Université de Montréal

Inférence statique et par contraintes des relations de composition dans des programmes Java

par
Norddin Habti

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Janvier, 2009

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Inférence statique et par contraintes des relations de composition dans des programmes Java

présenté par :

Norddin Habti

a été évalué par un jury composé des personnes suivantes :

Julie Vachon
président-rapporteur

Stefan Monnier
directeur de recherche

Yann-Gaël Guéhéneuc
codirecteur

Victor Ostromoukhov
membre du jury

Mémoire accepté le

RÉSUMÉ

La relation de composition est une relation entre deux classes qui nécessite qu'une instance d'une classe (appelée partie ou composant) soit incluse, au plus, dans *une* instance d'une autre classe (appelée tout ou objet composite) à un moment donné. Cette nécessité impose des contraintes sur le lien qui unit le tout à ses parties ; elles concernent principalement les propriétés d'exclusivité (non partageabilité) et de dépendance (simultanéité) de vie des parties par rapport au tout.

Les langages de modélisation par objets permettent diverses sémantiques de la relation tout–partie, cependant les principaux langages de programmation orientés objets n'offrent pas une syntaxe pour représenter de façon précise et explicite une relation entre un tout et ses parties. De plus, il n'existe aucun outil efficace qui peut vérifier et valider les relations de composition dans un programme. La plupart des outils d'*ingénierie inverse* échouent à identifier statiquement ces relations ; généralement, ils se limitent à utiliser la déclaration d'un champ pour inférer une relation tout–partie. Il y a donc une rupture entre la modélisation et l'implémentation et, par conséquent, il est difficile de vérifier si les exigences définies lors de la phase de spécification ont été respectées durant la phase d'implémentation.

Dans ce mémoire, nous présentons une nouvelle approche pour définir la relation de composition dans un langage de programmation orienté objets ; nous définissons des propriétés qui nous permettent de formaliser la relation entre un tout et ses parties directement dans le langage de programmation. Nous avons implémenté notre approche pour identifier statiquement les relations de composition dans des programmes Java. Notre implémentation est basée sur des contraintes que nous avons établies pour contrôler l'utilisation des références et ainsi pouvoir vérifier la présence des relations de composition.

Mots clés: Composition, ingénierie inverse, langages de modélisation, Java.

ABSTRACT

Composition is a strong form of whole–part relationship between two classes that requires that an object of one class (known as *component* or *part*) be included in at most *one* instance of some other class (known as *composite* or *whole*) at a time. This restriction imposes constraints on the link between the whole and its parts; those constraints concern the exclusivity and lifetime dependency of the parts with respect to the whole.

Modeling languages provide various semantics of the whole–part relationship; however there is no corresponding facility in standard programming language. Moreover, there is no effective tool that can identify and verify composition relationships among classes in a program consistently. Most reverse engineering tools fail to identify statically these relationships. Usually, they simply infer a whole–part relationship using instance fields. Therefore, there is a gap between design and code, and it is difficult to verify whether the requirements defined in the analysis and design phases have been respected during the implementation phase.

In this thesis, we propose a formal definition of the composition relationship; we define properties that allow us to formalize the relationship between a whole and its parts directly in the Java programming language. To validate our approach, we have implemented our definition and we identify statically composition relationships in Java programs using a constraint-based system. The constraints enable the program users to better understand the results and thus adjust the program or add annotations in the code to make explicit the composition relationships and their rationale.

Keywords: Composition, reverse engineering, modeling languages, Java.

TABLE DES MATIÈRES

À mes parents et ma femme.

REMERCIEMENTS

J'aimerais tout d'abord remercier mes directeurs Stefan Monnier et Yann-Gaël Guéhéneuc pour leur soutien et aide précieuse tout au long de cette maitrise ainsi que pour le temps qu'ils ont consacré à la correction des versions préliminaires de ce mémoire.

Je tiens à remercier également tous les membres du laboratoire GEODES.

Je remercie aussi ma femme qui n'a jamais cessé de m'encourager et de m'appuyer.

Enfin, je remercie les membres de mon jury, Julie Vachon et Victor Ostromoukhov pour avoir accepté d'évaluer ce travail et pour leurs commentaires constructifs.

CHAPITRE 1

INTRODUCTION

1.1 Contexte : relation de composition, langages de programmation orientés objets, ingénierie inverse

Les langages de programmation orientés objets, tels que Java, offrent des syntaxes et des sémantiques pour représenter les différents concepts du paradigme orienté objet (classes, objets, encapsulation, héritage, polymorphisme). Cependant, ils n'offrent pas une syntaxe explicite pour représenter les relations entre les classes et, en particulier, représenter les relations tout–partie ; typiquement, l'implémentation de ces relations est faite en déclarant dans le tout un *champ* du type de la partie. Ainsi, il est impossible, à partir de la déclaration d'un champ, de savoir s'il s'agit d'une relation d'agrégation ou de composition. Cela crée des incompatibilités et des écarts entre les programmes et leurs modèles (e.g., diagrammes de classes UML) construits avec des outils d'ingénierie inverse et, par ailleurs, cela crée une rupture entre la phase de modélisation et la phase d'implémentation [GAA04].

Distinguer le type d'une relation entre deux classes et en particulier identifier statiquement les relations de composition constitue un défi très particulier pour les chercheurs ; la relation de composition définit des propriétés et des contraintes précises, notamment l'unicité d'appartenance (non partageabilité) et la dépendance existentielle (simultanéité de vie et de mort) entre un tout et ses parties. Une mauvaise implémentation de la relation de composition peut aller à l'encontre des exigences définies lors de la phase de spécification, par exemple sur le comportement fonctionnel qui a été définie pour un programme lors de la phase de modélisation ou sur les aspects de sécurité qui concerne les droits d'accès aux objets du programme (comme le bogue *signers* en Java 1.1 [Mil05]).

Le principal problème qui empêche l'identification des relations de composition dans un programme réside dans l'absence de concept et de constructions propres à la définition et à la déclaration de relations de composition dans les langages de programmation ; les

langages de programmation permettent la déclaration d'un champ (simple, tableau, ou collection) pour "exprimer" une relation tout–partie, mais n'offrent aucune syntaxe pour spécifier les contraintes à imposer sur l'utilisation de ce champ. Il revient aux programmeurs de s'assurer que les parties seront exclusives au tout et qu'elles seront détruites aussitôt que le tout sera détruit.

1.2 Objectif : proposer une définition de la relation de composition qui soit proche du code source

Notre objectif est d'analyser la sémantique de la relation de composition et d'étudier les différentes formes de son implémentation dans les programmes Java pour pouvoir formuler des propriétés et des contraintes qui nous permettent de l'identifier statiquement dans des programmes Java. Nous distinguons deux problèmes, le premier concerne la définition de la relation de composition dans un langage de programmation et le deuxième concerne l'identification et la validation de la relation de composition entre deux objets dans un programme donné. Dans notre étude, nous proposons une définition de la relation de composition et nous présentons un algorithme, basé sur cette définition, pour identifier statiquement les relations de composition dans des programmes Java. Nous estimons que notre définition est simple, abstraite et indépendante de la syntaxe du langage de programmation ; nous présentons une solution qui est directement liée aux problèmes d'implémentation des relations de composition dans le code source et nous proposons un algorithme pour identifier statiquement ces relations. Cependant, nous ne nous concentrons pas sur les problèmes liés aux différentes sémantiques de la relation de composition.

1.3 Contributions : une nouvelle approche pour définir et identifier statiquement les relations de composition dans des programmes Java

Il est difficile, par une analyse statique ou dynamique, de vérifier et de valider d'une manière sûre que la relation de composition définie entre deux classes sera respectée dans le programme. Une solution "simple" consiste à s'assurer que la création des parties

est faite par des méthodes définies dans le tout et à limiter l'accès de ces méthodes au tout (implicitement, les parties sont complètement englobées par le tout). Il suffit dans ce cas d'annoter le champ concerné par la relation de composition comme étant un composant et on peut vérifier statiquement que la relation de composition est valide pour toutes les exécutions possibles du programme. Cette solution est très contraignante et élimine beaucoup de cas très utilisés dans la pratique. Par exemple, une partie n'est pas nécessairement créée par son tout ; elle peut être créée ailleurs. Aussi, en pratique, on accepte l'accès direct à une partie pour accéder à ses champs et invoquer ses méthodes. C'est pourquoi nous étudions la relation de composition pour lui donner une définition proche de l'implémentation. L'apport principal de notre définition est qu'elle offre une grande flexibilité pour l'intégrer dans le langage de programmation, par exemple sous forme d'annotations : en plus de l'avantage que les résultats de l'analyse peuvent être expliqués et justifiés facilement par les programmeurs, ces derniers peuvent modifier directement le code pour ajouter des annotations pour explicitement indiquer qu'une référence correspond à une relation de composition par exemple.

1.4 Organisation du mémoire

Le reste de ce mémoire est organisé comme suit : le chapitre 2 présente un état de l'art des principaux études et travaux connexes qui ont proposé des définitions et des implémentations pour identifier la relation de composition. Le chapitre 3 présente notre définition de la relation de composition ; nous expliquons avec des exemples les différents cas de programmes où une relation de composition peut être identifiée. Nous étudions dans le chapitre 4 les différentes contraintes que nous imposons sur l'utilisation des références pour pouvoir identifier et vérifier la relation de composition. Le chapitre 5 présente notre implémentation et le chapitre 6 discute des résultats de nos tests ainsi que des limitations observées. Enfin, nous concluons ce mémoire et proposons de futures directions de recherche.

CHAPITRE 2

ÉTAT DE L'ART

2.1 Les relations entre les classes : association, agrégation et composition

Dans UML, les relations entre les classes sont une abstraction des liens qui lient les instances de ces classes. Ces liens offrent la possibilité aux objets de communiquer entre eux ; un objet lié à un autre objet peut accéder à ses champs et invoquer ses méthodes. Chaque type de relations entre deux classes définit des propriétés précises sur les liens qui associent les objets concernés et peut ainsi limiter leurs interactions avec les autres objets du programme :

- la relation d'association est la forme de base pour définir une relation entre deux classes ; elle exprime la capacité d'un objet d'une classe à envoyer un message à objet de l'autre classe ;
- la relation d'agrégation est une forme particulière de la relation d'association qui exprime un couplage entre les classes ; elle permet de représenter des relations tout–partie [Mul97] ;
- la relation de composition est une forme particulière de la relation d'agrégation où une partie doit appartenir au plus à un tout. Un tout est responsable de la destruction de ses parties quand il est lui-même détruit [Gro07].

2.1.1 Définition de la relation de composition dans UML

Ainsi, selon la spécification UML 2.1.2 [Gro07] :

> Une relation de composition est une forme très forte de la relation d'agrégation qui nécessite que l'instance d'une partie soit incluse au plus dans un tout à un moment donné. Si le tout est supprimé, toutes ses parties sont normalement supprimées avec lui. Une partie peut (là où permis) être enlevée du tout avant

que celui-ci ne soit supprimé et ainsi ne pas être supprimée comme partie du tout.

Cette définition se focalise sur les objets et nous l'exprimons autrement comme suit : une classe `T` est en relation de composition avec une classe `P` si chaque instance de la classe `T` détient, à un moment donné, l'unique référence vers une instance de la classe `P`. Si l'instance de la classe `T` est supprimée alors l'instance de la classe `P` doit être aussi supprimée.
Les aspects importants de cette définition sont :

- la partie peut être liée à d'autres objets mais pas à un autre tout au même moment : par exemple, une voiture (*tout*) possède quatre roues (*parties*) et il est impossible qu'une roue soit partagée entre deux voitures mais une roue peut être enlevée temporairement de la voiture pour des raisons d'entretien ;

- la partie peut être détachée du tout : par exemple, les roues d'une voiture peuvent être enlevée d'une voiture et attachée à une autre voiture. Cet aspect est délicat à implémenter et nous n'en proposons pas une solution dans notre implémentation.

Booch et al. [BRJ98] soulignent de plus que les parties peuvent être créées après la création du tout ; de même les parties peuvent être explicitement enlevées avant la destruction du tout. Nous mentionnons que nous n'essayons pas, dans ce mémoire, de prendre position par rapport aux divergences que peuvent poser ces particularités.

2.1.2 Exemple d'implémentation de la relation de composition en Java

Typiquement, l'implémentation, avec Java, de la relation tout–partie est faite en déclarant, dans la classe qui représente le tout, un champ (simple, tableau ou collection) dont le type est celui de la classe qui représente la partie. Une relation tout–partie entre les classes `T` et `P` est implémentée dans l'exemple suivant :

```
1 class P {}
2 class T {
3     private P p = new P();
4     ...
5 }
```

Cependant, cette déclaration est insuffisante pour pouvoir affirmer une relation de composition entre les classes `T` et `P`. Nous pouvons inférer une relation de composition entre les classes `T` et `P` si et seulement si :

- les méthodes définies dans `T` utilisent le champ p uniquement pour invoquer directement les méthodes de `P` ; sans retourner ou passer en paramètre la référence sur `P` à un autre contexte ;
- lorsque le modificateur du champ p est de type protected ou default, la condition ci-dessus n'est violée par aucune des sous–classes de `T` qui redéfinissent ses méthodes ou définissent de nouvelles méthodes qui utilisent ce champ.

Dans les autres cas, par exemple, où la référence de `P` est passée en paramètre ou retournée à un autre contexte, ou lorsque le modificateur de la variable p est de type public, l'inférence de la relation de composition devient difficile et nécessite des analyses plus complexes (nous présentons dans les chapitres suivants notre approche pour gérer ces cas).

2.2 Propriétés de la relation tout–partie

Plusieurs travaux de recherche ont été menés pour formaliser les différentes propriétés de la relation tout–partie, notamment dans le domaine de la représentation des connaissances, des ontologies et du génie logiciel. Winston et al. [WCH87] sont les premiers à proposer une classification des sémantiques de la relation entre un tout et ses parties. Ils proposent six types de relation tout–partie et les classent suivant trois propriétés :

- fonctionnalité : propriété qui indique si la relation d'une partie avec un tout est fonctionnelle ou non ;
- homéomérie : propriété qui indique si les parties sont ou non homéomères ;
- séparabilité : propriété qui indique si la partie et le tout sont séparables.

Barbier et al. [BHSPLB03] présentent une extension de UML 1.4 et introduisent un fragment de métamodèle pour la relation tout–partie basé sur trois méta-classes : *Whole-Part*, *Aggregation* et *Composition*. Ils reprennent les propriétés primaires et secondaires

de la relation tout–partie introduites par Kilov [Kil99] et les formalisent en utilisant OCL :

- propriétés primaires : tout–partie (nature binaire), propriété émergeante, propriété résultante, asymétrie au niveau instance, antisymétrie au niveau type ;
- propriétés secondaires : encapsulation, chevauchement des durées de vie (9 cas), transitivité, partageabilité, configurabilité, séparabilité, mutabilité, et dépendance existentielle.

Gensel et al. [GCGZ06] proposent une intégration des diverses sémantiques de la relation tout–partie. Ils proposent une extension du métamodèle AROM et distinguent agrégation et composition ainsi qu'un certain nombre de propriétés primaires et secondaires. Cependant, ils n'offrent pas de formalisation de leurs propriétés. Dans leur article, ils distinguent trois types principaux de relations tout–partie :

- relation de type agrégation : sa principale propriété est la partageabilité, ce qui veut dire qu'une partie peut appartenir à plusieurs tout ;
- relation de type appartenance : un cas particulier de l'agrégation, dont la principale propriété est la séparabilité, ce qui veut dire que la partie est séparable de son tout ;
- relation de type composition : ce type de relation n'admet pas la partageabilité, donc une partie doit nécessairement appartenir à un seul tout.

2.3 Analyse de pointeurs

L'analyse de pointeurs (*pointer analysis*) permet d'estimer l'ensemble des locations en mémoire sur lesquelles les pointeurs peuvent pointer. Elle permet donc de déterminer les locations en mémoire qui peuvent être accessibles avec plus d'un seul moyen [Muc97]. C'est un outil important, pour les compilateurs, pour effectuer correctement la majorité des optimisations [Muc97]. Dans notre mémoire, nous utilisons l'analyse de pointeurs pour déterminer les points du programme où des variables référencent la même location en mémoire ; l'analyse des références de ces variables nous permet, comme nous allons le

détailler dans la suite de ce mémoire, de vérifier si une relation de composition existe ou non.

Lhoták dans son mémoire [Lho02] détaille les différents travaux faits pour améliorer l'efficacité et la précision des analyses de pointeurs avec les langages *C* et *Java*. Il cite en particulier le travail d'Emami et al. [EGH94] qui introduit une analyse "pointe-sur" (*point-to analysis*) pour les structures de données allouées dans la pile. L'analyse est sensible au contexte (*contexte sensitive*) et utilise le concept des locations abstraites de la pile pour saisir toutes les relations possibles entre les locations accessibles dans la pile. Elle calcule les ensembles (*pointe-sur ensemble*) des variables sur lesquelles un pointeur peut pointer. Andersen [And94] propose une analyse pointe-sur pour le langage de programmation *C*. Cette analyse est insensible au contexte et au flot mais présente une bonne modélisation de la mémoire dynamique (*tas*). L'implémentation utilise des contraintes pour représenter les données de l'analyse et vise ainsi à résoudre ces contraintes.

2.4 Inférence de l'unicité des références

Dans les études concernant la gestion de la mémoire et l'inférence d'unicité, la référence est définie comme étant unique s'il n'existe pas d'autres locations en mémoire qui référencent le même objet [CR07].

Cherem et al. [CR07] proposent un algorithme pour inférer l'unicité des références dans un programme Java ; si la référence d'un champ est unique dans un programme, l'algorithme ajoute un destructeur explicite pour ce champ aussitôt qu'il réfère un autre objet. Ils effectuent une analyse de pointeurs de type "must-alias" qui permet de savoir précisément à chaque point du programme quels sont les objets référencés par les variables (et les champs) [Muc97] ; leur algorithme effectue, dans une première étape, une analyse intra-procédurale (*flow sensitive must-alias*) pour chaque méthode et par la suite effectue une analyse inter-procédurale (*flow insensitive*) pour construire des ensembles des références vers un objet. Si l'ensemble contient une seule référence, alors cette référence est unique. Leur algorithme permet le partage temporaire d'une référence avec d'autres champs à l'intérieur d'une méthode s'il est prouvé que ces champs sont vraiment uniques

à l'appel et à la sortie de la méthode. Typiquement, cela est pratique pour les structures de données qui gardent des références sur des objets ; si le partage est unique à l'intérieur des méthodes définies dans ces structures de données alors il est possible d'y ajouter des destructeur pour libérer les objets référencés.

Ma et al. [MF07] présentent aussi un algorithme pour inférer l'unicité des références dans des programmes Java ; leur implémentation utilise une analyse de pointeurs locale combinée avec une analyse interprocédurale basée sur des contraintes. L'analyse intra-procédurale est sensible au flot pour les variables locales mais insensible au flot pour les champs. L'affectation aux champs est limitée aux champs de l'objet courant ; les autres affectations aux champs d'un objet externe ou aux champs statiques sont considérées automatiquement comme une violation de la propriété d'unicité.

2.5 Identification de la relation de composition

La définition et l'implémentation des algorithmes pour l'inférence de la relation de composition sont l'objet de plusieurs travaux de recherche. Notamment, Guéhéneuc et al. [GAA04] présentent une définition formelle de la relation de composition basée principalement sur les propriétés d'exclusivité et de durée de vie :

- l'exclusivité exprime la possibilité pour un objet d'être référencé par un ou plusieurs autres objets à un moment donné. Une instance de B est dite exclusive à une instance de A si l'instance de B n'est pas référencée par d'autres instances au même moment ; cela ne devrait pas donc exclure le transfert d'exclusivité. Dans le cas d'une relation de composition entre A (le tout) et B (la partie), une instance de B doit être exclusive à une instance de A, mais le cas inverse n'est pas nécessaire ;

- la dépendance existentielle exprime une contrainte sur la durée de vie de l'instance d'une classe par rapport à la durée de vie de l'instance de la classe à laquelle elle est liée. Dans le cas d'une relation de composition entre A et B, une instance de B ne peut pas vivre plus longtemps que l'instance correspondante de A. En pratique, cela exclut donc la possibilité de transfert d'exclusivité.

Leur implémentation est basée sur une analyse dynamique pour inférer la relation de composition entre deux classes ; ils utilisent une technique d'analyse de la trace pour saisir les événements de l'exécution du programme (lecture de la valeur d'un champ, destruction d'une instance et fin du programme). L'analyse de ces événements est faite avec des prédicats (Prolog) dont le calcul donne des valeurs pour les propriétés d'exclusivité et de durée de vie et qui permettent ainsi d'inférer des relations de composition.

Kollmann et al. [KG01] présentent également une définition de la relation de composition basée sur les propriétés de *strong ownership* et de coïncidence de durée de vie :

- le *strong ownership* nécessite qu'une partie soit incluse dans *un* seul tout. Cette propriété est une forme forte d'exclusivité qui n'admet pas les références temporaires et le transfert d'unicité ;
- la coïncidence de durée de vie nécessite que l'initialisation du tout soit faite avant l'initialisation des parties et que la destruction des parties soit faite, au plus tard, au moment de la destruction du tout.

Ils proposent des algorithmes, utilisant aussi l'analyse dynamique, pour la détection de ces propriétés. Ils vérifient à chaque création d'un nouvel objet si des références vers des objets existants ont été établies et les excluent du groupe des parties potentielles. Ils vérifient aussi pour tous les nouveaux objets si le nombre de références est inférieur à 2, sinon ils seront exclus. Tous les objets référencés dont la durée de vie dépasse celle des objets qui les référencent sont exclus aussi. À la fin du programme, ils infèrent une relation de composition pour les objets qui n'ont pas été exclus.

La vérification des propriétés d'exclusivité (ou non partageabilité) et de dépendance de durée de vie de la partie avec un tout présentent des limitations ; les implémentations utilisant uniquement la propriété de dépendance de durée vie sont moins performantes [YSC+05]. Aussi, il n'est pas possible avec ces analyses d'identifier les cas où la partie est référencée par des objets qui sont, eux mêmes, parties du tout, ou d'identifier le cas où la partie est utilisée dans le contexte des patrons de conceptions *itérateurs*, *décorateurs* et *fabriques* où les références vers les parties sont souvent temporaires [Mil05]. Une autre limitation est que les résultats obtenus à partir de l'analyse dynamique ne peuvent pas être généralisées

pour toutes les exécutions [Ern03; JR00].

Pour remédier à ces problèmes, Milanova [Mil05] présente une définition de la relation de composition basée sur le modèle du tout dominateur (*owners-as-dominators model*) qui ne nécessite pas de relation exclusive des parties avec le tout ; dans ce modèle, la notion de dominance d'un nœud sur les autres nœuds dans un graphe permet de définir une frontière d'accès à un objet et par conséquent d'ignorer les appels à l'intérieur de cette frontière.

> Un nœud `m` domine un nœud `n` si chaque chemin de la racine du graphe qui atteint le nœud `n` doit passer par le nœud `m`. Le nœud `m` domine immédiatement le nœud `n` si `m` domine `n` et il n'existe pas de nœud `p` tel que `m` domine `p` et `p` domine `n` [Mil05].

L'implémentation de cette approche utilise l'analyse de pointeurs qui opère sur des programmes incomplets (*fragment points-to analysis*) ; l'analyse procède en créant une méthode *main* qui approxime toutes les exécutions possibles du programme. Le résultat de cette analyse est un graphe qui approxime tous les graphes possibles d'un objet à l'exécution ; ce graphe est une abstraction statique de la mémoire dynamique (tas) à l'exécution où chaque nœud correspond à un objet et les arcs entre ces nœuds correspondent aux références entre ces objets. L'analyse ensuite utilise ce graphe pour déterminer pour chaque nœud `m` la frontière du sous graphe dont la racine est ce nœud. Si le nœud `n` est visible uniquement dans le sous-graphe dont la racine est `m` alors une relation de composition est identifiée. L'approche de Milanova se distingue par l'utilisation de l'analyse statique et la possibilité d'analyser des programmes incomplets.

CHAPITRE 3

NOTRE DÉFINITION DE LA RELATION DE COMPOSITION

3.1 Définitions : variable, référence et objet

Un nom de variable dans un programme désigne un emplacement ou une location particulière en mémoire à l'exécution ; plus précisément, la valeur d'une variable, à l'exécution, est une référence sur une location particulière en mémoire (nous excluons de cette définition les variables de type primitives int, char, ...). La lecture de cette valeur correspond à la création d'une nouvelle copie de la référence sur la location en mémoire référencée par cette variable. Par exemple, l'instruction suivante :

```
1 | x = y;
```

permettra aux deux variables y et x de référencer la même location en mémoire, chacune aura une copie de la référence.

Dépendamment du contexte et du flot de contrôle, une variable peut, au cours de l'exécution, référencer différentes locations en mémoire à différents points du programme. De même, une location en mémoire peut être référencée par plusieurs variables. Il est donc important de distinguer les références dans un programme :

```
1 | x = y;
2 | x = z;
```

En effet, la référence contenue dans la variable x à la ligne 1 peut être différente de la référence contenue dans x à la ligne 2.

Nous étudions plus en détails dans les sections suivantes l'importance pour notre approche de distinguer les références dans le programme et nous présentons dans les chapitres suivants les différents cas qui nécessitent une représentation spéciale des références.

Dans le contexte du langage Java, nous pouvons dire indifféremment qu'une variable *référence* ou *pointe* sur une location particulière en mémoire. Dans ce qui suit, nous utiliserons la terminologie suivante :

- variable : pour désigner une variable locale (déclarée dans une méthode) ;

- champ : pour désigner une variable de classe (statique) ou d'instance ;
- objet (ou instance) : pour désigner une location particulière en mémoire ;
- copie de la référence : pour désigner la référence contenue dans une variable ou un champ à un point du programme.

3.2 Notre définition de la relation de composition

Une relation entre deux classes `A` et `B` est dite de composition si on peut prouver que la référence est unique entre les instances de ces deux classes : si `A` déclare un champ b de type `B`, alors la référence contenue dans b doit être unique (si une instance de `A` doit être supprimée de la mémoire alors l'instance de `B` référencée par b doit être aussi supprimée). L'unicité de la référence ne doit pas exclure l'existence d'autres références temporaires, ainsi une référence peut être :

- unique : dans ce cas l'instance référencée a un parent principal qui, si supprimé, implique que cette instance doit être aussi supprimée (ceci est vrai même s'il y a d'autres références sur cette instance, qui doivent être dans ce cas temporaires) ;
- temporaire : une référence est dite temporaire, s'il ne peut exister un point du programme où cette référence est utilisée après la suppression, dans la mémoire, de l'instance référencée ;
- normale : en présence de plus d'une référence non temporaire, aucune référence ne peut être qualifiée d'unique ; dans ce cas ces références sont considérées normales et l'instance référencée n'a pas de parent connu.

Le problème principal de cette définition est d'identifier les cas où une référence peut être considérée temporaire. Milanova [Mil05] identifie deux cas où la référence vers une instance est ignorée parce qu'elle est considérée comme une sorte de relation d'accès temporaire :

- le premier cas d'utilisation temporaire d'une référence est celui où un objet est créé dans un contexte et retourné, sans être utilisé, à un autre contexte ; cela correspond

à dire que la méthode créatrice renvoie une référence unique. Par exemple, si un objet est créé et retourné immédiatement (i.e., `return new Object()`), ou passé à un autre contexte (i.e., `new FileWriter(new File())`), cela implique que dans le code du constructeur, la référence this est aussi temporaire ;

- le deuxième cas d'utilisation temporaire d'une référence concerne l'accès local à un champ ; la référence obtenue n'est pas renvoyée à un autre contexte (affectée à un champ, passée en paramètre d'une méthode ou retournée). Ce cas est très utile pour accéder aux champs ou invoquer des méthodes définies dans la classe de la référence.

Il est important de mentionner que l'aspect temporaire d'une référence concerne sa durée de vie uniquement dans le contexte où elle est utilisée. Autrement dit, il se peut qu'une référence soit considérée comme temporaire mais sa durée de vie est relative au temps d'exécution du programme (par exemple, une variable qui est définie dans la méthode principale du programme *main* va exister le temps d'exécution de tout le programme).

3.3 Exemples de relations de composition

Cette section présente des exemples d'implémentation de la relation de composition et explique pour chaque exemple les conditions pour lesquelles nous pouvons affirmer (ou non) la présence d'une relation de composition.

3.3.1 Exemple 1 : portée d'une référence

```
1  class B {}
2
3  class A {
4    private B b1 = new B();
5  }
6
7  class CompositionExample1 {
8    void main () {
9      A a1 = new A();
10   }
11 }
```

La référence vers l'instance de `A` dans la méthode *main* est sauvegardée dans une variable locale ; cette variable sera supprimée de la pile à la sortie de la méthode *main*. L'instance de `A` pourra alors être libérée de la mémoire ; la machine virtuelle pourrait donc supprimée cette instance lors du "ramassage des miettes" (*garbage collection*) des objets en mémoire puisqu'il n'y a pas d'autres références dessus. L'instance de `B` sera aussi susceptible d'être libérée de la mémoire vu que la référence de `B` dans `A` est sauvegardée dans un champ dont le modificateur est privé (*private*) et elle n'est pas affectée ou retournée à un autre contexte à l'extérieur de `A` (ceci est vrai même si on suppose que cet exemple fait partie d'un programme qui contient d'autres lignes du code en dehors de `A` et `B`. De même si `A` devait avoir des sous-classes, la référence contenue dans *b1* ne peut être modifiée. On peut donc déduire une relation de composition entre la classe `A` et la classe `B`.

3.3.2 Exemple 2 : référence temporaire sur un objet

```
1  class B {}
2
3  class A {
4    private B b1;
5
6    public void setB(B b) {
7      this.b1 = b;
8    }
9  }
10
11 class CompositionExample2 {
12   void main() {
13     B b2 = new B();
14     A a1 = new A();
15     a1.setB(b2);
16   }
17 }
```

De même que dans l'exemple précédent, l'instance de `A` est susceptible d'être libérée de la mémoire à la sortie de la méthode *main* puisqu'il n'y a aucune autre référence dessus. L'instance de `B` créée à la ligne 10 est référencée par le champ *b1* et la variable *b2*, mais la variable *b2* sera supprimée de la pile, par conséquent l'instance de `B` sera référencée uniquement par l'instance de `A`. On peut donc déduire une relation de composition entre la

classe `A` et la classe `B`. Dans cet exemple, l'instance de `B` (composant) est crée à l'extérieur de `A` (objet composite) et avant la création de l'instance de `A`.

3.3.3 Exemple 3 : références non-temporaires sur un objet

```
1  class B {}
2
3  class A {
4    public static B b1;
5  }
6
7  class C {
8    public static B b2;
9  }
10
11 class CompositionExample3 {
12   void main() {
13     B b3 = new B();
14     A.b1 = b3;
15     C.b2 = b3;
16   }
17 }
```

Dans cet exemple, à la sortie de la méthode *main*, la variable *b3* sera supprimée de la pile, donc l'instance de B créée à la ligne 10 sera référencée par les champs *b1* de `A` et *b2* de `C`. Nous pouvons affirmer qu'il n'y a pas de relation de composition entre les classes `A` et `B` d'un côté ni entre les classes `C` et `B` d'un autre coté.

3.3.4 Exemple 4 : références entre composants du même objet composite

```
1  class B {}
2
3  class A {
4    private B b1;
5    private C c1 = new C();
6
7    public void setB(B b) {
8    this.b1 = b;
9      c1.setB(b);
10   }
11 }
12
13 class C {
14   private B b2;
15
```

```
16    public void setB(B b) {
17      this.b2 = b;
18    }
19  }
20
21  class CompositionExample4 {
22    void main() {
23      B b3 = new B();
24      A a1 = new A();
25      a1.setB(b3);
26    }
27  }
```

Dans cet exemple, à la sortie de la méthode *main*, l'instance de `B` sera référencée dans `A` et `C` et l'instance de `C` sera référencée uniquement dans `A`, donc, on peut déduire une relation de composition entre la classe `A` et la classe `C` mais il n'y a pas de relation de composition entre la classe `C` et la classe `B`. Par contre, lorsque la machine virtuelle réclamera l'instance de `A`, elle pourra réclamer l'instance de `C` et celle de `B`. On peut donc déduire aussi une relation de composition entre la classe `A` et la classe `B`. En fait, c'est un cas plus délicat qui n'est pas inclus dans toutes les définitions de "composition", et pas dans la nôtre.

3.4 Modélisation de la relation de composition

Notre approche pour modéliser la relation de composition entre deux classes `A` et `B` consiste à annoter la référence entre ces deux classes avec un descripteur d'unicité. Ce descripteur d'unicité est défini en fonction des deux propriétés suivantes :

- exclusive ;
- transitoire.

L'exclusivité est une propriété qui distingue une référence particulière des autres références au même objet ; au plus une référence sur un objet donné peut être exclusive. Cette propriété est nécessaire mais pas suffisante pour garantir l'unicité ; il faut que toutes les autres références soient transitoires. Si on considère l'ensemble des références possibles sur un objet, une référence peut être définie comme étant :

- exclusive : la référence est exclusive si toutes les autres références sont transitoires ;

- transitoire : la référence est considérée temporaire ; elle ne peut pas empêcher la suppression de l'objet référencé et donc n'altère pas la propriété d'unicité ;
- non-exclusive et non-transitoire : la référence est considérée normale ; elle empêche de conclure une relation de composition ;
- exclusive et transitoire : cas dégénéré et inintéressant où toutes les références sur un objet sont considérées comme transitoires.

CHAPITRE 4

DÉFINITION DES CONTRAINTES SUR L'UTILISATION DES RÉFÉRENCES

Notre approche pour identifier une relation de composition entre deux classes `A` et `B` est basée sur la vérification de l'unicité de la référence entre `A` et `B`. Dans l'exemple suivant, si nous voulons vérifier que le champ b de la classe `A` correspond à une implémentation de la relation de composition entre `A` et `B`, alors nous devons valider que toutes les références que peut contenir le champ b, à l'exécution, sont uniques.

```
1  class B {}
2  class A {
3    B b;
4    void main() {
5      A a = new A();
6      B b1 = new B();
7      a.b = b1;
8      B b2 = a.b;
9    }
10 }
```

Par exemple, à la ligne 6 du programme ci-dessus, on crée une nouvelle instance de type `B` (la référence vers cette instance est sauvegardée dans *b1*). À la ligne 7, on crée une nouvelle copie de la référence sur la même instance (sauvegardée dans b). De même, à la ligne 8, on crée une nouvelle copie de la référence sur la même instance (sauvegardée dans *b2*).

L'idée de base de notre approche est de gérer ces copies de références qui référencent la même instance et d'imposer des contraintes sur leurs propriétés (exclusive et transitoire) ; ces contraintes sont typiquement définies à tout point du programme où on crée une nouvelle copie d'une référence.

Dans ce chapitre nous étudions en détails les propriétés exclusive et transitoire des références et nous présentons les contraintes que nous avons définies sur ces propriétés et leur analyse. Nous présentons aussi les différentes instructions du langage Java qui permettent de créer une nouvelle copie d'une référence.

4.1 Analyse de la relation de composition

Cette section présente notre analyse et explique les différentes transformations que nous appliquons sur les instructions du code Java. Nous exprimons une partie simplifiée du langage Java comme suit : $var = exp$;
avec :

$$\begin{aligned} var &::= r \mid r.f \\ exp &::= var \mid \mathsf{new}\ C \mid \mathsf{invoke}\ r.m(p_0, p_1, ...) \mid \mathsf{null} \end{aligned}$$

où :

- r ($\in$ à l'ensemble des noms de variables V) : représente une variable (inclut les paramètres de méthodes p0, p1, ...) ;
- f ($\in$ à l'ensemble des noms de champs F) : représente un champ ;
- C : représente une classe ;
- m ($\in$ à l'ensemble des noms de méthodes M) : représente une méthode ;
- invoke : la spécification de la machine virtuelle Java définit quatres primitives pour les appels de méthodes invokevirtual, invokestatic, invokespecial et invokeinterface. Nous allons présenter dans le chapitre suivant les détails que définit chaque primitive (nous signalons que ces primitives sont visibles au niveau du bytecode Java).

La syntaxe du langage présentée ci-dessus exprime une vue globale et simplifiée du langage Java que nous traitons dans notre analyse (nous nous intéressons particulièrement aux différentes instructions du langage Java qui permettent de créer une nouvelle copie d'une référence) ; nous ajouterons des détails dans les sections suivantes pour expliquer comment nous gérons les autres éléments du langage Java comme les tableaux, les structures de contrôle (if-else), les types et les exceptions.

4.1.1 Représentation du code source Java

Notre analyse se base sur la représentation SSA du code source Java (*static single assignment*) ; SSA est une représentation intermédiaire qui garantit une définition sta-

tique unique pour chaque variable [CFR+91]. Ainsi, une variable ne sera modifiée qu'une seule fois : si, dans le programme, on affecte plusieurs valeurs à la même variable alors la représentation SSA va créer de nouvelles variables pour sauvegarder les différentes valeurs. SSA permet d'avoir une variable distincte pour chaque nouvelle référence vers un objet. Le programme suivant :

```
1  x = y;
2  x = z;
```

sera remplacé par :

```
1  x1 = y;
2  x2 = z;
```

Dans le reste de ce chapitre, toutes les transformations du code Java correspondent à la représentation intermédiaire telle que définie par Shimple (chapitre 5) ou à l'extension, nous le mentionnerons explicitement, que nous avons ajoutée à cette représentation.

Dans le cas où une variable pourrait provenir de plusieurs points du programme :

```
1  if (y ≠ null) {
2    x = y;
3  } else {
4    x = z;
5  }
6  w = x;
```

SSA nous permet d'identifier les différents points du programme où la variable a été définie ; elle crée des nœuds Φ qui listent tous les points du programme où la valeur d'une variable est susceptible d'être modifiée à l'exécution du programme :

```
1  if (y ≠ null) {
2    x1 = y;
3  } else {
4    x2 = z;
5  }
6  x3 = φ(x1, x2);
7  w = x3;
```

ainsi, la variable *x3* peut, dépendamment du flot du contrôle, avoir la valeur contenue dans la variable *x1* ou avoir la valeur contenue dans *x2*.

Le fait d'avoir un nouveau nom de variable pour chaque affectation est très important. Cela nous permet de considérer chaque variable comme une référence distincte sur un

objet.

Nous avons besoin aussi de distinguer les différentes lectures d'une variable. Chaque lecture d'une variable implique un partage de la référence avec une autre variable, donc les propriétés d'unicité peuvent être différentes à la prochaine lecture. Nous avons étendu la représentation SSA pour supporter ce cas particulier. Par exemple, le code suivant ne sera pas modifié par SSA :

```
1  x = y;
2  z = y;
3  w = y;
```

Notre analyse transformera ce code en :

```
1  x = y#1;
2  z = y#2;
3  w = y#3;
```

pour nous permettre de distinguer les propriétés de la référence contenue dans la variable y aux différents points du programme. Par exemple, les propriétés de la référence contenue dans la variable y à la ligne 2 dépendent de l'instruction de la lecture de la valeur de y à la ligne 1. De même les propriétés de la référence de y à la ligne 3 dépendent de l'instruction de la lecture de la valeur de y à la ligne 1. Il est important de mentionner que les numéros associés à la variable y n'affectent pas la référence contenue dans cette variable, plutôt, ces numéros permettent de distinguer les différents points du programme où on lit la valeur de cette variable et donc pouvoir exprimer des propriétés différentes de la référence aux différents points du programme.

Un cas particulier est celui où on lit la valeur de la variable dans une structure de contrôle :

```
1  if (v ≠ null) {
2    x = y;
3  } else {
4    z = y;
5  }
6  w = y;
```

ce code sera transformé en :

```
1  if (v ≠ null) {
2    x = y#1;
```

```
3 | } else {
4 |   z = y#2;
5 | }
6 | y#3 = φ(y#1, y#2);
7 | w = y#3;
```

Notre analyse nous permet de représenter le code en fonction des références contenues dans les variables et ainsi nous donne une approximation de l'état de la mémoire à l'exécution. À partir de cette représentation, nous construisons des liens qui indiquent des dépendences et induisent des contraintes sur les copies d'une référence, par exemple, si on considère le programme suivant :

```
1 | x = y#1;
2 | z = y#2;
```

nous allons créer alors les liens de contrainte suivants :

- ligne 1 : un lien entre les références des variables y ($r_{y_{\#1}}$), x (r_x) et y ($r_{y_{\#2}}$) (pour des raisons de simplicité nous allons écrire, dans ce qui suit, r_x pour désigner la référence de la variable x, ou $r_{y_{\#1}}$ pour désigner la référence de la variable y à un point du programme, dans ce cas à la ligne 1) ;
- ligne 2 : un lien entre $r_{y_{\#2}}$ et r_z.

Dans notre implémentation, les liens correspondent à une structure de données qui regroupe les références associées par les instructions que nous allons détailler ci-dessous (seules ces instructions sont considérées lors de la création des liens entre les références).

Les références associées par un lien référencent le même objet, ainsi, comme nous allons l'expliquer dans le chapitre suivant, nous pouvons exprimer des contraintes sur les propriétés de ces références. Nous expliquerons aussi dans ce chapitre pourquoi nous avons associée les références de la variable y aux points du programme 1 et 2 avec la référence de la variable x.

4.1.2 Instructions d'affectation

- Instruction de copie `x = y` : nous créons un lien entre r_y et r_x ;
- Lecture d'un champ `x = y.f` : nous créons un lien entre r_f et r_x ;

- Écriture dans un champ `x.f = y` : nous créons un lien entre r_y et r_f.

4.1.3 Création d'objet : `T` x = **new** `T()`

Nous créons une variable spéciale pour représenter l'opération d'instanciation.
Le programme suivant :

```
1 | a = new T();
```

sera remplacé par celui ci :

```
1 | x = new T;
2 | invoke x.T.init ();
3 | a = x;
```

Donc, notre analyse va créer les liens suivants :

- ligne 1 : un lien entre r_x et `new T` ;
- ligne 3 : un lien entre r_a et r_x.

Dans notre implémentation, nous remplaçons le `new T` par un nom de varaible unique pour tout le programme. Le `new` nous permet de propager les propriétés de la référence de façon cohérente avec notre approche de résolution de contraintes. Le `new` est considéré comme une référence spéciale qui est unique au point du programme où elle est utilisée mais temporaire après ce point du programme.
À la ligne 2, l'invocation du constructeur de la classe `T` est gérée de la même manière qu'un appel d'une méthode standard.

4.1.4 Nullification : `T` x = **NULL**

Nous créons une variable spéciale pour représenter l'opération de nullification associée à une variable x. Le `NULL` nous permet de propager les propriétés de la référence de façon cohérente avec notre approche de résolution de contraintes. Le `NULL` est considéré comme une référence spéciale qui est unique au point du programme où elle est utilisée mais temporaire après ce point du programme.

4.1.5 Appel de méthode : invoke r.m(p0, p1, ...)

Dans le cas d'un appel de méthode, on lie toutes les variables passées en paramètres avec les paramètres formels de la méthode. Dépendamment du flot de contrôle, une méthode peut retourner plusieurs valeurs à différents points du programme, on lie donc toutes les variables retournées par la méthode avec la variable qu'on lui affecte la valeur. Prenons l'exemple suivant :

```
1  T bar(T obj) {
2    return obj;
3  }
4  void main() {
5    T y = new T();
6    T x = bar(y);
7  }
```

Le programme précédent sera remplacé par (simplifié pour des raisons de lisibilité) :

```
1  void main() {
2    w1 := @this;
3    y = new T;
4    x = invoke w1.bar(y);
5  }
6  java.lang.Object bar(T) {
7    w2 := @this;
8    w3 := @parameter0: T;
9    return w3;
10 }
```

et notre analyse va créer les liens suivants :

- ligne 2 : un lien entre $r_{@this}$ et r_{w1} ;
- ligne 3 : un lien entre $r_{new\ java.lang.Object}$ et r_y ;
- ligne 4 : un lien entre r_{w1} et $r_{@this}$;
- ligne 4 : un lien entre r_y et $r_{@parameter0}$;
- ligne 7 : un lien entre $r_{@this}$ et r_{w2} ;
- ligne 8 : un lien entre $r_{@parameter0}$ et r_{w3} ;
- ligne 9 : un lien entre r_{w3} et r_x.

Dans notre implémentation, nous remplaçons le @this par un nom de variable unique pour tout le programme ; nous identifions le @this comme une variable locale déclarer dans la méthode. Ce changement de nom de variable est nécessaire, sinon on aura le même nom @this pour toutes les méthodes.

Si on suppose que la variable *w1* est de type `T`, alors nous allons créer aussi ces liens pour toutes les classes qui héritent de `T` (ou implémente `T` dans le cas d'une interface) et qui redéfinissent la méthode bar. Ces cas concernent le *virtual invoke* et le *interface invoke*.
Les variables *@this*, *@parameter0*, et *w3* sont uniques pour tous les appels de la méthode *bar*.

4.2 État de la copie de la référence à un point de programme

Nous exprimons les propriétés d'une référence comme suit :

$$\begin{aligned} t &::= 0 \mid 1 \\ e &::= 0 \mid 1 \\ u &::= (e, t) \end{aligned}$$

où :

- t : définit la propriété transitoire qui peut prendre la valeur 1 si la référence est transitoire, sinon 0 ;
- e : définit la propriété exclusive qui peut prendre la valeur 1 si la référence est exclusive, sinon 0 ;
- u : définit l'unicité de la référence exprimée avec le couple (e, t).

Si on considère l'affectation `x = y`, alors les valeurs admissibles des propriétés exclusive et transitoire (e, t) de la référence contenue dans la variable y dépendent de ses propriétés avant (e_b, t_b) et après (e_a, t_a) ce point de programme.

- si (e=0, t=0) alors (e_b= ?, t_b=0) et (e_a=0, t_a= ?) : si la référence est non-exclusive et non-transitoire à un point du programme, alors la référence peut être exclusive ou

non avant ce point du programme (le point d'interrogation est utilisé pour exprimer cette ambigüité), mais nécessairement non-exclusive après ce point du programme. Aussi elle est non-transitoire avant mais peut être transitoire ou non après ce point du programme parce que ce n'est pas grave si elle devient transitoire. Toutefois, cela ne peut servir à rien de la rendre transitoire (elle était (0,0) dans le passé donc elle ne peut pas participer à une relation de composition de toute manière) ;

- si (e=0, t=1) alors (e_b= ?, t_b= ?) et (e_a=0, t_a=1) : si la référence est non-exclusive et transitoire à un point de programme, alors elle peut prendre tous les cas avant ce point de programme mais elle est non-exclusive et transitoire après ce point de programme ;

- si (e=1, t=0) alors (e_b=1, t_b=0) et (e_a= ?, t_a= ?) : si la référence est exclusive et non-transitoire à un point de programme, alors la référence est exclusive et non-transitoire avant ce point de programme, mais on ne peut rien déduire sur les valeurs de ses propriétés après ce point de programme. Seule la propagation des contraintes, que nous allons définir ci-dessous, peut nous confirmer si une référence conserve son unicité ou pas.

Ces différents cas peuvent être décomposer comme suit :

- e=1 est un "sous-type" de e=0, donc on peut toujours remplacer un e=1 par un e=0 : on y perd de l'information mais le résultat est valide ;

- t=0 est un sous-type de t=1, donc on peut toujours remplacer un t=0 par un t=1 : on y perd de l'information mais le résultat est valide ;

- (1,t) se dédouble en (1,t) et (0,1) ;

- (0,t) se dédouble en (0,t) et (0,t).

Le système de contraintes a toujours au moins une solution, où tous les e=0 : c'est une solution triviale qui ne nous intéresse pas vu qu'elle correspond à un échec de notre recherche de composition.

Pour l'exemple :

```
1 | x = y_#1;
2 | z = y_#2;
3 | w = x;
```

si y est unique à la ligne 1, alors à la ligne 2 nous avons deux cas possibles :

- si la référence de y (après la ligne 1) doit être unique, alors celle de x (ligne 1) doit être temporaire, ce qui veut dire que nous allons imposer la même contrainte sur la variable w (ligne 3), c.à.d. la référence de w doit être aussi temporaire ;
- si la référence de x (ligne 1) doit être unique, alors nécessairement la référence de y (ligne 1) est unique, mais y sera désormais considérée temporaire (après la ligne 1), et donc nous allons imposer la même contrainte sur la variable z (lignes 2), c.à.d. la référence de z doit être aussi temporaire.

Pour conserver un historique du changement de l'unicité de la référence, nous définissons les variables logiques suivantes :

1. Nous définissons les deux variables logiques ($e_{r_{y_{\#1}}}$, $t_{r_{y_{\#1}}}$) :
   - $e_{r_{y_{\#1}}}$: définit la valeur de la propriété exclusive de la référence de la variable y à la ligne 1 ;
   - $t_{r_{y_{\#1}}}$: définit la valeur de la propriété transitoire de la référence d'origine de la variable y à la ligne 1.
2. Nous définissons les deux variables logiques ($e_{r_{y_{\#2}}}$, $t_{r_{y_{\#2}}}$) :
   - $e_{r_{y_{\#2}}}$: définit la valeur de la propriété exclusive de la référence de la variable y à la ligne 2 ;
   - $t_{r_{y_{\#2}}}$: définit la valeur de la propriété transitoire de la référence de la variable y à la ligne 2.
3. Nous définissons les deux variables logiques (e_{r_x}, t_{r_x}) :
   - e_{r_x} : définit la valeur de la propriété exclusive de la référence de la variable x à la ligne 1 ;

- t_{r_x} : définit la valeur de la propriété transitoire de la référence de la variable x à la ligne 1.

Nous pouvons donc distinguer les références des différentes variables, mais aussi distinguer la référence d'une même variable aux différents points du programme. Cela nous permet de gérer facilement le changement de l'unicité des références ; par exemple si la référence de y est unique à la ligne 1 alors il est possible de la considérer autrement à la ligne 2.

4.3 Définition des contraintes sur les variables logiques

Nous avons présenté dans la section précédente les règles qui déterminent les valeurs des propriétés de la référence contenue dans une variable aux différents points du programme où on lit la valeur de cette variable. Nous avons aussi défini des variables logiques qui caractérisent la référence en entrée et les deux références en sortie d'une affectation. Nous définissons dans cette section les contraintes sur les valeurs de ces variables logiques :

1. Si $(t_{r_{y_{\#1}}} = 1)$ alors $(t_{r_x} = 1)$ et $(t_{r_{y_{\#2}}} = 1)$: si une référence est transitoire, alors toutes les nouvelles références créées à partir de cette référence devront être aussi transitoires. Cette contrainte est très importante pour conserver l'unicité de la référence en cas de transfert d'unicité ; il devrait y avoir une seule référence non-transitoire vers un objet pour pouvoir identifier une relation de composition.

2. Si $(t_{r_{y_{\#2}}} = 0)$ alors $(t_{r_{y_{\#1}}} = 0)$: si, à un point de programme, une référence est non-transitoire, alors, on peut conclure qu'elle est nécessairement non-transitoire avant ce point de programme.

3. Si $(e_{r_{y_{\#1}}} = 0)$ alors $(e_{r_x} = 0)$ et $(e_{r_{y_{\#2}}} = 0)$: si une référence est non-exclusive, alors toutes les nouvelles références créées à partir de cette référence devront être non-exclusives.

4. Si $(e_{r_{y_{\#1}}} = 1)$ alors $(e_{r_x} = 0)$ ou/et $(e_{r_{y_{\#2}}} = 0)$: si la référence détenue par la variable y à la ligne 1 est exclusive alors soit que la référence de la variable y à la ligne 2 devienne non-exclusive, soit que celle obtenue par la variable x, soit non-exclusive. Autrement dit, il est normal que l'une des deux références soit exclusive

ou que les deux soient non-exclusives, mais elles ne peuvent pas être exclusives toutes les deux.

5. Si $(e_{r_x} = 1)$ alors $(e_{r_{y_{\#1}}} = 1)$ et $(e_{r_{y_{\#2}}} = 0)$ et $(t_{r_{y_{\#2}}} = 1)$: si la référence de la variable y doit êtr exclusive, alors il est naturel que la référence de la variable y (ligne 1) soit exclusive et que sa référence deviennent non-exclusive à la ligne 2. Cela permet le transfert d'unicté à une autre référence, mais il faut que la référence de la variable y (ligne 2) soit transitoire.

6. Si $(e_{r_{y_{\#2}}} = 1)$ alors $(e_{r_{y_{\#1}}} = 1)$ et $(e_{r_x} = 0)$ et $(t_{r_x} = 1)$: si la référence de la variable y (ligne 2) est exclusive, alors on peut conclure que celle de la variable y (ligne 1) est exclusive et celle de la variable x est non-exclusive ; naturellement, la référence de la variable x doit être transitoire pour respecter la propriété d'unicité.

4.4 Formalisation

Notre algorithme de calcul de contraintes est en fait un algorithme qui essaie de trouver une dérivation valide pour le programme dans le système de règles suivant.

D'abord les règles de "sous-typage" qui permettent de perdre de l'information lorsque nécessaire. Le jugement a la forme $u \Rightarrow u'$ ce qui signifie que si une variable a l'unicité u, alors il est valide de la traiter comme ayant l'unicité u' :

$$\frac{}{(1,t) \Rightarrow (0,t)}(\text{weak-e}) \qquad \frac{}{(e,0) \Rightarrow (e,1)}(\text{weak-t})$$

$$\frac{}{u \Rightarrow u}(\text{refl}) \qquad \frac{(e_1,t_1) \Rightarrow (e_2,t_2) \qquad (e_2,t_2) \Rightarrow (e_3,t_3)}{(e_1,t_1) \Rightarrow (e_3,t_3)}(\text{trans})$$

Les deux dernières règles représentent la réflexivité et la transitivité.

Ensuite, les règles de dédoublement. Le jugement a la forme $u_1 \Leftarrow u \Rightarrow u_2$ ce qui signifie que u peut se dédoubler en u_1 et u_2 :

$$\frac{}{(1,t) \Leftarrow (1,t) \Rightarrow (0,1)}(\text{split-e}) \qquad \frac{}{(0,t) \Leftarrow (0,t) \Rightarrow (0,t)}(\text{split-s})$$

$$\frac{u_1 \Leftarrow u \Rightarrow u_2}{u_2 \Leftarrow u \Rightarrow u_1}(\text{sym})$$

$$\frac{u_1 \Leftarrow u \Rightarrow u_2 \qquad u' \Rightarrow u \qquad u_1 \Rightarrow u'_1 \qquad u_2 \Rightarrow u'_2}{u'_1 \Leftarrow u' \Rightarrow u'_2}(\text{weaken})$$

Ces règles de sous–typage et de dédoublement garantissent que la propriété d'exclusivité ne peut pas être copiée (elle ne peut au mieux qu'être perdue), et que la propriété d'être transitoire à l'inverse ne peut pas être perdue.

Finalement, les règles pour chaque opération. Elles utilisent un environnement Γ qui associe à chaque variable son unicité, un autre environnement Δ qui associe à chaque méthode et chaque champ son unicité, et elles ont la forme $\Delta; \Gamma \ \vdash \ i \ \Rightarrow \ \Gamma'$ ce qui signifie que l'instruction i lorsqu'exécutée dans un contexte Δ et Γ renvoie un nouveau contexte Γ'.

$$\frac{u_y \Leftarrow u \Rightarrow u_x}{\Delta; \Gamma, y : u \ \vdash \ x = y \ \Rightarrow \ \Gamma, y : u_y, x : u_x}$$

$$\frac{u_y \Leftarrow u \Rightarrow (e_f, 0)}{\Delta, f : (e_f, 0); \Gamma, y : u \ \vdash \ x.f = y \ \Rightarrow \ \Gamma, y : u_y}$$

$$\frac{u_x \Leftarrow (e_f, 0) \Rightarrow (e_f, 0)}{\Delta, f : (e_f, 0); \Gamma \ \vdash \ x = y.f \ \Rightarrow \ \Gamma, x : u_x}$$

$$\frac{(1,0) \Rightarrow u}{\Delta; \Gamma \ \vdash \ x = \mathsf{new}\ C \ \Rightarrow \ \Gamma, x : u}$$

$$\frac{u'_z \Leftarrow u_z \Rightarrow u_a \qquad u'_y \Leftarrow u_y \Rightarrow u_s \qquad u_r \Rightarrow u_x}{\Delta, m : (u_s, u_a) \to u_r; \Gamma, y : u_y, z : u_z \ \vdash \ x = \mathsf{invoke}\ y.m(z) \ \Rightarrow \ \Gamma, y : u'_y, z : u'_z, x : u_x}$$

Lors des jonctions de flot de contrôle, il faut bien sûr unifier les environnements de chaque branche entrante.

CHAPITRE 5

IMPLÉMENTATION

5.1 Présentation de notre implémentation

La première partie de l'implémentation de notre approche concerne l'analyse du programme et la construction d'un graphe de dépendance entre les variables et les champs. L'API Shimple de Soot nous permet de traverser un programme et d'identifier les points du programme où de nouvelles copies d'une référence sont créées. La représentation Shimple nous garantit des points de définition uniques pour chaque variable ; si une variable est définie deux fois dans le corps d'une méthode, alors Shimple crée une nouvelle variable pour sauvegarder la nouvelle valeur. S'il peut y avoir une ambiguïté pour distinguer les deux variables, alors des nœuds Phi sont ajoutés pour spécifier quels sont les différents points du programme qui ont redéfini la variable.

Une particularité de notre analyse est de considérer ce même comportement aussi en cas de lecture de la valeur d'une variable ; si on assigne la valeur d'une variable en deux points de programmes différents, alors notre système va donner un nom unique à chaque variable en chaque point de programmes mais en assurant que les deux variables pointent sur la même location en mémoire. Cela nous permet de mieux contrôler l'évolution de l'état de la référence pour chaque point de programme. Le résultat de la traversée du programme est une structure de données qui représente le graphe de dépendance entre les variables et les champs du programme.

La deuxième partie de notre analyse concerne la construction des contraintes sur les nouvelles copies des références. Chaque instruction de création d'une nouvelle copie d'un référence implique trois variables et champs ; pour chaque variable et champ, nous associons des variables logiques qui représentent l'état de la copie de la référence à un point de programme précis. Nous utilisons un système de résolution de contraintes pour vérifier la persistance des contraintes établies dans le programme.

5.2 Cadriciel Soot

Nous nous basons dans notre implémentation sur le cadriciel Soot. Soot est un cadriciel d'optimisation Java [VRCG+99]. Il permet de convertir le bytecode Java en quatre représentations intermédiaires possibles dont Shimple (une représentation intermédiaire typée du programme à 3-adresses). Nous considérons uniquement la représentation Shimple car elle offre une forme idéale pour effectuer l'analyse du programme, en particulier, nous signalons que :

- Shimple représente le programme en séquence d'instructions à trois adresses ; l'analyse du programme devient plus facile vu qu'une instruction à trois adresses véhicule, au plus, une opération [ALSU07] ;
- Shimple offre une implémentation de SSA, cela nous garantit des points de définitions distincts pour toutes les variables du programme et donc nous pouvons estimer que chaque variable détient une copie de la référence unique et distincte dans tout le programme.

Il est important de signaler que Soot fournit plusieurs implémentations d'analyse et d'optimisations du code. Nous n'utilisons presque aucune de ces analyses et optimisations dans notre implémentation. Seul la représentation Shimple du code est utilisée ; elle offre une implémentation de SSA qui nous permet de construire notre analyse que nous avons détaillée dans le chapitre précédent.

Nous présentons, ci-dessous, les instructions importantes de la représentation Shimple [Lho02].

5.2.1 Instruction de copie

Une instruction de copie (*assignment statement*) a la forme `p = q` ; nous considérons dans notre analyse uniquement les types non-primitives. Après l'exécution de cette instruction, les variables p et q référencent le même objet.

5.2.2 Instruction d'identité

L'instruction d'identité (*identity statement*) est une instruction de copie particulière. La source de la copie est une variable virtuelle (définie par Shimple) qui représente un paramètre de méthode ou un paramètre d'une exception. Par exemple dans notre analyse, nous considérons les instructions `p := @this : T`; (où this est le paramètre implicite qui pointe sur une instance de la classe `T`), et `p := @parameter0 : java.lang.Object`; (où *parameter0* est le premier paramètre formel de la méthode de type java.lang.Object).

5.2.3 Création d'objet

La création d'objet (*allocation statement*) a la forme `p = new T()`; elle permet à la variable p de référencer un nouveau objet alloué par cette instruction. Nous considérons dans notre analyse les instructions de création d'objets simples et de tableaux. L'appel du constructeur associé à cette instruction d'allocation est gérée dans une instruction d'invocation séparée.

5.2.4 Écriture dans un champ

L'écriture dans un champ (*field store*) a la forme `p.f = q`; où f représente un champ. Cette instruction sauvegarde la valeur de la variable q dans le champ f de l'objet pointé par la variable p.

5.2.5 Lecture d'un champ

La lecture d'un champ (*field load*) a la forme `p = q.f`; elle permet de charger la valeur du champ f de l'objet pointé par la variable q et de la sauvegarder dans la variable p.

5.2.6 Écriture dans un champ statique

L'écriture dans un champ statique (*static field store*) a la forme `T.f = p`; elle sauvegarde la valeur de la variable p dans le champ statique f défini dans la classe dont le

type est `T`. Un champ statique dans une classe est instancié une seule fois pour toutes les instances de cette classe.

5.2.7 Lecture d'un champ statique

La lecture d'un champ statique (*static field load*) a la forme `p = T.f` ; elle charge la valeur du champ statique f et la sauvegarde dans la variable p.

5.2.8 Écriture dans le tableau

L'écriture dans un tableau (*array store*) a la forme `p[i] = q`; elle sauvegarde la valeur de la variable q dans l'ième élément du tableau pointé par la variable p.

5.2.9 Lecture d'un élément du tableau

La lecture d'un élément du tableau (*array load*) a la forme `p = q[i]` ; elle charge la valeur du ième élément du tableau pointé par la variable q et la sauvegarde dans la variable p.

5.2.10 Instruction de coercition

L'instruction de coercition de type (*typecast statement*) a la forme `p = (T) q`; elle permet d'affecter la valeur de la variable q à la variable p. La machine virtuelle Java vérifie que le type de la variable q est un sous type de `T`; si ce n'est pas le cas, une exception est levée.

5.2.11 Appel de méthode

L'appel de méthode (*invocation statement*) permet d'invoquer une méthode. Si la méthode définit des paramètres, cette instruction contiendra les variables qui seront passées en paramètres. Si la méthode retourne une valeur, cette instruction peut contenir une variable qui recevra la valeur retournée. Soot supporte quatre types d'invocation specialinvoke, staticinvoke, virtualinvoke et interfceinvoke.

5.2.12 Retour de méthode

Le retour de méthode (*return statement*) a la forme de `return` ou `return p` ; elle permet de retourner du contexte de la méthode courante au contexte de la méthode qui exécute l'instruction de l'invocation de la méthode courante. En cas de présence d'une valeur de retour, elle peut affecter cette valeur à une variable associée à l'instruction de l'invocation de la méthode courante.

5.2.13 Instruction Phi

Cette instruction est une particularité de la représentation Shimple ; elle a la forme `p = Phi(q0 (0), q1 (1))`. Ce qui veut dire que la variable p peut, en fonction du flot de contrôle, avoir la valeur de la variable *q0* définit au point de programme 0 ou la valeur de la variable *q1* définie au point de programme 1.

5.3 Cadriciel Choco

Nous avons choisi d'utiliser Choco pour la résolution des contraintes que nous avons définies sur les propriétés des références. Choco est un cadriciel à code source ouvert en Java (accessible à partir du site Web sourceforge.net) qui définit une couche abstraite pour construire et résoudre des problèmes de contraintes. L'élément central d'un programme Choco est l'objet `Problem` qui permet de définir des variables, leurs domaines et des contraintes sur ces variables. Choco fournit aussi diverses interfaces pour ajouter de nouvelles contraintes avec des algorithmes de propagation personnalisés.

5.3.1 Problèmes

Typiquement la création d'un problème est faite en créant une instance de la classe `Problem` :

```
1 | AbstractProblem pb = new Problem();
```

5.3.2 Variables et domaines

La création des variables est faite par des méthodes accessibles à partir de l'instance de la classe `Problem`. L'API Choco supporte plusieurs types de variables ; chaque type (par exemple IntDomainVar) fournit des méthodes spécifiques pour gérer le domaine associé à une variable.

```
1 | IntDomainVar v1 = pb.makeEnumIntVar(“var1”, 1, 3);
```

où *v1* est une variable énuméré dont le nom est `var1` et le domaine est l'ensemble [1, 3].

Dans notre implémentation les variables correspondent aux variables logiques (exclusive et transient) que nous associons aux références. Par exemple, pour le code suivant, nous créons les variables logiques $e_{r_{y_{\#1}}}$, $t_{r_{y_{\#1}}}$, $e_{r_{y_{\#2}}}$, $t_{r_{y_{\#2}}}$, e_{r_x}, et t_{r_x}.

```
1 | x = y;
2 | z = y;
```

Le domaine que nous associons à chaque variable logique est [0, 1]. Malheureusement avec 6 variables logiques, la résolution des contraintes devient extrêmement longue. C'est pourquoi nous avons choisit de réduire ce nombre de variables à 3 ; par exemple, les deux varaibles logiques e_{r_x}, et t_{r_x} associées à la référence de la variable x à la ligne 1 seront remplacées par la variable logique u_{r_x} dont le domaine est initialisé comme suit :

- si $e_{r_x} = 0$ et $t_{r_x} = 0$ alors $u_{r_x} = 0$;
- si $e_{r_x} = 0$ et $t_{r_x} = 1$ alors $u_{r_x} = 1$;
- si $e_{r_x} = 1$ et $t_{r_x} = 0$ alors $u_{r_x} = 2$.

Le code pour créer de nouvelles variables en utilisant l'API Choco est comme suit :

```
1 | IntDomainVar ux = pb.makeEnumIntVar(“ux”, 0, 2);
```

Le domaine de la variable présente ci-dessous correspond au cas générique où nous n'avons aucune information sur les valeurs que peut prendre les propriétés de la référence. Cependant il est possible d'optimiser encore le domaine de la variable si nous connaissons une partie ou toute l'information sur ces valeurs. Le code suivant montre les différents cas où nous créons des domaines différents en fonction des valeurs des propriétés de la référence :

```
1  //the parameter e is true if exclusive, false if not exclusive, otherwise null
2  //the parameter t is true if transient, false if not transient, otherwise null
3  public IntDomainVar getIntDomainVar(Problem pb,
4      String variableName,
5      Boolean e, //
6      Boolean t) {
7    if (e == null) {
8      if (t == null) {
9        // can be normal, transient or unique
10       return pb.makeEnumIntVar((variableName + "(0,1,2)"), 0, 2);
11     }
12     else if (t == false) {
13       // can be normal or unique
14       return pb.makeEnumIntVar((variableName + "(0,2)"), new int[]{0, 2});
15     }
16   }
17   else if (e == true) {
18     if (t == false) {
19       // can be just unique
20       return pb.makeEnumIntVar((variableName + "(2)"), 2, 2);
21     }
22   }
23   else if (e == false) {
24     if (t == true) {
25       // can be just transient
26       return pb.makeEnumIntVar((variableName + "(1)"), 1, 1);
27     }
28   }
29   return null;
30 }
```

5.3.3 Contraintes

À partir de l'instance de `Problem`, il est possible de créer des contraintes (arithmétique, logiques ou préfinies) sur les domaines des variables.

```
1  IntDomainVar v1 = pb.makeEnumIntVar("var1", 1, 10);
2  IntDomainVar v2 = pb.makeEnumIntVar("var2", 5, 20);
3  Constraint c1 = pb.neq(v1, v2);
4  pb.post(c1);
```

où *c1* correspond à la contrainte de différence entre les variables *v1* et *v2*. Cette contrainte sera accessible au problème en utilisant la méthode *post*. Le code suivant présente une vue, simplifiée, des contraintes créées dans notre implémentation :

```
1  //if we have statement : x = y;
2  //the parameter yiu specify the uniqueness property value of y at this statement
3  //the parameter xu specify the uniqueness property value of x at this statement
4  //the parameter you specify the uniqueness property value of y after this statement
5  public void getConstraintsB(Problem pb,
6      IntDomainVar yiu,
7      IntDomainVar xu,
8      IntDomainVar you) {
9    //when yiu is normal then xu and you can't be unique
10   Constraint constraintA = pb.and(
11     pb.eq(yiu, 0),
12     pb.neq(xu, 2),
13     pb.neq(you, 2));
14
15   //when yiu is transient then xu and you are also transient
16   Constraint constraintB = pb.and(
17     pb.eq(yiu, 1),
18     pb.eq(xu, 1),
19     pb.eq(you, 1));
20
21   //when yiu is unique and xu is normal then you can't be unique
22   Constraint constraintC = pb.and(
23     pb.eq(yiu, 2),
24     pb.eq(xu, 0),
25     pb.neq(you, 2));
26
27   //when yiu is unique and xu is transient then you is free to take all the values
28   Constraint constraintD = pb.and(
29     pb.eq(yiu, 2),
30     pb.eq(xu, 1),
31     pb.or(pb.eq(you, 0), pb.eq(you, 1), pb.eq(you, 2)));
32
33   //when yiu is unique and xu is uniqe then you must be transient
34   Constraint constraintE = pb.and(
35     pb.eq(yiu, 2),
36     pb.eq(xu, 2),
37     pb.eq(you, 1));
38
39   Constraint[] allConstraints =
40     new Constraint[]{constraintA, constraintB, constraintC, constraintD, constraintE};
41   pb.post(pb.or( allConstraints ));
42 }
```

5.3.4 Résolution

Résoudre un problème consiste à trouver la ou les solutions qui correspondent aux contraintes définies sur les domaines des variables du problème en question. Typiquement, cela est fait en utilisant la méthode *solve* de la classe `Problem` :

```
1  if (pb.solve()) {
2    do {
3      for(int i = 0; i < pb.getNbIntVars(); i++) {
4        print(pb.getIntVar(i) + ":" + ((IntDomainVar) pb.getIntVar(i)).getVal());
5      }
6    } while(pb.nextSolution());
7  }
```

L'API Choco fournit des outils pour optimiser la recherche en utilisant les méthodes *minimize* et *maximize*, aussi limiter l'espace de recherche, ou même définir son propre algorithme de propagation de la recherche.

5.4 Un exemple pour illustrer le fonctionnement de notre implémentation

Dans l'exemple suivant, nous présentons la représentation Shimple du programme qui sera utilisée pour construire notre graphe de dépendance entre les variables et les champs. L'exemple contient la classe principale *CompositionExample1* qui contient la méthode *main* et aussi deux classes `A` et `B` (nous considérons uniquement les programmes complets).

```
1  package thesisExample1;
2
3  class CompositionExample1 {
4    public static void main(String[] args) {
5      A a1 = new A();
6    }
7  }
8
9  class A {
10   private B b1 = new B();
11 }
12
13 class B {}
```

Analyse du programme : L'API *Soot* nous permet de faire une traversée du programme ; chaque méthode a un corps composé de plusieurs unités. Chaque unité correspond à une instruction du programme. Le pseudo programme suivant correspond à la représentation *Shimple* de notre exemple.

```
1  class thesisExample1.CompositionExample1 extends java.lang.Object {
2    void <init>() {
3      thesisExample1.CompositionExample1 r0;
4      r0 := @this: thesisExample1.CompositionExample1;
5      specialinvoke r0.<java.lang.Object: void <init>()>();
6      return;
7    }
8
9    public static void main(java.lang.String[]) {
10     java.lang.String[] r0;
11     thesisExample1.A $r1, a1;
12     r0 := @parameter0: java.lang.String[];
13     $r1 = new thesisExample1.A;
14     specialinvoke $r1.<thesisExample1.A: void <init>()>();
15     a1 = $r1;
16     return;
17   }
18 }
```

```
1  class thesisExample1.A extends java.lang.Object {
2    private thesisExample1.B b1;
3
4    void <init>() {
5      thesisExample1.A r0;
6      thesisExample1.B $r1;
7      r0 := @this: thesisExample1.A;
8      specialinvoke r0.<java.lang.Object: void <init>()>();
9      $r1 = new thesisExample1.B;
10     specialinvoke $r1.<thesisExample1.B: void <init>()>();
11     r0.<thesisExample1.A: thesisExample1.B b1> = $r1;
12     return;
13   }
14 }
```

L'analyse du programme et la construction du graphe de dépendance entre les variables et champs du programme commence par la méthode d'entrée principale du programme (*Soot* offre la possibilité de spécifier la classe principale du programme qui contient la méthode *main*). Par la suite, l'appel de méthode constitue un point d'entrée à un autre contexte qui sera analysé à son tour.

Dans notre exemple, l'analyse de la méthode *main* de la classe `CompositionExample1` donne les dépendances suivantes :

```
1 <r0 − java.lang.String[]> : 14db0e3 [1982fc1]
2 <@parameter0 − java.lang.String[]> : 676437 [1e4853f]
```

```
1 <$r1 − thesisExample1.A> : 110c31 [1e808ca]
2 <new thesisExample1.A − thesisExample1.A> : 992bae [2bd3a]
```

```
1 <@this − thesisExample1.A> : 1d53f5b [1329642]
2 <$r1 − thesisExample1.A> : 110c31 [1e808ca]
3 <$r1 − thesisExample1.A> : 110c31 [26d607]
```

```
1 <a1 − thesisExample1.A> : fefe3f [1ad98ef]
2 <$r1 − thesisExample1.A> : 110c31 [26d607]
```

Pour chaque variable, nous associons deux codes qui les distinguent dans le programme. Typiquement, Soot nous garantit cette unicité pour tous les points du programme où une variable est définit . Mais, nous avons besoin de garantir l'unicité aussi pour tous les points du programme où on lit la valeur d'une variable. Ainsi, chaque variable a deux codes, le premier le distingue dans le programme (correspond à sa définition), le deuxième code distingue son utilisation. Nous présentons une description des dépendances citées ci-dessus :

- le premier lien associe le paramètre formel de la méthode *main* *@parameter0* avec la variable *r0* définie par *Soot* ;
- le deuxième lien associe la variable `new` (que nous avons créé pour représenter l'opération d'instanciation) avec la variable *$r1* définie par *Soot*. La variable `new` est unique dans le programme ;
- le troisième lien concerne l'appel du constructeur par défaut de la classe `A`. Soot sépare l'instanciation et l'appel du constructeur en deux instructions distinctes. Le paramètre implicite qui représente le `this` est utilisé dans la méthode *init* de la classe `A`. Puisque la variable *$r1* sera utilisée plus tard dans le programme (elle sera affectée à la variable *a1*), nous lui donnons des codes différents pour l'identifier dans les deux points de programme ;

- le quatrième lien associe la variable *$r1* avec la vraiable *a1* définie dans la méthode *main*.

L'analyse de la méthode *init* de la classe `A` donne les dépendances suivantes :

```
<r0 − thesisExample1.A> : 138c63 [165f738]
<@this − thesisExample1.A> : 1d53f5b [1329642]
```

```
<$r1 − thesisExample1.B> : c2b2f6 [149b290]
<new thesisExample1.B − thesisExample1.B> : 16a38b5 [b1074a]
```

```
<@this − thesisExample1.B> : 55a338 [4ee70b]
<$r1 − thesisExample1.B> : c2b2f6 [149b290]
<$r1 − thesisExample1.B> : c2b2f6 [22ab57]
```

```
<b1 − thesisExample1.B> : 1eec35 [25c828]
<$r1 − thesisExample1.B> : c2b2f6 [22ab57]
```

```
<r0 − thesisExample1.B> : 77ef83 [d85cc]
<@this − thesisExample1.B> : 55a338 [4ee70b]
```

Les deux codes que nous associons à chaque variable identifient la copie de la référence que contient cette variable. Cette copie de la référence est unique pour tout le programme. Cela nous permet d'établir des contraintes sur chaque copie de la référence. La résolution de ces contraintes nous donne les solutions possibles pour chaque copie de la référence ; si une solution pour une copie de la référence affirme qu'elle est unique alors le système identifie une relation de composition pour la classe qui déclare le champ qui détient cette copie de la référence.

Définition des contraintes : notre implémentation produit plusieurs résultats dont chacune exprime un contexte de la propagation des contraintes dans le programme. Nous étudions le cas où la relation de composition est confirmée. Nous avons présenté les résultats de l'analyse du programme ; la construction du graphe de dépendance des copies de la référence. Pour des raisons de lisibilité, nous présentons uniquement les dépendances qui concernent le champ *b1* :

```
<$r1 − thesisExample1.B> : c2b2f6 [149b290] : (0, 1, 2):2
<new thesisExample1.B − thesisExample1.B> : 16a38b5 [b1074a] : (2):2
<new thesisExample1.B − thesisExample1.B> : 16a38b5 [e63ab8] : (1):1
```

```
1 <@this − thesisExample1.B> : 55a338 [4ee70b] : (0, 1, 2):1
2 <$r1 − thesisExample1.B> : c2b2f6 [149b290] : (0, 1, 2):2
3 <$r1 − thesisExample1.B> : c2b2f6 [22ab57] : (0, 1, 2):2
```

```
1 <r0 − thesisExample1.B> : 77ef83 [d85cc] : (0, 1, 2):1
2 <@this − thesisExample1.B> : 55a338 [4ee70b] : (0, 1, 2):1
3 <@this − thesisExample1.B> : 55a338 [ac2b57] : (0, 1, 2):1
```

```
1 <b1 − thesisExample1.B> : 1eec35 [25c828] : (0, 2):2
2 <$r1 − thesisExample1.B> : c2b2f6 [22ab57] : (0, 1, 2):2
3 <$r1 − thesisExample1.B> : c2b2f6 [34ec2e] : (0, 1, 2):1
```

Une copie de la référence peut avoir trois états possibles (0 : normale, 1 : temporaire, 2 : unique). La résolution des contraintes nous donne les états pour lesquelles il n'y a pas de conflit avec notre système de contrainte.

- première dépendance : l'operateur d'instanciation `new()` détient une copie de la référence qui est considérée unique. Nous imposons que cette copie de la référence soit temporaire dans le reste du programme. Ce qui nous autorise à propager cette unicité dans le programme ;

- deuxième dépendance : le passage implicite de la variable `this` comme paramètre au constructeur de la classe `B`, ne change pas l'état de la référence qui reste unique. Naturellement, cela est vrai parce que la variable `this` détient une copie de la référence qui est considèrée temporaire et que le autres contraintes ne causent pas de conflit avec cet état ;

- troisième dépendance : puisque la copie de la référence dans la variable `this` est temporaire alors celle de la variable *r0* doit être aussi temporaire. Ce qui est valide et donc pas de conflit ;

- quatrième dépendance : puisqu'il n'y a pas d'autres dépendances, notre système de contraintes autorise le transfert d'unicité au champ *b1*.

Vu que la copie de la référence dans le champ *b1* peut être unique, notre système identifie une relation de composition entre la classe `A` et la classe `B`.

CHAPITRE 6

VALIDATION ET LIMITATIONS

6.1 Validation

La validation de notre implémentation est faite en deux parties. La première partie de l'expérimentation concerne le test d'opérationnalité de notre approche. Nous avons appliqué notre système sur les exemples que nous avons introduits dans le chapitre 3. Les résultats correspondaient parfaitement à nos prévisions et s'ajustaient avec nos choix de modélisation et d'implémentation de la relation de composition. La deuxième partie de l'expérimentation concerne l'application de notre système sur un jeu de test de référence ; le but est de vérifier la cohérence des résultats sur de grands programmes et d'évaluer les performances et les limitations de notre système.

Nous avons présenté dans le chapitre 3 quatre exemples. Nous étudions ci-dessous les résultats obtenus en y appliquant notre système.

Application de notre implémentation sur l'exemple 1, 2, 3 et 4 : notre implémentation identifie les relations de composition dans l'exemple 1, 2 et échoue à identifier ces relations dans les exemples 3 et 4.

- les exemples 1 et 2 représentent les cas où ils existent uniquement des références temporaires avec une seule référence unique ;
- l'exemple 3 représente le cas où il existe plus d'une référence non-temporaire sur un objet ;
- l'exemple 4 représente le cas où une partie est référencée par une autre partie du même tout. Notre implémentation échoue à gérer ce genre de cas.

Cas où un champ de `A` est de type "liste d'éléments de type `B`" : ce cas peut être implémenté avec Java en utilisant un tableau, un conteneur (i.e., `Vector`) ou une liste chainée (où chaque instance possède une référence vers une autre instance). Nous

étudions le cas du conteneur en utilisant un exemple similaire à celui fourni dans l'article de Milanova [Mil05] :

```
class X {
  public void m() {}
}
```

```
class MilanovaCompositionExample extends Object {
  public static void main(String[] args) {
    Vector v = new Vector(5);
    X x = new X();
    v.addElement(x, 0);
    VIterator e = v.elements();
    x = (X) e.nextElement();
    x.m();
  }
}
```

```
class Vector extends Object {
  protected Object[] data;
  public int count;

  public Vector(int size) {
    data = new Object[size];
    count = 0;
  }

  public void addElement(Object e, int at) {
    data[at] = e;
    count++;
  }

  public Object elementAt(int at) {
    return data[at];
  }

  public VIterator elements() {
    return new VIterator(this);
  }
}
```

```
final class VIterator extends Object {
  Vector vector;
  int count;

  public VIterator(Vector v) {
    vector = v;
```

```
7       count = 0;
8     }
9
10    public Object nextElement() {
11      Object[] data = vector.data;
12      int i = this.count;
13      this.count++;
14      return data[i];
15    }
16
17    public boolean hasMoreElements() {
18      return (vector ≠ null && vector.data ≠null && count < vector.count);
19    }
20  }
```

Nous nous focalisons sur les références du champ *data* de la classe `Vector` et du champ *vector* de la classe `VIterator`.

Nous faisons une distinction entre l'objet tableau référencé par le champ *data* et les objets qui sont éléments du tableau référencés par une variable virtuelle *data[0]* ; nous rappelons que les assignations et les lectures des éléments du tableau sont remplacées par des assignations et des lectures du premier élément du tableau.

Notre implémentation identifie une relation de composition pour les champs *data*, *data[0]* et *vector*.

Comme nous l'avons mentionné dans différentes sections des chapitres précédents, nous nous intéressons uniquement aux utilisations des références ; cela explique les résultats obtenus. Nous n'offrons pas dans notre implémentation une solution directe pour gérer les cas où un champ de `A` est de type "liste d'éléments de type `B`", ainsi notre implémentation ne permet pas d'inférer une relation de composition entre un conteneur et ses composants : elle pourrait inférer qu'il y a une relation de composition entre `A` et le conteneur, mais probablement pas entre le conteneur et `B`, et encore moins entre `A` et `B`.

Application de notre implémentation sur la bibliothèque Zip du SDK Java (version 1.6 update 6) et JUnit (version 3.8.1) : Nous avons choisi d'utiliser la méthode *main* fournie dans l'article de Milanova [Mil05] pour exécuter notre test sur la bibliothèque Zip :

```
1  class Main {
2    public static void main(String[] args) {
3      ZipEntry ph_ZE;
4      ZipInputStream ph_ZIS;
5      ZipOutputStream ph_ZOS;
6
7      ph_ZE = new ZipEntry("");
8      ph_ZIS = new ZipInputStream(null);
9      ph_ZOS = new ZipOutputStream(null);
10     ph_ZE.setCrc(0);
11
12     try {
13       ph_ZE = ph_ZIS.getNextEntry();
14       ph_ZOS.putNextEntry(ph_ZE);
15       ph_ZOS.closeEntry();
16       ph_ZOS.finish();
17     }
18     catch (Exception e) {
19       e.printStackTrace();
20     }
21   }
22 }
```

Le but principal de ces expérimentations est de valider l'opérationnalité de notre implémentation sur de vrais programmes. La partie analyse produit des résultats corrects. Les performances d'exécutions sont bonnes, cela prend globalement moins de 60 seconds pour faire une analyse complète du programme. La deuxième partie qui concerne la résolution des contraintes a cependant quelques limites de performance. Le temps de résolution des contraintes est long mais nous estimons que ce temps peut être réduit significativement en implémentant un algorithme adapté à nos contraintes ; la résolution des contraintes ne doit pas nécessairement vérifier tous les cas possibles, il est possible de limiter le champ de recherche des solutions et cela peut aider à diminuer le temps de calcul. Concernant l'exactitude des résultats, il n'y a pas de grande différence par rapport aux résultats obtenus pour les exemples de tests ; la limite principale de notre implémentation est liée au fait que nous considérons toutes les références des champs comme non-temporaires.

6.2 Limitations

Les liens que nous avons créés dans la section précédente nous permettent de construire des contraintes sur les propriétés des références dans un programme ; la résolution de ces contraintes nous donne les différentes solutions que peut avoir une référence. Un point très important à considérer dans la résolution des contraintes est l'aspect temporaire d'une référence. Nous avons choisi de considérer les champs comme étant non-transitoires. Cette restriction nous empêche de considérer une référence d'un champ comme étant temporaire ; par conséquent, notre système va échouer à identifier une relation de composition si deux champs pointent sur le même objet. Ce choix est tout à fait correct en général, néanmoins il a quelques limitations ; à savoir qu'on ne peut pas identifier le cas où une partie est référencée par des objets qui sont eux mêmes parties du tout. Dans l'exemple 4 du chapitre 3, on peut remarquer facilement que `A` est en relation de composition avec `C`, donc `A` est aussi en relation de composition avec `B`, mais notre système va identifier uniquement une relation de composition entre `A` et `C` et va échouer à inférer une relation de composition entre `A` et `B` car l'instance de `B` est référée aussi dans `C`.

Il y a plusieurs approches pour résoudre ce problème, par exemple Milanova [Mil05] se base sur la notion de dominance d'un nœud sur d'autres nœuds dans un graphe pour définir une frontière d'accès à un objet (cela permet d'ignorer les appels à l'intérieur de cette frontière), aussi elle utilise une technique de simplification du graphe en supprimant tous les sous graphes issus d'une création ou utilisation temporaire d'une variable. Une extension possible de notre approche serait de pouvoir accepter la référence d'un champ comme transitoire si les références vers son objet parent sont toutes transitoires.

6.3 Solutions conservatrices pour l'analyse des cas particuliers

L'API Soot nous donne une grande flexibilité à analyser le programme et offre différentes fonctions pour manipuler les différents éléments du programme (classes, méthodes, champs, variables et instructions). Cependant, nous avons choisi de considérer des approches conservatrices pour gérer certains cas particuliers comme les tableaux et la portée dynamique.

Tableaux : nous avons considéré un tableau comme un simple objet ; les instructions de lecture et d'écriture dans un élément d'un tableau sont traitées respectivement comme une lecture et écriture dans la variable ou le champ qui correspond à la déclaration du tableau.

Portée Dynamique : la portée d'une déclaration est déterminée seulement à l'exécution du programme. Par exemple, avec :

```
1  void main(X r) {
2    r.bar();
3  }
```

il est difficile de savoir statiquement quelle méthode sera invoquée. Si on considère que `Y` est une sous-classe de `X` et que `Y` redéfinit la méthode *bar* alors on ne connaîtra le type de la référence de la variable *r* seulement à l'exécution.

Dans notre analyse, nous considérons la définition de la méthode dans le type courant (s'il s'agit d'une classe) ainsi que toutes les redéfinitions de cette méthode dans les sous types du type courant (ou les implémentations de la méthode au cas d'une interface).

Chargement dynamique : ce cas est très compliqué à gérer ; en fait, le type est connu à l'exécution du programme. Il peut s'agir d'une classe du programme, ou peut-être importée d'un autre module ou bibliothèque. Par exemple, l'utilisation de la méthode `java.lang.Class.forName(java.lang.String)`, le paramètre de cette méthode peut êre inconnu à la compilation, donc il serait impossible de résoudre statiquement un appel de méthode. Une possible manière de gérer ce problème est de vérifier si le type est spécifié explicitement sous forme de chaîne de caractères dans le programme ou parfois exploiter les instructions de coercition de type pour donner une approximation du type d'objets créés dynamiquement [ALSU07].

CHAPITRE 7

CONCLUSION ET PERSPECTIVES

Ce mèmoire présente une nouvelle approche pour identifier statiquement la relation de composition dans le code source. Nous avons proposé une définition de la relation de composition qui permet de l'appliquer directement dans le langage de programmation Java sous forme d'annotations. Ainsi, notre approche permet aux différents intervenants dans la mise en œuvre d'un programme d'intervenir pour annoter les champs du programme où on espère conserver l'unicité des références vers des parties ; ces champs seront annotés comme uniques. Aussi, la possibilité d'annoter les autres champs comme temporaires.

Nous avons implémenté un système pour prouver la validité de notre concept. Nous avons utilisé Soot qui nous a permis d'avoir une grande flexibilité dans l'analyse du code. Notre système a été testé sur des programmes complets et les résultats obtenus correspondaient bien à ceux attendus. Un apport principal de notre système est sa capacité à donner une explication pour chaque solution. Ceci est basé principalement sur notre définition de la relation de composition et aussi sur les contraintes que nous avons établies sur l'utilisation des références. Ainsi, une personne qui effectue l'analyse du programme pourra savoir pourquoi un champ a été identifié comme élément d'un tout et donc correspond à une relation de composition et aussi savoir pourquoi le système a échoué à identifier une relation de composition pour un champ donné. Cette information peut être utile et importante pour faire des ajustements dans le programme pour mieux annoter les champs du programme. Par exemple, un champ dont la durée de vie est temporaire pourra être annoté comme transitoire et donc aider le système à identifier une relation de composition.

Notre travail futur consistera à autoriser des champs pour qu'ils soient temporaires (principalement le cas où une partie d'un tout référence une autre partie du même tout), gérer le cas des conteneurs et faire d'autres tests sur des benchmarks pour mieux expérimenter et valider notre approche.

BIBLIOGRAPHIE

[ALSU07] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilateurs : principes, techniques et outils - 2e édition*. Pearson Education, 2007.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Phd thesis, University of Copenhagen, 1994.

[BHSPLB03] F. Barbier, B. Henderson-Sellers, A.L. Parc-Lacayrelle, and J.-M. Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE Transactions on Software Engineering*, 29(5) :459–470, 2003.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Brown University*, 1991.

[CR07] S. Cherem and R. Rugina. Uniqueness inference for compile-time object deallocation. *Proceedings of the 6th international symposium on Memory management*, pages 117–128, 2007.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, 1994.

[Ern03] M. D. Ernst. Static and dynamic analysis : synergy and duality. *In Proceedings of the ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[GAA04] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships : Putting icing on the uml cake. *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2004.

[GCGZ06] J. Gensel, C. Capponi, P. Genoud, and D. Ziébelin. Vers une intégration des relations partie-tout en arom. *In Langage et Modèles à Objets*, pages 53–70, 2006.

[Gro07] Object Management Group. Unified modeling language specification, v2.1.2. 2007.

[JR00] D. Jackson and M. Rinard. Software analysis : a roadmap. *Proceedings of the Conference on The Future of Software Engineering*, pages 133–145, 2000.

[KG01] R. Kollmann and M. Gogolla. Application of uml associations and their adornments in design recovery. *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 81–91, 2001.

[Kil99] H. Kilov. *Business Specifications : The Key to Successful Software Engineering, 1st edition.* Prentice Hall, 1999.

[Lho02] O. Lhoták. A flexible points-to analysis framework for java. Master's thesis, McGill University, 2002.

[MF07] K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for java. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 423–440, 2007.

[Mil05] A. Milanova. Precise identification of composition relationships for uml class diagrams. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 76–85, 2005.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[Mul97] P.-A. Muller. *Modélisation objet avec UML.* Eyrolles, 1997.

[VRCG+99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13, 1999.

[WCH87] M. Winston, R. Chaffin, and D. Herrmann. A taxonomy of part-whole relations. *Cognitive Science*, 11 :417–444, 1987.

[YSC+05] D. Yeh, P.-C. Sun, W. Chu, C.-L. Lin, and H. Yang. An empirical study of a reverse engineering method for aggregation relationship based on operation propagation. *Proceedings of the 29th Annual International Computer Software and Applications Conference*, 1 :95–100, 2005.