
Rapport de projet de fin d'études

Mise en œuvre et évaluation d'analyses
de *bytecode* Java spécifiées avec DATALOG

Étienne André

5^e année d'informatique à l'Institut National des Sciences Appliquées de Rennes

Version du 13 juillet 2006

Encadreurs Frédéric Besson
Thomas Jensen

Rapporteur Édouard Monnier

Équipe Lande
Institut de Recherche en Informatique et Systèmes Aléatoires

Résumé

L'analyse statique permet d'analyser des programmes en ayant la seule connaissance du code source, c'est-à-dire sans réaliser d'exécutions, afin de déduire ou de vérifier des propriétés. L'analyse de classes permet en particulier de déduire la hiérarchie de classes, le graphe d'appels de méthodes, ou encore le diagramme d'objets d'un ensemble de classes analysées. Des recherches récentes ont porté sur l'analyse de classes modulaire, afin de pouvoir travailler sur des programmes incomplets ou soumis à modification.

Frédéric Besson et Thomas Jensen ont décrit une théorie conduisant, à partir d'un langage orienté objet fictif et basique, à la génération de contraintes DATALOG (langage pouvant être défini comme un sous-ensemble de PROLOG). Une fois ces contraintes résolues, de nombreuses informations peuvent être déduites de l'analyse, dont les objets créés par le programme, leurs relations avec les autres objets, les appels de méthodes, etc.

Au cours de ce projet de fin d'études, réalisé au sein de l'équipe Lande de l'Irisa, nous avons adapté cette théorie à l'intégralité du *bytecode* Java, et mis en œuvre un prototype qui génère puis résout les contraintes relatives aux classes passées en paramètres. Ce logiciel permet notamment de réaliser des analyses modulaires et génère plusieurs graphes de visualisation des résultats de l'analyse.

Abstract

Static analysis is an analysis of computer programs that is performed without actually executing these programs, but only knowing the source code, in order to deduce or verify properties. In particular, class analysis is a technique allowing to deduce the class hierarchy of a set of classes, the call graph of methods, or the objects diagram of a set of analyzed classes. Recent researches were done on modular class analysis, in order to analyze incomplete or modified programs.

Frédéric Besson and Thomas Jensen describe a theory generating, for a basic and fictive object-oriented language, DATALOG constraints (DATALOG can be defined as a subset of PROLOG). One these constraints are resolved, various information can be deduced from the analysis, for example objects created in the program, their relationship with other objects, method calls, etc.

The subject of this final project in the Lande team of Irisa was to adapt this existing theory to Java *bytecode*, and to implement a prototype allowing to generate and to solve constraints relative to the analyzed classes. This prototype allows modular analysis and generates various graphics.

Remerciements

Plusieurs personnes m'ont apporté leur soutien au cours de ce projet de fin d'études.

Je tiens à remercier tout d'abord mes deux maîtres de stage Frédéric Besson et Thomas Jensen pour leur écoute et leur soutien au cours de ce stage, Tiphaine Turpin pour ses remarques sur la rédaction de ce rapport, ainsi que l'ensemble de l'équipe Lande pour son accueil chaleureux.

Je remercie également Édouard Monnier, mon responsable de stage à l'Insa, qui m'a en outre converti aux joies du langage ocaml.

De plus, je souhaiterais avoir une pensée pleine de reconnaissance pour l'ensemble de mes enseignants de l'Insa de Rennes et de la *Technische Universität* de Dresde (Allemagne) pour la qualité de leur enseignement.

Par ailleurs, je tiens à remercier Adrien Delaye pour m'avoir suggéré le nom du logiciel développé, à partir d'un jeu de mot pour le moins bancal.

Enfin, un petit remerciement au personnel de la cafétéria et au babyfoot de l'Irisa pour leur précieuse collaboration...

Sommaire

1	Environnement du projet	11
1.1	Sujet du projet de fin d'études	11
1.2	L'équipe Lande de l'Irisa	11
1.2.1	Présentation de l'Irisa	11
1.2.2	Présentation de l'équipe Lande	12
1.2.3	Axes de recherche de l'équipe Lande	12
1.3	Durée du projet de fin d'études	13
2	Description du projet	15
2.1	Contexte	15
2.1.1	Analyse statique	15
2.1.2	Interprétation abstraite	15
2.1.3	Analyse de classes	16
2.1.4	Analyse de classes modulaire	18
2.2	Théorie existante pour l'analyse d'un langage fictif	18
2.2.1	DATALOG	18
2.2.2	Un langage orienté objet minimaliste	19
2.2.3	Génération des contraintes DATALOG	19
2.2.4	Généricité de l'analyse	20
2.3	Détermination d'une théorie pour l'analyse du <i>bytecode</i>	21
2.3.1	Le <i>bytecode</i> Java	21
2.3.2	Détermination d'une théorie pour la génération de contraintes	22
2.4	Choix d'implémentation	25
2.4.1	Génération des contraintes	25
2.4.2	Résolution des contraintes	26
2.4.3	Génération de graphes	26
2.5	Résultats	27
2.5.1	Hiérarchie de classes	29
2.5.2	Diagramme d'appels	29
2.5.3	Diagramme d'objets	31
2.5.4	Modularité	33
2.6	Perspectives	36
2.6.1	Analyseur syntaxique JavaLib	36
2.6.2	Ligne de commande	36
2.6.3	Modularité	36
2.6.4	Analyse des bibliothèques Java	36
2.6.5	Initialisation des champs des objets	37

2.6.6	Parallélisme	37
2.6.7	Solveur de contraintes DATALOG	37
3	Conclusion	39
A	Génération des contraintes	43
A.1	Contraintes relatives aux instructions <i>bytecode</i>	43
A.1.1	Remarques	43
A.1.2	Opérations sur la pile et les variables locales	43
A.1.3	Création d'objet	44
A.1.4	Accès aux champs	44
A.1.5	Définitions, appels et retours de méthodes	45
A.1.6	Instructions relatives aux tableaux	48
A.1.7	Exceptions	49
A.1.8	Appel du programme	50
A.2	Contraintes relatives à l'analyse	51
A.3	Contraintes d'ordre général	52
B	Résultats intéressants	53
B.1	Chaîne d'éléments	53
B.1.1	Code source	53
B.1.2	Graphe d'appels	54
B.1.3	Diagrammes d'objets	54
B.2	Exceptions	54
B.2.1	Exemple de graphe d'appels	54
B.3	Comparaison entre analyses globale et modulaire	55
B.3.1	Code source	55
B.3.2	Hierarchie de classes	59
B.3.3	Diagramme d'appels	59
B.3.4	Diagramme d'objets	59

Chapitre 1

Environnement du projet

1.1 Sujet du projet de fin d'études

Le sujet de mon projet de fin d'études à l'Insa concerne la mise en œuvre et l'évaluation d'analyses modulaires de *bytecode* Java spécifiées avec DATALOG.

Ce stage est effectué sous la responsabilité de Frédéric Besson et Thomas Jensen au sein de l'équipe Lande de l'Irisa.

1.2 L'équipe Lande de l'Irisa

1.2.1 Présentation de l'Irisa

L'Institut de Recherche en Informatique et Systèmes Aléatoires (Irisa), est un pôle de recherche public à Rennes regroupant environ 530 personnes dont 205 chercheurs ou enseignants chercheurs, 175 chercheurs en thèse, 90 ingénieurs, techniciens, administratifs et de nombreux collaborateurs contractuels ou invités internationaux pour des séjours de plus courte durée. L'Inria, le CNRS, l'Université de Rennes 1 et l'Insa de Rennes sont les partenaires de cette unité mixte de recherche.

Les recherches se développent au sein d'une trentaine d'équipes de recherche et autour de grands thèmes scientifiques tels que :

- les réseaux et systèmes informatiques,
- les langages de programmation et la conception logicielle,
- le traitement de données (signal, texte, son, images) et des connaissances,
- la réalité virtuelle,
- la robotique,
- la modélisation du vivant (en imagerie médicale, en bio-informatique),
- la modélisation, la simulation et l'optimisation de systèmes complexes.

Ces thèmes génériques se déclinent dans de nombreux domaines d'application, sources de collaborations avec d'autres acteurs tant du monde académique qu'industriel. Parmi ces applications se trouvent notamment les télécommunications, le multimédia et les technologies avancées pour la santé.

Les missions essentielles de l'Irisa sont d'une part de conduire des recherches dont l'excellence scientifique est attestée par des publications et des échanges internationaux, et d'autre part d'assurer la diffusion de ces connaissances et de ce savoir-faire, en particulier auprès de partenaires industriels, des PME-PMI régionales, regroupées dans le club Irisatech, aux structures

R&D de grands groupes industriels mondiaux. La dimension européenne, par la participation des équipes de recherche de l'Irisa aux programmes européens est bien sûr fortement présente. En complément, le laboratoire s'investit dans la démarche d'incubation d'entreprises lancée tant par Inria-Transfert sur le plan national que par l'incubateur Emergys en Bretagne.

L'Irisa, en relation étroite avec l'Institut de Formation Supérieure en Informatique et Communication (Ifsic), l'école doctorale Matisse de l'Université de Rennes 1, l'Insa de Rennes et l'antenne de Bretagne de l'ENS-Cachan, participe activement à la formation par et pour la recherche. L'accueil de stagiaires et l'encadrement de thèses sont des éléments essentiels de dynamisme et de progression des recherches entreprises à l'Irisa. Enfin, l'accueil de visiteurs en séjours post-doctoraux ou sabbatiques internationaux vient renforcer ce potentiel.

1.2.2 Présentation de l'équipe Lande

Lande est l'une des équipes de l'Irisa. Le thème de recherche central du projet Lande est la conception d'outils d'aide au développement et à la validation de logiciels (le nom de l'équipe provient de *Logiciel : ANalyse et DEveloppement*). Il s'agit d'un projet commun avec le CNRS, l'Université de Rennes 1 et l'Insa de Rennes.

Les différents axes de recherche concernent plusieurs domaines, dont l'analyse, la sécurité et le test de logiciels, ainsi que les systèmes d'information logiques.

1.2.3 Axes de recherche de l'équipe Lande

Analyse statique

La validation d'un logiciel utilise des méthodes d'analyse et de test de programmes. L'équipe Lande s'intéresse à différents aspects de l'analyse statique de programmes, aussi bien sur le plan des fondements (spécification d'analyses à partir de règles d'inférence) que des applications (analyse de flot de données et de contrôle dans des programmes Java et Java Card, analyse de protocoles cryptographiques) et à la mise en œuvre d'analyses statiques par des techniques de résolution itérative de systèmes d'équations et de réécriture d'automates d'arbres.

Test

Pour faciliter l'analyse dynamique de programmes, Lande développe un outil d'analyse de traces d'exécution permettant à l'utilisateur d'exprimer des requêtes dans un langage de programmation logique. Ces requêtes peuvent être traitées à la volée, ce qui permet d'analyser des traces de grande taille. L'outil peut être utilisé pour le débogage de programmes séquentiels et son application au problème de la détection d'intrusion est maintenant à l'étude.

Sécurité logicielle

La sécurité logicielle constitue un domaine d'application privilégié pour le projet. L'équipe élabore un cadre pour la définition de propriétés de sécurité et une technique pour leur vérification automatique. Cette technique intègre des techniques d'analyses statiques et de vérification de modèle (*model checking*). Ce cadre a été appliqué à la formalisation et la vérification de politiques de sécurité d'applications programmées avec la nouvelle architecture de sécurité de Java 2 et à la vérification de propriétés de sécurité des cartes à puce multi-applicatives programmées avec le langage Java Card.

Systèmes d'information logiques

Pour faciliter la navigation et l'organisation de logiciels de taille importante, l'équipe cherche à définir un cadre logique pour les systèmes d'information qui décrit uniformément la navigation, l'interrogation et l'analyse des données. Ce cadre est générique par rapport à la logique utilisée pour naviguer et interroger ; en particulier il peut être appliqué à plusieurs types de logiques de programme, comme les types ou des propriétés statiques. Ces travaux se basent sur une extension de la théorie de l'analyse de concepts.

À noter que cette sous-équipe deviendra une équipe à part entière — au nom de Géolis — dès la rentrée 2006.

1.3 Durée du projet de fin d'études

Ce projet de fin d'études porte sur la période du lundi 13 mars 2006 au lundi 17 juillet de la même année, impliquant une durée totale de 18 semaines.

Chapitre 2

Description du projet

Mon projet de fin d'études réalisé au printemps 2006 concerne la génération et l'implémentation d'une théorie permettant d'analyser statiquement du *bytecode* Java de manière modulaire, c'est-à-dire en ayant la possibilité de traiter les différentes classes de manière séparée, et ce à l'aide de contraintes DATALOG. Le logiciel résultant, répondant au nom de Toutânkhamel, permet de réaliser cette analyse et génère différents graphes, notamment d'appels de méthodes et de suivi des objets au cours du déroulement du programme Java.

2.1 Contexte

2.1.1 Analyse statique

Ce projet s'inscrit dans le contexte de l'analyse statique de programmes. À l'inverse de l'analyse dynamique, ou test, qui exécute un programme sur certaines entrées jugées représentatives pour en vérifier les résultats, l'analyse statique est une famille de méthodes formelles permettant de dériver des résultats sur le déroulement des programmes *sans* les exécuter — et donc uniquement en ayant connaissance du code source.

Une application possible de l'analyse statique est l'aide automatisée au débogage, et en particulier la recherche d'erreurs à l'exécution.

Il existe deux grandes familles d'analyses statiques formelles de programmes :

- la vérification de modèle (*model checking*), qui considère des systèmes à états finis, ou qui peuvent être réduits vers des systèmes à états finis par abstraction,
- l'analyse statique par interprétation abstraite, qui approxime le comportement du système.

Ce projet s'inscrit dans le cadre de l'interprétation abstraite.

2.1.2 Interprétation abstraite

L'interprétation abstraite est une théorie d'approximation de la sémantique de programmes informatiques basée sur les fonctions monotones pour ensembles ordonnés, en particulier les treillis (*lattice*). Cette discipline a été formalisée à la fin des années 1970 par le professeur Patrick Cousot et le docteur Radhia Cousot [CC77].

Cette discipline consiste à abstraire la valeur des variables utilisées par un programme, afin de simplifier la vérification de certaines propriétés — parfois au détriment de la précision de l'analyse. Par exemple, on pourra choisir de représenter l'état d'un programme manipulant des

variables entières en omettant les valeurs proprement dites et en ne gardant que leurs signes (+, – ou 0). On note alors :

$$\begin{aligned} \llbracket -5 \rrbracket &= - \\ \llbracket 0 \rrbracket &= 0 \\ \llbracket 1 \rrbracket &= + \\ \llbracket 2006 \rrbracket &= + \end{aligned}$$

Pour certaines opérations élémentaires comme la multiplication, une telle abstraction ne perd pas de précision : pour connaître le signe d'un produit, il est suffisant de connaître le signe de ses opérandes.

$$\llbracket a * b \rrbracket = \begin{cases} 0 & \text{si } \llbracket a \rrbracket = 0 \text{ ou } \llbracket b \rrbracket = 0, \\ - & \text{si } \llbracket a \rrbracket \neq 0, \llbracket b \rrbracket \neq 0 \text{ et } \llbracket a \rrbracket \neq \llbracket b \rrbracket, \\ + & \text{sinon} \end{cases}$$

Pour d'autres opérations en revanche, l'abstraction perdra de la précision : ainsi, il est impossible de connaître le signe d'une somme dont les opérandes sont de signe contraire.

$$\llbracket a + b \rrbracket = \begin{cases} 0 & \text{si } \llbracket a \rrbracket = 0 \text{ et } \llbracket b \rrbracket = 0, \\ - & \text{si } \llbracket a \rrbracket = - \text{ et } \llbracket b \rrbracket = -, \\ + & \text{si } \llbracket a \rrbracket = + \text{ et } \llbracket b \rrbracket = +, \\ ? & \text{sinon} \end{cases}$$

2.1.3 Analyse de classes

L'un des aspects les plus importants des analyses de langages orientés objet concerne l'analyse de classes. Les applications de cette technique sont nombreuses : on pourra citer la vérification de types, garantissant qu'une méthode est uniquement invoquée sur des objets implémentant cette méthode, ou encore la construction de graphes d'appels de méthodes impliquant une *dévirtualisation* des appels de méthodes ou, en d'autres termes, la résolution de nom des méthodes virtuelles, permettant de connaître quelle méthode d'une hiérarchie de classes est effectivement appelée. En effet, la classe de la méthode véritablement exécutée lors d'un appel à une méthode sur un objet sera la première implémentant effectivement cette méthode dans la hiérarchie des classes mères de l'objet.

L'ensemble de ces opérations sont notamment à la base de vérifications ou optimisations du programme ainsi analysé.

Il existe plusieurs possibilités pour raffiner une analyse de classes. Barbara G. Ryder décrit de manière formelle les différentes dimensions que peut revêtir une analyse de classes [Ryd03b].

Définition 1 Une analyse de classes est dite sensible au flot (*en anglais* : flow sensitive) si l'ordre des instructions des méthodes est pris en compte dans l'analyse.

Les analyses considérées au cours de ce projet de fin d'études sont toutes sensibles au flot.

Définition 2 Une analyse de classes est dite sensible au contexte (*en anglais* : contexte sensitive) si les différents contextes d'appels de méthodes sont différenciés.

En d'autres termes, si une méthode est appelée plusieurs fois, on peut choisir ou non de différencier ces appels, ou de les unir sous la forme d'un appel unique dans l'analyse. Le domaine abstrait des contextes d'appel est symbolisé par *Contexte*. La précision de l'analyse peut être améliorée en gardant en mémoire la liste des points de programme auxquels cette méthode a

été appelée. Par exemple, si une méthode f appelle une méthode h , et qu'une autre méthode g appelant cette même méthode h , ces deux appels peuvent être différenciés si l'on considère le point d'appel (méthode f dans un cas, méthode g dans l'autre).

Une autre façon de séparer les appels de méthodes est de les distinguer en fonction de la classe de leurs arguments. Ainsi, une méthode attendant un objet de la classe `animal` peut être appelée avec un objet de la classe `mammifere` ou un objet de la classe `oiseau`. Si l'on distingue le contexte d'appel par la classe des arguments, ces deux appels seront différenciés.

Par abus de langage, on appellera par la suite contexte ce qu'il conviendrait de nommer contexte d'appel de méthode.

Enfin, on peut souhaiter avoir différents niveaux de modélisation du tas des objets. Basiquement, on peut séparer les analyses en deux groupes relativement à la dimension des objets : on peut abstraire simplement un objet par sa classe (le domaine abstrait des objets est donc défini par $Objet = Classe$), ou prendre en compte leur *contexte de création* : par exemple, les objets peuvent être différenciés par leur point de programme de création, auquel cas on obtient $Objet = Classe * PP$, où $PP = Classe * Methode * PC$ est l'ensemble des points de programme. Il va sans dire que, si les analyses du premier groupe sont plus efficaces, les secondes sont plus précises.

Les paragraphes suivants décrivent brièvement quatre différentes analyses de classes, considérées dans ce projet de fin d'études.

Analyse 0-CFA

L'analyse 0-CFA¹ est une analyse au contexte dégénéré dans laquelle les objets sont abstraits par leur classe, et où le contexte, unique, est représenté par la constante ctx [Shi91, GC01]. Ainsi, nous avons $Objet = Classe$ et $Contexte = \{ctx\}$. Cette analyse est donc insensible au contexte d'appel.

Analyse 1/2-CFA

Bien que voisine de l'analyse 0-CFA, l'analyse 1/2-CFA² est, elle, sensible au contexte. Les objets sont toujours abstraits par leur classe, mais les appels de méthodes sont différenciés par l'objet appelant. Nous avons ainsi $Objet = Classe$ et $Contexte = Objet$.

Analyse k -l-CFA

Le principe de l'analyse k -l-CFA est de conserver une chaîne d'appels de longueur k et une chaîne de créations d'objets de longueur l . Ainsi, le contexte d'appel est un n -uplet des k derniers appels de méthode ayant abouti à cet appel. De façon similaire, les objets conservent une chaîne des l derniers objets ayant abouti à leur création : ainsi, un objet o_1 contient désormais des informations sur l'objet o_2 l'ayant créé, et ainsi de suite jusqu'à l'objet o_l ayant créé l'objet o_{l-1} . Nous avons par conséquent $Objet = Classe * PP^l$ et $Contexte = PP^k$.

Algorithme du produit cartésien

Introduit par Agesen, l'algorithme du produit cartésien (en anglais *Cartesian Product Algorithm*) conserve un contexte d'appel créé à partir des arguments de l'appel [Age95]. Les appels à la même méthode sont donc différenciés à partir du moment où les arguments sont différents. L'ensemble des contextes d'appel d'une méthode de cardinalité n est donc $Contexte_n = Objet^n$.

¹CFA représente les initiales de *Control Flow Analysis* (analyse de contrôle de flot)

²Le nom de cette analyse se prononce « un demi CFA », car à mi-chemin entre les analyses 0-CFA et 1-CFA.

Ainsi, dans cette analyse, la précision de l'analyse globale dépend de l'abstraction réalisée sur les objets. Nous prendrons le parti d'abstraire les objets à partir de leur point de création : $Objet = Classe * PP$.

2.1.4 Analyse de classes modulaire

Les analyses de classes existantes considèrent des programmes complets, et nécessitent donc que l'ensemble des classes soit présent lors de l'analyse. Toutefois, il peut paraître fastidieux de réanalyser l'ensemble des classes du programme si une seule classe est modifiée. En outre, le traitement de langages autorisant l'inclusion de classes dynamique (*dynamic class loading*) implique que l'ensemble du code n'est pas nécessairement présent au moment de l'analyse.

C'est pourquoi l'analyse de classes modulaire a été l'objet d'études récentes. Cousot et Cousot ont notamment examiné les différentes approches de l'analyse modulaire de programme, et les ont resituées dans le cadre de l'interprétation abstraite [CC02]. Le fondement de leur analyse est de considérer une analyse modulaire de programme comme le calcul d'approximations d'un point fixe. Thomas Jensen et Frédéric Besson ont par la suite démontré comment ce calcul de point fixe peut être appliqué au cas d'une analyse de classes modulaire exprimée à l'aide de clauses DATALOG [BJ03]. Dans ce cas, le résultat d'une analyse de classes est défini comme la plus petite solution d'un ensemble de clauses généré automatiquement à partir du programme.

2.2 Théorie existante pour l'analyse d'un langage fictif

Thomas Jensen et Frédéric Besson ont jeté les fondements d'une théorie permettant de générer des clauses DATALOG à partir d'un langage orienté objet basique et fictif, et ce afin de réaliser une analyse de classes modulaire [BJ03].

2.2.1 DATALOG

Syntaxiquement, DATALOG peut être défini comme un sous-ensemble de PROLOG dont les noms d'atomes ne seraient que des constantes ; la majeure partie des caractéristiques de DATALOG hérite ainsi de PROLOG. DATALOG peut également être décrit comme un langage de requêtes relationnelles avec récursivité.

La signification d'un programme DATALOG est le plus petit ensemble d'atomes satisfaisant les clauses de ce programme. La différence fondamentale avec PROLOG réside dans le fait que le plus petit modèle de Herbrand est calculable.

Rappelons très brièvement quelques notions de logique.

Définition 3 Soit Π un ensemble (fini) de symboles de prédicats et V , respectivement C , un ensemble de variables, respectivement de symboles de constantes.

Un atome est un terme $p(x_1, \dots, x_n)$ d'arité n , où $p \in \Pi$ est un symbole de prédicat et chaque x_i ($i \in [1, \dots, n]$) est soit une variable soit une constante ($x_i \in V \uplus C$).

Définition 4 Une clause est une formule $H \leftarrow B$ où H , appelée la tête, est un atome tandis que B , appelé le corps, est un ensemble fini d'atomes.

Définition 5 Un programme P est un ensemble de clauses.

Pour des raisons de lisibilité et de cohérence avec la syntaxe des interpréteurs, nous écrivons les clauses dans un style proche de celui de PROLOG. De plus, on utilisera par convention des symboles commençant par une majuscule pour les variables, et par une minuscule pour les

$$\begin{aligned} & p(a, b). \\ p(X, Y) & \quad :- \quad q(X, Z), p(Z, Y). \end{aligned}$$

FIG. 2.1 – Exemples de clauses dans un style proche de PROLOG

$$\begin{aligned} P & ::= \{C_1, \dots, C_n\} \\ C & ::= \text{class } c\{M_1, \dots, M_n\} \mid \\ & \quad \text{class } c \text{ extends } c'\{M_1, \dots, M_n\} \\ M & ::= m(x_1, \dots, x_n) \text{ IL} \\ \text{IL} & ::= [I_1, \dots, I_n] \\ I & ::= x := \text{new } c \mid x.f.d := y \mid x := y.f.d \mid \\ & \quad x := x_0.f(x_1, \dots, x_n) \mid \text{ret } x \end{aligned}$$

FIG. 2.2 – Un langage orienté objet minimaliste

constantes. La figure 2.1 présente des exemples de clauses écrites dans un style voisin de celui de PROLOG.

2.2.2 Un langage orienté objet minimaliste

La théorie existante de génération de contraintes s'applique à un langage orienté objet basique et fictif. Ce langage accepte les programmes composés de plusieurs classes, héritant éventuellement les unes des autres; ces classes sont elles-mêmes composées de plusieurs méthodes. Les instructions autorisées dans les méthodes sont les créations d'objets (`new c`), les accès en lecture ou écriture aux champs des objets, ainsi que les appels de méthodes et les retours d'objets en fin de méthode.

Bien que ce langage présente déjà des caractéristiques intéressantes en matière de langage orienté objet, on notera qu'il ne gère ni les structures de contrôle (instructions similaires au `if`), ni les types prédéfinis (entier, booléens, tableaux, etc.), ni les exceptions.

La grammaire décrivant ce langage minimaliste est décrite sur la figure 2.2. L'instruction `x.f.d := y` assigne la valeur de la variable `y` au champ `fd` de l'objet référencé par `x`, tandis que l'instruction `x := y.f.d` transfère le contenu du champ `fd` de `y` vers la variable `x`. L'instruction `x := x_0.f(x_1, \dots, x_n)` appelle la méthode `f` sur l'objet référencé par `x_0` avec les arguments `x_1, \dots, x_n` et place le résultat dans `x`. Enfin, `ret x` termine l'exécution d'une méthode appelée en retournant la valeur de `x`. Dans un corps de méthode, en conformité avec les conventions des langages orientés objet, l'objet actif est référencé par `self`.

La relation d'héritage entre les classes d'un programme est acyclique. Puisque ce langage ne permet pas l'héritage multiple, la hiérarchie de classes est une forêt (d'arbres). La résolution de méthode virtuelle est définie par une méthode `lk` (pour *lookup*) qui retourne, pour une classe `c` et une méthode `m`, la classe `c'` implémentant la méthode `m` pour la classe `c`. Afin de prendre l'héritage en compte, cet algorithme parcourt la hiérarchie de classes depuis la classe `c` et rend éventuellement la première classe `c'` définissant une méthode `m` dans les conditions recherchées.

2.2.3 Génération des contraintes DATALOG

À partir de ce langage fictif orienté objet, a été décrit un algorithme de génération automatique des contraintes en fonction des instructions rencontrées dans le programme. La résolution des contraintes ainsi générées et de quelques contraintes d'ordre général — notamment la définition de la méthode `lk` —, permet le calcul du plus petit point fixe et la récupération des informations

$$\llbracket \text{ret } x \rrbracket_{c,m,pc} = \text{m.ret}(0, \text{Ctx}) :- \\ \text{c.m.x}(0, \text{Ctx}).$$

FIG. 2.3 – Contrainte générée pour le retour d'un objet

$$\llbracket x := \text{new } c' \rrbracket_{c,m,pc} = \text{c.m.x}(0, \text{Ctx}) :- \\ \text{c.m.self}(0', \text{Ctx}), \\ \text{objCtx}(c, m, pc, c', \text{Ctx}, 0', 0).$$

FIG. 2.4 – Contrainte générée pour la création d'un objet

recherchées.

Il n'est pas question ici de présenter l'algorithme dans son intégralité. Le lecteur intéressé par une étude plus poussée que le cadre de ce rapport se reportera à [BJ03]. Toutefois, dans un souci de clarté et de meilleure situation du contexte, deux exemples vont être présentés.

Tout d'abord, la figure 2.3 décrit la contrainte générée dans le cas d'une instruction de retour d'un objet x . Les indices c,m,pc indiquent l'environnement dans lequel l'instruction a été rencontrée, en l'occurrence la l'instruction pc de la méthode m de la classe c . Cette contrainte peut être ainsi verbalement décrite : l'objet 0 est retourné dans le contexte Ctx par la méthode m , sous réserve que x référençait bien 0 dans ce même contexte Ctx dans la méthode m de la classe c .

Le second exemple — figure 2.4 — décrit la contrainte générée dans le cas d'une instruction de création d'un objet de classe c' par x . L'objet 0 sera référencé par x dans la méthode m de la classe c dans le contexte Ctx , sous réserve que l'objet courant (self) référence $0'$ dans ce même contexte. Le prédicat restant objCtx permet, quant à lui, la création de l'objet 0 à partir de plusieurs éléments, tout en maintenant la généralité de l'analyse.

2.2.4 Généralité de l'analyse

L'algorithme défini ci-dessus représente un cadre générique (*framework*) pour l'analyse de classes. En effet, la théorie définie précédemment permet de raffiner l'analyse en ne changeant la définition que d'un nombre réduit de prédicats, et ce sans modifier d'aucune façon la génération automatique des contraintes. Les trois prédicats permettant le raffinement de l'analyse sont objCtx , methCtx et classOf . Le prédicat objCtx modélise une fonction qui crée un nouvel objet en fonction d'un point de programme syntaxique (classe, méthode, numéro d'instruction), d'une classe à instancier et d'un contexte d'analyse. Ainsi, si $\text{objCtx}(c, m, pc, c', \text{ctx}, \text{self}, \text{newObj})$ est vérifié, alors newObj est un nouvel objet de classe c' créé au point de programme (c, m, pc) dans un contexte ctx dont l'objet courant est self .

Le prédicat methCtx modélise une fonction créant un nouveau contexte d'appel, en fonction d'un point de programme syntaxique, des n arguments de l'appel et du contexte d'appel. Si

$$\text{methCtx}(c, m, pc, \text{self}, [\text{this}, o_1, \dots, o_n], \text{ctx}, \text{newCtx})$$

est valide, alors newCtx est le nouveau contexte d'appel créé à partir du point de programme (c, m, pc) , du contexte ctx et des arguments o_1, \dots, o_n .

Enfin, le prédicat $\text{classOf}(o, c)$ permet d'obtenir la classe c de l'objet o . Les prédicats objCtx et classOf doivent vérifier la contrainte de cohérence suivante :

$$\text{objCtx}(c, m, pc, c', \text{ctx}, \text{self}, o) \Rightarrow \text{classOf}(o, c')$$

Dans un souci de concision, seuls deux des quatre types d'analyses présentés en section 2.1.3 vont être étudiés ci-dessous. Le lecteur intéressé par une étude plus en profondeur se reportera

à [BJ03]. Ces analyses se différencient les unes des autres par la personnalisation de la définition des trois prédicats `objCtx`, `methCtx` et `classOf`.

On notera qu'une légère entorse est faite à `DATALOG` de par l'usage de listes. Cette extension n'est cependant pas un problème théorique puisque les listes de longueur finie (c'est le cas ici) pourraient être représentées par des prédicats. Ceci dit, dans un souci de clarté, les listes seront conservées ici dans leur forme habituelle.

Analyse 0-CFA

On rappelle que, dans cette analyse au contexte dégénéré, nous avons *Objet = Classe* et *Contexte = {ctx}*.

```
objCtx(C, M, Pc, C', Ctx, C, C').
methCtx(C, M, Pc, [Self, O1, ..., On], Ctx, ctx).
classOf(C, C).
```

Analyse *k-l*-CFA

On rappelle que le principe de l'analyse *k-l*-CFA est de garder une chaîne d'appels de longueur *k* et une chaîne de créations d'objets de longueur *l*.

```
objCtx(C, M, Pc, C', Ctx, O, O') :-
  O = [C', [P1, ..., Pl]],
  O' = [C', [[C, M, Pc], P1, ..., Pl-1]].
methCtx(C, M, Pc, [Self, O1, ..., On], Ctx, Ctx') :-
  Ctx = [P1, ..., Pk],
  Ctx' = [[C, M, Pc], P1, ..., Pk-1].
classOf[[C, L], C] :- object([C, L]).
```

2.3 Détermination d'une théorie pour l'analyse du *bytecode*

L'un des objectifs de ce projet de fin d'études a été d'adapter la théorie précédemment exposée pour l'appliquer à du *bytecode* Java.

2.3.1 Le *bytecode* Java

Pourquoi le *bytecode* Java ?

Java est un langage très utilisé par les temps qui courent, notamment dans les applications embarquées telles que la téléphonie mobile, domaine dans lequel Lande travaille régulièrement, en particulier dans le cadre d'un partenariat avec un opérateur de téléphonie bien connu. Cependant, la complexité de ce langage rendrait l'analyse du code source d'un programme particulièrement délicate — et tout à fait irréalisable dans le cadre d'un projet de fin d'études.

De plus, le *bytecode* a une structure plus contraignante que celle de Java, et une analyse automatisée s'en trouve ainsi facilitée.

Enfin, il arrive que des classes soient fournies directement compilées en *bytecode*, sans accès au code source Java correspondant. Une analyse portant sur le Java ne pourrait donc pas analyser ces classes, ce qui pourrait avoir des conséquences fâcheuses.

C'est pour ces raisons que notre analyse portera sur le *bytecode* Java.

Bref rappel sur le *bytecode* Java

Le *bytecode* Java est le résultat de la compilation, par le compilateur Java, d'un programme dont le code source est en Java. Ce *bytecode* est un *code intermédiaire* qui peut être exécuté sous de nombreux systèmes d'exploitation par l'interpréteur Java ou la machine virtuelle Java.

Le *bytecode* est un code binaire³, ce qui permet un traitement plus rapide que le code source Java.

Un fichier de *bytecode* (portant généralement l'extension *.class*) est illisible pour un humain, puisqu'il ne s'agit que de chiffres hexadécimaux. Il existe une transcription du *bytecode* dans un langage plus « lisible », en l'occurrence le Jasmin. L'exemple ci-dessous montre le même code dans ces trois langages.

Java	<i>bytecode</i>	Jasmin
byte x = arr[0];	#2B	aload_1
	#03	iconst_0
	#33	baload_1
	#3D	istore_2

Dans ce qui suivra, on assimilera généralement le *bytecode* au Jasmin par abus de langage. En effet, le second n'est qu'une transcription littérale du premier et permet une meilleure clarté du propos : on mentionnera donc par exemple `iconst_0` comme instruction *bytecode* — et non `#03` comme cela devrait être le cas.

En quelques mots, une classe *bytecode* est un ensemble de méthodes comportant des instructions, chacune étant suivie par un nombre *prédéfini* de paramètres. Le traitement d'un tel langage peut donc être effectué de manière automatisée. Chaque classe possède également un ensemble de constantes (*constant pool*) regroupant par exemple les noms de classes utilisés dans le programme, ainsi que d'autres informations, comme la table des exceptions de chaque méthode.

De manière similaire à l'assembleur, le *bytecode* utilise une structure de *pile*. Chaque instruction peut donc empiler ou dépiler un ou plusieurs éléments. Toute méthode doit obligatoirement déclarer dans son en-tête la taille maximale de la pile qu'elle utilisera.

L'autre spécificité du *bytecode* est l'utilisation de variables locales à une méthode, comparables aux registres d'une machine. Certaines instructions peuvent donc placer sur la pile le contenu de l'une de ces variables locales pour une utilisation ultérieure, ou ranger dans une variable locale la valeur en sommet de pile. Comme pour la taille de la pile, le nombre maximal de variables locales utilisées doit être déclaré en début de méthode.

Enfin, le *bytecode* gère, comme Java, les types (types prédéfinis, tableaux et classes), la portée des méthodes (publiques, privées, etc.), les exceptions, etc.

Pour de plus amples informations sur la structure de ce langage, le lecteur se reportera aux spécifications du *bytecode* Java [LY99].

2.3.2 Détermination d'une théorie pour la génération de contraintes

Le *bytecode* Java possédant une structure particulièrement différente de celle du langage fictif introduit précédemment, la théorie a dû être remaniée en profondeur pour s'adapter aux spécificités du *bytecode*. De plus, il ne s'agissait pas d'appliquer cette théorie à un quelconque sous-ensemble du *bytecode* Java, mais bien à l'intégralité de ce langage, avec ses cas particuliers et spécificités.

Nous avons mis au point un algorithme de génération de contraintes en fonction des instructions *bytecode* ; toutefois, en raison du nombre élevé d'instructions, il n'est pas question de le

³Pour être tout à fait exact, il faudrait traduire *bytecode* par code octet.

reproduire ici, ni même en annexe. Quelques cas parmi les plus marquants seront donnés dans ce qui va suivre, et la plupart des cas significatifs sont regroupés en annexe A. Le lecteur intéressé par l'intégralité de l'algorithme et ses cas particuliers se reportera directement aux commentaires du code source du logiciel développé.

Contraintes relatives à la pile

Le *bytecode* possédant une structure de pile, il est important de modéliser, à chaque point du programme, l'état de la pile. Celui-ci est modélisé par le prédicat suivant :

$$\text{stack}(\mathbf{C}, \mathbf{M}, \mathbf{Pc}, \mathbf{Ctx}, \mathbf{S})$$

On comprend aisément que \mathbf{C} représente la classe, \mathbf{M} la méthode, \mathbf{Pc} le numéro d'instruction (*program counter*), \mathbf{Ctx} le contexte et \mathbf{S} la pile elle-même, modélisée par une liste dont le premier élément est le sommet de pile. Comme indiqué en section 2.2.4 page 21, cette petite entorse à DATALOG n'est pas à l'origine d'un problème majeur. Par ailleurs, comme l'état de la pile au point précédent n'est pas systématiquement connu, on utilisera la notation PROLOG $[A, B \mid S]$ indiquant que les éléments A et B sont au sommet d'une pile inconnue S .

Au début de la méthode, la pile est initialisée à la valeur vide :

$$\llbracket \text{.method } m \rrbracket_c = \text{stack}(c, m, 0, \text{Ctx}, []) \text{ :- } \dots .$$

On notera que l'instruction `.method` constitue le début de la définition d'une méthode en *bytecode*.

Les points de suspension \dots indiquent que d'autres prédicats composent le corps de cette clause — prédicats qui ne seront pas détaillés ici pour des raisons de concision. En bref, il s'agit des prédicats assurant que la méthode a bien été appelée avec des arguments correspondant à ceux attendus. De plus, d'autres clauses sont également générées lors de la définition d'une méthode. Plus d'informations à ce sujet se trouvent en annexe A.1.5.

Si l'instruction *bytecode* ne modifie pas la pile — par exemple l'instruction `nop` qui a le mérite de ne rien faire —, des contraintes de non-modification de pile doivent être générées :

$$\llbracket \text{nop} \rrbracket_{c,m,pc} = \text{stack}(c, m, pc, \text{Ctx}, \mathbf{S}) \text{ :- } \\ \text{stack}(c, m, pc-1, \text{Ctx}, \mathbf{S}).$$

Enfin, si l'instruction *bytecode* modifie la pile, il convient de générer la contrainte correspondante. Par exemple, dans le cas d'un simple dépilement (instruction `pop`), la contrainte suivante est générée :

$$\llbracket \text{pop} \rrbracket_{c,m,pc} = \text{stack}(c, m, pc, \text{Ctx}, \mathbf{S}) \text{ :- } \\ \text{stack}(c, m, pc-1, \text{Ctx}, [0 \mid \mathbf{S}]).$$

L'objet 0 en sommet de pile \mathbf{S} à l'instruction $pc-1$ n'est plus présent à l'instruction pc .

Contraintes relatives aux variables locales

Les variables locales à une méthode utilisées par le *bytecode* sont modélisées par le prédicat suivant :

$$\text{locals}(\mathbf{C}, \mathbf{M}, \mathbf{Pc}, \mathbf{Ctx}, [0_0, 0_1, \dots, 0_n])$$

from	to	target	type
0	19	20	<Class java.io.FileNotFoundException>
0	30	30	<Class java.io.ArrayIndexOutOfBoundsException>
0	30	40	<Class java.io.IOException>

FIG. 2.5 – Exemple de table d’exceptions en *bytecode*

En d’autres termes, au point de programme (C, M, Pc), les variables locales numéro i contiendront les objets 0_i , $i \in [0; max_locals - 1]$ où max_locals représente le nombre maximal de variables locales utilisées par la méthode courante.

De manière similaire à la pile, si une instruction modifie une variable locale, il conviendra de générer la contrainte correspondante. Sinon, une contrainte de non-modification des variables locales est générée. L’annexe A.1.2 donne plus de détails sur le cas des variables locales.

Contraintes relatives aux exceptions

L’un des aspects les plus poussés de ce projet de fin d’études aura été l’inclusion du système d’exceptions à l’algorithme de génération de contraintes.

Pour traiter les erreurs, Java propose un mécanisme qualifié d’*exception*, qui consiste à lancer (*throw*) une exception lors de certaines actions jugées non conformes. Les exceptions peuvent être prédéfinies ou définies par l’utilisateur. Lorsqu’une exception est lancée, la méthode en cours s’arrête et l’exception atteint la méthode appelante. Si l’appel a été effectué dans un bloc *try*, le programme parcourt dans l’ordre les différents traitants (*handler*) *catch* associés à ce bloc. Chaque traitant peut intercepter une exception de classe différente. Le premier traitant conforme à la classe de l’exception — c’est-à-dire traitant une classe mère — va intercepter l’exception. Si aucun traitant conforme n’est trouvé, l’exception est relancée vers le niveau supérieur, et ainsi de suite jusqu’à l’interception par un traitant conforme ou, sinon, l’arrêt du programme.

En *bytecode*, les blocs *try* et *catch* sont symbolisés par une table des exceptions placée au niveau de la définition de méthode. Sur l’exemple de la figure 2.5, on peut noter que, si une exception héritant de `java.io.FileNotFoundException` est lancée entre les instructions 0 et 19, le programme commutera à l’instruction 20. De plus, si une exception est lancée par exemple entre les instructions 20 et 30, le programme regardera d’abord s’il s’agit d’une sous-classe de `java.io.ArrayIndexOutOfBoundsException`. Si c’est le cas, il commutera à l’instruction 30. Sinon, s’il s’agit d’une sous-classe de `java.io.IOException`, il commutera à l’instruction 40. Sinon, l’exception sera relancée, et l’opération sera réitérée au niveau supérieur.

Pour chaque méthode, nous allons analyser la table des exceptions et générer, pour chaque entrée, les contraintes suivantes :

```
handle(c, m, from, to, target, e).
stack(c, m, target, Ctx, E) :- active(c, m, target, Ctx, E).
```

from et *to* délimitent le bloc *try*, *target* fait référence au numéro d’instruction de début du traitant d’exception et *e* représente la classe d’exception traitée par ce traitant. On notera que la première clause n’est pas à elle seule suffisante pour déterminer quel traitant va effectivement traiter l’exception. Cependant, de manière similaire à la méthode `lk` qui détermine quelle méthode va effectivement être appelée sur un objet en particulier, des prédicats sont définis globalement — notamment le prédicat `active` dont on remarque ici une utilisation —, permettant de trouver le traitant conforme à une exception. Ceci est détaillé en annexe A.1.7.

De plus, les exceptions sont générées explicitement au moyen de l'instruction `athrow`, qui lance l'objet présent en sommet de pile à ce point de programme. La contrainte générée pour une telle instruction est la suivante :

$$\llbracket \text{athrow} \rrbracket_{c,m,pc} = \text{throw}(c, m, pc, E) \text{ :- } \text{stack}(c, m, pc-1, \text{Ctx}, [E \mid S]).$$

On comprend aisément que l'objet sur la pile va bien être lancé, au moyen du prédicat `throw`.

Enfin, plusieurs contraintes sont définies globalement (en en-tête du fichier donné au solveur de contraintes). Ces contraintes ne seront pas détaillées ici mais se trouvent en annexe A.1.7.

Ainsi, la théorie partiellement présentée ici permet de décrire l'intégralité du *bytecode* Java et de générer les contraintes DATALOG correspondant à l'ensemble des instructions. Afin de mettre en pratique cette théorie et de pouvoir en vérifier la validité, un logiciel a été développé.

2.4 Choix d'implémentation

La théorie exposée dans la section précédente permet d'effectuer une analyse modulaire sur un ensemble de classes de *bytecode* Java. Afin de vérifier le gain de temps apporté par une analyse modulaire, par rapport à une analyse globale, nous avons développé le logiciel Toutânkhamel. Celui-ci génère des contraintes DATALOG à partir de plusieurs classes et les résout au moyen d'un solveur existant.

2.4.1 Génération des contraintes

Ce paragraphe détaille comment, à partir d'un fichier source *bytecode*, nous appliquons l'algorithme de génération des contraintes évoqué dans la section précédente. Cette opération a été entièrement codée en utilisant l'implémentation *Objective Caml system release 3.09* du langage caml.

Ocaml

Caml est un langage de programmation généraliste conçu pour la sécurité et la fiabilité des programmes. Il se prête à des styles de programmation fonctionnelle, impérative et orientée objet. C'est de plus un langage fortement typé.

Le langage Caml est développé depuis 1985 par les équipes Formel puis Cristal de l'Inria.

Caml pourrait être le langage parfait, s'il n'avait l'inconvénient de créer une forte dépendance chez le programmeur ocaml, dépendance accompagnée de pertes de mémoires, puisque celui-ci finit rapidement par oublier l'existence d'autres langages de programmation.

Analyse syntaxique

La génération de contraintes s'appuie sur l'analyseur syntaxique (*parser*) JavaLib développé sous licence libre par Nicolas Cannasse⁴, transformant un fichier *bytecode* en une structure de données Caml. Cet analyseur syntaxique a par la suite été modifié par Laurent Hubert⁵ (stagiaire chez Lande) au cours de l'été 2005, puis par moi-même dans le cadre de ce projet de fin d'études.

⁴<http://ncannasse.free.fr/>

⁵<http://www.trebuh.net/>

Algorithme de génération de contraintes

L'algorithme de génération des contraintes est codé dans plusieurs modules. Profitant du typage fort offert par ocaml, les contraintes sont d'abord générées sous forme *typée* ; cela permet de s'assurer, par exemple, que le prédicat `stack(C, M, Pc, Ctx, S)` prendra bien cinq arguments correctement typés (en l'occurrence un nom de classe, un nom de méthode, un numéro d'instruction, une variable de contexte, et une variable de pile). Ce typage fort réduit considérablement le risque d'erreurs dans le programme.

Ces contraintes sont ensuite converties dans une forme non typée plus proche de la définition logique du prédicat (une chaîne de caractères et une liste de variables ou constantes). Cette opération de conversion utilise un temps d'exécution tout à fait négligeable.

Enfin, les contraintes sont converties dans une forme fidèle à la syntaxe DATALOG puis sont écrites dans un fichier.

2.4.2 Résolution des contraintes

L'ensemble de contraintes ainsi obtenu est ensuite résolu par un solveur. L'idéal aurait été d'utiliser un solveur dédié à DATALOG. Malheureusement, les solveurs existants gèrent des systèmes de contraintes réduits (n'oublions pas que DATALOG est à la base un langage de requêtes relationnelles), or les fichiers de contraintes générés par le programme peuvent rapidement atteindre la dizaine de milliers de contraintes, même pour des classes de taille très modeste. De plus, ces solveurs ne gèrent ni les listes, ni la négation.

Il a été envisagé d'écrire un solveur adapté à DATALOG mais ce travail aurait largement dépassé les limites temporelles de ce projet de fin d'études. Il faut donc espérer qu'un futur stagiaire s'intéresse au problème...

Le choix s'est alors porté vers un solveur existant, en l'occurrence XSB.

XSB

XSB est un solveur de contraintes orienté recherche fonctionnant sur les plates-formes Unix et Windows. En plus d'être un solveur PROLOG classique, il résout également les contraintes DATALOG (puisque'il s'agit d'un sous-ensemble de PROLOG) et possède l'avantage majeur de gérer les récursions à gauche — ce que ne fait pas un solveur classique — en tabulant les résultats pour éviter les récursions infinies.

À l'aide de contraintes prédéfinies, XSB génère des fichiers permettant par la suite la création de graphes.

2.4.3 Génération de graphes

L'un des objectifs d'une analyse de classes est la génération de graphes d'appel, afin de savoir quelle méthode a appelé quelle autre. D'autres objectifs peuvent être la génération de graphes de relations entre objets, ainsi que, plus simplement, le diagramme de hiérarchie de classes.

À partir des fichiers générés par XSB, on peut aisément créer des graphes à l'aide de GraphViz.

GraphViz

L'application GraphViz permet de représenter sous forme d'images des graphes initialement codés sous forme de texte. Elle a été conçue par une équipe des laboratoires de recherche de AT&T (*American Telephone & Telegraph*).

```

digraph G {
  "ornithorynque" -> "mammifere";
  "mammifere" -> "animal";
  "canard" -> "oiseau";
  "oiseau" -> "animal";
  "ornithorynque" -> "pond des oeufs";
  "canard" -> "pond des oeufs";
}

```

FIG. 2.6 – Fichier source basique pour GraphViz

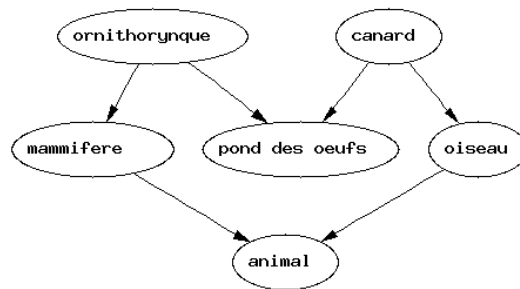


FIG. 2.7 – Graphe basique généré par GraphViz

Cette application convient à la représentation de graphes très denses comprenant un grand nombre de nœuds grâce à des algorithmes puissants. GraphViz est particulièrement rapide à l'exécution et le rendu est optimisé afin que les liens ne recouvrent pas les nœuds et qu'ils ne se croisent pas — dans le cas de graphes planaires, tout du moins.

De plus, entièrement paramétrable, l'application permet de personnaliser le rendu des graphes par le choix des formes, couleurs et polices de caractères. Les formats de sortie sont très variés.

L'avantage majeur réside dans le format particulièrement simple de l'entrée. La figure 2.7 montre l'image générée par GraphViz à partir du code décrit en figure 2.6. On notera la simplification à l'extrême du langage permettant la génération du graphe ; le graphe généré, bien que très basique, a néanmoins une présentation convenable.

Des améliorations à ce graphique basique sont possibles à peu de frais. En ajoutant quelques instructions supplémentaires au fichier source (figure 2.8), en particulier des couleurs et des variations de la forme des nœuds, on obtient un graphique nettement plus agréable à regarder (figure 2.9).

2.5 Résultats

Le logiciel Toutânkhamel crée, pour chaque analyse, trois graphes. Bien entendu, la génération de graphes est une étape particulièrement simple et un utilisateur pourrait aisément choisir d'en générer d'autres sans avoir ni à remodifier la théorie, ni même à toucher au code ocaml.

```

digraph G {
  bgcolor=azure;
  node [shape=box, color=lightblue2, style=filled];
  edge [color=gold];
  "pond des oeufs" [shape=ellipse, color=white, style=filled];
  "ornithorynque" -> "mammifere";
  "mammifere" -> "animal";
  "canard" -> "oiseau";
  "oiseau" -> "animal";
  "ornithorynque" -> "pond des oeufs" [arrowsize=1, color=red, label="peu"];
  "canard" -> "pond des oeufs" [arrowsize=1, color=red, label="beaucoup"];
}

```

FIG. 2.8 – Fichier source plus évolué pour GraphViz

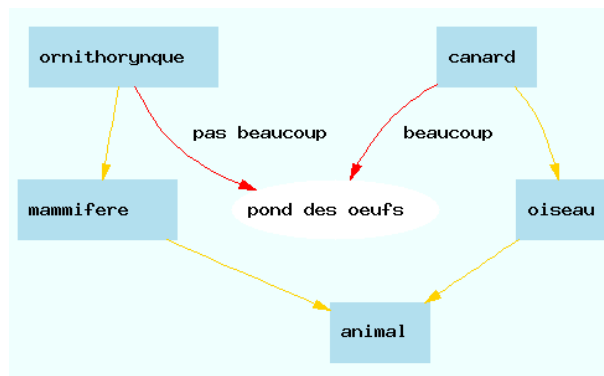


FIG. 2.9 – Graphe plus évolué généré par GraphViz

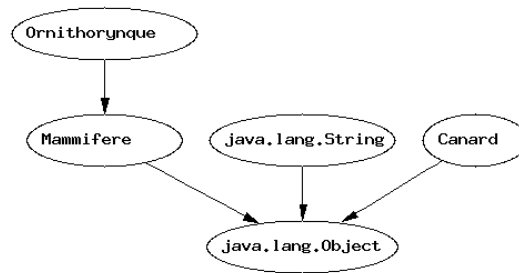


FIG. 2.10 – Hiérarchie de classes

2.5.1 Hiérarchie de classes

Le premier graphe généré indique la hiérarchie des classes passées à l'analyse. On notera bien entendu qu'une simple analyse syntaxique des en-têtes de classe suffirait pour déterminer cette hiérarchie, et qu'une analyse par contraintes n'est pas nécessaire. Cependant, ce diagramme pouvant s'avérer utile, on profite de l'analyse de classes effectuée pour le générer.

Les classes sont représentées par des cercles et une flèche pointant d'une classe A vers une classe B indique que A hérite de B. Une flèche de couleur bleue indique une relation d'interface : A implémente B.

Ainsi, sur l'exemple de hiérarchie de classes de la figure 2.10, la classe `Ornithorynque` hérite de `Mammifere` et, conformément à la règle en vigueur en Java, toutes les classes sont des sous-classes de `java.lang.Object`.

2.5.2 Diagramme d'appels

Le second diagramme généré indique les méthodes ayant été appelées, en précisant le point de programme où s'est fait l'appel, ainsi que la nature des arguments passés en paramètres. Une méthode m d'arité⁶ n dans une classe c sera représentée sur le graphe par une ellipse contenant $c \gg m/n$. En outre, le contexte des appels est représenté par le numéro d'instruction auquel l'appel est fait, ainsi que la nature (classe ou type prédéfini) des arguments passés en paramètres. L'analyse permettrait bien entendu d'indiquer le véritable contexte d'appel (dépendant du type d'analyse) mais, afin de simplifier la visualisation, le graphe s'en tiendra à ces deux informations.

Considérons l'exemple des deux classes de la figure 2.11. Nous créons dans la méthode `main` de la classe `Saucisse` un barbecue, modélisé par un tableau de saucisses. L'ensemble des objets contenus dans ce barbecue est ensuite grillé (comme il se doit dans un bon barbecue); cependant, en étudiant le graphe d'appels figure 2.12, on s'aperçoit que seule la méthode `Merguez » grille()` est appelée, et non `Saucisse » grille()`. Cela est dû au fait que les éléments du tableau `barbecue` sont tous des objets de type `Merguez`, sous-classe de `Saucisse`. La méthode `Saucisse » grille()` n'est donc jamais appelée, et il devient possible d'optimiser le programme, par exemple en décrétant que le tableau `barbecue` ne contiendra que des objets de type `Merguez`, ou encore en supprimant la méthode `Saucisse » grille()`.

⁶L'arité fait référence au nombre d'arguments de la méthode, plus l'objet passé en référence s'il s'agit d'une méthode non statique.

```

public class Saucisse{
    public void grille(){
    }
    public static void main(String[] args){
        Saucisse[] barbecue = new Saucisse[2];
        barbecue[0] = new Merguez();
        barbecue[1] = new Merguez();
        for(int i = 0; i < barbecue.length; i++){
            barbecue[i].grille();
        }
    }
}

public class Merguez extends Saucisse{
    public void grille(){
    }
}

```

FIG. 2.11 – Exemple de classes

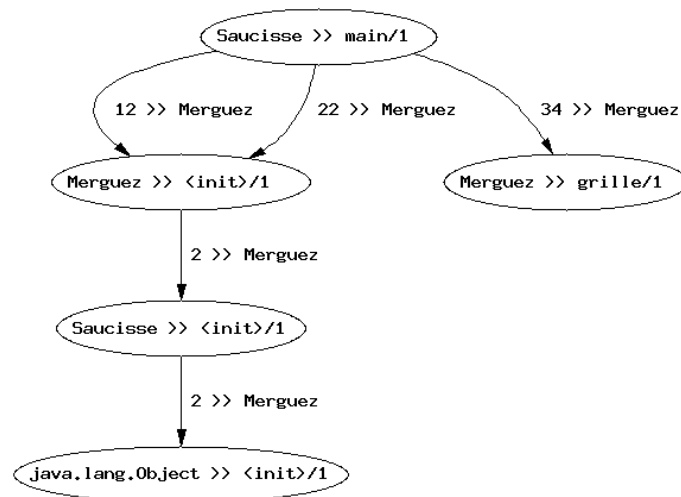


FIG. 2.12 – Graphe d'appels

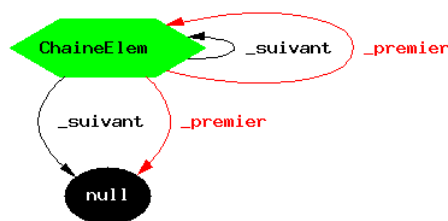


FIG. 2.13 – Diagramme d’objets en 0-CFA

2.5.3 Diagramme d’objets

Les objets créés au cours d’un programme dont les classes sont analysées par Toutânkhémel sont représentés sur un diagramme d’objets, avec leurs relations respectives : les objets peuvent se référencer entre eux via leurs champs (statiques ou non) et via les tableaux. Les types prédéfinis sont également représentés sur ce diagramme.

Champs

Considérons l’exemple d’une chaîne d’éléments symbolisée par la classe `ChaîneElem`, dont le code source est donné en annexe B.1. Cette classe permet de représenter une chaîne basique à partir de ses éléments : chaque objet ainsi chaîné a un successeur `_suivant`, et la classe référence le premier élément via le champ statique `_premier`.

Le diagramme d’objets correspondant à l’analyse 0-CFA est représenté sur la figure 2.13. Les objets étant abstraits par leur classe dans une telle analyse, on constate donc que la classe `ChaîneElem` se référence elle-même via ses champs. Une flèche noire indique un champ, et une flèche rouge indique un champ statique. Le nom des champs est également représenté.

Dans le cadre d’une analyse 2-2-CFA, dont le diagramme d’objets est représenté sur la figure 2.14, les différents objets sont distingués par leur contexte de création. On observe donc de manière plus explicite la chaîne des éléments. Les objets sont représentés par des ellipses au fond vert dans lequel s’inscrit le nom de leur classe et la chaîne des points de programme ayant abouti à leur création, ici de longueur 2 puisque nous nous situons dans le cadre de l’analyse 2-2-CFA. On notera par ailleurs que tous les éléments de la chaîne référencent `null` via leur champ `_suivant` alors que, à un instant donné, seul un élément pointe vers `null`. En effet, le diagramme ainsi représenté n’est pas l’état à un point du programme mais l’ensemble des états que peuvent prendre les objets au cours de l’exécution. On retrouve donc ici la notion d’approximation introduite par l’interprétation abstraite. On notera enfin la présence de la classe `ChaîneElem` au milieu des objets. Il s’agit d’une facilité introduite pour la représentation des champs statiques ; en effet, tous les objets d’une classe devraient référencer le même objet via un champ statique. Toutefois, afin de limiter le nombre de flèches sur le diagramme, on choisit d’introduire directement la classe sur celui-ci, qui se charge de référencer les différents champs statiques.

Enfin, il est important de noter qu’une approximation importante est effectuée sur les champs, puisque ceux-ci ne prennent pas en compte le contexte : ainsi, il est impossible de déterminer à quel endroit du programme un objet référence un autre via un champ, la seule information disponible étant que cet objet est *susceptible* de référencer cet autre objet quelque part dans le programme.

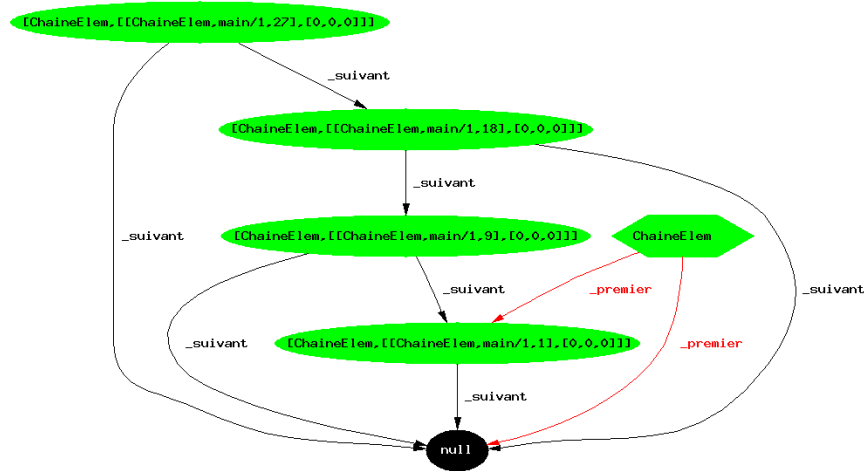


FIG. 2.14 – Diagramme d’objets en 2-2-CFA

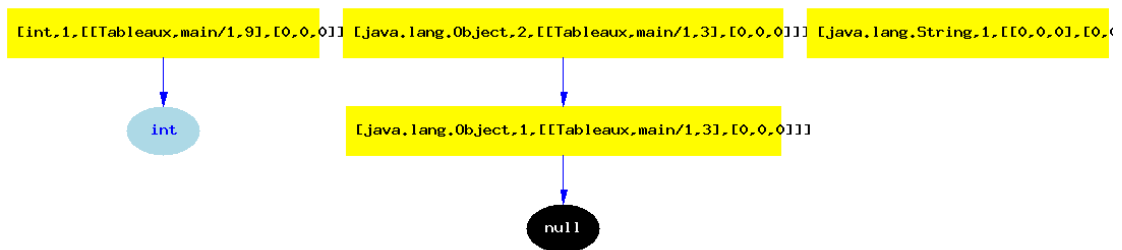


FIG. 2.15 – Diagramme d’objets contenant des tableaux

Tableaux

Les tableaux sont représentés sur les diagrammes d’objets de Toutânkhamel par des rectangles jaunes contenant un triplet constitué de leur classe, leur dimension et, comme pour les objets, leur liste de points de création dans le cas d’une analyse suffisamment précise. Les tableaux peuvent référencer soit d’autres tableaux (dans le cas de tableaux multi-dimensionnels), soit des objets éventuellement nuls, soit des types prédéfinis. Sur le diagramme d’objets figure 2.15 dans le cadre d’une analyse 2-2-CFA, on peut ainsi observer un tableau de dimension 1 contenant des entiers, un tableaux de dimension 2 pouvant contenir des objets de type `Object` (mais pour l’instant vide, puisque ne référençant que `null`), et un tableau de chaînes (`String`). On notera que ce tableau de chaînes est en principe présent sur tous les diagrammes d’objets, puisqu’il s’agit du tableau de paramètres passé à la méthode `main` du programme.

De même que pour les champs, il est important de noter qu’une approximation importante est effectuée sur les tableaux, puisque ceux-ci ne prennent pas en compte le contexte : ainsi, il est impossible de déterminer à quel endroit du programme un objet appartient à un tableau, ni même dans quelle « case » du tableau ledit objet est contenu. Les tableaux sont ainsi abstraits par des *tas*, puisque la position de l’objet n’est pas connue dans le tableau.

2.5.4 Modularité

L'objectif de ce projet de fin d'études était de réaliser une analyse *modulaire*, dont l'un des intérêts est un *gain de temps* par rapport à une analyse globale. La théorie générant des contraintes permettant une analyse du *bytecode* Java autorise l'analyse modulaire : les contraintes peuvent être générées à partir d'une classe même si les autres classes ne sont pas présentes à ce moment et, une fois les contraintes des différentes classes regroupées, l'obtention des résultats est possible au même titre que dans une analyse globale. Ainsi, en cas de modification d'une seule classe, l'exécution de l'analyse durera moins longtemps que dans le cas où il faudrait tout réanalyser.

L'une des possibilités pour réaliser un gain de temps est d'effectuer des traitements sur les contraintes avant leur résolution. La technique du dépliage a notamment été implémentée dans Toutânkhamel et des tests concluants en début de projet avaient montré une diminution drastique du nombre de contraintes relatives à la pile et aux variables locales, et par conséquent un temps de résolution nettement réduit. Cette technique de résolution a momentanément été laissée de côté, notamment pour permettre la fin de l'implémentation de l'algorithme.

Afin que l'analyse modulaire soit fonctionnelle, il est nécessaire d'implémenter un mécanisme permettant de sauver les contraintes des classes déjà analysées dans des fichiers, et de charger ces contraintes lors de la résolution des contraintes émanant des différentes classes. Ce mécanisme de sauvegarde apporte un gain de temps puisque, en cas d'analyse de classes déjà analysées par le passé, il suffira de charger les contraintes précédemment générées au lieu de tout réanalyser.

Enfin, le gain de l'analyse modulaire se trouve principalement dans le fait que des traitements de présolution des contraintes sont effectués avant leur sauvegarde, traitements qu'il ne faudra donc pas réeffectuer tant que la classe correspondante ne sera pas modifiée.

Mise en œuvre

Lors de l'appel au programme Toutânkhamel, deux types de paramètres sont donnés : d'une part les classes à analyser en totalité et d'autre part les fichiers de contraintes de classes déjà analysées.

Concrètement, si l'on souhaite analyser trois classes A, B et C, on passe tout d'abord à Toutânkhamel ces trois classes. Elles sont entièrement analysées : analyse syntaxique, génération des contraintes, conversion des contraintes, traitements, et enfin sauvegarde des contraintes dans des fichiers portant l'extension `.ttk`. Puis les contraintes sont regroupées, passées au solveur, résolues et les graphes sont générés.

Dans un second temps, on suppose que l'on modifie la classe A. Nous nous trouvons donc dans un cas d'analyse modulaire : seule cette classe sera réanalysée. Après compilation de cette classe, on relance Toutânkhamel avec A en tant que classe à analyser en totalité, et B et C en tant que fichiers de contraintes portant l'extension `.ttk`. B et C ne sont donc pas analysées à nouveau mais leurs contraintes générées et traitées dans le cas de la première analyse sont importées. Puis à nouveau, les contraintes sont résolues et les graphes sont générés.

Traitements sur les contraintes

Afin d'obtenir un gain de temps, des traitements sont effectués sur les contraintes afin de les simplifier, et ce avant leur sauvegarde dans le fichier `.ttk`.

Ces traitements sont de nature diverse :

- simplification par élimination des clauses redondantes,
- simplification par dépliages successifs,

- suppression des contraintes concernant la pile ou les variables locales, et n'apparaissant pas ailleurs.

Ces traitements sont *coûteux* et le gain en temps n'est pas évident par rapport à une analyse globale ; il se peut même que le temps d'analyse modulaire soit supérieur à celui demandé par une analyse globale. En revanche, le gain devient net dès lors qu'on analyse un programme dont seule une classe a été modifiée par rapport à une précédente analyse.

Valeurs expérimentales

Afin de prendre un exemple concret, considérons la classe `String` de Java, la deuxième classe prédéfinie la plus importante en taille.

Dans le cas d'une analyse globale, le temps d'exécution de `caml` est de l'ordre d'une seconde. Ainsi, 6942 contraintes sont générées, puis résolues par XSB en 27 secondes par XSB. Le total est par conséquent de 28 secondes.

Si l'on effectue des traitements sur les contraintes générées, le temps d'exécution passe d'une à 423 secondes, pendant lesquelles le nombre de contraintes est réduit à 2747. Ainsi, la résolution par XSB descend à 10 secondes. L'intérêt de cette analyse modulaire est donc, à première vue, tout à fait inutile puisque le temps global est passé de 28 à 433 secondes.

Cependant, il convient de rappeler que l'immense majorité des programmes Java utilise la classe `String`. Dans le cas d'une analyse globale, il faudrait par conséquent réanalyser à *chaque fois* cette classe, impliquant une durée de 28 secondes au total. Or l'analyse modulaire permet de réutiliser les contraintes précédemment générées (la classe `String` ne variera pas), contraintes dont le chargement représente un temps négligeable ; ces contraintes ayant été traitées, elles sont en outre simplifiées et, dans notre cas, le temps de résolution par XSB ne sera plus que de 10 secondes. Ainsi, on passe d'un temps total de 28 secondes à 10 secondes, soit un gain de plus de 60 %.

Prenons maintenant l'exemple d'un programme complet utilisant la classe `String`. Ce programme est composé des classes (prédéfinies ou non) suivantes :

- `Saucisse.class`,
- `Merguez.class`,
- `Chipo.class`,
- `Barbecue.class`,
- `java/lang/Object.class`,
- `java/lang/String.class`.

Une comparaison entre analyse globale et analyse modulaire sur cet exemple est présentée sur la figure 2.16. La première colonne présente une analyse globale (toutes les classes sont analysées sans effectuer de traitements), tandis que la seconde présente une analyse modulaire pour la première fois et la troisième colonne une analyse modulaire où on ne réanalyse que `Merguez.class`.

Ainsi, sur cet exemple⁷, l'utilisation de l'analyse modulaire apporte un gain de 66 % par rapport à une analyse globale, dans le cas où on réanalyse une classe.

À noter que les perspectives envisagées devraient permettre d'augmenter encore ce gain, les contraintes restantes étaient visiblement encore simplifiables.

⁷Le code source des classes de cet exemple est entièrement détaillé à l'annexe B.3.

	Analyse globale	1 ^{ère} analyse modulaire	2 ^{nde} analyse modulaire
Classes analysées	Saucisse.class Merguez.class Chipo.class Barbecue.class Object.class String.class	Saucisse.class Merguez.class Chipo.class Barbecue.class Object.class String.class	Merguez.class
Contraintes importées			Merguez.ttk Chipo.ttk Barbecue.ttk Object.ttk String.ttk
Fichiers générés		Saucisse.ttk Merguez.ttk Chipo.ttk Barbecue.ttk Object.ttk String.ttk	Merguez.ttk
Contraintes générées	8542	3169	3169
Temps de génération	1,39 s	1,39 s	0,07 s
Temps de traitement	-	450,98 s	0,31 s
Temps de résolution	31,19 s	11,52 s	11,52 s
Temps total	32,58 s	463,89 s	11,90 s

FIG. 2.16 – Comparaison entre analyses globale et modulaire

2.6 Perspectives

Plusieurs pans du logiciel Toutânkhameh développé au cours de ce projet de fin d'études peuvent encore être améliorés ou diversifiés.

2.6.1 Analyseur syntaxique JavaLib

L'analyseur syntaxique JavaLib a la fâcheuse habitude de n'analyser qu'une seule classe par fichier. Or, en Java, il est possible de définir plusieurs classes au sein d'un même fichier `.java`, notamment dans le cas des sous-classes. Il serait donc souhaitable d'apporter une petite modification à cet analyseur syntaxique afin de pallier ce problème.

2.6.2 Ligne de commande

Bien que cela relève du détail, il serait souhaitable de reprogrammer l'appel de la partie `caml` de Toutânkhameh avec ses différentes options ; en effet, cette ligne de commande a été programmée sans utiliser la bibliothèque `Arg`, dédiée aux lignes de commandes.

Concrètement, la fonction à améliorer est la fonction `get_parameters()` définie dans le module `ttkMain.ml`.

2.6.3 Modularité

Afin d'augmenter le gain de temps apporté par l'analyse modulaire, les techniques de résolution des contraintes devraient être améliorées. La technique du dépliage a été implémentée dans Toutânkhameh mais n'est pas utilisée par l'analyse actuelle. Le dépliage permet une résolution partielle des contraintes et constitue la base des solveurs PROLOG. Le dépliage utilise notamment les techniques de renommage et d'unification, techniques également implémentées (mais non utilisées) dans Toutânkhameh. Cependant, dans notre cas, le dépliage diverge et l'analyse ne s'arrête pas, le nombre de contraintes augmentant à chaque dépliage.

C'est pourquoi une technique de dépliage simplifié est utilisée : cette technique ne déplie une clause que s'il n'y a qu'un seul dépliage possible. Après, on supprime les clauses du système relatives aux variables locales ou à la pile, et dont la tête n'est jamais utilisée dans un autre corps de clause. L'état de la pile et des variables locales n'est en effet utile que pour résoudre les appels de méthodes ou les objets. Ainsi, le nombre de contraintes est réduit d'environ 60 % après atteinte d'un point fixe.

D'autres techniques de simplification des contraintes pourraient être implémentées ; la base d'ores et déjà disponible dans Toutânkhameh pour l'analyse modulaire devrait permettre la mise en œuvre de la minimisation, du dépliage itératif, ou d'une combinaison de ces techniques.

2.6.4 Analyse des bibliothèques Java

Un cadre a été développé au cours de ce projet pour l'analyse des bibliothèques existantes Java. Ces classes prédéfinies (telles que `java.lang.Object`, `java.util.Stack`, etc.) utilisent, en plus des méthodes « classiques », des méthodes dites *natives* faisant appel à du code C^{++} . L'objet de ce projet de fin d'études n'étant pas de s'attaquer au langage C^{++} , l'algorithme de génération de contraintes se borne à analyser la définition des méthodes natives, et génère une contrainte rendant le type d'objet attendu en cas d'appel de cette méthode. Le mécanisme d'analyse des classes prédéfinies a été correctement implémenté ; pour plus de détails sur la génération des contraintes relatives aux méthodes natives, le lecteur se reportera à l'annexe A.1.5 page 47.

Une amélioration souhaitable serait alors d'analyser une fois pour toutes chacune de ces classes prédéfinies et de sauvegarder les contraintes ainsi générées dans des fichiers séparés. Par la suite, lors d'une analyse de classes, si une référence quelconque (appel de méthode, création d'objet, etc.) est faite à une telle classe prédéfinie, le logiciel inclura les contraintes correspondantes avant traitement et résolution par le solveur.

2.6.5 Initialisation des champs des objets

À l'heure actuelle, lors de la création d'un objet, les champs de ce dernier sont initialisés à des valeurs par défaut, à savoir `null` pour les champs référençant des objets, et leur propre type pour des champs contenant un type prédéfini. Concrètement, ces contraintes sont générées par l'algorithme dans un cas particulier des définitions de méthodes, c'est-à-dire dans le cas où la méthode s'appelle `<init>`.

Mais cette théorie initialisant les champs des objets à la définition de la méthode `<init>` est inexacte. En effet, cette initialisation est nécessaire pour tout champ n'étant pas explicitement initialisé dans cette méthode; cependant, si l'un des champs est initialisé dans la suite de la méthode, l'analyse déduira que ce champ peut prendre comme valeur soit `null` soit l'objet créé dans la suite de la méthode. Mais cela est inexact, puisque l'objet n'aura en réalité jamais la valeur `null`, et cela peut partiellement fausser l'analyse — surtout si le but de celle-ci est de vérifier que tous les objets sont correctement initialisés en tout point du programme. Il serait alors judicieux de n'initialiser que les champs qui ne seront pas initialisés explicitement dans le corps de la méthode `<init>`. Toutefois, cette opération peut s'avérer difficile à mettre en œuvre.

2.6.6 Parallélisme

Si l'ensemble des instructions est traité, il convient cependant de rappeler que la partie concernant les processus (*threads*) n'a pas été testée; il se peut tout à fait que notre analyse ne se prête pas à la gestion de processus en parallèle.

2.6.7 Solveur de contraintes DATALOG

Un solveur de contraintes DATALOG serait également le bienvenu. L'idéal serait de l'appliquer à la structure de données ocaml créée dans Toutânkhamel, afin de minimiser le temps d'exécution. Toutefois, la construction d'un tel solveur est complexe et pourrait constituer le sujet d'un stage à part entière, d'autant que des propriétés externes à DATALOG sont nécessaires, notamment l'usage de la négation.

Chapitre 3

Conclusion

À partir d'une théorie permettant de résoudre par contraintes — et de manière modulaire — un ensemble de classes d'un langage fictif, ce projet de fin d'études nous a permis de générer un algorithme permettant l'analyse de l'ensemble du langage *bytecode* Java au moyen de contraintes DATALOG. Cette théorie est générique et peut être appliquée de manière immédiate à diverses analyses de classes plus ou moins précises. De plus, le développement du logiciel Toutânkhamel a permis la mise en pratique et la validation de cette théorie sur des exemples de taille moyenne. Ainsi, il devient possible de déduire ou vérifier de nombreuses propriétés du programme, ainsi que générer, par exemple, des diagrammes de hiérarchies de classes, des diagrammes d'objets ou encore d'appels de méthodes. De plus, ce logiciel permet la modularité de l'analyse et permet de ne réanalyser que les classes modifiées d'un programme, ce qui produit de surcroît un gain de temps dans l'analyse. En outre, plusieurs parties du logiciel restent ouvertes et des applications variées restent de ce fait possibles.

L'équipe Lande de l'Irisa dispose désormais d'un outil pouvant servir de base à des déductions ou des vérifications de propriétés de programmes, permettant dans un second temps des optimisations (élagage de méthodes non appelées, simplifications de la chaîne des appels, etc.). Les applications sont nombreuses et variées : sécurité au niveau *bytecode*, tout particulièrement dans le cadre du partenariat entre Lande et un opérateur de téléphonie mobile bien connu, assistance des étudiants de premier cycle de l'Ifsic dans le cadre de leurs séances de travaux pratiques de Java, etc.

Enfin, ce projet de fin d'études aura été pour moi l'occasion de confirmer mon orientation vers le monde la recherche, orientation initiée au sein de la *Technische Universität* de Dresde. En étant intégré au sein d'une équipe de l'Irisa, j'ai pu améliorer de manière significative mes aptitudes en compréhension et synthèse de documents de recherche, ainsi que mes connaissances en analyse de programme, et tout particulièrement concernant l'interprétation abstraite et l'analyse statique. Et, bien entendu, ma passion sans borne pour ce langage formidable qu'est caml s'en est trouvée encore grandie.

Bibliographie

- [Age95] Ole Agesen. The cartesian product algorithm : Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.
- [And06] Étienne André. *Toutânkhamel — Manuel du développeur*, 2006.
- [BJ03] Frédéric Besson and Thomas Jensen. Modular class analysis with datalog. In R. Cousot, editor, *Proc. of 10th Static Analysis Symposium (SAS 2003)*, pages 19–36. Springer LNCS vol. 2694, 2003.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
- [CC02] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In R.N. Horspool, editor, *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, pages 159–178, Grenoble, France, April 6–14 2002. LNCS 2304, Springer, Berlin.
- [GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6) :685–746, 2001.
- [LY99] Tim Lindholm and Frank Yellin. *The JavaTM Virtual Machine Specification — Second Edition*, 1999.
- [Mon03] Édouard Monnier. *Introduction à la programmation fonctionnelle — cours de 3^e année*, 2003.
- [Ryd03a] B. Ryder. Analysis of object-oriented programming languages, February 2003.
- [Ryd03b] B. Ryder. Dimensions of precision in reference analysis of object-oriented languages, 2003.
- [Shi91] Olin Grigsby Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, 1991.
- [SSW⁺05] Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, Luis de Castro, Steve Dawson, and Michael Kifer. *The XSB System — Version 2.7.1 — Volume 1 : Programmer’s Manual*, 2005.

Annexe A

Génération des contraintes

A.1 Contraintes relatives aux instructions *bytecode*

A.1.1 Remarques

Cette section présente les contraintes DATALOG générées en fonction des instructions *bytecode* rencontrées. Il n'est en aucun cas envisagé de cataloguer de manière exhaustive les contraintes, mais simplement de donner des indications pour les cas les plus courants afin de pouvoir mieux comprendre le code.

À chaque point de programme, on génère des contraintes permettant de déduire l'état des variables locales et de la pile à ce point de programme en fonction de leur état au point précédent. Si l'état de la pile est le plus souvent modifié, celui des variables locales n'est changé que lors d'une instruction de modification explicite des variables locales. En cas de non-modification, il faudra néanmoins générer la clause stipulant que leur état reste inchangé; toutefois, pour des raisons évidentes de clarté et de concision, ces clauses de non-modification des variables locales seront omises de ce document.

Par ailleurs, un objet tient généralement sur un mot sur la pile du *bytecode* et est représenté par un élément dans les variables locales. Toutefois, dans le cas d'un élément de type prédéfini `long` ou `double`, il faudra procéder aux modifications nécessaires pour faire tenir l'élément en question sur deux mots de la pile, ou dans deux variables locales contiguës.

`c`, `m` et `pc` représentent respectivement la classe, la méthode et le numéro de l'instruction rencontrée.

Les contraintes sont écrites dans un style proche de PROLOG.

A.1.2 Opérations sur la pile et les variables locales

Placement d'une donnée sur la pile (`iconst_0`, `ldc n`, etc.)

```
stack(c, m, pc, Ctx, [E | S]) :-  
    stack(c, m, pc-1, Ctx, S).
```

E représente un type prédéfini ou un objet. Noter qu'il faudra charger deux fois le même élément E si celui-ci tient sur deux mots (type prédéfini `long` ou `double`).

Placement d'une donnée en local (`astore n`, `istore n`, etc.)

```
stack(c, m, pc, Ctx, S) :-
    stack(c, m, pc-1, Ctx, [E | S]).
```

```
locals(c, m, pc, Ctx, [_ , _ , ... , 0 , ... , _]) :-
    stack(c, m, pc-1, Ctx, [0 | S]).
```

Si E tient sur deux mots, la variable locale n+1 contient désormais ce même élément E.

Récupération de la donnée locale numéro *i* (aload *i*, fload *i*, etc.)

```
stack(c, m, pc, Ctx, [0i | S]) :-
    locals(c, m, pc-1, Ctx, [_ , _ , ... , 0i , ... , _]),
    stack(c, m, pc-1, Ctx, S).
```

Si E tient sur deux mots, la variable locale n+1 contenait également l'élément E.

A.1.3 Création d'objet

Création d'un objet de classe *c'* (new *c'*)

```
stack(c, m, pc, Ctx, [0' | S]) :-
    stack(c, m, pc-1, Ctx, S),
    self(c, m, Ctx, 0),
    objCtx(c, m, pc, c', Ctx, 0, 0').
```

```
object(0') :-
    stack(c, m, pc-1, Ctx, S),
    self(c, m, Ctx, 0),
    objCtx(c, m, pc, c', Ctx, 0, 0').
```

À noter que les champs de l'objet ainsi créé ne sont pas initialisés lors du `new` mais, comme précisé ci-après, lors de la définition de la méthode `<init>`.

A.1.4 Accès aux champs

Les champs non statiques des objets sont initialisés par des contraintes générées lors de la définition de la méthode `<init>`. Les champs statiques, quant à eux, sont initialisés par des contraintes générées lors de la définition de la classe en question. Dans un souci de concision, ces contraintes ne seront pas présentées dans ces annexes, car particulièrement simples.

Accès au champ statique *name* d'une classe *c'* (getstatic)

```
stack(c, m, pc, Ctx, [0' | S]) :-
    static(c', name, 0'),
    stack(c, m, pc-1, Ctx, S).
```

Si le champ contient des objets de type `long` ou `double`, deux mots seront placés sur la pile.

Modification du champ statique name d'une classe c' (putstatic)

```
static(c', name, 0') :-
    stack(c, m, pc-1, Ctx, [0' | S])

stack(c, m, pc, ctx, S) :-
    stack(c, m, pc-1, Ctx, [0' | S]).
```

Si le champ contient des objets de type long ou double, deux mots devaient se trouver sur la pile.

Accès au champ name d'un objet (getfield)

```
stack(c, m, pc, Ctx, [0' | S]) :-
    field(0, name, 0'),
    stack(c, m, pc-1, Ctx, [0 | S]).
```

Si le champ contient des objets de type long ou double, deux mots seront placés sur la pile.

Modification du champ name d'un objet (putfield)

```
field(0, name, 0') :-
    stack(c, m, pc-1, Ctx, [0', 0 | S])

stack(c, m, pc, Ctx, S) :-
    stack(c, m, pc-1, Ctx, [0', 0 | S]).
```

Si le champ contient des objets de type long ou double, deux mots devaient se trouver sur la pile.

A.1.5 Définitions, appels et retours de méthodes**Définition d'une méthode virtuelle m au sein d'une classe c**

La méthode attend en paramètre la référence This de l'objet, ainsi que n arguments.

```
define(c, m).

sig(m).

stack(c, m, 0, Ctx, []) :- BODY.

self(c, m, Ctx, This) :- BODY.

locals(c, m, 0, Ctx, [This, O1, ..., On, ..., _]) :- BODY.
```

La notation BODY représente le corps de clause suivant :

```
call(C', M', Pc', Ctx', c, m, [This, O1, ..., On], Ctx),
PREDI
```

La notation PREDI représente l'ensemble des prédicats pour i de 1 à n tels que :

– si l'argument i est un type t_i prédéfini :

- $O_i = t_i$
- si l'argument i est une classe `classi` différente de `java.lang.Object` :
`classOf(Oi, Ci), subclassStar(Ci, classi)`
- si l'argument i est la classe `java.lang.Object` :
`classOrArrayOf(Oi, Ci, Ni),`
- si l'argument i est un tableau de type prédéfini `arraytypei` et de dimension `depthi` :
`arrayOf(Oi, arraytypei, depthi)`
- si l'argument i est un tableau de classe `arrayclassi` et de dimension `depthi` :
`arrayOf(Oi, Ci, depthi), subclassStar(Ci, arrayclassi),`

Définition d'une méthode statique m au sein d'une classe c

La méthode attend en paramètre n arguments.

```
define(c, m).
```

```
sig(m).
```

```
stack(c, m, 0, Ctx, []) :- BODY.
```

```
self(c, m, Ctx, This) :- BODY.
```

```
locals(c, m, 0, Ctx, [O1, ..., On]) :- BODY.
```

La notation `BODY` représente le corps de clause suivant :

```
call(C', M', Pc', Ctx', c, m, [O1, ..., On], Ctx),
objInit(This),
PREDI.
```

La notation `PREDI` fait référence à celle définie dans le paragraphe précédent.

Appel d'une méthode non statique f (`invokevirtual`, `invokespecial`)

```
call(c, m, pc, Ctx, C', f, [This, O1, .., On], Ctx') :-
  stack(c, m, pc-1, Ctx, [On, ..., O1, This | S],
  methCtx(c, m, pc, [This, O1, ..., On], Ctx, Ctx'),
  classOf(This, C''),
  lk(C'', f, C').
```

```
stack(c, m, pc, Ctx, [O | S] :-
  ret(C', f, O, Ctx'),
  stack(c, m, pc-1, Ctx, [On, ..., O1, This | S],
  methCtx(c, m, pc, [This, O1, ..., On], Ctx, Ctx'),
  classOf(This, C''),
  lk(C'', f, C').
```

À noter que l'objet `O` peut être composé de 0, 1 ou 2 mot(s) selon le type de retour attendu.

Par ailleurs, dans le cas d'un appel spécial (`invokespecial`), les prédicats `classOf(This, C'')` et `lk(C'', f, C')` ne sont pas nécessaires puisque l'on connaît déjà la classe `C'` dont on compte appeler la méthode `f`.

Appel d'une méthode statique f d'une classe c' (invokestatic)

```
call(c, m, pc, Ctx, c', f, [O1, ..., On], Ctx') :-
    stack(c, m, pc-1, Ctx, [On, ..., O1 | S]),
    methStaticCtx(c, m, pc, [O1, ..., On], Ctx, Ctx')

stack(c, m, pc, Ctx, [O | S] :-
    ret(c', f, O, Ctx'),
    stack(c, m, pc-1, Ctx, [On, ..., O1 | S]),
    methStaticCtx(c, m, pc, [O1, ..., On], Ctx, Ctx')
```

À noter que l'objet `O` peut être composé de 0, 1 ou 2 mot(s) selon le type retourné.

Retour de méthode (ireturn, return, etc.)

```
ret(c, m, O, Ctx) <-
    stack(c, m, pc-1, Ctx, [O | S]),
    PRED
```

La notation `PRED` permet de personnaliser la contrainte en fonction du type d'objet dont le retour est attendu :

- si le type de retour est vide :
 `O = <rien>`
- si le type de retour est un type `ti` prédéfini :
 `O = ti`
- si le type de retour est une classe `C'` différente de `java.lang.Object` :
 `classOf(O, C'')`, `subclassStar(C'')`, `C'`
- si le type de retour est la classe `java.lang.Object` :
 `classOrArrayOf(O, C', N')`
- si le type de retour est un tableau de type prédéfini `arraytype` et de dimension `depth` :
 `arrayOf(O, arraytype, depth)`
- si le type de retour est un tableau de classe `arrayclass` et de dimension `depth` :
 `arrayOf(O, C', depth)`, `subclassStar(C', arrayclass)`

On notera que, contrairement à toutes les autres instructions, on ne propage pas la valeur des variables locales et de la pile au point de programme suivant car il n'existe pas de lien entre ces deux points de programmes.

Cas des méthodes natives

Le cas des méthodes natives doit être considéré séparément : ces méthodes appellent du code C^{++} et il n'est en aucun cas envisageable d'aller effectuer une analyse de ce code, tout à fait hors des objectifs de ce projet. C'est pourquoi, lors de la définition d'une méthode native, les seules contraintes générées donneront uniquement les éléments minima permettant de poursuivre l'analyse — en l'occurrence la définition et le retour de l'objet attendu si la méthode a été appelée.

```
define(c, m).
```

```
sig(m).
```

```
ret(c, m, O, Ctx) <-
    call(C', M', Pc', Ctx', c, m, [This, O1, ..., On], ctx),
```

```

objCtx(c, m, 0, c'', ctx, This, 0)

object(0) <-
  call(C', M', Pc', Ctx', c, m, [This, O1, ..., On], ctx),
  objCtx(c, m, 0, c'', ctx, This, 0)

```

Il faudra ajouter au corps de ces deux dernières clauses les prédicats — similaires à PRED et PREDI définis plus haut — permettant de contraindre 0 à avoir le type de retour `c''` — ou une éventuelle sous-classe — de la méthode native; de même pour le type des arguments `[This, O1, ..., On]` de cette méthode.

A.1.6 Instructions relatives aux tableaux

Création de tableau (`anewarray`, `multianewarray`, etc.)

```

stack(c, m, pc, Ctx, [Array | S]) :-
  stack(c, m, pc-1, Ctx, [int, int..., int | S]), /* array_depth times int */
  self(c, m, Ctx, This),
  arrayCtx(c, m, pc, array_class, array_depth, Ctx, This, Array).

```

```

array(Array) :-
  stack(c, m, pc-1, Ctx, [int, int..., int | S]), /* array_depth times int */
  self(c, m, Ctx, This),
  arrayCtx(c, m, pc, array_class, array_depth, Ctx, This, Array).

```

`array_depth` représente la dimension du tableau, `array_class` sa classe ou son type.

En cas de tableau de type prédéfini, le tableau sera initialisé avec son type. En cas de tableau d'objets, il conviendra de l'initialiser à `null`, comme ci-dessous.

```

inArray(null, Array) :-
  stack(c, m, pc-1, Ctx, [int, int..., int | S]), /* array_depth times int */
  self(c, m, Ctx, This),
  arrayCtx(c, m, pc, array_class, array_depth, Ctx, This, Array).

```

En cas de tableau multidimensionnel, il conviendra d'effectuer des initialisations récursives des tableaux imbriqués. Par exemple, pour un tableau de dimension 2 de type `int`, les contraintes suivantes seront ajoutées à la contrainte décrite en début de paragraphe.

```

inArray(A1, Array) :-
  stack(c, m, pc-1, Ctx, [int, int..., int | S]), /* array_depth times int */
  self(c, m, Ctx, This),
  arrayCtx(c, m, pc, 2, int, Ctx, This, Array),
  arrayCtx(c, m, pc, 1, int, Ctx, This, A1).

```

```

array(A1) :-
  stack(c, m, pc-1, Ctx, [int, int..., int | S]), /* array_depth times int */
  self(c, m, Ctx, This),
  arrayCtx(c, m, pc, 2, int, Ctx, This, Array),
  arrayCtx(c, m, pc, 1, int, Ctx, This, A1).

```

```

inArray(int, A1) :-

```

```

stack(c, m, pc-1, Ctx, [int, int..., int | S]), /* array_depth times int */
self(c, m, Ctx, This),
arrayCtx(c, m, pc, int, 1, Ctx, This, A1).

```

Récupération d'une donnée depuis un tableau (aaload, laload, etc.)

```

stack(c, m, pc, Ctx, [0 | S] :-
    stack(c, m, pc-1, Ctx, [int, Array | S]),
    inArray(0, Array).

```

Si le tableau contient des objets de type long ou double, l'objet 0 occupera deux mots sur la pile.

Placement d'une donnée dans un tableau (iastore, fastore, etc.)

```

stack(c, m, pc, Ctx, S) :-
    stack(c, m, pc-1, Ctx, [0, int, Array | S]),
    elemOfArray(0, Array).

```

```

inArray(0, Array) :-
    stack(c, m, pc-1, Ctx, [0, int, Array | S]),
    elemOfArray(0, Array).

```

Si le tableau contient des objets de type long ou double, l'objet 0 occupera deux mots sur la pile.

A.1.7 Exceptions

Lancement d'une exception (athrow)

```

throw(c, m, pc, E) :-
    stack(c, m, pc-1, Ctx, [E | S]).

```

Contraintes définies globalement

Quelques prédicats globaux sont également nécessaires pour la gestion des exceptions.

```

throw(C, M, Pc, E) :-
    call(C, M, Pc, _, C', M', _, _),
    escape(C', M', E).

```

```

escape(C, M, E) :-
    throw(C, M, Pc, E),
    classOf(E, Ec),
    not(catchable(C, M, Pc, Ec)).

```

```

catchable(C, M, Pc, Ec) :-
    throw(C, M, Pc, _),
    subclassStar(Ec, Ec'),
    handle(C, M, From, To, _, Ec'),
    in(Pc, From, To).

```

```

catch(C, M, Pc, Ctx, E) :-
    throw(C, M, Pc, Ctx, E),
    classOf(E, Ec),
    subclassStar(Ec, Ecprime),
    handle(C, M, From, To, _, Ecprime),
    in(Pc, From, To).

in(A, B, C) :-
    B =< A, C >= A.

active(C, M, Goto, E) :-
    throw(C, M, Pc, E),
    classOf(E, Ec),
    subclassStar(Ec, Ecprime),
    handle(C, M, From, To, Goto, Ecprime),
    in(Pc, From, To),
    tnot(handleBefore(C, M, Pc, Ec, Goto)).
active(C, M, Goto, E) :-
    handle(C, M, _, To, Goto, _),
    To =< Goto,
    stack(C, M, To, Ctx, [E]).

handleBefore(C, M, Pc, Ec, Goto) :-
    throw(C, M, Pc, E),
    classOf(E, Ec),
    handle(C, M, From1, To1, Goto, Ec1),
    handle(C, M, From2, To2, Goto2, Ec2),
    subclassStar(Ec, Ec1),
    in(Pc, From2, To2),
    in(Pc, From1, To1),
    subclassStar(Ec, Ec2),
    Goto2 < Goto.

```

On notera que le `tnot` dans la définition du prédicat `active` est utilisé à la place du `not` car le prédicat nié (`handleBefore`) est tabulé par XSB.

A.1.8 Appel du programme

main

```

callMeth(0, 0, 0, Ctx, _, main/1, [0'], Ctx') :-
    arrayCtx(0, 0, 0, java.lang.String, 1, Ctx, 0, 0'),
    initObj(0),
    methStaticCtx(0, 0, 0, [0], Ctx, Ctx'),
    initCtx(Ctx).

array(0') :-
    arrayCtx(0, 0, 0, java.lang.String, 1, Ctx, 0, 0'),
    initObj(0),
    methStaticCtx(0, 0, 0, [0], Ctx, Ctx'),

```

```
initCtx(Ctx).
```

A.2 Contraintes relatives à l'analyse

Analyse 0-CFA

```
objCtx(C, M, I, C', Ctx, C, C').
arrayCtx(C, M, I, C', N', Ctx, C, [C', N']).
methCtx(C, M, I, Arg, Ctx, ctx).
methStaticCtx(C, M, I, Arg, Ctx, ctx).
classOf(C, C).
arrayOf([C, N], C, N).
initObj('0').
initCtx('0').
```

Analyse 1/2-CFA

```
objCtx(C, M, I, C', Ctx, C, C').
arrayCtx(C, M, I, C', N', Ctx, C, [C', N']).
methCtx(C, M, I, [This | Arg], Ctx, This).
methStaticCtx(C, M, I, Arg, Ctx, Ctx') :- initCtx(Ctx').
classOf(C, C).
arrayOf([C, N], C, N).
initObj('0').
initCtx('0').
```

Analyse k -l-CFA

```
objCtx(C, M, I, C', Ctx,
      [C'', [P1, ..., P1]],
      [C', [[C, M, I], P1, ..., P1-1]]).
arrayCtx(C, M, I, C', N', Ctx,
      [C'', [P1, ..., P1]],
      [C', N', [[C, M, I], P1, ..., P1-1]]).
methCtx(C, M, I, Arg, [P1, ..., Pk], [[C, M, I], P1, ..., Pk-1]).
methStaticCtx(C, M, I, Arg, [P1, ..., Pk], [[C, M, I], P1, ..., Pk-1]).
classOf([C, L], C) :- object([C, L]).
arrayOf([C, N, L], C, N) :- array([C, N, L]).
initObj('0', 0, [['0', '0', 0], ..., ['0', '0', 0]]). /* 1 fois */
initCtx(['0', '0', 0], ..., ['0', '0', 0]). /* k fois */
```

Algorithme du produit cartésien

```
objCtx(C, M, I, C', Ctx, 0, [C', [C, M, I]]).
arrayCtx(C, M, I, C', N', Ctx, 0, [C', N', [C, M, I]]).
methCtx(C, M, I, Arg, Ctx, Arg).
methStaticCtx(C, M, I, Arg, Ctx, Arg).
classOf([C, L], C) :- object([C, L]).
arrayOf([C, N, L], C, N) :- array([C, N, L]).
initObj('0').
initCtx('0').
```

A.3 Contraintes d'ordre général

```

lk(C,M,C) :- define(C,M).
lk(C1,M,C3) :- notDefine(C1,M), subclass(C1,C2), lk(C2,M,C3).
notDefine(C,M) :- class(C), sig(M), not(define(C,M)).

subclassStar(C, C) :- class(C).
subclassStar(C1, C3) :- subclass(C1,C2), subclassStar(C2, C3).

classOrArrayOf(O, C, N) :- classOf(O, C).
classOrArrayOf(O, C, N) :- arrayOf(O, C, N).

classOf('null', C) :- class(C).
arrayOf('null', C, _) :- class(C).

elemOfArray(Type, Array) :-
    arrayOf(Array, Type, 1),
    predefined(Type).

elemOfArray(O, Array) :-
    arrayOf(Array, Aclass, 1),
    classOf(O, C'),
    subclassStar(C', Aclass).

elemOfArray(O, Array) :-
    arrayOf(Array, Type, Adepth),
    arrayOf(O, Type, Adepth-1),
    predefined(Type).

elemOfArray(O, Array) :-
    arrayOf(Array, Aclass, Adepth),
    arrayOf(O, C', Adepth-1),
    subclassStar(C', Aclass).

```

Annexe B

Résultats intéressants

Cette section présente quelques résultats susceptibles de concrétiser l'analyse effectuée par Toutânkhamel. Généralement, ceux-ci seront présentés sous forme d'une ou plusieurs classes dont le code sera donné, ainsi que les graphiques générés et, dans la mesure du possible, quelques indications pour la compréhension.

B.1 Chaîne d'éléments

La classe `ChaineElem` permet de représenter une chaîne basique à partir de ses éléments. Chaque objet ainsi chaîné a un successeur `_suivant`, et la classe référence le premier élément via le champ statique `_premier`.

B.1.1 Code source

```
public class ChaineElem{
    public static ChaineElem _premier;
    private ChaineElem _suivant;

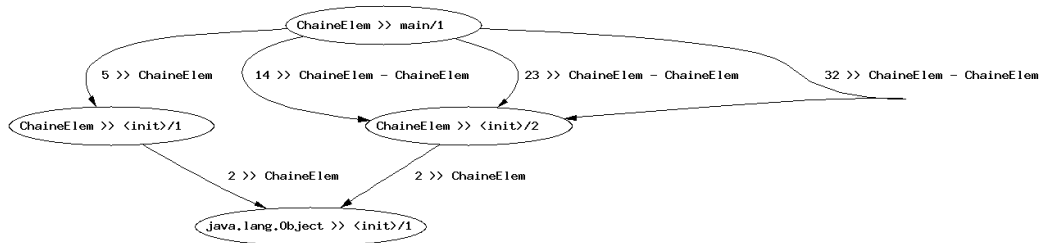
    public ChaineElem(){
        _premier = this;
        _suivant = null;
    }

    public ChaineElem(ChaineElem suivant){
        _suivant = suivant;
    }

    public static void main(String argv[]){
        ChaineElem e1 = new ChaineElem();
        ChaineElem e2 = new ChaineElem(e1);
        ChaineElem e3 = new ChaineElem(e2);
        ChaineElem e4 = new ChaineElem(e3);
        _premier = e1;
    }
}
```

La hiérarchie de classes étant triviale, le diagramme ne sera pas représenté.

B.1.2 Graphe d'appels



B.1.3 Diagrammes d'objets

Analyse 0-CFA

Le diagramme d'objets correspondant à l'analyse 0-CFA est représenté sur la figure 2.13 page 31.

Analyse 2-2-CFA

Le diagramme d'objets de l'analyse 2-2-CFA est représenté sur la figure 2.14 page 32.

Quelques éléments de compréhension quant à ces diagrammes sont également donnés dans le corps du rapport dans la section 2.5.3 page 31.

B.2 Exceptions

Lors de l'existence d'exceptions dans le programme, celles-ci sont représentées en rouge sur le graphe d'appel des méthodes.

Une flèche allant d'une méthode à une autre indique qu'une exception est *susceptible* d'être lancée à cet endroit. Une boucle sur une méthode indique qu'une exception est attrapée dans cette méthode.

L'existence d'une ou plusieurs flèches (et donc exceptions) aboutissant à l'ellipse `0 » 0` (contexte « vide » appelant le programme) indique que le programme est susceptible d'être brutalement arrêté en raison d'une exception non attrapée.

B.2.1 Exemple de graphe d'appels

L'exemple présenté en figure B.1 est le résultat de l'analyse pour une seule classe `Exc2`, faisant appel à plusieurs autres classes préanalysées des bibliothèques Java. Ainsi, la méthode `onemethod` peut lancer une exception de classe `java.lang.Exception` mais, si tel est le cas, elle sera attrapée dans `Exc2 » main`. En revanche, une exception `Exc2` peut être lancée depuis le `main` et causer l'arrêt du programme.

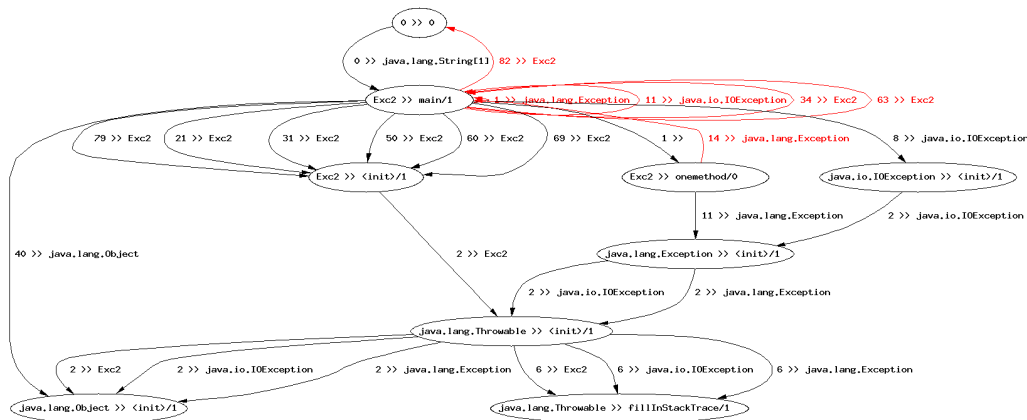


FIG. B.1 – Exemple de graphe d’appels avec exceptions

B.3 Comparaison entre analyses globale et modulaire

Bien que la mesure du gain de temps entre analyse globale et analyse modulaire ait été réalisée juste avant de terminer ce rapport, il convient d’en présenter un exemple.

Les quatre classes présentées ci-après sont tout d’abord analysées entièrement. Il s’agit donc d’une analyse globale puisque l’on connaît l’ensemble des classes au moment de l’analyse. Ensuite, on ne réanalyse qu’une seule classe, et les contraintes correspondant aux autres classes sont réutilisées.

B.3.1 Code source

Classe Saucisse.java

```
public interface Saucisse{
    public void retourne();
    public boolean brulee();
    public boolean mangeable();
    public void grille(int temps);
}
```

Classe Merguez.java

```
public class Merguez implements Saucisse{
    /* Degre de piment */
    private int _piment;
    /* L’avancement de la grillade */
    protected int _avancementGrille;
    /* Vitesse ou la saucisse grille */
    private static final int vitesseGrille = 3;
    /* Seuil de grillade a partir duquel la saucisse est mangeable */
    private static final int seuilMangeable = 15;
    /* Seuil de grillade au dela duquel la saucisse est brulee */
    private static final int seuilBrule = 20;
```

```

public Merguez(int piment){
    _avancementGrille = 0;
    _piment = piment;
}

public void retourne(){}

public boolean brulee(){
    return _avancementGrille > seuilBrule;
}

public boolean mangeable(){
    return _avancementGrille >= seuilMangeable;
}

public void grille(int temps){
    _avancementGrille += temps * vitesseGrille;
}
}

```

Classe Chipo.java

```

public class Chipo implements Saucisse{
    /* L'avancement de la grillade */
    protected int _avancementGrille;
    /* Vitesse ou la saucisse grille */
    private static final int vitesseGrille = 2;
    /* Seuil de grillade a partir duquel la saucisse est mangeable */
    private static final int seuilMangeable = 10;
    /* Seuil de grillade au dela duquel la saucisse est brulee */
    private static final int seuilBrule = 15;

    public Chipo(){
        _avancementGrille = 0;
    }

    public void retourne(){}

    public boolean brulee(){
        return _avancementGrille > seuilBrule;
    }

    public boolean mangeable(){
        return _avancementGrille >= seuilMangeable;
    }

    public void grille(int temps){
        _avancementGrille += temps * vitesseGrille;
    }
}

```

```
    }  
}
```

Classe Barbecue.java

```
public class Barbecue{  
    /* La grille du barbecue */  
    private Saucisse[] _grille;  
    private int _nbGrille;  
    /* L'assiette pour servir */  
    private Saucisse[] _assiette;  
    private int _nbAssiette;  
    /* La poubelle */  
    private Saucisse[] _poubelle;  
    private int _nbPoubelle;  
  
    public Barbecue(int nb_saucisses){  
        _grille = new Saucisse[nb_saucisses];  
        _assiette = new Saucisse[nb_saucisses];  
        _poubelle = new Saucisse[nb_saucisses];  
        _nbGrille = 0;  
        _nbAssiette = 0;  
        _nbPoubelle = 0;  
    }  
  
    /* Ajoute une saucisse sur la grille */  
    public boolean ajouteSaucisse(Saucisse s){  
        if(_nbGrille < _grille.length){  
            _grille[_nbGrille] = s;  
            _nbGrille++;  
            return true;  
        }else{  
            return false;  
        }  
    }  
  
    /* Fait griller les saucisses */  
    public void grille(int temps){  
        for(int i = 0; i < _nbGrille; i++){  
            _grille[i].grille(temps);  
        }  
    }  
  
    /* Retourne les saucisses */  
    public void retourne(){  
        for(int i = 0; i < _nbGrille; i++){  
            _grille[i].retourne();  
        }  
    }  
}
```

```
/* Trie les saucisses entre mangeables et brulees */
public void trie(){
    for(int i = 0; i < _nbGrille; i++){
        /* Cas : saucisse brulee */
        if(_grille[i].brulee()){
            _poubelle[_nbPoubelle] = _grille[i];
            _nbPoubelle++;
        /* Decalage des autres */
        for(int j = i; j < _nbGrille - 1; j++){
            _grille[j] = _grille[j+1];
        }
        _nbGrille--;
        /* Cas : saucisse cuite */
        }else if(_grille[i].mangeable()){
            _assiette[_nbAssiette] = _grille[i];
            _nbAssiette++;
        /* Decalage des autres */
        for(int j = i; j < _nbGrille - 1; j++){
            _grille[j] = _grille[j+1];
        }
        _nbGrille--;
    }
}

/* Vide l'assiette */
public void mange(){
    _nbAssiette = 0;
}

/* Vide la poubelle */
public void videPoubelle(){
    _nbPoubelle = 0;
}

public static void main(String[] args){
    Barbecue barbecue = new Barbecue(4);
    barbecue.ajouteSaucisse(new Merguez(5));
    barbecue.ajouteSaucisse(new Chipo());
    barbecue.ajouteSaucisse(new Merguez(4));
    barbecue.grille(2);
    barbecue.retourne();
    barbecue.ajouteSaucisse(new Chipo());
    barbecue.grille(2);
    barbecue.trie();
    barbecue.mange();
    barbecue.grille(25);
    barbecue.trie();
}
```

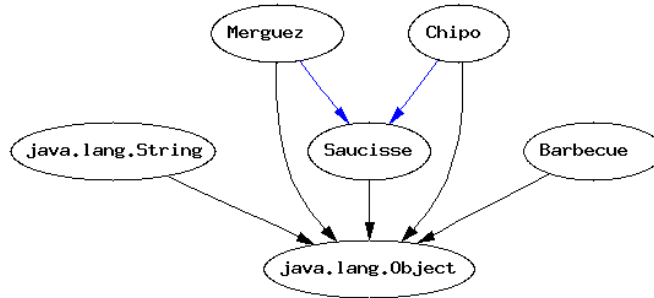
```

    barbecue.videPoubelle();
}
}

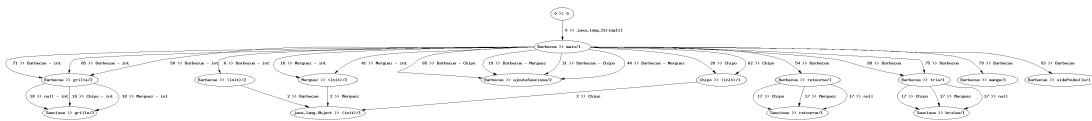
```

Cet exemple est détaillé dans le corps du rapport dans la section 2.5.4 à la page 33. La comparaison chiffrant le gain de temps apporté par une analyse modulaire dans le cas d'une modification de la classe `Saucisse.java` est présenté sur la figure 2.16 page 35.

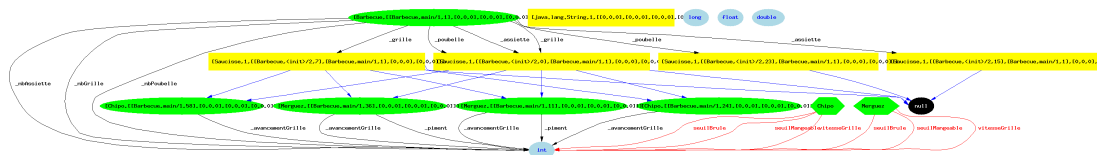
B.3.2 Hiérarchie de classes



B.3.3 Diagramme d'appels



B.3.4 Diagramme d'objets



On notera que les diagrammes standards générés par Toutânkhamel peuvent rapidement devenir illisibles en cas de programmes à la taille relativement conséquente. L'utilisateur peut, à cet effet, générer ses propres diagrammes en extrayant les seules informations l'intéressant.