$\dfrac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$
(IF-IF$_I$)	$\dfrac{\Gamma \vdash I \prec_i J}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Iface}\ J}$	(IF-OB$_I$)	$\dfrac{\mathsf{is\_iface}\ \Gamma\ I;\ \mathsf{is\_class}\ \Gamma\ \mathsf{Object}}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Class\ Object}}$
(CL-CL$_I$)	$\dfrac{\Gamma \vdash C \prec_c D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D}$	(CL-IF$_I$)	$\dfrac{\Gamma \vdash C \rightsquigarrow I}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ I}$
(AR-AR$_I$)	$\dfrac{\Gamma \vdash \mathsf{RefT}\ S \preceq \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT}\ S)[] \preceq (\mathsf{RefT}\ T)[]}$	(AR-OB$_I$)	$\dfrac{\mathsf{is\_type}\ \Gamma\ T;\ \mathsf{is\_class}\ \Gamma\ \mathsf{Object}}{\Gamma \vdash T[] \preceq \mathsf{Class\ Object}}$

TAB. 2.10: Relation de conversion implicite.

C'est-à-dire, D et C représentent la même classe, ou bien D est une sous-classe de C ou bien C est une sous-classe de D. Les deux premières conditions sont remplies par la règle (IMPL) qui fait appel aux règles de conversion implicite (dans ce cas (IDENT) et (CL-CL$_I$)). La dernière condition est spécifiée explicitement dans la règle.

- Les règles (CL-IF$_C$) et (IF-CL$_C$) stipulent qu'il est possible statiquement de convertir une classe en une interface. En effet, même si la classe n'implante pas l'interface, il est possible qu'une de ses sous-classes le fasse.
- Une coercition d'une interface I en une interface J, telle que I et J contiennent des méthodes ayant la même signature, est permise uniquement si ces dernières possèdent des types de retour compatibles. C'est-à-dire que le type du résultat de la méthode de I peut être converti implicitement en le type de retour de la méthode de J. En effet, à l'exécution, la coercition entre deux types de références n'est permise que dans le cas où ils contiennent la même référence vers un objet. Ainsi, il devient possible à une méthode de les implanter toutes les deux. Cette condition est vérifiée dans la règle (IF-IF$_C$) grâce au prédicat _ **hiding** _ **entails** _.

2.8.3 Règles de typage

Cette section présente l'environnement de types, la forme des règles de typage et les règles de typage des expressions et des instructions de BALI.

Environnement de types

L'environnement statique *env* est un couple. La première composante est le programme Γ à typer. La deuxième composante Λ est une table de variables locales dont le domaine est un ensemble de noms de variables et le co-domaine est un ensemble de types de ces variables. La table Λ inclut également la classe courante qui représente le type de `this`. L'environnement est défini comme suit :

$$\begin{aligned} lenv &= (ename, ty)table \\ cnv &= prog \times lenv \end{aligned}$$

$$\text{(IMPL)}\quad \frac{\Gamma \vdash S \preceq T}{\Gamma \vdash S \preceq_? T}$$

$$\text{(CL-CLC)}\quad \frac{\Gamma \vdash C \prec_c D}{\Gamma \vdash \mathsf{Class}\ D \preceq_? \mathsf{Class}\ C}$$

$$\text{(CL-IFC)}\quad \frac{\mathsf{is\_class}\ \Gamma\ C;\ \mathsf{is\_iface}\ \Gamma\ I}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Iface}\ I}$$

$$\text{(AR-ARC)}\quad \frac{\Gamma \vdash \mathsf{RefT}\ S \preceq_? \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT}\ S)[] \preceq_? (\mathsf{RefT}\ T)[]}$$

$$\text{(OB-ARC)}\quad \frac{\mathsf{is\_class}\ \Gamma\ \mathtt{Object};\ \mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash \mathsf{Class}\ \mathtt{Object} \preceq_? T[]}$$

$$\text{(IF-CLC)}\quad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \mathsf{is\_class}\ \Gamma\ C}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Class}\ C}$$

$$\text{(IF-IFC)}\quad \frac{\begin{array}{c}\mathsf{is\_iface}\ \Gamma\ J;\ \neg\Gamma \vdash I \prec_i J;\\ \mathsf{imethds}\ \Gamma\ I\ \mathsf{hidings}\ \mathsf{imethds}\ \Gamma\ J\ \mathsf{entails}\\ (\lambda(m_1,(pn_1,rT_1))\ (m_2,(pn_2,rT_2)).\ \Gamma \vdash rT_1 \preceq rT_2)\end{array}}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Iface}\ J}$$

TAB. 2.11: Relation de coercition de types.

Les règles de typage font appel à des opérations de projection sur l'environnement de types qui consistent à extraire l'une des composantes de celui-ci. Ces opérations sont définies comme suit :

prg :: $env \Rightarrow prog \stackrel{\text{def}}{=} \lambda(\Gamma, \Lambda).\ \Gamma$ *projection sur le programme*

lcl :: $env \Rightarrow lenv \stackrel{\text{def}}{=} \lambda(\Gamma, \Lambda).\ \Lambda$ *projection sur la table des variables locales*

Dans le reste de ce chapitre, l'environnement de types sera représenté par E.

Forme des règles de typage

La sémantique statique exprime le bon typage des instructions et des expressions inductivement par rapport à un environnement. Elle se sert de deux relations. La première est utilisée pour typer les instructions :

$$\_ \vdash \_ :: \diamond \quad :: \quad env \Rightarrow stmt \Rightarrow bool$$

signifie que sous l'environnement de types *env*, l'instruction *stmt* est bien typée. La deuxième est utilisée pour typer les expressions :

$$\_ \vdash \_ :: \_ \quad :: \quad env \Rightarrow expr \Rightarrow ty \Rightarrow bool$$

signifie que sous l'environnement de types *env*, l'expression *expr* est bien typée et que son type est *ty*.

$$(\text{SKIP}_T) \quad \frac{}{E \vdash \textsf{SKIP} :: \Diamond}$$

$$(\text{EXPR}_T) \quad \frac{E \vdash e :: T}{E \vdash \textsf{Expr}\ e :: \Diamond}$$

$$(\text{SEQN}_T) \quad \frac{E \vdash c_1 :: \Diamond;\ E \vdash c_2 :: \Diamond}{E \vdash c_1;\ c_2 :: \Diamond}$$

$$(\text{IF-ELSE}_T) \quad \frac{E \vdash e :: \textsf{PrimT boolean};\quad E \vdash c_1 :: \Diamond;\quad E \vdash c_2 :: \Diamond}{E \vdash \textbf{if}(e)\ c_1\ \textbf{else}\ c_2 :: \Diamond}$$

$$(\text{WHILE}_T) \quad \frac{E \vdash e :: \textsf{PrimT boolean};\quad E \vdash c :: \Diamond}{E \vdash \textbf{while}(e)\ c :: \Diamond}$$

$$(\text{FINALLY}_T) \quad \frac{E \vdash c_1 :: \Diamond;\ E \vdash c_2 :: \Diamond}{E \vdash c_1\ \textbf{finally}\ c_2 :: \Diamond}$$

$$(\text{THROW}_T) \quad \frac{E \vdash e :: \textsf{Class}\ tn;\quad \textsf{prg}\ E \vdash \textsf{Class}\ tn \preceq \textsf{Class (SXcpt Throwable)}}{E \vdash \textbf{throw}\ e :: \Diamond}$$

$$(\text{TRY-CATCH}_T) \quad \frac{\begin{array}{c}(\Gamma, \Lambda) \vdash c_1 :: \Diamond;\quad \Gamma \vdash \textsf{Class}\ tn \preceq \textsf{Class (Sxcpt Throwable)};\\ \Lambda\ vn = \textsf{None};\ (\Gamma, \Lambda[vn \mapsto \textsf{Class}\ tn]) \vdash c_2 :: \Diamond\end{array}}{(\Gamma, \Lambda) \vdash \textbf{try}\ c_1\ \textbf{catch}(tn\ vn)\ c_2 :: \Diamond}$$

TAB. 2.12: Règles de typage des instructions de BALI.

Règles de typage des instructions

Les règles de typage des instructions sont triviales et elles sont présentés dans le tableau 2.12. Cependant, nous fournissons ci-dessous quelques éclaircissements.

- La relation $E \vdash \textsf{Class}\ tn \preceq \textsf{Class (SXcpt Throwable)}$, utilisée dans les règles (THROW_T) et (TRY-CATCH_T), permet de s'assurer qu'une expression levée ou capturée en tant qu'exception est un objet d'exception qui est une instance de la classe prédéfinie **Throwable**.
- Dans la règle (TRY-CATCH_T), l'instruction c_2 relative au bloc **catch** doit être évaluée en présence de l'environnement augmenté par les informations de types relatives à l'objet d'exception vn. Ce dernier ne doit pas avoir le même nom qu'une variable locale. Ceci est assuré par la condition : $\Lambda\ vn = \textsf{None}$.

Règles de typage des expressions

Les règles de typage concernant les expressions sont moins triviales et sont présentées dans le tableau 2.13. Avant de donner les explications relatives à ces règles, nous tenons à rappeler la généralisation présentée au début de la section 2.3 : le type d'une expression d'affectation est donné par le type de l'expression du côté droit de cette expression. Cette généralisation s'applique aux règles (LOCAL-VAR2$_T$), (FIELD2$_T$) et (ARRAY2$_T$).

- La règle (NEW-CLASS$_T$) permet de typer une expression de création d'un objet. Le type de la création d'un nouvel objet instance d'une classe C est le type de celle-ci.
- La règle (NEW-ARRAY$_T$) permet de typer l'expression de création d'un nouveau tableau. Le type de cette expression est le type du tableau.
- La règle (CAST$_T$) permet de typer une expression de coercition de types. Une coercition d'une expression e de type T vers un type T' est permise ssi T peut être converti par une coercition en T'. Le type de l'expression qui a subi la coercition a T' comme type.
- La fonction **typeof**, utilisée dans la règle (LITERAL$_T$), permet de fournir le type d'un littéral. Elle est définie formellement comme suit :

```
typeof :: (loc ⇒ ty option) ⇒ val ⇒ ty option

typeof dt Unit     = Some (PrimT void)
typeof dt (Bool b) = Some (PrimT boolean)
typeof dt (Intg i) = Some (PrimT int)
typeof dt Null     = Some (RefT NullT)
typeof dt (Addr a) = dt a
```

  La fonction *dt* donnée en paramètre à **typeof** n'est pas utile lors de la recherche du type d'un littéral. C'est pour cette raison que la fonction **typeof** est appelée, dans la règle (LITERAL$_T$), avec l'argument (λa. **None**) qui est une fonction qui retourne une valeur arbitraire. La fonction **typeof** sera appelée plus loin dans ce chapitre en l'appliquant à un argument *dt* plus intéressant, soit une fonction qui calcule le type dynamique d'une référence.
- La règle (LOCAL-VAR1$_T$) permet de typer l'accès à une variable locale. Elle s'assure que le type de la variable locale, obtenu à partir de Λ, est bel est bien un type valide du programme.
- La règle (LOCAL-VAR2$_T$) illustre le typage de l'affectation d'une variable locale. Lorsque la valeur d'une expression v est affectée à une variable vn, il faut s'assurer que le type de l'expression peut être converti en le type de la variable. Cette règle vérifie également que la variable en question n'est pas l'expression `this`, puisque l'affectation à cette variable n'est pas permise.
- La règle (FIELD1$_T$) permet de typer une expression d'accès à un champ nommé *fn* via une expression e. Normalement, l'expression e devrait être un type de références. Cependant, les champs des interfaces et des tableaux ne font pas partie

(NEW-CLASS$_T$) $$\frac{\mathsf{is\_class}\ (\mathsf{prg}\ E)\ C}{E \vdash \mathbf{new}\ C :: \mathsf{Class}\ C}$$

(NEW-ARRAY$_T$) $$\frac{\mathsf{is\_type}\ (\mathsf{prg}\ E)\ T;\ \ E \vdash i :: \mathsf{PrimT}\ \mathbf{int}}{E \vdash \mathbf{new}\ T[i] :: T[]}$$

(CAST$_T$) $$\frac{E \vdash e :: T;\ \ \mathsf{prg}\ E \vdash T \preceq_? T'}{E \vdash (T')e :: T'}$$

(LITERAL$_T$) $$\frac{\mathsf{typeof}\ (\lambda a.\ \mathsf{None})\ x = \mathsf{Some}\ T}{E \vdash \mathsf{Lit}\ x :: T}$$

(INSTANCEOF$_T$) $$\frac{E \vdash e :: \mathsf{RefT}\ T;\ \ \mathsf{prg}\ E \vdash \mathsf{RefT}\ T \preceq_? \mathsf{RefT}\ T'}{E \vdash e\ \mathbf{instanceof}\ T' :: \mathsf{PrimT}\ \mathbf{boolean}}$$

(LOCAL-VAR1$_T$) $$\frac{\mathsf{lcl}\ E\ vn = \mathsf{Some}\ T;\ \ \mathsf{is\_type}\ (\mathsf{prg}\ E)\ T}{E \vdash vn :: T}$$

(LOCAL-VAR2$_T$) $$\frac{E \vdash vn :: T;\ \ E \vdash v :: T';\ \ \mathsf{prg}\ E \vdash T' \preceq T;\ \ vn \neq \mathbf{this}}{E \vdash vn := v :: T'}$$

(FIELD1$_T$) $$\frac{E \vdash e :: \mathsf{Class}\ C;\ \ \mathsf{cfield}\ (\mathsf{prg}\ E)\ C\ fn = \mathsf{Some}\ (fd, fT)}{E \vdash \{fd\}e.fn :: fT}$$

(FIELD2$_T$) $$\frac{E \vdash \{fd\}e.fn :: T;\ E \vdash v :: T';\ \mathsf{prg}\ E \vdash T' \preceq T}{E \vdash \{fd\}e.fn := v :: T'}$$

(ARRAY1$_T$) $$\frac{E \vdash a :: T[];\ E \vdash i :: \mathsf{PrimT}\ \mathbf{int}}{E \vdash a[i] :: T}$$

(ARRAY2$_T$) $$\frac{E \vdash a[i] :: T;\ E \vdash v :: T';\ \mathsf{prg}\ E \vdash T' \preceq T}{E \vdash a[i] := v :: T'}$$

(METHOD$_T$) $$\frac{\begin{array}{c} E \vdash e :: \mathsf{RefT}\ T;\quad E \vdash p :: pT \\ \mathsf{max\_spec}\ (\mathsf{prg}\ E)\ T\ (mn, pT) = \{((md, (pn, rT), pT')\} \end{array}}{E \vdash e.mn(\{pT'\}p) :: rT}$$

TAB. 2.13: Règles de typage des expressions de BALI.

de la syntaxe de BALI. Ceci limite le type de cette expression à un type de classe : Class C. La fonction cfield permet de chercher la définition d'un champ et elle doit retourner une paire composée du type de la classe *fd* qui contient la définition du champ et du type fT de celui-ci. Cette fonction est définie formellement comme suit :

$$\mathsf{cfield} \quad :: \quad \mathit{prog} \Rightarrow \mathit{tname} \Rightarrow (\mathit{ename}, \mathit{ref\_ty} \times \mathit{field})\mathit{table}$$

$$\mathsf{cfield}\ \Gamma\ C \quad \stackrel{\mathsf{def}}{=} \quad \mathsf{table\_of}\ ((\mathsf{map}\ (\lambda((n.d),t).\ (n,(d,t))))\ (\mathsf{fields}\ \Gamma\ C))$$

La fonction cfield ressemble à *fields* (définie dans la section 2.7), sauf qu'au lieu de chercher le champ en se basant sur son nom et la classe qui contient sa définition, la recherche se fait en considérant uniquement le nom du champ. Ainsi, uniquement les champs non cachés par d'autres définitions de champs sont visibles. La recherche se fait en traversant la hiérarchie des classes pour le nom fn en commençant par la classe C de e. Les annotations de types qui apparaissant entre accolades :$\{\ldots\}$, sont utilisées pour implanter les liaisons statiques des champs et pour résoudre la surcharge des noms des méthodes statiquement. Chaque annotation est déterminée par la valeur d'une fonction dans la prémisse de la règle. Ceci est aussi valable pour la règle (METHOD_T). Par exemple, l'accès au champ $\{fd\}e.fn$ est annoté par la classe fd qui contient sa définition. Celle-ci est retournée par la fonction cfield. L'annotation $\{fd\}$ ainsi obtenue est utilisée au moment de l'exécution pour accéder au champ via la paire (fn, fd).

- La règle (FIELD2_T) représente le typage de l'affectation d'un champ. Elle doit s'assurer que le type de l'expression v peut être converti en le type du champ.
- La règle (METHOD_T) permet de typer une expression d'invocation d'une méthode via l'expression e. Cette dernière doit être un type de références et plus précisément le type d'une classe ou d'une interface. En effet, la syntaxe de BALI ne permet pas aux tableaux d'avoir des méthodes. La fonction max_spec est utilisée dans la règle afin de trouver la déclaration de la méthode qui soit *spécifique au maximum* à la méthode invoquée. D'après la spécification de Java [11], une déclaration d'une méthode est dite *spécifique au maximum* à une invocation d'une méthode si elle est applicable et accessible à cette dernière et qu'il n'existe aucune autre méthode applicable et accessible à cette invocation qui soit plus spécifique qu'elle. Cependant, l'accessibilité n'est pas prise en compte dans la formalisation, puisque les modificateurs ne font pas partie de la syntaxe de BALI. La définition de cette fonction est donnée ci-dessous :

$$\mathsf{max\_spec} \quad :: \quad \mathit{prog} \Rightarrow \mathit{ref\_ty} \Rightarrow \mathit{sig} \Rightarrow ((\mathit{ref\_ty} \times \mathit{mhead}) \times \mathit{ty})\mathit{set}$$

$$\begin{aligned}\mathsf{max\_spec}\ \Gamma\ T\ \mathit{sig} \quad \stackrel{\mathsf{def}}{=} \quad & \{m \mid m \in \mathsf{appl\_methds}\ \Gamma\ T\ \mathit{sig} \wedge \\ & \quad (\forall\, m' \in \mathsf{appl\_methds}\ \Gamma\ T\ \mathit{sig}. \\ & \qquad \mathsf{more\_spec}\ \Gamma\ m'\ m \rightarrow m' = m)\}\end{aligned}$$

Cette fonction effectue un certain nombre d'opérations que nous détaillons dans les paragraphes qui suivent :

- La fonction trouve premièrement l'ensemble des méthodes applicables à l'invocation de la méthode, c'est-à-dire l'ensemble des méthodes qui possèdent un type de paramètre qui peut être converti en le type du paramètre correspondant dans l'invocation de la méthode. La fonction qui permet de trouver l'ensemble des méthodes applicables est définie comme suit

$$\mathsf{appl\_methds} \;::\; prog \Rightarrow ref\_ty \Rightarrow sig \Rightarrow ((ref\_ty \times mhead) \times ty)set$$

$$\mathsf{appl\_methds}\ \Gamma\ T\ (mn, pT) \stackrel{\mathsf{def}}{=} \{(m, pT') \mid m \in \mathsf{mheads}\ \Gamma\ T\ (mn, pT') \wedge \Gamma \vdash pT \preceq pT'\}$$

La fonction **mheads** permet de chercher la table des méthodes qui sont membres du type T qui correspond au type de l'expression e. Lorsque ce type est une classe, la fonction **mhead** fait appel à la fonction **option_map** afin d'ignorer la définition des corps des méthodes et ce, pour permettre à la méthode **mheads** de retourner des tables ayant le même type. Les définitions de ces deux fonctions sont données dans ce qui suit :

$$\mathsf{mheads} \;::\; prog \Rightarrow ref\_ty \Rightarrow sig \Rightarrow (ref\_ty \times mhead)set$$

$$\begin{aligned}
&\mathsf{mheads}\ \Gamma\ \mathsf{NullT} &&\stackrel{\mathsf{def}}{=} \lambda sig.\ \{\}\\
&\mathsf{mheads}\ \Gamma\ (\mathsf{IfaceT}\ I) &&\stackrel{\mathsf{def}}{=} \mathsf{imethds}\ \Gamma\ I\\
&\mathsf{mheads}\ \Gamma\ (\mathsf{ClassT}\ C) &&\stackrel{\mathsf{def}}{=} \mathsf{o2s} \circ \mathsf{option_map}\ (\lambda(d,(h,b)).(d,h)) \circ \mathsf{cmethd}\ \Gamma\ C\\
&\mathsf{mheads}\ \Gamma\ (\mathsf{ArrayT}\ T) &&\stackrel{\mathsf{def}}{=} \lambda sig.\ \{\}
\end{aligned}$$

$$\mathsf{option\_map} \;::\; (\alpha \Rightarrow \beta) \Rightarrow (\alpha\ option \Rightarrow \beta\ option)$$

$$\mathsf{option\_map}\ f \stackrel{\mathsf{def}}{=} \lambda y.\ \mathsf{case}\ y\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{None} \mid \mathsf{Some}\ x \Rightarrow \mathsf{Some}\ (f\ x)$$

- Lorsqu'il existe plusieurs méthodes applicables à l'invocation de la méthode, il faut choisir la méthode la *plus spécifique*. Plus précisément, si la méthode m ayant un paramètre de type p est déclarée dans une classe ou une interface de type T et que la méthode m' ayant un paramètre de type p' est déclarée dans la classe ou l'interface de type T', alors m est dite plus spécifique que m' ssi T peut être converti implicitement en T' et p peut être converti implicitement en p'. La définition de cette fonction est donnée comme suit :

$$\begin{aligned}
&\mathsf{more_spec} \;::\; prog \Rightarrow (ref_ty \times mhead) \times ty \Rightarrow\\
&\qquad\qquad (ref_ty \times mhead) \times ty \Rightarrow bool
\end{aligned}$$

$$\begin{aligned}
&\mathsf{more_spec}\ \Gamma\ ((md, mh), pT)\ ((md', mh'), pT') \stackrel{\mathsf{def}}{=}\\
&\qquad \Gamma \vdash \mathsf{RefT}\ md \preceq \mathsf{RefT}\ md' \wedge \Gamma \vdash pT \preceq pT'
\end{aligned}$$

La fonction **max_spec** doit retourner un ensemble qui contient exactement une seule paire, sinon l'accès à la méthode est considéré comme ambigu. Ainsi, l'appel de la méthode est annoté par pT' qui est le type du paramètre de la méthode la plus spécifique mn. Le type rT de l'expression d'invocation de la méthode est celui du résultat de la méthode mn.

2.9 Bonne formation des déclarations

Tout programme doit être bien formé, c'est-à-dire qu'il doit satisfaire certaines conditions de bonne formation. Ces dernières sont exprimées sous la forme de prédicats portant sur les champs, les méthodes, les interfaces, les classes et toutes les déclarations du programme. Dans ce qui suit nous présentons la bonne formation de chaque déclaration à part, ensuite nous introduisons le prédicat global qui exprime la bonne formation d'un programme BALI.

2.9.1 Bonne formation des champs

Une déclaration de champ est dite bien formée ssi son type existe. Le prédicat permettant de s'assurer de la bonne formation d'un champ est défini comme suit :

$$\mathsf{wf\_fdecl} \;::\; \mathit{prog} \Rightarrow \mathit{fdecl} \Rightarrow \mathit{bool}$$

$$\mathsf{wf\_fdecl}\ \Gamma\ (\mathit{fn}, \mathit{ft}) \stackrel{\mathsf{def}}{=} \mathsf{is\_type}\ \Gamma\ \mathit{ft}$$

2.9.2 Bonne formation des méthodes

Une déclaration de méthode est dite bien formée, entre autres lorsque ses arguments ainsi que le type de son expression de retour sont bien définis et que le nom du paramètre est différent de `this`. Ceci est vérifié grâce au prédicat **wf_mhead** qui sera appelé plus loin par le prédicat **wf_mdecl**. Il est défini de la façon suivante :

$$\mathsf{wf\_mhead} \;::\; \mathit{prog} \Rightarrow \mathit{sig} \times \mathit{mhead} \Rightarrow \mathit{bool}$$

$$\mathsf{wf\_mhead}\ \Gamma\ ((\mathit{mn}, \mathit{pT}), (\mathit{pn}, \mathit{rT})) \stackrel{\mathsf{def}}{=} \mathsf{is\_type}\ \Gamma\ \mathit{pT} \wedge \mathsf{is\_type}\ \Gamma\ \mathit{rT} \wedge \mathit{pn} \neq \mathtt{this}$$

En plus, si la déclaration apparaît dans une classe, alors les conditions suivantes doivent être remplies :

- Les noms des variables locales, modélisés dans la table *ltab*, doivent être uniques (voir la fonction **unique** dans la section 2.6).
- Aucune variable locale ne doit avoir le nom spécial `this` : *ltab* `this` = **None**.
- Aucun nom de variable locale ne doit cacher celui du paramètre : *ltab* pn = **None**.
- Tous les types des variables locales doivent exister.
- Le corps de la méthode *blk* doit être bien typé. Il doit être évalué en présence de l'environnement statique E qui contient toutes les variables locales *lvars* de la méthode, le type de la classe courante `this` ainsi que le paramètre de la méthode pn et son type pt. L'environnement est mis à jour grâce à l'expression : $E = (\Gamma.\mathit{ltab}[\mathtt{this} \mapsto \mathsf{Class}\ C][\mathit{pn} \mapsto \mathit{pT}])$.
- Le type T de l'expression de retour *res* de la méthode peut être converti implicitement (voir le tableau 2.10) en le type du résultat rT spécifié dans la déclaration de la méthode.

Les conditions susmentionnées sont vérifiées par le biais du prédicat **wf_mdecl** défini comme suit :

$$\mathsf{wf\_mdecl} :: \mathit{prog} \Rightarrow \mathit{tname} \Rightarrow \mathit{mdecl} \Rightarrow \mathit{bool}$$

$$\begin{aligned}
&\mathsf{wf_mdecl}\ \Gamma\ C\ ((mn, pT), (pn, rT), \mathit{lvars}, \mathit{blk}, \mathit{res}) \stackrel{\mathsf{def}}{=} \\
&\quad \mathsf{let}\ \mathit{ltab} = \mathsf{table_of}\ \mathit{lvars};\ E = (\Gamma, \mathit{ltab}[\mathtt{this} \mapsto \mathsf{Class}\ C][pn \mapsto pT]) \\
&\quad \mathsf{in}\ \mathsf{wf_mhead}\ \Gamma\ ((mn, pT), (pn, rT)) \wedge \\
&\qquad \mathsf{unique}\ \mathit{lvars} \wedge \mathit{ltab}\ \mathtt{this} = \mathsf{None} \wedge \mathit{ltab}\ pn = \mathsf{None} \wedge \\
&\qquad (\forall (vn, T) \in \mathsf{set}\ \mathit{lvars}.\ \mathsf{is_type}\ \Gamma\ T) \wedge \\
&\qquad E \vdash \mathit{blk} :: \Diamond \wedge \exists T.\ E \vdash \mathit{res} :: T \wedge \Gamma \vdash T \preceq rT
\end{aligned}$$

2.9.3 Bonne formation des interfaces

Une déclaration d'une interface est dite bien formée lorsque les conditions suivantes sont satisfaites :

- Aucune classe du programme ne possède le même nom que cette interface.
- Toutes ses super-interfaces existent, sont définies dans le programme, et aucune d'elles n'est en même temps sa sous-interface. Ceci empêche la circularité dans le programme.
- Toutes les méthodes déclarées dans l'interface possèdent des noms différents et elles sont bien formées.
- Toute méthode qui sur-définit un ensemble de méthodes des super-interfaces (en ayant le même nom et le même type de paramètre) doit avoir un type de retour qui soit convertible implicitement en le type de retour des méthodes sur-définies. Cette condition est vérifiée par le prédicat _ hidings _ entails _.

Le prédicat qui permet de vérifier toutes ces conditions est défini comme suit :

$$\mathsf{wf\_idecl} :: \mathit{prog} \Rightarrow \mathit{idecl} \Rightarrow \mathit{bool}$$

$$\begin{aligned}
&\mathsf{wf_idecl}\ \Gamma\ (I, (is, ms)) \stackrel{\mathsf{def}}{=} \neg\mathsf{is_class}\ \Gamma\ I \wedge \\
&\quad (\forall J \in \mathsf{set}\ is.\ \mathsf{is_iface}\ \Gamma\ J \wedge \neg\Gamma \vdash J \prec_i I) \wedge \\
&\quad \mathsf{unique}\ ms \wedge (\forall\, m \in \mathsf{set}\ ms.\ \mathsf{wf_mhead}\ \Gamma\ m \wedge \\
&\quad \mathsf{let}\ \mathit{mtab} = \mathsf{Un_tables}\ ((\lambda J.\ \mathsf{imethds}\ \Gamma\ J)\text{`}\mathsf{set}\ is) \\
&\quad \mathsf{in}\ (\mathsf{o2s} \circ \mathsf{table_of}\ ms)\ \mathsf{hidings}\ \mathit{mtab}\ \mathsf{entails} \\
&\qquad (\lambda (pn, rT)\ (m, (pn', rT')).\ \Gamma \vdash rT \preceq rT'))
\end{aligned}$$

2.9.4 Bonne formation des classes

De la même façon, une déclaration d'une classe est dite bien formée lorsque toutes les conditions suivantes sont vérifiées :

- Aucune interface du programme ne possède le même nom que cette classe.
- Toutes les interfaces implantées existent et la classe fournit une définition pour chaque méthode qui est membre de ces interfaces. En plus, le type de retour de la méthode de la classe doit être convertible implicitement en le type des méthodes implantées.
- Tous les champs et toutes les méthodes déclarés dans la classe sont nommés différemment et ils sont bien formés.

- Si la classe n'est pas la classe mère **Object**, alors elle doit avoir une super-classe qui ne soit pas en même temps sa sous-classe. Ceci permet d'empêcher d'avoir un programme circulaire.
- Toute méthode qui sur-définit une méthode de la super-classe doit avoir un type de retour qui soit convertible implicitement en le type de retour de la méthode sur-définie. Cette condition est vérifiée par le prédicat _ **hidings** _ **entails** _.

Ci-dessous est présenté le prédicat qui vérifie les conditions susnommées :

$$\begin{aligned}
&\mathsf{wf_cdecl} :: \mathit{prog} \Rightarrow \mathit{cdecl} \Rightarrow \mathit{bool}\\
&\mathsf{wf_cdecl}\ \Gamma\ (C,(sc,si,fs,ms)) \stackrel{\mathsf{def}}{=} \neg\mathsf{is_iface}\ \Gamma\ C \wedge\\
&\quad(\ \forall I \in \mathsf{set}\ si.\ \mathsf{is_iface}\ \Gamma\ I \wedge\\
&\qquad \forall s.\forall(m_1,(pn_1,rT_1)) \in \mathsf{imethds}\ \Gamma\ I\ s.\\
&\qquad\quad \exists m_2\ pn_2\ rT_2\ b.\ \mathsf{cmethd}\ \Gamma\ C\ s = \mathsf{Some}\ (m_2,(pn_2,rT_2),b) \wedge\\
&\qquad\qquad\qquad \Gamma \vdash rT_2 \preceq rT_1) \wedge\\
&\quad \mathsf{unique}\ fs \wedge (\forall f \in \mathsf{set}\ fs.\ \mathsf{wf_fdecl}\ \Gamma\ f) \wedge\\
&\quad \mathsf{unique}\ ms \wedge (\forall m \in \mathsf{set}\ ms.\ \mathsf{wf_mdecl}\ \Gamma\ m) \wedge\\
&\quad (\mathsf{case}\ sc\ \mathsf{of}\ \mathsf{None} \Rightarrow C = \mathbf{Object}\\
&\qquad\quad |\ \mathsf{Some}\ D \Rightarrow \mathsf{is_class}\ \Gamma\ D \wedge \neg\Gamma \vdash D \prec_c C \wedge\\
&\qquad\qquad\qquad \mathsf{table_of}\ ms\ \mathsf{hiding}\ \mathsf{cmethd}\ \Gamma\ D\ \mathsf{entails}\\
&\qquad\qquad\qquad (\lambda((pn_1,rT_1),b)\ (m,(pn_2,rT_2),b')).\ \Gamma \vdash rT_1 \preceq rT_2)
\end{aligned}$$

2.9.5 Bonne formation d'un programme

Un programme BALI est considéré bien formé lorsque toutes les classes et les interfaces sont nommées différemment et sont à leur tour bien formées. Afin de conserver l'uniformité dans la formalisation, l'ensemble des classes du programme doit inclure les déclarations des classes prédéfinies de BALI qui sont **Object** et toutes les classes d'exceptions système. Avant d'introduire le prédicat qui exprime la bonne formation d'un programme, voici les déclarations des classes prédéfinies :

$$\begin{aligned}
&\mathsf{ObjectsC} \stackrel{\mathsf{def}}{=} (\mathbf{Object},(\mathsf{None},[],[],[]))\\
&\mathsf{SXcptC}\ xn \stackrel{\mathsf{def}}{=} \mathsf{let}\ sc = \mathsf{if}\ xn = \mathbf{Throwable}\ \mathsf{then}\ \mathbf{Object}\ \mathsf{else}\ \mathsf{SXcpt}\ \mathbf{Throwable}\ \mathsf{in}\\
&\qquad\qquad (\mathsf{SXcpt}\ xn,(\mathsf{Some}\ sc,[],[],[]))
\end{aligned}$$

Enfin, le prédicat qui suit est utilisé pour vérifier qu'un programme BALI est bien formé :

$$\begin{aligned}
&\mathsf{wf_prog} :: \mathit{prog} \Rightarrow \mathit{bool}\\
&\mathsf{wf_prog}\ \Gamma \stackrel{\mathsf{def}}{=} \mathsf{let}\ is = \mathsf{set}\ (\mathsf{fst}\ \Gamma);\ cs = \mathsf{set}\ (\mathsf{snd}\ \Gamma)\\
&\qquad\qquad \mathsf{in}\ \mathsf{ObjectC} \in cs \wedge \forall xn.\ \mathsf{SXcptC}\ xn \in cs \wedge\\
&\qquad\qquad\quad \mathsf{unique}\ (\mathsf{fst}\ \Gamma) \wedge \forall i \in is.\ \mathsf{wf_idecl}\ \Gamma\ i \wedge\\
&\qquad\qquad\quad \mathsf{unique}\ (\mathsf{snd}\ \Gamma) \wedge \forall c \in cs.\ \mathsf{wf_cdecl}\ \Gamma\ c
\end{aligned}$$

2.10 Sémantique opérationnelle

Nous avons, jusque là, présenté la syntaxe abstraite, le système de types ainsi que les conditions de bonne formation des déclarations de BALI. Cette section introduit

sa sémantique opérationnelle qui décrit le comportement opérationnel, l'interprétation dynamique des instructions et des expressions. L'approche adoptée pour représenter cette sémantique est l'utilisation d'une sémantique qui s'intéresse seulement aux résultats de l'évaluation. Les auteurs ont utilisé cette approche puisqu'elle est plus simple, plus précise et plus modulaire que la sémantique à réduction. Cependant, nous pensons qu'une extension de BALI par l'ajout des processus légers *threads* nécessite l'utilisation d'une sémantique à réduction *à petits pas*.

L'exécution de certaines exceptions n'a pas été décrite dans [11]; les auteurs ont introduit quelques définitions afin de spécifier ces exceptions :

- l'instruction **throw** lève une exception **NullPointer** lorsque son argument s'évalue en la référence **Null**;
- chaque exception système levée produit un nouvel objet d'exception;
- s'il n'existe pas assez de mémoire pour allouer une exception **OutOfMemory**, l'exécution du programme est interrompue.

Avant d'introduire les règles d'évaluation opérationnelles pour les instructions et les expressions, nous introduisons la notion d'état ainsi que le format de ces règles.

2.10.1 Notion d'état

Un *état* est constitué d'une exception de type $(xcpt)option$, d'un tas *heap* et d'un cadre courant d'appels qui contient les valeurs des variables locales (incluant les paramètres des méthodes, des exceptions ainsi que le pointeur **this**). Un état est défini formellement comme suit :

$$\begin{aligned} state &= (xcpt)option \times st \\ st &= heap \times locals \end{aligned}$$

Afin d'extraire l'une des composantes de l'état, les auteurs font appel à des opérateurs de projection qu'ils définissent comme suit :

$$\begin{aligned} \mathsf{heap} &:: st \Rightarrow heap \stackrel{\mathsf{def}}{=} \lambda(h,l).\, h \\ \mathsf{locals} &:: st \Rightarrow locals \stackrel{\mathsf{def}}{=} \lambda(h,l).\, l \end{aligned}$$

Nous rappelons que les tuples sont associatifs à droite. Ce qui veut dire que si, par exemple, l'état σ est égal à (x,σ'), alors x représente l'exception, tandis que la deuxième composante de l'état est un tuple de type *st* qui est un état n'incluant pas l'exception. Celui-ci sera appelé *petit état*.

Une exception est une référence vers une instance d'une classe d'exception sous-classe de la classe **Throwable**. Normalement, lorsqu'une exception est levée, un nouvel objet d'exception est alloué et son adresse est retournée afin de représenter cette exception. Dans [18], l'allocation est différée, et donc juste le nom de l'exception est rapporté, jusqu'à ce qu'une clause **catch** appropriée réussisse à filtrer l'exception. Ceci permet d'éviter les subtilités des effets de bord dans le tas et des conditions de dépassement de la mémoire. Ainsi, une exception peut être représentée par une adresse d'un objet *loc* ou bien par son nom. Elle est modélisée comme suit :

$$\begin{array}{lcl} xcpt & = & \mathsf{XcptLoc}\ loc \\ & | & \mathsf{SysXcpt}\ xname \end{array}$$

Le tas *heap* est modélisé par une table qui associe des objets à des adresses, tandis que le cadre local d'appels *locals* est représenté par une table qui associe des valeurs à des noms de variables locales. Tous les deux sont modélisés de la façon suivante :

$$\begin{array}{lcl} heap & = & (loc, obj)table \\ locals & = & (ename, val)table \end{array}$$

La demande d'allocation d'une nouvelle adresse a dans le tas échoue lorsqu'il n'existe pas suffisamment d'espace mémoire pour l'allouer. Cette non-disponibilité de la mémoire est exprimée explicitement comme suit :

$$(\mathsf{heap}\ \sigma)\ a \quad = \quad \mathsf{None}$$

L'allocation de la mémoire est explicite, elle est exprimée grâce à la fonction suivante :

$$\mathsf{new\_Addr} \ :: \ heap \Rightarrow (loc \times (xcpt)option)option$$

$$\begin{array}{l} \mathsf{new\_Addr}\ h \stackrel{\mathsf{def}}{=} \varepsilon y.\ (y = \mathsf{None} \wedge (\forall a.\ h\ a \neq \mathsf{None})) \vee \\ \qquad\qquad (\exists a\ x.\ y = \mathsf{Some}\ (a, x) \wedge h\ a = \mathsf{None} \wedge \\ \qquad\qquad\qquad (x = \mathsf{None} \vee x = \mathsf{Some}\ (\mathsf{SysXcpt\ OutOfMemory}))) \end{array}$$

Cette fonction retourne **None** lorsqu'il n'existe pas un espace disponible dans le tas. Dans ce cas, l'allocation de mémoire échoue. Dans le cas contraire, la fonction retourne une adresse libre et, s'il ne reste qu'une seule adresse disponible, elle retourne également l'exception `OutOfMemory`. Ceci permet de s'assurer qu'il est toujours possible d'allouer de l'espace dans le tas lorsque cette exception est levée pour la première fois.

Un objet peut être une instance d'une classe ou d'un tableau. Dans le premier cas, l'objet peut être vu comme une paire constituée du nom de sa classe et d'une table représentant ses champs. Cette dernière associe une valeur à chaque paire composée du nom du champ et de la classe qui le déclare. Dans le deuxième cas, l'objet peut être vu comme une paire constituée du type des composantes du tableau et d'une table qui associe des valeurs (les éléments du tableau) à des entiers (les indices du tableau). Ainsi, un objet est introduit de la façon suivante :

$$\begin{array}{lcl} fields & = & (ename \times ref\_ty, val)table \\ components & = & (int, val)table \end{array}$$

$$\begin{array}{lcl} obj & = & \mathsf{Obj}\ tname\ fields \\ & | & \mathsf{Arr}\ ty\ components \end{array}$$

Les différentes fonctions de projection permettant d'extraire les composantes d'un objet et de fournir son type sont présentées dans le tableau 2.14.

En utilisant la fonction **obj_ty**, il devient possible de définir un prédicat $\Gamma, \sigma \vdash v$ fits T, qui signifie que dans le contexte du programme Γ et de l'état σ, la valeur v peut être affectée à une variable de type T :

$$\_ , \_ \vdash \_ \ \mathsf{fits}\ \_ \ :: \ prog \Rightarrow st \Rightarrow val \Rightarrow ty \Rightarrow bool$$

the_Obj	::	$(obj)option \Rightarrow tname \times fields$
the_Arr	::	$(obj)option \Rightarrow ty \times components$
Obj_ty	::	$obj \Rightarrow ty$
the_Obj (Some (Obj C fs))	=	(C, fs)
the_Arr (Some (Arr T cs))	=	(T, cs)
obj_ty (Obj C fs)	=	Class C
obj_ty (Arr T cs)	=	$T[]$

TAB. 2.14: Fonctions de projection sur les objets

lupd$[v \mapsto x]$ (h, l)	$\stackrel{def}{=}$	$(h, l[v \mapsto x])$
hupd$[a \mapsto obj]$ (h, l)	$\stackrel{def}{=}$	$(h[a \mapsto obj], l)$
x_case x σ' σ	$\stackrel{def}{=}$	$(x,$ if $x =$ None σ' else $\sigma)$
init_vars	$\stackrel{def}{=}$	table_of $\circ$ map $(\lambda(n, T).\ (n,$ default_val $T))$
init_Obj Γ C	$\stackrel{def}{=}$	Obj C (init_vars (fields Γ C))
init_Arr T i	$\stackrel{def}{=}$	Arr T $(\lambda j.$ if $0 \leq j \wedge j < i$ then Some (default_val T) else None)
raise_if c xn xo	$\stackrel{def}{=}$	if $c \wedge (xo =$ None) then Some (SysXcpt xn) else xo
np v	$\stackrel{def}{=}$	raise_if $(v =$ Null) `NullPointer`

TAB. 2.15: Fonctions de construction et de mise à jour de l'état

$$\Gamma, \sigma \vdash v \text{ fits } T \stackrel{def}{=} (\exists pt.\ T = \text{PrimT } pt) \vee v = \text{Null} \vee \Gamma \vdash \text{obj\_ty (the (heap } \sigma \text{ (the\_Addr } v))) \preceq T$$

Enfin, il existe plusieurs fonctions pour la construction et la mise à jour de l'état qui sont présentées dans le tableau 2.15. Ces fonctions exigent quelques explications que nous donnons ci-dessous.

- La fonction **lupd** permet d'ajouter à l'état une nouvelle variable locale v ayant une valeur x. Il suffit d'ajouter une nouvelle association à la table **locals** des variables locales :

  lupd$[_ \mapsto _]_$ :: $ename \Rightarrow val \Rightarrow st \Rightarrow st$

- La fonction **hupd** permet d'ajouter à l'état un nouvel objet obj ayant l'adresse a. Une nouvelle association est donc ajoutée à la table **heap** :

  hupd$[_ \mapsto _]_$ :: $loc \Rightarrow obj \Rightarrow st \Rightarrow st$

- La fonction **x_case** fait un choix entre deux états selon la valeur d'une exception x :

$$\text{x\_case} \;::\; (xcpt)option \Rightarrow st \Rightarrow st \Rightarrow state$$

- La fonction **init_vars** initialise une liste de variables locales. Celle-ci conserve une paire pour chaque variable constituée du nom et du type de la variable. La fonction **map** applique une fonction à toutes les paires de cette liste et retourne des paires composées des noms des variables et de leur valeur par défaut et ce, grâce à **default_val**. Enfin cette liste est transformée en une table :

$$\text{init\_vars} \;::\; (\alpha \times ty)list \Rightarrow (\alpha, val)table$$

- La fonction **init_Obj** initialise les objets nouvellement créés. Ceci revient à initialiser leurs champs en appelant la fonction **init_vars** :

$$\text{init\_Obj} \;::\; prog \Rightarrow tname \Rightarrow obj$$

- La fonction **init_Arr** ressemble à la précédente, sauf qu'elle effectue une vérification supplémentaire afin de ne pas dépasser les bornes du tableau :

$$\text{init\_Arr} \;::\; ty \Rightarrow int \Rightarrow obj$$

- La fonction **raise_if** permet de propager une exception xo si elle a été préalablement levée, sinon, et si une certaine condition c est vraie (cette condition permet de vérifier s'il est nécessaire de lever une exception système), alors elle soulève une exception système xn fournie explicitement en argument à cette fonction. Toutefois, les exceptions déjà survenues sont toujours prioritaires. Cette condition est vérifiée par l'expression $(xo = \textbf{None})$:

$$\text{raise\_if} \;::\; bool \Rightarrow xname \Rightarrow (xcpt)option \Rightarrow (xcpt)option$$

- La fonction **np** est une application de la fonction précédente. Elle vérifie si l'accès à un pointeur nul se fait via la valeur v en levant une exception **NullPointer** dans ce cas :

$$\text{np} \;::\; val \Rightarrow (xcpt)option \Rightarrow (xcpt)option$$

Les deux prochaines sections sont dédiées à la présentation des règles d'évaluation opérationnelle des expressions et des instructions de BALI.

2.10.2 Format des règles d'évaluation

La relation d'évaluation est définie par un ensemble de règles d'inférence qui utilisent des relations qui sont données sous la forme de prédicats. La relation utilisée dans les règles de typage des instructions est présentée comme suit :

$$\Gamma \vdash \sigma - c \rightarrow \sigma' \;::\; prog \Rightarrow state \Rightarrow stmt \Rightarrow state \Rightarrow bool$$

qui signifie que dans le contexte du programme Γ, l'exécution de l'instruction c transforme l'état σ en l'état σ'. Quant à la relation utilisée dans les règles de typage des expressions, elle est définie comme suit :

$$\Gamma \vdash \sigma - e \rhd v \rightarrow \sigma' \;::\; prog \Rightarrow state \Rightarrow expr \Rightarrow val \Rightarrow state \Rightarrow bool$$

signifie que dans le contexte du programme Γ, en s'évaluant en v, l'expression e transforme l'état σ en l'état σ'.

Il est évident qu'à l'état initial, il n'est pas nécessaire d'inclure une exception. De même, une expression retourne ou bien une valeur ou bien une exception et non les deux en même temps.

Les auteurs ne distinguent pas le cas où une exception est levée dans un état intermédiaire dans les règles de typage. Afin d'élucider ce point, nous présentons un exemple extrait de [18] qui illustre l'évaluation des instructions séquentielles en distinguant le cas où une exception est levée et le cas où aucune ne l'est. Supposons que la relation $\Gamma \vdash \sigma \ -c \rightarrow \sigma'$ est de type $prog \Rightarrow st \Rightarrow stmt \Rightarrow state \Rightarrow bool$. Rappelons qu'aucune exception n'est présente à l'état initial ce qui explique le fait que σ est de type st. Les règles de typage de la composition séquentielle devraient être :

$$\frac{\Gamma \vdash \sigma_0 - c_1 \rightarrow (\mathsf{None}, \sigma_1); \quad \Gamma \vdash \sigma_1 - c_2 \rightarrow \sigma_2}{\Gamma \vdash \sigma_0 - c_1; c_2 \rightarrow \sigma_2}$$

$$\frac{\Gamma \vdash \sigma_0 - c_1 \rightarrow (\mathsf{Some}\ xs, \sigma_1)}{\Gamma \vdash \sigma_0 - c_1; c_2 \rightarrow (\mathsf{Some}\ xs, \sigma_1)}$$

Cependant, les auteurs évitent cette redondance dans les règles d'évaluation en évitant cette distinction. Pour ce faire, ils ont introduit deux règles générales qui stipulent que lors de l'évaluation d'une série d'instructions ou d'expressions en présence d'exceptions, ces dernières se propagent. Ces règles sont les suivantes :

$$\overline{\Gamma \vdash (\mathsf{Some}\ xc, \sigma) \ -c \rightarrow (\mathsf{Some}\ xc, \sigma)}$$

$$\overline{\Gamma \vdash (\mathsf{Some}\ xc, \sigma) \ -e \rhd \mathsf{arbitrary} \rightarrow (\mathsf{Some}\ xc, \sigma)}$$

Dans toutes les règles d'évaluation, l'état initial sera décrit par **Norm** σ qui est une abréviation de (**None**, σ). Ceci est possible puisqu'à l'état initial il n'y a pas d'exception.

2.10.3 Règles d'évaluation opérationnelle

Dans cette section, nous présentons les règles d'évaluation pour les instructions et les expressions.

Règles d'évaluation des instructions

Les règles d'évaluation concernant les instructions sont présentées dans le tableau 2.16. Nous donnons ci-dessous quelques explications relatives à ces règles.

- Dans la règle(SEQN$_D$), l'évaluation d'une séquence d'instructions est exprimée par l'évaluation successive des instructions qui la composent.
- La règle (EXPR$_D$) stipule que l'instruction **Expr** est exécutée en évaluant l'expression correspondante. Si celle-ci retourne une valeur, alors elle est ignorée. L'évaluation de cette instruction dépend entièrement de celle de l'expression **Expr**.

$$(\text{SKIP}_D)\quad \Gamma \vdash \text{Norm } \sigma \ -\text{Skip} \rightarrow \text{Norm } \sigma$$

$$(\text{SEQN}_D)\quad \frac{\Gamma \vdash \text{Norm } \sigma_0 \ -c_1 \rightarrow \sigma_1;\ \Gamma \vdash \sigma_1 \ -c_2 \rightarrow \sigma_2}{\Gamma \vdash \text{Norm } \sigma_0 \ -c_1;c_2 \rightarrow \sigma_2}$$

$$(\text{EXPR}_D)\quad \frac{\Gamma \vdash \text{Norm } \sigma_0 \ -e \rhd v \rightarrow \sigma_1}{\Gamma \vdash \text{Norm } \sigma_0 \ -\text{Expr } e \rightarrow \sigma_1}$$

$$(\text{IF-ELSE}_D)\quad \frac{\Gamma \vdash \text{Norm } \sigma_0 \ -e \rhd v \rightarrow \sigma_1;\quad \Gamma \vdash \sigma_1 \ -\text{if the\_Bool } v \text{ then } c_1 \text{ else } c_2 \rightarrow \sigma_2}{\Gamma \vdash \text{Norm } \sigma_0 \ -\mathbf{if}(e)\ c_1\ \mathbf{else}\ c_2 \rightarrow \sigma_2}$$

$$(\text{WHILE}_D)\quad \frac{\Gamma \vdash \text{Norm } \sigma_0 \ -\mathbf{if}(e)\ (c;\ \mathbf{while}(e)\ c)\ \mathbf{else}\ \text{SKIP} \rightarrow \sigma_1}{\Gamma \vdash \text{Norm } \sigma_0 \ -\mathbf{while}(e)\ c \rightarrow \sigma_1}$$

$$(\text{THROW}_D)\quad \frac{\Gamma \vdash \text{Norm } \sigma_0 \ -e \rhd a' \rightarrow (x_1,\sigma_1);\quad x_1' = \text{np } a'\ x_1;\quad x_1'' = (\text{if } x_1' = \text{None then (Some (XcptLoc (the\_Addr } a'))) \text{ else } x_1')}{\Gamma \vdash \text{Norm } \sigma_0 \ -\mathbf{throw}\ e \rightarrow (x_1'',\sigma_1)}$$

$$(\text{TRY-CATCH}_D)\quad \frac{\begin{array}{l}\Gamma \vdash \text{Norm } \sigma_0 - c_1 \rhd \sigma_1;\quad \Gamma \vdash \sigma_1 - \text{salloc} \rhd (x_1',\sigma_1');\\ \text{case } x_1' \text{ of None} \Rightarrow \sigma_1'' = (x_1',\sigma_1') \wedge c_2' = \text{Skip}\\ \quad \mid \text{Some } xc \Rightarrow \text{let } a = \text{Addr(the\_XcptLoc } xc) \text{ in}\\ \quad\quad \text{if } \Gamma,\sigma_1' \vdash a \text{ fits Class } tn\\ \quad\quad \text{then } \sigma_1'' = \text{Norm (lupd}[vn \mapsto a]\sigma_1') \wedge c_2' = c_2\\ \quad\quad \text{else } \sigma_1'' = (x_1',\sigma_1') \wedge c_2' = \text{SKIP};\\ \Gamma \vdash \sigma_1'' - c_2' \rightarrow \sigma_2\end{array}}{\Gamma \vdash \text{Norm } \sigma_0 - (\mathbf{try}\ c_1\ \mathbf{catch}(tn\ vn)\ c_2) \rightarrow \sigma_2}$$

$$(\text{FINALLY}_D)\quad \frac{\Gamma \vdash \text{Norm } \sigma_0 \ -c_1 \rightarrow (x_1,\sigma_1);\quad \Gamma \vdash \text{Norm } \sigma_1 \ -c_2 \rightarrow (x_2,\sigma_2);\quad x_2' = (\text{if } x_1 \neq \text{None} \wedge x_2 = \text{None then } x_1 \text{ else } x_2)}{\Gamma \vdash \text{Norm } \sigma_0 \ -(c_1\ \mathbf{finally}\ c_2) \rightarrow (x_2',\sigma_2)}$$

TAB. 2.16: Règles d'évaluation des instructions

- La règle (IF-ELSE$_D$) évalue l'instruction conditionnelle **if _ else _**. Ceci revient à évaluer, en premier lieu, l'expression e. Ensuite, si cette évaluation se termine normalement, et selon la valeur v de e, l'une des instructions c_1 et c_2 est exécutée.
- La règle (WHILE$_D$) permet d'évaluer une boucle **while**. Ceci revient à évaluer une séquence d'instructions **if _ else _** jusqu'à ce que la condition de la boucle devienne fausse. À ce moment, l'instruction vide SKIP est exécutée.
- La règle (THROW$_D$) permet d'évaluer l'instruction **throw** e. Si aucune exception ne se produit lors de l'évaluation de l'expression e et que l'objet d'exception a', obtenu par l'évaluation de cette expression, n'est pas la référence nulle (vérifiée grâce à la fonction np), alors l'instruction copie l'adresse de l'objet d'exception dans la composante exception de l'état. Sinon, c'est l'exception levée par l'évaluation de l'expression e qui est copiée dans celui-ci.
- L'évaluation de l'instruction **try _ catch _** est représentée dans la règle (TRY-CATCH$_D$). Afin d'évaluer cette instruction, il faut distinguer le cas où une exception a été levée et par la suite capturée ou non par une clause **catch**. Dans le premier cas, la clause **catch** doit être exécutée en lui passant comme paramètre la valeur de l'exception capturée. Dans le deuxième cas, la clause **catch** doit être ignorée. Ceci est modélisé dans la règle par l'exécution de l'instruction Skip. Constatez que dans ce dernier cas, l'état ne change pas et ce, en positionnant σ_2 à (x_1', σ_1'). D'une manière plus détaillée, l'instruction de la clause **try** est évaluée en premier et retourne un nouvel état σ_1. Il faut ensuite vérifier que l'objet d'exception résultant de l'évaluation de cette clause, retourné dans σ_1, peut être affecté au type du paramètre de l'exception de la clause **catch**. Ceci est vérifié par le prédicat fits. Cependant, étant donné que l'allocation des exceptions systèmes est différée lors de l'évaluation des expressions, il faut s'assurer qu'un objet d'exception est alloué dans le tas pour cette exception. Ceci est réalisé par la relation $\Gamma \vdash \sigma - \mathsf{salloc} \rightarrow \sigma' :: \mathit{prog} \Rightarrow \mathit{state} \Rightarrow \mathit{bool}$ définie comme suit :

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ - \mathsf{salloc} \rightarrow \mathsf{Norm}\ \sigma}$$

$$\frac{}{\Gamma \vdash (\mathsf{Some}(\mathsf{XcptLoc}\ a), \sigma) - \mathsf{salloc} \rightarrow (\mathsf{Some}(\mathsf{XcptLoc}\ a), \sigma)}$$

$$\frac{\begin{array}{c}\mathsf{new\_Addr}(\mathsf{heap}\ \sigma) = \mathsf{Some}\ (a, x);\\ xobj = \mathsf{init\_Obj}\ \Gamma\ (\mathsf{SXcpt}(\mathsf{if}\ x = \mathsf{None}\ \mathsf{then}\ xn\ \mathsf{else}\ \mathtt{OutOfMemory}))\end{array}}{\Gamma \vdash (\mathsf{Some}(\mathsf{SysXcpt}\ xn), \sigma) - \mathsf{salloc} \rightarrow (\mathsf{Some}(\mathsf{XcptLoc}\ a), \mathsf{hupd}[a \mapsto xobj]\sigma)}$$

  Si aucune exception n'est levée, la relation ne change pas l'état, sinon elle alloue un objet d'exception correspondant à l'exception levée et modifie l'état en conséquence. Il convient de remarquer que l'allocation n'aura pas lieu, et par conséquent l'exécution du programme s'arrête, lorsqu'il n'y a aucune adresse libre.
- La règle (FINALLY$_D$) ressemble à la composition séquentielle, sauf que l'exécution de la deuxième clause de la composition dépend du fait qu'une exception soit levée dans la première clause ou non. Si une exception se produit dans l'une des clauses,

alors elle est levée après l'exécution de l'instruction _ **finally** _. Sinon, dans le cas où les deux clauses lèvent des exceptions, la première a la priorité.

Règles d'évaluation des expressions

Les règles d'évaluation des expressions sont présentées dans les tableaux 2.17 et 2.18. Nous donnons dans ce qui suit quelques explications les concernant.

- La règle (NEW-CLASS$_D$) permet d'évaluer une expression de création d'un nouvel objet instance d'une classe. Ceci revient à trouver une adresse a libre, à l'aide de la fonction **New_Addr**, et de mettre à jour le tas à cette adresse avec le nouvel objet. Les champs de celui-ci doivent être initialisés par des valeurs par défaut selon leur type. Remarquez que la règle n'est pas applicable si la fonction **New_Addr** échoue ; dans ce cas **New_Addr** retourne **None**. La mise à jour de l'état par le nouvel objet n'a lieu que lorsqu'il y a suffisamment d'espace mémoire pour stocker l'objet. Ceci est assuré par la fonction x_case utilisée dans la conclusion de la règle.
- La règle (NEW-ARRAY$_D$) suit le même principe que la règle précédente, sauf qu'elle effectue une vérification supplémentaire sur la taille du tableau qui ne doit pas être négative.
- Une coercition de types, représentée par la règle (CAST$_T$), retourne la valeur v de son paramètre e ayant subit la coercition. Elle lève une exception **ClassCast** dans le cas où la coercition n'est pas permise, c'est-à-dire que la valeur v de e ne peut pas être affectée à une variable de type T. Ceci est assuré par la fonction fits.
- La règle (INSTANCEOF$_D$) s'assure que la valeur v de e n'est pas nulle et qu'elle peut être affectée à une variable de type T.
- La règle (LITERAL$_D$) stipule que le résultat de l'évaluation d'un littéral est simplement sa valeur.
- L'accès à une variable locale ou au pointeur **this** est représenté par la règle (LOCAL-VAR1$_D$). La valeur accédée est lue à partir de la table des variables locales.
- La règle (LOCAL-VAR2$_D$) permet d'évaluer l'affectation d'une variable locale. Si l'exécution de la sous-expression e ne lève pas une exception, alors l'affectation de celle-ci à une variable locale met à jour l'état par la nouvelle valeur de la variable.
- La règle (FIELD1$_D$) évalue l'accès à un champ appelé fn. Elle lit la valeur de celui-ci à partir de la table contenant les champs membres de l'objet a', qui accède au champ fn, en se servant de l'annotation de type T déterminée statiquement et qui fournit la classe définissant le champ fn. Elle vérifie également que l'accès au champ ne se fait pas via une référence nulle (grâce à np) en retournant une exception **NullPointer** dans ce cas. Toutefois, toute autre exception produite lors de l'évaluation de l'expression e détient la priorité.
- Le même principe qui est utilisé dans la règle précédente est adopté pour l'évaluation d'une affectation d'un champ modélisée par la règle (FIELD2$_D$). En plus, la table des champs fs de l'objet qui contient la déclaration du champ fn doit être

$$\text{(NEW-CLASS}_\text{D}\text{)}\quad \frac{\mathsf{New\_Addr\ (heap\ }\sigma\mathsf{) = Some\ }(a,x)}{\Gamma \vdash \mathsf{Norm\ }\sigma\ \mathsf{-New\ }C \rhd \mathsf{Addr\ }a \to \mathsf{x\_case\ }x\ \mathsf{(hupd}[a \mapsto \mathsf{init\_Obj\ }\Gamma\ C]\sigma)\ \sigma}$$

$$\text{(NEW-ARRAY}_\text{D}\text{)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}e \rhd i' \to (x_1,\sigma_1);\ i = \mathsf{the\_Intg\ }i';\\ \mathsf{New\_Addr\ (heap\ }\sigma_1\mathsf{) = Some\ }(a,x);\\ x_1' = \mathsf{raise\_if\ }(i<0)\ \mathtt{NegArrSize}\ \mathsf{(if\ }x_1 = \mathsf{None\ then\ }x\ \mathsf{else\ }x_1)\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-New\ }T[e] \rhd \mathsf{Addr\ }a \to \mathsf{x\_case\ }x_1'\\ \mathsf{(hupd}[a \mapsto \mathsf{init\_Arr\ }T\ i]\sigma_1)\ \sigma_1\end{array}}$$

$$\text{(CAST}_\text{D}\text{)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}e \rhd v \to (x_1,\sigma_1);\\ x_1' = \mathsf{raise\_if}(\neg\ \Gamma,\sigma_1 \vdash v\ \mathsf{fits}\ T)\ \mathtt{ClassCast}\ x_1\end{array}}{\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}(T)e \rhd v \to (x_1',\sigma_1)}$$

$$\text{(INSTANCEOF}_\text{D}\text{)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}e \rhd v \to \sigma_1;\\ b = (v \neq \mathsf{Null}\ \wedge\ \Gamma, \mathsf{snd\ }\sigma_1 \vdash v\ \mathsf{fits\ RefT}\ T)\end{array}}{\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}e\ \mathtt{instanceof}\ T \rhd \mathsf{Bool\ }b \to \sigma_1}$$

$$\text{(LITERAL}_\text{D}\text{)}\quad \frac{}{\Gamma \vdash \mathsf{Norm\ }\sigma\ \mathsf{-Lit\ }v \rhd v \to \mathsf{Norm\ }\sigma}$$

$$\text{(LOCAL-VAR1}_\text{D}\text{)}\quad \frac{}{\Gamma \vdash \mathsf{Norm\ }\sigma\ \mathsf{-}vn \rhd \mathsf{the\ (locals\ }\sigma\ vn) \to \mathsf{Norm\ }\sigma}$$

$$\text{(LOCAL-VAR2}_\text{D}\text{)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}e \rhd v \to (x_1,\sigma_1);\\ \sigma_1' = \mathsf{(if\ }x = \mathsf{None\ then\ lupd}[vn \mapsto v]\ \sigma_1\ \mathsf{else\ }\sigma_1)\end{array}}{\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}vn\ \mathsf{:=}e \rhd v \to (x,\sigma_1')}$$

$$\text{(FIELD1}_\text{D}\text{)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}e \rhd a' \to (x_1,\sigma_1);\\ v = \mathsf{the\ (snd\ (the\_Obj\ (heap\ }\sigma_1\ \mathsf{(the\_Addr\ }a'))))\ (fn,T))\end{array}}{\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}\{T\}e.fn \rhd v \to \mathsf{(np\ }a'\ x_1,\sigma_1)}$$

$$\text{(FIELD2}_\text{D}\text{)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}e_1 \rhd a' \to (x_1,\sigma_1);\ a = \mathsf{the\_Addr\ }a';\\ \Gamma \vdash \mathsf{(np\ }a'\ x_1,\sigma_1)\ \mathsf{-}e_2 \rhd v \to (x_2,\sigma_2);\\ (c,fs) = \mathsf{the\_Obj\ (heap\ }\sigma_2\ a);\ obj = \mathsf{Obj\ }c\ (fs[(fn,T) \mapsto v])\end{array}}{\Gamma \vdash \mathsf{Norm\ }\sigma_0\ \mathsf{-}(\{T\}e_1.fn\ \mathsf{:=}e_2) \rhd v \to \mathsf{x\_case\ }x_2\ \mathsf{(hupd}[a \mapsto obj]\sigma_2)\ \sigma_2}$$

TAB. 2.17: Règles d'évaluation des expressions : partie 1

$$(\text{ARRAY1}_\text{D}) \quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1 \rhd a' \to \sigma_1;\ \Gamma \vdash \sigma_1\ -e_2 \rhd i' \to (x_2, \sigma_2);\\ vo = \mathsf{snd}\ (\mathsf{the\_Arr}\ (\mathsf{heap}\ \sigma_2\ (\mathsf{the\_Addr}\ a')))\ (\mathsf{the\_Intg}\ i');\\ x_2' = \mathsf{raise\_if}\ (vo = \mathsf{None})\ \mathtt{IndOutBound}\ (\mathsf{np}\ a'\ x_2)\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1[e_2] \rhd \mathsf{the}\ vo \to (x_2', \sigma_2)}$$

$$(\text{ARRAY2}_\text{D}) \quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1 \rhd a' \to \sigma_1;\ a = \mathsf{the\_Addr}\ a';\\ \Gamma \vdash \sigma_1\ -e_2 \rhd i' \to \sigma_2;\ i = \mathsf{the\_Intg}\ i';\\ \Gamma \vdash \sigma_2\ -e_3 \rhd v \to (x_3, \sigma_3);\\ (T, cs) = \mathsf{the\_Arr}\ (\mathsf{heap}\ \sigma_3\ a);\ obj = \mathsf{Arr}\ T\ (cs[i \mapsto v]);\\ x_3' = \mathsf{raise\_if}\ (\neg\ \Gamma, \sigma_3 \vdash v\ \mathsf{fits}\ T)\ \mathtt{ArrStore}\ (\\ \mathsf{raise\_if}\ (cs\ i = \mathsf{None})\ \mathtt{IndOutBound}\ (\mathsf{np}\ a'\ x_3))\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(e_1[e_2] := e_3) \rhd v \to (\mathsf{x\_case}\ x_3'\ \mathsf{hupd}[a \mapsto obj]\sigma_3)\ \sigma_3}$$

$$(\text{METHOD}_\text{D}) \quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e \rhd a' \to \sigma_1;\\ \Gamma \vdash \sigma_1 - p \rhd pv \to (x_2, \sigma_2);\\ dynT = \mathsf{fst}\ (\mathsf{the\_Obj}\ (\mathsf{heap}\ \sigma_2\ (\mathsf{the\_Addr}\ a')));\\ (md, (pn, rT), lvars, blk, res) = \mathsf{the}\ (\mathsf{cmethd}\ \Gamma\ dynT\ (mn, pT));\\ \Gamma \vdash (\mathsf{np}\ a'\ x_2, (\mathsf{heap}\ \sigma_2, \mathsf{init\_vars}\ lvars[\mathtt{this} \mapsto a'][pn \mapsto pv])) - blk \to \sigma_3;\\ \Gamma \vdash \sigma_3\ -res \rhd v \to (x_4, \sigma_4)\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(e.mn(\{pT\}p)) \rhd v \to (x_4, (\mathsf{heap}\ \sigma_4,\ \mathsf{locals}\ \sigma_2))}$$

TAB. 2.18: Règles d'évaluation des expressions : partie 2

mise à jour par la nouvelle valeur v de celui-ci. Comme dans les règles (NEW-CLASS$_D$) et (NEW-ARRAY$_D$), la fonction x_case utilisée dans la conclusion de la règle assure que la mise à jour n'aura lieu que lorsqu'aucune exception n'a été levée et que l'objet a' qui accède au champ n'est pas la référence nulle.

- La règle (ARRAY1$_D$) permet de typer une expression d'accès à un élément d'un tableau. Elle vérifie que l'entier i' correspond bien à un indice valide du tableau et lève une exception **IndOutBound** dans le cas contraire. Toutefois, cette exception n'est levée que si aucune autre exception ne se produit lors de l'évaluation de e_1 et de e_2, et que l'expression de référence au tableau a' n'est pas nulle.
- Une évaluation d'une expression d'affectation d'un tableau est exprimée par la règle (ARRAY2$_D$). Celle-ci met à jour la composante appropriée après avoir effectué quelques vérifications :
  - elle évalue les sous-expressions e_1, e_2 et e_3 ;
  - elle vérifie que a' n'est pas une référence nulle, sinon elle lève une exception **NullPointer** ;
  - elle s'assure que i coïncide avec un indice valide du tableau, sinon elle lève l'exception **IndOutBound** ;
  - elle vérifie que la valeur de la variable v peut être affectée à une variable de type T des éléments du tableau et elle lève une exception **ArrStore** dans le cas contraire.

  La mise à jour de l'état n'a lieu que lorsqu'aucune exception ne résulte de l'évaluation de toutes les sous-expressions.
- La règle (METHOD$_D$) représente l'évaluation d'un accès à une méthode et elle est la plus complexe. Elle évalue, en premier lieu, l'expression e en l'adresse a' de l'objet et le paramètre p de la méthode en la valeur pv. Ensuite, la fonction cmethd, invoquée avec le programme courant ainsi que $dyrT$, type de l'objet stocké à l'adresse a', extrait la table des méthodes membres de ce dernier. L'accès à cette table, à l'aide de la signature de la méthode (nom de la méthode mn et type du paramètre pT), permet de récupérer les informations relatives à cette méthode. En particulier, la table de ses variables locales *lvars*, le bloc de ses instructions *blk* et son expression de retour *res*. Pour des raisons de simplicité, les variables locales doivent être initialisées avec leur valeur par défaut et ce, pour s'assurer qu'aucune variable n'est accédée avant son initialisation. Afin d'évaluer *blk* et *res*, il faut construire une nouvelle unité locale d'appels Λ qui doit contenir la table des variables locales de la méthode augmentée par l'association $[pn \mapsto pv]$, représentant son paramètre, et de l'association $[\texttt{this} \mapsto a']$ qui correspond à la classe courante. Enfin, et après l'exécution de *blk* et *res*, l'ancien cadre local d'appels est restauré et le résultat *res* est retourné.

2.11 Cohérence de la sémantique statique vis-à-vis de l'opérationnelle

La cohérence de la sémantique statique vis-à-vis de la sémantique opérationnelle revient à montrer que le type de toute valeur produite par l'exécution d'un programme correspond à son type fourni statiquement. Les auteurs expriment cette correction du typage comme suit : pour toutes les transformations subies par l'état et engendrées par l'exécution d'une instruction ou l'évaluation d'une expression, si dans l'état initial toutes les variables sont *conformes* à leurs types respectifs, alors cette condition doit rester valide dans l'état final. En plus, si l'évaluation d'une expression rend une certaine valeur, alors cette valeur doit être *conforme* au type de l'expression. Évidemment, ceci doit être prouvé en supposant que le programme est bien formé et que les expressions et les instructions sont bien typées.

Afin de donner un sens à la définition donnée ci-dessus, il convient d'expliquer le mot *conforme*. Relativement à un programme Γ et un état σ, une valeur v est dite conforme à un type T, et nous écrivons $\Gamma, \sigma \vdash v ::\preceq T$, ssi le type dynamique de v peut être converti implicitement en T. Cette notion de conformité peut être généralisée afin de s'assurer qu'un état σ est conforme à un environnement E. En effet, la proposition $\sigma ::\preceq E$ signifie que la valeur de toute variable accessible dans l'état σ est conforme à son type statique. D'une manière plus formelle, il existe quatre relations de conformité définies comme suit :

- Une relation qui permet de vérifier qu'une valeur est conforme à un type :

$$\_,\_ \vdash \_ ::\preceq \_ \quad :: \quad prog \Rightarrow st \Rightarrow val \Rightarrow ty \Rightarrow bool$$

$$\Gamma, \sigma \vdash v ::\preceq T \quad \stackrel{\text{def}}{=} \quad \begin{array}{l} \mathsf{let}\ dyn\_ty = \mathsf{option\_map}\ obj\_ty \circ heap\ \sigma \\ \mathsf{in}\ \exists T'.\ \mathsf{typeof}\ dyn\_ty\ v = \mathsf{Some}\ T' \wedge \Gamma \vdash T' \preceq T \end{array}$$

  L'expression $(\mathsf{option\_map}\ obj\_ty \circ heap\ \sigma)\ a$ permet de calculer le type dynamique de l'objet qui possède l'adresse a dans le tas, s'il existe.

- Une relation qui vérifie que toutes les valeurs dans une table sont conformes à leurs types respectifs :

$$\_,\_ \vdash \_[::\preceq]\_ \quad :: \quad prog \Rightarrow st \Rightarrow (\alpha, val)table \Rightarrow (\alpha, ty)table \Rightarrow bool$$

$$\Gamma, \sigma \vdash vs[::\preceq]Ts \quad \stackrel{\text{def}}{=} \quad \begin{array}{l} \forall n\ T.\ Ts\ n = \mathsf{Some}\ T \longrightarrow \\ \quad (\exists v.\ vs\ n = \mathsf{Some}\ v \wedge \Gamma, \sigma \vdash v ::\preceq T) \end{array}$$

- Une relation qui vérifie que toutes les composantes d'un objet sont conformes à leurs types respectifs :

$$\_,\_ \vdash \_ ::\preceq \Diamond \quad :: \quad prog \Rightarrow st \Rightarrow obj \Rightarrow bool$$

$$\begin{array}{lcl} \Gamma, \sigma \vdash \mathsf{Obj}\ C\ fs ::\preceq \Diamond & = & \Gamma, \sigma \vdash fs[::\preceq]\mathsf{table\_of}(\mathsf{fields}\ \Gamma\ C) \\ \Gamma, \sigma \vdash \mathsf{Arr}\ T\ cs ::\preceq \Diamond & = & \Gamma, \sigma \vdash cs[::\preceq]\mathsf{option\_map}(\lambda i.\ T) \circ cs \end{array}$$

- Une relation qui vérifie qu'un état est conforme à un environnement :

$$\_ ::\preceq \_ \quad :: \quad state \Rightarrow env \Rightarrow bool$$

$$\begin{aligned}(x,\sigma) ::\preceq (\Gamma,\Lambda) \quad &\stackrel{def}{=} \quad \Gamma,\sigma \vdash \mathsf{locals}\ \sigma[::\preceq]\Lambda\wedge\\ &(\forall a\ obj.\ \mathsf{heap}\ \sigma\ a = \mathsf{Some}\ obj \longrightarrow \Gamma,\sigma \vdash obj ::\preceq \Diamond)\wedge\\ &(\forall a.\ x = \mathsf{Some}(\mathsf{XcptLoc\ a}) \longrightarrow \Gamma,\sigma \vdash \mathsf{Addr}\ a ::\preceq \mathsf{Class}(\mathsf{SXcpt\ Throwable}))\end{aligned}$$

Il convient de signaler que, telle qu'elle est définie, la relation de conformité ne tient pas compte des variables inaccessibles, c'est-à-dire les valeurs qui se trouvent dans l'état et qui ne possèdent pas de composantes respectives dans l'environnement. Entre autres, ceci permet d'éviter de libérer les paramètres d'exception après l'exécution de la clause `catch`.

Grâce aux relations introduites ci-dessus, il devient possible d'énoncer les propositions à prouver et qui expriment la correction du typage.

Proposition 2.11.1 *Dans le contexte d'un programme bien formé, l'exécution d'une instruction bien typée transforme un état conforme à l'environnement en un autre état également conforme à celui-ci :*

$$E = (\Gamma,\Lambda) \wedge \mathsf{wf\_prog}\ \Gamma \wedge E \vdash s :: \Diamond \wedge \sigma ::\preceq E \wedge \Gamma \vdash \sigma - s \rightarrow \sigma' \longrightarrow \sigma' ::\preceq E$$

Proposition 2.11.2 *Dans le contexte d'un programme bien formé, l'évaluation d'une expression bien typée transforme un état conforme à l'environnement en un autre état également conforme à cet environnement. En plus, à moins qu'une exception ne soit levée, la valeur de l'expression doit être conforme à son type trouvé statiquement :*

$$\begin{aligned}&E = (\Gamma,\Lambda) \wedge \mathsf{wf\_prog}\ \Gamma \wedge E \vdash e :: T \wedge \sigma ::\preceq E \wedge \Gamma \vdash \sigma - e \rhd v \rightarrow (x',\sigma') \longrightarrow\\ &(x',\sigma') ::\preceq E \wedge (x' = \mathsf{None} \longrightarrow \Gamma,\sigma' \vdash v ::\preceq T)\end{aligned}$$

Dans ce qui suit, nous présentons le théorème principal. La preuve de celui-ci n'a pas été fournie dans l'article. Les lemmes et les corollaires du théorème ne sont pas présentés dans ce mémoire. Pour les consulter, nous référons le lecteur à l'article [18].

Théorème 2.11.3 *Pour un programme Γ bien formé, si l'exécution d'une instruction transforme un état (x,σ) en un état (x',σ'), alors pour tout environnement de variables locales Λ, si l'instruction est bien typée relativement à l'environnement (Γ,Λ) et que l'état de départ (x,σ) est conforme à celui-ci, alors l'état final (x',σ') l'est aussi, et le nouveau tas σ' est une extension de l'ancien σ'. Ceci est valable également pour les expressions, mais en plus, si aucune exception n'est levée, la valeur de l'expression doit être conforme à son type trouvé statiquement. D'une manière plus formelle, le théorème de correction du typage est donné comme suit :*

$$\begin{aligned}&\mathsf{wf\_prog}\ \Gamma \longrightarrow\\ &\quad(\Gamma \vdash (x,\sigma) - c \rightarrow (x',\sigma') \longrightarrow\\ &\qquad\forall\Lambda.\ (x,\sigma) ::\preceq (\Gamma,\Lambda) \longrightarrow\\ &\qquad\quad(\Gamma,\Lambda) \vdash c :: \Diamond \longrightarrow\\ &\qquad\quad(x',\sigma') ::\preceq (\Gamma,\Lambda) \wedge \sigma \trianglelefteq \sigma')\\ &\wedge\\ &\quad(\Gamma \vdash (x,\sigma) - e \rhd v \rightarrow (x',\sigma') \longrightarrow\\ &\qquad\forall\Lambda.\ (x,\sigma) ::\preceq (\Gamma,\Lambda) \longrightarrow\\ &\qquad\forall T.\ (\Gamma,\Lambda) \vdash e :: T \longrightarrow\\ &\qquad\quad(x',\sigma') ::\preceq (\Gamma,\Lambda) \wedge \sigma \trianglelefteq \sigma' \wedge (x' = \mathsf{None} \longrightarrow \Gamma,\sigma' \vdash v ::\preceq T))\end{aligned}$$

La relation $\_ \trianglelefteq \_$, appelée *extension du tas*, est un pré-ordre sur les états de type $st \Rightarrow st \Rightarrow bool$, où $\sigma \trianglelefteq \sigma'$ signifie que tout objet existant dans le tas de σ existe aussi dans σ' et qu'il a le même type dans les deux. Cette relation est définie comme suit :

$$\sigma \trianglelefteq \sigma' \stackrel{\text{def}}{=} \forall a\ obj.\ \mathsf{heap}\ \sigma\ a = \mathsf{Some}\ obj \longrightarrow \exists obj'\, \mathsf{heap}\ \sigma'\ a = \mathsf{Some}\ obj' \wedge \mathsf{obj\_ty}\ obj' = \mathsf{obj\_ty}\ obj$$

2.12 Conclusion

Nous avons présenté dans ce chapitre un survol de l'état de l'art en matière de théories sémantiques pour Java et nous avons focalisé sur la formalisation d'un sous-ensemble de Java, BALI. Nous avons décrit sa syntaxe abstraite, son système de types, les conditions de bonne formation des déclarations d'un programme ainsi que sa sémantique opérationnelle. L'expressivité de Isabelle/HOL a permis de formaliser une partie de la spécification de Java d'une façon naturelle tout en gardant la présentation du formalisme assez claire et assez concise. En outre, la structuration du formalisme et la modularité des définitions et des preuves permettent une meilleure maintenabilité de la formalisation et rend l'extension du langage plus faisable. Cependant, une bonne compréhension du formalisme requiert certaines connaissances du langage ML qui est à la base de HOL.

Ce travail, ainsi que [6, 7, 8, 28, 29], sont certes très intéressants, ils nous ont permis de nous familiariser avec la sémantique formelle de Java et ils nous ont servi pour développer une sémantique statique d'un plus grand sous-ensemble de Java. Cependant, ces travaux comportent une importante limitation : ils ne traitent pas plusieurs caractéristiques importantes de Java, comme les modificateurs et le parallélisme. En plus, plusieurs constructions du langage Java ont été simplifiées. Il est vrai que ces travaux ont prouvé la sûreté de types d'un sous-ensemble de Java, mais aucun de tous ces langages désormais sûrs n'est Java.

Chapitre 3

Évaluation de la spécification et des théories sémantiques

Comme nous l'avons mentionné dans l'introduction de ce mémoire, le langage Java est très puissant. Il est, entres autres, fortement typé et il a été spécialement conçu pour supporter des applications qui requièrent un haut niveau de sûreté de programmation. Ceci a fait que le langage comporte plusieurs exceptions et a rendu sa sémantique assez subtile et peu habituelle. En plus, le langage est encore immature, il est instable et plusieurs modifications sont apportées à celui-ci que ce soit pour corriger des failles découvertes dans le langage ou pour l'enrichir avec d'autres caractéristiques. Il est sûrement intéressant d'augmenter la puissance du langage en le dotant de nouvelles caractéristiques et de donner plus de possibilités aux programmeurs. Cependant, ceci engendre la révision continuelle de ses théories sémantiques.

Étant donnée la subtilité du langage, la spécification de Java [11] est d'une grande aide pour la compréhension du langage et sa formalisation. Elle représente un avantage que d'autres langages n'offrent pas. Nous estimons que sans cette description, il est difficile de donner une théorie sémantique pour Java. Toutefois, la spécification n'est qu'une description textuelle et informelle qui contient plusieurs erreurs et qui est souvent ambiguë. En effet, afin de déterminer le comportement exact du compilateur Java, nous avons procédé à plusieurs tests. Là encore, le comportement de celui-ci varie selon la version. Il est vrai qu'il existe des efforts fournis pour corriger, clarifier et commenter la spécification que nous trouvons sur Internet [12, 14], mais il n'y a pas encore de version complète et révisée depuis 1996.

Nous essayons à travers ce chapitre de montrer la subtilité du langage et l'ambiguïté de sa description. Nous faisons également quelques remarques sur le travail présenté dans le chapitre précédent. Nous montrons comment plusieurs aspects importants et subtils du langage sont omis. Enfin, nous dégageons les difficultés que soulève la formalisation de ces derniers.

3.1 Spécification du langage Java

La spécification de Java décrit la syntaxe et la sémantique du langage. Elle est la référence de base pour les concepteurs de compilateurs Java et pour la formalisation de celui-ci. Étant donnée son importance, le fait qu'elle puisse être ambiguë et erronée est inadmissible. Malheureusement, la spécification n'est pas aussi fiable que nous le pensions, ce qui va être démontré dans la suite du chapitre.

3.1.1 Évolution du langage

La spécification du langage Java n'a pas été révisée depuis 1996. Depuis, les versions de compilateurs Java se sont succédées. Chacune offre un comportement qui est différent des autres sur plusieurs points, que ce soit pour enrichir ou corriger les versions antérieures. En effet, il existe plusieurs programmes qui sont refusés par le compilateur JDK 1.2 et acceptés par les versions antérieures, comme JDK 1.1.7. Ceci nous a amené à changer notre système de types, alors que nous étions sur le point de l'achever. Nous donnons dans ce qui suit deux exemples qui montrent l'instabilité du langage.

Héritage de deux méthodes avec la même signature

La spécification de Java énonce qu'il est possible pour une interface d'hériter plus d'une méthode ayant la même signature. En outre, elle stipule que l'invocation d'une méthode `f` via un type qui hérite de deux méthodes abstraites ayant la même signature résulte en une erreur de compilation, puisque l'accès est considéré ambigu. Ceci est également le comportement du compilateur JDK 1.1.7 et les versions antérieures. Cependant, il n'existe pas d'ambiguïté, puisque les deux déclarations possèdent la même signature et elles n'ont pas encore de définition. Cette erreur a été corrigée dans JDK 1.2. Ci-dessous, un exemple qui est accepté par le compilateur JDK 1.2 et non par JDK 1.1.7 :

```
interface I {
   void f ( ) ;
}
interface J {
   void f ( ) ;
}
interface K extends I , J { }
class Test {
   void f' ( K k ) {
      k.f( ) ;
   }
}
```

Cet exemple montre comment l'implantation du langage évolue, sans que la spécification suive cette évolution.

Instruction `synchronized` et référence nulle

La spécification énonce qu'une expression représentant l'argument de l'instruction `synchronized` doit être un type de références. Entre autres, cette expression peut être le littéral `null` dont le type est un type de références. Ceci est le comportement de JDK 1.1.7 et les versions antérieures à celui-ci, mais n'est pas celui de JDK 1.2. En effet, le programme suivant est refusé à la compilation par ce dernier :

```
class C {
   void f ( ) { synchronized ( null ) { } }
}
```

Nous ne comprenons pas la logique derrière cette restriction. Nous pensons cependant que l'implantation de Java est faite d'une façon à détecter dès la compilation les programmes qui seront de toute façon refusés à l'exécution. En effet, dans l'exemple précédent, le compilateur détecte l'existence d'un pointeur `null` et refuse le programme au lieu d'attendre l'étape de l'exécution qui devra lancer l'erreur `NullPointerException`. Cependant, nous pensons que cette restriction ne fait qu'augmenter les exceptions dans le langage et complique son traitement, surtout que l'exemple ci-dessous est accepté à la compilation même par JDK 1.2 :

```
class C {
   void f ( ) { synchronized ( (T)null ) { } }
```

où T est un type de références quelconque. L'exécution de ce programme résulte en l'erreur `NullPointerException` et le compilateur ne détecte pas que l'argument de `throw` s'évalue en la référence nulle. L'analyse de flot de données que réalise le compilateur afin de détecter les erreurs assez tôt au moment de la compilation est donc très limitée et ne fait qu'augmenter la subtilité et les exceptions dans la sémantique du langage. Cet exemple montre que la spécification est erronée et que la sémantique de Java est assez subtile.

Invocation de méthode versus les clauses `throws`

Lorsqu'une classe hérite de deux méthodes avec la même signature, il faut nous assurer que ces méthodes possèdent le même type de retour afin qu'une méthode puisse les implanter toutes les deux. Cependant, le compilateur JDK 1.1.7, et pas JDK 1.2, accepte l'exemple suivant :

```
interface I {
   char f( ) ;
}
interface J {
   void f( ) ;
}
interface K extends I , J { }
```

Cet exemple montre un cas où la spécification a pris de l'avance sur l'implantation de Java et où cette dernière s'est ajustée à la spécification. Ce genre d'exemple est en

faveur de la spécification, puisqu'il montre que celle-ci est plus fiable que l'implantation. Cependant, comme nous l'avons montré ci-dessus, dans plusieurs autres cas, la spécification ne suit pas l'évolution du langage ce qui la rend dépassée. Le problème est de deviner qui parmi l'implantation et la spécification faut-il croire !

3.1.2 Quelques erreurs découvertes dans la spécification

Lors du développement d'une sémantique statique pour Java, nous avons remarqué l'existence de quelques erreurs dans la spécification. Dans cette section, nous présentons quelques-unes de ces erreurs, connues et non connues.

Instruction `throw` et référence nulle

La spécification de Java énonce que l'expression relative à l'instruction **`throw`**, qui représente le type d'une exception levée par cette dernière, doit être un type de références qui peut être affecté à **`Throwable`**. Le type du littéral **`null`** peut être affecté à n'importe quel type de références, entre autres le type de la classe **`Throwable`**. Cependant, le programme suivant est refusé par toutes les versions du compilateur Java :

```
class C {
   void f ( )  { throw null ; }
}
```

Cet exemple montre que la spécification est erronée. Il montre également que la sémantique du langage est subtile. En effet, si l'expression est une coercition du littéral `null` vers un autre type de références : `(T)null`, alors le programme serait accepté par le compilateur. Pourtant, la valeur de cette expression est nulle.

Traitement de l'exception levée par l'instruction `throw`

La spécification de Java énonce que lorsqu'une instruction **`throw`** lance une exception autre que **`RuntimeException`**, **`Error`** ou l'une de leurs sous-classes (voir chapitre 4), la compilation du programme résulte en une erreur à moins qu'une de ces deux conditions soit vraie :

- L'instruction **`throw`** se trouve dans un bloc **`try`** d'une instruction **`try`**. En outre, le type de l'exception levée par **`throw`** peut être affecté au paramètre d'au moins une clause **`catch`** de l'instruction **`try`**. Dans ce cas, l'exception est dite capturée.
- L'instruction **`throw`** se trouve dans une méthode ou un constructeur. En outre, le type de l'exception peut être affecté au type d'au moins un type d'exception déclaré dans la clause **`throws`** du constructeur ou de la méthode.

Cependant, la spécification ne mentionne pas qu'il existe une troisième condition sous laquelle une exception est considérée traitée. Cette erreur a été toutefois corrigée dans [12]. Ceci peut se produire si l'instruction **`throw`** se trouve dans un bloc **`try`** ou `catch` d'une instruction **`try`** et que cette dernière possède un block **`final`** qui se termine anormalement (voir chapitre 4). Dans ce cas, l'exception levée ne pourra pas se propager en dehors de l'instruction et ne nécessite donc pas d'être capturée ou déclarée. Cette

condition est difficilement détectable si la spécification du langage ne la mentionne pas. Nous pensons que ce genre d'oubli fait que les théories sémantiques proposées jusqu'ici pour Java sont erronées.

Instructions inaccessibles

Le langage Java effectue une analyse de flots de données afin de détecter les instructions qui ne sont pas accessibles dans un programme. Ceci permet de déceler le code mort dans celui-ci. Le chapitre 14 de la spécification décrit les règles que suit le compilateur afin de trouver les instructions qui sont inaccessibles. Ce chapitre énonce clairement qu'un bloc **catch C** est accessible si et seulement si les deux conditions suivantes sont remplies :

- une expression ou une instruction **throw** dans le bloc **try** est accessible et elle lève une exception dont le type peut être affecté au paramètre de la clause **catch** ;
- il n'existe pas une autre clause **catch A** qui le précède dans la même instruction **try**, tel que le type du paramètre de **C** est le même ou bien un sous-type du type du paramètre de **A**.

Ceci est contradictoire avec l'implantation de Java. En effet, lorsque le bloc **try** est vide, et donc ne lève aucune exception, le bloc **catch** n'est pas considéré inaccessible. Ceci est le comportement de toutes les versions du compilateur. Ci-dessous un exemple de programme qui est accepté :

```
class A {
   void f( ) {
      try { } catch( Exception e) {
      System.out.println(" capturée ") ; }
   }
}
```

Les interfaces héritent de `Object`

Toutes les interfaces et les tableaux sont considérés comme des sous-types de la classe mère `Object`. C'est-à-dire que tous les membres de cette dernière sont accessibles à partir de n'importe quelle interface ou classe. Cependant, la spécification de Java énonce que les interfaces, contrairement aux classes, n'héritent qu'à partir des super-interfaces et qu'il n'existe pas une super-interface commune équivalente à la classe `Object` pour les interfaces. Cette erreur a été mentionnée dans [28, 29], mais le fait que les interfaces puissent hériter de `Object` n'a pas été décrit jusqu'à présent dans aucune formalisation de Java.

Accès à un champ

Nous avons remarqué l'existence d'une erreur et d'une ambiguïté dans la règle grammaticale pour l'accès à un champ. Afin d'expliquer ceci, nous donnons ci-dessous cette règle :

FieldAccess :
Primary . *Identifer*
super . *Identifier*

L'accès à un champ qui est un tableau est donné par une autre règle sous le nom de *ArrayAccess*. Cette distinction entre l'accès à un champ qui n'est pas un tableau et un autre qui est un tableau porte à confusion. Nous pensons que la grammaire devrait être modifiée, comme nous le faisons dans le chapitre suivant. En plus, l'accès à un champ en utilisant uniquement son nom simple : *Identifier* ou bien composé via une variable locale ou un paramètre qui est un objet : *ExpressionName*. *Identifier*, ne sont pas couverts par cette règle. Nous pouvons toutefois retrouver ces accès dans la règle *Name* qui, cependant, ne couvre l'accès que via une expression parenthésée : (*ExpressionName*) . *Identifier*. Ces ambiguïtés dans la description du langage compliquent davantage son traitement.

Erreurs dans les exemples

Nous avons également remarqué l'existence de quelques erreurs peu significatives dans les exemples donnés dans la spécification, mais qui portent des fois à confusion. À titre d'exemple, voici un programme tiré de la spécification :

```
interface Fish { int getNumberOfScales( ); }
interface Piano { int getNumberOfScales( ); }
class Tuna implements Fish, Piano {
   int getNumberOfScales( ) { return 91; }
}
```

Ce programme n'est pas correct, puisque la classe **Tuna** implante les méthodes `getNumberOfScales` de ses super-interfaces avec une méthode qui est plus privée qu'elles. En effet, les méthodes des interfaces sont par défaut publiques, alors que celles des classes permettent un accès par défaut **package**. Ceci n'est pas permis dans le langage Java.

3.1.3 Ambiguïté de la spécification

Plusieurs phrases dans la spécification de Java sont ambiguës et/ou contiennent des oublis. Afin de deviner ce qui est entendu par ces phrases, il a fallu faire plusieurs tests. Ici, nous ne présentons qu'un petit exemple d'ambiguïté afin de donner au lecteur une petite idée de ce que représente l'interprétation du langage naturel.

Le chapitre 16 de la spécification décrit l'affectation des variables locales et comment le compilateur s'y prend pour s'assurer qu'une variable locale ait été initialisée d'une manière sûre avant qu'elle ne soit utilisée. C'est-à-dire que dans le corps de la méthode ou du constructeur qui contient la définition d'une variable, il faut qu'il existe un chemin d'exécution entre la déclaration de la variable et son utilisation. Nous avons mis beaucoup de temps avant de deviner les étapes que suit le compilateur pour déterminer si une variable est initialisée d'une manière sûre. À titre d'exemple, ci-dessous une petite phrase extraite du chapitre 16 de la spécification :

> *V is definitely assigned after a local variable declaration statement that contains initializers if and only if either it is definitely assigned after the last initializer expression in the local variable statement or the last initializer expression in the declaration is in the declarator that declares V.*

Cette phrase est ambiguë et tout ce qu'elle veut exprimer est énoncé plus clairement par la phrase suivante :

> *Une variable locale V est considérée initialisée d'une manière sûre après l'exécution d'une expression d'initialisation d'une variable locale si et seulement si l'une des deux conditions suivantes est vraie :*
>
> - *la variable V est initialisée dans l'une des sous-expressions de l'expression d'initialisation ;*
> - *l'expression d'initialisation est l'expression d'initialisation de la variable V.*

Dans cette section, nous avons essayé de montrer l'inconsistance de la spécification ; il existe plusieurs autres erreurs ainsi que des intérogations quant à sa complétude et sa correction. Cependant, il ne faut pas ignorer qu'il existe sur Internet quelques corrections et clarifications qui viennent compléter et commenter la spécification. Cependant, ces dernières changent, puisque le langage lui-même évolue très rapidement et des erreurs surgissent et dans la spécification et dans l'implantation du langage. La description officielle de Java [11] reste le document de base pour formaliser le langage. Le fait qu'il n'existe pas une version complète et révisée de la spécification depuis 1996 n'est pas admissible, surtout qu'il est difficile de se retrouver avec tous les commentaires et les débats autour de celle-ci. En plus, la spécification s'étale sur à peu près huit cents pages. À titre d'exemple, la spécification du comportement statique d'une invocation de méthode est décrite sur six pages au niveau du typage et elle est traitée en trois étapes avec plusieurs cas et sous-cas, sans oublier qu'elle comporte huit erreurs.

Cette section montre que la spécification du langage, aussi bonne soit-elle, reste toujours de la prose et que quelques règles sont, sans aucun doute, plus facile à interpréter et à suivre que du texte.

3.2 Subtilité de la sémantique du langage

Comme nous l'avons mentionné au début de ce chapitre, la sémantique du langage Java est subtile et elle comporte plusieurs exceptions. Nous essayons dans cette section de donner une preuve de ce que nous avançons.

3.2.1 Héritage de plusieurs méthodes avec la même signature

Comme nous l'avons mentionné plus haut, lorsqu'une classe ou une interface hérite de deux méthodes `f` avec la même signature, l'appel de `f` n'est pas considéré ambigu. Cependant, lorsque ces deux méthodes déclarent des exceptions différentes, quelles exceptions la méthode appelante doit-elle déclarer ? À titre d'exemple, nous donnons le programme suivant :

```
interface I {
   void f ( )  throws X;
}
interface J {
   void f ( ) throws Y;
}
interface K extends I , J { }
class Test {
   void f' ( K k ) throws Z {
      k.f( ) ;
   }
}
```

Dans cet exemple, que doit comporter la liste des exceptions `Z` afin que les exceptions levées par l'invocation de l'une des méthodes ayant le même nom `f` soient traitées? Selon les corrections dans [12] et le comportement du compilateur JDK 1.2, mais pas celui des versions antérieures, la méthode `f'` doit déclarer les exceptions `X` ou bien une de leurs sous-classes si toutes les exceptions dans `X` sont des sous-types d'au moins une exception dans `Y` et vice versa. Si, par contre, `X` et `Y` ne sont pas reliés par un tel lien d'héritage, alors l'accès à la méthode ne devrait pas être permis. Dans JDK 1.2, afin d'éviter que ce problème d'invocation de méthode ne se produise, il n'est pas permis pour un type d'hériter de deux méthodes ayant la même signature si elles déclarent des exceptions qui ne sont pas reliées par un lien d'héritage. En effet, aucune méthode ne peut les implanter toutes les deux simultanément. Pour mieux comprendre, voici un autre exemple :

```
interface I {
   void f ( )  throws X , Y;
}
interface J {
   void f ( )  throws Z;
}
interface K extends I , J { }
```

ce programme n'est accepté par le compilateur JDK 1.2 que dans la mesure où `Z` est une sous-classe de `X` ou une sous-classe de `Y`. Si la méthode `f` de `J` ne déclare aucune exception le programme est aussi accepté.

La raison de cette restriction est qu'une méthode abstraite n'est utile que lorsqu'elle est implantée. Si l'interface `K` hérite de deux méthodes avec la même signature, il faut nous assurer qu'il est possible pour une méthode de les implanter toutes les deux. Cependant, ce qui n'est pas compréhensible c'est que cette restriction ne change rien à ce problème. En effet, dans le dernier exemple, même si `Z` n'est ni une sous-classe de `X` ni une sous-classe de `Y`, la méthode ayant la même signature que `f` et qui ne déclare pas d'exception peut implanter les deux méthodes `f` des interfaces `I` et `J`. À notre avis, ce problème devrait être résolu lorsqu'une méthode vient réellement les implanter, ainsi l'utilisateur aura plus de facilités et nous nous confronterons à de moindres exceptions dans la formalisation du langage. Cet exemple montre que même la dernière version de l'implantation de Java peut compliquer davantage la sémantique du langage.

3.2.2 Utilisation d'un champ qui n'est pas encore déclaré

Dans le langage Java, il n'est pas permis qu'une expression d'initialisation d'un champ contienne une référence vers un autre champ se trouvant dans la même classe ou interface que le premier et qui n'a pas encore été déclaré. Nous pensons que cette restriction sert à éviter une forme de circularité. Cependant, voici un exemple qui est accepté par le compilateur :

```
class C {
   int f ( ) { return b; }
   int a = f ( );
   int b = a;
}
```

Ce programme est circulaire, pourtant il est compilé sans erreurs et il s'exécute ainsi aussi sans erreurs. Par un petit test d'affichage, nous avons remarqué que la valeur de a et de b est zéro. Ceci nous mène à croire que la circularité dans l'initialisation des champs ne cause pas de problème, puisque les champs sont initialisés par défaut. Nous ne comprenons alors pas pourquoi il y a cette restriction dans le langage.

3.2.3 Coercition

Normalement, la coercition entre deux types de références n'est permise que si les deux types peuvent référencer un objet de la même classe à l'exécution. C'est pour cette raison que la coercition entre deux interfaces qui contiennent des méthodes abstraites avec la même signature et des types de retour différents n'est pas permise en Java. En effet, aucune classe ne pourra les implanter toutes les deux. Cependant, les classes abstraites et les interfaces peuvent contenir des méthodes abstraites qui possèdent la même signature et des types de retour différents. Pourtant, il n'existe aucune règle qui interdit la coercition entre de tels types.

3.2.4 Accessibilité

Comme nous l'avons mentionné dans la section précédente, la spécification donne les règles que suit le compilateur afin de trouver les instructions qui sont inaccessibles. Ces règles décrivent l'accessibilité de chaque instruction du langage, entre autres les instructions `if` et `while`. À titre d'exemple, l'instruction suivante n'est pas acceptée par le compilateur :

`while ( false )` *Instruction*

puisque l'instruction *Instruction* est inaccessible. Toutefois, lorsque nous déclarons une constante booléenne `b` qui possède la valeur `false` grâce au modificateur `final`, l'instruction suivante est acceptée par le compilateur :

`while ( b )` *Instruction*

Pourtant, la spécification énonce clairement qu'il suffit que la variable conditionnelle soit une constante dont la valeur est fausse. La section décrivant l'accessibilité dans la spécification fait référence à des constantes booléennes, alors que le compilateur ne

vérifie jamais ceci. Il teste tout simplement si l'expression est un littéral **true** ou **false**. Cette remarque est aussi valable pour les règles décrivant l'initialisation des variables locales. Cet exemple montre qu'il existe une différence entre la spécification et l'implantation du langage. Nous ne savons pas encore quel comportement il faut adopter, mais le bon sens nous porte à croire que la spécification décrit un comportement plus raisonnable.

Considérons maintenant l'exemple suivant :

```
while ( false );
int v = 5;
```

Cette instruction est compilée sans erreurs. Ceci signifie que l'instruction de déclaration de la variable v est accessible. À notre avis, ceci est normal, puisque l'instruction vide n'est pas considérée comme un code mort et donc l'instruction qui suit sera exécutée et elle est accessible. Cependant, ce qui n'est pas normal c'est que lorsque cette instruction est un bloc contenant l'instruction vide «{ ;}», le compilateur affiche une erreur de compilation, étant donné que l'instruction de déclaration de la variable est inaccessible. Toutefois, la spécification énonce clairement qu'une instruction **while** peut se terminer normalement si et seulement si l'une des deux conditions suivantes est vraie :

- l'instruction **while** est accessible et l'expression conditionnelle n'est pas une constante dont la valeur est vraie ;
- il existe une instruction **break** accessible qui permet de sortir du bloc **while**.

La spécification ne mentionne pas l'existence d'un cas particulier lorsque l'instruction qui représente le corps du **while** est l'instruction vide. Dans notre formalisation nous adoptons la description de la spécification, puisqu'elle comporte moins d'exceptions.

Nous donnons ci-dessous un autre exemple concernant l'accessibilité et les instructions conditionnelles.

La description des règles que suit le compilateur afin de détecter les instructions inaccessibles lors de l'exécution d'une instruction **if** n'est pas conventionnel. En effet, contrairement au traitement de l'instruction **while**, l'instruction suivante est acceptée par le compilateur :

```
if ( false ) Instruction
```

La spécification déclare que ceci permettra au programmeur de définir une variable déclarée avec le modificateur **final** comme suit :

```
final boolean b = false
```

qui sera ensuite utilisée comme une expression conditionnelle dans l'instruction **if**. L'idée est de laisser la possibilité à l'utilisateur de changer la valeur de **b** sans toutefois rien changer dans le programme. Ce qui est contradictoire c'est que lorsqu'il s'agit de déterminer si une variable est initialisée d'une façon sûre, le compilateur réagit différemment. En effet, si nous avons une initialisation d'une variable comme suit :

```
if ( true ) a = 8;
```

la variable **a** est considérée initialisée après l'exécution de cette instruction. Ceci est contradictoire et crée une non-uniformité dans le traitement des programmes Java.

Instruction `if-else`

Le compilateur possède un comportement incompréhensible vis-à-vis de l'évaluation de l'instruction **`if-else`**. Afin de comprendre ceci, nous donnons l'exemple suivant dont la compilation se termine avec succès :

```
class A {
   void f( ) {
      int a = 9;
      int b;
      int c;
      if ( true ) c = a else c = b
   }
}
```

Dans cet exemple, l'instruction **`else`** fait référence à la variable `b` qui n'a pas été initialisée. Ceci ne devrait pas être fait en Java. Cependant, cela nous porte à croire qu'étant donné que le compilateur est capable de détecter que l'instruction **`else`** ne sera jamais exécutée, il ne vérifie pas cette instruction. Toutefois, l'exemple suivant montre le contraire :

```
class A {
   void f( ) {
      int a = 9;
      int b;
      if ( true ) b = a else return 3;
   }
}
```

ce programme ne passe pas l'étape de la compilation avec succès parce que l'instruction `else` fait appel à une instruction **`return`** qui retourne une valeur entière, alors que la méthode est déclarée **`void`**. Ceci confirme que le compilateur n'a pas de comportement uniforme qui est en plus parfois inexplicable. Dans notre formalisation, nous typons toutes les instructions du programme même celles qui ne seront jamais exécutées.

Nous donnons dans ce qui suit un autre exemple concernant l'instruction **`if-else`**. Le programme suivant ne réussit pas l'étape de la compilation, puisque l'erreur `ClassNotFoundException` n'est pas traitée dans le corps de la méthode ni déclarée dans celle-ci :

```
class A {
   void f( ) {
      if ( true ) throw new InstanciationException( );
      else throw new ClassNotFoundException( );
   }
}
```

Cependant, normalement le compilateur sait que l'instruction **`else`** ne sera jamais exécutée et que cette exception ne sera jamais lancée.

Ci-dessus, nous avons donné quelques exemples qui montrent la subtilité de la sémantique de Java. Cependant, la formalisation de celui-ci dans le chapitre suivant montre davantage ces subtilités. Nous pensons que le compilateur ne peut pas être pris pour une spécification. Par exemple, les instructions non accessibles (code mort) ne devraient pas être décrites dans la spécification, il faut laisser une certaine flexibilité à l'auteur du compilateur. Ceci aurait l'avantage de garder la spécification simple, plus claire et permettrait de la formaliser d'une manière plus simple. La spécification devrait se contenter de décrire le langage plutôt que de spécifier les détails des messages d'erreurs. En effet, il existe certaines erreurs détectées par le compilateur et qui devraient être de simples avertissements. À titre d'exemple, lorsqu'une clause **catch** ne capture pas au moins une exception parmi les exceptions levées dans le bloc **try** la précédant, le compilateur devrait donner un avertissement et non un message d'erreur. Par ailleurs, un compilateur peut faire des analyses de flots de données poussées et complexes afin d'aider l'utilisateur, mais nous pensons qu'une spécification ne devrait pas décrire ce genre d'analyse.

Dans la formalisation que nous proposons dans le chapitre suivant, et afin d'augmenter la lisibilité des règles de typage, nous traitons l'instruction vide comme n'importe quelle autre instruction. En plus, même si une instruction **catch** ne capture aucune exception, nous ne considérons pas qu'elle est inaccessible.

3.3 Fondements théoriques de Java

Cette section présente quelques difficultés qu'engendre la formalisation du langage Java. En particulier, nous nous intéressons aux aspects du langage qui ont été omis dans la littérature. Nous émettons également quelques critiques concernant la formalisation de BALI présentée dans le chapitre 2. Une grande partie de ces critiques est valable pour les travaux de Drossopoulou, Eisenbach et Syme [8, 29]. Dans cette section, nous dégageons progressivement la valeur ajoutée de notre travail tout en soulevant les difficultés qu'engendre la mise au point d'une sémantique statique formelle qui couvre les aspects omis.

3.3.1 Quelques difficultés dans la formalisation de Java

Toutes les théories sémantiques qui ont été proposées pour Java au niveau du langage ont contribué à lever quelques ambiguïtés sur sa sémantique. Elles ont toutes du mérite, certes, toutefois plusieurs aspects très importants du langage ont été ignorés. Pourtant, plusieurs de ces caractéristiques omises ont contribué à faire de Java ce qu'il est aujourd'hui. Notamment, les threads qui sont considérés comme la clef de voûte du langage et les modificateurs d'accès qui renforcent la sécurité dans Java.

Parmi les aspects ignorés du langage Java, nous citons l'initialisation, les constructeurs, les variables et les méthodes de classe, les variables locales, les modificateurs d'accès, les champs des interfaces, le mot clé **super**, les chaînes de caractère, l'aspect parallèle d'accès, le traitement de plusieurs exceptions, certains types primitifs, les paquetages et les règles de portée des variables. En plus, parmi les aspects considérés,

plusieurs structures ont été simplifiées. Nous pensons que la mise au point d'une sémantique statique formelle couvrant ces caractéristiques soulève plusieurs difficultés que nous essayons d'exposer ci-dessous, bien que non exhaustivement.

Modificateurs d'accès

Les modificateurs d'accès sont conçus pour renforcer la sécurité qu'offre le langage. C'est pour cette raison que leur traitement est assez complexe. Ci-dessous nous présentons quelques restrictions dictées par la spécification de Java qu'exige le traitement des modificateurs :

- Les champs finaux doivent être initialisés lors de leur déclaration. Ceci implique que l'ordre des déclarations est important et que nous devons le considérer dans notre formalisation.
- Les classes abstraites ne doivent pas être instanciées. Ceci crée des exceptions qui font que la formalisation est plus complexe et moins lisible.
- Les méthodes sur-définies ou masquées ne doivent pas être plus privées que les méthodes qui les masquent ou les sur-définissent. Ceci crée également des exceptions.
- Chaque classe qui n'est pas abstraite doit implanter toutes les méthodes abstraites héritées des super-classes et des super-interfaces. Ce qui complique le traitement de cette restriction est que les classes abstraites peuvent fournir des implantations à certaines de leurs méthodes.
- Les méthodes d'instance ne peuvent pas sur-définir les méthodes de classe. De même que les méthodes de classe ne peuvent pas sur-définir les méthodes d'instance.
- Les membres d'instance ne doivent pas être invoqués à partir d'une méthode ou d'un champ de classe. En effet, les membres d'instance sont propres à un objet et leur valeur change d'un objet à un autre, alors que les membres de classe sont propres à une classe. Cette restriction requiert une connaissance de la nature des déclarations à typer, c'est-à-dire s'il s'agit de déclarations de membres d'instance ou de classe. En outre, il faut introduire des fonctions utilitaires dans la formalisation (voir le chapitre suivant).

Constructeurs

Lorsqu'une classe ne déclare pas explicitement de constructeur, le constructeur de la super-classe sans argument, s'il existe, sera son constructeur par défaut. Si par contre la classe déclare un constructeur, mais que celui-ci ne commence pas par une invocation explicite d'un autre constructeur, alors le constructeur de la super-classe sans argument sera invoqué automatiquement pour l'initialisation de la classe. Toutefois, si la classe supérieure ne possède pas un tel constructeur, alors la compilation du programme échoue. En plus, même si le constructeur est trouvé, il existe des restrictions quant aux exceptions qu'il déclare (pour plus de détails voir le chapitre 4). D'un autre côté, lors de l'invocation explicite d'un constructeur, il n'est pas permis d'invoquer un membre

d'instance dans le paramètre de **this** ou dans celui de **super** ni d'invoquer **this** ou **super**.

Initialisation et portée des variables locales

Les variables locales n'ont pas été formalisées dans [8, 29] et elles le sont partiellement dans [18]. En effet, dans celui-ci, aucune information n'est fournie afin de vérifier si une variable a été déclarée ou non dans un programme. Nous savons juste que la table des variables locales contient toujours les variables qui sont déjà déclarées. Cependant, aucune information ou traitement n'est présenté afin de montrer comment ces variables sont gérées et comment elles sont ajoutées à la table. Dans le langage Java, il n'est pas permis à la compilation qu'une variable locale soit utilisée avant qu'elle ne soit initialisée. Toutefois, dans [18] aucun traitement n'a été prévu pour gérer cette restriction.

Étant donné que les blocs ne font pas partie de la syntaxe des instructions de BALI, les variables locales déclarées dans le corps d'une méthode sont toutes utilisables dans celui-ci. Cependant, l'existence des blocs en Java permet de limiter la portée des variables locales au bloc dans lequel elles sont déclarées et nécessitent l'introduction de la notion de portée que nous présentons dans le chapitre 4.

Déclaration des exceptions

Quelques exceptions ont été considérées dans [18], mais leur traitement est incomplet, puisque la déclaration des exceptions par la clause **throws** n'est pas modélisée. Le traitement des clauses **throws** n'est pas simple et nécessite plusieurs vérifications par le système de types. Entre autres, lorsqu'une classe hérite de deux méthodes ayant la même signature, il faut faire certaines vérifications sur les exceptions qu'elles déclarent dans leur clause **throws** afin de nous assurer que ces exceptions sont compatibles (voir chapitre 4). Ces vérifications doivent aussi se faire lorsqu'une méthode implante une méthode abstraite.

Champs des interfaces

Les champs des interfaces n'ont pas été considérés dans aucune des formalisations de Java. La raison est que ces variables sont explicitement statiques et finales et nécessitent l'ajout de certaines conditions dans les règles de typage. En plus, l'héritage multiple de ces champs par l'intermédiaire des interfaces impose une vérification supplémentaire lors de l'invocation d'un champ qui est hérité de deux différentes interfaces, puisque l'accès ne doit pas être ambigu.

Ci-dessus, nous n'avons présenté que quelques-unes des difficultés qu'engendre l'introduction des caractéristiques omises dans la littérature. Cependant, les règles de typage seront un bon indicateur sur la complexité de la formalisation. Nous pensons qu'il est difficile de modifier le langage BALI afin qu'il respecte la spécfication de Java telle que décrite dans [11]. En effet, l'introduction des aspects omis nécessite non seulement l'ajout de quelques règles, mais également le changement de la plupart des règles

existantes et l'introduction de nouvelles notions. Cela nécessite une révision majeure de toute la sémantique.

3.3.2 Conformité de l'état de l'art avec la spécification de Java

Dans cette section, nous rapportons quelques erreurs commises dans le système de types de BALI qui a été présenté dans le chapitre précédent et nous discutons sa conformité avec la spécification de Java [11].

3.3.3 Exceptions

Malgré que les exceptions sont dites traitées dans [18], il reste qu'hormis quelques exceptions présentées, leur traitement est omis. En effet, parmi les exceptions qui ont été considérées nous citons celles de la classe **`RuntimeException`** et ses dérivées. Ces exceptions ne nécessitent pas un traitement particulier (voir chapitre 4) et l'utilisateur n'est pas obligé de les traiter ou même de les déclarer. Le traitement des exceptions restantes est assez compliqué comme nous le montrerons lors de la présentation de la sémantique statique de Java dans le chapitre suivant.

Par ailleurs, outre les exceptions de la classe **`RuntimeException`**, l'exception `Throwable` fait partie de l'ensemble des exceptions de BALI. Lorsque, dans le corps d'une méthode, une exception instance de cette classe est levée, elle doit être capturée par une clause **`catch`** ou bien déclarées dans la clause **`throws`** de la méthode. Cependant, aucun traitement n'a été prévu pour cette exception. Cette même erreur a été commise également dans [8] avec la classe **`Exception`**.

Mot-clé `super`

Dans [18], l'expression **`this`** est modélisée comme une variable locale qui ne peut pas être affectée. Le mot clé **`super`** n'a pas été traité. Néanmoins, il est mentionné que celui-ci peut être simulé par une expression **`this`** convertie explicitement, par une coercition, en le type de la classe supérieure de la classe courante. Cependant, ceci n'est pas possible puisque cette transformation peut rendre un membre, jusqu'alors accessible, inaccessible [12]. En outre, l'obtention de la classe supérieure par une coercition de **`this`** ne permet pas d'accéder aux méthodes sur-définies. Effectivement, si une méthode `m` d'une classe `C` est sur-définie dans la sous-classe `D` de `C`, alors l'expression qui apparaît dans le corps de la classe `D` suivante :

```
( (C)this ).m
```

permet d'invoquer la méthode de `D` et pas celle de `C`, puisque les méthodes sur-définies ne sont accessibles que par l'intermédiaire du mot-clé **`super`**. De ce fait, le traitement de `super` n'est pas aussi trivial qu'une simple coercition et nécessite un traitement particulier.

Type du littéral `null`

Le type de références qui représente la valeur du littéral **`null`** est modélisé dans [18] : toutefois son traitement a engendré un conflit avec la spécification. En effet, à

partir des règles de typage que nous avons présentées dans le chapitre précédent, nous pouvons inférer la création d'un tableau dont les éléments sont de type NullT, alors que celui-ci n'est pas un type en Java. Nous pensons que ceci découle de la non-distinction entre les types syntaxiques et les types sémantiques du langage. Cette distinction nous apparaît une démarche naturelle que nous adoptons dans notre formalisation.

En rapport avec le type NullT, l'expression d'affectation (que nous appelons par e) suivante :

```
Test T = null
```

est de type NullT selon la généralisation adoptée dans [18] et qui stipule que le type d'une affectation est déterminé par le coté droit et non par le coté gauche de l'expression d'affectation. Si nous supposons que la classe **Test** déclare une méthode **m** sans arguments, alors il est permis, statiquement, d'appeler cette méthode comme suit :

```
e.m()
```

Cependant, avec la généralisation adoptée dans [18], cette invocation de méthode sera rejetée, puisque le type de **e** n'est pas la classe **Test**, mais NullT.

Malgré que cet exemple montre la non conformité du système de types de BALI avec la spécification de Java, nous pensons que le traitement adopté est intéressant, puisqu'il permet de détecter une erreur assez tôt au niveau du typage.

3.3.4 Création d'un tableau

Avant d'expliquer une erreur que nous avons trouvée dans la règle de typage de création d'un tableau, nous donnons ci-dessous quelques notions que nous avons extraites de [18] :

$$\begin{array}{l}\mathsf{is\_type}\ \Gamma\ (\mathsf{PrimT}\ \_) \;=\; \mathsf{True}\\ \mathsf{is\_type}\ \Gamma\ (T[]) \;=\; \mathsf{is\_type}\ \Gamma\ T\end{array}$$

$$\begin{array}{lcl} prim\_ty & = & \texttt{void} \\ & | & \texttt{boolean} \\ & | & \texttt{int} \end{array}$$

$$\begin{array}{lcl} ty & = & \mathsf{PrimT}\ prim\_ty \\ & | & \mathsf{RefT}\ ref\_ty \end{array}$$

En se servant des informations ci-dessus et à partir de la règle de typage de la création d'un tableau ci-dessous :

$$\frac{\mathsf{is\_type}\ (\mathsf{prg}\ E)\ T \quad E \vdash i :: \mathsf{PrimT}\ \texttt{int}}{E \vdash \texttt{new}\ T[i] :: T[][]}$$

nous pouvons inférer :

$$\frac{\mathsf{is\_type}\ (\mathsf{prg}\ E)\ int[] \quad E \vdash i :: \mathsf{PrimT}\ \texttt{int}}{E \vdash \texttt{new}\ int[][i] :: int[]}$$

ce qui n'a pas de sens et est illégal en Java.

Membres d'un tableau

Dans [18] et même dans [8], les tableaux ne possèdent pas de membres. Pourtant les tableaux héritent de tous les membres de la classe `Object` et déclarent un champ appelé `length` ainsi qu'une méthode appelée `clone`.

Expressions parenthésées

Dans [18], les expressions parenthésées ne sont pas considérées. Si nous supposons que `C` et `D` sont deux classes et `f` une méthode, alors l'expression suivante :

```
( (D)C ).f( )
```

ne peut être représentée dans la syntaxe de BALI que par l'expression :

```
(D)C.f( )
```

ce qui est ambigu et non permis en Java, puisque nous ne sommes pas en mesure de deviner si la coercition est réalisée après ou avant l'invocation de la méthode.

3.3.5 Instruction d'expression

D'après la syntaxe des expressions présentée dans [18], toute expression est une instruction d'expression. Ceci n'est pas conforme avec la spécification. Cette dernière ne permet pas à certaines expressions d'être considérées comme des instructions.

Nous avons essayé dans cette section de montrer quelques difficultés qu'engendre la formalisation des aspects du langage Java omis dans la littérature et de donner quelques erreurs découvertes dans cette dernière. Nous tenons également à signaler que toutes les formalisations de Java comportent plusieurs simplifications, comme le fait d'exiger qu'il existe une seule instruction **return** dans un programme et qu'elle soit la dernière instruction dans celui-ci.

3.4 Conclusion

Dans ce chapitre, nous avons montré quelques subtilités dans la sémantique du langage Java ainsi que l'ambiguïté de sa spécification. Nous avons également présenté quelques erreurs que nous avons découvertes lors de notre étude de l'état de l'art en matière de sémantique formelle de Java. Nous avons abordé les difficultés que soulève la formalisation d'un plus grand ensemble de Java que celui traité dans la littérature. Le chapitre suivant est dédié à la présentation d'une sémantique statique formelle pour Java et il sera un bon indicateur sur la subtilité du langage et la difficulté de sa formalisation.

Chapitre 4

Sémantique statique pour Java

Dans ce chapitre, nous proposons une sémantique statique formelle qui couvre un ensemble de Java jusque là non formalisé. Nous n'avons négligé aucun détail et nous nous sommes penchés sur toutes les difficultés rencontrées. Nous avons traité les caractéristiques les plus subtiles de Java, comme les constructeurs, l'initialisation et les modificateurs d'accès. Par ce travail, nous visons à résoudre tous les problèmes rattachés à la sémantique statique formelle de Java et à fournir une description formelle de la spécification officielle du langage. Cependant, nous présentons la sémantique statique et nous l'expliquons sans toutefois émettre de jugements sur la spécification de Java, même si certains comportements du compilateur, exprimés dans les règles de typage, sont parfois inattendus. Cela évitera la redondance, puisque nous avons discuté de la spécification de Java dans le chapitre précédent.

Dans ce qui suit, nous présentons la syntaxe abstraite du langage considéré, ensuite nous introduisons la sémantique statique du langage incluant l'algèbre de types, l'environnement statique, les relations de types, la visibilité des déclarations, les conditions de bonne formation de programmes ainsi que les règles de typage des déclarations, des instructions et des expressions.

4.1 Syntaxe abstraite

Dans cette section, nous présentons la syntaxe du langage Java formalisé. Nous avons corrigé les omissions et les erreurs commises dans la spécification de Java [11] dont la description est disponible sur Internet, mais qui n'ont pas encore été intégrées à la spécification. Nous avons également proposé des modifications afin de corriger quelques erreurs qui n'ont pas été reportées. La syntaxe du langage Java formalisé couvre tous les aspects de celui-ci hormis les éléments ci-dessous :

- **Les paquetages :** Ce sont des librairies de classes et d'interfaces représentant des unités de compilation. Nous avons choisi de différer la formalisation des paquetages puisqu'ils présentent plusieurs difficultés et quelques inconsistances. En effet, nous avons constaté l'existence de quelques failles dans le fonctionnement de ces derniers. Nous les inclurons dès que nous comprendrons leur comportement exact.

abstract	**boolean**	**byte**	**class**	**double**	**else**	**extends**
final	**finally**	**float**	**if**	**implements**	**instanceof**	**int**
interface	**long**	**native**	**new**	**private**	**public**	**return**
short	**static**	**super**	**synchronized**	**this**	**throw**	**throws**
transient	**try**	**void**	**volatile**	**while**	**( )**	**[ ]**
{ }	**;**	**,**	**.**	**=**		

TAB. 4.1: Mots clés

- **Les constructions du langage introduites récemment :** Quelques changements ont été apportés récemment au langage Java, comme la possibilité d'imbriquer des classes (*inner classes*). Nous n'avons pas considéré ces modifications, puisque leur spécification n'est pas encore fournie d'une façon officielle. Cependant, une future introduction de ces modifications ne changera pas considérablement notre formalisation. En effet, la majeure partie de ces changements consiste en l'ajout de quelques nouveaux paquetages et de plusieurs classes aux paquetages existants. Elle ne touche qu'une mineure partie des caractéristiques du langage Java.
- **Les blocs statiques :** Les blocs statiques sont exécutés lorsque la classe qui les contient est initialisée. Ils ont le même rôle que les expressions d'initialisation de variables. Cependant, l'initialisation des variables de classe est regroupée dans un même bloc précédé du modificateur **static**. Nous ne traitons pas ce genre de blocs, puisque nous pouvons les représenter par une série d'initialisation de variables de classe.

De plus, et afin d'alléger la présentation et augmenter la lisibilité des règles de typage, nous avons éliminé les instructions pouvant être codées en termes d'éléments du langage considéré. Notamment, les différentes variantes de boucles, comme **loop** et **for**, ainsi que les instructions **break**, **continue** et **case** peuvent être remplacées par des combinaisons des instructions conditionnelles et d'itération. De même, les opérateurs unaires et binaires standards ne font pas partie de la syntaxe, puisque leur formalisation est simple et que leur intégration n'augmente pas l'expressivité de la formalisation et diminue la compacité et la lecture des règles de typage. En plus, le langage Java offre deux constructions syntaxiques pour les déclarations des tableaux : les crochets «[]» peuvent faire partie du type du tableau ou bien de son nom. Nous n'avons gardé que la première. De même, nous n'avons considéré que les tableaux unidimensionnels ; une éventuelle extension vers les tableaux multidimensionnels est facile.

4.1.1 Mots-clés

L'ensemble des mots-clés est composé des mots réservés du langage Java considéré et est présenté dans le tableau 4.1.

4.1.2 Grammaire du langage Java

Dans cette section nous donnons les règles grammaticales du langage Java formalisé. Un non-terminal peut se réduire à une séquence de terminaux et de non-terminaux ou bien à la variable ε qui représente le mot vide. Les non-terminaux apparaissent en italique, tandis que les terminaux représentant l'ensemble des mots-clés de Java sont en télétype. La syntaxe BNF du langage Java est présentée dans les tableaux 4.2, 4.3, 4.4, 4.5, 4.6 et . Nous avons choisi d'utiliser, quand c'est possible, les même noms de non-terminaux que dans [11]. Ceci permet au lecteur de se référer plus facilement à la spécification du langage Java. Toutefois, nous avons été menés à changer quelques règles grammaticales pour les besoins de la formalisation et pour lever quelques ambiguïtés dans celles-ci. Par exemple, dans la grammaire du langage Java, *FieldAccess* n'inclut pas l'accès à un champ en y référant par son nom simple qui est un identificateur, ou par son nom composé qui est une série d'identificateurs séparés par un point. Cet accès est donné par le non terminal *Name*. Celui-ci inclut l'accès à une variable locale. En plus, l'accès à un champ de type tableau n'est pas couvert par *FieldAccess*. Nous trouvons que cela porte à confusion. Nous modifions donc la grammaire de Java afin d'inclure dans l'accès à un champ toutes les expressions possibles d'accès à celui-ci. En plus, nous n'utilisons pas l'expression d'accès à une variable locale comme une sous-expression dans l'expression d'affectation de celle-ci, puisque ceci ne nous permet pas de vérifier si une variable locale est initialisée avant qu'elle ne soit utilisée.

Une méthode ou un constructeur doit inclure, dans sa déclaration, l'ensemble des exceptions *contrôlables* dont le soulèvement est à sa portée et qui ne sont pas traitées dans ses blocs **catch**. Nous entendons par exceptions à sa portée, les exceptions qui sont levées dans la méthode ou le constructeur, que ce soit par une instruction **throw** ou par une invocation d'une méthode ou d'un constructeur. Cette déclaration se fait par l'emploi du mot-clé **throws** après le nom de la méthode et la déclaration de ses paramètres. Dans le langage Java, il existe deux sortes d'exceptions : les exceptions *non contrôlables* et les exceptions *contrôlables*. Les premières sont susceptibles d'être levées un peu n'importe où dans le code. Ceci rend leur traitement difficile, voire impossible, et leur déclaration engendre la lourdeur du code. C'est pour cette raison qu'il n'est pas nécessaire de traiter ou de déclarer ces exceptions lorsqu'elles sont levées dans un programme Java. Il y a deux sortes d'exceptions non contrôlables :

- L'exception **RuntimeException** ainsi que toutes ses sous-classes. Elles sont levées par le noyau exécutable.
- L'exception **Error** et toutes ses sous-classes. Elles sont réservées aux erreurs matérielles.

Les deuxièmes, c'est-à-dire les exceptions contrôlables, doivent être traitées dans des blocs **catch** ou bien déclarées dans la clause **throws** de la méthode ou du constructeur. Ces exceptions sont toutes les exceptions autres que **RuntimeException**, **Error** et toutes leurs sous-classes. Afin de savoir si une exception est contrôlable, nous utilisons une fonction que nous appelons **checkedException**. Cette dernière sera introduite plus loin dans ce chapitre, puisque sa définition requiert quelques notions que nous n'avons pas encore présentées.

Comme dans le chapitre 2, nous considérons que chaque méthode possède exactement un paramètre. Nous appelons signature d'une méthode ou d'un constructeur, la paire composée de son nom et du type de son paramètre.

Nous supposons que les déclarations des classes, des interfaces, des champs, des méthodes et des constructeurs contiennent tous les modificateurs par défaut. À titre d'exemple, nous supposons que toutes les déclarations des méthodes des interfaces contiennent les modificateurs `public` et `abstract`.

Le reste de ce chapitre est dédié à la présentation de la sémantique statique du langage Java. Tout au long de celui-ci, nous utilisons les conventions typographiques suivantes : les mots réservés de Java sont en télétype, le métalangage est décrit en police de caractère sans sérif, tandis que les non-terminaux et les méta-variables sont en italiques.

Dans le métalangage, comme dans le chapitre 2, nous utilisons deux projections sur les tuples, soient **fst** et **snd**.

4.2 Algèbre de Types

Il existe deux types en Java : les types primitifs et les types de références. Un type de références est un type d'une classe, d'une interface ou d'un tableau. Nous utilisons, dans nos règles de typage, les types syntaxiques du langage Java présentés dans le tableau 4.7. Chaque non-terminal représente une catégorie de type syntaxique. Pour les besoins de la formalisation, nous introduisons également d'autres catégories de types sémantiques que nous présentons dans le tableau 4.8. Nous avons besoin de ces nouveaux types pour les deux raisons suivantes :

- Le littéral `null` est du type particulier *null*. Celui-ci ne possède pas de nom en Java. Il n'est pas possible de déclarer une variable de ce type ou même de faire une coercition vers celui-ci. La référence nulle est la seule valeur possible que peut prendre une expression de ce type. Afin de gérer ce type inexistant dans Java et pouvoir typer le littéral `null`, nous introduisons le type sémantique **Null**.
- Afin d'assurer la compacité des règles de typage, nous introduisons un type spécial appelé **Unit**. Celui-ci est le type du mot vide ε. Ce type permet d'éviter de dupliquer les règles de typage lors de l'évaluation statique des déclarations et des appels de méthodes et de constructeurs n'ayant pas de paramètre.

Le type *ExpressionType* représente le type d'une expression. Il peut être **Null** ou de n'importe quel type syntaxique incluant `void`. En effet, le type d'une invocation de méthode peut être inexistant si celle-ci ne déclare pas de type de retour. La catégorie sémantique représentant le type d'un paramètre est nommée *ParameterType* et peut être ou bien un type syntaxique qui n'est pas `void` ou bien **Unit**. Le type *ArgumentType* représente le type d'un argument qui, contrairement à un paramètre, peut être une expression. Les arguments sont utilisés dans les expressions d'appel de méthodes et de constructeurs. Enfin, pour augmenter la précision sur les types dans nos règles de typage, nous introduisons une autre catégorie de types *NullOrSimpleType* qui peut être un type primitif, le type d'une classe ou d'une interface ou bien **Null**.

Program	::=	*ClassDeclaration Program*
	|	*InterfaceDeclaration Program*
	|	ϵ
ClassDeclaration	: :=	*Modifiers* **class Identifier** *Extends Implements*
	|	**{***ClassBody***}**
Modifiers	: :=	**public** *Modifiers*
	|	**private** *Modifiers*
	|	**static** *Modifiers*
	|	**abstract** *Modifiers*
	|	**final** *Modifiers*
	|	**synchronized** *Modifiers*
	|	**native** *Modifiers*
	|	**transient** *Modifiers*
	|	**volatile** *Modifiers*
	|	ϵ
Extends	: :=	**extends** *ClassType*
	|	ϵ
Implements	: :=	**implements** *InterfaceTypeList*
	|	ϵ
InterfaceTypeList	: :=	*InterfaceType*
	|	*InterfaceType* , *InterfaceTypeList*
ClassBody	: :=	*FieldDeclaration ClassBody*
	|	*MethodDeclaration ClassBody*
	|	*AbstractMethodDeclaration ClassBody*
	|	*ConstructorDeclaration ClassBody*
	|	ϵ
FieldDeclaration	: :=	*Modifiers Type* **Identifier**
	|	*Modifiers Type* **identifier** = *Expression*
	|	*Modifiers SimpleType*[] **identifier** = *ArrayInitializer*
ArrayInitializer	: :=	{ }
	|	{ *ExpressionInitializer* }
ExpressionInitializer	: :=	*Expression*
	|	*Expression* , *ExpressionInitializer*
MethodDeclaration	: :=	*Modifiers ResultType* **Identifier** (*Parmeter*) *Throws Block*

TAB. 4.2: Syntaxe abstraite de Java : partie 1

Parameter	: :=	***Type Identifier***
	|	ϵ
Throws	: :=	**throws** *ClassTypeList*
	|	ϵ
ClassTypeList	: :=	*ClassType*
	|	*ClassType* , *ClassTypeList*
ConstructorDeclaration	: :=	*Modifiers* **Identifier** (*Parmeter*) *Throws* *ConstructorBody*
ConstructorBody	: :=	{ *ExplicitConsInvocation* *BlockStatementsOrEmpty* }
ExplicitConsInvocation	: :=	**this** (*Argument*) ;
	|	**super** (*Argument*) ;
	|	ϵ
Argument	: :=	*Expression*
	|	ϵ
InterfaceDeclaration	: :=	*Modifiers* **interface** *Identifier* *ExtendsInterfaces* { *InterfaceBody* }
ExtendsInterfaces	: :=	**extends** *InterfaceTypeList*
	|	ϵ
InterfaceBody	: :=	*FieldDeclaration*
	|	*AbstractMethodDeclaration*
AbstractMethodDeclaration	: :=	*Modifiers ResultType Identifier* (*Parmeter*) *Throws* ;
Block	: :=	{ *BlockStatementsOrEmpty* }
BlockStatementsOrEmpty	: :=	*BlockStatements*
	|	ϵ
BlockStatements	: :=	*BlockStatement BlockStatements*
	|	*BlockStatement*
BlockStatement	: :=	*LocalVariableDeclaration*
	|	*Statement*

TAB. 4.3: Syntaxe abstraite de Java : partie 2

LocalVariableDeclaration	: :=	*Type* **Identifier** ;
	|	*Type Identifier* = *Expression*
	|	*SimpleType*[] *Identifier* = *ArrayInitializer* ;
Statement	: :=	;
	|	*Block*
	|	*ExpressionStatement*
	|	*IfStatement*
	|	*WhileStatement*
	|	*ThrowStatement*
	|	*SynchronizedStatement*
	|	*TryStatement*
	|	*ReturnStatement*
ExpressionStatement	: :=	*StatementExpression* ;
StatementExpression	: :=	*AssignmentExpression*
	|	*MethodInvocation*
	|	*ClassInstanceCreation*
IfStatement	: :=	**if** (*Expression*) *Statement* **else** *Statement*
	|	**if** (*Expression*) *Statement*
WhileStatement	: :=	**while** (*Expression*) *Statement*
ThrowStatement	: :=	**throw** *Expression* ;
SynchronizedStatement	: :=	**synchronized** (*Expression*) *Block*
TryStatement	: :=	**try** *Block Catches* **finally** *Block*
	|	**try** *Block Catches*
	|	**try** *Block* **finally** *Block*
ReturnStatement	: :=	**return** *Expression* ;
	|	**return** ;
Catches	: :=	*Catch*
	|	*Catch Catches*
Catch	: :=	**catch** (*ClassType Identifier*) *Block*
Primary	: :=	*ArrayCreation*
	|	*PrimaryNoNewArray*
ArrayCreation	: :=	**new** *SimpleType* [*Expression*]

TAB. 4.4: Syntaxe abstraite de Java : partie 3

PrimaryNoNewArray	: :=	**Literal**
	|	**this**
	|	**(** *Expression* **)**
	|	*ClassInstanceCreation*
	|	*SimpleFieldAccess*
	|	*ArrayFieldAccess*
	|	*MethodInvoction*
ClassInstanceCreation	: :=	**new** *ClassType* **(** *Argument* **)**
SimpleFieldAccess	: :=	*Primary* **.** **Identifier**
	|	**super** **.** *Identifier*
	|	*FieldName*
FieldName	: :=	*Identifier*
	|	*ClassOrInterfaceType* **.** *Identifier*
	|	*ExpressionName* **.** *Identifier*
ExpressionName	: :=	*FieldName*
	|	*SimpleLocalVarAccess*
SimpleLocalVarAccess	: :=	*Identifier*
ArrayFieldAccess	: :=	*PrimaryNoNewArray* **[** *Expression* **]**
MethodInvocation	: :=	*MethodName* **(** *Argument* **)**
	|	*Primary* **.** *Identifier* **(** *Argument* **)**
	|	**super** **.** **Identifier** **(** *Argument* **)**
MethodName	: :=	*Identifier*
	|	*ClassType* **.** *Identifier*
	|	*ExpressionName* **.** *Identifier*
Expression	: :=	*AssignmentExpression*
	|	*CastExpression*
	|	*Primary*
	|	*SimpleLocalVarAccess*
	|	*ArrayLocalVarAccess*
AssignmentExpression	: :=	*SimpleFieldAccess* **=** *Expression*
	|	*ArrayFieldAccess* **=** *Expression*
	|	*Identifier* **=** *Expression*
	|	*Identifier***[** *Expression* **]** **=** *Expression*

TAB. 4.5: Syntaxe abstraite de Java : partie 4

CastExpression	: :=	(*Type*) *Expression*
ArrayLocalVarAccess	: :=	*Identifier* [*Expression*]

TAB. 4.6: Syntaxe abstraite de Java : partie 5

$\tau_r \in$ *ResultType*	: :=	*Type*
	|	`void`
$\tau \in$ *Type*	: :=	*PrimitiveType*
	|	*ReferenceType*
$\rho \in$ *ReferenceType*	: :=	*ClassOrInterfaceType*
	|	*ArrayType*
$\pi \in$ *PrimitiveType*	: :=	`boolean`
	|	`byte`
	|	`short`
	|	`int`
	|	`long`
	|	`char`
	|	`float`
	|	`double`
$\mu \in$ *ClassOrInterfaceType*	: :=	*ClassType*
	|	*InterfaceType*
$\alpha \in$ *ArrayType*	: :=	*SimpleType* []
$\sigma* \in$ *SimpleType*	: :=	*PrimitiveType*
	|	*ClassOrInterfaceType*
$\varsigma \in$ *ClassType*	: :=	*Identifier*
$\iota \in$ *InterfaceType*	: :=	*Identifier*

TAB. 4.7: Types syntaxiques de Java

$\tau_a \in$ *ArgumentType*	: :=	*ParameterType*
	|	Null
$\tau_e \in$ *ExpressionType*	: :=	*ResultType*
	|	Null
$\tau_p \in$ *ParameterType*	: :=	*Type*
	|	Unit
$\sigma \in$ *NullOrSimpleType*	: :=	*SimpleType*
	|	Null

TAB. 4.8: Types sémantiques

4.3 Quelques notations

Étant donnés deux ensembles A et B, nous désignons par $A \overrightarrow{m} B$ les mappes de A vers B (fonctions partielles de A vers B avec un domaine fini), par **dom**(m) le domaine de la mappe m et par **rang**(m) son co-domaine. La mappe vide est notée []. Une mappe $m \in A \overrightarrow{m} B$ peut être définie en énumérant explicitement ses associations par $[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$. Nous notons $m_1 \dagger m_2$, avec m_1 et m_2 des mappes de A vers B, pour désigner l'augmentation de la mappe m_1 par les associations de la mappe m_2 comme suit : $(m_1 \dagger m_2)(a) = m_1(a)$ si $a \in$ **dom**(m_1) et $m_2(a)$ sinon. De même, nous écrivons $m_1 \backslash m_2$ pour désigner la diminution de la mappe m_1 par les associations de la mappe m_2 comme suit : $(m_1 \backslash m_2)(a) = m_1(a)$ si $a \in$ **dom**(m_1) et $a \notin$ **dom**(m_2). Enfin, l'expression m_1/m_2 désigne la mappe m_1 réduite aux associations de la mappe m_2, tel que : $(m_1/m_2)(a) = m_1(a)$ si $a \in m_1$ et $a \in m_2$.

Nous notons par (*Type*)**set** l'ensemble dont les valeurs sont de type *Type* et par (*Type*)**multi-set** le multi-ensemble dont les valeurs sont de type *Type*. Les multi-ensembles possèdent la même forme que les ensembles simples, à l'exception des délimiteurs qui sont «{|» et «|}» au lieu de «{» et «}».

Enfin, nous écrivons ssi pour abréger l'expression si et seulement si.

4.4 Environnement statique

Le langage Java est très puissant. Il offre plusieurs facilités de programmation à ses utilisateurs. Toutefois, cette souplesse engendre l'augmentation de la complexité de la formalisation. Parmi les facteurs de complications nous citons ce qui suit :

- Il est possible en Java d'utiliser les classes et les interfaces avant que leur définition ne soit fournie.
- Java effectue certaines vérifications sur les types et exige que les programmes satisfassent certaines propriétés de bonne formation. Nous citons la non-circularité de la hiérarchie des classes et des interfaces, les classes non abstraites doivent

implanter toutes les méthodes abstraites héritées, les variables locales ne doivent pas cacher les paramètres de la méthode ou du constructeur et la satisfaction de certaines conditions concernant la sur-définition et le masquage.

- Java gère le sous-typage et il offre la possibilité de passer un type pour un autre sous certaines conditions.

Afin de gérer ces difficultés, nous introduisons un environnement de types. Il contient toutes les informations concernant les déclarations des classes et des interfaces du programme incluant leurs membres avec toutes les informations de type qui s'y rapportent. Il est obtenu à partir du programme Java à typer et il est représenté par une combinaison d'enregistrements et de mappes. Nous avons choisi les enregistrements plutôt que les tuples afin de faciliter la lecture et des règles de typage et des fonctions de bonne formation des programmes. En effet, cela évitera d'utiliser des fonctions de projection pour obtenir le contenu des tuples. Grâce à l'environnement de types, nous serons capables de gérer plusieurs caractéristiques non triviales du langage et de nous assurer de la bonne formation des programmes en posant des conditions sur l'environnement.

La structure d'un environnement est donnée dans le tableau 4.9. Malgré que l'héritage multiple par les classes n'existe pas en Java, nous avons décidé de représenter la composante super-classe, notée *super* dans l'environnement, d'une classe par un ensemble. Ceci uniformise le traitement et facilite la gestion de l'héritage lorsque la superclasse est inexistante. Ceci est le cas de la classe mère `Object`. Toutefois, afin d'être conforme à la spécification de Java, l'ensemble représentant la classe supérieure doit contenir au maximum un élément. Les variables locales des méthodes et des constructeurs ne font pas partie de l'environnement. Elles seront introduites plus tard dans les règles de typage. Ainsi, nous pourrons gérer la portée des variables locales et leur initialisation. Nous supposons que toutes les déclarations sont nommées différemment, c'est-à-dire qu'il n'existe pas deux classes ou deux interfaces qui possèdent le même nom dans un même programme et qu'il n'existe pas deux champs ou deux méthodes qui possèdent le même nom ou la même signature dans une même classe ou interface. Ceci évitera d'alourdir la présentation, d'autant plus que nous avons montré comment ceci peut être géré et ce, dans le chapitre 2. Nous supposons également que toutes les méthodes et constructeurs qui ne possèdent pas de paramètre ont Unit comme type de paramètre dans l'environnement.

4.5 Objets sémantiques et abréviations

Ci-dessous nous donnons quelques-uns des objets sémantiques qui seront utilisés dans les règles de typage. Ils seront aussi utilisés dans les fonctions sémantiques que nous proposons. Ceci facilitera la lecture de notre formalisation :

Γ	$\in$	*Environment*	environnement de types
cs ou *tcs*	$\in$	*ClassType*(**set**)	ensemble de classes
is	$\in$	*InterfaceType*(**set**)	ensemble d'interfaces
ms	$\in$	*ModifierName*(**set**)	ensemble de modificateurs

Nous présentons dans le reste de cette section le glossaire des termes les plus souvent employés pour désigner certaines constructions syntaxiques.

Environment	=	⟨ *classMap* : *ClassMap*, *interfaceMap* : *InterfaceMap* ⟩
ClassMap	=	*ClassType* ⇀ *ClassDecl*
InterfaceMap	=	*InterfaceType* ⇀ *InterfaceDecl*
ClassDecl	=	⟨ *modifiers* : (*ModifierName*)**set**, *super* : (*ClassType*)**set**, *interfaces* : (*InterfaceType*)**set**, *fields* : *Identifier* ⇀ *FieldInfo*, *methods* : *Sig* ⇀ *MethodInfo*, *constructors* : *Sig* ⇀ *ConstructorInfo* ⟩
InterfaceDecl	=	⟨ *modifiers* : (*ModifierName*)**set**, *interfaces* : (*InterfaceType*)**set**, *fields* : *Identifier* ⇀ *FieldInfo*, *methods* : *Sig* ⇀ *MethodInfo* ⟩
Sig	=	*Identifier* × *ParameterType*
ModifierName	=	{**public**,**static**,**abstract**,**final**,**private**, **synchronized**,**native**,**transient**,**volatile**}
FieldInfo	=	⟨ *fieldType* : *Type*, *modifiers* : (*ModifierName*)**set** ⟩
MethodInfo	=	⟨ *resultType* : *ResultType*, *modifiers* : (*ModifierName*)**set**, *throws* : (*ClassType*)**set** ⟩
ConstructorInfo	=	⟨ *modifiers* : (*ModifierName*)**set**, *throws* : (*ClassType*)**set** ⟩

TAB. 4.9: Environnement statique

$$
\begin{array}{lcl}
\mathit{MethodMap} & = & \mathit{Sig} \twoheadrightarrow \mathit{MethodInfo} \mid \\
 & & \mathit{Sig} \twoheadrightarrow (\mathit{MethodInfo} \times \mathit{ReferenceType})\mathbf{set} \\
\mathit{ConstructorMap} & = & \mathit{Sig} \twoheadrightarrow \mathit{ConstructorInfo} \mid \\
 & & \mathit{Sig} \twoheadrightarrow (\mathit{ConstructorInfo} \times \mathit{ReferenceType}) \\
\mathit{FieldMap} & = & \mathit{Identifier} \twoheadrightarrow \mathit{FieldInfo} \mid \\
 & & \mathit{Identifier} \twoheadrightarrow (\mathit{FieldInfo} \times \mathit{ReferenceType})\mathbf{set}
\end{array}
$$

n	$\in$	*ClassOrIfaceType*	nom d'une classe ou d'une interface
mn	$\in$	*Identifier*	nom d'une méthode qui est un identificateur
pn	$\in$	*Identifier*	nom d'un paramère qui est un identificateur
sig	$\in$	*Sig*	signature d'une méthode
fn	$\in$	*Identifier*	nom d'un champ qui est un identificateur
mm	$\in$	*MethodMap*	mappe de méthodes
cm	$\in$	*ConstructorMap*	mappe de constructeurs
fm	$\in$	*FieldMap*	mappe de champs
clm	$\in$	*ClassMap*	mappe de classes
ifm	$\in$	*InterfaceMap*	mappe d'interfaces

4.6 Validité des types

Afin de nous assurer que les types utilisés dans un programme Java sont valides, sont reconnus par le compilateur, nous introduisons les prédicats présentés dans le tableau 4.10. Tous les types élémentaires sont des types valides. Le type d'une classe ou d'une interface est considéré comme un type valide ssi la classe ou l'interface est déclarée dans le programme. Le type d'un tableau est valide ssi le type de ses éléments l'est. Nous introduisons également le prédicat **validIfaces** qui permet de vérifier qu'un ensemble d'interfaces ne contient que des interfaces valides.

4.7 Relations de types

Comme dans le chapitre 2, nous introduisons les différentes relations existantes entre les types de Java. Nous distinguons les relations de sous-classe, de sous-interface, d'implantation et de conversion. Ces relations dépendent de la hiérarchie des classes et des interfaces du programme. Elles sont donc exprimées en fonction de celui-ci.

4.7.1 Relation de sous-classe

La relation de sous-classe $\Gamma \vdash \varsigma_1 \sqsubset_{class} \varsigma_2$ signifie que dans l'environnement Γ, la classe ς_1 est une sous-classe de ς_2. Elle est présentée dans le tableau 4.11.

4.7.2 Relation de sous-interface

La relation de sous-interface $\Gamma \vdash \iota_1 \sqsubset_{interface} \iota_2$ signifie que dans l'environnement Γ, l'interface ι_1 est une sous-interface de ι_2. Cette relation est différente de la précédente, puisque l'héritage multiple par les interfaces est permis en Java. Elle est présentée dans le tableau 4.12.

$$
\begin{array}{l}
\mathsf{validType} \;:\; \mathit{Environment} \times \mathit{Type} \rightarrow \mathsf{bool} \\
\\
\mathsf{validType}(\Gamma, \pi) \;=\; \mathsf{true} \\
\\
\mathsf{validType}(\Gamma, \varsigma) \;=\; \mathsf{validClass}(\Gamma, \varsigma) \\
\\
\mathsf{validType}(\Gamma, \iota) \;=\; \mathsf{validIface}(\Gamma, \iota) \\
\\
\mathsf{validType}(\Gamma, \sigma * []) \;=\; \mathsf{validType}(\Gamma, \sigma*) \\
\\
\mathsf{validIface}(\Gamma, \tau) \;=\; \Gamma.\mathit{interfaceMap}(\tau) \neq \bot \\
\\
\mathsf{validClass}(\Gamma, \tau) \;=\; \Gamma.\mathit{classMap}(\tau) \neq \bot \\
\\
\mathsf{validIfaces}(\Gamma, \emptyset) \;=\; \mathsf{true} \\
\mathsf{validIfaces}(\Gamma, \{\iota\} \cup \mathit{is}) \;=\; \mathsf{validIface}(\Gamma, \iota) \wedge \mathsf{validIfaces}(\Gamma, \mathit{is})
\end{array}
$$

TAB. 4.10: Validité des types

$$
\frac{\mathsf{validType}(\Gamma, \varsigma_1) \quad \Gamma.\mathit{classMap}(\varsigma_1).\mathit{super} = \varsigma_2}{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_2}
$$

$$
\frac{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_2 \quad \Gamma \vdash \varsigma_2 \sqsubset_{\mathit{class}} \varsigma_3}{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_3}
$$

TAB. 4.11: Relation de sous-classe

$$
\frac{\mathsf{validType}(\Gamma, \iota_1) \quad \iota_2 \in \Gamma.\mathit{interfaceMap}(\iota_1).\mathit{interfaces}}{\Gamma \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_2}
$$

$$
\frac{\Gamma \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_2 \quad \Gamma \vdash \iota_2 \sqsubset_{\mathit{interface}} \iota_3}{\Gamma \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_3}
$$

TAB. 4.12: Relation de sous-interface

$$\frac{\mathsf{validType}(\Gamma,\varsigma) \quad \iota \in \Gamma.\mathit{classMap}(\varsigma).\mathit{interfaces}}{\Gamma \vdash \varsigma \sqsubset_{\mathit{implements}} \iota}$$

$$\frac{\Gamma \vdash \varsigma \sqsubset_{\mathit{implements}} \iota_1 \quad \Gamma \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_2}{\Gamma \vdash \varsigma \sqsubset_{\mathit{implements}} \iota_2}$$

$$\frac{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_2 \quad \Gamma \vdash \varsigma_2 \sqsubset_{\mathit{implements}} \iota}{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{implements}} \iota}$$

TAB. 4.13: Relation d'implantation

$$\frac{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_2}{\Gamma \vdash \varsigma_1 \sqsubseteq_{\mathit{class}} \varsigma_2} \qquad \frac{\mathsf{validType}(\Gamma,\varsigma)}{\Gamma \vdash \varsigma \sqsubseteq_{\mathit{class}} \varsigma}$$

$$\frac{\Gamma \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_2}{\Gamma \vdash \iota_1 \sqsubseteq_{\mathit{interface}} \iota_2} \qquad \frac{\mathsf{validType}(\Gamma,\iota)}{\Gamma \vdash \iota \sqsubseteq_{\mathit{interface}} \iota}$$

TAB. 4.14: Relations réflexives de sous-classe et de sous-interface

4.7.3 Relation d'implantation

La relation d'implantation $\Gamma \vdash \varsigma \sqsubset_{\mathit{implements}} \iota$ signifie que dans l'environnement Γ, la classe ς fournit une implantation à l'interface ι. Elle est présentée dans le tableau 4.13.

Pour les besoins de la formalisation, nous introduisons dans le tableau 4.14 deux autres relations :

- la relation $\Gamma \vdash \varsigma_1 \sqsubseteq_{\mathit{class}} \varsigma_2$ qui signifie que dans l'environnement Γ, ς_1 est ou bien la même classe que ς_2 ou bien l'une de ses sous-classes ;
- la relation $\Gamma \vdash \iota_1 \sqsubseteq_{\mathit{interface}} \iota_2$ qui signifie que dans l'environnement Γ, l'interface ι_1 est ou bien la même interface que ι_2 ou bien l'une de ses sous-interfaces.

À partir de la relation $\sqsubseteq_{\mathit{class}}$, nous pouvons définir les deux prédicats suivants :

- un prédicat qui permet de vérifier sous un certain environnement, si une classe est une sous-classe d'au moins une classe d'un autre ensemble de classes :

$$\mathsf{subClass} : \mathit{Environment} \times \mathit{ClassType} \times (\mathit{ClassType})\mathsf{set} \rightarrow \mathsf{bool}$$

$$\begin{aligned} \mathsf{subClass}(\Gamma, \varsigma, \emptyset) &= \mathsf{false} \\ \mathsf{subClass}(\Gamma, \varsigma_1, \{\varsigma_2\} \cup cs) &= \Gamma \vdash \varsigma_1 \sqsubseteq_{\mathit{class}} \varsigma_2 \vee \mathsf{subClass}(\Gamma, \varsigma, cs) \end{aligned}$$

- un prédicat qui permet de vérifier que chaque classe d'un ensemble de classes admet une super-classe dans un autre ensemble de classes :

$$\mathsf{subClasses} : \mathit{Environement} \times (\mathit{ClassType})\mathsf{set} \times (\mathit{ClassType})\mathsf{set} \rightarrow \mathsf{bool}$$

$$\begin{aligned} \mathsf{subClasses}(\Gamma, \emptyset, cs) &= \mathsf{true} \\ \mathsf{subClasses}(\Gamma, \{\varsigma\} \cup cs_1, cs_2) &= \mathsf{subClass}(\Gamma, \varsigma, cs_2) \wedge \mathsf{subClasses}(\Gamma, cs_1, cs_2) \end{aligned}$$

$$\frac{\text{validType}(\Gamma, \pi)}{\Gamma \vdash \pi \sqsubseteq_{implP} \pi}$$

$$\frac{}{\Gamma \vdash \texttt{byte} \sqsubseteq_{implP} \texttt{short}} \quad \frac{}{\Gamma \vdash \texttt{byte} \sqsubseteq_{implP} \texttt{int}} \quad \frac{}{\Gamma \vdash \texttt{byte} \sqsubseteq_{implP} \texttt{long}}$$

$$\frac{}{\Gamma \vdash \texttt{byte} \sqsubseteq_{implP} \texttt{float}} \quad \frac{}{\Gamma \vdash \texttt{byte} \sqsubseteq_{implP} \texttt{double}} \quad \frac{}{\Gamma \vdash \texttt{short} \sqsubseteq_{implP} \texttt{int}}$$

$$\frac{}{\Gamma \vdash \texttt{short} \sqsubseteq_{implP} \texttt{long}} \quad \frac{}{\Gamma \vdash \texttt{short} \sqsubseteq_{implP} \texttt{float}} \quad \frac{}{\Gamma \vdash \texttt{short} \sqsubseteq_{implP} \texttt{double}}$$

$$\frac{}{\Gamma \vdash \texttt{int} \sqsubseteq_{implP} \texttt{long}} \quad \frac{}{\Gamma \vdash \texttt{int} \sqsubseteq_{implP} \texttt{float}} \quad \frac{}{\Gamma \vdash \texttt{int} \sqsubseteq_{implP} \texttt{double}}$$

$$\frac{}{\Gamma \vdash \texttt{long} \sqsubseteq_{implP} \texttt{float}} \quad \frac{}{\Gamma \vdash \texttt{long} \sqsubseteq_{implP} \texttt{double}} \quad \frac{}{\Gamma \vdash \texttt{float} \sqsubseteq_{implP} \texttt{double}}$$

TAB. 4.15: Relation de conversion implicite entre les types primitifs

4.7.4 Relation de conversion implicite entre les types primitifs

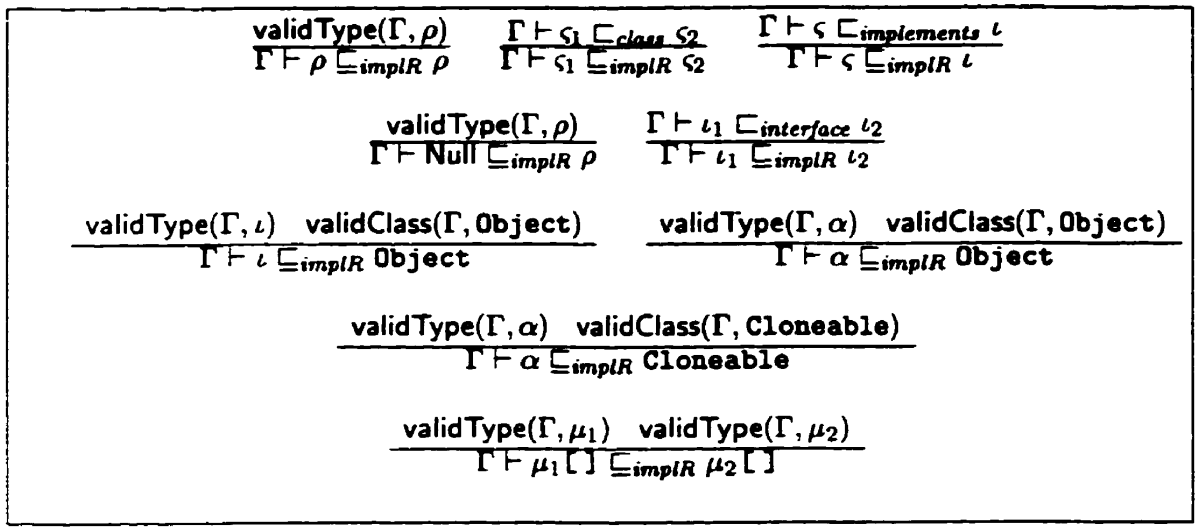
La conversion d'élargissement primitive (*Widening Primitive Conversion*), que nous appelons conversion implicite entre les types primitifs, ne fait perdre aucune précision au nombre converti. Elle ne requiert aucune conversion explicite par l'opérateur de coercition. Le compilateur effectue implicitement cette conversion lorsque c'est nécessaire. La relation $\Gamma \vdash \pi_1 \sqsubseteq_{implP} \pi_2$, signifie que dans l'environnement Γ, le type primitif π_1 peut remplacer le type primitif π_2 sans aucune conversion explicite. Cette relation est présentée dans le tableau 4.15.

4.7.5 Relation de conversion implicite entre les types de références

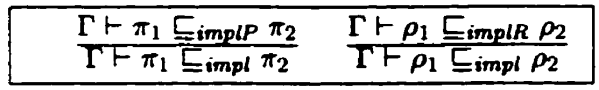
La relation $\Gamma \vdash \rho_1 \sqsubseteq_{implR} \rho_2$, signifie que dans l'environnement Γ, le type de références ρ_1 peut remplacer le type de références ρ_2 sans aucune conversion explicite. Cette relation est présentée dans le tableau 4.16.

En utilisant les deux relations implicites sus-dénommées, nous définissons une troisième relation $\Gamma \vdash \tau_1 \sqsubseteq_{impl} \tau_2$ qui signifie que dans l'environnement Γ, le type τ_1 peut remplacer le type τ_2 sans aucune conversion explicite. Nous appelons cette relation une relation de conversion implicite et nous la présentons dans le tableau 4.17.

Toutes les conversions présentées seront très utiles dans la formalisation, entre autres pour typer une expression d'affectation. En effet, lorsque la valeur d'une expression est affectée à une variable, le type de l'expression doit être converti en le type de la variable. Cette conversion est implicite. Nous la vérifions grâce à la relation $\sqsubseteq_{impl}$.



TAB. 4.16: Relation de conversion implicite entre les types de références



TAB. 4.17: Relation de conversion implicite

Cependant, dans le langage Java, la conversion utilisée est appelée conversion par affectation. Elle contient toutes les conversions permises par la relation de conversion implicite. En plus, elle permet à la valeur d'une expression de type **int** d'être affectée à une variable de type **byte**, **short** ou même **char** pourvu que la valeur de l'expression puisse être représentée dans le type de la variable. Nous avons choisi de garder uniquement la conversion implicite afin d'augmenter la lisibilité de la formalisation et pour éviter que le lecteur ne s'embrouille dans différentes relations. Nous tenons à remarquer que cette restriction n'affecte pas les règles de typage.

4.7.6 Relation de coercition

Le langage Java permet à certains types d'être convertis explicitement, par l'utilisateur, en d'autres types. Cette opération de conversion est appelée coercition (*cast*). Afin de gérer cette conversion, nous introduisons la relation $\Gamma \vdash \tau_1 \sqsubseteq_{cast} \tau_2$ qui signifie que dans l'environnement Γ, le type τ_1 peut être converti explicitement, par une coercition, en le type de τ_2. La relation de coercition est présentée dans le tableau 4.18. Si une classe n'est pas finale, il est permis à la compilation, de faire une coercition entre cette classe et une interface, puisque même si la classe n'implante pas l'interface, il est possible que l'une de ses sous-classes le fasse. Cependant, une classe finale ne peut pas avoir de sous-classe. La coercition n'est donc permise que dans le cas où la classe implante directement l'interface. D'un autre côté, une coercition entre deux types de référence abstraits (une interface ou une classe abstraite) est permise seulement si toutes les méthodes ayant la même signature déclarées dans les deux types possèdent

le même type de retour (voir chapitre 2). Le prédicat qui permet de vérifier que ces méthodes possèdent le même type de retour est défini comme suit :

$$\mathsf{sameResultType} \;:\; (\mathit{Sig} \overset{m}{\rightarrow} \mathit{MethodInfo}) \times (\mathit{Sig} \overset{m}{\rightarrow} \mathit{MethodInfo}) \rightarrow \mathsf{bool}$$

$$\begin{array}{lcl}
\mathsf{sameResultType}([], mm) & = & \mathsf{true} \\
\mathsf{sameResultType}([a_1 \mapsto b_1], [a_2 \mapsto b_2] \dagger mm) & = & \mathsf{if} \quad a_1 = a_2 \wedge \\
 & & \qquad \mathbf{abstract} \in b_1.\mathit{modifiers} \\
 & & \mathsf{then} \;\; b_1.\mathit{resultType} = b_2.\mathit{resultType} \\
 & & \mathsf{else} \;\; \mathsf{sameResultType}([a_1 \mapsto b_1], mm) \\
 & & \mathsf{end\ if}
\end{array}$$

$$\begin{array}{l}
\mathsf{sameResultType}([a_1 \mapsto b_1] \dagger mm_1, [a_2 \mapsto b_2] \dagger mm_2) \;\; = \\
\quad \mathsf{sameResultType}([a_1 \mapsto b_1], [a_2 \mapsto b_2] \dagger mm_2) \wedge \\
\quad \mathsf{sameResultType}(mm_1, [a_2 \mapsto b_2] \dagger mm_2)
\end{array}$$

Il est maintenant possible de donner la définition du prédicat **checkedException** qui a été introduit plus haut et qui permet de vérifier si une exception est contrôlable ou non :

$$\mathsf{checkedException} \;:\; \mathit{Environment} \times \mathit{ClassType} \rightarrow \mathsf{bool}$$

$$\begin{array}{lcl}
\mathsf{checkedException}(\Gamma, \varsigma) & = & \Gamma \vdash \varsigma \sqsubseteq_{class} \mathbf{Throwable} \wedge \neg(\, \Gamma \vdash \varsigma \sqsubseteq_{class} \mathbf{Error} \,) \wedge \\
 & & \neg(\, \Gamma \vdash \varsigma \sqsubseteq_{class} \mathbf{RuntimeException} \,)
\end{array}$$

En utilisant ce prédicat, nous pouvons définir un autre prédicat qui retourne un ensemble composé uniquement des exceptions contrôlables d'un autre ensemble d'exceptions :

$$\mathsf{getCheckedException} \;:\; \mathit{Environment} \times (\mathit{ClassType})\mathbf{set} \rightarrow (\mathit{ClassType})\mathbf{set}$$

$$\begin{array}{lcl}
\mathsf{getCheckedException}(\Gamma, \emptyset) & = & \emptyset \\
\mathsf{getCheckedException}(\Gamma, \{\varsigma\} \cup cs) & = & (\; \mathsf{if} \quad \mathsf{checkedException}(\Gamma, \varsigma) \\
 & & \quad \mathsf{then} \;\; \{\varsigma\} \\
 & & \quad \mathsf{else} \;\; \emptyset \;) \;\; \cup \\
 & & \mathsf{getCheckedException}(\Gamma, cs)
\end{array}$$

4.8 Visibilité

Dans cette section, nous introduisons des fonctions permettant de chercher les champs, les méthodes et les constructeurs visibles à partir d'une classe, d'une interface et d'un tableau. Tous les membres d'une classe, d'une interface ou d'un tableau sont visibles à partir de celle-ci ou de celui-ci.

Visibilité des méthodes

Les méthodes membres d'une classe sont toutes les méthodes déclarées dans la classe et celles héritées de la super-classe et des super-interfaces, si elles existent, qui

$$\frac{\Gamma \vdash \tau_e \sqsubseteq_{impl} \tau}{\Gamma \vdash \tau_e \sqsubseteq_{cast} \tau} \qquad \frac{\square}{\Gamma \vdash \texttt{byte} \sqsubseteq_{cast} \texttt{char}} \qquad \frac{\square}{\Gamma \vdash \texttt{short} \sqsubseteq_{cast} \texttt{byte}}$$

$$\frac{\square}{\Gamma \vdash \texttt{short} \sqsubseteq_{cast} \texttt{char}} \qquad \frac{\square}{\Gamma \vdash \texttt{char} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\Gamma \vdash \texttt{char} \sqsubseteq_{cast} \texttt{short}}$$

$$\frac{\square}{\Gamma \vdash \texttt{int} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\Gamma \vdash \texttt{int} \sqsubseteq_{cast} \texttt{short}} \qquad \frac{\square}{\Gamma \vdash \texttt{int} \sqsubseteq_{cast} \texttt{char}}$$

$$\frac{\square}{\Gamma \vdash \texttt{long} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\Gamma \vdash \texttt{long} \sqsubseteq_{cast} \texttt{short}} \qquad \frac{\square}{\Gamma \vdash \texttt{long} \sqsubseteq_{cast} \texttt{char}}$$

$$\frac{\square}{\Gamma \vdash \texttt{long} \sqsubseteq_{cast} \texttt{int}} \qquad \frac{\square}{\Gamma \vdash \texttt{float} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\Gamma \vdash \texttt{float} \sqsubseteq_{cast} \texttt{short}}$$

$$\frac{\square}{\Gamma \vdash \texttt{float} \sqsubseteq_{cast} \texttt{char}} \qquad \frac{\square}{\Gamma \vdash \texttt{float} \sqsubseteq_{cast} \texttt{int}} \qquad \frac{\square}{\Gamma \vdash \texttt{float} \sqsubseteq_{cast} \texttt{long}}$$

$$\frac{\square}{\Gamma \vdash \texttt{double} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\Gamma \vdash \texttt{double} \sqsubseteq_{cast} \texttt{short}} \qquad \frac{\square}{\Gamma \vdash \texttt{double} \sqsubseteq_{cast} \texttt{char}}$$

$$\frac{\square}{\Gamma \vdash \texttt{double} \sqsubseteq_{cast} \texttt{int}} \qquad \frac{\square}{\Gamma \vdash \texttt{double} \sqsubseteq_{cast} \texttt{long}} \qquad \frac{\square}{\Gamma \vdash \texttt{double} \sqsubseteq_{cast} \texttt{float}}$$

$$\frac{\Gamma \vdash \varsigma_1 \sqsubseteq_{class} \varsigma_2}{\Gamma \vdash \varsigma_2 \sqsubseteq_{cast} \varsigma_1} \qquad \frac{\mathsf{validClass}(\Gamma, \texttt{Object}) \quad \mathsf{validTye}(\Gamma, \alpha)}{\Gamma \vdash \texttt{Object} \sqsubseteq_{cast} \alpha}$$

$$\frac{\begin{array}{c}\mathsf{validClass}(\Gamma, \varsigma) \quad \mathsf{validIface}(\Gamma, \iota) \\ \texttt{final} \notin \Gamma.\mathit{classMap}(\varsigma).\mathit{modifiers} \quad \neg(\Gamma \vdash \varsigma \sqsubset_{implements} \iota) \\ (\texttt{abstract} \in \Gamma.\mathit{classMap}(\varsigma) \Rightarrow \\ \mathsf{sameResultType}(\Gamma, \Gamma.\mathit{classMap}(\varsigma).\mathit{methods}, \Gamma.\mathit{interfaceMap}(\iota).\mathit{methods}))\end{array}}{\Gamma \vdash \varsigma \sqsubseteq_{cast} \iota}$$

$$\frac{\begin{array}{c}\mathsf{validIface}(\Gamma, \iota) \quad \mathsf{validClass}(\Gamma, \varsigma) \\ \texttt{final} \notin \Gamma.\mathit{classMap}(\varsigma).\mathit{modifiers} \quad \neg(\Gamma \vdash \varsigma \sqsubset_{implements} \iota) \\ (\texttt{abstract} \in \Gamma.\mathit{classMap}(\varsigma) \Rightarrow \\ \mathsf{sameResultType}(\Gamma, \Gamma.\mathit{classMap}(\varsigma).\mathit{methods}, \Gamma.\mathit{interfaceMap}(\iota).\mathit{methods}))\end{array}}{\Gamma \vdash \iota \sqsubseteq_{cast} \varsigma}$$

$$\frac{\mathsf{validIface}(\Gamma, \iota_1) \quad \mathsf{validIface}(\Gamma, \iota_2) \quad \neg(\Gamma \vdash \iota_1 \sqsubseteq_{interface} \iota_2) \quad \mathsf{sameResultType}(\Gamma, \Gamma.\mathit{interfaceMap}(\iota_1).\mathit{methods}, \Gamma.\mathit{interfaceMap}(\iota_2).\mathit{methods})}{\Gamma \vdash \iota_1 \sqsubseteq_{cast} \iota_2}$$

$$\frac{\Gamma \vdash \mu_1 \sqsubseteq_{cast} \mu_2}{\Gamma \vdash \mu_1[\,] \sqsubseteq_{cast} \mu_2[\,]}$$

TAB. 4.18: Relation de conversion par coercition

ne sont pas sur-définies ou cachées dans la classe. Les méthodes privées ne sont pas héritées. De même, les méthodes membres d'une interface sont toutes les méthodes déclarées dans celle-ci ainsi que toutes les méthodes héritées des super-interfaces, si elles existent, qui ne sont pas sur-définies dans l'interface. En plus, une interface hérite de toutes les méthodes de la classe `Object` qui ne sont pas sur-définies dans l'interface. Une interface ou une classe abstraite peut hériter plus d'une méthode avec la même signature.

Afin de représenter les méthodes visibles à partir d'une classe, d'une interface ou même d'un tableau, nous définissons une mappe de *Sig* vers B_1, telle que :

$$B_1 = (\mathit{MethodInfo} \times \mathit{ReferenceType})\mathbf{set},$$

qui associe à une signature un ensemble décrivant des méthodes. Cette mappe est différente de celle utilisée dans l'environnement qui associe une seule méthode à chaque signature. Afin de pouvoir passer de cette mappe à une mappe de *Sig* vers B_1, nous définissons la fonction $\mathsf{transform_m}$ définie comme suit :

$$\mathsf{transform_m} \;:\; \mathit{ReferenceType} \times (\mathit{Sig} \rightharpoonup \mathit{MethodInfo}) \to (\mathit{Sig} \rightharpoonup B_1)$$

$$\begin{array}{lcl}
\mathsf{transform_m}(\rho, []) & = & [] \\
\mathsf{transform_m}(\rho, [\mathit{sig} \mapsto \langle rt, ms, tcs\rangle] \dagger m) & = & [\mathit{sig} \mapsto \{(\langle rt, ms, tcs\rangle, \rho)\}] \dagger \\
 & & \mathsf{transform_m}(\rho, m)
\end{array}$$

La fonction ci-dessus ajoute à chaque description de méthode la classe, l'interface ou le tableau qui la définit.

Dans ce qui suit, nous donnons les définitions des fonctions qui permettent de créer des mappes contenant les méthodes visibles à partir d'une classe ou d'une interface. L'opérateur $\ddagger_m$ utilisé dans ces fonctions sera expliqué après la présentation de ces dernières.

La fonction $\mathsf{methods_i}$ permet de créer une mappe contenant les méthodes membres d'une interface et elle est définie ci-dessous :

$$\mathsf{methods_i} \;:\; \mathit{Environment} \times \mathit{InterfaceType} \to (\mathit{Sig} \rightharpoonup B_1)$$

$$\begin{array}{lcl}
\mathsf{methods_i}(\Gamma, \iota) & = & (\mathsf{transform_m}(\iota, \Gamma.\mathit{interfaceMap}(\iota).\mathit{methods}) \dagger \\
 & & \mathsf{transform_m}(\texttt{Object}, \Gamma.\mathit{classMap}(\texttt{Object}).\mathit{methods})) \dagger \\
 & & \mathsf{methodsIfaceSet}(\Gamma, \Gamma.\mathit{interfaceMap}(\iota).\mathit{interfaces})
\end{array}$$

Cette fonction calcule la mappe des méthodes d'une interface et l'augmente par celle des méthodes de la classe `Object` qui ne sont pas sur-définies dans l'interface. Ensuite, elle augmente cette mappe par la mappe des super-interfaces de l'interface qui ne sont pas sur-définies dans celle-ci. La sur-définition est gérée par l'opérateur $\dagger$ qui n'augmente une mappe que par les associations qu'elle ne contient pas. Étant donné que l'héritage multiple par les interfaces est permis, nous faisons appel à la fonction methodsIfaceSet qui retourne une mappe contenant toutes les méthodes qui sont membres de l'ensemble des super-interfaces. Elle est définie comme suit :

$$\mathsf{methodsIfaceSet} \;:\; \mathit{Environment} \times (\mathit{InterfaceType})\mathbf{set} \to (\mathit{Sig} \rightharpoonup B_1)$$

$$\begin{array}{lcl}
\mathsf{methodsIfaceSet}(\Gamma, \emptyset) & = & [] \\
\mathsf{methodsIfaceSet}(\Gamma, \{\iota\} \cup is) & = & \mathsf{methods_i}(\Gamma, \iota) \;\ddagger_m\; \mathsf{methodsIfaceSet}(\Gamma, is)
\end{array}$$

La fonction **methods$_c$** permet de créer une mappe des méthodes membres d'une classe et elle est définie comme suit :

$$\mathsf{methods_c} \quad : \quad \mathit{Environment} \times \mathit{ClassType} \rightarrow (\mathit{Sig} \overset{m}{\rightarrow} B_1)$$

$$\begin{array}{l}
\mathsf{methods_c}(\Gamma, \varsigma_1) \quad = \\
\quad \mathsf{if} \quad \varsigma_1 = \mathsf{Object} \\
\quad \mathsf{then} \quad \mathsf{transform_m}(\varsigma_1, \Gamma.\mathit{classMap}(\varsigma_1).\mathit{methods}) \\
\quad \mathsf{else} \quad \mathsf{let}\ \langle ms, \{\varsigma_2\}, is, fm, mm, cm\rangle = \Gamma.\mathit{classMap}(\varsigma_1) \\
\qquad \mathsf{in}\ \mathsf{transform_m}(\varsigma_1, mm) \ \dagger \\
\qquad\quad (\ (\ \mathsf{methods_c}(\Gamma, \varsigma_2) \setminus \mathsf{privateMethods}(\\
\qquad\qquad\qquad\qquad \mathsf{transform_m}(\varsigma_1, \Gamma.\mathit{classMap}(\varsigma_2).\mathit{methods}))\ )\ \ddagger_m \\
\qquad\quad\ \mathsf{methodsIfaceSet}(\Gamma, is)\ ) \\
\qquad \mathsf{end\ let} \\
\quad \mathsf{end\ if}
\end{array}$$

La fonction **methods$_c$** commence par créer une mappe contenant toutes les méthodes de la classe courante. Ensuite, si cette dernière n'est pas la classe mère `Object`, elle augmente la mappe par les méthodes des super-classes et des super-interfaces qui ne sont pas sur-définies dans la classe. Les méthodes privées des super-classes ne sont pas héritées. La mappe des méthodes privées est calculée grâce à la fonction **privateMethods** définie ci-dessous :

$$\mathsf{privateMethods} \quad : \quad (\mathit{Sig} \overset{m}{\rightarrow} B_1) \rightarrow (\mathit{Sig} \overset{m}{\rightarrow} B_1)$$

$$\begin{array}{l}
\mathsf{privateMethods}([]) \quad = \quad [] \\
\mathsf{privateMethods}([\mathit{sig} \mapsto \{(\langle rt, ms, tcs, \rangle, \rho)\}] \dagger mm) \quad = \\
\quad (\ \mathsf{if} \quad \mathtt{private} \in ms \\
\quad\ \mathsf{then}\ [\mathit{sig} \mapsto \{(\langle rt, ms, tcs, \rangle, \rho)\}] \\
\quad\ \mathsf{else}\ [] \\
\quad\ \mathsf{endif}\ )\ \dagger \\
\quad \mathsf{privateMethods}(mm)
\end{array}$$

Il est possible pour une classe ou une interface d'hériter de deux méthodes ayant la même signature. Dans ce cas, au moins l'une des deux méthodes doit être abstraite. Si l'une des méthodes n'est pas abstraite (dans le cas d'une interface ceci est possible si cette méthode provient de `Object`), alors elle est considérée implanter, et donc sur-définir, la méthode abstraite. Si les deux méthodes sont abstraites, alors la classe ou l'interface hérite des deux méthodes. Dans les fonctions **methods$_i$** et **methods$_c$**, la mappe contenant les méthodes héritées des super-classes et des super-interfaces est calculée grâce à l'opérateur $\ddagger_m$ défini comme suit :

$$\_ \ddagger_m \_ \quad : \quad (\mathit{Sig} \overset{m}{\rightarrow} B_1) \times (\mathit{Sig} \overset{m}{\rightarrow} B_1) \rightarrow (\mathit{Sig} \overset{m}{\rightarrow} B_1)$$

$$\begin{array}{lll}
[] \ddagger_m mm \quad = \quad mm & & \\
([\mathit{sig} \mapsto b] \dagger mm_1) \ \ddagger_m \ mm_2 = & \mathsf{if} & mm_2(\mathit{sig}) = \emptyset \\
& \mathsf{then} & [\mathit{sig} \mapsto b] \ \dagger \ (mm_1 \ \ddagger_m \ mm_2) \\
& \mathsf{else} & \mathsf{if} \quad \mathtt{abstract} \in \mathsf{modifiers}(b) \\
& & \mathsf{then}\ [\mathit{sig} \mapsto b \cup mm_2(\mathit{sig})] \ \dagger \ (mm_1 \ \ddagger_m \ mm_2) \\
& & \mathsf{else}\ [\mathit{sig} \mapsto b] \ \dagger \ (mm_1 \ \ddagger_m \ mm_2)
\end{array}$$

endif

endif

Si l'ensemble des méthodes ayant la même signature contient plus d'un élément, alors forcément toutes ces méthodes sont abstraites. Par conséquent, afin de vérifier si les méthodes d'un ensemble sont abstraites, il suffit qu'une méthode de cet ensemble soit abstraite. La fonction **modifiers** permettant de chercher l'ensemble des modificateurs de n'importe quelle méthode dans un ensemble est définie comme suit :

$$\mathsf{modifiers} \quad : \quad (\mathit{MethodInfo} \times \mathit{ClassOrIfaceType})\mathbf{set} \to (\mathit{ModifierName})\mathbf{set}$$

$$\mathsf{modifiers}(\emptyset) \quad = \quad \emptyset$$
$$\mathsf{modifiers}(\{(\langle rt, ms, tcs\rangle, \mu)\} \cup s) \quad = \quad ms$$

Nous présentons, enfin, la visibilité des méthodes à partir d'un tableau. Les méthodes membres d'un tableau sont celles héritées de la classe `Object` hormis la méthode `clone` qui est sur-définie pour les tableaux. La fonction $\mathbf{methods_a}$ permettant de calculer la mappe de ces méthodes est définie comme suit :

$$\mathsf{methods_a} \quad : \quad \mathit{Environment} \times \mathit{ArrayType} \to (\mathit{Sig} \twoheadrightarrow B_1)$$

$$\mathsf{methods_a}(\Gamma, \alpha) \quad = \quad [(\texttt{clone}, \mathsf{Unit}) \mapsto \{(\langle \texttt{Object}, \{\texttt{public}\}, \emptyset\rangle, \alpha)\}] \quad \dagger$$
$$\mathsf{transform_m}(\texttt{Object}, \Gamma.\mathit{classMap}(\texttt{Object}).\mathit{methods})$$

Dans les règles de typage nous avons besoin d'une fonction qui, selon le type de références donné en paramètre, retourne la mappe des méthodes visibles à partir de ce type. Cette fonction est définie comme suit :

$$\mathsf{methodsVar} \quad : \quad \mathit{Environment} \times \mathit{ReferenceType} \to (\mathit{Sig} \twoheadrightarrow B_1)$$

$$\begin{aligned}\mathsf{methodsVar}(\Gamma, \rho) \quad = \quad & \mathbf{case}\ \rho\ \mathbf{of}\ \varsigma \Rightarrow \mathsf{methods_c}(\Gamma, \varsigma) \\ & \qquad\qquad\ \ \iota \Rightarrow \mathsf{methods_i}(\Gamma, \iota) \\ & \qquad\qquad\ \ \alpha \Rightarrow \mathsf{methods_a}(\Gamma, \alpha) \\ & \mathbf{endcase}\end{aligned}$$

Visibilité des champs

Un champ est membre d'une interface ssi il est déclaré dans cette interface ou bien il est hérité par celle-ci. Comme pour la visibilité des méthodes et afin de représenter les champs visibles à partir d'une classe, d'une interface ou d'un tableau, nous introduisons une mappe de *Identifier* vers B_2, telle que :

$$B_2 = (\mathit{FieldInfo} \times \mathit{ReferenceType})\mathbf{set}$$

La fonction qui permet de retourner la mappe des champs visibles à partir d'une interface est définie comme suit :

$$\mathsf{fields_i} \quad : \quad \mathit{Environment} \times \mathit{InterfaceType} \to (\mathit{Identifier} \twoheadrightarrow B_2)$$

$$\begin{aligned}\mathsf{fields_i}(\Gamma, \iota) \quad = \quad & \mathsf{transform_f}(\iota, \Gamma.\mathit{interfaceMap}(\iota).\mathit{fields}) \dagger \\ & \mathsf{fieldsIfaceSet}(\Gamma, \Gamma.\mathit{interfaceMap}(\iota).\mathit{interfaces})\end{aligned}$$

$$\mathsf{fieldsIfaceSet} \;:\; \mathit{Environment} \times (\mathit{InterfaceType})\mathbf{set} \to (\mathit{Identifier} \twoheadrightarrow B_2)$$

$$\begin{aligned}
&\mathsf{fieldsIfaceSet}(\Gamma, \emptyset) \;=\; [] \\
&\mathsf{methodsIfaceSet}(\Gamma, \{\iota\} \cup \mathit{is}) \;=\; \mathsf{fields_i}(\Gamma, \iota) \;\ddagger_f\; \mathsf{fieldsIfaceSet}(\Gamma, \mathit{is})
\end{aligned}$$

La fonction $\mathsf{fields_i}$ augmente la mappe des champs d'une interface par la mappe des champs des super-interfaces en adoptant la sur-définition. La fonction $\mathsf{transform_f}$ joue le même rôle que la fonction $\mathsf{transform_m}$, sauf qu'elle est appliquée à des mappes de champs et elle est définie comme suit :

$$\mathsf{transform_f} \;:\; \mathit{ReferenceType} \times (\mathit{Identifier} \twoheadrightarrow \mathit{FieldInfo}) \to (\mathit{Identifier} \twoheadrightarrow B_2)$$

$$\begin{aligned}
&\mathsf{transform_f}(\rho, []) \;=\; [] \\
&\mathsf{transform_f}(\rho, [\mathit{fn} \mapsto \langle \mathit{ft}, \mathit{ms}\rangle] \dagger \mathit{fm}) \;=\; [\mathit{fn} \mapsto \{((\mathit{ft}, \mathit{ms}\rangle, \rho)\}] \dagger \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{transform_f}(\rho, \mathit{fm})
\end{aligned}$$

Une interface ou une classe peut hériter de champs ayant le même nom des super-classes et des super-interfaces. Ceci est réalisé grâce à l'opérateur $\ddagger_f$ défini comme suit :

$$\_ \ddagger_f \_ \;:\; (\mathit{Identifier} \twoheadrightarrow B_2) \times (\mathit{Identifier} \twoheadrightarrow B_2) \to (\mathit{Identifier} \twoheadrightarrow B_2)$$

$$\begin{aligned}
&[] \ddagger_f \mathit{fm} \;=\; \mathit{fm} \\
&([\mathit{fn} \mapsto b] \dagger \mathit{fm}_1) \ddagger_f \mathit{fm}_2 \;=\; \mathsf{if}\;\; \mathit{fm}_2(\mathit{fn}) = \emptyset \\
&\qquad\qquad\qquad\qquad\quad \mathsf{then}\;\; [\mathit{fn} \mapsto b] \dagger (\mathit{fm}_1 \ddagger_f \mathit{fm}_2) \\
&\qquad\qquad\qquad\qquad\quad \mathsf{else}\;\; [\mathit{fn} \mapsto b \cup \mathit{fm}_2(\mathit{fn})] \dagger (\mathit{fm}_1 \ddagger_f \mathit{fm}_2) \\
&\qquad\qquad\qquad\qquad\quad \mathsf{endif}
\end{aligned}$$

Un champ est membre d'une classe ssi il est déclaré dans cette classe ou bien il est hérité par celle-ci. La fonction qui permet de retourner la mappe des champs visibles à partir d'une classe est définie comme suit :

$$\mathsf{fields_c} \;:\; \mathit{Environment} \times \mathit{ClassType} \to (\mathit{Identifier} \twoheadrightarrow B_2)$$

$$\begin{aligned}
&\mathsf{fields_c}(\Gamma, \varsigma_1) \;= \\
&\quad \mathsf{if}\;\; \varsigma_1 = \mathsf{Object} \\
&\quad \mathsf{then}\;\; \mathsf{transform_f}(\varsigma_1, \Gamma.\mathit{classMap}(\varsigma_1).\mathit{fields}) \\
&\quad \mathsf{else}\;\; \mathsf{let}\ \langle \mathit{ms}, \{\varsigma_2\}, \mathit{is}, \mathit{fm}, \mathit{mm}, \mathit{cm}\rangle = \Gamma.\mathit{classMap}(\varsigma_1)\ \mathsf{in} \\
&\qquad\quad \mathsf{transform_f}(\varsigma_1, \mathit{fm}) \;\dagger \\
&\qquad\quad (\ (\ \mathsf{fields_c}(\Gamma, \varsigma_2) \setminus \mathsf{privateFields}(\mathsf{transform_f}(\varsigma_1, \Gamma.\mathit{classMap}(\varsigma_2).\mathit{fields})\ )\ \ddagger_f \\
&\qquad\qquad\ \mathsf{fieldsIfaceSet}(\Gamma, \mathit{is})\ ) \\
&\qquad\quad \mathsf{endlet} \\
&\quad \mathsf{endif}
\end{aligned}$$

La fonction privateFields retourne la mappe des champs privés et elle est définie comme suit :

$$\mathsf{privateFields} \;:\; (\mathit{Identifier} \twoheadrightarrow B_2) \to (\mathit{Identifier} \twoheadrightarrow B_2)$$

$$\begin{aligned}
&\mathsf{privateFields}([]) \;=\; [] \\
&\mathsf{privateFields}([\mathit{fn} \mapsto \{(\langle \mathit{ft}, \mathit{ms}\rangle, \rho)\}] \dagger \mathit{fm}) \;= \\
&\quad (\ \mathsf{if}\ \mathtt{private} \in \mathit{ms}\ \mathsf{then}\ [\mathit{fn} \mapsto \{(\langle \mathit{ft}, \mathit{ms}\rangle, \rho)\}]
\end{aligned}$$

```
                    else []
                    end if ) †
privateFields(fm)
```

Enfin, le seul champ visible à partir d'un tableau est le champ **length**. La fonction qui permet de retourner la mappe des champs visibles à partir d'un tableau est définie comme suit :

$$\mathsf{fields_a} \;:\; \mathit{Environment} \times \mathit{ArrayType} \to (\mathit{Identifier} \twoheadrightarrow B_2)$$

$$\mathsf{fields_a}(\Gamma, \alpha) \;=\; [\mathtt{length} \mapsto \{(\langle \mathtt{int}, \{\mathtt{public}, \mathtt{final}\}\rangle, \alpha)\}]$$

Comme pour la visibilité des méthodes, nous avons besoin d'une fonction qui selon le type de référence donné en paramètre, retourne la mappe des champs visibles à partir de ce type. Cette fonction est définie comme suit :

$$\mathsf{fieldsVar} \;:\; \mathit{Environment} \times \mathit{ReferenceType} \to (\mathit{Identifier} \twoheadrightarrow B_2)$$

$$\begin{aligned}\mathsf{fieldsVar}(\Gamma, \rho) \;=\; & \mathsf{case}\ \rho\ \mathsf{of}\ \varsigma \Rightarrow \mathsf{fields_c}(\Gamma, \varsigma_1)\\ & \qquad\qquad \iota \Rightarrow \mathsf{fields_i}(\Gamma, \iota)\\ & \qquad\qquad \alpha \Rightarrow \mathsf{fields_a}(\Gamma, \alpha)\\ & \mathsf{endcase}\end{aligned}$$

Visibilité des constructeurs

Les constructeurs ne sont pas considérés comme membres d'une classe et ne sont jamais hérités. Ainsi, les constructeurs visibles à partir d'une classe sont ceux qui y sont déclarés. Cependant, si cette dernière ne définit pas explicitement de constructeur, le noyau en définit un par défaut qui ne prend pas d'argument. Si cette classe est la classe `Object`, alors le constructeur par défaut est un constructeur sans argument ayant un corps vide. Sinon, le constructeur sans argument de la super-classe directe est le constructeur par défaut. Ainsi la mappe des constructeurs visibles à partir d'une classe est calculée grâce à la fonction suivante :

$$\mathsf{constructors} \;:\; \mathit{Environment} \times \mathit{ClassType} \to (\mathit{Sig} \twoheadrightarrow (\mathit{ConstructorInfo} \times \mathit{ClassType}))$$

$$\begin{aligned}&\mathsf{constructors}(\Gamma, \varsigma_1) \;=\\ &\quad \mathsf{let}\ \Gamma.\mathit{classMap}(\varsigma_1) = \langle ms, is, cs, fm, mm, cm\rangle\ \mathsf{in}\\ &\qquad \mathsf{if}\quad cm = []\\ &\qquad \mathsf{then}\ \mathsf{if}\quad \mathsf{defaultConstructorInvocation}(\Gamma, \varsigma_1) = (\mathsf{true}, tcs, \{\varsigma_2\})\\ &\qquad\qquad \mathsf{then}\ [(\varsigma_1.\mathsf{Unit}) \mapsto ((\{\mathit{public}\}, tcs), \varsigma_2)]\\ &\qquad\qquad \mathsf{else}\ []\\ &\qquad\qquad \mathsf{endif}\\ &\qquad \mathsf{else}\ \mathsf{transforme_c}(\varsigma_1, cm)\\ &\qquad \mathsf{endif}\\ &\quad \mathsf{endlet}\end{aligned}$$

Cette fonction fait appel à la fonction **defaultConstructorInvocation** lorsque la mappe des constructeurs de la classe est vide. Cette fonction retourne un triplet composé d'une variable booléenne, d'un ensemble d'exceptions et d'une classe. La première est vraie

si le constructeur par défaut qui ne prend pas d'arguments est bel et bien trouvé et qu'il n'est pas privé et la valeur faux, sinon. La deuxième représente l'ensemble des exceptions déclarées dans le constructeur par défaut, s'il existe. La troisième est la classe qui définit le constructeur, s'il existe. Cette fonction est définie comme suit :

$$\begin{aligned}&\mathsf{defaultConstructorInvocation} \;:\; \mathit{Environment} \times \mathit{ClassType} \rightarrow \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{bool} \times (\mathit{ClassType})\mathsf{set} \times (\mathit{ClassType})\mathsf{set}\end{aligned}$$

$$\begin{array}{l}\mathsf{defaultConstructorInvocation}(\Gamma, \varsigma_1) \;=\\ \quad \mathsf{if} \quad \varsigma_1 \neq \mathtt{Object}\\ \quad \mathsf{then} \;\; \mathsf{let}\ \{\varsigma_2\} = \Gamma.\mathit{classMap}(\varsigma_1).\mathit{super}\ \mathsf{in}\\ \qquad\quad \mathsf{if} \quad \Gamma.\mathit{classMap}(\varsigma_2).\mathit{constructors} \neq [\,]\\ \qquad\quad \mathsf{then} \;\; \mathsf{case}\ \Gamma.\mathit{classMap}(\varsigma_2).\mathit{constructors}(\varsigma_2, \mathsf{Unit})\\ \qquad\qquad\quad \mathsf{of} \quad \langle ms, tcs\rangle \Rightarrow (\mathtt{private} \notin ms, tcs, \{\varsigma_2\})\\ \qquad\qquad\qquad\quad\ \ \bot \qquad\ \Rightarrow (\mathsf{false}, \emptyset, \emptyset)\\ \qquad\qquad\quad \mathsf{endcase}\\ \qquad\quad \mathsf{else} \;\; \mathsf{defaultConstructorInvocation}(\Gamma, \varsigma_2)\\ \qquad\quad \mathsf{endif}\\ \qquad \mathsf{endlet}\\ \quad \mathsf{else} \;\; (\mathsf{true}, \emptyset, \emptyset)\\ \quad \mathsf{endif}\end{array}$$

La fonction $\mathsf{transform_c}$, utilisée dans la fonction **constructors**, permet d'ajouter la classe qui définit le constructeur. Nous tenons à remarquer que nous n'avons pas besoin de transformer les éléments du co-domaine de la mappe des constructeurs en un ensemble, puisque deux constructeurs ayant la même signature ne peuvent pas être visibles à partir d'une même classe. La fonction $\mathsf{transform_c}$ est définie comme suit :

$$\begin{aligned}&\mathsf{transform_c} \;:\; \mathit{ClassType} \times (\mathit{Sig} \twoheadrightarrow \mathit{ConstructorInfo}) \rightarrow (\mathit{Sig} \twoheadrightarrow \mathit{ConstructorInfo}) \times \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathit{ClassType}\end{aligned}$$

$$\begin{aligned}&\mathsf{transform_c}(\varsigma, [\,]) \;=\; [\,]\\ &\mathsf{transform_c}(\varsigma, [sig \mapsto \langle ms, tcs\rangle] \dagger cm) \;=\; [sig \mapsto (\langle ms, tcs\rangle, \varsigma)] \dagger \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{transform_c}(\varsigma, cm)\end{aligned}$$

4.9 Bonne formation de l'environnement statique

Il est nécessaire de vérifier que l'environnement satisfait certaines propriétés très importantes du langage Java. À titre d'exemple, une classe qui n'est pas abstraite doit fournir une implantation à toutes les méthodes abstraites qu'elle hérite.

4.9.1 Bonne formation des champs des classes

Un champ déclaré dans une classe est considéré bien formé lorsque son type est valide et que l'ensemble de ses modificateurs est composé uniquement des modificateurs `public`, `private`, `final`, `static`, `volatile` et `transient`. De plus, les champs publics ne peuvent pas être également privés. De même, les champs finaux ne peuvent pas être

volatiles. Le prédicat qui permet de vérifier que la déclaration d'un champ est bien formée est donné ci-dessous :

$$\mathsf{wellFormedField_c} \;:\; \mathit{Environment} \times (\mathbf{Identifier} \overset{m}{\rightarrow} \mathit{FieldInfo}) \rightarrow \mathbf{bool}$$

$$\begin{aligned}
&\mathsf{wellFormedField_c}(\Gamma, []) \;=\; \mathbf{true}\\
&\mathsf{wellFormedField_c}(\Gamma, [fn \mapsto \langle ft, ms\rangle] \dagger fm) \;=\\
&\quad \mathsf{validType}(\Gamma, ft) \wedge\\
&\quad ms \subseteq \{\mathbf{public}, \mathbf{private}, \mathbf{final}, \mathbf{static}, \mathbf{volatile}, \mathbf{transient}\} \wedge\\
&\quad \{\mathbf{public}, \mathbf{private}\} \not\subseteq ms \wedge\\
&\quad \{\mathbf{volatile}, \mathbf{final}\} \not\subseteq ms \wedge\\
&\quad \mathsf{wellFormedField_c}(\Gamma, fm)
\end{aligned}$$

4.9.2 Bonne formation des méthodes des classes

Une méthode déclarée dans une classe est dite bien formée ssi les conditions ci-dessous sont remplies :

- son type de paramètre ainsi que son type de retour sont valides ;
- l'ensemble de ses modificateurs d'accès est composé uniquement des mots clés `public`, **`private`**, **`final`**, **`static`**, **`abstract`**, **`native`** et **`synchronized`** ;
- la méthode ne possède pas en même temps le modificateur **`private`** et `public` ;
- si la déclaration de la méthode contient le modificateur **`abstract`**, alors cette déclaration ne doit pas contenir l'un des modificateurs **`private`**, **`final`**, **`static`**, **`native`** ou **`synchronized`** ;
- toutes les exceptions déclarées dans la méthode doivent être des sous-classes de la classe d'exception **`Throwable`**.

Le prédicat qui permet de vérifier que les conditions ci-dessus sont remplies est défini comme suit :

$$\mathsf{wellFormedMethod_c} \;:\; \mathit{Environment} \times (\mathit{Sig} \overset{m}{\rightarrow} \mathit{MethodInfo}) \rightarrow \mathbf{bool}$$

$$\begin{aligned}
&\mathsf{wellFormedMethod_c}(\Gamma, []) \;=\; \mathbf{true}\\
&\mathsf{wellFormedMethod_c}(\Gamma, [(mn, \tau_p) \mapsto \langle rt, ms, tcs\rangle] \dagger mm) \;=\\
&\quad \mathsf{validType}(\Gamma, \tau_p) \wedge \mathsf{validType}(\Gamma, rt) \wedge\\
&\quad ms \subseteq \{\mathbf{public}, \mathbf{private}, \mathbf{final}, \mathbf{static}, \mathbf{abstract}, \mathbf{native}, \mathbf{synchronized}\} \wedge\\
&\quad \{\mathbf{public}, \mathbf{private}\} \not\subseteq ms \wedge\\
&\quad \mathsf{if} \quad \mathbf{abstract} \in ms\\
&\quad \mathsf{then} \;\; \{\mathbf{private}, \mathbf{final}, \mathbf{static}, \mathbf{native}, \mathbf{synchronized}\} \cap ms = \emptyset\\
&\quad \mathsf{endif} \wedge\\
&\quad \mathsf{subClasses}(\Gamma, tcs, \{\mathbf{Throwable}\}) \wedge\\
&\quad \mathsf{wellFormedMethod_c}(\Gamma, mm)
\end{aligned}$$

4.9.3 Bonne formation des constructeurs

Un constructeur est considéré bien formé ssi les conditions suivantes sont remplies :

- le nom du constructeur est le même que le nom de la classe ;

- l'ensemble de ses modificateurs d'accès est composé uniquement des mots clés **public** et **private** ;
- le constructeur ne possède pas en même temps le modificateur **private** et **public** ;
- toutes les exceptions déclarées dans le constructeur doivent être des sous-classes de la classe d'exception **Throwable**.

Le prédicat qui permet de vérifier que ces conditions sont remplies est défini comme suit :

$$\begin{array}{l}\mathsf{wellFormedConstructor_c} \ : \ \mathit{Environment} \times \mathit{ClassType} \times (\mathit{Sig} \overset{m}{\rightarrow} \mathit{ConstructorInfo}) \\ \qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathsf{bool}\end{array}$$

$$\begin{array}{l}\mathsf{wellFormedConstructors_c}(\Gamma, \varsigma, []) \ = \ \mathsf{true} \\ \mathsf{wellFormedConstructors_c}(\Gamma, \varsigma_1, [(\varsigma_2, \tau_a) \mapsto \langle ms, tcs\rangle] \dagger cm) \ = \\ \quad \varsigma_1 = \varsigma_2 \\ \quad \mathsf{validType}(\Gamma, \tau_a) \wedge \\ \quad ms \subseteq \{\mathbf{public}, \mathbf{private}\} \wedge \\ \quad \{\mathbf{public}, \mathbf{private}\} \not\subseteq ms \wedge \\ \quad \mathsf{subClasses}(\Gamma, tcs, \{\mathbf{Throwable}\}) \wedge \\ \quad \mathsf{wellFormedConstructors_c}(\Gamma, \varsigma_1, cm)\end{array}$$

4.9.4 Bonne formation des classes

Le prédicat permettant de vérifier qu'une classe est bien formée est donné ci-dessous :

$$\mathsf{wellFormedClass} \ : \ \mathit{Environment} \times \mathit{ClassType} \rightarrow \mathsf{bool}$$

$$\begin{array}{l}\mathsf{wellFormedClass}(\Gamma, \varsigma_1) \ = \\ \quad \mathsf{let} \ \ \Gamma.\mathit{classMap}(\varsigma_1) = \langle ms, cs, is, fm, mm, cm\rangle \ \ \mathsf{in} \\ \quad\ \neg(\mathsf{validIface}(\Gamma, \varsigma_1)) \ \wedge \\ \quad\ ms \subseteq \{\mathbf{public}, \mathbf{abstract}, \mathbf{final}\} \ \wedge \\ \quad\ \{\mathbf{final}, \mathbf{abstract}\} \not\subseteq ms \ \wedge \\ \quad\ \mathsf{validIfaces}(\Gamma, is) \ \wedge \\ \quad\ \mathsf{case}\ cs\ \mathsf{of} \\ \quad\quad \emptyset \ \Rightarrow \varsigma_1 = \mathbf{Object} \\ \quad\quad \{\varsigma_2\} \ \Rightarrow \mathsf{validClass}(\Gamma, \varsigma_2) \ \wedge \\ \quad\quad\quad\quad \mathbf{final} \notin \Gamma.\mathit{classMap}(\varsigma_2).\mathit{modifiers} \ \wedge \\ \quad\quad\quad\quad \mathsf{if} \ \ \mathsf{abstractMethods}(\mathsf{methods_c}(\Gamma, \varsigma_1)) \neq [] \\ \quad\quad\quad\quad \mathsf{then} \ \ \mathbf{abstract} \in ms \\ \quad\quad\quad\quad \mathsf{endif} \ \wedge \\ \quad\quad\quad\quad \mathsf{wellOverriddenHidden}(\mathsf{transform_m}(mm), \mathsf{methods_c}(\Gamma, \varsigma_2) \setminus \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathsf{privateMethods}(\mathsf{methods_c}(\Gamma, \varsigma_2)) \ \wedge \\ \quad\quad\quad\quad \mathsf{wellOverriddenHidden}(\mathsf{transform_m}(mm), \mathsf{methodsIfaceSet}(is)) \ \wedge \\ \quad\quad\quad\quad \mathsf{wellOverriddenHidden}((\mathsf{methods_c}(\Gamma, \varsigma_2) \setminus \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathsf{abstractMethods}(\mathsf{methods_c}(\Gamma, \varsigma_2))) \setminus \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathsf{privateMethods}(\mathsf{methods_c}(\Gamma, \varsigma_2)),\end{array}$$

$$\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{methodsIfaceSet}(\Gamma, is)) \quad \wedge \\
\qquad\qquad \mathsf{wellInherited}(\mathsf{methods}_c(\Gamma, \varsigma_1)) \\
\quad \mathsf{endcase} \quad \wedge \\
\quad \mathsf{wellFormedField}_c(\Gamma, fm) \quad \wedge \\
\quad \mathsf{wellFormedMethod}_c(\Gamma, mm) \quad \wedge \\
\quad \mathsf{if} \quad cm = [] \\
\quad \mathsf{then} \quad \mathsf{defaultConstructorInvocation}(\Gamma, \varsigma_1) = (\mathsf{true}, tcs, \{\varsigma_3\}) \wedge \\
\qquad \neg\mathsf{checkedException}(\Gamma, tcs) \\
\quad \mathsf{else} \quad \mathsf{wellFormedConstructor}_c(\Gamma, \varsigma_1, cm) \\
\quad \mathsf{endif} \\
\mathsf{endlet}
\end{array}$$

Les conditions que vérifie le prédicat ci-dessus sont les suivantes :

- Aucune interface du programme ne possède le même nom que la classe.
- La classe ne contient pas un modificateur autre que **`public`**, **`abstract`** ou **`final`**.
- La classe ne peut pas avoir dans sa déclaration les modificateurs **`final`** et **`abstract`** en même temps, puisque les classes abstraites sont considérées incomplètes. En plus, si elles sont en même temps finales, alors leur définition ne sera jamais complétée.
- Toutes les interfaces qu'elle implante existent et elles sont valides.
- Si la classe n'est pas la classe mère **`Object`**, alors elle doit avoir une super-classe valide qui n'est pas finale et qui n'est pas en même temps sa sous-classe. Cette condition sera vérifiée dans le prédicat de bonne formation d'un programme.
- Si la classe contient des méthodes abstraites, alors elle doit avoir le modificateur **`abstract`** dans sa déclaration. Cette condition permet de nous assurer également qu'une classe qui n'est pas abstraite fournit une implantation à toutes les méthodes abstraites dont elle hérite. La fonction qui retourne la mappe des méthodes abstraites est définie comme suit :

$$\begin{array}{l}
\mathsf{abstractMethods} \quad : \quad (Sig \twoheadrightarrow B_1) \rightarrow (Sig \twoheadrightarrow B_1) \\
\\
\mathsf{abstractMethods}([]) \quad = \quad [] \\
\mathsf{abstractMethods}([sig \mapsto \{(\langle rt, ms, tcs, \rangle, \mu)\}] \dagger mm) \quad = \\
\quad (\ \mathsf{if} \quad \mathbf{abstract} \in ms \\
\quad \mathsf{then} \quad [sig \mapsto \{(\langle rt, ms, tcs, \rangle, \mu)\}] \\
\quad \mathsf{else} \quad [] \\
\quad \mathsf{endif}\ ) \dagger \\
\quad \mathsf{abstractMethods}(mm)
\end{array}$$

- Tous les champs de la classe sont bien formés.
- Toutes les méthodes de la classe sont bien formées.
- Tous les constructeurs de la classe doivent être bien formés. Comme nous l'avons expliqué plus haut, lorsqu'une classe ne définit pas de constructeur, un constructeur par défaut lui est alloué. Cependant, si le constructeur par défaut n'existe pas ou qu'il est privé, une erreur de compilation se produit. Si par contre ce constructeur existe, il faut nous assurer qu'il ne déclare aucune exception contrôlable dans

sa clause **throws**. En effet, lorsqu'un constructeur déclare des exceptions contrôlables, ceci indique qu'il est possible que ces exceptions soient levées dans le corps du constructeur. Dans ce cas, l'appel par défaut de ce constructeur pourra également lever ces exceptions. Il faut donc définir explicitement un constructeur pour cette classe afin de pouvoir gérer ces exceptions. Nous représentons ces faits par un appel de la fonction **defaultConstructorInvocation** qui, lorsque la classe ne déclare pas de constructeur, doit trouver un constructeur par défaut accessible. En outre, nous exigeons que les exceptions déclarées par ce constructeur ne soient pas des exceptions contrôlables.

- Toute méthode de cette classe qui sur-définit ou cache des méthodes des super-classes et des super-interfaces doit satisfaire les conditions suivantes :
  - La méthode doit avoir le même type de retour que la méthode sur-définie ou cachée.
  - La méthode ne doit pas déclarer plus d'exceptions contrôlables que les méthodes sur-définies ou cachées. En plus, pour chaque exception contrôlable déclarée dans cette méthode, la même exception ou l'une de ses super-classes doit être déclarée dans chaque méthode sur-définie ou cachée.
  - La méthode ne peut pas être privée, puisqu'une méthode ne peut donner moins de droits d'accès que la classe sur-définie ou cachée.
  - Aucune des méthodes sur-définies ou cachées ne peut être finale, puisque les méthodes finales ne peuvent pas être sur-définies ou cachées.
  - Une méthode de classe ne peut pas sur-définir une méthode d'instance et une méthode d'instance ne peut pas sur-définir une méthode de classe.

Toutes ces conditions peuvent être vérifiées grâce au prédicat **wellOverriddenHidden** défini formellement comme suit :

$$\mathsf{wellOverriddenHidden} \quad : \quad (Sig \overset{m}{\rightarrow} B_1) \times (Sig \overset{m}{\rightarrow} B_1) \rightarrow \mathsf{bool}$$

$$\mathsf{wellOverriddenHidden}([], mm) \quad = \quad \mathsf{true}$$

$$\mathsf{wellOverriddenHidden}([sig \mapsto \{(\langle rt_1, ms_1, tcs_1\rangle, \mu_1)\}] \dagger mm_1, mm_2) \quad =$$

$$\quad \mathsf{if}\ mm_2(sig) \neq \{\}\ \mathsf{then}\ \forall \langle rt_2, ms_2, tcs_2\rangle, \mu_2\rangle \in mm_2(sig).$$

$$\qquad rt_1 = rt_2 \ \wedge\ \mathsf{notConflictThrows}_1(tcs_1, tcs_2) \ \wedge$$

$$\qquad \mathbf{private} \notin ms_1 \ \wedge\ \mathbf{final} \notin ms_2 \ \wedge$$

$$\qquad \mathbf{static} \in ms_1 \leftrightarrow \mathbf{static} \in ms_2$$

$$\quad \mathsf{else\ true}$$

$$\quad \mathsf{endif} \ \wedge$$

$$\quad \mathsf{wellOverriddenHidden}(mm_1, mm_2)$$

Le prédicat qui permet de vérifier que les exceptions déclarées dans une méthode sur-définie par une autre méthode sont compatibles avec les exceptions déclarées dans cette dernière est défini ci-dessous :

$$\mathsf{notConflictThrows}_1 \quad :: \quad \Gamma \times (ClassType)\mathsf{set} \times (ClassType)\mathsf{set} \rightarrow \mathsf{bool}$$

$$\begin{array}{lcl}
\mathsf{notConflictThrows_1}(\Gamma, \emptyset, tcs) & = & \mathsf{true} \\
\mathsf{notConflictThrows_1}(\Gamma, tcs_1, tcs_2) & = & \forall \varsigma \in tcs_2.\ \mathsf{if}\ \ \mathsf{checkedException}(\Gamma, \varsigma) \\
 & & \quad\ \mathsf{then}\ \ \mathsf{subClass}(\Gamma, \varsigma, tcs_2) \\
 & & \quad\ \mathsf{endif}
\end{array}$$

Dans le langage Java, les méthodes sur-définies et cachées ne sont pas héritées et elles ne sont donc pas visibles à partir de la sous-classe. C'est pour cette raison que nous devons vérifier ces conditions dans différents niveaux du processus de l'héritage. Premièrement, il faut nous assurer que toutes les méthodes de la classe courante respectent ces conditions lorsqu'elles sur-définissent les méthodes visibles à partir de la super-classe qui ne sont pas privées. Deuxièmement, lorsque les méthodes de la classe courante sur-définissent les méthodes des super-interfaces, nous vérifions que ces conditions sont respectées. Enfin, lorsqu'une classe hérite de méthodes de sa super-classe, ces dernières sont considérées implanter toutes les méthodes abstraites héritées par la classe de ses super-interfaces. Il faut donc nous assurer troisièmement que cette implantation respecte ces conditions. Pour procéder à toutes ces vérifications, nous appelons le prédicat **wellOverriddenHidden** avec différents arguments. Enfin, il reste à vérifier, grâce à la fonction **wellInherited**, que lorsque la classe hérite de plusieurs méthodes abstraites avec la même signature, toutes ces méthodes sont compatibles. Ceci veut dire qu'elles ont le même type de retour et que les exceptions qu'elles déclarent sont compatibles. Ici, il ne s'agit pas de la même notion de compatibilité utilisée dans la fonction wellOverriddenHidden. Il faut juste vérifier qu'il est possible pour une méthode de les implanter toutes les deux. C'est-à-dire que les exceptions contrôlables déclarées dans l'une sont des sous-classes des exceptions déclarées dans l'autre.

$$\mathsf{wellInherited} \ : \ (Sig \xrightarrow{m} B_1) \to \mathsf{bool}$$

$$\begin{array}{lcl}
\mathsf{wellInherited}([]) & = & \mathsf{true} \\
\mathsf{wellInherited}([sig \mapsto b] \dagger mm) & = & (\ \forall(\langle rt_1, ms_1, tcs_1\rangle, \mu_1) \in b \wedge \\
 & & \quad \forall(\langle rt_2, ms_2, tcs_2\rangle, \mu_2) \in b. \\
 & & \quad\ rt_1 = rt_2 \wedge \\
 & & \quad\ \mathsf{notConflictThrows_2}(tcs_1, tcs_2)\ ) \wedge \\
 & & \mathsf{wellInherited}(mm)
\end{array}$$

$$\mathsf{notConflictThrows_2} \ : \ (ClassType)\mathsf{set} \times (ClassType)\mathsf{set} \to \mathsf{bool}$$

$$\begin{array}{l}
\mathsf{notConflictThrows_2}(tcs_1, tcs_2) \ = \\
\quad tcs_1 = \emptyset \vee tcs_2 = \emptyset \vee \\
\quad \mathsf{let}\ ce_1 = \mathsf{getCheckedException}(tcs_1)\ ; \\
\quad\quad ce_2 = \mathsf{getCheckedException}(tcs_2) \\
\quad \mathsf{in}\ \ \forall e \in ce_1.\ \mathsf{subClass}(e, ce_2) \vee \\
\quad\quad \forall e \in ce_2.\ \mathsf{subClass}(e, ce_1) \\
\quad \mathsf{endlet}
\end{array}$$

4.9.5 Bonne formation des champs des interfaces

Le champ d'une interface est bien formé ssi son type est valide et que son ensemble de modificateurs ne contient pas des mots-clés autres que **public**, **final** ou **static**. Le prédicat qui permet de vérifier qu'un champ d'une interface est bien formé est défini comme suit :

$$\mathsf{wellFormedField_i} \quad : \quad \mathit{Environment} \times (\mathit{Identifer} \xrightarrow{m} \mathit{FieldInfo}) \rightarrow \mathsf{bool}$$

$$\begin{aligned}
&\mathsf{wellFormedField_i}(\Gamma, []) \quad = \quad \mathsf{true} \\
&\mathsf{wellFormedField_i}(\Gamma, [fn \mapsto \langle ft, ms \rangle] \dagger fm) \quad = \\
&\quad \mathsf{validType}(\Gamma, ft) \quad \wedge \\
&\quad ms \subseteq \{\texttt{public}, \texttt{final}, \texttt{static}\} \quad \wedge \\
&\quad \mathsf{wellFormedField_i}(\Gamma, fm)
\end{aligned}$$

4.9.6 Bonne formation des méthodes des interfaces

Une méthode d'une interface est dite bien formée ssi les conditions ci-dessous sont remplies :

- son type de paramètre ainsi que son type de retour sont valides ;
- l'ensemble de ses modificateurs d'accès est composé uniquement des mots clés `public` et **`abstract`** ;
- toutes les exceptions déclarées dans la méthode doivent être des sous-classes de la classe d'exception **`Throwable`**.

Le prédicat qui permet de s'assurer que les conditions ci-dessus sont remplies est défini comme suit :

$$\mathsf{wellFormedMethod_i} \quad : \quad \mathit{Environment} \times (\mathit{Sig} \xrightarrow{m} \mathit{MethodInfo}) \rightarrow \mathsf{bool}$$

$$\begin{aligned}
&\mathsf{wellFormedMethod_i}(\Gamma, []) \quad = \quad \mathsf{true} \\
&\mathsf{wellFormedMethod_i}(\Gamma, [(mn, \tau_p) \mapsto \langle rt, ms, tcs \rangle] \dagger mm) \quad = \\
&\quad \mathsf{validType}(\Gamma, \tau_p) \ \wedge \ \mathsf{validType}(\Gamma, rt) \quad \wedge \\
&\quad ms \subseteq \{\texttt{public}, \texttt{abstract}\} \quad \wedge \\
&\quad \mathsf{subClasses}(\Gamma, tcs, \texttt{Throwable}) \quad \wedge \\
&\quad \mathsf{wellFormedMethod_i}(\Gamma, mm)
\end{aligned}$$

4.9.7 Bonne formation des interfaces

Une interface est considérée bien formée ssi les conditions suivantes sont remplies :

- Aucune classe du programme ne possède le même nom que l'interface.
- Toutes ses super-interfaces existent et elles sont valides.
- L'ensemble de ses modificateurs est composé uniquement des mots-clés **`public`** et **`abstract`**.
- Ses super-interfaces ne sont pas en même temps ses sous-interfaces. Cette condition sera vérifiée par le prédicat de bonne formation de l'environnement.

- Les méthodes d'une interface qui sur-définissent les méthodes des super-interfaces doivent remplir certaines conditions spécifiées par le prédicat **wellOverriddenHidden**.
- Lorsqu'une interface hérite plus d'une méthode ayant la même signature, ces méthodes doivent êtres compatibles. Ceci est vérifié grâce au prédicat **wellInherited**.
- Tous ses champs sont bien formés.
- Toutes ses méthodes sont bien formées.

Toutes ces conditions sont vérifiées par le prédicat suivant :

$$\mathsf{wellFormedIface} \quad : \quad \mathit{Environment} \times \mathit{InterfaceType} \rightarrow \mathsf{bool}$$

$$\begin{array}{l}
\mathsf{wellFormedIface}(\Gamma, \iota) \quad = \\
\quad \mathsf{let} \quad \Gamma.\mathit{interfaceMap}(\iota) = \langle ms, is, fm, mm \rangle \\
\quad \mathsf{in} \quad \neg(\mathsf{validClass}(\Gamma, \iota)) \quad \wedge \\
\qquad \mathsf{validIfaces}(\Gamma, is) \quad \wedge \\
\qquad ms \subseteq \{\mathtt{public}, \mathtt{abstract}\} \quad \wedge \\
\qquad \mathsf{wellOverriddenHidden}(\mathsf{transform_m}(mm), \mathsf{methodsIfaceSet}(is)) \quad \wedge \\
\qquad \mathsf{wellInherited}(\mathsf{methods_i}(\Gamma, \iota)) \quad \wedge \\
\qquad \mathsf{wellFormedField_i}(\Gamma, fm) \quad \wedge \\
\qquad \mathsf{wellFormedMethod_i}(\Gamma, mm) \\
\quad \mathsf{end\ let}
\end{array}$$

4.9.8 Bonne formation de l'environnement global

Un environnement est considéré bien formé ssi toutes les classes et les interfaces qu'il contient sont bien formées. Nous supposons que toutes les classes prédéfinies, comme `Object` et `Throwable`, sont bien formées et qu'elles sont déjà dans l'environnement. Les fonctions récursives définies dans la section 4.8, et qui retournent les mappes des membres visibles à partir des interfaces, des classes et des tableaux, ne peuvent être appelées que si le programme ne contient pas de circularité dans la hiérarchie des classes et des interfaces. Afin d'éviter ce problème, nous n'appelons ces fonctions que lorsqu'un programme n'est pas circulaire. Le prédicat permettant de vérifier qu'un environnement est bien formé est défini comme suit :

$$\mathsf{wellFormedEnv} \quad : \quad \mathit{Environment} \rightarrow \mathsf{bool}$$

$$\begin{array}{l}
\mathsf{wellFormedEnv}(\Gamma) \quad = \\
\quad \mathsf{let}\ \Gamma = \langle clm, ifm \rangle \\
\quad \mathsf{in}\ \ \mathsf{wellFormedMap_C}(\Gamma, clm) \\
\qquad \mathsf{wellFormedMap_I}(\Gamma, ifm) \\
\quad \mathsf{endlet}
\end{array}$$

$$\mathsf{wellFormedMap_C} \quad : \quad \mathit{Environment} \times \mathit{ClassDecl} \rightarrow \mathsf{bool}$$

$$\begin{array}{l}
\mathsf{wellFormedMap_C}(\Gamma, []) \quad = \quad \mathsf{true} \\
\mathsf{wellFormedMap_C}(\Gamma, [\varsigma_1 \mapsto \langle ms, cs, is, fm, mm, cm \rangle] \dagger clm) \quad =
\end{array}$$

```
    case cs of ∅    ⇒ true
             {ς2}⇒ ( if    ¬(Γ ⊢ ς2 ⊑class ς1)
                     then  wellFormedClass(Γ, ς2)
                     else  false
                     endif )
    endcase ∧
    wellFormedMapc(clm)

wellFormedMapi  :  Environment × InterfaceDecl → bool

wellFormedMapi(Γ, [])  =  true
wellFormedMapi(Γ, [ι1 ↦ ⟨ms, is, fm, mm⟩] † ifm)  =
    ( if    ∀ι1 ∈ is. ¬(Γ ⊢ ι2 ⊑interface ι1)
      then  wellFormedIface(Γ, ι1)
      else  false
      endif ) ∧
    wellFormedMapi(ifm)
```

4.10 Autres objets sémantiques

Nous introduisons dans cette section d'autres objets sémantiques qui seront utilisés dans les règles de typage et nous expliquons les raisons pour lesquelles nous avons besoin de ces objets.

$\mathcal{LV}$	$\in$	$LocalVarMap = Identifier \xrightarrow{m} (Type, \mathsf{bool})$
$\mathcal{E}, \mathcal{UE}$	$\in$	$ExceptSet = (ClassType)\mathsf{multi\text{-}set}$
$\mathcal{PN}$	$\in$	$PosName = \{\text{'f'},\text{'m'},\text{'c'},\text{'a'}\}$
$\mathcal{PI}$	$\in$	$PosInfo = Type + Sig + \mathsf{Unit}$
$\mathcal{P}$ ou $(\mathcal{PN}, \mathcal{PI})$	$\in$	$Position = PosName \times PosInfo$
$\mathcal{B}$	$\in$	$BoolCouple = \mathsf{bool} \times \mathsf{bool}$
$\mathcal{C}$	$\in$	bool

Variable d'état $\mathcal{B}$

Avant d'expliquer la variable $\mathcal{B}$, que nous appelons variable d'état, il convient d'expliquer quelques notions que nous présentons dans ce qui suit.

Le compilateur du langage Java effectue une analyse de flot de données conservatrice afin de s'assurer que toutes les instructions d'un programme Java sont *accessibles*. Dès le moment où il existe une instruction inaccessible dans un programme Java, la compilation de celui-ci résulte en une erreur. Ceci est une optimisation qu'offre le langage afin d'interdire les codes morts dans un programme. Nous essayons dans ce

qui suit d'expliquer l'accessibilité en Java et la manière dont elle est gérée par le compilateur. Toutes les définitions données sont tirées de la spécification, mais nous n'avons transcrit que les parties couvrant le cadre de cette étude. À titre d'exemple, le traitement des instructions inaccessibles impliquant les instructions comme **break** ou `for` n'est pas présenté. En outre, nous avons corrigé les erreurs qui ont été découvertes dans la spécification.

Une instruction est dite accessible ssi il existe un chemin d'exécution possible à partir du début d'un constructeur ou bien d'une méthode qui contient cette instruction vers l'instruction elle-même. Dans les règles que suit le compilateur afin de déterminer si une instruction est accessible, une instruction ne peut se terminer normalement que si elle est accessible. Ci-dessous, nous définissons les termes *accessible* et *peut se terminer normalement* :

- Un bloc représentant le corps d'une méthode ou d'un constructeur est accessible.
- Un bloc vide peut se terminer normalement ssi il est accessible. Un bloc non vide peut se terminer normalement ssi la dernière instruction dans ce bloc peut se terminer normalement. La première instruction dans un bloc non vide est accessible ssi le bloc est accessible. N'importe quelle instruction d'un bloc non vide est accessible ssi l'instruction qui la précède peut se terminer normalement. Lorsqu'une instruction se termine annormalement, nous modifions la variable $\mathcal{B}$ de telle sorte que le typage échoue lorsque le contrôle est transféré à l'instruction suivante.
- Une déclaration d'une variable locale peut se terminer normalement ssi elle est accessible.
- Une instruction vide peut se terminer normalement ssi elle est accessible.
- Une instruction d'expression peut se terminer normalement ssi elle est accessible.
- Une instruction **`while`** peut se terminer normalement ssi elle est accessible et que l'expression conditionnelle n'est pas le littéral **true**. L'instruction représentant le corps de l'instruction **`while`** est accessible ssi l'instruction **`while`** est accessible et que l'expression conditionnelle n'est pas le littéral **`false`**.
- Les instructions **`return`** et **`throw`** se terminent toujours anormalement.
- L'instruction **`synchronized`** peut se terminer normalement ssi l'instruction représentant son corps peut se terminer normalement. Celle-ci est accessible ssi l'instruction **`synchronized`** est accessible.
- L'instruction **`try`** peut se terminer normalement ssi les deux conditions suivantes sont remplies :
  - le bloc **`try`** peut se terminer normalement ou bien l'un des blocs **`catch`** peut se terminer normalement ;
  - si l'instruction **`try`** possède un bloc **`finally`**, alors celui-ci peut se terminer normalement.
- Un bloc **`try`** est accessible ssi l'instruction **`try`** est accessible.
- Un bloc **`catch`** est accessible ssi l'instruction **`try`** est accessible.

- Un bloc **finally**, s'il existe, est accessible ssi l'instruction **try** est accessible.
- Une instruction **if-then** peut se terminer normalement ssi elle est accessible. L'instruction représentant le corps de la branche **then** est accessible ssi l'instruction **if-then** est accessible.
- Une instruction **if-then-else** peut se terminer normalement ssi l'instruction représentant le corps de la branche **then** peut se terminer normalement ou bien l'instruction représentant le corps de la branche **else** peut se terminer normalement. L'instruction représentant le corps de la branche **then** est accessible ssi l'instruction **if-then-else** est accessible. L'instruction représentant le corps de la branche **else** est accessible ssi l'instruction **if-then-else** est accessible.

Cette notion d'accessibilité cause un problème lorsque nous voulons nous assurer qu'une instruction **return** avec une expression de retour existe et qu'elle est accessible dans le corps d'une méthode dont le type de retour est différent de **void**.

Afin de gérer l'accessibilité en Java, nous introduisons la variable d'état $\mathcal{B}$ composée de deux variables booléennes. La première a la valeur vraie dès que le programme se termine anormalement. La deuxième a la valeur vraie dès que l'instruction **return** existe et qu'elle est accessible. Dans nos règles de typage, afin de nous assurer qu'une instruction ou un bloc est accessible, il suffit que la première condition dans l'état soit fausse. La variable d'état est fournie avec deux prédicats $\wedge$ et $\vee$ surchargés afin qu'ils puissent être appliqués à des couples de variables booléennes au lieu de variables booléennes simples. Ces prédicats sont définis comme suit :

$$\begin{aligned} \_ \wedge \_ &: (\mathsf{bool} \times \mathsf{bool}) \times (\mathsf{bool} \times \mathsf{bool}) \to (\mathsf{bool} \times \mathsf{bool}) \\ \_ \vee \_ &: (\mathsf{bool} \times \mathsf{bool}) \times (\mathsf{bool} \times \mathsf{bool}) \to (\mathsf{bool} \times \mathsf{bool}) \end{aligned}$$

$$\begin{aligned} (b_1, c_1) \wedge (b_2, c_2) &= (b_1 \wedge b_2, c_1 \wedge c_2) \\ (b_1, c_1) \vee (b_2, c_2) &= (b_1 \vee b_2, c_1 \vee c_2) \end{aligned}$$

Mappe des variables locales $\mathcal{LV}$

La mappe des variables locales est une fonction qui associe à chaque nom de variable un type et une variable booléenne. Le type est celui de la variable locale. La variable booléenne est un indicateur sur l'état d'initialisation de la variable : elle prend la valeur vraie si la variable est initialisée et la valeur fausse, sinon. Cette variable booléenne joue un rôle important puisqu'elle permet de vérifier qu'aucune variable n'est utilisée avant qu'elle ne soit préalablement initialisée. En effet, dans le langage Java, le compilateur effectue une analyse de flots de données conservatrice afin de s'assurer qu'il existe un chemin d'exécution possible entre l'initialisation d'une variable locale et son utilisation dans le corps de la méthode ou du constructeur. Dans ce cas, la variable est dite *certainement initialisée*. Nous ne donnons pas les règles que suit le compilateur pour déterminer si une variable locale est certainement initialisée, puisque ceci fait l'objet d'un chapitre dans la spécification de Java. Cependant, nous fournissons quelques explications au fur et à mesure que nous présentons les règles de typage.

La mappe des variables locales est aussi très utile pour déterminer la portée d'une variable. En effet, les variables déclarées dans un bloc, délimité par «{» et «}», ne sont

pas utilisables qu'à l'intérieur du bloc. Afin de représenter ceci dans nos règles, nous restaurons la mappe des variables locales après la fin de chaque bloc.

Les opérateurs $\wedge$ et $\vee$ permettent de réaliser des compositions de deux mappes. Ils sont utilisés dans les règles de typage afin de mettre à jour la mappe $\mathcal{LV}$ après l'évaluation d'une expression ou d'une instruction qui peut modifier l'état d'une variable locale. L'application de l'opérateur $\wedge$ à deux mappes retourne une mappe similaire à son premier argument, sauf que la variable booléenne indiquant l'initialisation d'une variable locale n'a la valeur vraie que si la variable locale est initialisée dans les deux mappes. De même, l'application de l'opérateur $\vee$ à deux mappes retourne une mappe similaire à son premier argument, sauf que la variable booléenne indiquant l'initialisation d'une variable locale n'a la valeur vraie que si la variable locale est initialisée dans au moins l'une des deux mappes. Ces deux opérateurs sont définis comme suit :

$$\begin{array}{lcl} \_ \wedge \_ & : & \mathit{LocalVarMap} \times \mathit{LocalVarMap} \rightarrow \mathit{LocalVarMap} \\ \_ \vee \_ & : & \mathit{LocalVarMap} \times \mathit{LocalVarMap} \rightarrow \mathit{LocalVarMap} \end{array}$$

$$m \wedge [] = m$$

$$[] \wedge m = []$$

$$[v_1 \mapsto (t_1, b_1)] \wedge ([v_2 \mapsto (t_2, b_2)] \dagger m_2) = \textsf{if } v_1 = v_2 \textsf{ then } [v_1 \mapsto (t_1, b_1 \wedge b_2)] \textsf{ else } [v_1 \mapsto (t_1, b_1)] \wedge m_2 \textsf{ endif}$$

$$([v_1 \mapsto (t_1, b_1)] \dagger m_1) \wedge ([v_2 \mapsto (t_2, b_2)] \dagger m_2) = ([v_1 \mapsto (t_1, b_1)] \wedge ([v_2 \mapsto (t_2, b_2)] \dagger m_2)) \dagger (m_1 \wedge ([v_2 \mapsto (t_2, b_2)] \dagger m_2))$$

$$m \vee [] = m$$

$$[] \vee m = []$$

$$[v_1 \mapsto (t_1, b_1)] \vee ([v_2 \mapsto (t_2, b_2)] \dagger m_2) = \textsf{if } v_1 = v_2 \textsf{ then } [v_1 \mapsto (t_1, b_1 \vee b_2)] \textsf{ else } [v_1 \mapsto (t_1, b_1)] \vee m_2 \textsf{ endif}$$

$$([v_1 \mapsto (t_1, b_1)] \dagger m_1) \vee ([v_2 \mapsto (t_2, b_2)] \dagger m_2) = ([v_1 \mapsto (t_1, b_1)] \vee ([v_2 \mapsto (t_2, b_2)] \dagger m_2)) \dagger (m_1 \vee ([v_2 \mapsto (t_2, b_2)] \dagger m_2))$$

Multi-ensemble d'exceptions $\mathcal{E}$

Le multi-ensemble $\mathcal{E}$ est composé des exceptions contrôlables levées ou susceptibles d'être levées dans le corps d'une méthode ou d'un constructeur. Ce multi-ensemble sert pour vérifier que les exceptions levées dans une méthode ou un constructeur sont traitées dans leur corps ou bien déclarées dans leur clause **throws**. L'utilisation des multi-ensembles pour les exceptions permet d'avoir plusieurs occurrences d'une même exception ce qui s'avère utile pour typer les instructions.

Position atteinte du programme $\mathcal{P}$

Pour typer un programme Java, il faut connaître la position atteinte du programme. Entre autres, ceci permet de vérifier que l'expression **this** ne peut pas apparaître dans le corps d'une méthode de classe. La position $\mathcal{P}$ est une paire composée d'un caractère $\mathcal{PN}$ et d'une information $\mathcal{PI}$. Le caractère vaut 'f' lorsque l'instruction ou l'expression à typer apparaît dans une expression d'initialisation d'un champ, 'm' lorsqu'elle apparaît dans une méthode, 'c' lorsqu'elle apparaît dans un constructeur et 'a' lorsque l'expression à typer est un argument utilisé dans l'appel explicite d'un constructeur. Nous avons été amenés à préciser cette dernière position, puisque l'invocation explicite d'un constructeur ne peut pas contenir des invocations de méthodes ou de variables d'instance ni utiliser les mots-clés **this** ou **super**. L'information $\mathcal{PI}$ représente ou bien le type du champ *Type*, la signature de la méthode ou du constructeur *Sig* ou bien, lorsqu'il s'agit d'un argument, le type **Unit**.

Variable booléenne $\mathcal{C}$

Dans le langage Java, il n'est pas permis d'affecter une valeur à un champ final, puisque celui-ci est considéré comme une constante. Afin de représenter cette restriction, nous introduisons une variable booléenne $\mathcal{C}$ qui aura la valeur vraie dès qu'un accès à un champ final est fait. La valeur de cette variable sera vérifiée dans les expressions d'affectation d'un champ.

4.11 Contexte

Nous appelons contexte toute combinaison d'objets ou types sémantiques utilisés avant l'opérateur $\vdash$ dans les règles de typage. Nous détaillons le contenu du contexte au fur et à mesure que nous donnons les règles de typage.

4.12 Règles de typage de la sémantique statique

La sémantique statique est présentée par un ensemble de règles d'inférence où la conclusion est considérée valide à partir du moment où toutes les prémisses sont établies. Lorsque ces dernières sont toujours vraies, nous les représentons par $\square$. La sémantique statique manipule plusieurs formes de séquents selon la nature de la phrase de la grammaire à typer. Cette diversité des séquents découle de la nécessite de véhiculer des informations qui varient selon la nature des éléments à typer. Par exemple, afin de nous assurer de la bonne formation d'une déclaration d'un champ, nous avons besoin d'avoir l'environnement de types et de connaître la classe ou à l'interface courante. À ce niveau du programme, nous ne possédons pas encore de variables locales ni d'un ensemble d'exceptions. Toutefois, lorsqu'il s'agit de typer une instruction ces informations sont disponibles. Elles nous sont utiles afin de nous assurer de la validité de certaines variables locales, de savoir si elles ont été initialisées et afin de propager les exceptions levées et pouvoir les traiter.

Chaque règle grammaticale qui introduit une nouvelle déclaration doit être conforme à sa définition dans l'environnement statique. La relation de conformité est testée dans les règles de typage de la façon suivante : l'enregistrement correspondant à la classe ou l'interface courante ou à un membre est obtenu à partir de l'environnement. Les informations dans cet enregistrement doivent correspondre à celles fournies par la déclaration à typer. Cependant, dans certaines règles de typage, nous ne savons pas si la déclaration à typer se trouve dans une classe ou dans une interface. Dans ce cas, nous faisons appel à la fonction classOrIfaceDecl qui retourne l'enregistrement dans l'environnement qui correspond à la déclaration de la classe ou de l'interface courante. La définition de cette fonction est donnée ci-dessous :

classOrIfaceDecl : $Environment \times ClassOrIfaceType \rightarrow ClassDecl$

classOrIfaceDecl(Γ, μ) =
  if $\Gamma.classMap(\mu) = \langle ms, \{\varsigma\}, is, fm, mm, cm\rangle$ then $\langle ms, \{\varsigma\}, is, fm, mm, cm\rangle$
  else if $\Gamma.interfaceMap(\mu) = \langle ms, is, fm, mm\rangle$ then $\langle ms, \emptyset, is, fm, mm, []\rangle$
    endif
  endif

Lorsqu'il s'agit de retourner un enregistrement correspondant à la déclaration d'une interface, nous retournons l'ensemble vide à la place de la super-classe et une mappe vide de constructeurs.

Afin d'augmenter la lisibilité des règles de typage, lorsque nous obtenons la valeur d'un enregistrement à partir de l'environnement, nous ne spécifions que les champs de l'enregistrement utilisés dans la règle. Les autres champs seront représentés par le caractère « _ ».

4.12.1 Règles de typage des déclarations

Dans cette section, nous présentons les règles de la sémantique statique des déclarations. Pour des raisons de modularité, nous présentons ces règles selon la forme des séquents qu'elles manipulent.

Séquents de la forme : $\Gamma \vdash Declaration: \diamond$

Cette forme de séquents signifie que sous l'environnement statique Γ, la déclaration de la grammaire à typer *Declaration* est bien formée. Les règles de typage qui utilisent cette forme de séquents sont présentées dans le tableau 4.19. Un programme est considéré bien formé ssi toutes les déclarations des classes et des interfaces qu'il contient sont bien formées. De même, une classe ou une interface est bien formée ssi son corps est bien formé et sa déclaration est conforme à sa définition dans l'environnement statique.

Séquents de la forme : $\Gamma, \varsigma \vdash Extends : cs$

Cette forme de séquents signifie que sous un environnement statique Γ et une classe ς. la déclaration *Extends* de la grammaire est bien formée et elle retourne un ensemble de classes. Celui-ci est composé de la classe spécifiée dans la clause **extends**, si elle

PROGRAMME ..... $\Gamma \vdash \mathit{Program} : \diamond$

$$\frac{\square}{\Gamma \vdash \epsilon : \diamond}$$

$$\frac{\Gamma \vdash \mathit{ClassDeclaration} : \diamond \quad \Gamma \vdash \mathit{Program} : \diamond}{\Gamma \vdash \mathit{ClassDeclaration}\ \mathit{Program} : \diamond}$$

$$\frac{\Gamma \vdash \mathit{InterfaceDeclaration} : \diamond \quad \Gamma \vdash \mathit{Program} : \diamond}{\Gamma \vdash \mathit{InterfaceDeclaration}\ \mathit{Program} : \diamond}$$

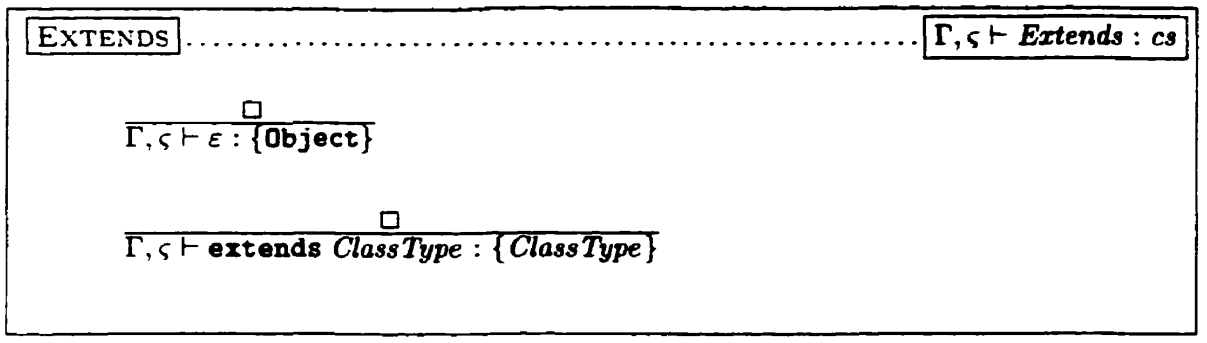
DECL-CLASSE ..... $\Gamma \vdash \mathit{ClassDeclaration} : \diamond$

$$\frac{\begin{array}{c}\Gamma.\mathit{classMap}(\mathit{ClassType}) = \langle ms, \{\varsigma\}, is, fm, mm, cm\rangle \\ \Gamma \vdash \mathit{Modifiers} : ms \quad \Gamma, \mathit{ClassType} \vdash \mathit{Extends} : \{\varsigma\} \quad \Gamma \vdash \mathit{Implements} : is \\ \Gamma, \mathit{ClassType} \vdash \mathit{ClassBodyDeclaration} : \diamond\end{array}}{\begin{array}{c}\Gamma \vdash \mathit{Modifiers}\ \texttt{class}\ \mathit{ClassType}\ \mathit{Extends}\ \mathit{Implements} \\ \{\ \mathit{ClassBodyDeclaration}\ \} : \diamond\end{array}}$$

DECL-IFACE ..... $\Gamma \vdash \mathit{InterfaceDeclaration} : \diamond$

$$\frac{\begin{array}{c}\Gamma.\mathit{interfaceMap}(\mathit{InterfaceType}) = \langle ms, is, fm, mm\rangle \\ \Gamma \vdash \mathit{Modifiers} : ms \quad \Gamma \vdash \mathit{ExtendsInterfaces} : is \\ \Gamma, \mathit{InterfaceType} \vdash \mathit{InterfaceBodyDeclaration} : \diamond\end{array}}{\begin{array}{c}\Gamma \vdash \mathit{Modifiers}\ \texttt{interface}\ \mathit{InterfaceType}\ \mathit{ExtendsInterfaces} \\ \{\ \mathit{InterfaceBodyDeclaration}\ \} : \diamond\end{array}}$$

TAB. 4.19: Règles de typage des déclarations : partie 1



TAB. 4.20: Règles de typage des déclarations : partie 2

existe, et de la classe `Object`, sinon. Les règles de typage manipulant ce genre de séquents sont fournies dans le tableau 4.20.

Séquents de la forme : $\Gamma \vdash$ *Modifiers* : *ms*

Cette forme de séquents signifie que sous l'environnement statique Γ, le non terminal *Modifiers* est bien formé et il retourne un ensemble de modificateurs. Ces règles sont présentées dans le tableau 4.21.

Séquents de la forme : $\Gamma \vdash$*Declaration*: (μ)set

Cette forme de séquents signifie que sous l'environnement statique Γ, la déclaration *Declaration* de la grammaire est bien formée et elle retourne un ensemble d'interfaces ou de classes. Ces règles sont présentées dans le tableau 4.22.

Séquents de la forme : $\Gamma \vdash$ *Parameter* : pn, τ_p

Cette forme de séquents signifie que le non terminal *Parameter* de la grammaire, représentant le paramètre d'une méthode ou d'un constructeur, est bien formé et qu'il s'évalue statiquement en une paire composée du nom du paramètre et de son type. Lorsque la méthode ou le constructeur ne déclare pas de paramètre, l'évaluation statique de *Parameter* doit retourner le couple (ε, Unit). Les règles qui utilisent ce séquent sont présentées dans le tableau 4.23.

Séquents de la forme : $\Gamma, \mu \vdash$*Declaration*: $\diamond$

Cette forme de séquents signifie que sous l'environnement statique Γ et la classe ou l'interface courante μ, la déclaration *Declaration* de la grammaire est bien formée. Les règles de typage manipulant des séquents de cette forme sont celles qui concernent l'évaluation statique des corps des classes et des interfaces incluant les déclarations des champs, des méthodes et des constructeurs qui les constituent. Elles sont présentées dans les tableaux 4.24 et 4.25.

Les déclarations des corps des classes et des interfaces sont considérées bien formées du moment où toutes les déclarations qui les constituent sont bien formées.

MODIFICATEURS ..... $\Gamma \vdash$ *Modifiers* : *ms*

$$\frac{}{\Gamma \vdash \epsilon : \emptyset}\ \square$$

$$\frac{\Gamma \vdash \textit{Modifiers} : ms}{\Gamma \vdash \texttt{public}\ \textit{Modifiers} : \{\texttt{public}\} \cup ms}$$

$$\frac{\Gamma \vdash \textit{Modifiers} : ms}{\Gamma \vdash \texttt{private}\ \textit{Modifiers} : \{\texttt{private}\} \cup ms}$$

$$\frac{\Gamma \vdash \textit{Modifiers} : ms}{\Gamma \vdash \texttt{static}\ \textit{Modifiers} : \{\texttt{static}\} \cup ms}$$

$$\frac{\Gamma \vdash \textit{Modifiers} : ms}{\Gamma \vdash \texttt{abstract}\ \textit{Modifiers} : \{\texttt{abstract}\} \cup ms}$$

$$\frac{\Gamma \vdash \textit{Modifiers} : ms}{\Gamma \vdash \texttt{final}\ \textit{Modifiers} : \{\texttt{final}\} \cup ms}$$

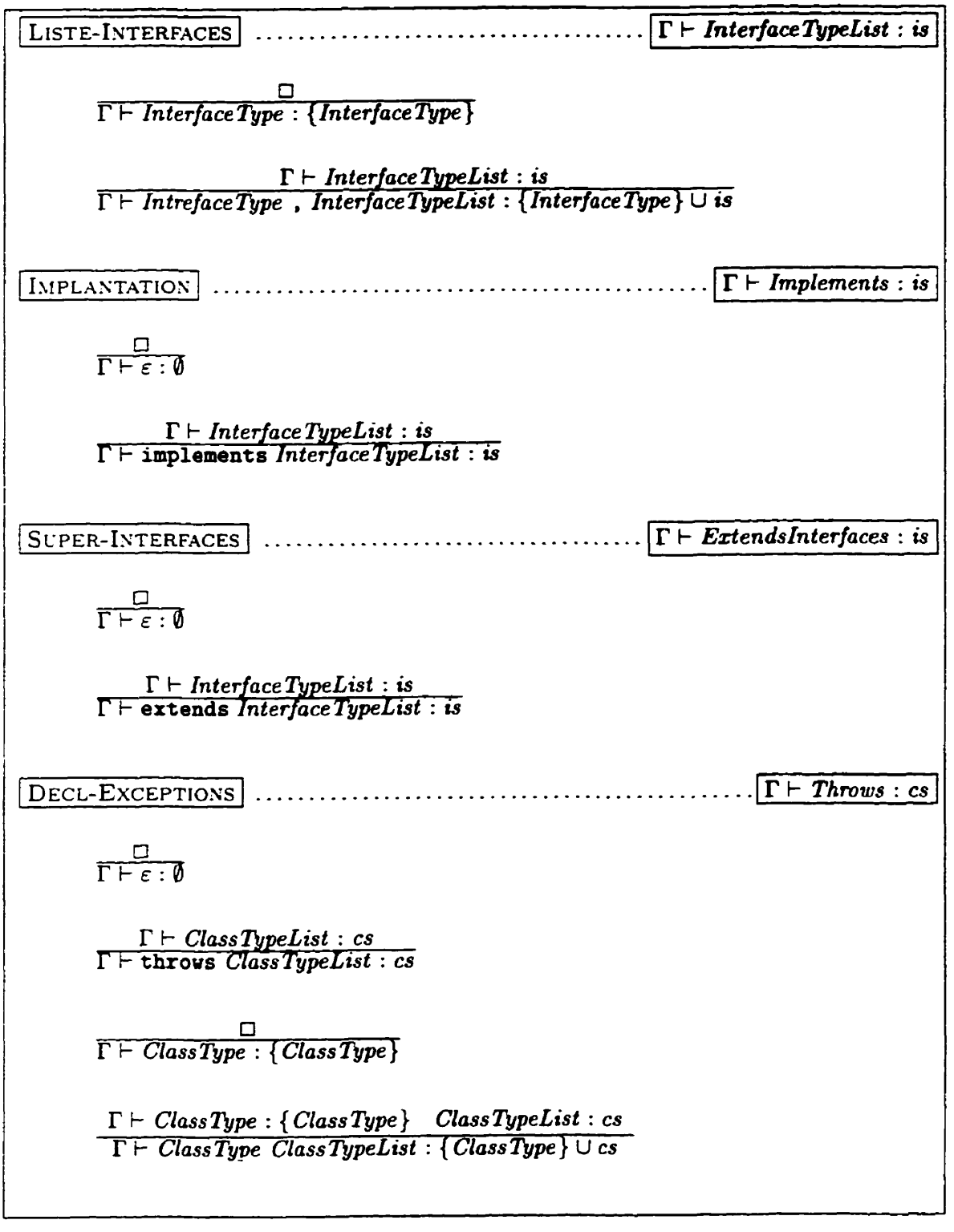
$$\frac{\Gamma \vdash \textit{Modifiers} : ms}{\Gamma \vdash \texttt{synchronized}\ \textit{Modifiers} : \{\texttt{synchronized}\} \cup ms}$$

$$\frac{\Gamma \vdash \textit{Modifiers} : ms}{\Gamma \vdash \texttt{native}\ \textit{Modifiers} : \{\texttt{native}\} \cup ms}$$

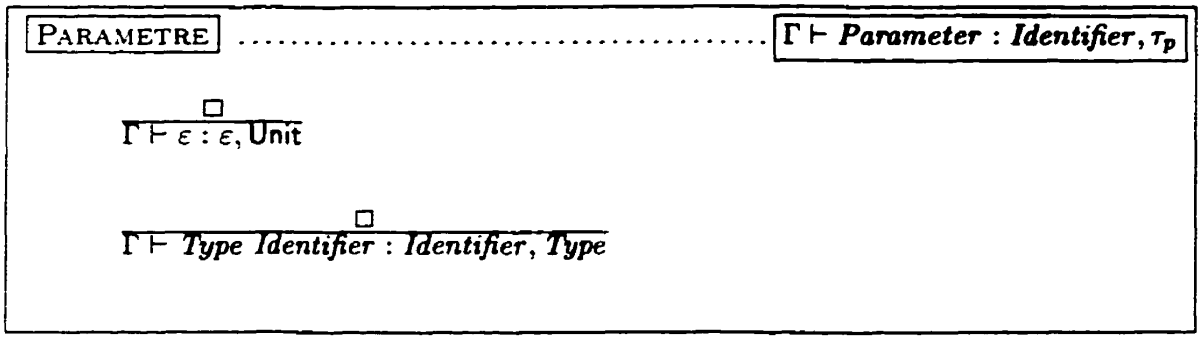
$$\frac{\Gamma \vdash \textit{Modifiers} : ms}{\Gamma \vdash \texttt{volatile}\ \textit{Modifiers} : \{\texttt{volatile}\} \cup ms}$$

$$\frac{\Gamma \vdash \textit{Modifiers} : ms}{\Gamma \vdash \texttt{transient}\ \textit{Modifiers} : \{\texttt{transient}\} \cup ms}$$

TAB. 4.21: Règles de typage des déclarations : partie 3



TAB. 4.22: Règles de typage des déclarations : partie 4



TAB. 4.23: Règles de typage des déclarations : partie 5

Dans ce qui suit, nous donnons des explications relatives aux règles de typage des déclarations de champs, de méthodes et de constructeurs.

- DECL-CHAMP : La valeur d'un champ final est donnée par sa déclaration au moyen d'une expression d'initialisation. Ce champ possédera cette même valeur tout au long du programme. Il n'est donc pas permis de déclarer un champ final sans l'initialiser au moment de sa déclaration. Pour respecter cette restriction, nous exigeons que le modificateur **final** ne fasse pas partie de l'ensemble des modificateurs lorsque sa déclaration n'inclut pas une expression d'initialisation. Comme le lecteur l'a sans doute remarqué, lorsqu'il s'agit d'une déclaration de champ n'incluant pas une expression d'initialisation, la composante du contexte indiquant la classe ou l'interface courante doit être une classe ς, puisque les champs des interfaces sont implicitement finaux. Dans les règles de déclaration d'un champ incluant une expression d'initialisation, l'évaluation statique de cette expression doit rendre un type qui peut être converti implicitement en le type du champ. Lors de l'évaluation de cette expression, la mappe des variables locales est initialement vide, puisque les déclarations des variables locales font partie des blocs Java qui se trouvent uniquement dans les corps des méthodes et des constructeurs. L'évaluation statique de l'expression doit rendre un ensemble vide d'exceptions contrôlables, puisqu'il n'est pas permis en Java que l'initialisation d'un champ se termine anormalement en levant une exception contrôlable. Les règles d'initialisation d'un tableau sont données plus loin dans le tableau 4.27.
- DECL-METHODE : Les méthodes qui possèdent un corps qui n'est pas le terminal « ; » ne peuvent pas contenir l'un des modificateurs **abstract** ou **native** dans leur déclaration. Ceci est vérifié dans la règle au moyen de la condition $\{\texttt{abstract}, \texttt{native}\} \cap M = \emptyset$. Nous rappelons que les méthodes des interfaces sont implicitement abstraites, ce qui ramène notre composante μ du contexte à une classe ς. Le corps de la méthode est un bloc. Les règles de typage des blocs sont présentées plus loin dans le tableau 4.28. Le corps de la méthode doit s'évaluer dans le contexte composé, entre autres, de la mappe des variables locales $[pn \mapsto (\tau_p, \textsf{true})]$. En effet, le nom d'un paramètre peut être utilisé exactement comme une variable locale qui est initialisée lors de l'appel de la méthode. C'est pour cette raison que la variable booléenne indiquant l'initialisation de la variable

possède la valeur vraie. Après l'évaluation du corps de la méthode, la mappe des variables locales doit être vide, puisque les variables déclarées dans la méthode deviennent inutilisables à la fin de celle-ci. Avant l'évaluation du corps de la méthode, l'ensemble des exceptions est vide, mais cette évaluation peut lever des exceptions sans toutefois les traiter. L'évaluation statique de la déclaration de la méthode n'est considérée bien formée que si ces exceptions ou l'une de leurs super-classes sont déclarées dans la clause **throws** de la méthode. Ceci est testé dans la règle via la fonction **unCaughtExceptions** qui doit retourner un ensemble vide d'exceptions. Cette fonction est définie comme suit :

$$\begin{array}{lll} \mathsf{unCaughtExceptions} & :: & \mathit{Environment} \times \mathit{ExceptSet} \times (\mathit{ClassType})\mathsf{set} \rightarrow \\ & & (\mathit{ClassType})\mathsf{set} \\ \\ \mathsf{unCaughtExceptions}(\Gamma, \emptyset, \mathit{tcs}) & = & \emptyset \\ \mathsf{unCaughtExceptions}(\Gamma, \{\!|\varsigma|\!\} \cup \mathcal{E}, \mathit{tcs}) & = & \mathsf{if} \quad \mathsf{subClass}(\Gamma, \varsigma, \mathit{tcs}) \\ & & \mathsf{then} \ \emptyset \\ & & \mathsf{else} \ \{\varsigma\} \\ & & \mathsf{endif} \ \cup \\ & & \mathsf{unCaughtExceptions}(\Gamma, \mathcal{E}, \mathit{tcs}) \end{array}$$

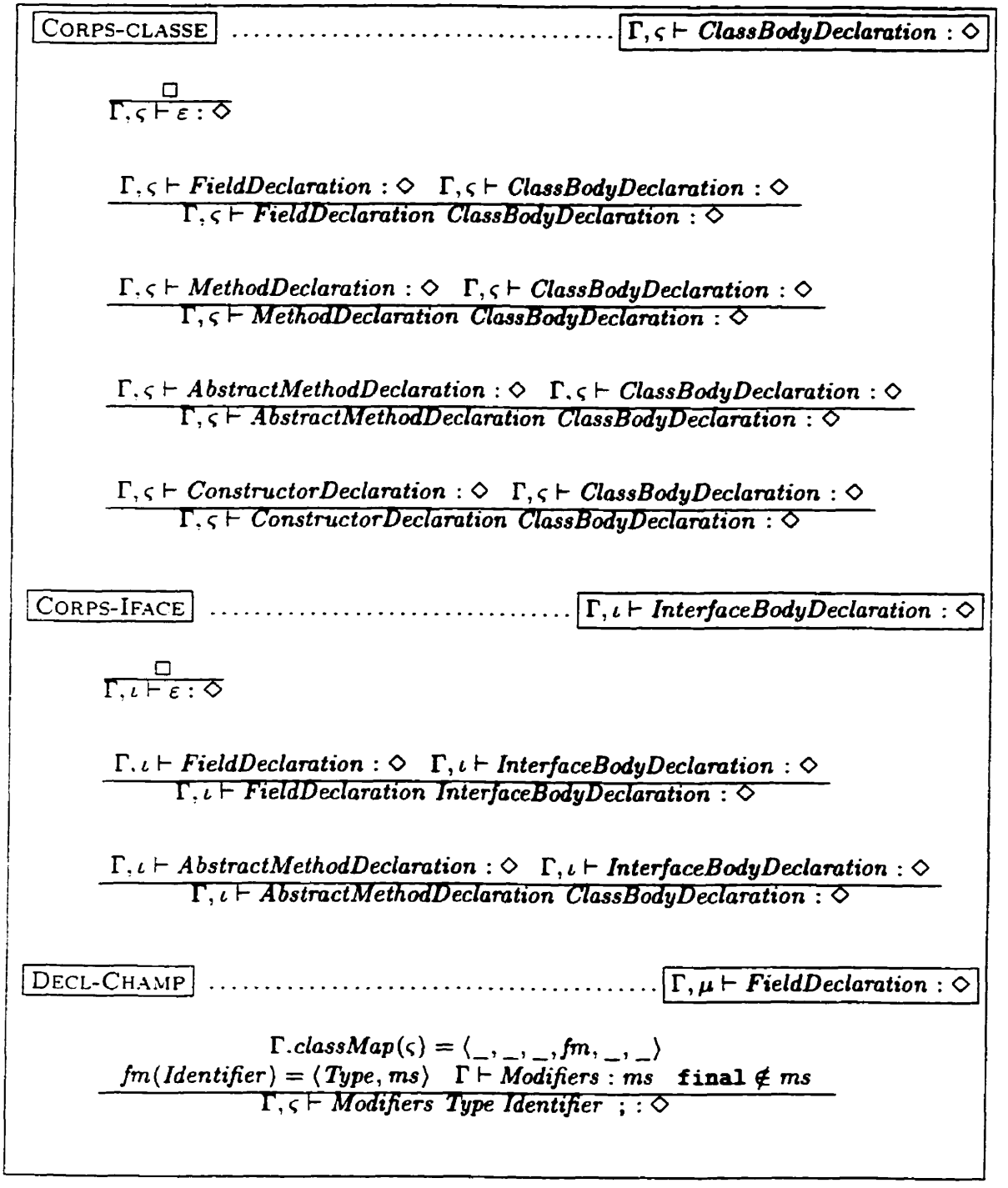
En plus, lorsque le type de retour d'une méthode n'est pas **void**, le corps de la méthode doit contenir au moins une instruction **return** qui est accessible.

- DECL-METHODE-ABSTRAITE : Cette règle ressemble à la précédente, sauf que la méthode à typer possède le terminal « ; » comme corps. Une telle méthode ne peut être déclarée qu'abstraite ou native. Ceci est vérifié grâce à la condition $\{\texttt{abstract}, \texttt{native}\} \cap ms \neq \emptyset$.
- DECL-CONSTRUCRTEUR : La règle d'évaluation statique d'un constructeur suit le même principe que la règle d'évaluation statique d'une déclaration de méthode non abstraite qui possède **void** comme type de retour.

Séquents de la forme : $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{ConstructorBody}: \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2$ et de la forme $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ExplicitConsInvocation}: \mathcal{LV}_2, \mathcal{E}_2$:

La première forme de séquents signifie que sous le contexte $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P}$, le corps d'un constructeur *ConstructorBody* est bien formé et que son évaluation peut modifier les composantes $\mathcal{LV}_1$, $\mathcal{E}_1$ et $\mathcal{B}_1$. La deuxième forme de séquents est utilisée pour typer une invocation explicite d'un constructeur et elle ressemble à la première sauf que le contexte utilisé ne comprend pas la variable d'état. Les règles de typage utilisant des séquents de ces formes sont présentées dans le tableau 4.26.

- CORPS-CONSTRUCTEUR : Le corps d'un constructeur est dit bien formé lorsque l'invocation explicite de constructeur qu'il contient est bien formée et que tous les blocs le constituant sont bien formés. Le contexte d'évaluation statique de l'invocation explicite d'un constructeur ne contient pas la variable d'état, puisque celle-ci ne peut être modifiée que par l'exécution d'une instruction.
- INV-EXPLICITE-CONS : Lorsque le corps d'un constructeur ne commence pas par un appel explicite d'un constructeur (l'invocation explicite est représentée par ε),



TAB. 4.24: Règles de typage des déclarations : partie 6

$$\frac{\mathsf{classOrIfaceDecl}(\Gamma,\mu) = \langle \_,\_,\_,fm,\_,\_\rangle \quad fm(\texttt{Identifier}) = \langle Type, ms\rangle \quad \Gamma \vdash Modifiers : ms \quad \Gamma,\mu,[],\emptyset,(\text{'f'}, Identifier) \vdash Expression : \tau_e,[],\emptyset,\mathcal{C} \quad \Gamma \vdash \tau_e \sqsubseteq_{impl} Type}{\Gamma,\mu \vdash Modifiers\ Type\ Identifier = Expression\ ; : \diamond}$$

$$\frac{\mathsf{classOrIfaceDecl}(\Gamma,\mu) = \langle \_,\_,\_,fm,\_,\_\rangle \quad fm(Identifier) = \langle SimpleType\,[], ms\rangle \quad \Gamma \vdash Modifiers : ms \quad \Gamma,\mu,[],\emptyset,(\text{'f'}, Identifier) \vdash ArrayInitializer : \sigma,[],\emptyset \quad \Gamma \vdash \sigma \sqsubseteq_{impl} SimpleType}{\Gamma,\mu \vdash Modifiers\ SimpleType\ [\,]\ Identifier = ArrayInitializer\ ; : \diamond}$$

DECL-METHODE

$$\frac{\Gamma.classMap(\varsigma) = \langle \_,\_,\_,\_,mm,\_\rangle \quad \Gamma \vdash Parameter : pn,\tau_p \quad mm(Identifier,\tau_p) = \langle ResultType, ms, tcs\rangle \quad \Gamma \vdash Modifiers : ms \quad \Gamma \vdash Throws : tcs \quad \{\texttt{abstract},\texttt{native}\} \cap ms = \emptyset \quad \mathcal{B} = (\mathsf{false},\mathsf{false}) \quad \Gamma,\varsigma.[pn \mapsto (\tau_p,\mathsf{true})],\emptyset,\mathcal{B},(\text{'m'},(Identifier,\tau_p)) \vdash Block : [],\mathcal{E},(b_1,b_2) \quad \mathsf{unCaughtExceptions}(\Gamma,\mathcal{E},tcs) = \emptyset \quad ResultType \neq \texttt{void} \Rightarrow b_2 = \mathsf{true}}{\Gamma,\varsigma \vdash Modifiers\ ResultType\ Identifier\ (\ Parameter\ )\ Throws\ Block : \diamond}$$

DECL-METHODE-ABSTRAITE

$$\frac{\mathsf{classOrIfaceDecl}(\Gamma,\mu) = \langle \_,\_,\_,\_,mm,\_\rangle \quad \Gamma \vdash Parameter : pn,\tau_p \quad mm(Identifier,\tau_p) = \langle ResultType, ms, tcs\rangle \quad \Gamma \vdash Modifiers : ms \quad \Gamma \vdash Throws : tcs \quad \{\texttt{abstract},\texttt{native}\} \cap ms \neq \emptyset}{\Gamma,\mu \vdash Modifiers\ ResultType\ Identifier\ (\ Parameter\ )\ Throws\ ;\ : \diamond}$$

DECL-CONSTRUCTEUR

$$\frac{\Gamma.classMap(\varsigma) = \langle \_,\_,\_,\_,\_,cm\rangle \quad \Gamma \vdash Parameter : pn,\tau_p \quad cm(ClassType,\tau_p) = \langle ms, tcs\rangle \quad \Gamma \vdash Modifiers : ms \quad \Gamma \vdash Throws : tcs \quad \mathcal{B} = (\mathsf{false},\mathsf{false}) \quad \Gamma, ClassType,[pn \mapsto \tau_p],\emptyset,\mathcal{B},(\text{'c'},(ClassType,\tau_p)) \vdash ConstructorBody : [],\mathcal{E},\mathcal{B} \quad \mathsf{unCaughtExceptions}(\Gamma,\mathcal{E},tcs) = \emptyset}{\Gamma,\varsigma \vdash Modifiers\ ClassType\ (\ Parameter\ )\ Throws\ ConstructorBody : \diamond}$$

TAB. 4.25: Règles de typage des déclarations : partie 7

un appel implicite du constructeur sans argument de la super-classe est fait. Comme dans 4.9.4, nous appelons la fonction **defaultConstructorInvocation** afin de vérifier que ce constructeur existe. Si celui-ci est trouvé et qu'il déclare des exceptions contrôlables, alors pour chaque exception contrôlable, le constructeur, dont nous évaluons statiquement le corps, doit déclarer cette même exception ou bien l'une de ses super-classes. Ceci permet d'éviter qu'une exception soit levée lors de l'appel explicite du constructeur sans qu'elle ne soit traitée. L'ensemble des exceptions contrôlables est obtenu grâce à la fonction **getCheckedException**. L'enregistrement correspondant au constructeur courant est obtenu dans la règle grâce à $\Gamma.\mathit{classMap}(\varsigma).\mathit{constructors}(\mathsf{snd}(\mathcal{P}))$. Par ailleurs, lorsque le constructeur commence par une invocation explicite d'un autre constructeur déclaré dans la même classe, il faut chercher ce constructeur et vérifier les exceptions qu'il déclare, comme dans la règle précédente. En outre, il faut nous assurer que le constructeur ne s'appelle pas lui-même. Ceci est vérifié dans la règle par $\mathsf{snd}(\mathsf{snd}(\mathcal{P})) \neq \tau_a$. Enfin, l'évaluation statique d'une invocation explicite du constructeur de la super-classe suit le même principe que la règle précédente. Cependant, au lieu de vérifier que le constructeur ne s'appelle pas lui-même, il faut vérifier que le constructeur cherché dans la mappe des constructeurs visibles à partir de la super-classe n'est pas privé.

Séquents de la forme : $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Declaration}: \sigma, \mathcal{LV}_2, \mathcal{E}_2$

Cette forme de séquents signifie que dans le contexte $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P}$, la déclaration de la grammaire *Declaration* est de type σ et qu'elle peut modifier les composantes $\mathcal{LV}_1$ et $\mathcal{E}_1$ du contexte. Les règles de typage qui utilisent cette forme de séquents sont présentées dans le tableau 4.27. Une expression d'initialisation d'un tableau permet d'initialiser certains de ses éléments. Elle est donnée par une liste d'expressions séparées par une virgule et délimitées par «{» et «}». Lorsqu'aucune expression d'initialisation n'est spécifiée à l'intérieur de ces accolades, l'expression globale d'initialisation peut être utilisée dans l'initialisation des tableaux de n'importe quel type. Si par contre, des expressions d'initialisation sont spécifiées entre ces accolades, toutes ces expressions doivent avoir un type supérieur commun. Ceci permet de vérifier que le type global de l'expression d'initialisation peut être affecté au type du tableau.

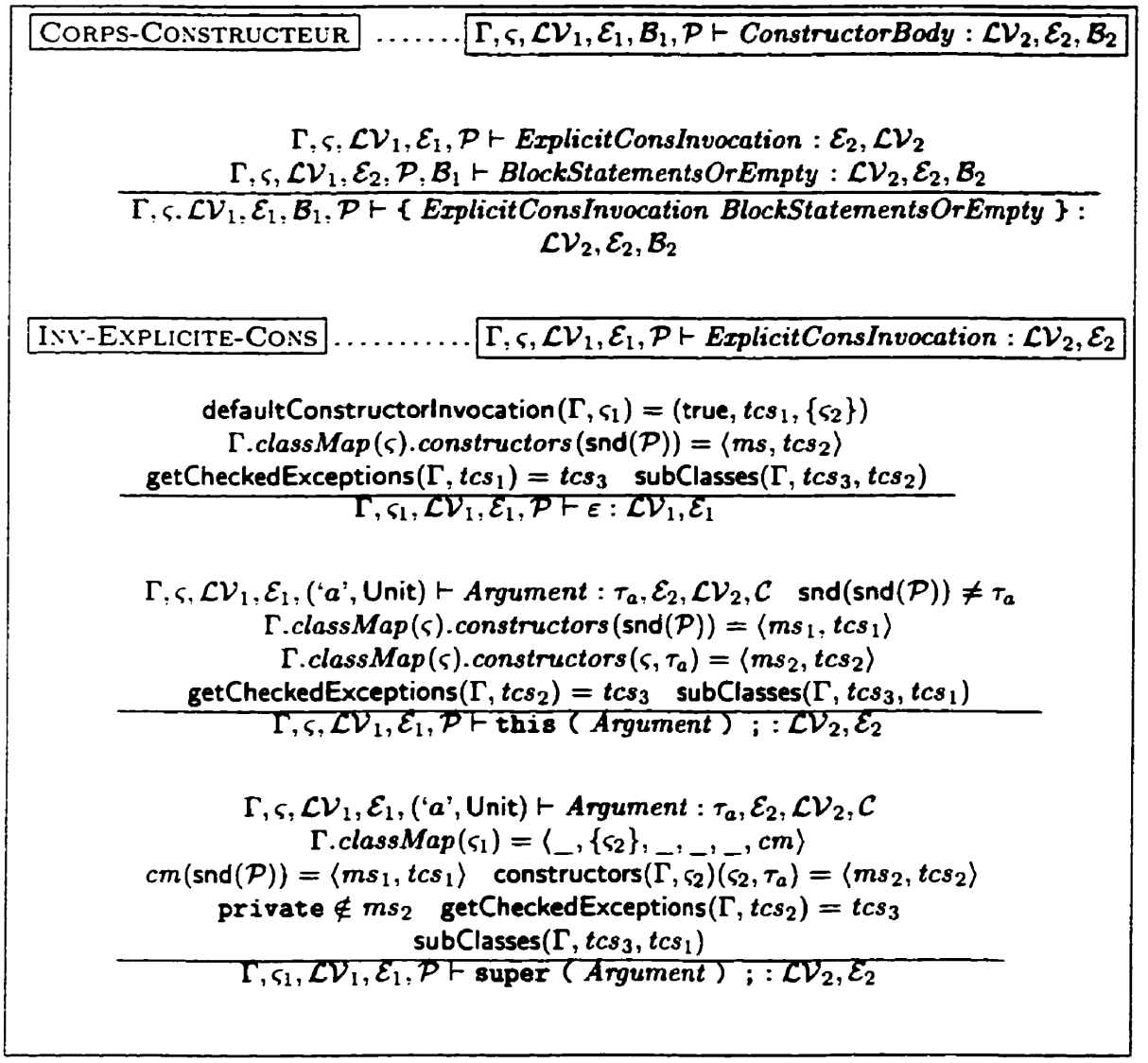
4.12.2 Règles de typage des blocs

Dans cette section, nous donnons les règles de typage des blocs Java. Ces derniers sont constitués de déclarations de variables locales et d'instructions. La sémantique statique qui évalue les blocs utilise des séquents de la forme suivante :

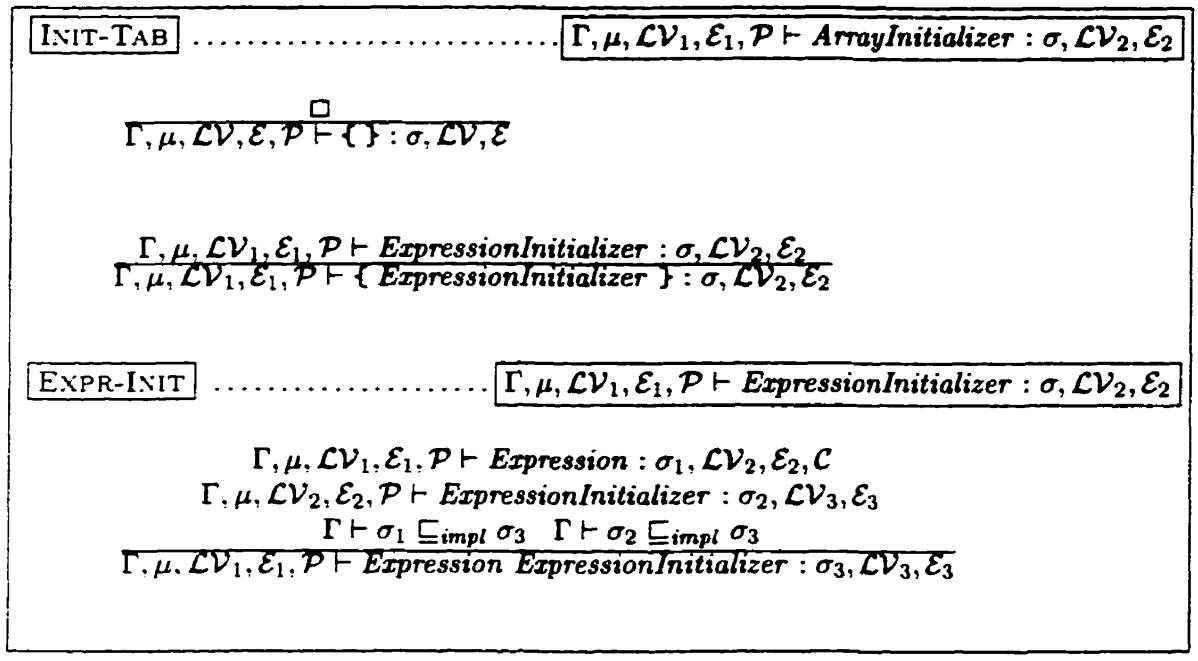
$$\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Bloc} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2$$

qui signifie que dans le contexte statique $\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P}$, le bloc *Bloc* est bien formé et que son évaluation peut modifier les composantes $\mathcal{LV}_1$, $\mathcal{E}_1$ et $\mathcal{B}_1$ du contexte.

La composante du contexte indiquant la classe ou l'interface courante ne peut être qu'une classe ς. En effet, les blocs composent les corps des méthodes non abstraites et



TAB. 4.26: Règles de typage des déclarations : partie 8



TAB. 4.27: Règles de typage des déclarations : partie 9

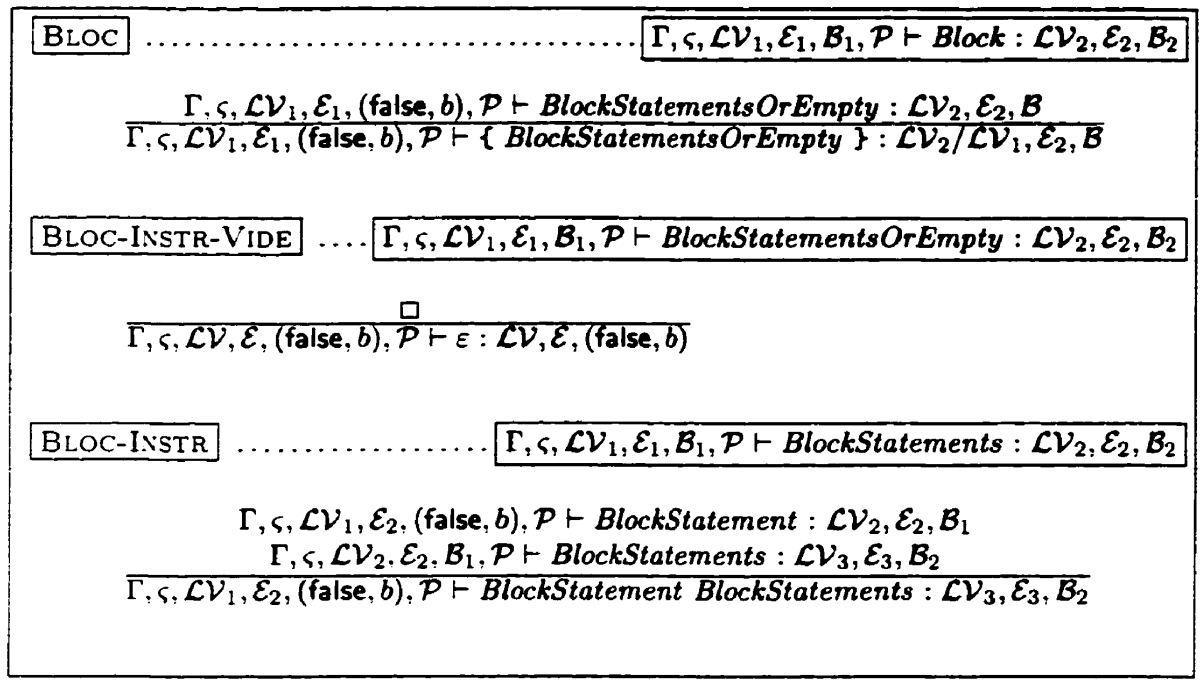
ceux des constructeurs qui ne peuvent pas être déclarés dans une interface. Pour des raisons de clarté, nous décomposons les règles de la sémantique statique concernant les blocs en trois parties. La première introduit les règles de typage des constructions permettant d'introduire une déclaration d'un bloc d'une façon générale. La deuxième concerne les règles de typage des déclarations des variables locales. La dernière concerne l'évaluation statique des instructions.

Blocs

Les règles de la sémantique statique concernant les blocs sont présentées dans le tableau 4.28. Nous donnons ci-dessous quelques explications relatives à ces règles.

Les variables locales déclarées dans un bloc ne sont utilisables que dans celui-ci. À la sortie du bloc, nous devons les supprimer de la mappe des variables locales. Cependant, les modifications apportées dans ce bloc aux variables locales qui ont été déclarées avant le début de celui-ci doivent être visibles après son évaluation. La mappe des variables locales rendue par l'évaluation statique du bloc doit donc contenir exactement les mêmes associations que la mappe des variables locales fournie à l'entrée de celui-ci. Ceci en tenant compte des modifications apportées à ces associations par l'évaluation du bloc. La nouvelle mappe des variables locales est donnée par : $\mathcal{LV}_2/\mathcal{LV}_1$.

L'évaluation statique d'un bloc peut lever une exception contrôlable ce qui explique le changement potentiel de l'ensemble des exceptions $\mathcal{E}_1$ en $\mathcal{E}_2$.



TAB. 4.28: Règles de typage des déclarations de blocs

Variables locales

Les règles de typage concernant les déclarations des variables locales sont présentées dans le tableau 4.29. Dans ce qui suit, nous donnons quelques explications concernant ces règles.

La déclaration d'une variable locale introduit une nouvelle variable. L'évaluation statique d'une déclaration doit donc augmenter la mappe des variables locales $\mathcal{LV}$ par l'association correspondant à cette nouvelle déclaration. Lorsque celle-ci n'inclut pas une expression d'initialisation, la variable locale est considérée non initialisée. Dans le cas contraire, elle est dite initialisée. La condition booléenne représentant l'état d'initialisation de la nouvelle variable doit être fausse dans le premier cas et vraie dans le deuxième. En plus, une variable locale est visible à partir de l'expression d'initialisation incluse dans sa déclaration. À titre d'exemple, la portion du programme Java suivant introduisant une déclaration d'une variable locale est acceptée par le compilateur Java :

```
int m(int p) { return  p ;}
int v = m(v = 3) ;
```

Cependant, la variable v est considérée non initialisée tant que son initialisation n'est pas encore terminée. Ainsi, l'expression d'initialisation est évaluée statiquement en présence de la mappe des variables locales courante augmentée par l'association correspondant à la déclaration de la variable. Nous veillons à ce que la variable d'état indiquant l'initialisation de la variable ait la valeur fausse.

$$\text{VAR-LOCALE} \quad \frac{}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{LocalVariable} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$$

$$\frac{\mathsf{validType}(\Gamma, \mathit{Type}) \quad \mathcal{LV}(\mathit{Identifier}) = \bot}{\Gamma, \varsigma, \mathcal{LV}, \mathcal{E}, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Type}\ \mathit{Identifier}\ ; : \mathcal{LV} \dagger [\mathit{Identifier} \mapsto (\mathit{Type}, \mathsf{false})], \mathcal{E}, (\mathsf{false}, b)}$$

$$\frac{\begin{array}{c}\mathsf{validType}(\Gamma, \mathit{Type}) \quad \mathcal{LV}(\mathit{Identifier}) = \bot \\ \Gamma, \varsigma, \mathcal{LV}_1 \dagger [\mathtt{Identifier} \mapsto (\mathit{Type}, \mathsf{false})], \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \Gamma \vdash \tau_e \sqsubseteq_{\mathit{impl}} \mathit{Type}\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Type}\ \mathit{Identifier} = \mathit{Expression}\ ; : \mathcal{LV}_2 \dagger [\mathit{Identifier} \mapsto (\mathit{Type}, \mathsf{true})], \mathcal{E}_2, (\mathsf{false}, b)}$$

$$\frac{\begin{array}{c}\mathsf{validType}(\Gamma, \mathit{SimpleType}) \quad \mathcal{LV}(\mathit{Identifier}) = \bot \\ \Gamma, \varsigma, \mathcal{LV}_1 \dagger [\mathtt{Identifier} \mapsto (\mathit{SimpleType}\,[\,], \mathsf{false})], \mathcal{E}_1, \mathcal{P} \vdash \mathit{ArrayInitializer} : \sigma, \mathcal{LV}_2, \mathcal{E}_2 \\ \Gamma \vdash \sigma \sqsubseteq_{\mathit{impl}} \mathit{SimpleType}\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{SimpleType}\,[\,]\ \mathit{Identifier} = \mathit{ArrayInitializer}\ ; : \mathcal{LV}_2 \dagger [\mathit{Identifier} \mapsto (\mathit{SimpleType}\,[\,], \mathsf{true})], \mathcal{E}_2, (\mathsf{false}, b)}$$

TAB. 4.29: Règles de typage des déclarations des variables locales

4.12.3 Règles de typage des instructions

Dans cette section, nous présentons les règles de typage des instructions. Les tableaux 4.30, 4.31 et 4.32 présentent la sémantique statique des instructions de Java.

- VIDE : Cette instruction ne fait rien et son évaluation se termine toujours normalement.
- INSTR-EXPR : Une instruction d'expression est évaluée en évaluant l'expression correspondante. Si l'évaluation dynamique de l'expression rend une valeur, alors elle sera ignorée. Statiquement, nous évaluons l'expression et nous ignorons le type rendu.
- IF-THEN : L'exécution du corps de la branche **then** d'une instruction **if-then** dépend de la valeur de la variable conditionnelle. Ainsi, nous ne pouvons pas confirmer qu'après l'évaluation statique de l'instruction **if-then**, les modifications apportées au contexte par une éventuelle évaluation de l'instruction **then** seront réellement effectuées. Par conséquent, même si l'instruction **then** se termine anormalement, nous ne pouvons pas être sûr que l'instruction globale **if-then** se termine également anormalement. De même, si une instruction **return** fait partie du corps de la branche **then**, nous ne pouvons pas confirmer qu'elle sera réellement exécutée. De ce fait, nous ne changeons pas la variable d'état après l'évaluation de l'instruction **if-then**. Cependant, lorsque l'expression conditionnelle est le littéral **true**, il est possible statiquement d'affirmer que le corps de

la branche **then** sera exécutée. Dans ce cas, nous pouvons modifier le contexte après l'évaluation de cette instruction. Toutefois, ceci n'est valable que pour les modifications concernant l'initialisation des variables locales et non la terminaison anormale d'un programme. Ceci est également applicable à l'instruction **if-then-else** et représente l'une des subtilités du langage Java (voir chapitre 3). La fonction qui permet de retourner la mappe des variables locales selon la syntaxe de l'expression conditionnelle est définie formellement comme suit :

$$\begin{aligned}\mathsf{getLocalVarMap}_1 \quad &: \quad \mathit{Expression} \times \mathit{LocalVarMap} \times \mathit{LocalVarMap} \rightarrow \\ &\quad \mathit{LocalVarMap}\end{aligned}$$

$$\begin{aligned}\mathsf{getLocalVarMap}_1(e, m_1, m_2) \quad = \quad & \mathsf{if} \quad e = \texttt{true} \\ & \mathsf{then} \quad m_2 \\ & \mathsf{else} \quad m_1 \\ & \mathsf{endif}\end{aligned}$$

- IF-THEN-ELSE : Dans une instruction **if-then-else**, toute modification du contexte est visible après l'évaluation de cette instruction lorsqu'elle est apportée au contexte et dans le corps de la branche **then** et dans celui de la branche **else** de l'instruction **if**. Pour la mise à jour de la variable d'état, ceci est réalisé dans la règle grâce à l'expression $\mathcal{B}_2 \wedge \mathcal{B}_3$. En plus, une variable locale est considérée initialisée lorsqu'elle est initialisée dans le corps de la branche **then** et que l'expression conditionnelle est le littéral **true** ou bien elle initialisée dans le corps de la branche **else** et que l'expression conditionnelle est le littéral **false**. La fonction qui permet de faire la mise à jour de la mappe des variables locales est donnée par la définition suivante :

$$\begin{aligned}\mathsf{getLocalVarMap}_2 \quad &: \quad \mathit{Expression} \times \mathit{LocalVarMap} \times \mathit{LocalVarMap} \rightarrow \\ &\quad \mathit{LocalVarMap}\end{aligned}$$

$$\begin{aligned}\mathsf{getLocalVarMap}_2(e, m_1, m_2) \quad = \quad & \mathsf{if} \quad e = \texttt{true} \\ & \mathsf{then} \quad m_1 \\ & \mathsf{else} \quad \mathsf{if} \quad e = \texttt{false} \\ & \qquad \mathsf{then} \quad m_2 \\ & \qquad \mathsf{else} \quad m_1 \wedge m_2 \\ & \qquad \mathsf{endif} \\ & \mathsf{endif}\end{aligned}$$

- WHILE : Si l'expression conditionnelle est le littéral **false**, alors le corps de l'instruction **while** est considéré inaccessible. C'est pour cette raison que nous vérifions, dans les règles, que l'expression conditionnelle n'est pas le littéral **false**. Si l'expression conditionnelle est le littéral **true**, alors il y a une détection d'une boucle infinie. Dans ce cas, la condition booléenne représentant la terminaison anormale doit être vraie après l'évaluation de l'instruction **while**. Si par contre, l'expression conditionnelle n'est ni le littéral **true** ni le littéral **false**, alors nous ne pouvons pas confirmer que le corps de l'instruction **while** sera exécutée. Ainsi, le contexte ne doit pas changer après l'évaluation de l'instruction **while**. Afin d'augmenter la lisibilité des règles, nous avons donné deux règles pour l'instruction **while**.

- SYNCHRONIZED : L'évaluation statique de l'instruction **synchronized** est simple. La seule vérification se fait au niveau du type de *Expression* qui doit être un type de références ρ.
- THROW : L'instruction **throw** est utilisée pour lever des exceptions qui doivent être capturées dans une instruction **try** englobante ou bien déclarées dans la méthode qui contient l'instruction **throw**. Afin de nous assurer que ces exceptions seront traitées, nous les ajoutons à l'ensemble des exceptions $\mathcal{E}$. Cependant, ceci n'est utile que si les exceptions levées sont des exceptions contrôlables. En effet, les exceptions non contrôlables ne nécessitent pas qu'elles soient traitées ou déclarées. La mise à jour de cet ensemble se fait dans la règle grâce à $\mathcal{E}_2 \cup \{\!|\varsigma|\!\}$. L'exécution de l'instruction **throw** se termine toujours anormalement. Par conséquent, la variable booléenne indiquant la terminaison anormale du programme doit avoir la valeur vraie après l'évaluation de cette instruction.
- RETURN : Lorsque l'instruction **return** est exécutée dans le corps d'une méthode, il faut nous assurer que le type de l'expression retournée peut être converti implicitement en le type de retour de la méthode. Il est à noter qu'une instruction **return** sans une expression de retour ne peut s'exécuter que dans le corps d'un constructeur ou d'une méthode ayant **void** comme type de retour. Afin de gérer ceci, nous faisons appel à la fonction **getResultType** qui, lorsque la position courante est une méthode, retourne le type de retour de la méthode. Par contre, lorsque c'est un constructeur, elle retourne **void**. Cette fonction est définie comme suit :

$$\mathsf{getResultType} \;:\; \mathit{Environment} \times \mathit{ClassType} \times \mathit{Position} \rightarrow \mathit{ResultType}$$

$$\begin{array}{lcl}
\mathsf{getResultType}(\Gamma, \varsigma, \mathcal{P}) & = & \mathsf{if} \quad \mathsf{fst}(\mathcal{P}) = \text{'m'} \\
& & \mathsf{then} \;\; \Gamma.\mathit{classMap}(\varsigma).\mathit{methods}(\mathsf{snd}(\mathcal{P})).\mathit{resultType} \\
& & \mathsf{else} \;\; \mathsf{if} \quad \mathsf{fst}(\mathcal{P}) = \text{'c'} \\
& & \quad\;\; \mathsf{then} \;\; \mathtt{void} \\
& & \quad\;\; \mathsf{endif} \\
& & \mathsf{endif}
\end{array}$$

- TRY-CATCHES : Dans une instruction **try** possédant au moins une clause **catch**, le flot d'exécution ne passe dans un bloc **catch** que si une exception du même type que le paramètre du **catch** ou un sous-type de celui-ci a été levée dans le bloc **try**. L'ordre des clauses **catch** est important. En effet, slorsqu'une exception est levée dans un bloc **try**, le premier bloc **catch** susceptible de la traiter est celui qui sera exécuté au détriment des autres. L'ensemble des exceptions levées dans le bloc **try** est calculé dans la règle grâce à $\mathcal{E}_2 - \mathcal{E}_1$. L'utilisation des multi-ensembles permet d'avoir toutes les exceptions levées dans le bloc **try** même si ces mêmes exceptions ont déjà été levées avant celui-ci.
  L'évaluation de *Catches*, qui représente les clauses **catch** suivant le bloc **try**, doit rendre, entre autres, un ensemble $\mathcal{UE}$ des exceptions qui n'ont pas été capturées par aucune clause **catch**. Ainsi, l'évaluation de l'instruction **try** globale doit rendre l'ensemble des exceptions levées avant son évaluation augmenté par les exceptions levées dans le bloc **try** qui n'ont pas été capturées dans ses clauses

`catch`. Une variable locale est certainement initialisée après l'évaluation statique d'une instruction `try` ayant des clauses `catch` ssi elle est initialisée dans le bloc `try` et dans tous les blocs `catch` de l'instruction `try`. Ceci est représenté dans la règle par $\mathcal{LV}_2 \wedge \mathcal{LV}_3$.

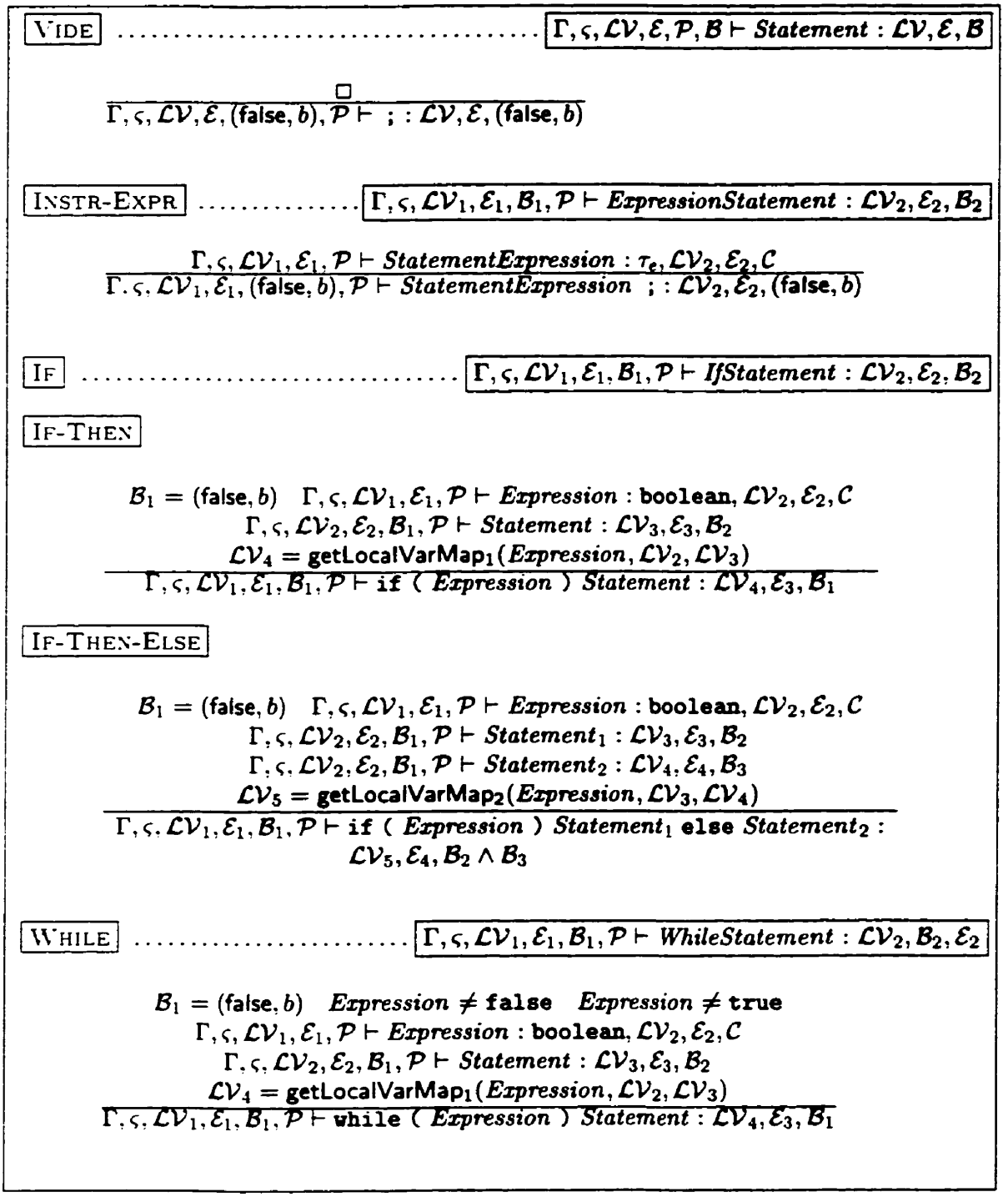
- TRY-CATCHES-FINALLY : L'évaluation d'une instruction `try` ayant des clauses `catch` et une clause `finally` suit le même principe que la règle précédente. Cependant, l'instruction `finally` est toujours exécutée quoi qu'il arrive. Normalement, après l'exécution du bloc `finally`, toutes les exceptions qui ont été levées dans le bloc `try` et dans tous les blocs `catch` et qui n'ont pas été capturées doivent être relancées. Toutefois, il est possible que le bloc `finally` se termine anormalement. Ceci implique que ces exceptions ne seront jamais relancées et donc nous ne sommes pas obligés de les traiter. Pour gérer ceci, nous faisons appel à la fonction remainingExceptions qui retourne l'ensemble des exceptions restantes qui doivent être traitées. Elle est définie comme suit :

$$\begin{array}{lll} \mathsf{remainingExceptions} & : & \mathsf{bool} \times \mathit{ExcepSet} \rightarrow \mathit{ExceptSet} \\ \\ \mathsf{remainingExceptions}(b, \mathcal{E}_1, \mathcal{E}_2) & = & \begin{array}{ll} \mathsf{if} & b = \mathsf{true} \\ \mathsf{then} & \mathcal{E}_1 - \mathcal{E}_2 \\ \mathsf{else} & \mathcal{E}_1 \\ \mathsf{end\ if} & \end{array} \end{array}$$

  Cette instruction se termine anormalement ssi le bloc `try` se termine anormalement et que tous les blocs `catch` se terminent anormalement, ou bien le bloc `finally` se termine anormalement. Ceci est réalisé dans la règle grâce à $(\mathcal{B}_1 \wedge \mathcal{B}_2) \vee \mathcal{B}_3$. De même, une variable locale n'est considérée certainement initialisée après l'évaluation de cette instruction que si elle a été initialisée et dans le bloc `try` et dans tous les blocs `catch`, ou bien qu'elle a été initialisée dans le bloc `finally`. Ceci est réalisé dans la règle par $(\mathcal{LV}_2 \wedge \mathcal{LV}_3) \vee \mathcal{LV}_4$.

- TRY-FINALLY : Cette règle suit le même principe que la règle précédente. Une variable locale est considérée initialisée après l'évaluation de cette instruction lorsqu'elle est initialisée ou bien dans le bloc `try` ou bien dans le bloc `finally`.

- CATCHES : Ces règles permettent de typer les clauses `catch` qui suivent un bloc `try`. Chaque clause `catch` doit avoir exactement un paramètre, appelé *paramètre d'exception*, et doit être la classe `Throwable` ou l'une de ses sous-classes. Ce paramètre ne peut pas avoir le même nom qu'une variable locale ou que le paramètre de la méthode ou du constructeur courant : $\mathcal{LV}_1(\texttt{Identifier}) = \bot$. Ce paramètre d'exception n'est utilisable que dans le bloc du `catch`. Lorsque ce bloc est exécuté, le paramètre reçoit l'objet d'exception capturée. C'est pour cette raison que ce paramètre est considéré initialisé. Enfin, et après l'évaluation de la clause `catch`, il faut mettre à jour l'ensemble des exceptions en supprimant les exceptions qui ont été capturées par la clause `catch`. Ceci est réalisé grâce à la fonction unCaughtExceptions. Une variable locale est considérée initialisée après l'évaluation des blocs `catch` ssi elle a été initialisée dans tous ces blocs.



TAB. 4.30: Règles de typage des instructions : partie 1

$$\frac{\mathcal{B}_1 = (\mathsf{false}, b) \quad \mathit{Expression} \neq \mathtt{false} \quad \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \mathtt{boolean}, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \quad \Gamma, \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Statement} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2 \quad \mathcal{LV}_4 = \mathsf{getLocalVarMap}_1(\mathit{Expression}, \mathcal{LV}_2, \mathcal{LV}_3)}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathtt{while}\ (\ \mathtt{true}\ )\ \mathit{Statement} : \mathcal{LV}_4, \mathcal{E}_3, (\mathsf{true}, b)}$$

SYNCHRONIZED ..... $\boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Synchronized} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$$\frac{\mathcal{B}_1 = (\mathsf{false}, b) \quad \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \rho, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \quad \Gamma, \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Block} : \mathcal{E}_3, \mathcal{LV}_3, \mathcal{B}_2}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathtt{synchronized}\ (\ \mathit{Expression}\ )\ \mathit{Block} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2}$$

THROW ..... $\boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{ThrowStatement} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$$\frac{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \quad \Gamma \vdash \varsigma \sqsubseteq_{class} \mathtt{Error}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathtt{throw}\ \mathit{Expression}\ ; : \mathcal{LV}_2, \mathcal{E}_2, (\mathsf{true}, b)}$$

$$\frac{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \quad \Gamma \vdash \varsigma \sqsubseteq_{class} \mathtt{RuntimeException}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathtt{throw}\ \mathit{Expression}\ ; : \mathcal{LV}_2, \mathcal{E}_2, (\mathsf{true}, b)}$$

$$\frac{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \quad \mathsf{checkedException}(\Gamma, \varsigma)}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathtt{throw}\ \mathit{Expression}\ ; : \mathcal{LV}_2, \mathcal{E}_2 \cup \{\!|\varsigma|\!\}, (\mathsf{true}, b)}$$

RETURN ..... $\boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{ReturnStatement} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$$\frac{\mathsf{getResultType}(\Gamma, \varsigma, \mathcal{P}) \neq \mathtt{void} \quad \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P}_1 \vdash \mathit{Expression} : \tau, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \quad \tau \sqsubseteq_{impl} \mathsf{getResultType}(\Gamma, \varsigma, \mathcal{P})}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathtt{return}\ \mathit{Expression}\ ; : \mathcal{LV}_2, \mathcal{E}_2, (\mathsf{true}, \mathsf{true})}$$

$$\frac{\mathsf{getResultType}(\Gamma, \varsigma, \mathcal{P}) = \mathtt{void}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathtt{return}\ ; : \mathcal{LV}_2, \mathcal{E}_2, (\mathsf{true}, \mathsf{true})}$$

TAB. 4.31: Règles de typage des instructions : partie 2

TRY ..... $\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{TryStatement} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2$

TRY-CATCHES

$$\frac{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Block} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1 \\ \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{E}_2 - \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Catches} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{UE}, \mathcal{B}_2\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathtt{try}\ \mathit{Block}\ \mathit{Catches} : \mathcal{LV}_2 \wedge \mathcal{LV}_3, \mathcal{E}_3 \cup \mathcal{UE}, \mathcal{B}_1 \wedge \mathcal{B}_2}$$

TRY-CATCHES-FINALLY

$$\frac{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Block}_1 : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1 \\ \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{E}_2 - \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Catches} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{UE}, \mathcal{B}_2 \\ \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_3 \cup \mathcal{UE}, (\mathsf{false}, b) \vdash \mathit{Block}_2 : \mathcal{LV}_4, \mathcal{E}_4, \mathcal{B}_3 \\ \mathcal{E}_5 = \mathsf{remainingExceptions}(\mathsf{fst}(\mathcal{B}_3), \mathcal{E}_4, (\mathcal{E}_3 \cup \mathcal{UE}) - \mathcal{E}_1)\end{array}}{\begin{array}{l}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathtt{try}\ \mathit{Block}_1\ \mathit{Catches}\ \mathtt{finally}\ \mathit{Block}_2 : \\ \qquad (\mathcal{LV}_2 \wedge \mathcal{LV}_3) \vee \mathcal{LV}_4, \mathcal{E}_5, (\mathcal{B}_1 \wedge \mathcal{B}_2) \vee \mathcal{B}_3\end{array}}$$

TRY-FINALLY

$$\frac{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Block}_1 : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1 \\ \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_2, (\mathsf{false}, b), \mathcal{P}, \vdash \mathit{Block}_2 : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2 \\ \mathcal{E}_4 = \mathsf{remainingExceptions}(\mathsf{fst}(\mathcal{B}_2), \mathcal{E}_3, \mathcal{E}_2 - \mathcal{E}_1)\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathtt{try}\ \mathit{Block}_1\ \mathtt{finally}\ \mathit{Block}_2 : \mathcal{LV}_3 \vee \mathcal{LV}_4, \mathcal{E}_2, \mathcal{B}_1 \vee \mathcal{B}_2}$$

CATCHES

$$\frac{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{UE}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{CatchClause} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{UE}_2, \mathcal{B}_2 \\ \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_2, \mathcal{UE}_2, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Catches} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{UE}_3, \mathcal{B}_3\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{UE}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{CatchClause}\ \mathit{Catches} : \mathcal{LV}_2 \wedge \mathcal{LV}_3, \mathcal{E}_3, \mathcal{UE}_3, \mathcal{B}_2 \wedge \mathcal{B}_3}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{ClassType} \sqsubseteq_{\mathit{class}} \mathbf{Throwable} \\ \mathcal{LV}_1(\mathit{Identifier}) = \bot \\ \Gamma, \varsigma, \mathcal{LV}_1 \dagger [\mathit{Identifier} \mapsto (\mathit{ClassType}, \mathsf{true})], \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Block} : \\ \mathcal{LV}_2 \dagger [\mathbf{Identifier} \mapsto (\mathit{ClassType}, \mathsf{true})], \mathcal{E}_2, \mathcal{B}_2 \\ \mathsf{unCaughtExceptions}(\Gamma, \mathcal{UE}_1, \mathit{ClassType}) = \mathcal{UE}_2\end{array}}{\begin{array}{l}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{UE}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathtt{catch}\ (\ \mathit{ClassType}\ \mathit{Identifier}\ )\ \mathit{Block} : \\ \qquad \mathcal{LV}_2, \mathcal{E}_2, \mathcal{UE}_2, \mathcal{B}_2\end{array}}$$

TAB. 4.32: Règles de typage des instructions : partie 3

4.12.4 Règles de typage des expressions

Cette section est dédiée à la présentation de la sémantique statique des expressions. Les règles de typage des expressions sont présentées dans les tableaux 4.33, 4.34, 4.35 et 4.36. Ces règles utilisent des séquents de la forme suivante :

$$\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{Expression} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}$$

qui signifie que dans le contexte $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P}$, l'expression *Expression* est de type τ_e et que son évaluation introduit une variable booléenne $\mathcal{C}$ et peut modifier les composantes $\mathcal{LV}_1$ et $\mathcal{E}_1$ du contexte.

Dans ce qui suit, nous donnons quelques explications relatives à ces règles :

- ARGUMENT : Cette règle permet de typer une expression vide (ε) qui est de type Unit.
- NOUVEL-TAB : Une expression de création de tableau crée un objet qui représente un nouveau tableau dont les éléments sont du type spécifié par *SimpleType*. Celui-ci ne peut pas être de type tableau et doit être un type valide. La dimension du tableau, spécifiée par *Expression*, doit être de type `int` ou bien un type primitif qui peut être converti implicitement en celui-ci.
- LITTERAL : Un littéral dénote une valeur fixe. L'évaluation statique d'un littéral ne peux jamais lever une exception. La fonction **typeOf** permet de calculer le type d'un littéral. Nous omettons délibérément la définition de cette fonction étant donné qu'elle est simple. En outre, il convient de signaler que le type du littéral `null`, retourné par cette fonction, est **Null**.
- THIS : Le mot clé `this` ne peut être utilisé que dans le corps d'une méthode d'instance, d'un constructeur ou bien dans l'initialisation d'une variable d'instance. Cette restriction est testée par la fonction **licitInvokeThis** définie comme suit :

$$\begin{array}{l}
\mathsf{licitInvokeThis} \;:\; \textit{Environment} \times \textit{ClassType} \times \textit{Position} \rightarrow \mathsf{bool} \\
\\
\mathsf{licitInvokeThis}(\Gamma, \varsigma, \mathcal{P}) \;=\; \\
\quad \mathsf{validClass}(\Gamma, \varsigma) \;\wedge\; \mathsf{fst}(\mathcal{P}) \neq \text{'a'} \;\wedge \\
\quad \mathsf{if} \;\; \mathsf{fst}(\mathcal{P}) = \text{'m'} \\
\quad \mathsf{then} \;\; \texttt{static} \notin \Gamma.\textit{classMap}(\varsigma).\textit{methods}(\mathsf{snd}(\mathcal{P})).\textit{modifiers} \\
\quad \mathsf{else} \;\; \mathsf{if} \;\; \mathsf{fst}(\mathcal{P}) = \text{'f'} \\
\quad\quad \mathsf{then} \;\; \texttt{static} \notin \Gamma.\textit{classMap}(\varsigma).\textit{fields}(\mathsf{snd}(\mathcal{P})).\textit{modifiers} \\
\quad\quad \mathsf{else} \;\; \mathsf{true} \\
\quad\quad \mathsf{endif} \\
\quad \mathsf{endif}
\end{array}$$

- PARENTHESE : Le type d'une expression parenthésée est le type de l'expression qu'elle contient.
- NOUVEL-OBJET : Les classes abstraites sont considérées incomplètes, il n'est donc pas possible de créer une instance d'une classe abstraite. Lorsqu'un objet instance d'une classe est créé, le constructeur de la classe qui possède un paramètre de type τ_a, le type de *Argument*, est appelé. La fonction **maxSpecConstructor**, utilisée dans la règle, permet de trouver l'ensemble des constructeurs qui

sont *spécifiques au maximum* à cet appel. Nous ne nous attarderons pas sur cette fonction, puisque nous avons déjà expliqué comment trouver une méthode spécifique au maximum dans le chapitre 2. Le même principe est adopté pour trouver le constructeur spécifique au maximum. Cette fonction est définie comme suit :

$$\begin{array}{l}\mathsf{maxSpecConstructors} \;:\; \mathit{Environment} \times \mathit{ClassType} \times \mathit{ArgumentType} \rightarrow \\ \qquad\qquad (\mathit{ConstructorInfo} \times \mathit{ClassType})\mathsf{set}\end{array}$$

$$\begin{array}{l}\mathsf{maxSpecConstructors}(\Gamma, \varsigma, \tau_a) \;=\; \\ \quad \{\; ((\langle ms', tcs'\rangle, \varsigma'), \tau'_a) \mid (((\langle ms', tcs'\rangle, \varsigma'), \tau'_a) \in \mathsf{applConstructors}(\Gamma, \varsigma, \tau_a) \;\wedge \\ \qquad\qquad \forall\, (((\langle ms'', tcs''\rangle, \varsigma'), \tau''_a) \in \mathsf{applConstructors}(\Gamma, \varsigma, \tau_a). \\ \qquad\qquad\quad \mathsf{if} \quad \Gamma \vdash \tau''_a \sqsubseteq_{impl} \tau'_a \\ \qquad\qquad\quad \mathsf{then} \;\; (\langle ms', tcs'\rangle, \varsigma') = (\langle ms'', tcs''\rangle, \varsigma') \\ \qquad\qquad\quad \mathsf{endif}\;\}\end{array}$$

$$\begin{array}{l}\mathsf{applConstructors} \;:\; \mathit{Environment} \times \mathit{ClassType} \times \mathit{ArgumentType} \rightarrow \\ \qquad\qquad ((\mathit{ConstructorInfo} \times \mathit{ClassType}) \times \mathit{ArgumentType})\mathsf{set}\end{array}$$

$$\begin{array}{l}\mathsf{applConstructors}(\Gamma, \varsigma, \tau_a) \;=\; \{\; (ci, \tau'_a) \mid ci = \mathsf{constructors}(\Gamma, \varsigma)(\varsigma, \tau'_a) \;\wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \tau_a \sqsubseteq_{impl} \tau'_a \;\}\end{array}$$

La fonction maxSpecConstructors doit trouver exactement un constructeur. Si elle trouve plus d'un constructeur, alors l'accès est considéré ambigu. Par contre, si elle ne trouve aucun constructeur, alors le constructeur appelé n'existe pas. Dans la règle, cette fonction doit donc retourner une paire composée de l'enregistrement correspondant au constructeur trouvé et de la classe ς qui contient sa définition. Cette dernière sert à vérifier que si le constructeur est privé, alors l'appel au constructeur ne peut se faire qu'à partir de la classe qui le définit. Ceci veut dire que la classe ou l'interface courante μ n'est autre que la classe qui définit le constructeur. Enfin, lorsque le constructeur appelé déclare des exceptions contrôlables, il faut les ajouter à l'ensemble des exceptions afin de nous assurer qu'elles seront traitées. Ceci est réalisé dans la règle par la fonction suivante :

$$\mathsf{throwsExceptions} \;:\; \mathit{Environment} \times (\mathit{ClassType})\mathsf{set} \rightarrow \mathit{ExcepSet}$$

$$\begin{array}{lcl}\mathsf{throwsExceptions}(\Gamma, \emptyset, \mathcal{E}) & = & \mathcal{E} \\ \mathsf{throwsExceptions}(\Gamma, \{\varsigma\} \cup tcs, \mathcal{E}) & = & \mathsf{if} \quad \mathsf{checkedException}(\Gamma, \varsigma) \\ & & \mathsf{then} \;\; \{\!|\varsigma|\!\} \\ & & \mathsf{else} \;\; \emptyset \\ & & \mathsf{endif} \cup \\ & & \mathsf{throwsExceptions}(\Gamma, tcs, \mathcal{E})\end{array}$$

- ACCES-CHAMP-SIMPLE : La fonction **fieldsVar** doit rendre exactement une paire composée d'un enregistrement, correspondant au champ invoqué, et du type de références qui contient sa définition. Cette fonction échoue lorsque le champ n'est pas trouvé ou bien qu'elle trouve plus qu'un. Lorsque le champ invoqué est privé, l'accès à celui-ci n'est permis qu'à partir de la classe ou l'interface dans laquelle il est déclaré. La variable $\mathcal{C}$ doit recevoir la valeur vraie lorsque le champ est final. La règle de typage qui permet de typer une expression d'accès à un champ via

le mot clés **super** suit le même principe que la règle précédente. Cependant, le champ est cherché dans la mappe des champs visibles à partir de la super-classe de la classe courante. Cette règle s'assure également que le mot clé **super** est utilisé correctement :

$$\begin{array}{l}\mathsf{licitInvokeSuper} \;:\; \mathit{Environment} \times \mathit{ClassOrIfaceType} \times \mathit{Position} \to \mathsf{bool} \\ \\ \mathsf{licitInvokeSuper}(\Gamma, \mu, \mathcal{P}) \;=\; \\ \quad \mathsf{validClass}(\Gamma, \mu) \;\wedge\; \mu \neq \mathtt{Object} \;\wedge \mathsf{fst}(\mathcal{P}) \neq \text{'a'} \;\wedge \\ \quad \mathsf{if} \quad \mathsf{fst}(\mathcal{P}) = \text{'m'} \\ \quad \mathsf{then} \;\; \mathtt{static} \notin \Gamma.\mathit{classMap}(\mu).\mathit{methods}(\mathsf{snd}(\mathcal{P})).\mathit{modifiers} \\ \quad \mathsf{else} \;\; \mathsf{if} \quad \mathsf{fst}(\mathcal{P}) = \text{'f'} \\ \qquad \mathsf{then} \;\; \mathtt{static} \notin \Gamma.\mathit{classMap}(\mu).\mathit{fields}(\mathsf{snd}(\mathcal{P}).\mathit{modifiers} \\ \qquad \mathsf{else} \;\; \mathsf{true} \\ \qquad \mathsf{endif} \\ \quad \mathsf{endif}\end{array}$$

- ACCES-NOM-CHAMP-SIMPLE : Ces règles permettent de typer une expression d'accès à un champ simple en indiquant tout simplement son nom (un identificateur ou bien une série d'identificateurs séparés par un point). Étant donné que les noms des variables locales cachent les noms des champs, il faut nous assurer qu'aucune variable locale ne possède le même nom que le champ. Comme dans les deux règles précédentes, la fonction **fieldsVar** doit trouver un seul champ. Lorsque celui-ci est un nom simple (*Identifier*), nous n'avons pas besoin de vérifier s'il est privé ou non comme dans les règles précédentes. En effet, un nom simple est cherché dans la mappe des champs visibles à partir de la classe courante qui ne peut pas contenir les champs privés des super-classes. Le prédicat **licitAppear** permet de vérifier qu'il n'est pas possible d'invoquer un champ d'instance n'importe où dans un programme. Il est défini comme suit :

$$\begin{array}{l}\mathsf{licitAppear} \;:\; \mathit{Environment} \times \mathit{ClassOrIfaceType} \times \\ \qquad\qquad (\mathit{ModifierName})\mathsf{set} \times \mathit{Position} \to \mathsf{bool} \\ \\ \mathsf{licitAppear}(\Gamma, \mu, ms, \mathcal{P}) \;=\; \\ \quad \mathsf{if} \quad \mathtt{static} \notin ms \\ \quad \mathsf{then} \;\; (\mathsf{fst}(\mathcal{P}) = \text{'m'} \wedge \\ \qquad\quad \mathtt{static} \notin \Gamma.\mathit{classMap}(\mu).\mathit{methods}(\mathsf{snd}(\mathcal{P})).\mathit{modifiers}\ ) \vee \\ \qquad\quad \mathsf{fst}(\mathcal{P}) = \text{'c'} \vee \\ \qquad\quad (\mathsf{fst}(\mathcal{P}) = \text{'f'} \wedge \mathsf{validClass}(\Gamma, \mu) \wedge \\ \qquad\quad \mathtt{static} \notin \Gamma.\mathit{classMap}(\mu).\mathit{fields}(\mathsf{snd}(\mathcal{P})).\mathit{modifiers}) \\ \quad \mathsf{else} \;\; \mathsf{true} \\ \quad \mathsf{endif}\end{array}$$

Il n'est pas permis, en Java, à un champ d'instance de s'appeler lui-même dans son expression d'initialisation. En outre, il ne peut pas appeler un autre champ d'instance dont la déclaration apparaît textuellement après lui dans la même classe ou interface. De même, il n'est pas permis à un champ de classe de s'appeler lui-même dans son expression d'initialisation. Il ne peut pas, également appeler

un autre champ de classe dont la déclaration apparaît textuellement après lui dans la même classe ou interface. Ceci est vérifié dans la règle par la fonction isDeclared définie comme suit :

$$\begin{array}{l}\mathsf{isDeclared} \;:\; \mathit{Environment} \times \mathit{ClassOrIfaceType} \times \mathit{Identifier} \times \\ \qquad\qquad (\mathit{ModifierName})\mathsf{set} \times \mathit{Position} \rightarrow \mathsf{bool}\end{array}$$

$$\begin{array}{l}\mathsf{isDeclared}(\Gamma, \mu, fn, ms, \mathcal{P}) \;=\\ \quad \mathsf{if} \quad \mathsf{fst}(\mathcal{P}) = \text{'f'}\\ \quad \mathsf{then}\ \mathsf{if} \quad (\mathtt{static} \in ms \leftrightarrow\\ \qquad\qquad \mathtt{static} \in \mathsf{classOrIfaceDecl}(\Gamma, \mu).\mathit{fields}(\mathsf{snd}(\mathcal{P})).\mathit{modifiers})\\ \qquad \mathsf{then}\ \mathsf{textuallyBefore}(\Gamma, \mu, fn, \mathsf{fst}(\mathcal{P}))\\ \qquad \mathsf{else}\ \mathsf{true}\\ \qquad \mathsf{endif}\\ \quad \mathsf{else}\ \mathsf{true}\\ \quad \mathsf{endif}\end{array}$$

Nous supposons que les champs sont triés dans la mappe des champs selon l'ordre de leur déclaration. Ainsi, la fonction **textuallyBefore** peut vérifier si un champ apparaît textuellement avant un autre dans une classe ou une interface. Nous ne donnons pas la définition de cette fonction, mais son type est présenté ci-dessous :

$$\begin{array}{l}\mathsf{textuallyBefore} \;:\; \mathit{Environment} \times \mathit{ClassOrIfaceType} \times \mathit{Identifier} \times \\ \qquad\qquad \mathit{Identifier} \rightarrow \mathsf{bool}\end{array}$$

Enfin, lorsque l'accès à un champ se fait via une classe ou une interface, il doit être statique.

- ACCES-VAR-SIMPE : Cette règle est simple. La recherche de la variable locale dans la mappe $\mathcal{LV}$ doit réussir et la variable booléenne indiquant l'état de l'initialisation de la variable locale doit avoir la valeur vraie. Ceci permet de vérifier que la variable a été initialisée avant son utilisation.
- ACCES-CHAMP-TAB : Une expression d'accès à un tableau contient deux sous-expressions. La première, appelée expression de référence au tableau, ne peut pas être une expression de création d'un tableau. Quant à la deuxième sous-expression apparaissant entre crochets, elle représente l'indexe du tableau. Elle doit être de type `int` ou d'un type primitif qui peut être converti implicitement en `int`. Le type de l'expression d'accès au tableau est de type σ qui est le type des éléments du tableau. Il convient de préciser que les éléments d'un tableau ne sont jamais finaux, même si le tableau lui-même est final.
- ACCES-VAR-TAB : Cette règle ressemble beaucoup à la précédente. Cependant, le type de l'expression de référence au tableau, qui est ici le nom du tableau *Identifier*, est obtenu à partir de la mappe des variables locales. Si le nom du tableau n'est pas trouvé dans la mappe ou que le tableau n'a pas été initialisé, alors la règle échoue.
- COERCITION : Cette règle permet de typer une expression de coercition de types. Le type de cette expression est le type de l'expression qui apparaît entre parenthèses.

- NOM-METHODE : Ces règles utilisent une forme de séquents différente de celle utilisée dans les expressions. En effet, le nom d'une méthode tout seul n'est pas une expression. Ces règles permettent d'évaluer statiquement un nom de méthode que ce soit un nom simple, représenté par un identificateur, ou bien un nom composé de plusieurs identificateurs séparés par un point. Elles rendent le nom de la méthode, le type de la classe, de l'interface ou du tableau à partir duquel il faut chercher la déclaration de la méthode et un caractère représenté par *whichName*. Ce dernier vaut 's' lorsque l'accès à la méthode se fait en donnant tout simplement le nom de la méthode, 'c' lorsque l'accès à la méthode se fait via le nom d'une classe et 'e' lorsque l'accès à la méthode se fait via un nom d'expression composé (le nom d'un champ ou d'une variable locale).
- INVOC-METHODE : La fonction **unAmbiguousMaxSpec** prend en argument l'environnement statique, le type de références à partir duquel il faut chercher la déclaration de la méthode, la signature de cette dernière et la classe ou l'interface courante. Elle fait appel à la fonction **maxSpecMethods** afin de trouver l'ensemble des méthodes qui sont spécifiques au maximum à l'invocation d'une méthode. Lorsque cette fonction retourne plusieurs méthodes, la fonction **unAmbiguousMaxSpec** s'assure qu'elles aient la même signature, sinon l'accès sera considéré ambigu. Cette fonction est définie comme suit :

$$\begin{array}{l}\mathsf{unAmbiguousMaxSpec} \ : \ \mathit{Environment} \times \mathit{ReferenceType} \times \mathit{Sig} \times \\ \qquad\qquad\qquad\qquad \mathit{ClassOrIfaceType} \rightarrow \\ \qquad\qquad\qquad\qquad (\mathit{MethodInfo} \times \mathit{RefernceType})\mathsf{set}\end{array}$$

$$\begin{array}{l}\mathsf{unAmbiguousMaxSpec}(\Gamma, \rho, sig, \mu) \ = \\ \quad \{(m', \rho') \mid ((m', \rho'), \tau'_a) \in \mathsf{maxSpecMethods}(\Gamma, \rho, sig, \mu) \ \wedge \\ \qquad\qquad \forall \{((m'', \rho''), \tau''_a)\} \in \mathsf{maxSpecMethods}(\Gamma, \rho, sig, \mu).\ \tau'_a = \tau''_a\end{array}$$

$$\begin{array}{l}\mathsf{maxSpecMethods} \ : \ \mathit{Environment} \times \mathit{ReferenceType} \times \mathit{Sig} \times \mathit{ClassOrIfaceType} \\ \qquad\qquad\qquad\qquad \rightarrow ((\mathit{MethodInfo} \times \mathit{RefernceType}) \times \mathit{ArgumentType})\mathsf{set}\end{array}$$

$$\begin{array}{l}\mathsf{maxSpecMethods}(\Gamma, \rho, sig, \mu) \ = \\ \quad \{((m', \rho'), \tau'_a) \mid ((m', \rho'), \tau'_a) \in \mathsf{applAccessMethods}(\Gamma, \rho, sig, \mu) \ \wedge \\ \qquad\qquad \forall\ ((m'', \rho''), \tau''_a) \in \mathsf{applAccessMethods}(\Gamma, \rho, sig, \mu). \\ \qquad\qquad\quad \mathsf{if} \quad \Gamma \vdash \rho'' \sqsubseteq_{impl} \rho' \ \wedge \ \Gamma \vdash \tau''_a \sqsubseteq_{impl} \tau'_a \\ \qquad\qquad\quad \mathsf{then} \ \ ((m', \rho'), \tau'_a) = ((m'', \rho''), \tau''_a) \\ \qquad\qquad\quad \mathsf{end\ if}\end{array}$$

$$\begin{array}{l}\mathsf{applAccessMethods} \ : \ \mathit{Environment} \times \mathit{ReferenceType} \times \mathit{Sig} \times \mathit{Sig} \times \\ \qquad\qquad\qquad\qquad \mathit{ClassOrIfaceType} \rightarrow \\ \qquad\qquad\qquad\qquad (\mathit{MethodInfo} \times \mathit{RefernceType}) \times \mathit{ArgumentType})\mathsf{set}\end{array}$$

$$\begin{array}{l}\mathsf{applAccessMethods}(\Gamma, \rho, (mn, \tau_a), \mu) \ = \\ \quad \{(((ms, rt, tcs), \rho'), \tau'_a) \mid (\langle ms, rt, tcs\rangle, \rho') \in \mathsf{methodsVar}(\Gamma, \rho)(mn, \tau'_a) \wedge \\ \qquad\qquad\qquad\qquad \Gamma \vdash \tau_a \sqsubseteq_{impl} \tau'_a\} \wedge \\ \qquad\qquad\qquad\qquad (\ \mathsf{if}\ \mathtt{private} \in ms\ \mathsf{then}\ \rho' = \mu\ )\ \}\end{array}$$

La fonction **maxSpecMethods** ressemble beaucoup à la méthode **max_spec** définie dans le chapitre 2. Cependant, les méthodes applicables trouvées par cette fonction doivent être également accessibles. Ceci revient à vérifier, grâce à la fonction **applAccessMethods**, que si ces méthodes sont privées, alors l'accès ne peut se faire qu'à partir de la classe courante. La fonction **methodDeclarationInfo**, utilisée dans la règle, permet d'extraire à partir d'un ensemble de méthodes leur ensemble de modificateur, leur type de retour et un ensemble d'exceptions qui peut concilier les différents ensembles d'exceptions déclarées dans toutes les méthodes trouvées. En effet, lorsque la fonction **unAmbiguousMaxSpec** retourne un ensemble contenant plusieurs méthodes, ces dernières sont forcément abstraites et elles possèdent le même type de retour. De ce fait, afin d'obtenir l'ensemble des modificateurs de ces méthodes et leur type de retour, il suffit de prendre ces informations de n'importe quelle méthode de l'ensemble de méthodes. Quant à l'ensemble d'exceptions, il faut choisir celui qui comprend les exceptions représentant les sous-types communs de tous les ensembles d'exceptions. Ceci permettra de vérifier que lors de l'appel de la méthode, toutes les exceptions levées par cet appel seront traitées. La fonction **methodDeclInfo** est définie comme suit :

$$\begin{array}{l}
\mathsf{methodDeclInfo} \;:\; (\mathit{MethodInfo} \times \mathit{ReferenceType})\mathsf{set} \rightarrow \mathit{MethodInfo} \\
\\
\mathsf{methodDeclInfo}(\{(\langle ms, rt, tcs\rangle, \rho)\}) \;=\; \langle ms, rt, tcs\rangle \\
\mathsf{methodDeclInfo}(\{(\langle ms, rt, tcs\rangle, \rho)\} \cup s) \;=\; \langle ms, rt, \mathsf{narrower}(tcs, s)\rangle \\
\\
\mathsf{narrower} \;:\; (\mathit{ClassType})\mathsf{set} \times (\mathit{MethodInfo} \times \mathit{ReferenceType})\mathsf{set} \rightarrow \\
\qquad (\mathit{ClassType})\mathsf{set} \\
\\
\mathsf{narrower}(tcs, \emptyset) \;=\; tcs \\
\mathsf{narrower}(tcs_1, \{(\langle ms, rt, tcs_2\rangle, \rho)\} \cup s) \;= \\
\quad \mathsf{let}\ ce_1 = \mathsf{getCheckedException}(tcs_1)\ ; \\
\qquad ce_2 = \mathsf{getCheckedException}(tcs_2) \\
\quad \mathsf{in}\ \mathsf{if}\ \ \forall e \in ce_1.\ \mathsf{subClass}(e, ce_2) \\
\qquad \mathsf{then}\ \ \mathsf{narrower}(ce_1, m) \\
\qquad \mathsf{else}\ \ \mathsf{if}\ \ \forall e \in ce_2.\ \mathsf{subClass}(e, ce_1) \\
\qquad\qquad \mathsf{then}\ \ \mathsf{narrower}(ce_2, s) \\
\qquad\qquad \mathsf{else}\ \ \emptyset \\
\qquad\qquad \mathsf{endif} \\
\qquad \mathsf{endif} \\
\quad \mathsf{endlet}
\end{array}$$

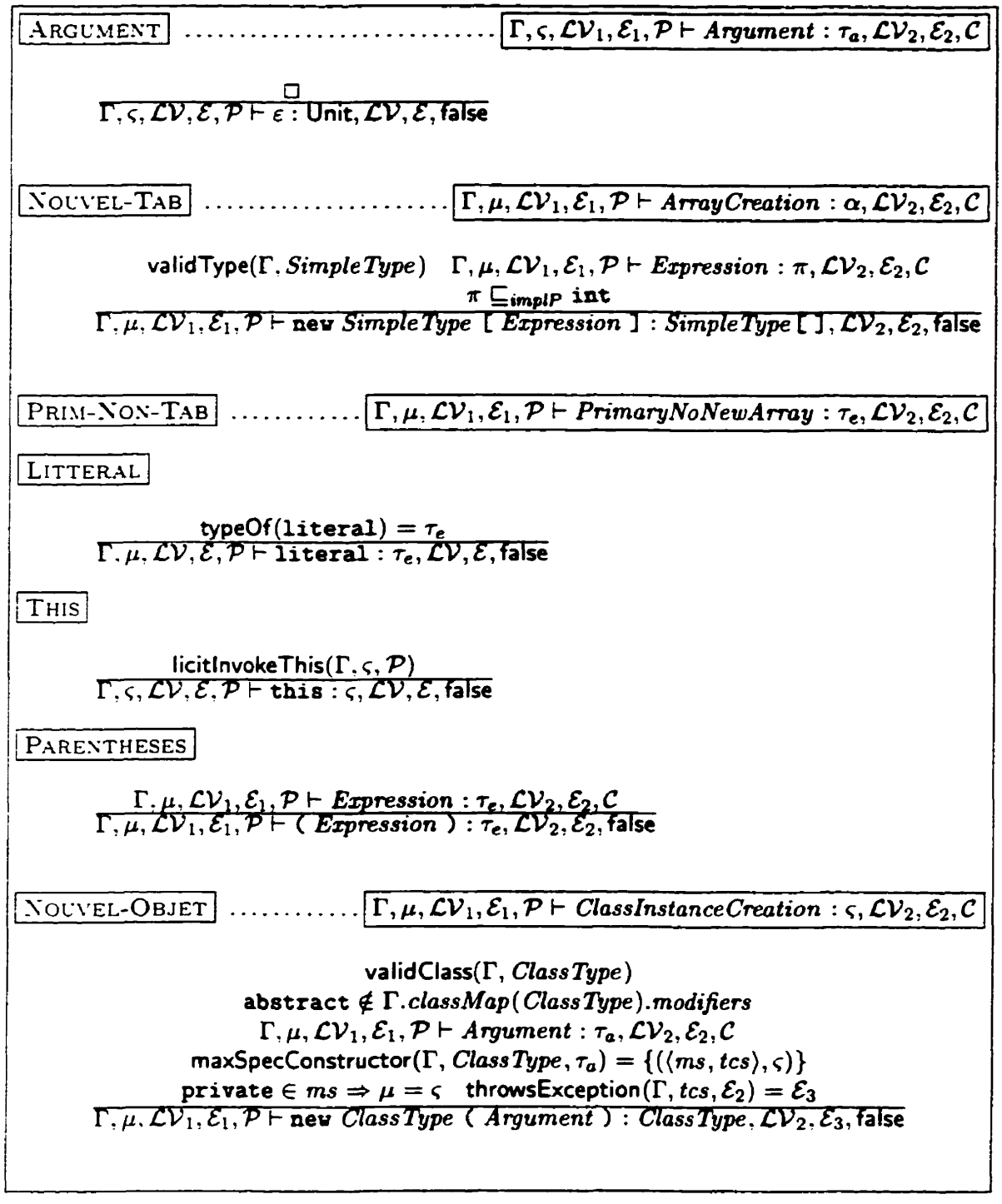
La fonction **throwsException** joue le même rôle que celle utilisée dans l'appel à un constructeur. Lorsque le nom de la méthode est *MethodName*, comme dans l'accès à un champ, il faut vérifier qu'il n'est pas possible d'invoquer une méthode d'instance n'importe où dans le programme. Lorsque l'accès à la méthode ce fait via une classe, il faut vérifier que la méthode est statique. Enfin, lorsque l'accès se fait via **super**, la méthode ne doit pas être abstraite.

- AFFECTATION : Les règles d'affectation d'un champ (les deux premières règles) et d'une variable locale (les deux deuxièmes règles) sont simples. Il suffit de

vérifier que le type de l'expression apparaissant du côté droit de l'expression d'affectation peut être converti implicitement en le type de la variable du côté gauche de l'expression d'affectation. Dans le cas d'une affectation d'un champ, il faut nous assurer que ce champ n'est pas une constante, c'est-à-dire la condition C est fausse. Lorsqu'il s'agit d'une affectation d'une variable locale, il faut mettre à jour la mappe des variables locales afin d'indiquer que cette variable est désormais initialisée.

4.13 Conclusion

Nous avons présenté dans ce chapitre une sémantique statique pour le langage Java. Celle-ci est une description formelle de la spécification de celui-ci qui permettra de mieux raisonner sur le langage et de mieux le comprendre. Nous avons résolu les problèmes de la portée des variables, d'initialisation, de visibilité, de la modélisation des constructeurs ainsi que d'autres. Nous avons par cette sémantique franchi un grand pas vers une théorie sémantique pour Java. Celle-ci doit inclure une sémantique statique, une sémantique opérationnelle et une preuve de correction du typage.



TAB. 4.33: Règles de typage des expressions : partie 1

ACCES-CHAMP-SIMPLE ..... $\boxed{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{SimpleFieldAcess} : \tau, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$

$$\frac{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \textit{Primary} : \rho_1, \mathcal{LV}_2, \mathcal{E}_2 \quad \mathsf{fieldsVar}(\Gamma, \rho_1)(\textit{Identifier}) = \{(\langle \tau, ms \rangle, \rho_2)\} \quad \texttt{private} \in ms \Rightarrow \rho_2 = \mu \quad \mathcal{C} = \texttt{final} \in ms}{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{Primary.Identifier} : \tau, \mathcal{E}_2, \mathcal{LV}_2, \mathcal{C}}$$

$$\frac{\mathsf{licitInvokeSuper}(\Gamma, \varsigma_1, \mathcal{P}) \quad \Gamma.\textit{classMap}(\varsigma_1).\textit{super} = \varsigma_2 \quad \mathsf{fields_c}(\Gamma, \varsigma_2)(\textit{Identifier}) = \{(\langle \tau, ms \rangle, \rho)\} \quad \texttt{private} \in ms \Rightarrow \rho = \varsigma_1 \quad \mathcal{C} = \texttt{final} \in ms}{\Gamma, \varsigma_1, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \texttt{super}.\textit{Identifier} : \tau, \mathcal{E}, \mathcal{LV}, \mathcal{C}}$$

ACCES-NOM-CHAMP-SIMPLE ..... $\boxed{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \textit{FieldName} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}}$

$$\frac{\mathcal{LV}(\textit{Identifier}) = \bot \quad \mathsf{fieldsVar}(\Gamma, \mu)(\textit{Identifier}) = \{(\langle \tau, ms \rangle, \rho)\} \quad \mathcal{C} = \texttt{final} \in ms \quad \mathsf{licitAppear}(\Gamma, \mu, ms, \mathcal{P}) \quad \mathsf{isDeclared}(\Gamma, \mu, \textit{Identifier}, ms, \mathcal{P})}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \textit{Identifier} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}}$$

$$\frac{\mathsf{validClass}(\Gamma, \textit{ClassOrInterfaceType}) \vee \mathsf{validIface}(\Gamma, \textit{ClassOrInterfaceType}) \quad \mathsf{fieldsVar}(\Gamma, \textit{ClassOrInterfaceType})(\textit{Identifier}) = \{(\langle \tau, ms \rangle, \rho)\} \quad \texttt{static} \in ms \quad \texttt{private} \in ms \Rightarrow \rho = \mu \quad \mathcal{C} = \texttt{final} \in ms}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \textit{ClassOrInterfaceType.Identifier} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}}$$

$$\frac{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \textit{ExpressionName} : \rho_1, \mathcal{LV}, \mathcal{E}, \mathcal{C}_1 \quad \mathsf{fieldsVar}(\Gamma, \rho_1)(\textit{Identifier}) = \{(\langle \tau, ms \rangle, \rho_2)\} \quad \texttt{private} \in ms \Rightarrow \rho_2 = \mu \quad \mathcal{C}_2 = \texttt{final} \in ms}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \textit{ExpressionName.Identifier} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}_2}$$

ACCES-VAR-SIMPLE ..... $\boxed{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \textit{LocalVarName} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}}$

$$\frac{\mathcal{LV}(\textit{Identifier}) = (\tau, \mathsf{true})}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \textit{Identifier} : \tau, \mathcal{E}, \mathcal{LV}, \mathsf{false}}$$

ACCES-CHAMP-TAB ..... $\boxed{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{ArrayAccess} : \sigma*, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$

$$\frac{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{PrimaryNoNewArray} : \sigma * [\,], \mathcal{LV}_2, \mathcal{E}_2 \quad \Gamma, \mu, \mathcal{LV}_2.\mathcal{E}_2, \mathcal{P} \vdash \textit{Expression} : \pi, \mathcal{LV}_3, \mathcal{E}_3 \quad \Gamma \vdash \pi \sqsubseteq_{\textit{implP}} \texttt{int}}{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{PrimaryNoNewArray} \; [\; \textit{Expression} \;] : \sigma*, \mathcal{LV}_3, \mathcal{E}_3, \mathsf{false}}$$

TAB. 4.34: Règles de typage des expressions : partie 2

ACCES-VAR-TAB ..... $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ArrayAccess} : \sigma*, \mathcal{LV}_2, \mathcal{E}_2.\mathcal{C}$

$$\frac{\mathsf{fst}(\mathcal{LV}_1(\mathbf{Identifier})) = \sigma * [\,] \quad \mathsf{snd}(\mathcal{LV}_1(\mathit{Identifier})) = \mathsf{true} \quad \Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \pi, \mathcal{LV}_2, \mathcal{E}_2 \quad \Gamma \vdash \pi \sqsubseteq_{implP} \mathbf{int}}{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Identifier}\ [\ \mathit{Expression}\ ] : \sigma*, \mathcal{LV}_2, \mathcal{E}_2, \mathsf{false}}$$

COERCITION ..... $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{CastExpression} : \tau, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}$

$$\frac{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2 \quad \Gamma \vdash \tau_e \sqsubseteq_{cast} \mathit{Type}}{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash (\ \mathit{Type}\ )\ \mathit{Expression} : \mathit{Type}, \mathcal{LV}_2, \mathcal{E}_2, \mathsf{false}}$$

NOM-METHODE ..... $\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{MethodName} : \mathit{Identifier}, \rho, \mathit{wichName}$

$$\frac{\square}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathbf{Identifier} : \mathit{Identifier}, \mu, \text{'s'}}$$

$$\frac{\mathsf{validType}(\Gamma, \mathit{ClassType})}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{ClassType.Identifier} : \mathit{Identifier}, \mathit{ClassType}, \text{'c'}}$$

$$\frac{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{ExpressionName} : \rho, \mathcal{E}, \mathcal{LV}, \mathcal{C}}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{ExpressionName.Identifier} : \mathit{Identifier}, \rho, \text{'e'}}$$

INVOC-METHODE ..... $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{MethodInvocation} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}$

$$\frac{\begin{array}{c}\Gamma, \mu, \mathcal{LV}_1, \mathcal{P} \vdash \mathit{MethodName} : \mathit{Identifier}, \rho, \mathit{wichName} \\ \Gamma, \mu, \mathcal{LV}, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Argument} : \tau_a, \mathcal{LV}_1, \mathcal{E}_2, \mathcal{C} \\ \mathsf{unAmbiguousMaxSpec}(\Gamma, \rho, (\mathbf{Identifier}, \tau_a), \mu) = \mathit{mths} \\ \mathsf{methodDeclInfo}(\mathit{mths}) = \langle \mathit{ms}, \tau_e, \mathit{tcs} \rangle \\ \mathit{wichName} = \text{'s'} \Rightarrow \mathsf{licitAppear}(\Gamma, \mu, \mathit{ms}, \mathcal{P}) \\ \mathit{wichName} = \text{'c'} \Rightarrow \mathbf{static} \in \mathit{ms} \quad \mathsf{throwsException}(\Gamma, \mathit{tcs}, \mathcal{E}_2) = \mathcal{E}_3\end{array}}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}_1, \mathcal{P} \vdash \mathit{MethodName}\ (\ \mathit{Argument}\ ) : \tau_e, \mathcal{LV}_2, \mathcal{E}_3, \mathsf{false}}$$

$$\frac{\begin{array}{c}\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Primary} : \rho, \mathcal{LV}_2, \mathcal{E}_2 \\ \Gamma, \mu, \mathcal{LV}'_1, \mathcal{E}_2, \mathcal{P} \vdash \mathit{Argument} : \tau_a, \mathcal{LV}_3, \mathcal{E}_3, \mathcal{C} \\ \mathsf{unAmbiguousMaxSpec}(\Gamma, \rho, (\mathit{Identifier}, \tau_a)) = \mathit{mths} \\ \mathsf{methodDeclInfo}(\mathit{mths}) = \langle \mathit{ms}, \tau_e, \mathit{tcs} \rangle \quad \mathsf{throwsException}(\Gamma, \mathit{tcs}, \mathcal{E}_3) = \mathcal{E}_4\end{array}}{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Primary.Identifier}\ (\ \mathit{Argument}\ ) : \tau_e, \mathcal{LV}_3, \mathcal{E}_4, \mathsf{false}}$$

TAB. 4.35: Règles de typage des expressions : partie 3

$$\frac{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Argument} : \tau_a,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C} \quad \mathsf{licitInvokeSuper}(\Gamma,\mu,\mathcal{P}) \quad \Gamma.\mathit{classMap}(\mu).\mathit{super} = \varsigma \quad \mathsf{unAmbiguousMaxSpec}(\Gamma,\varsigma,(\mathtt{Identifier},\tau_a)) = \mathit{mths} \quad \mathsf{methodDeclInfo}(\mathit{mths}) = \langle \mathit{ms},\tau_e,\mathit{tcs}\rangle \quad \mathtt{abstract} \notin \mathit{ms} \quad \mathsf{throwsException}(\Gamma,\mathit{tcs},\mathcal{E}_2) = \mathcal{E}_3}{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathtt{super}.\mathit{Identifier}\ (\ \mathit{Argument}\ ) : \tau_e,\mathcal{LV}_2,\mathcal{E}_3,\mathsf{false}}$$

AFFECTATION ..... $\boxed{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{AssignmentExpression} : \tau,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C}}$

$$\frac{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{SimpleFieldAccess} : \tau,\mathcal{LV}_2,\mathcal{E}_2,\mathsf{false} \quad \Gamma,\mu,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{P} \vdash \mathit{Expression} : \tau_e,\mathcal{LV}_3,\mathcal{E}_3,\mathcal{C} \quad \tau_e \sqsubseteq_{\mathit{impl}} \tau}{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{SimpleFieldAccess} = \mathit{Expression} : \tau,\mathcal{LV}_3,\mathcal{E}_3,\mathsf{false}}$$

$$\frac{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{ArrayFieldAccess} : \sigma*,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C}_1 \quad \Gamma,\mu,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{P} \vdash \mathit{Expression} : \tau_e,\mathcal{LV}_3,\mathcal{E}_3,\mathcal{C}_2 \quad \tau_e \sqsubseteq_{\mathit{impl}} \sigma*}{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{ArrayFieldAccess} = \mathit{Expression} : \sigma*,\mathcal{LV}_3,\mathcal{E}_3,\mathsf{false}}$$

$$\frac{\mathsf{fst}(\mathcal{LV}_1(\mathit{Identifier})) = \tau \quad \Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Expression} : \tau_e,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C} \quad \tau_e \sqsubseteq_{\mathit{impl}} \tau}{\begin{array}{c}\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Identifier} = \mathit{Expression} : \\ \tau,\mathcal{LV}_2[\mathit{Identifier} \mapsto (\tau,\mathsf{true})],\mathcal{E}_2,\mathsf{false}\end{array}}$$

$$\frac{\mathsf{fst}(\mathcal{LV}(\mathit{Identifier})) = \varsigma * [\,] \quad \Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Expression}_1 : \pi,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C}_1 \quad \pi \sqsubseteq_{\mathit{impl}} \mathtt{int} \quad \Gamma,\mu,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{P} \vdash \mathit{Expression}_2 : \tau_e,\mathcal{LV}_3,\mathcal{E}_3,\mathcal{C}_2 \quad \tau_e \sqsubseteq_{\mathit{impl}} \sigma*}{\begin{array}{c}\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Identifier}\ [\ \mathit{Expression}_1\ ] = \mathit{Expression}_2 : \\ \sigma*,\mathcal{LV}_2[\mathit{Identifier} \mapsto (\sigma*,\mathsf{true})],\mathcal{E}_3,\mathsf{false}\end{array}}$$

TAB. 4.36: Règles de typage des expressions : partie 4

Chapitre 5

Conclusion

Nous nous sommes proposés dans ce mémoire d'étudier les fondements théoriques du langage Java. Pour ce faire, nous avons examiné l'état de l'art en matière de sémantique formelle pour Java et nous avons présenté un exemple représentatif de la littérature. Cet exemple inclut une sémantique statique, une sémantique opérationnelle et une preuve de correction du typage pour un sous-ensemble de Java nommé BALI. Cette formalisation a été réalisée en utilisant le démonstrateur de théorèmes Isabelle/HOL. Ce travail est très intéressant, puisqu'il a permis de lever quelques difficultés sur la formalisation de Java. En outre, ce travail constitue une base solide pour établir d'autres preuves concernant la sûreté du langage Java. BALI est un langage assez puissant, certes, mais il n'inclut pas plusieurs détails et caractéristiques importantes de Java. Il comporte également quelques erreurs que nous avons essayé de montrer dans le chapitre 3 de ce mémoire.

Le langage Java est très puissant. Il est conçu comme une technologie révolutionnaire dans le monde de la programmation moderne. Cependant, la puissance a un coût : le langage est doté d'une sémantique subtile et sa description est ambiguë et comporte quelques erreurs. En plus, Java est utilisé dans des applications qui requièrent un haut niveau de sécurité et de sûreté de programmation. Il est vrai qu'il est réputé être un langage sûr, mais nous ne pouvons pas confirmer ce propos tant que nous ne disposions pas d'une théorie sémantique qui couvre tout le langage Java. La solution pour mieux comprendre le langage et s'assurer du niveau de sûreté qu'il offre est une théorie sémantique formelle pour celui-ci. Cependant, la description formelle du langage est assez complexe étant donné la taille du langage et la subtilité de sa sémantique. Il existe plusieurs travaux pour formaliser le langage, mais aucun ne le couvre en totalité.

Les principales contributions de ce mémoire sont une syntaxe abstraite et une sémantique statique formelle qui couvrent un ensemble de Java qui n'a jamais été formalisé. Celui-ci inclut toutes les caractéristiques du langage hormis les paquetages. Nous avons décrit, dans cette sémantique, l'algèbre de types, l'environnement statique, les relations de types, la visibilité des entités déclarées, la bonne formation d'un programme et les règles de typage des déclarations, des instructions et des expressions. L'environnement représente la hiérarchie des types d'un programme. Il joue un rôle important pour assurer la bonne formation du programme. Malgré que notre sémantique inclut presque tous les détails et les caractéristiques du langage Java, nous pensons que notre

formalisation est claire et concise comparée à la spécification.

La sémantique statique que nous avons décrite dans ce mémoire constitue, à notre avis, une bonne base pour bâtir des fondements théoriques pour Java qui doit inclure une sémantique statique, une sémantique dynamique et une preuve de correction du typage. En plus, elle sera d'une grande aide pour la compréhension du langage, puisqu'elle représente une description formelle et surtout non ambiguë de la spécification de Java. Cependant, notre sémantique statique est encore incomplète, elle n'inclut pas une importante caractéristique du langage : les paquetages. La sémantique de ces derniers est très subtile et comporte quelques ambiguïtés. L'ajout des paquetages consiste à modifier nos règles de visibilité et à ajouter l'accès **package** par défaut aux membres des classes.

Nous travaillerons désormais sur l'intégration des paquetages et l'élaboration d'une sémantique dynamique pour Java. Nous tenterons aussi de fournir une preuve de cohérence de notre sémantique statique étendue aux paquetages par rapport à la sémantique dynamique que nous proposerons. À ce moment, nous pouvons étudier des techniques de compilation associées à ce langage, élaborer et implanter une série d'analyses statiques d'applications écrites et développer des procédures de vérification de propriétés du langage.

À terme, nous planifions d'utiliser une telle théorie sémantique pour la définition d'analyses statiques robustes qui contribueront à une compilation efficace et sécurisée de Java. En effet, lorsqu'une théorie sémantique complète qui couvre tout le langage Java verra enfin le jour, nous pourrons discuter du vrai potentiel du langage Java et du niveau de sûreté qu'il offre.

Bibliographie

[1] Aczel, P., D. P. Carlisle et N. Mendler, *Two frameworks of theories and their implementation in Isabelle, Logical frameworks*, Huet, G. et G. Plotkin, éditeurs, p. 3–39, Cambridge University Press, 1991.

[2] Campione, M. et K. Walrath, *The Java tutorial : Object-oriented programming for the internet*, Java Series, Addison Wesley, 1998.

[3] David Wragg, S. D. et S. Eisenbach, *Java binary compatibility is almost correct*, rapport technique, Imperial College, 1998.

[4] Debbabi, M. et B. Ktari, *Langages de programmation*, Presses de l'Université LAVAL, septembre 1997, Ce document est un extrait d'une monographie en cours de rédaction.

[5] Doyon, S., *On the Security of Java : The Java Bytecode Verifier*, mémoire de maîtrise, Département d'informatique, Université Laval, avril 1999.

[6] Drossopolou, S. et S. Eisenbach, « Java is type safe –probably », *Proceedings of the 11th European conference on object-oriented programming*, Jyväskylä, Finland, LNCS, 1997.

[7] Drossopoulou, S. et S. Eisenbach, « Is the Java type system sound? », *Fourth international workshop on foundations of object-oriented languages*, p. 22, janvier 1997.

[8] Drossopoulou, S. et S. Eisenbach, « Towards an operational semantics and proof of type soundness for Java », *Formal syntax and semantics of Java*, Alves Foss, J., éditeur, vol. 1523, LNCS, Springer-Verlag, 1999.

[9] Freund, S. N. et J. C. Mitchell, « A type system for object initialization in the Java bytecode language », *Proceedings of the 1998 conference on object-oriented programming systems, languages and applications (OOPSLA'98)*, p. 310–327, Vancouver, ACM, octobre 1998.

[10] Goldberg, A., *A specification of Java loading and bytecode verification*, Kestrel University, 1997, http ://www.kestrel.edu/goldberg.

[11] Guy Steele, B. J. et J. Gosling, *The Java language specification*, The Java Series, Addison-Wesley, 1996.

[12] Guy Steele, B. J. et J. Gosling, *Errors and omissions in the Java language specification 1.0*, `http://www.dina.kvl.dk/~jsr/java-jls-errors.html`, 1998.

[13] Isabelle Attali, D. C. et M. Russo, « A formal executable semantics for Java », *Proceedings of formal underpinnings of Java*, octobre 1998.

[14] Microsystems, S., *Clarifications and amendments to the Java language specification*, **http://java.sun.com/docs/books/jls/clarify.html**, 1998.

[15] Nipkow, T., « Equational reasoning in Isabelle », *Science of Computer Programming*, 12, p. 123–149, 1989.

[16] Nipkow, T., « More Church-Rosser proofs (in Isabelle/HOL) », *Proceedings of the 13th international conference on automated deduction (cade-13)*, McRobbie, M. et J. Slaney, éditeurs, p. 733–747, New Brunswick, New Jersey, Springer-Verlag, juillet 1996.

[17] Nipkow, T. et D. von Oheimb, « Java_{light} is type-safe definitely », *Proceeedings of the 25th ACM symp. principles of programming languages*, p. 61–170, ACM Press, New York, 1998.

[18] Nipkow, T. et D. von Oheimb, *Machine-checking the Java specification : Proving type-safety. Formal syntax and semantics of Java*, Alves Foss, J., éditeur, vol. 1523, LNCS, Springer-Verlag, 1998.

[19] Paulson, L. C., *Isabelle : The next 700 theorem provers*, Academic Press, 1990.

[20] Qian, Z., *A formal specification of a large subset of Java virtual machine instructions*, rapport technique, Universität Bremen, FB3 Informatik, D-28334 Bremen, Germany, novembre 1997.

[21] Rémy, D. et J. Vouillon, « Objective ML : A simple object-oriented extension of ML », *Proceedings of the 24th ACM conference on principles of programming languages*, p. 40–53, Paris, France, janvier 1997.

[22] Rémy, D. et J. Vouillon, « Objective ML : An effective object-oriented extension to ML », *Theory and practice of object systems*, Paris, France, 1998.

[23] Saraswat, V., *The Java bytecode verification problem*, AT&T Research, 1997, http ://www.research.att.com/ vj/bcv.html.

[24] Saraswat, V., *Java is not type-safe*, rapport technique, AT&T Research, 180 Park Avenue, Florham Park NJ 07932, 1997.

[25] Slotosch, O., « Higher order quotients and their implementation in Isabelle HOL », *Proceedings of the international conference on theorem proving in higher order logics*, Gunter, E., éditeur, Murray Hill, New Jersey, août 1997.

[26] Stata, R. et M. Abadi, « A type system for Java bytecode subroutines », *Proceeedings of the 25th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, San Diego, California, USA, ACM Press, janvier 1998, **http ://www.research.digital.com/SRC/personal/Martin_Abadi/Papers/**.

[27] Syme, D., *A prototype declarative proof system for higher order logic*, rapport technique, Cambridge University, mars 1997.

[28] Syme, D., *Proving Java type sound*, rapport technique, Cambridge University Computer Laboratory, juin 1997.

[29] Syme, D., *Proving Java type sound, Formal syntax and semantics of Java*, Alves Foss, J., éditeur, vol. 1523, LNCS, Springer-Verlag, 1998.