UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PRÉVENTION DE DÉRÉFÉRENCEMENTS DE NUL

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

JEAN-SÉBASTIEN GÉLINAS

JUIN 2012

REMERCIEMENTS

Ce mémoire a été réalisé au LAboratoire de recherche sur les TEchnologies du Commerce Électronique (LATECE) ainsi qu'en tant que membre du Groupe de recherche sur l'étude, la spécification et l'implémentation des langages informatique (GRÉSIL) de l'Université du Québec À Montréal (UQAM). Je tiens donc à remercier les directeurs de ces deux groupes : prof. Hafedh Mili et prof. Étienne M. Gagnon.

Je désire aussi remercier le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) pour la bourse que prof. Étienne M. Gagnon a pu m'accorder grâce à ses subventions de recherche.

Je tiens de plus à remercier mon directeur de recherche prof. Étienne M. Gagnon ainsi que mon codirecteur prof. Jean Privat. Sans leur support et leur patience, jamais je n'aurais pu terminer ce mémoire.

À mes amis, à mes proches et à ma famille, je vous remercie de votre tolérance envers mon manque de disponibilité, c'est maintenant terminé ;)

Finalement, j'adresse mes remerciements aux membres du jury et aux membres de l'administration du programme de la maîtrise en Informatique.

TABLE DES MATIÈRES

TABLE DES FIGURES

RÉSUMÉ

La bonne gestion des déréférencements de nul dans les langages à objets statiquement typés est un problème complexe qui est loin d'être nouveau. Plusieurs approches existent, mais aucune n'est parfaite. Les diverses solutions au déréférencement de nul se partagent trois dimensions : (1) la simplicité, (2) l'exactitude des résultats et (3) le typage statique. Présentement, à notre connaissance, aucune solution ne présente ces trois dimensions.

Un des problèmes majeurs de la gestion des déréférencements de nul est la gestion des attributs, notamment lors de l'instantiation des objets.

L'approche que nous présentons ici pour gérer les déréférencement de nul est une approche (1) simple et (2) exacte. Bien que notre approche présente un composant de typage statique, nous ajoutons aussi des tests dynamiques pour la compléter, ce qui fait qu'elle ne respecte pas totalement les trois dimensions.

Nous introduisons aussi des analyses statiques pour détecter l'initialisation d'attributs dans les constructeurs pour ensuite, à l'aide d'optimisations, supprimer les tests dynamiques que nous avons ajoutés à la phase précédente.

Pour un programme de grande taille (110.000 lignes de code), nous parvenons, avec nos analyses et nos optimisations, à supprimer 100% des tests dynamiques que nous avions introduits, tout en garantissant de façon statique qu'aucune erreur en lien avec le déréférencement de nul ne pourra se produire lors de l'exécution du programme. En conséquence, notre solution améliorée par les analyses statiques proposées permet de présenter les trois dimensions recherchées pour un sous-ensemble des programmes, ce qui fait un compromis très intéressant.

Mots clés : compilation, analyse statique, optimisation, types nullables, validation d'attributs, Nit

NOTATIONS

Dans cette section, nous allons présenter quelques notations que nous allons utiliser dans ce document.

Notation	Exemple	Explication
$\exists!$	$\exists! z \in E \vert ...$	Il existe un z unique élément de E tel que ...
$\wedge$	$x \wedge y$	x et y
$\vee$	$x \vee y$	x ou y
$f : X \to Y$	$f : V \to W$	La fonction f a comme domaine V et comme co-domaine W
X^+	$f : V^+ \to W$	La fonction f a comme domaine un n-uplet composé d'un nombre positif de valeurs contenues dans V et comme co-domaine W
X^z	$f : V^3 \to W$	La fonction f a comme domaine un tuple composé de 3 valeurs contenues dans V et comme co-domaine W, synonyme de $f : V \times V \times V \to W$
X_y	T_i	i^e élément du n-uplet T
$x \in K$	soit $K = (V, \sqsubseteq)$, $x \in V$	L'élément x est membre de l'ensemble des éléments du treillis K
$\forall_{x=1}^n$	$\forall_{x=1}^n$	$\forall x$ tel que $1 \leq x \leq n$

CHAPITRE I

INTRODUCTION

La bonne gestion des déréférencements de nul est un problème complexe qui est loin d'être nouveau. Plusieurs approches existent (Chalin et James, 2007; Evans, 1994; Evans, 1996; Rutar, Almazan et Foster, 2004; Hubert, 2008; Hovemeyer, Spacco et Pugh, 2006; Hovemeyer et Pugh, 2007; Male et al., 2008; Evans et al., 1994), mais aucune n'est parfaite. Les diverses solutions au déréférencement de nul se partagent trois dimensions : (1) la simplicité, (2) l'exactitude des résultats et (3) l'utilisation d'une composante de typage statique plutôt que des tests dynamiques.

La solution que nous proposons est basée sur un typage statique strict et sur des tests dynamiques ajoutés automatiquement par le compilateur à des endroits clés dans le programme. Le but principal de ces derniers est de s'assurer de la bonne construction des objets ainsi que d'empêcher la déconstruction [1] de ceux-ci.

À ce jour, au meilleur de nos connaissances, aucune solution ne répond à chacune de ces trois dimensions. La technique présentée ici s'attaque directement à deux de ces dimensions : elle est (1) une solution simple et (2) une solution exacte. La troisième

---

1. La déconstruction d'un objet est le fait de prendre un objet complètement construit et de le rendre dans un état qui est incohérent pour un objet construit. Un exemple de ceci est de prendre un attribut non-nullable, un attribut qui ne peut pas contenir la valeur *null*, et de faire en sorte qu'il contienne cette valeur.

dimension, le fait qu'elle utilise du typage statique, n'est que partiellement respectée puisque la technique ajoute des tests dynamiques aux programmes.

En plus de présenter cette nouvelle solution, nous montrerons à travers ce document que les tests dynamiques ajoutés sont généralement enlevés par une optimisation ajoutée au compilateur. En fait, nous montrons qu'avec cette optimisation, notre approche réussit à couvrir les trois dimensions pour certains programmes. En particulier, nos expérimentations montrent que cette couverture est assurée pour un programme non-trivial de plus de 100.000 lignes.

1.1 Présentation détaillée

Cette section se veut un exemple de ce que l'approche présentée dans ce document peut produire comme résultat. Plus précisément, seront présentés : un exemple de programme à analyser, le même programme une fois les tests dynamiques ajoutés, les résultats de l'optimisation de suppression de tests dynamiques et, finalement, le programme une fois optimisé.

La figure 1.1 présente le programme qui est utilisé tout au long de cette section. Il est écrit en Nit, tel qu'il serait écrit par un programmeur, et introduit une classe Personne, ayant les attributs _nom, _telephone et _age. En plus du constructeur de la classe (méthode `init`), une seconde méthode est présente : `to_string`.

La figure 1.2 présente le programme une fois que les tests dynamiques ont été ajoutés. Il est écrit en Nit et est une interprétation de ce que le compilateur génère. Les tests numérotés 1, 2 et 3 sont des tests ajoutés à la fin de la construction de l'objet, tandis que les tests 4, 5 et 6 sont ajoutés avant l'accès en lecture à un attribut.

Cette section présente les résultats de l'optimisation de suppression de tests dynamiques lorsqu'elle est effectuée sur le programme présenté dans les figures 1.1 et 1.2. Voici les lignes qui sont optimisées, donc les tests qui sont enlevés :

```
class Personne
    var _nom: String
    var _telephone: String
    var _age: Int

    init(nom: String, telephone: String, age: Int) do
        _nom = nom
        _telephone = telephone
        _age = age
    end

    fun to_string: String do
        var str = ""
        str = str + "Nom: " + _nom + "\n"
        str = str + "Age: " + _age.to_s + "\n"
        str = str + "Telephone: " + _telephone
        return str
    end
end

var p = new Personne("Jonh Doe", "1-555-555-5555", 99)
print (p.to_string)
```

FIGURE 1.1: Programme Nit, classe Personne telle qu'écrite par un programmeur

- constructeur `Personne::init` : les tests #1, #2, #3 sont enlevés puisque les attributs `_nom`, `_telephone` et `_age` sont toujours initialisés ;
- méthode `Personne::to_string` : les tests #4, #5, #6 sont enlevés puisqu'il est impossible que cette méthode soit appelée pendant la construction d'un objet de type `Personne` dans ce programme.

La figure 1.3 présente le programme une fois optimisé. Il est écrit en Nit et est une interprétation de ce que le compilateur génère. On voit clairement que tous les tests qui étaient ajoutés par le compilateur sont maintenant enlevés.

Bien que l'exemple choisi soit simple, il est, selon l'auteur, représentatif du genre de code trouvé dans la partie métier d'un système. Les résultats présentés montrent que, dans cet exemple, tous les tests ajoutés par l'approche présentée ont été enlevés. Puisque tous les tests dynamiques ont été retirés, il est impossible que le programme

```
class Personne
    var _nom: String
    var _telephone: String
    var _age: Int

    init(nom: String, telephone: String, age: Int) do
        _nom = nom
        _telephone = telephone
        _age = age
        assert isset _nom           #1
        assert isset _telephone     #2
        assert isset _age           #3
    end

    fun to_string: String do
        var str = ""
        assert isset _nom           #4
        str = str + "Nom: " + _nom + "\n"
        assert isset _age           #5
        str = str + "Age: " + _age.to_s + "\n"
        assert isset _telephone     #6
        str = str + "Telephone: " + _telephone
        return str
    end
end

var p = new Personne("Jonh Doe", "1-555-555-5555", 99)
print (p.to_string)
```

FIGURE 1.2: Programme Nit, classe Personne avec tests dynamiques ajoutés

arrête son exécution dû à un test échoué. Ainsi, le programmeur obtient une sûreté supplémentaire.

1.2 Contributions

Les principales contributions de ce mémoire sont :

- la présentation d'une nouvelle solution pour la gestion des références nulles efficace et facile à utiliser pour le programmeur ;
- l'introduction d'une nouvelle analyse interprocédurale et des preuves associées

```
class Personne
    var _nom: String
    var _telephone: String
    var _age: Int

    init(nom: String, telephone: String, age: Int) do
        _nom = nom
        _telephone = telephone
        _age = age
    end

    fun to_string: String do
        var str = ""
        str = str + "Nom: " + _nom + "\n"
        str = str + "Age: " + _age.to_s + "\n"
        str = str + "Telephone: " + _telephone
        return str
    end
end

var p = new Personne("Jonh Doe", "1-555-555-5555", 99)
print (p.to_string)
```

FIGURE 1.3: Programme Nit, classe Personne une fois optimisée

pour connaître les attributs initialisés. Nous avons nommés cette analyse "analyse des attributs initialisés" et celle-ci repose sur deux autres analyses que nous présentons et prouvons : (1) analyse des références définitives vers l'objet en construction et (2) analyse de références définitives vers un objet construit ;

- l'introduction d'une nouvelle optimisation des tests dynamiques de validation d'initialisation des attributs ;
- la présentation de preuves de validité des algorithmes généraux d'analyse statique intraprocédurales [2] ;
- l'introduction d'une nouvelle méthode d'analyse interprocédurale avec les preuves reliées ;

2. Les preuves en soit doivent exister quelque part. Par contre, nous n'avons pas trouvé d'ouvrages qui les présentaient.

- l'implémentation de la solution et des analyses proposées ainsi que la collection de données empiriques.

1.3 Structure de ce mémoire

Le reste de ce document est divisé comme suit : le second chapitre présente l'approche proposée pour gérer la partie statique des déréférencements de nul. Cette approche est basée sur un typage statique strict ainsi que sur des tests dynamiques localisés.

Le troisième chapitre présente les notions mathématiques de relation d'ordre et de treillis. Ces deux notions sont cruciales à la bonne présentation des chapitres suivants.

Le quatrième chapitre présente plusieurs notions importantes. Plus précisément, le langage Nit, les graphes de flux de contrôle ainsi que les graphes d'appels y sont présentés.

Les chapitres cinq et six présentent respectivement la base des analyse statiques intraprocédurales, et la base des analyses statiques interprocédurales.

À partir de ces notions, le chapitre sept présente la technique que nous utilisons pour supprimer les tests dynamiques, plus précisément : les trois analyses statiques introduites par notre approche ainsi que l'optimisation qui utilise leurs résultats. Les trois analyses qui y sont présentées sont :

- l'analyse des références définitives vers l'objet en construction ;
- l'analyse des références définitives vers un objet qui n'est pas en construction ;
- l'analyse des attributs initialisés.

Finalement, les résultats expérimentaux ainsi que les détails d'implémentation et limitation de notre approche sont discutés dans le chapitre huit.

Le dernier chapitre présente quelques travaux reliés à notre approche, nos analyses et notre optimisation.

CHAPITRE II

PRÉVENTION DE DÉRÉFÉRENCEMENTS DE NUL

Ce chapitre présente le problème de la gestion des références nulles ainsi que l'approche que nous proposons pour prévenir les déréférencements de nul dans les langages à objets. Cette dernière consiste en deux composantes : (1) un typage statique strict et (2) des tests dynamiques localisés. L'utilisation de ces deux composantes permet l'expressivité et la robustesse de l'approche.

La première section présente la problématique de la gestion des références nulles. Les sections suivantes présenteront l'approche proposée pour résoudre cette problématique : la seconde section présente la composante de typage statique tandis que la troisième présente la raison d'être des tests dynamiques ainsi que leur composition. Finalement, la quatrième section présente les garanties apportées par l'approche.

2.1 Le problème de la gestion de références nulles

Une référence est dite nulle lorsqu'elle ne contient pas d'objet. Dans certains cas, ce sera lorsqu'une référence n'a pas été initialisée, dans d'autres lorsque la valeur *null* a été affectée à la référence, que ce soit de façon directe, à partir de la valeur *null*, ou à partir d'une autre référence elle même nulle. Une référence nulle est problématique puisqu'elle ne référencie pas d'objet et ne peut donc pas être receveur d'un appel de méthode ou d'un accès d'attribut. Si une référence nulle est la cible d'un appel de méthode ou d'un accès à un attribut, une exception est lancée lors de l'exécution, un

```
class MaClasse
       var attr1: Int
       init do                    #2
              # ici, attr1 n'est pas initialise
       end

       fun work do                #4
              attr1 = attr1 + 1 # Boum car attr1 n'est pas initialise
       end
end

# Ailleurs ...
var obj: MaClasse = new MaClasse # 1
obj.work                         # 3
```

FIGURE 2.1: Exemple de problème de gestion de références nulles pour un attribut

arrêt brutal de l'application en cours est effectué ou un comportement indéterminé est effectué, selon le langage.

Le problème de la gestion des références nulles, plus particulièrement dans les constructeurs, est illustré dans la figure 2.1. On y voit une classe, ayant un seul attribut, ainsi qu'un constructeur qui n'initialise pas cet attribut. On voit par la suite un appel à une méthode d'une instance de cette classe qui va causer une erreur lors de l'exécution (la séquence d'exécution est indiquée par les chiffres en commentaire à droite des lignes).

À la base du problème de la gestion de références nulles se trouve un problème de typage : comment savoir quelles références sont nulles ou possiblement nulles et comment garantir la bonne gestion de ces références. Même avec un système de type très strict, tel qu'il sera présenté dans le chapitre 9, il est difficile de garantir la bonne construction d'objets, plus particulièrement l'initialisation des attributs non-nullable. La solution que nous proposons est composée de 2 parties :

1. un système de type statique qui tient en compte les références nullables ;
2. des tests dynamiques injectés dans le code par le compilateur à certains endroits stratégiques.

2.2 Typage statique

Cette section présente la composante de typage statique de l'approche proposée. Cette composante va s'assurer de la validité des affectations entre les variables, paramètres et attributs. Elle va notamment s'assurer que la valeur null n'est pas affectée à des endroits où elle n'est pas valide.

Plus précisément, cette section présente le système de types, les restrictions sur les objets définis statiquement comme étant nullables, les façons dont le programmeur peut changer le type statique de nullable vers non-nullable ainsi que les effets du système de type sur l'expressivité du langage.

2.2.1 Système de types

Dans le système de types proposé, se retrouve, pour chaque classe dans le logiciel, deux types. L'un d'eux sera le type nullable, l'autre le type non-nullable. Le type non-nullable sera un sous-type du type nullable puisqu'il accepte moins de valeurs que celui-ci : la valeur *null* n'est pas une valeur valide pour une référence dont le type est défini comme étant non-nullable alors que toute autre valeur est valide.

Syntaxiquement, pour indiquer au système de type qu'il veut utiliser la version nullable d'un type, le programmeur doit annoter le type utilisé avec le mot clé *nullable*. La figure 2.2 présente un exemple de déclaration d'attributs avec types nullables : deux attributs y sont visibles, l'un ne pouvant pas contenir la valeur null (_attr1), l'autre pouvant la contenir (_attr2).

La figure 2.3 présente la façon dont le sous-typage fonctionne en cas d'héritage. Plus précisément, elle présente les types associés à trois classes (*Object*, *Collection* et *HashSet*), pour démontrer qu'en cas d'héritage, les types nullables sont sous-types des types nullables associés aux classes parents et que les types normaux sont sous-types des types normaux des classes parents.

Soit la relation de sous-typage entre les types X et Y, X étant un sous-type de

```
class MaClass
       var _attr1: Int
       var _attr2: nullable Int
end
```

FIGURE 2.2: Déclaration de deux attributs en Nit, l'un ne pouvant pouvant pas contenir la valeur null (attr1), l'autre pouvant la contenir (attr2)

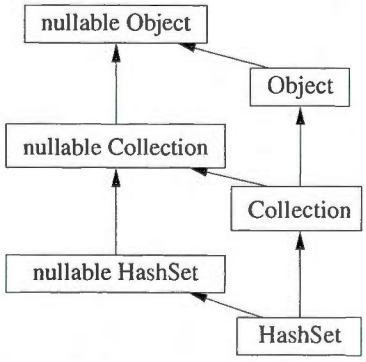

FIGURE 2.3: Sous-typage dans l'approche proposée, présentation des classes Object, Collection, HashSet et de leur pendant *nullable*

```
var a : Int
var b : nullable Int
a = 1
b = 1
b = null
# a = null    -> Refuse lors de la compilation
# a = b       -> Refuse lors de la compilation
```

FIGURE 2.4: Exemple de restriction d'affectation d'objets nullables, où l'affectation d'un entier est valide autant vers une référence *nullable* que *non-nullable*, et que l'affectation de la valeur explicite *null* est possible vers une référence *nullable*, mais impossible vers une référence *non-nullable*

Y, noté $X <: Y$. Pour chaque type t et son type nullable n(t) nous posons formellement les règles suivantes :

$$\forall t, \quad t <: n(t)$$

et

$$\forall t, t', \quad t <: t' \Leftrightarrow n(t) <: n(t')$$

2.2.2 Restrictions sur un objet nullable

Tout objet dont le type statique est défini comme étant nullable est sujet aux restrictions suivantes :

- restriction d'affectation ;
- restriction de passage de paramètre ;
- restriction d'accès aux attributs ;
- restriction d'appel de méthode.

La restriction d'affectation (figure 2.4) est une conséquence directe du système de types : la version non-nullable du type étant sous-type à la version nullable, il est impossible d'affecter directement un objet nullable à une variable non-nullable. Par contre, l'affectation d'un objet non-nullable à une variable nullable est possible.

```
fun foo (a: Int) : Int do return a
fun bar (a: nullable Int) : nullable Int do return a

var a: Int
var b: nullable Int
a = 2
b = 2

foo(a)
foo(2)
#foo(b)       -> Refuse lors de la compilation

bar(a)
bar(2)
bar(b)
```

FIGURE 2.5: Exemple de restriction de passage de paramètres d'objets nullables, où un appel vers une méthode avec paramètre *nullable* peut être effectué avec une variable *non-nullable*, *nullable* ou un entier, et où un appel de méthode avec paramètre *non-nullable* peut être effectué avec une variable *non-nullable* ou un entier mais pas avec une variable *nullable*

```
class MaClass
	var attr1: Int
	var attr2: nullable Int

	init do
		attr1 = 2
	end
end

var first: nullable MaClass = null
var second: MaClass = new MaClass

#var a: Int = first.attr1          -> Refuse lors de la compilation
#var b: nullable Int = first.attr2 -> Refuse lors de la compilation

var a: Int = second.attr1
var b: nullable Int = second.attr2
```

FIGURE 2.6: Exemple de restriction d'accès aux attributs d'objets nullables, où il est impossible d'accéder aux attributs d'une variable *nullable*

```
var a: Object
var b: nullable Object
a = 2

print a.to_s
# print b.to_s          -> Refuse lors de la compilation
```

FIGURE 2.7: Exemple de restriction d'appels sur un objet nullable, où il est impossible de faire un appel de méthode sur une variable *nullable*

```
class MaClasse
       var attr: nullable Int

       init do
              attr = 1
       end
end

var obj: MaClasse = new MaClasse
#var x: Int = obj.attr + 1   -> Refuse lors de la compilation
var x: Int = obj.attr.as(not null) + 1
```

FIGURE 2.8: Exemple d'adaptation de type par coercition de type descendante

```
var a: nullable Int = null
var b: nullable Int = a
# b = a + 1          -> Refuse lors de la compilation
if a != null then b = a + 1
```

FIGURE 2.9: Exemple d'adaptation de type par test dynamique

La restriction de passage de paramètre (figure 2.5) est aussi une conséquence directe du système de types puisque le passage de paramètre doit lui aussi respecter le système de types.

Les restrictions d'accès aux attributs (figure 2.6) et d'appels de méthode (figure 2.7) sont liées puisqu'elles font toutes deux référence à un accès à une propriété de l'objet. Lors d'un appel à une propriété d'une référence nullable, il est impossible de garantir statiquement l'inexistence de déréférencement de nul.

2.2.3 Adaptation de type statique

Il existe deux façons d'adapter le type statique d'une référence nullable :

- coercition de type descendante (*cast*) ;
- test dynamique.

```
class MaClasse
	var attr: nullable Int = 1

	fun fonction do
		var a: Int = 0
		# if attr != null then a = attr + 1 -> Refuse a la compilation
		var b: nullable Int = attr
		if b != null then a = b + 1
	end
end
```

FIGURE 2.10: Exemple d'adaptation de type par test dynamique pour un attribut

Pour utiliser la coercition de type descendante, le programmeur doit utiliser la construction inst.as(class) où class est un nom de classe ou l'argument explicite not null, et inst est une référence. Le cas qui nous intéresse ici est celui où l'argument est not null et que l'instance est une variable ou un paramètre défini statiquement comme étant nullable. Dans ce cas, le type statique de la valeur de retour de cette instruction sera le type non-nullable associé au type de la variable *inst*. Une démonstration de cette technique est présentée dans la figure 2.8.

La seconde façon de faire consiste à garantir au compilateur que le type dynamique de la variable ou du paramètre sera bien un type non-nullable. Pour ce faire, le programmeur doit faire un test explicite de comparaison entre la valeur null et la référence en question, tel que montré dans la figure 2.9. Une fois le test effectué, le type statique de la variable sera changé dans le bloc correspondant du test :

- si le test est un if (v != null), c'est le bloc de code dans la partie then du if qui sera changé ;
- si le test est un if (v == null), c'est le bloc de code dans la partie else du if qui sera changé.

Il est important de noter que cette seconde technique n'est pas valide avec les attributs. En fait, elle n'est valide qu'avec les variables locales et les paramètres de

fonction. La raison de cette particularité est qu'il est impossible de s'assurer qu'un attribut n'est pas modifié après le test, tandis qu'il suffit d'une simple analyse intra-procédurale pour s'assurer qu'une variable ou un paramètre ne sont pas modifiés. Ainsi, la façon privilégiée pour passer outre cette restriction est d'affecter le contenu de l'attribut dans une variable avant de faire l'appel ou l'affectation (tel que montré dans la figure 2.10). Cette façon de faire est d'ailleurs préconisée dans la majorité des langages à objets, tout particulièrement dans le cas de code exécuté dans un environnement à mémoire partagée pour garantir que la valeur de la référence n'a pas été modifiée entre le test et l'utilisation.

2.2.4 Effet sur l'expressivité

Cette section démontre que l'approche proposée n'a aucun effet sur l'expressivité originale d'un langage, qu'elle ne restreint en rien l'écriture d'un programme et qu'elle ne *coûte* que si l'on s'en sert.

Soit deux langages L et L', le second étant le premier augmenté de l'approche de types nullables proposée. Prenons un programme p écrit dans le langage L, et prouvons que l'on peut faire une transformation en deux étapes pour avoir un programme p' équivalent écrit dans le langage L'. Les deux étapes de la transformation à effectuer sur tout le programme sont :

1. annoter chaque type explicite avec le qualificatif nullable ;
2. modifier chaque appel de méthode et chaque accès à un attribut pour ajouter `.as(not null)`, par exemple, `a.x` deviendrait `a.as(not null).x`.

En effectuant ces deux étapes sur tout le programme, on obtient un programme p' ayant le même comportement que le programme p, mais qui est écrit en utilisant l'approche proposée. Cette démarche montre que l'approche proposée n'enlève rien à l'expressivité du langage, mais aussi qu'une utilisation trop simpliste n'apporte rien, si ce n'est que du code inutile.

```
class MaClasse
      var attr: Int

      init (other: MaClasse) do
            other.attr = self.attr # statiquement valide

            print other.attr # Boom! dynamiquement
      end
end
```

FIGURE 2.11: Exemple de déconstruction d'objet possible s'il n'y avait pas les tests dynamiques où un objet construit voit l'un de ses attributs non-nullable devenir *null*

2.3 Tests dynamiques

Cette section présente la composante de tests dynamiques de l'approche proposée. Elle est nécessaire pour garantir la bonne utilisation d'attributs lors de la création d'une instance : elle s'assure qu'un attribut accédé est bien initialisé et que tous les attributs sont initialisés à la fin de la construction d'un objet. Elle prévient aussi la déconstruction d'objets.

Plus précisément, cette section présente en premier lieu la notion de déconstruction d'objet. Par la suite, les deux types de tests dynamiques ajoutés par l'approche proposée seront exposés. La troisième section présente les garanties apportées par ces tests. Finalement, un opérateur pour effectuer manuellement un test dynamique pour valider l'initialisation d'un attribut sera introduit.

2.3.1 Déconstruction d'objet

La figure 2.11 présente un exemple de déconstruction d'objet. L'objet reçu en paramètre au constructeur est *déconstruit* en affectant une valeur non initialisée à un attribut qui était initialisé. Les tests dynamiques vont lancer une exception automati-

```
class MaClasse
       var attr: Int

       init do
              if 1.0.rand > 0.5 then
                     attr = 2
              end
       end
end
```

FIGURE 2.12: Exemple de constructeur qui ne peut pas être validé statiquement

quement lorsque l'attribut non-initialisé sera accédé en lecture, lors de l'exécution de la ligne self.attr, et vont donc prévenir ce genre de cas.

Notre typage statique n'est pas assez puissant pour valider statiquement certaines parties du langage. La figure 2.12 présente un tel cas où le flot d'exécution exact ne peut pas être connu de façon statique et dépend entièrement de l'exécution du programme, dans ce cas, du contenu du fichier lu.

2.3.2 Emplacement des tests dynamiques

L'approche proposée insère des tests dynamiques à deux endroits dans le code généré par le compilateur :

- lors de l'accès en lecture à un attribut dont le type statique est défini comme étant non-nullable ;
- à la fin de la construction de l'objet.

2.3.2.1 Test lors de l'accès en lecture à un attribut

Chaque attribut défini comme étant non-nullable est marqué comme étant non-initialisé lors de la création de l'objet. Lorsqu'un attribut est accédé en lecture, un test dynamique valide qu'il a bien été initialisé avant son utilisation. Si un attribut est accédé en lecture avant d'avoir été initialisé, une exception est levée. Puisque l'état de

non-initialisation ne peut pas être lu, il ne peut pas être affecté. Ainsi, le système de type va garantir que la valeur affectée dans un attribut non-nullable est non-nulle et initialisée.

Il est important de bien comprendre la différence entre une référence nulle et une référence non-initialisée. La première contient une valeur valide : la valeur *null*. Celle-ci peut être affectée, comparée et propagée à travers le programme. La seconde ne contient pas une valeur valide. Elle est dans un état de non-initialisation. Cet état ne doit pas être ni affecté, ni propagé à travers le programme.

2.3.2.2 Tests à la fin de la construction de l'objet

À la fin de la construction de l'objet, le compilateur ajoute un test systématiquement sur chacun des attributs définis comme étant non-nullable pour garantir que ceux-ci sont bien initialisés. Ces tests sont exactement les mêmes que ceux effectués lors de l'accès à un attribut.

L'emplacement de ces tests est important. On peut voir la construction d'un objet comme étant effectuée en plusieurs phases : (1) allocation de la mémoire et création des structures représentant l'objet, (2) initialisation des attributs par défaut de l'instance, puis (3) appel de l'initialiseur (les sous-programmes appelés *constructeurs* dans la plupart des langages à objets) et de tous les constructeurs des classes parentes. C'est à la suite de cette dernière étape que les tests présentés dans cette section sont effectués : une seule fois à la toute fin de l'initialisation de l'objet.

2.3.3 Garanties

Les emplacements et la composition des tests dynamiques garantissent que lorsqu'un objet est complètement construit, tous les tests lui étant associés ne lèveront jamais d'exceptions. Ceci est évident car : (1) à la fin de la construction, si les tests réussissent, il n'y a pas d'attributs non-initialisés et (2) par la suite, les tests d'accès protègent l'objet contre la déconstruction.

```
class MaClass
	var _attr: Int

	fun safe_get_attr: nullable Int do
		if isset _attr then return _attr
		return null
	end

	init do
		print (safe_get_attr == null) # affiche: true
		_attr = 1
		print (safe_get_attr == null) # affiche: false
	end
end
```

FIGURE 2.13: Exemple d'utilisation de l'opérateur *isset*, démonstration d'une fonction de test pour garantir le bon accès à un attribut non-nullable, même lors de l'initialisation de l'instance

2.3.4 Opérateur de test dynamique

Puisqu'un accès en lecture à un attribut non initialisé lève une exception, un nouvel opérateur de test dynamique, nommé `isset`, permettant de savoir si un attribut est initialisé est accessible aux programmeurs. La figure 2.13 présente un exemple d'utilisation de cet opérateur. Dans cet exemple, l'opérateur est utilisé pour s'assurer d'un accès sécuritaire à un attribut lors de l'initialisation de l'instance.

Toutefois, en pratique on s'est rendu compte que l'utilisation de cet opérateur est très anecdotique.

2.4 Garanties apportées

La synergie entre les composantes de typage statique et de tests dynamiques procure dynamiquement une politique d'*échec précoce* en ce qui concerne la propagation des références non-initialisées. Toute tentative de propagation d'une telle référence, que

ce soit par une affectation vers une autre référence, un déréférencement ou la fin du constructeur va causer une exception lors de l'exécution. C'est cette propriété que l'on appelle politique d'échec précoce.

On peut ainsi dire que la *zone de danger* dans un programme utilisant l'approche proposée, la zone où il y a une possibilité d'erreur avec les références nulles illégales, est circonscrite dynamiquement au code contenu dans les constructeurs et au code étant atteignable à partir de ceux-ci.

Cette approche est compatible avec les langages courants où l'on doit déjà faire attention à la manipulation et à l'échappement d'un objet en cours de construction puisque celui-ci ne satisfait pas nécessairement les invariants de classe.

2.4.1 Discussion sur les choix

Dans cette section, nous allons justifier certains des choix faits dans l'approche proposée. Plus particulièrement, nous allons présenter (1) la gestion des accès aux membres d'une référence nullable et (2) le choix d'utiliser une annotation *nullable* plutôt qu'une annotation *not nullable*.

Point 1 :

Les deux dernières restrictions présentées dans la section 2.2.2 sont sujettes à discussion. Ces deux restrictions sont :

- restriction d'accès aux attributs (Exemple 2.6) ;
- restriction d'appel (Exemple 2.7).

Laisser ces restrictions force le programmeur à utiliser une technique d'adaptation de type statique, telle que présentée dans la section 2.2.3, ce qui rend le code plus verbeux. Ceci est particulièrement problématique lorsque le programmeur veut faire des appels chaînés qui deviennent fastidieux à écrire. Il est important de noter que l'un ou l'autre de ces choix a exactement le même effet lors de l'exécution de l'application puisqu'un test dynamique sera effectué lors de l'appel ou de l'accès à l'attribut. Dans

l'implémentation présente de notre approche, nous avons choisi de laisser le programmeur appeler directement les méthodes ainsi qu'accéder directement aux attributs pour des buts de lisibilité du code.

Point 2 :

Le choix que nous avons effectué pour le système de typage statique, le fait d'annoter explicitement les références nullables plutôt que l'inverse, est basé sur une étude publiée (Chalin et James, 2007). Cette étude démontre notamment que la majorité des références dans les langages orientés objets sont de natures non-nullables, ainsi, puisque le cas *commun* est celui d'une référence non-nullable ; le cas spécial, le fait qu'une référence soit *nullable*, devient celui qui doit être annoté.

2.5 Conclusion

Ce chapitre a présenté notre solution à la problématique entourant la bonne gestion des références nulles. Les deux parties à notre solution sont : (1) un système de typage statique strict et (2) des tests dynamiques ajoutés par le compilateur.

Dans les deux sections subséquentes, le système de typage statique, basé sur un seul mot clé, *nullable*, et conservant l'expressivité complète du langage, ainsi que les tests dynamiques ont été approfondis.

La quatrième section, a rendu explicites les garanties statiques apportées par l'approche proposée, plus particulièrement la propriété d'échec précoce, prévenant toute propagation de valeur erronée dans le programme ainsi que la description de la zone de danger, la zone où il y a une possibilité d'erreur avec les références nulles illégales.

Finalement, nous avons justifié les choix que nous avons faits pour les deux composantes, notamment sur le choix de ne pas restreindre les appels aux attributs et aux méthodes dans l'implémentation de référence et sur celui d'utiliser le mot clé *nullable* plutôt qu'une approche avec mot clé *not nullable*.

CHAPITRE III

RELATIONS D'ORDRES ET TREILLIS

Ce chapitre présente un rappel sur les notions mathématiques de relations d'ordres et de treillis. Toutes les informations présentées proviennent de diverses références (Davey et Priestly, 1990; Stoll, 1979; Blyth, 2005).

Nous débutons ce chapitre en présentant la notion d'ordre partiel. Par la suite, nous présentons le concept de treillis. Celui-ci est lié de près avec les ordres partiels puisqu'il utilise une relation d'ordre dans sa construction. La dernière section présente les *méta-treillis* : des treillis qui sont bâtis à partir de la puissance cartésienne d'autres treillis.

3.1 Ordre partiel

Un ordre partiel (Blyth, 2005), noté $(V, \sqsubseteq)$, est défini par un ensemble V (potentiellement infini) et une relation binaire $\sqsubseteq$ sur V qui respecte, pour tous éléments x, y et z de V, les propriétés suivantes :

- réflexivité : $x \sqsubseteq x$;
- transitivité : $(x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$;
- antisymétrie : $(x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$.

Pour un ordre partiel $(V, \sqsubseteq)$, nous étendons la notation $\sqsubseteq$ et posons les opérateurs (Birkhoff, 1967) suivants pour tous les éléments x et y appartenant à V :

- si $x \sqsubseteq y$ alors $y \sqsupseteq x$;

- si $(x \sqsubseteq y \wedge x \neq y)$ alors $x \sqsubset y$;
- si $x \sqsubset y$ alors $y \sqsupset x$.

Un ordre total (Birkhoff, 1967) est un ordre partiel $(V, \sqsubseteq)$ où tout couple d'élément est en relation dans un sens ou dans l'autre, formellement : $\forall x, y \in V;\ x \sqsubseteq y \vee y \sqsubseteq x$. Par exemple, $(\mathbb{N}, \geq)$ est un ordre total [1].

3.1.1 Représentation graphique d'un ordre partiel

Les diagrammes de Hasse sont généralement utilisés pour représenter graphiquement un ordre fini (Freese, 2004). Un tel diagramme, pour un ordre fini $(V, \sqsubseteq)$, respecte les règles suivantes :

- soient deux éléments x et y, tous deux éléments de V, tels que $x \sqsubseteq y$, y est plus haut *visuellement* que x dans le graphique ;
- on ne représente que la réduction transitive de la relation. Ainsi, pour trois éléments x, y et z, éléments de V, si $x \sqsubseteq y \sqsubseteq z$, on retrouve un segment entre x et y ainsi qu'un second entre y et z. Il n'y a pas de segment entre x et z puisque la relation est déjà représentée par les deux autres segments ;
- si deux éléments sont en relation directe, donc qu'ils sont en relations mais qu'il n'existe pas d'éléments entre ces deux éléments, ils sont reliés par un segment. Ce segment ne se termine pas par une flèche puisque l'on sait que la relation va du bas vers le haut ;
- on ne représente pas les boucles (réduction réflexive) ;
- pour rendre le graphique plus lisible, on essaie autant que possible de ne pas croiser les segments.

La figure 3.1 présente un diagramme de Hasse où est représenté l'ordre partiel $(\mathcal{P}(\{x, y, z\}), \subseteq)$.

1. Puisque ce document n'utilise pas d'ordres stricts ni de relations d'ordres strictes qui imposent l'anti-reflexivité comme propriété nécessaire, nous avons choisi de ne pas discuter de ceux-ci dans cette section.

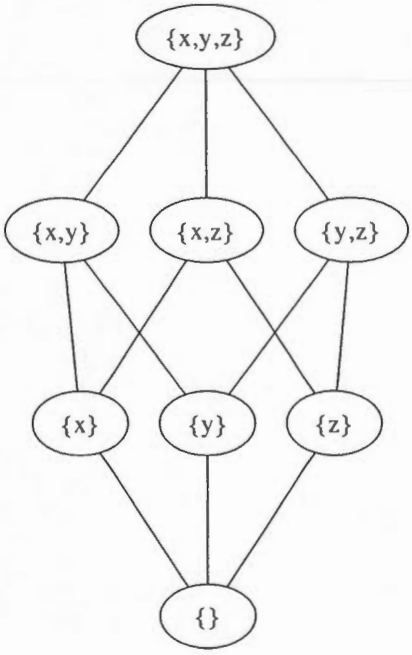

FIGURE 3.1: Diagramme de Hasse représentant l'ordre partiel $(\mathcal{P}(\{x, y, z\}), \subseteq)$

3.1.2 Majorant et minorant

Soit un sous-ensemble fini non-vide F d'un ordre partiel $(V, \sqsubseteq)$. On dit de $x \in V$ qu'il est (Gratzer, 1978; Kaufmann, A. and Boulaye, G.G., 1978) :

- un majorant de F si $\forall y \in F \quad y \sqsubseteq x$;
- un minorant de F si $\forall y \in F \quad x \sqsubseteq y$.

Nous allons définir la fonction majorants(x) comme étant la fonction qui permet de retourner l'ensemble des majorants d'un élément x, formellement, majorants(x) = $\{y | x \sqsubseteq y\}$. De façon analogue, nous allons aussi définir la fonction minorants(x) comme étant la fonction qui permet de retourner l'ensemble des minorants d'un élément x, formellement, minorants(x) = $\{y | y \sqsubseteq x\}$.

3.1.3 Monotonie d'une fonction

Cette section aborde le thème de la monotonie d'une fonction (Fuchs, 1963). Nous allons d'abord parler de monotonie au sens large, puis distinguer celle-ci de la monotonie stricte.

Soient deux ordres partiels $(X, \sqsubseteq^X)$ et $(Y, \sqsubseteq^Y)$, une fonction $f : X \rightarrow Y$ est monotone au sens large si

$$\forall x_1, x_2 \in X \quad x_1 \sqsubseteq^X x_2 \Rightarrow f(x_1) \sqsubseteq^Y f(x_2)$$

La monotonie stricte se définit plutôt par la relation :

$$\forall x_1, x_2 \in X \quad x_1 \sqsubset^X x_2 \Rightarrow f(x_1) \sqsubset^Y f(x_2)$$

Remarque : dans ce document, le terme *monotonie* est utilisé pour indiquer la monotonie au sens large.

3.2 Treillis

Un ordre partiel $(V, \sqsubseteq)$ est un treillis (Birkhoff, 1967) s'il respecte les deux règles supplémentaires :

1. chaque couple d'éléments possède un plus petit majorant commun ;
2. chaque couple d'éléments possède un plus grand minorant commun.

Formellement, soient x et y des éléments d'un ordre partiel $(V, \sqsubseteq)$:

1. $\exists! z \in V, \forall a \in \text{majorants}(x) \cap \text{majorants}(y) | z \sqsubseteq a$, que nous allons noter $x \sqcap y$. Il est important de noter que bien que z soit un élément de V, il est plus précisément un élément de l'intersection entre les majorants de x et les majorants de y ;
2. $\exists! z \in V, \forall a \in \text{minorants}(x) \cap \text{minorants}(y) | z \sqsupseteq a$, que nous allons noter $x \sqcup y$. Il est important de noter que bien que z soit un élément de V, il est plus précisément un élément de l'intersection entre les minorant de x et les minorants de y.

3.2.1 Treillis bornés

Un treillis est dit borné lorsqu'il possède une valeur minimale et une valeur maximale et que ces deux valeurs font partie de ce treillis. Formellement, pour un treillis borné $(V, \sqsubseteq)$, il existe un minimum, noté $\bot$, ainsi qu'un maximum, noté $\top$, tels que $\bot \in V$ et $\top \in V$, qui respectent la règle suivante :

$$\forall e \in V \quad \bot \sqsubseteq e \sqsubseteq \top$$

3.2.2 Treillis complets

Formellement, un treillis $(V, \sqsubseteq)$ est dit complet si, pour tout sous-ensemble fini non-vide E formé à partir de V :

$$\forall E \subseteq V \quad \exists m \in V \quad \forall e \in E \quad m \sqsubseteq e$$

$$\forall E \subseteq V \quad \exists n \in V \quad \forall e \in E \quad e \sqsubseteq n$$

Dit autrement, un treillis est dit complet si, pour tout sous-ensemble de ce treillis, il existe un plus petit majorant commun ainsi qu'un plus grand minorant commun à tous les éléments de ce sous-ensemble. Il est important de voir que bien que n et m doivent être éléments de V, ils ne font pas nécessairement partie du sous-ensemble E.

Pour pouvoir aisément présenter les équations dans ce document, nous introduisons une nouvelle notation pour le plus petit majorant commun des éléments d'un ensembles ainsi que pour le plus grand minorant commun des éléments d'un ensembles. Soit un sous-ensemble fini non-vide E d'un treillis $(V, \sqsubseteq)$:

1. la notation $\bigsqcap E$ est utilisée pour représenter le plus petit majorant commun aux éléments de E ;
2. la notation $\bigsqcup E$ est utilisée pour représenter le plus grand minorant commun aux éléments de E.

3.2.3 Treillis finis

Un treillis fini est un treillis où est présent un nombre fini d'éléments. Un treillis fini est automatiquement complet (Blyth, 2005).

Théorème 3.2.1. *Le plus petit majorant commun des plus petits majorants communs de deux ensembles non-vides est le même élément que le plus petit majorant commun de l'union de ces deux ensembles. Formellement, pour deux ensembles non vides A et B dont les éléments font partie d'un treillis $(V, \sqsubseteq)$, $\bigsqcap(A \cup B) = (\bigsqcap A) \sqcap (\bigsqcap B)$.*

Démonstration. Soient A et B deux sous-ensembles non-vides d'éléments du treillis fini $(V, \sqsubseteq)$ et C tel que $C = A \cup B$. Montrons que $\bigsqcap C = \bigsqcap A \sqcap \bigsqcap B$.

Par définition, nous savons que $\bigsqcap A$ est le plus petit élément commun à tous les éléments de A. Nous savons la même chose pour $\bigsqcap B$, par rapport aux éléments de B. Nous savons aussi que $\bigsqcap C$ est un parent (pas nécessairement le plus petit) commun à tous les éléments de A et de B. Nous savons donc que $\bigsqcap B \sqsubseteq \bigsqcap C$ et que $\bigsqcap A \sqsubseteq \bigsqcap C$.

Puisque $\bigsqcap C$ est un parent de $\bigsqcap A$ et de $\bigsqcap B$, nous savons qu'il ne peut pas être plus petit que le plus petit parent commun de $\bigsqcap A$ et $\bigsqcap B$, formellement, $\bigsqcap A \sqcap \bigsqcap B \sqsubseteq \bigsqcap C$.

De plus, par définition, nous savons que $\bigsqcap C$ est le plus petit parent commun à tous les éléments de A et B. Ainsi, nous savons qu'il n'existe pas de parent commun à $\bigsqcap A$ et $\bigsqcap B$ plus petit que $\bigsqcap C$, formellement $\bigsqcap C \sqsubseteq \bigsqcap A \sqcap \bigsqcap B$.

Puisque nous avons montré que $\bigsqcap A \sqcap \bigsqcap B \sqsubseteq \bigsqcap C$ et $\bigsqcap C \sqsubseteq \bigsqcap A \sqcap \bigsqcap B$, nous savons que $\bigsqcap C = \bigsqcap A \sqcap \bigsqcap B$. □

3.3 Méta-treillis

Un *méta-treillis* est un treillis bâti par la puissance cartésienne (Birkhoff, 1967) d'un autre treillis[2]. Le treillis utilisé comme base pour le méta-treillis est appelé *base*. Les éléments d'un méta-treillis sont des tuples dont chacun des membres est élément du treillis de base. Un méta-treillis où chacun des tuples contient n éléments est dit d'ordre n. Il est important de noter que n doit être un entier positif non nul.

Formellement, un méta-treillis est noté $(V, \sqsubseteq^V, n)$ où $(V, \sqsubseteq^V)$ est un treillis (appelé base du méta-treillis) et n est un entier positif non nul (appelé ordre du méta-treillis). Chacun des membre d'un méta-treillis $(V, \sqsubseteq^V, n)$ est de la forme$(v_1, v_2, ..., v_n)$ où $v_i \in V$.

La relation d'ordre d'un méta-treillis est définie par la notion d'*ordre produit* (Neggers et Kim, 1998). Ainsi, pour un méta-treillis $W = (V, \sqsubseteq^V, n)$, la relation d'ordre est définie comme étant $a \sqsubseteq^W b \Leftrightarrow a_1 \sqsubseteq^V b_1 \wedge a_2 \sqsubseteq^V b_2 \wedge ...a_n \sqsubseteq^V b_n$, où a et b sont des éléments du méta-treillis W et a_i est l'élément à la position i du tuple a.

Par exemple, dans le méta-treillis $(\mathbb{N}, \leq, 2)$, la relation d'ordre est définie comme étant $\forall (a,b),(c,d) \in \mathbb{N} \times \mathbb{N} \quad (a,b) \leq (c,d) \Leftrightarrow a \leq c \wedge b \leq d$. Ainsi, on a $(5,8) \leq (10,9)$, mais il n'existe pas de relation entre $(5,9)$ et $(10,8)$, ni dans un sens ni dans l'autre.

---

2. Un méta-treillis est un treillis puisque le produit cartésien de deux treillis est un treillis (Singh, 2006).

Théorème 3.3.1. *Soit le treillis fini $(V, \sqsubseteq)$, alors un méta-treillis bâti à partir de celui-ci est fini, et ce peu importe son degré. Formellement, $(V, \sqsubseteq, n)$ est fini peu importe son degré.*

Démonstration. Soit un treillis fini $(V, \sqsubseteq)$ contenant m éléments et un méta-treillis W créé à partir de celui-ci, formellement $W = (V, \sqsubseteq, n)$. Le nombre de possibilités pour un seul élément du tuple W_i est m, le nombre d'éléments de V. Ainsi, nous savons que le nombre de possibilités pour l'ensemble des tuples/éléments contenus dans W est $\underbrace{m \times m \times ...m}_{n}$, donc m^n éléments. □

Exemple 3.3.2. *Soit V l'ensemble $\{1, 2, 3\}$, alors le méta-treillis $(V, \sqsubseteq, 2)$ sera composé des éléments : $(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)$, donc 3^2 (9) éléments.*

3.4 Conclusion

Dans ce chapitre, nous avons présenté une base théorique concernant les relations d'ordre ainsi que les treillis. Les informations présentées dans ce chapitre n'ont pas pour but d'être exhaustives, mais bien de donner une base suffisante à la compréhension des chapitres suivants.

Pour le reste de ce document, nous utiliserons généralement $(V, \sqsubseteq)$ pour représenter un treillis fini quelconque et W pour représenter le méta-treillis $(V, \sqsubseteq, n)$. Le degré du méta-treillis W sera toujours indiqué.

CHAPITRE IV

NOTIONS IMPORTANTES

Ce chapitre présente plusieurs notions importantes pour la compréhension du reste de ce document. En premier lieu, il présente le langage utilisé pour implémenter l'approche présentée dans ce document : Nit. La seconde section présente la notion de graphe de flux de contrôle et la dernière section la notion de graphe d'appel. Ces deux types de graphes seront utilisés par les analyses statiques présentées dans le chapitre suivant.

4.1 Le langage Nit

Nit (site web : http://nitlanguage.org/) est le langage utilisé pour l'implémentation de l'approche présentée dans ce document. Ce langage est une évolution du langage PRM développé par Jean Privat (Privat, 2006).

Nit est un langage orienté objet qui présente, entre-autre, les caractéristiques suivantes :

1. typage statique avec inférence de type légère ;
2. types nullables ;
3. généricité ;
4. héritage multiple ;
5. types virtuels ;
6. classes ouvertes ;

7. et plusieurs autres.

Le langage Nit est encore en développement et il est présentement utilisé à des fins de recherche. Le seul *vrai* logiciel écrit en Nit est le compilateur Nit lui même. Ce compilateur est tout de même un logiciel complexe de plus de 110.000 lignes de code, ayant évolué grandement et utilisant toutes les techniques possibles dans le langage.

Au départ du langage, le compilateur Nit n'était pas écrit pour supporter les types nullables. La migration de celui-ci pour supporter les types nullables (Gélinas, Privat et Gagnon, 2009) a été effectuée en deux étapes :

1. implémentation de la partie typage statique dans le compilateur et modifications rapides dans le code pour tout faire fonctionner (principalement en ajoutant des annotations nullable dans le code du compilateur et dans la librairie standard) ;
2. implémentation de la partie de validation dynamique dans le compilateur et modifications dans le code pour tout faire fonctionner.

Le but de cette première étape était de tester notre approche. Les résultats étaient encourageants : ajout de 7250 lignes de code au compilateur (+13%), moins de 10% des types statiques explicites nécessitent une annotation de types nullables et environ 30% des attributs sont définis comme étant nullables. Dans cette version, une grande partie des types et des attributs nullables sont localisés dans le *parser* et le *lexer*, générés par SableCC[1]. Plus précisément, on retrouve dans cette région du compilateur 80% d'attributs définis comme étant nullable.

Le compilateur Nit utilise une représentation intermédiaire du code pour effectuer ses optimisations. Cette représentation est plus simple que le code écrit par le programmeur et est donc plus facile à analyser. Les catégories d'instructions suivantes sont celles qui sont accessibles dans cette représentation : les affectations, les séquences d'instructions, les branchements conditionnels, les boucles, les break ainsi que les appels.

---

1. Site web : http ://sablecc.org/

4.2 Graphes de flux de contrôles

Pour gérer l'abstraction du flux d'exécution, les analyses statiques utilisent souvent des graphes de flux de contrôle. Un graphe de flux de contrôle est un graphe orienté où chacun des noeuds représente une instruction dans le programme et chacun des arcs représente un chemin d'exécution possible. Un tel graphe est utilisé lors d'analyses statiques pour connaître les divers chemins d'exécution possible pour se rendre à une instruction donnée.

Les algorithmes de construction de graphe de flux de contrôle sont dépendants du langage utilisé en entrée à l'analyse. Pour cette raison, cette section ne présente qu'un aperçu de ce qu'est un graphe de flux de contrôle et non la façon de le construire. Plus précisément, cette section aborde la notion même de ce qu'est un graphe de flux de contrôle et présente différentes structures typiques trouvées dans un tel graphe.

Le reste de cette section présente quelques exemples de structures typiques que l'on retrouve dans un graphe de flux de contrôle.

4.2.1 Séquence d'instructions

Les séquences d'instructions sont les constructions les plus simples à reconnaître dans un graphe de flux de contrôle. Une séquence d'instructions est représentée en créant un noeud par instruction, puis en ajoutant des arcs entre ces noeuds. Il est important de noter que, puisqu'une instruction simple ne modifie pas le flux d'exécution du programme, un seul arc sortant sera présent sur chacun des noeuds. Ainsi, dans l'exemple de la figure 4.1, il existe un arc entre la première instruction et la seconde, un autre arc entre la seconde et la troisième, etc.

Certaines approches regroupent les séquences d'instruction sans branchement conditionnels comme étant un seul noeud, appelé *bloc de base* (Allen, 1970), dans le graphe de flux de contrôle.

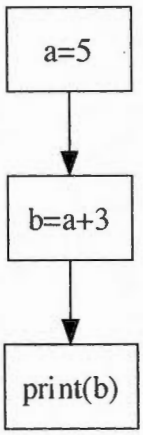

FIGURE 4.1: Exemple de graphe de flux de contrôle avec une séquence d'instruction

4.2.2 Branchement conditionnel

Puisque les arcs de notre graphe représentent un chemin d'exécution possible, un noeud qui est un branchement conditionnel aura toujours exactement deux arcs sortant plutôt qu'un seul. L'information sur la condition pour les branches (celle qui est la partie *then* et celle qui est la partie *else*) n'est pas toujours indiquée dans un graphe de contrôle de flux de contrôle. L'exemple de la figure 4.2 présente un graphe de flux avec un branchement conditionnel contenant à la fois des instructions dans la section `then` que dans la section `else`, tandis que la figure 4.3 présente un graphe de flux avec un branchement conditionnel qui n'a pas les deux sections. On voit dans cette dernière figure qu'une branche absente est représentée par une arrête vers la fin du code conditionnel.

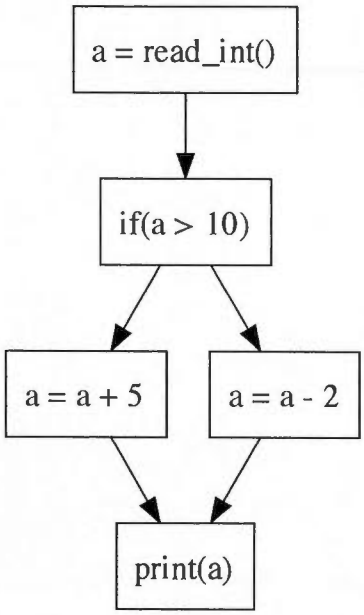

FIGURE 4.2: Exemple de graphe de flux de contrôle avec un branchement conditionnel

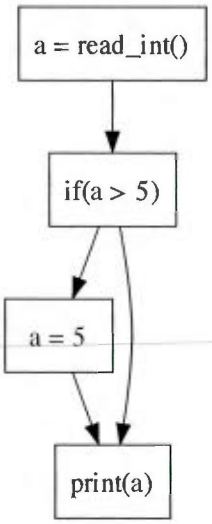

FIGURE 4.3: Exemple de graphe de flux de contrôle avec un branchement conditionnel avec du code dans une seule section

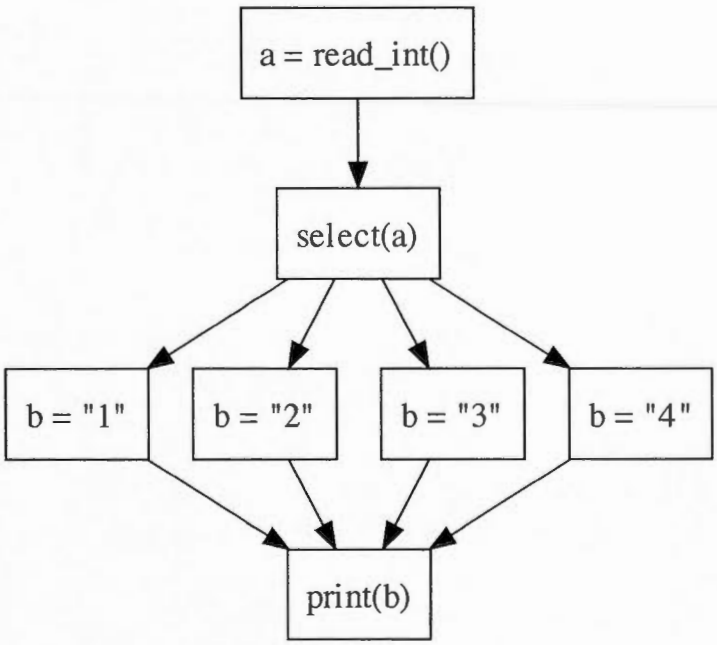

FIGURE 4.4: Exemple de graphe de flux de contrôle avec une instruction de sélection avec 4 choix possibles

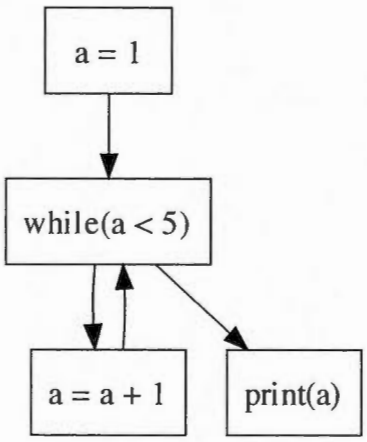

FIGURE 4.5: Exemple de graphe de flux de contrôle avec une boucle de type `while`

4.2.3 Sélection

Une sélection (*switch*) n'est en fait qu'un branchement conditionnel avec plus de deux chemins d'exécution. On voit dans la figure 4.4 un exemple de graphe de flux de contrôle avec une instruction de sélection avec 4 choix possibles pour la sélection.

4.2.4 Boucle

Cette section présente deux types de boucles de base : (1) les boucles `while` et (2) les boucles `do ... while`.

Le premier type de boucle, les boucles de type `while`, vont tester la condition de terminaison de la boucle avant d'exécuter le corps de la boucle. Ceci résulte en un graphe de flux de contrôle ressemblant à celui présenté dans la figure 4.5.

Le second type de boucle, les boucles de type `do ... while`, vont tester la condi-

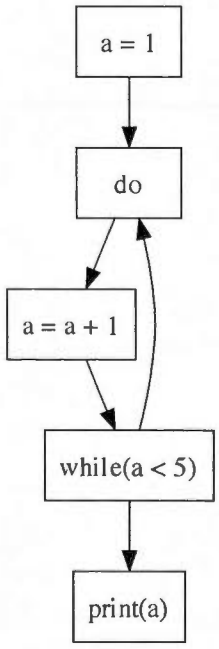

FIGURE 4.6: Exemple de graphe de flux de contrôle avec une boucle de type `do ... while`

tion de terminaison après l'exécution du corps de la boucle. Ainsi, ceci résulte en un graphe de flux de contrôle ressemblant à celui présenté dans la figure 4.6.

Les instructions de type `break`[2] vont simplement ajouter un arc entre l'instruction `break` et l'instruction suivant la boucle.

Les instructions de type `continue`[3] vont simplement ajouter un arc entre l'instruction `continue` et l'instruction de test de terminaison de la boucle.

4.3 Graphes d'appels

Cette section présente le concept de graphe d'appel. Un graphe d'appel est un graphe orienté où chacun des sommets représente une méthode du programme analysé et chacun des arcs un ou plusieurs appels de méthode. Les informations contenues dans un tel graphe sont utiles lors de l'analyse de programmes complexes pour connaître quelle méthode sera appelée dynamiquement à un site d'appel et d'où une méthode peut être appelée.

Plus précisément, cette section présente les propriétés d'un graphe d'appel ainsi que certains exemples simples de construction de graphe d'appel pour un langage procédural puis pour un langage à objets.

4.3.1 Propriétés

Les propriétés de graphes d'appel peuvent être très différentes d'une approche à l'autre. Cette section présente quelques caractéristiques généralement présentes dans les graphes d'appel. Les figures 4.7 et 4.8 présentent le code ainsi que le graphe corres-

2. Un instruction de type `break` est une instruction qui, dans une boucle, permet de sortir de celle-ci.

3. Un instruction de type `continue` est une instruction qui, dans une boucle, permet de passer directement à la prochaine itération de celle-ci.

```
void bar(){}
void ham(){}
void foo(){
        if(0 == 1){
                bar();
        }
}
void spam(){
        spam();
}
int main(){
        foo();
        bar();
        spam();
        foo();
        return 0;
}
```

FIGURE 4.7: Exemple de code C, utilisé pour le graphe d'appel de la figure 4.8

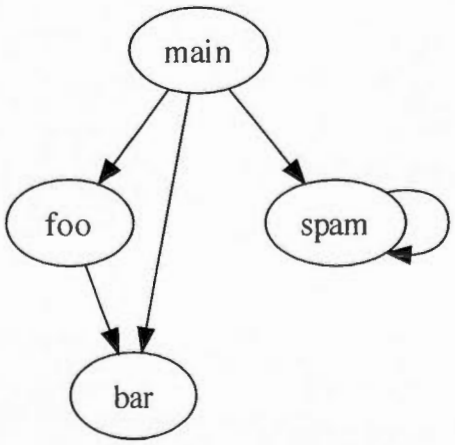

FIGURE 4.8: Graphe d'appel généré à partir du code de la figure 4.7 en utilisant la méthode *main* comme point d'entrée

pondant pour un exemple simple de graphe d'appel. Plusieurs propriétés sont visibles à partir de cet exemple :

- il n'existe qu'un seul arc entre deux sommets même si plusieurs appels vers la même méthode existent dans le code : bien qu'il existe deux appels vers la méthode `foo` à partir de la méthode `main`, il n'existe qu'un seul arc entre le sommet `main` et le sommet `foo` dans le graphe (en d'autres mots, c'est bien un graphe qui est utilisé, et non un multi-graphe) ;
- une analyse simple ne tient pas compte du fait qu'un appel soit possible lors de l'exécution ou non : bien qu'il est visible que l'appel à `bar` n'aura jamais lieu dans la méthode `foo`, il existe tout de même un arc entre les sommets `foo` et `bar` dans le graphe. Ceci est dû au fait que la construction du graphe d'appel ne tient pas compte des structures de contrôles ;
- les appels récursifs forment des cycles : dans un graphe d'appel, tout appel récursif va créer un cycle, que ce soit un appel récursif simple (tel que dans le cas de la méthode `spam` de l'exemple) ou un cas beaucoup plus complexe. Ainsi, détecter des appels récursifs devient un problème de détection de cycles dans un graphe ;
- les méthodes qui ne sont pas accessibles à partir du départ du programme ne sont pas dans le graphe : le code de l'exemple contient la déclaration d'une méthode nommée `ham`. Cette méthode n'est jamais appelée à partir de la méthode `main` ni à partir d'une méthode appelée directement ou indirectement à partir de celle-ci. Pour cette raison, il n'existe pas de sommet nommé `ham` dans le graphe. Cette particularité est très utile pour détecter et supprimer le *code mort* lors de la compilation d'une application.

4.3.2 Construction dans un langage procédural

La construction d'un graphe d'appel pour un langage qui ne supporte pas le polymorphisme est assez simple.

Une façon de construire un graphe d'appel est comme suit : en commençant au

point d'entrée du programme, l'algorithme analyse chacune des instructions et lorsqu'une instruction d'appel de méthode est trouvée, il ajoute au graphe d'appel un lien entre la méthode analysée présentement et celle qui vient d'être trouvée. Lorsqu'un sommet est ajouté, le corps de la méthode associée à celui-ci doit être analysé pour trouver d'autres appels. Avant d'ajouter une méthode à la liste de méthodes à analyser, l'algorithme doit s'assurer de ne pas avoir déjà visité cette méthode. Si cette validation n'avait pas lieu, l'analyse d'un appel récursif se transformerait en une boucle infinie dans l'algorithme.

Cet algorithme simple ne fonctionne qu'avec les langages où les cibles de chacun des appels sont connues lors de l'analyse. Ainsi, l'envoi de message, la programmation dynamique et la réflexivité sont des caractéristiques de langages qui ne sont pas supportées par cet algorithme.

4.3.3 Construction dans un langage à objets

Dans un langage avec appel de méthode, une technique telle que Rapid Type Analysis (RTA) (Bacon, 1997) peut être utilisée. Cette dernière technique utilise les informations suivantes pour détecter quelles méthodes sont les cibles potentielles à un site d'appel donné :

1. la hiérarchie de classe (autant les classes que les méthodes) ;
2. la liste des classes instanciées dans les méthodes atteignables à partir du point d'entrée du programme ;
3. le type statique des objets.

La figure 4.11 présente les méthodes atteignables pour les appels de la figure 4.9, tandis que la figure 4.10 présente le graphe d'appel pour les mêmes appels. Il est important de noter qu'il n'existe pas d'instantiation de la classe B ou d'une de ses sous-classe dans le code donné, ainsi, aucune méthode de cette classe n'est accessible. Plus précisément, ces figures présentent :

– un appel à `foo` dans `bar`, où le receveur est statiquement une instance de `A`,

```
class A
	fun foo do end
end

class B super A
	redef fun foo do end
end

class C super B
	redef fun foo do end
end

fun bar (a: A) do
	a.foo
end

fun baz (b: B) do
	b.foo
end

fun spam (c: C) do
	c.foo
end

var a: A = new A
var c: C = new C
bar(a)
baz(c)
spam(c)
```

FIGURE 4.9: Exemple de code Nit, utilisé pour le graphe d'appel de la figure 4.10

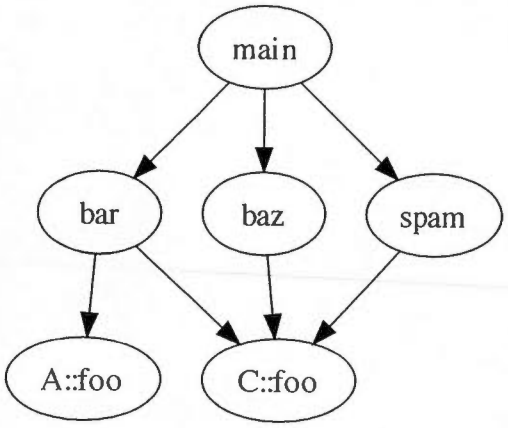

FIGURE 4.10: Graphe d'appel généré à partir du code de la figure 4.9

Appel	A::foo	B::foo	C::foo
Dans `bar` : `a.foo`	oui	non	oui
Dans `baz` : `b.foo`	non	non	oui
Dans `spam` : `c.foo`	non	non	oui

FIGURE 4.11: Méthodes atteignables à partir des divers appels du code de la figure 4.9

qui peut dynamiquement appeler `A::foo` ou `C::foo` selon le type dynamique du receveur ;

- un appel à `foo` dans `baz`, où le receveur est statiquement une instance de `B`, qui va dynamiquement appeler `C::foo` ;
- un appel à `foo` dans `spam`, où le receveur est statiquement une instance de `C`, qui va dynamiquement appeler `C::foo`.

Il existe plusieurs autres algorithmes de génération de graphe d'appel, mais ceux-ci sont hors de la portée de ce document.

4.4 Conclusion

Ce chapitre a présenté plusieurs notions importantes pour la compréhension du reste de ce document. En premier lieu, le langage Nit fut présenté. Celui-ci est le langage utilisé pour l'implémentation de l'approche présentée tout au long de ce document. Par la suite, deux types de graphes furent présentés : les graphes de flux de contrôles ainsi que les graphes d'appels. Ces deux types de graphes sont cruciaux aux analyses statiques.

CHAPITRE V

ANALYSE STATIQUE INTRAPROCÉDURALE

Les analyses statiques intraprocédurales sont une catégorie d'analyse qui s'effectuent lors de la compilation du programme (d'où le qualificatif *statique*) et qui analysent le corps des procédures, méthodes ou fonctions sans tenir compte des liens entre celles-ci.

Ce chapitre présente la base des analyses statiques intraprocédurales. Puisque le domaine des analyses statiques est très vaste, ce chapitre se concentre sur les analyses qui vont débuter au début du programme pour se propager vers la fin de celui-ci (*forward analysis*), qui utilisent un treillis fini pour représenter leurs informations, qui débutent avec la valeur $\perp$ pour *monter* dans le treillis associé à l'analyse et qui utilisent des fonctions monotones.

Les informations présentées dans ce chapitre proviennent de divers livres (Alfred, Sethi et Jeffrey, 1986; Appel et Ginsburg, 1998; Muchnick, 1997; Wilhelm, Maurer et Wilson, 1995). Plusieurs preuves sont présentées ici sans être présentées dans les ouvrages traditionels. Elles n'apportent pour autant rien de nouveau ; l'auteur jugeait qu'elles étaient nécessaires aux bonnes compréhension et présentation de ce document.

Plus précisément, ce chapitre présente tout d'abord les abstractions de l'information qui sont nécessaires pour mener à bien les analyses statiques puis la représentation qui sera utilisée dans ce document pour présenter ces informations. Par la suite, les sous-fonctions d'instruction, des fonctions utilisées à l'interne des algorithmes d'analyses statiques seront présentées. Suivant ces trois sections, un exemple de définition

d'analyse statique est présenté, suivi par la présentation de deux algorithmes d'analyse statique. Finalement, un exemple complet d'analyse statique clôture ce chapitre.

5.1 Abstraction de l'information

Les analyses présentées dans ce document sont statiques, donc effectuées lors de la compilation d'un programme. Ainsi, elles n'ont pas accès aux informations de l'exécution du programme. Pour cette raison, certains aspects de l'analyse sont gérés de façon abstraite (Muchnick, 1997). Plus précisément, les analyses statiques doivent abstraire la gestion de la mémoire, le flux d'exécution du programme ainsi que l'effet des instructions sur la mémoire. Ces abstractions seront présentées dans les trois premières parties de cette section.

Pour s'assurer d'avoir des résultats valides et aussi précis que possible, les analyses présentées dans ce document utilisent le concept de point fixe. Ce concept sera présenté en quatrième partie de cette section.

La notion de point fixe implique que le code source du programme ou du bout de code analysé soit analysé à plusieurs reprises. Le terme *itération* sera utilisé pour désigner une itération de l'algorithme d'analyse sur tout le code. Ce terme sera approfondi dans la dernière partie de cette section.

5.1.1 Abstraction de la mémoire

Pour gérer l'abstraction de la mémoire, les analyses présentées ici utilisent un treillis fini. Les données contenues dans le treillis sont dépendantes de l'analyse effectuée. Ainsi, le développeur d'une analyse doit définir le treillis à utiliser dans le cadre de celle-ci.

Puisqu'une valeur du treillis représente un état de la mémoire, deux valeurs de treillis sont associées à chaque instruction dans le code du programme : la première pour

l'état de la mémoire avant l'instruction et une seconde pour l'état de la mémoire après exécution de l'instruction.

5.1.2 Abstraction du flux d'exécution

Les analyses statiques utilisent souvent un graphe de flux de contrôle (voir section 4.2) pour gérer l'abstraction du flux d'exécution. En conjonction avec celui-ci, elles doivent utiliser une fonction de fusion (Muchnick, 1997) pour fusionner les informations provenant de divers chemins d'exécution.

Dans le contexte d'une analyse statique dont le treillis de valeur est $(V, \sqsubseteq)$, une fonction de fusion est une fonction $m : (V, \sqsubseteq)^+ \rightarrow (V, \sqsubseteq)$ qui reçoit en entrée un ensemble contenant des valeurs qui sont membres du treillis de l'analyse et qui va retourner une seule valeur de ce même treillis. Ces fonctions sont utilisées lorsque plusieurs entrées sont possibles à une instruction donnée. Ceci arrive, par exemple, à la sortie d'un branchement conditionnel où les informations des branches *then* et *else* doivent être combinées.

Puisque la fonction de fusion est liée de près avec le treillis de l'analyse, chaque analyse doit fournir sa propre fonction de fusion.

5.1.3 Abstraction de l'effet des instructions

La dernière abstraction que l'analyse doit gérer est l'abstraction de l'effet des instructions. On peut voir une instruction comme étant une fonction qui a un effet sur la mémoire de l'analyse. On appelle cette fonction : fonction de transition (Muchnick, 1997). Une fonction de transition est une fonction $f : (V, \sqsubseteq) \rightarrow (V, \sqsubseteq)$ que l'on applique sur une valeur du treillis représentant l'état de la mémoire à l'entrée de l'instruction pour connaître la valeur associée à la sortie de l'instruction. Chaque type d'instruction qui doit avoir un effet sur l'abstraction pour l'analyse en cours doit avoir une fonction de transition. Une instruction qui n'a pas d'effet utilise la fonction identité $I(x) = x$.

Une fonction de transition doit être monotone pour être utile à l'analyse statique (voir section 5.1.4).

5.1.4 Point fixe

Un point fixe pour une fonction $f : E \rightarrow E$ est un élément x, appartenant à E, tel que $f(x) = x$. Dans une analyse statique, un point fixe (Muchnick, 1997) est un moment où l'analyse a trouvé des informations jugées valides pour l'abstraction utilisée.

Le théorème de Knaster-Tarski (Tarski, 1955) nous montre que l'application successive d'une *fonction monotone* $f : V \rightarrow V$, où V est un treillis fini, en débutant la séquence d'application par la valeur $\perp$, présente plusieurs propriétés dont :

- un point fixe sera atteint en un temps fini ;
- l'ensemble des points fixes pouvant être atteint forme un sous-treillis du treillis V ;
- le point fixe atteint sera le plus petit parmi l'ensemble de points fixes atteignables, tel que défini par la relation d'ordre du treillis V.

Ainsi, pour pouvoir utiliser le théorème de point fixe de Knaster-Tarski, chacune des fonctions de transitions doit être monotone.

5.1.5 Itération

Puisqu'il est possible que l'algorithme analyse plusieurs fois l'ensemble du programme, le terme *itération* est utilisé pour représenter une itération de l'analyse sur l'ensemble du code source.

De façon théorique, chaque itération est composée de deux valeurs du treillis de l'analyse par instruction dans le programme : une valeur pour l'état de la mémoire du système avant l'évaluation de l'instruction et une seconde pour l'état de la mémoire du système après l'instruction.

En pratique, une seule de ces valeurs est stockée. Plus précisément, la valeur

en sortie d'une instruction à une itération donnée est stockée. La valeur en entrée est calculée à partir des valeurs de sortie des instructions de l'itération précédente.

Formellement, pour une analyse dont le treillis est $(V, \sqsubseteq)$ et pour un programme analysé contenant n instructions, une itération est une valeur dans le méta-treillis $(V, \sqsubseteq, n)$, donc un tuple où chacun des éléments est la valeur de treillis associé à la sortie d'une instruction.

Lors du départ de l'analyse, toutes les valeurs sont initialisées à $\bot$, la borne minimale du treillis de l'analyse. Ainsi, la valeur de la première itération sera $I = \underbrace{(\bot, \bot, ..., \bot)}_{n}$, n étant le nombre d'instructions dans le programme analysé.

5.2 Représentation des informations

Pour représenter graphiquement une séquence d'itérations, ce document utilise un tableau où chacune des colonnes est une itération et chacune des lignes est une instruction. Il est important de noter que l'implémentation d'une analyse statique ne devrait pas utiliser ce genre de structure de données puisqu'une telle structure de données nécessite beaucoup de mémoire, alors que les informations qui y sont stockées ne sont pas nécessaires tout au long de l'analyse. L'optimisation de la structure de données n'est pas le but de ce document.

La figure suivante présente un tel tableau représentant les itérations 1 à 4 de l'analyse d'un programme à 5 instructions :

	It. 1	It. 2	It. 3	It. 4
Instr. 1	v_1^1	v_2^1	v_3^1	v_4^1
Instr. 2	v_1^2	v_2^2	v_3^2	v_4^2
Instr. 3	v_1^3	v_2^3	v_3^3	v_4^3
Instr. 4	v_1^4	v_2^4	v_3^4	v_4^4
Instr. 5	v_1^5	v_2^5	v_3^5	v_4^5

où chaque valeur v_j^i est la valeur de sortie de la fonction de transition associée à l'instruction i pour l'itération j.

5.3 Sous-fonctions d'instruction

Une sous-fonction d'instruction est une fonction qui permet de calculer, à partir d'une itération complète, la valeur de sortie d'une instruction précise pour l'itération suivante.

Pour ce faire, la fonction est divisée en deux parties : (1) une fonction de sélection et (2) la fonction de transition de l'instruction.

Cette section présente les fonctions de sélection ainsi que la construction des sous-fonctions d'instruction.

5.3.1 Fonctions de sélection

Dans une analyse statique, une fonction de sélection est utilisée pour sélectionner les valeurs provenant des divers chemins d'exécution possibles pour une instruction donnée.

Les fonctions de sélection sont une famille de fonctions qui vont, comme leur nom l'indique, sélectionner un certain nombre de valeurs dans leur entrée pour les retourner dans un tuple en sortie. Les valeurs sélectionnées sont en relation avec leur emplacement dans le tuple reçu en entrée et la liste des emplacements choisis dépend de la fonction exacte.

Formellement, la fonction doit spécifier un tuple S, qui contient la séquence des positions des éléments à sélectionner. Ainsi, une fonction de sélection h est définie comme étant : $h_S : (V, \sqsubseteq, n) \rightarrow (V, \sqsubseteq)^M$, $h_S(I) = (I_{S_1}, I_{S_2}, ..., I_{S_M})$, où $(V, \sqsubseteq)$ est le treillis de l'analyse, $(V, \sqsubseteq, n)$ le méta-treillis qui est construit à partir de celui-ci, I est le tuple de valeurs reçues en entrée, S le tuple contenant les emplacements à sélectionner et M est le nombre d'éléments dans le tuple S.

Les fonctions de sélection sont monotones, tel que prouvé par le théorème qui suit.

Théorème 5.3.1. *Une fonction de sélection $h_S : (V, \sqsubseteq, n) \rightarrow (V, \sqsubseteq)^M$, où $(V, \sqsubseteq)$ est le treillis fini de l'analyse, n le nombre d'instructions dans le code analysé, est monotone et ce peu importe le tuple de sélection S.*

Démonstration. Soit $I \sqsubseteq I' \Rightarrow \forall_{x=1}^{n}, I_x \sqsubseteq I'_x$, pour I et I', deux tuples contenant n valeurs, tous deux éléments de $(V, \sqsubseteq, n)$. Nous devons prouver que h_S est monotone, ou formellement, que : $I \sqsubseteq I' \Rightarrow h_S(I) \sqsubseteq h_S(I')$

Nommons les éléments sélectionnés :

$$h_S(I) = (I_{S_1}, I_{S_2}, ..., I_{S_m}) = (s_1, s_2, ..., s_m)$$

et

$$h_S(I') = (I'_{S_1}, I'_{S_2}, ..., I'_{S_m}) = (s'_1, s'_2, ..., s'_m)$$

De par la relation $I \sqsubseteq I'$, nous savons que $\forall_{x=1}^{m}$, $I_{S_x} \sqsubseteq I'_{S_x}$, donc que $\forall_{x=1}^{m}$, $s_x \sqsubseteq s'_x$.

Nous savons ainsi que $h_S(I) \sqsubseteq h_S(I')$. □

5.3.2 Construction d'une sous-fonction d'instruction

La sous-fonction d'instruction prend en entrée une itération complète et retourne une valeur calculée associée à une instruction en particulier. Ainsi, la signature d'une telle fonction, pour une analyse utilisant le treillis $(V, \sqsubseteq)$ et un programme contenant n instructions, est : $g : (V, \sqsubseteq, n) \rightarrow (V, \sqsubseteq)$.

Pour chaque instruction e dans le programme analysé, une sous-fonction d'instruction est générée. Celle-ci utilise deux composantes : (1) une fonction de sélection et (2) la fonction de transition de l'instruction.

La première composante, la fonction de sélection, est générée spécifiquement pour cette instruction. Nommons h_e cette fonction de sélection. Le tuple S associé à cette fonction sera composé des indices de chacune des instructions qui peuvent précéder directement l'instruction en cours.

La seconde composante, la fonction de transition de l'instruction, est générique au type d'instruction analysé et est spécifiée par l'analyse. Cette fonction, telle que présentée dans la section 5.1.3, est une fonction monotone $f : (V, \sqsubseteq) \rightarrow (V, \sqsubseteq)$, où $(V, \sqsubseteq)$ est le treillis de l'analyse. Nommons f_e la fonction de transition associée à l'instruction e.

De façon générique, pour une instruction e dans un programme contenant n instruction, pour une analyse qui utilise le treillis $(V, \sqsubseteq)$, donc le méta-treillis $(V, \sqsubseteq, n)$, nommé W, la sous-fonction d'instruction est $f_e(m(h_e(I)))$ où $I \in W$ et m est la fonction de fusion, telle que présentée dans la section 5.1.2, définie par l'analyse.

Pour garantir la terminaison des algorithmes présentés plus loin, ces fonctions doivent être monotones, ce qui est prouvé par le théorème 5.3.2.

Théorème 5.3.2. *Pour un programme contenant n instructions et une analyse utilisant un treillis fini $(V, \sqsubseteq)$, donc le méta-treillis $(V, \sqsubseteq, n)$, les sous-fonctions d'instruction sont monotones et ce peu importe le tuple de sélection S.*

Démonstration. Soient $(V, \sqsubseteq, n)$ le méta-treillis de l'analyse, I et I', deux valeurs dans celui-ci, nous devons prouver que $I \sqsubseteq I' \Rightarrow g(I) \sqsubseteq g(I')$, où n est le nombre de valeurs dans les tuples I et I'.

En premier lieu, notons que $g(I) = f_e(m(h_e(I)))$ et que $g(I') = f_e(m(h_e(I')))$.

Puisque h_e est monotone (prouvé par le théorème 5.3.1) et que $I \sqsubseteq I'$, nous savons que $h(I) \sqsubseteq h(I')$.

Puisque m est monotone, de par les restrictions que nous posons aux analyses statiques, et que $h_e(I) \sqsubseteq h_e(I')$, nous savons que $m(h_e(I)) \sqsubseteq m(h_e(I'))$.

Puisque f_e est monotone, de par les restrictions que nous posons aux analyses statiques, et que $m(h_e(I)) \sqsubseteq m(h_e(I'))$, nous savons que $f_e(m(h_e(I))) \sqsubseteq f_e(m(h_e(I')))$.

Or $f_e(m(h_e(I))) = g(I)$ et $f_e(m(h_e(I'))) = g(I')$, nous savons donc que $I \sqsubseteq I' \Rightarrow g(I) \sqsubseteq g(I')$. □

5.4 Algorithmes d'analyse

Cette section présente deux algorithmes d'analyse de programme. Le premier qui est présenté est un algorithme itératif d'analyse et le second un algorithme utilisant un *worklist*. Ces deux algorithmes sont présentés en utilisant la représentation présentée dans la section 5.2.

5.4.1 Algorithme itératif

L'algorithme itératif (Alfred, Sethi et Jeffrey, 1986) est l'approche la plus naïve que l'on peut concevoir pour analyser un programme. Puisque approche est très simple, elle est présentée ici afin de démontrer le principe d'analyse de programme.

L'algorithme itératif applique l'approche naïve de calculer les nouvelles valeurs pour chacune des instructions du programme, que leur entrée ait changée ou non. Ainsi, lors de la transition entre les itérations, cet algorithme va appliquer les sous-fonctions associées à chacune des instructions du programme pour créer une nouvelle itération.

La fonction d'instruction, la fonction qui permet à l'analyse de passer d'une itération à l'autre est définie comme étant la fonction $F : (V, \sqsubseteq, n) \rightarrow (V, \sqsubseteq, n)$ pour une analyse utilisant le treillis $(V, \sqsubseteq)$ et un programme contenant n instructions. Dans l'algorithme itératif, cette fonction est composée d'une association entre les instructions du programme et la sous-fonction d'instruction qui y est associée. Plus précisément, la fonction prend en entrée une itération complète de l'analyse (donc, une valeur du méta-treillis de l'analyse), puis retourne une nouvelle itération. Formellement, la fonction

est :

$$F(I) = (g_1(I), g_2(I), ..., g_n(I))$$

où g_x est la sous-fonction de transition de l'instruction x.

Le théorème 5.4.1 prouve la monotonie de la fonction d'instruction. Il est possible de déduire, avec cette propriété et le théorème de Knaster-Tarski, tel que présenté dans la section 5.1.4, que :

- l'algorithme va atteindre un point fixe dans un temps fini ;
- l'ensemble des points fixes de l'analyse forment un sous-treillis du treillis de l'analyse ;
- le point fixe atteint sera le plus petit point fixe possible pour cette analyse.

Théorème 5.4.1. *La fonction d'instruction de l'algorithme itératif est monotone et ce peu importe les fonctions de transitions utilisées, tant que celles-ci respectent les règles dictées.*

Démonstration. Soit le méta-treillis de l'analyse $(V, \sqsubseteq, n)$ et I et I' deux valeurs dans celui-ci, prouvons que $I \sqsubseteq I' \Rightarrow F(I) \sqsubseteq F(I')$.

Décrivons en extension $F(I) = (g_1(I), g_2(I), ..., g_n(I))$ et $F(I') = (g_1(I'), g_2(I'), ..., g_n(I'))$, où la fonction g_x est la sous-fonction d'itération pour la ligne x, prouvée comme étant monotone par le théorème 5.3.2.

Puisque chacune des fonctions g_x est monotone, et que les entrées à celle-ci sont en relation, nous savons que les valeurs sortantes de $F(I)$ et $F(I')$ seront en relation, formellement, $\forall x \quad g_x(I) \sqsubseteq g_x(I')$.

Ceci nous montre que $I \sqsubseteq I' \Rightarrow F(I) \sqsubseteq F(I')$, donc que la fonction F est monotone. □

5.4.2 Algorithme de *worklist*

Les algorithmes de *worklist* (Muchnick, 1997) sont une technique intuitive pour optimiser les analyses statiques. Cette section présente l'approche globale de cet algorithme, par contre, beaucoup de détails lors de l'implémentation de celui-ci peuvent jouer tant sur la performance que sur l'utilisation mémoire de celui-ci.

Plutôt que de calculer les nouvelles valeurs pour chacune des instructions lors de la transition d'itération, tel que fait l'algorithme itératif, l'algorithme de *worklist* ne va calculer qu'une seule nouvelle valeur. La sélection de la valeur qui va être recalculée dépend de l'algorithme exact implémenté, quelques choix peuvent être :

- sélection aléatoire ;
- sélection séquentielle ;
- sélection à partir d'une liste d'entrées qui ont changées, soit par file ou par pile.

L'algorithme de *worklist* utilise un ensemble F, composé de fonction f_e. Chacune de celles-ci calcule une seule nouvelle valeur (applique une seule sous-fonction d'instruction), et réutilise le reste de la valeur de méta-treillis reçue en entrée.

Formellement : $f_e : (V, \sqsubseteq, n) \rightarrow (V, \sqsubseteq, n)$, $f_e(I) = (a_1, a_2, ..., a_n)$ où : $(V, \sqsubseteq)$ est le treillis défini par l'analyse, n est le nombre d'instructions dans le programme et

$$a_i = \begin{cases} g_i(I) & si\ i = e \\ I_i & sinon \end{cases}$$

où g_x est la sous-fonction de transition de l'instruction x.

Puisqu'un point fixe est atteint lorsque l'application d'une fonction donne le même résultat que son entrée, et que l'approche présentée ici utilise plusieurs fonctions, le point fixe est atteint lorsque peu importe la fonction utilisée, la valeur de sortie est la même que celle reçue en entrée. Formellement, le point fixe est atteint lorsque $\forall f_i \in F \quad f_i(I) = I$, où F est l'ensemble de fonctions valides pour cette analyse et I est la valeur de méta-treillis de l'itération courante.

Puisque le théorème de Knaster-Tarski (Tarski, 1955) est démontré en utili-

sant une seule fonction monotone et que cette approche utilise plusieurs fonctions, le théorème n'est pas applicable directement. Le théorème 5.4.7 et indirectement les théorèmes 5.4.2, 5.4.3, 5.4.4, 5.4.5 et 5.4.6, prouvent l'existence d'un point fixe et montre que celui-ci est le même que dans le cas du théorème de Knaster-Tarski.

Théorème 5.4.2. *Toutes les fonctions utilisées dans l'ensemble de fonctions de transitions pour l'algorithme de worklist sont monotones.*

Démonstration. Soient $(V, \sqsubseteq, N)$, le méta-treillis de l'analyse, I et I' deux valeurs dans le méta-treillis de l'analyse et f_e une fonction prise au hasard dans l'ensemble de fonction permises, prouvons que $I \sqsubseteq I' \Rightarrow f_e(I) \sqsubseteq f_e(I')$.

Nommons $a = f_e(I)$, sachant que a est un tuple $(a_1, a_2, ...a_N)$.

Nommons $a' = f_e(I')$, sachant que a' est un tuple $(a'_1, a'_2, ...a'_N)$.

Nous savons que tous les éléments qui ne sont pas à la position e sont les mêmes de par la description de la fonction f_e.

L'élément à la position e est calculé à partir de la fonction g_e. Formellement, $a_e = g_e(I)$ et $a'_e = g_e(I')$. Or, les sous-fonctions de transition de méta-treillis sont monotones (tel que prouvé dans la section 5.3.2), ce qui nous montre que $a_e \sqsubseteq a'_e$.

Puisque tous les éléments de a et de a' sont en relation, nous venons de prouver que $I \sqsubseteq I' \Rightarrow f_e(I) \sqsubseteq f_e(I')$. □

Théorème 5.4.3. *Les itérations dans l'algorithme de worklist sont toujours en progression les unes par rapport aux autres. Ainsi, pour une itération donnée, toutes les itérations précédant celle-ci sont plus petites que celle-ci et toutes les itérations qui vont suivre seront plus grandes. La notion de progression est définie par la relation d'ordre ($\sqsubseteq$) utilisée dans le méta-treillis utilisé dans l'analyse.*

Démonstration. Nous allons utiliser une preuve par induction pour montrer que les itérations sont toujours en progression. Pour ce faire, nous allons commencer par prouver

que lors de la toute première application d'une fonction en particulier dans l'ensemble de fonctions possibles, il y a progression. Par la suite, nous allons poser comme hypothèse que lorsque toutes les itérations sont en progression jusqu'à la N^e colonne, la $(N+1)^e$ est aussi en progression.

Cas de base

Soient le méta-treillis W, défini comme étant $(V, \sqsubseteq, N)$, f_e, une fonction d'analyse de l'approche sélectionnée aléatoirement qui modifie la ligne e et $I \in W$, la colonne précédant la première application de f_e. Prouvons que $I \sqsubseteq f_e(I)$.

De par la définition de f_e, nous savons qu'une seule valeur changera dans la colonne et celle-ci sera celle de la ligne e. Toutes les autres valeurs resteront les mêmes. De plus, nous savons que puisque cette fonction n'a jamais été exécutée sur I, la valeur de la ligne qui sera changée est $\bot$, formellement, $I_e = \bot$. Or, $\bot$ est la valeur minimale de notre treillis, ainsi, nous savons que la valeur changée ne pourra jamais être plus petite que celle-ci.

En d'autres mots, il existe deux cas pour les valeurs calculées par $f_e(I)$: (1) la valeur reste la même ou (2) la valeur change. Or, dans le second cas, nous savons que la valeur précédente était $\bot$ et que la nouvelle sera plus grande ou égale à $\bot$. Ainsi, toutes les valeurs générées par $f_e(I)$ seront plus grandes ou égales à celles de I, ce qui prouve que $I \sqsubseteq f_e(I)$, donc que la première application d'une fonction f_e est toujours en progression.

Induction

Soit le méta-treillis de l'analyse $(V, \sqsubseteq, N)$, nommé W, deux valeurs dans ce méta-treillis, K et L qui sont des *colonnes* adjacentes dans la suite d'itération de notre analyse et la fonction f_e, la fonction utilisée pour passer de K à L, formellement $L = f_e(K)$. Posons que toutes les colonnes avant ou égales à K sont en progression, et prouvons que $K \sqsubseteq L$.

Nommons la colonne J, la colonne la plus près de K générée par l'application de la fonction f_e, et I la colonne utilisée en entrée pour la générer, tels que $f_e(I) = J$. Notons qu'il est possible que $J = K$.

Par l'hypothèse d'induction, nous savons que $I \sqsubseteq J \sqsubseteq K$. Puisque la ligne e ne peut être modifiée que par la fonction f_e, nous déduisons que $J_e = K_e$ puisqu'il n'y a pas eu d'application de la fonction f_e entre les colonnes J et K. Ces relations peuvent se résumer par :

$$\forall_{n=1}^{N} \begin{cases} I_n \sqsubseteq J_n,\ J_n = K_n & si\ n = e \\ I_n \sqsubseteq J_n \sqsubseteq K_n & sinon \end{cases}$$

Nommons a, b, c et d, les valeurs associées à la ligne e dans les colonnes I, J, K et L, tels que $a = I_e$, $b = J_e$, $c = K_e$ et $d = L_e$.

Nous savons que la fonction g_e, la fonction utilisée pour calculer la valeur de la ligne e par f_e, est monotone. Nous savons aussi que toutes les valeurs calculées par f_e autre que celle de la ligne e, sont égales à leurs valeurs d'entrée. De plus, par le résumé des relations présenté plus haut, nous savons aussi que $a \sqsubseteq b$ et $b = c$, donc que $a \sqsubseteq c$.

La seule valeur différente entre K et L est celle calculée par g_e et cette fonction est monotone. Or, nous savons que $I \sqsubseteq K$. Ainsi, nous savons que $b \sqsubseteq d$, puisque $b = g_e(I)$ et $d = g_e(K)$. Nous pouvons en déduire que $c \sqsubseteq d$, donc que $K \sqsubseteq L$, puisque toutes les valeurs contenues dans L seront plus grandes ou égales à celles contenues dans K.

Résumé

Nous venons de prouver que lors la première application d'une fonction de l'ensemble de fonctions possibles il y a progression. Par la suite, nous avons prouvé que lorsque les itérations 1 à N sont en progression, la $(N+1)^e$ est aussi en progression. Ainsi, nous avons démontré que les itérations générées par cet algorithme sont toujours en progression. □

Théorème 5.4.4. *L'algorithme de worklist décrit dans la section 5.4.2 arrive à un point fixe dans un temps fini.*

Démonstration. Puisque les colonnes sont toujours en progression et que l'analyse utilise un treillis fini, nous savons qu'après un nombre fini d'itérations elle va arriver à un point fixe. □

Théorème 5.4.5. *Un point fixe trouvé par l'algorithme de worklist pour une fonction X sera aussi trouvé par l'algorithme itératif.*

Démonstration. Nous savons que la fonction de transition de colonne de la version itérative utilise les mêmes fonctions que la version *worklist*, mais qu'elle applique toutes les fonctions en même temps. Or, nous jugeons que nous avons atteint un point fixe avec la version *worklist* lorsqu'aucune fonction ne peut s'appliquer et changer les résultats.

Nommons X, un tuple représentant un point fixe dans l'algorithme de *worklist*. Puisque X est un point fixe de l'algorithme de *worklist*, nous savons que l'application de n'importe quelle des fonctions de transition ne changera pas X. Or, un point fixe dans la version itérative est simplement l'application de toutes les fonctions de transitions. Puisqu'aucune des fonctions de transitions ne changent X, l'application de toutes les fonctions ne changera pas X. Ainsi, nous savons que X est aussi un point fixe pour la version itérative. □

Théorème 5.4.6. *Pour un nombre d'itération donné, la colonne associée à l'algorithme worklist est toujours plus petit que son équivalent dans l'algorithme itératif. La notion d'une itération plus petite qu'une autre est liée à l'ordre utilisé pour le méta-treillis de l'analyse.*

Démonstration. Nous allons utiliser une preuve par induction pour prouver qu'à l'itération N, la valeur de méta-treillis associée à la colonne de l'approche par *worklist* est toujours plus petite ou égale que celle associée à l'approche itérative. Pour ce faire, nous allons montrer que la première application de chacune des fonction respecte cette relation.

Par la suite, nous allons poser comme hypothèse que si la relation est vraie pour les N premières itérations, elle est aussi vraie pour la $(N+1)^e$ itération.

Définitions

Nommons T^m la m^e colonne de la version itérative, une valeur du méta-treillis de l'analyse qui peut donc être vue comme étant un tuple composé de M valeurs de treillis.

Nommons L^m la m^e colonne de la version *worklist*, une valeur du méta-treillis de l'analyse qui peut donc être vue comme étant un tuple composé de M valeurs de treillis.

Nommons T_i^m l'élément à la i^e position de la m^e colonne de la version itérative, donc une valeur du treillis de l'analyse.

Nommons L_i^m l'élément à la i^e position de la m^e colonne de la version *worklist*, donc une valeur du treillis de l'analyse.

Cas de base

Prenons T^1 et L^1, les premières colonnes des versions itératives et de *worklist*. Nous savons que chacune de ces colonnes est initialisée avec $\perp$ dans chacune de leurs lignes, formellement $T^1 = L^1 = \perp$. Nommons F, la fonction d'instruction de l'algorithme itératif et E, une fonction de la version *worklist*, prise au hasard dans les fonctions valides, qui modifie la ligne e.

Nous devons prouver que $E(L^1) \sqsubseteq F(T^1)$.

Nous savons que $F(T^1) = (g_1(T^1), g_2(T^1), ..., g_M(T^1))$ et que $E(L^1) = (x_1, x_2, ..., x_M)$, où

$$x_i = \begin{cases} \perp & si\ n \neq e \\ g_n(L^1) & si\ n = e \end{cases}$$

Nous pouvons donc voir qu'au plus, une valeur sera modifiée par la version *worklist* et si une valeur est modifiée, ce sera celle de la ligne e. Toutes les autres valeurs resteront

$\perp$. Or, la fonction itérative va possiblement modifier chacune des lignes. Par contre, la ligne e, si elle est modifiée, sera modifiée de la même façon que la version *worklist*.

Puisque nous savons que $\forall i \perp \sqsubseteq g_i(T^1)$ et que $g_e(L^1) = g_e(T^1)$, puisque $L^1 = T^1 = \perp$, nous savons que $E(L^1) \sqsubseteq F(T^1)$.

Induction

Prenons T^n, la n^e colonne de la version itérative de l'analyse et L^n, la n^e colonne de la version *worklist*. Nommons F, la fonction d'instruction de l'algorithme itératif et E, une fonction de la version *worklist*, prise au hasard dans les fonctions valides, qui modifie la ligne e. Posons que $\forall_{n=1}^{N} L^n \sqsubseteq T^n$ et prouvons que $L^{n+1} \sqsubseteq T^{n+1}$, sachant que $L^n \sqsubseteq T^n \Rightarrow \forall_{m=1}^{M} L_m^n \sqsubseteq T_m^n$, que $L^{n+1} = E(L^n)$, que $T^{n+1} = F(T^n)$ et que M est le nombre d'éléments dans les itérations.

Nous savons que $T^{n+1} = (g_1(T^n), g_2(T^n), ..., g_M(T^n))$ et que $L^{n+1} = (x_1, x_2, ..., x_M)$, où

$$x_i = \begin{cases} L_i^n & si\ n \neq e \\ g_n(T^n) & si\ n = e \end{cases}$$

Or, de par la relation $L^n \sqsubseteq T^n$ et de par le fait que les fonctions g soient monotones, nous déduisons que $\forall_{m=1}^{M} L_m^n \sqsubseteq g_m(T^n)$ et que $g_m(L^n) \sqsubseteq g_m(T^n)$. Or, avec ces deux déductions, nous pouvons montrer que $L^{n+1} \sqsubseteq T^{n+1}$ puisque les deux cas possibles pour les valeurs de L^{n+1}, lorsque la valeur reste la même et lorsqu'elle change, seront toujours plus petits ou égaux aux valeurs correspondantes dans T^{n+1}.

En d'autres mots, (1) lorsque la valeur de L^{n+1} reste la même, la relation devient $L_i^n \sqsubseteq T_i^{n+1}$, où i est l'index d'une ligne qui n'est pas changé par la fonction L, ce qui est prouvé par la relation $L^n \sqsubseteq T^n$ et par le fait que la suite T^n soit croissante, et (2) lorsque la valeur est modifiée, la relation à prouver devient $g_n(L^n) \sqsubseteq g_n(T^n)$, ce qui est prouvé par la monotonicité de g_n.

Résumé

Nous avons prouvé que la première itération de l'algorithme de *worklist* est toujours plus petite ou égale à celle associée à l'algorithme itératif. Par la suite, nous avons démontré que, lorsque les N premières itérations de l'algorithme de *worklist* sont plus petites ou égales à leurs correspondant dans l'algorithme itératif, la $(N+1)^e$ l'est aussi.

Ainsi, nous avons démontré que toute itération de l'algorithme de *worklist* est plus petite ou égale à l'itération correspondante dans l'algorithme itératif. □

Théorème 5.4.7. *Le point fixe trouvé par l'algorithme de worklist est le même que celui trouvé par l'algorithme itératif.*

Démonstration. Soient L, le point fixe atteint à partir de $\perp$ pour la version *worklist* et T le point fixe atteint à partir de $\perp$ pour la version itérative, nous devons prouver que $L = T$.

Nous avons prouvé, dans le théorème 5.4.5, que tous les points fixes de la version *worklist* sont aussi des points fixes de la version itérative. Or, T, est prouvé par le théorème de Knaster–Tarski (Tarski, 1955) comme étant le plus petit point fixe atteignable, donc automatiquement plus petit que tous les autres points fixes, donc $T \sqsubseteq L$.

Nous avons prouvé, dans la section 5.4.6, que $L \sqsubseteq T$, puisque en tout temps la colonne de l'approche par *worklist* est *plus petite* que celle de l'approche itérative et que l'on peut voir le point fixe comme s'étendant à l'infini.

Puisque nous venons de montrer que $L \sqsubseteq T$ et que $T \sqsubseteq L$, nous pouvons déduire que $L = T$. □

5.5 Exemple complet d'analyse statique : détection de variables initialisées

5.5.1 Description de l'analyse

Cette section présente une définition d'analyse qui tente de connaître, pour un bout de code, les variables qui sont nécessairement initialisées.

Le treillis de valeurs utilisé pour l'analyse utilise un ensemble E qui est formé de l'ensemble puissance des noms de variables possibles. Avec cet ensemble, il est possible de définir le treillis de valeur utilisé pour l'analyse : $(E, \subseteq)$.

La fonction de fusion utilisée pour cette analyse est l'opération d'intersection (l'opérateur $\cap$). Cette opération est utilisée puisque l'analyse doit connaître les variables qui sont nécessairement initialisées. Ainsi, lors de fusion de plusieurs valeurs d'analyses, elle doit conserver les variables qui sont initialisées dans chacune d'entre-elles.

Cette analyse n'utilise qu'une seule fonction de transition : celle utilisée pour les affectations. Par simplicité de présentation, celle-ci sera présentée en deux cas : (a) lorsque la valeur affectée est une variable et (b) lorsque la valeur affectée est un littéral. Lorsque la valeur affectée est une variable, la fonction devra faire l'une de deux choses :

- si la valeur affectée est non initialisée, la variable où elle serait affectée doit être enlevée de l'ensemble de variables initialisées ;
- sinon, la variable doit être ajoutée à l'ensemble de variables initialisées.

Formellement, pour une instruction $x = y$, la fonction de transition utilisée pour les affectations lorsque la valeur affectée est une variable sera :

$$f(I) = \begin{cases} I - \{x\} & si\ y \notin I \\ I \cup \{x\} & si\ y \in I \end{cases}$$

Lorsque l'affectation s'effectue à partir d'un littéral, la fonction pour une instruction $x = y$, devient simplement $F(I) = I \cup \{x\}$ puisque le littéral y est toujours initialisé.

En résumé,

$$f(I) = \begin{cases} I - \{x\} & si\ y \notin I \wedge !isLiteral(y) \\ I \cup \{x\} & si\ y \in I \wedge !isLiteral(y) \\ I \cup \{x\} & si\ isLiteral(y) \end{cases}$$

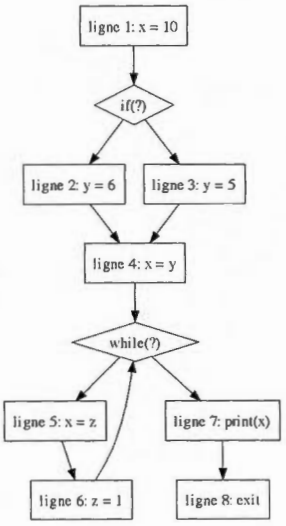

FIGURE 5.1: Graphe de flux de contrôle complet

La preuve de la monotonie de cette fonction est présentée dans le chapitre 7.

5.5.2 Graphe de flux de contrôle

Puisque la construction du graphe de flux de contrôle est dépendante du langage analysé, nous allons simplement présenter ici un graphe déjà construit. La figure 5.1 présente le graphe de flux de contrôle ainsi que les instructions qui seront utilisées pour cet exemple.

5.5.3 Déroulement de l'analyse, points communs aux algorithmes

Les deux algorithmes analysés dans ce chapitre, l'algorithme itératif et l'algorithme par *worklist*, utilisent le même graphe de flux de contrôle, donc auront tous deux 8 instructions à analyser. Ainsi, le méta-treillis de l'analyse sera d'ordre 8, donc chacune des itérations sera un tuple contenant 8 valeurs du treillis.

La création des sous-fonctions d'instruction est aussi un point commun aux deux analyses. Ainsi, les sous-fonctions qui sont créées sont les suivantes :

- Ligne 1 : $g_1(I) = f_1(m(h_1(I)))$, où $f_1(I) = I \cup \{x\}$, $h_1(I) = (\emptyset)$, $m(\emptyset) = \emptyset$ [1], donc $g_1(I) = f_1(\emptyset) = \emptyset \cup \{x\} = \{x\}$;
- Ligne 2 : $g_2(I) = f_2(m(h_2(I)))$, où $f_2(I) = I \cup \{y\}$, $h_2(I) = (I_1)$, donc $g_2(I) = I_1 \cup \{y\}$;
- Ligne 3 : $g_3(I) = f_3(m(h_3(I)))$, où $f_3(I) = I \cup \{y\}$, $h_3(I) = (I_1)$, donc $g_3(I) = I_1 \cup \{y\}$;
- Ligne 4 : $g_4(I) = f_4(m(h_4(I)))$, où $f_4(I) = \begin{cases} I - \{x\} & si\ y \notin I \\ I \cup \{x\} & si\ y \in I \end{cases}$, $h_4(I) = (I_2, I_3)$, donc $g_4(I) = \begin{cases} \bigcap(I_2, I_3) - \{x\} & si\ y \notin I \\ \bigcap(I_2, I_3) \cup \{x\} & si\ y \in I \end{cases}$;
- Ligne 5 : $g_5(I) = f_5(m(h_5(I)))$, où $f_5(I) = \begin{cases} I - \{x\} & si\ z \notin I \\ I \cup \{x\} & si\ z \in I \end{cases}$, $h_5(I) = (I_4, I_6)$, donc $g_5(I) = \begin{cases} \bigcap(I_4, I_6) - \{x\} & si\ z \notin I \\ \bigcap(I_4, I_6) \cup \{x\} & si\ z \in I \end{cases}$;
- Ligne 6 : $g_6(I) = f_6(m(h_6(I)))$, où $f_6(I) = I \cup \{z\}$, $h_6(I) = (I_5)$, donc $g_6(I) = I_5 \cup \{z\}$;
- Ligne 7 : $g_7(I) = f_7(m(h_7(I)))$, où $f_7(I) = I$, $h_7(I) = (I_6, I_4)$, donc $g_7(I) = \bigcap(I_6, I_4)$.
- Ligne 8 : $g_8(I) = f_8(m(h_8(I)))$, où $f_8(I) = I$, $h_8(I) = (I_7)$, donc $g_8(I) = I_7$.

Le départ de l'analyse est commun aux deux algorithmes. Ainsi, la valeur du méta-treillis initiale est $(\bot, \bot, \bot, \bot, \bot, \bot, \bot, \bot)$.

---

1. Cette fonction est un peu spéciale puisqu'elle est le départ de l'analyse.

	Init.	It. 1	It. 2
Instr. 1	$\perp$	$\{x\}$	$\{x\}$
Instr. 2	$\perp$	$\{x,y\}$	$\{x,y\}$
Instr. 3	$\perp$	$\{x,y\}$	$\{x,y\}$
Instr. 4	$\perp$	$\{x,y\}$	$\{x,y\}$
Instr. 5	$\perp$	$\{y\}$	$\{y\}$
Instr. 6	$\perp$	$\{y,z\}$	$\{y,z\}$
Instr. 7	$\perp$	$\{y\}$	$\{y\}$
Instr. 8	$\perp$	$\{y\}$	$\{y\}$

FIGURE 5.2: Application de l'algorithme présenté dans la section 5.5.4

5.5.4 Déroulement de l'analyse, algorithme itératif

La fonction d'instruction dans le contexte de l'algorithme itératif est définie comme étant $F(I) = (g_1(I), g_2(I), g_3(I), g_4(I), g_5(I), g_6(I), g_7(I), g_8(I))$. L'application de cette fonction donnera le tableau 5.2.

Ainsi, on peut voir que l'itération 1 est la même valeur que l'itération 2, donc qu'un point fixe est atteint. La valeur finale est donc :

- Instruction 1 : $\{x\}$;
- Instruction 2 : $\{x,y\}$;
- Instruction 3 : $\{x,y\}$;
- Instruction 4 : $\{x,y\}$;
- Instruction 5 : $\{y\}$;
- Instruction 6 : $\{y,z\}$;
- Instruction 7 : $\{y\}$;
- Instruction 8 : $\{y\}$;

5.5.5 Déroulement de l'analyse, algorithme de *worklist*

L'algorithme de *worklist* n'utilise pas une seule fonction de transition. Plutôt, il définit un ensemble de fonctions de transition, une par instruction dans le programme. Soit F, l'ensemble des fonctions de transition, tel que $F = \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8\}$, où :

- $f_1(I) = (g_1(I), I_2, I_3, I_4, I_5, I_6, I_7, I_8)$;
- $f_2(I) = (I_1, g_2(I), I_3, I_4, I_5, I_6, I_7, I_8)$;
- $f_3(I) = (I_1, I_2, g_3(I), I_4, I_5, I_6, I_7, I_8)$;
- $f_4(I) = (I_1, I_2, I_3, g_4(I), I_5, I_6, I_7, I_8)$;
- $f_5(I) = (I_1, I_2, I_3, I_4, g_5(I), I_6, I_7, I_8)$;
- $f_6(I) = (I_1, I_2, I_3, I_4, I_5, g_6(I), I_7, I_8)$;
- $f_7(I) = (I_1, I_2, I_3, I_4, I_5, I_6, g_7(I), I_8)$;
- $f_7(I) = (I_1, I_2, I_3, I_4, I_5, I_6, I_7, g_8(I))$.

Plutôt que d'utiliser un ordre aléatoire, nous allons utiliser la suite de fonctions suivantes : $f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_1, ...$ En ce faisant, le tableau de résultats serait le suivant :

	Init.	It. 1	It. 2	It. 3	It. 4	It. 5	It. 6	It. 7	It. 8	It. 9
Instr. 1	$\perp$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$
Instr. 2	$\perp$	$\perp$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$
Instr. 3	$\perp$	$\perp$	$\perp$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$
Instr. 4	$\perp$	$\perp$	$\perp$	$\perp$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$	$\{x,y\}$
Instr. 5	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\{y\}$	$\{y\}$	$\{y\}$	$\{y\}$	$\{y\}$
Instr. 6	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\{y,z\}$	$\{y,z\}$	$\{y,z\}$	$\{y,z\}$
Instr. 7	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\{y\}$	$\{y\}$	$\{y\}$
Instr. 8	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\{y\}$	$\{y\}$

Ainsi, on peut voir que l'itération 6 est la même valeur que l'itération 7 et ce peu importe la fonction appliquée, donc qu'un point fixe est atteint. La valeur finale est donc :

- Instruction 1 : $\{x\}$;
- Instruction 2 : $\{x, y\}$;
- Instruction 3 : $\{x, y\}$;
- Instruction 4 : $\{x, y\}$;
- Instruction 5 : $\{y\}$;
- Instruction 6 : $\{y, z\}$;
- Instruction 7 : $\{y\}$;
- Instruction 8 : $\{y\}$;

5.6 Conclusion

Ce chapitre a présenté la base des analyses statiques. Pour commencer, les abstractions utilisées dans le contexte d'analyses statiques, plus précisément l'abstraction de la mémoire, du flux d'exécution ainsi que de l'effet des instructions, ont été présentées. Avec celles-ci, la notion de point fixe et la définition d'itération ont été clarifiées. Par la suite, la représentation utilisée dans ce document a été présentée. La section suivante a présenté le concept de sous-fonctions d'instruction, composées de fonctions de sélection ainsi que de fonction de transition.

Par la suite, deux algorithmes simples d'analyse statique ont été présentés : un algorithme itératif ainsi qu'un algorithme basé sur des *worklists*.

Pour terminer, un exemple complet d'analyse statique, de la définition à l'exécution des deux algorithmes, a été présenté.

Les preuves présentées tout au long de ce chapitre sont importantes pour la bonne compréhension du lecteur. Or, nous n'avons pas trouvé ces preuves dans d'autres publications. Ainsi, nous avons présentés les preuves de validité pour chacun des algorithmes et chacune des formules trouvées dans ce chapitre. Il est important de noter que la

validité des deux algorithmes présentés (itératif et *worklist*) repose directement sur la monotonie des sous-fonctions d'instruction.

CHAPITRE VI

ANALYSE STATIQUE INTERPROCÉDURALE

Ce chapitre présente les analyses statiques interprocédurales, donc celles qui ont besoin d'information sur les méthodes contenues dans le programme pour pouvoir opérer. Ce chapitre se base sur la théorie présentée dans le chapitre 5.

Les informations et algorithmes présentés dans ce chapitre requièrent les propriétés suivantes :

- toutes les fonctions de transition utilisées dans les analyses sont monotones ;
- les analyses utilisent des fonctions de fusion monotone ;
- les analyses utilisent des treillis finis pour leurs valeurs.

Plus précisément, ce chapitre débute en expliquant le concept de version de méthode qui est utilisé par nos analyses. Par la suite, les particularités des analyses interprocédurales, en rapport avec les analyses intraprocédurales, sont présentées une par une : en premier lieu, le graphe de flux de contrôle, par la suite la fonction de transition de l'instruction d'appel de méthode, suivis par la génération du tableau de résultats et finalement le déroulement de l'analyse.

6.1 Concept de versions de méthode

Les techniques utilisées couramment en analyse statique interprocédurale gèrent la récursivité de façon dégradée : une partie des informations générées par les méthodes est perdue. Pour perdre le moins d'information possible, la technique présentée ici représente

```
fun foo do
   # Something ...
end
```

FIGURE 6.1: Simple code Nit présentant la méthode *foo* avec seulement le paramètre implicite *self*

les différentes façons dont une méthode donnée peut être appelée en créant des duplicatas de celle-ci pour chacune des valeurs d'entrées possibles[1]. Les valeurs d'entrées sont en fait une association entre chacune des références en entrées (sans oublier la référence implicite self) et une valeur dans le treillis de l'analyse pour chacune d'entre elles.

Ainsi, le graphe d'appel et les informations contenues dans l'analyse contiennent plusieurs *versions* d'une même méthode : une pour chacune des façons dont celle-ci puisse être appelée. La notion d'une *façon* d'appeler une méthode est dépendante de l'analyse. Ainsi, la fonction foo présentée dans la figure 6.1, utilisant seulement le paramètre implicite self, analysée à l'aide du treillis $(\{A, B, C\}, \sqsubseteq)$ présenterait 8 versions :

- $(\{\})$
- $(\{A\})$
- $(\{B\})$
- $(\{C\})$
- $(\{A, B\})$
- $(\{A, C\})$
- $(\{B, C\})$
- $(\{A, B, C\})$

Il est important de comprendre que chacune des versions d'une même méthode est analysée indépendamment des autres. De plus, si l'algorithme utilisé n'est pas itératif,

1. Le nombre de versions et les valeurs d'entrées sont dépendantes de l'analyse effectuée et ce nombre est fini puisque le treillis utilisé par l'analyse est fini. Par contre, ce nombre peut être grand selon l'analyse : par exemple, l'analyse d'une fonction qui contient 2 paramètres et un treillis d'analyse qui contient 10 valeurs contiendra 1024 versions. De façon générique, pour une fonction qui contient x paramètres et un treillis d'analyse qui contient y valeurs, il existe x^y versions de la méthode.

```
int fun (...) {
1.  x = ...
    if (...) {
2.    x = ...
    } else {
3.    return 0;
    }
4.  return x;
}
```

FIGURE 6.2: Simple code C avec une fonction

il est possible que certaines versions ne soient pas analysées ou ne soit que partiellement analysées puisqu'elles ne sont pas utilisées par le programme.

6.2 Graphe de flux de contrôle

Le concept de base de graphe de flux de contrôle est présenté dans la section 4.2. L'ajout de méthodes et d'instructions d'appel de méthode à la représentation demande quelques modifications. Cette section présente ces modifications, plus précisément :

1. la représentation des méthodes ;
2. la représentation des envois de messages.

Dans un graphe de flux de contrôle, une méthode nécessite la création d'un noeud d'entrée et d'un noeud de sortie. Ceux-ci sont nécessaires à la gestion des retours de méthode et des appels. L'algorithme de création de graphe va donc analyser chacune des méthodes indépendamment et ajouter un noeud *start* et un noeud *end*, qui seront respectivement le début et la fin de la méthode, autour de chacune de celles-ci. Si la méthode contient des instructions de type return, celles-ci vont simplement créer des arrêtes vers le noeud *end*. Une méthode simple est présentée dans les figures 6.2, 6.3 et 6.4. La première montre le code de la méthode, la seconde le graphe de flux de contrôle associé à celle-ci et la dernière le tableau de présentation des résultats d'analyse associé.

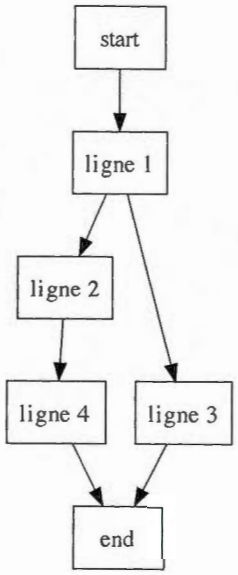

FIGURE 6.3: Représentation graphique d'une analyse de flux pour le code de la figure 6.2

Ligne	Itération 0	Itération 1	Itération 2
fun.start	$\perp$	?	...
fun.ligne1	$\perp$	?	...
fun.ligne2	$\perp$	?	...
fun.ligne3	$\perp$	?	...
fun.ligne4	$\perp$	?	...
fun.end	$\perp$	?	...

FIGURE 6.4: Représentation par tableau de l'analyse de la fonction présentée dans le code de la figure 6.2

Ces trois figures montrent que les noeuds *start* et *end* font vraiment partie de l'analyse et qu'ils sont utilisées tout au long de celle-ci.

Une façon simple de visualiser la notion d'envoi de message est de concevoir les appels comme un switch sur le type dynamique du receveur ou comme une série de ifs qui vont tester le type dynamique du receveur. Par exemple, les méthodes bar, baz, spam et ham du code de la figure 6.5 peuvent être réécrites telles que présentées dans la figure 6.6. Cette dernière figure utilise deux notations spéciales :

- l'opérateur isa : est utilisé pour comparer le type dynamique d'une référence ;
- la syntaxe classe :: methode est utilisée pour faire un appel direct (sans envoi de message) à une méthode.

6.3 Fonction de transition de l'instruction d'appel

Les instructions d'appel de méthode sont les seules instructions qui s'ajoutent lors de l'analyse interprocédurale (en comparaison avec les analyses intraprocédurales). La fonction de transition utilisée pour les instructions d'appels doit prendre la valeur la plus précise qu'elle peut trouver à partir de celles calculées parmi les différentes versions de la fonction qui sont comparables avec celle dont l'algorithme a besoin pour l'instruction analysée.

Pour garantir la monotonie de la fonction de transition de l'instruction d'appel, celle-ci doit tenir en compte toutes les versions de la méthode qui pourraient être utilisées plus loin dans l'analyse. Pour ce faire, la valeur calculée par la fonction de transition est le plus grand parent commun aux valeurs de retour des versions de la méthode dont l'entrée est autant ou plus précise que l'entrée calculée pour l'instruction en cours.

Par exemple, supposons une fonction f qui prend un seul argument et qui est analysé en utilisant le treillis $(\{a, b, c\}, \sqsubseteq)$ où $a \sqsubseteq b \sqsubseteq c$. Une instruction qui fait appel à la fonction f et qui a comme valeur d'entrée $\{b\}$ utilisera la valeur la plus *grande* entre les valeurs calculées lors de l'itération précédente pour les versions $\{a\}$ et $\{b\}$ de la fonction f.

```
class A
	fun foo do end
end

class B super A
	redef fun foo do end
end

class C super A
	redef fun foo do end
end

class D
	super B
	super C
	redef fun foo do end
end

fun bar (a: A) do
	a.foo
end

fun baz (b: B) do
	b.foo
end

fun spam (c: C) do
	c.foo
end

fun ham (d: D) do
	d.foo
end

var a: A = new A
var b: B = new B
var c: C = new C
var d: D = new D
bar(a)
baz(b)
spam(c)
ham(d)
```

FIGURE 6.5: Code Nit avec envoi de message

```
fun bar (a: A) do
       if (a isa D) then
              a.D::foo
       else if (a isa B) then
              a.B::foo
       else if (a isa C) then
              a.C::foo
       else // a isa A
              a.A::foo
       end
end

fun baz (b: B) do
       if (a isa B) then
              a.B::foo
       else // a isa A
              a.A::foo
       end
end

fun spam (c: C) do
       if (a isa C) then
              a.C::foo
       else // a isa A
              a.A::foo
       end
end

fun ham (d: D) do
       a.D::foo
end
```

FIGURE 6.6: Code Ni; réécrit pour ne pas avoir d'envoi de message

La fonction de transition de l'instruction d'appel a besoin d'accéder à plus d'informations que les fonctions de transitions présentées jusqu'à présent : elle doit connaître les valeurs de sortie de l'itération précédente pour chacune des versions de la fonction appelée. Ainsi, plutôt que de définir une fonction de transition, nous allons définir une sous-fonction de transition. En ce faisant, cette fonction recevra toutes les informations voulues (soit l'itération précédente complète) et devra donner en sortie la valeur de sortie de l'instruction donnée.

Formellement, la sous-fonction de transition de l'instruction d'appel s'applique sur le méta-treillis fini $(V, \sqsubseteq, n)$, a le domaine $f : (V, \sqsubseteq, n) \to (V, \sqsubseteq)$ et est la suivante :

$$f(I) = \bigsqcap_{i \in V, i \sqsubseteq m(h(I))} L_i$$

où :

- I est la valeur d'entrée de l'instruction et est un élément du méta-treillis $(V, \sqsubseteq, n)$;
- h est la fonction de sélection associée à l'instruction analysée ;
- m est la fonction de fusion, telle que définie par l'analyse ;
- L est un tuple bâti tel que présenté ci-dessous.

La bonne compréhension du contenu du tuple L utilisé dans la fonction précédente est cruciale. Les éléments contenus dans L proviennent de l'itération I reçue en paramètre. Chacun est un lien entre une version de la méthode liée à l'instruction analysée et la valeur associée à la sortie de cette même version de la méthode dans l'itération I. Formellement, le tuple L est généré tel que : $\forall i \in (V, \sqsubseteq), L_i = O_i$, où O_i est la valeur de sortie associée à la version i de la méthode dans l'itération I. Il est important de comprendre que, d'une itération à l'autre, la *structure* de l'itération n'est pas modifiée. Ainsi, d'une itération à l'autre, la valeur représentant la sortie d'une fonction reste au même *index* dans l'itération.

La sous-fonction de transition d'instruction associée à une instruction d'appel est prouvée monotone par le théorème 6.3.1.

Théorème 6.3.1. *Soit la sous-fonction de transition d'appel* $f(I) = \bigsqcap_{i \in (V,\sqsubseteq), i \sqsubseteq m(h(I))} L_i$ *où :*

- *I est la valeur d'entrée de l'instruction et est un élément de $(V, \sqsubseteq, n)$;*
- *h est la fonction de sélection associée à l'instruction analysée ;*
- *m est la fonction de fusion, telle que définie par l'analyse ;*
- *L est un tuple bâti tel que présenté précédemment.*

Cette fonction est monotone et ce peu importe le treillis $(V, \sqsubseteq)$ tant que celui-ci est fini.

Démonstration. Puisque les fonctions h et m sont monotones, nous allons réduire la fonction analysée à : $f(I) = \bigsqcap_{i \in (V,\sqsubseteq), i \sqsubseteq \alpha} L_i$, où $\alpha = m(h(I))$ et est donc un élément de $(V, \sqsubseteq)$. Puisque h et m sont monotones, nous savons que $I \sqsubseteq I' \Rightarrow \alpha \sqsubseteq \alpha'$, où $\alpha' = m(h(I'))$.

Avec cette réduction, nous devons prouver : $I \sqsubseteq I' \Rightarrow \bigsqcap_{i \in (V,\sqsubseteq), i \sqsubseteq \alpha} L_i \sqsubseteq \bigsqcap_{i \in (V,\sqsubseteq), i \sqsubseteq \alpha'} L'_i$.

Prenons une itération I et bâtissons le tuple L associé à celle-ci. Puisque la fonction f est associée à un appel de méthode en particulier dans un programme, nous connaissons toutes les versions de cette méthode. Formellement : $\forall v \in V, L_v = O_v$, où O_v est la sortie pour la version v de la méthode dans I.

Regardons une itération I', telle que $I' \sqsubseteq I$. Ainsi, $\forall v \in V, L'_v = O'_v$, où O'_v est la sortie pour la version v de la méthode dans I'.

Puisque la structure des itérations n'est pas modifiée entre les itérations et que $I' \sqsubseteq I$, alors nous savons que $\forall v \in V, L'_v \sqsubseteq L_v$.

Définissons X et X', tels que $X = \{i | i \in (V, \sqsubseteq), i \sqsubseteq \alpha\}$ et $X' = \{i | i \in (V, \sqsubseteq), i \sqsubseteq \alpha'\}$. Puisque $\alpha' \sqsubseteq \alpha$, nous savons que tous les éléments contenus dans X' sont aussi contenus dans X.

Définissons Y et Y', tels que $Y = \{L_x | \forall x \in X\}$ et $Y' = \{L'_x | \forall x \in X'\}$.

Réécrivons l'équation à prouver : $I' \sqsubseteq I \Rightarrow \bigsqcap Y' \sqsubseteq \bigsqcap Y$.

Nous savons que tous les éléments dans X' sont contenus dans X et que $\forall v \in V, L'_v \sqsubseteq L_v$. Ainsi, nous savons que $\bigsqcap\{L'_x | \forall x \in X'\} \sqsubseteq \bigsqcap\{L_x | \forall x \in X'\}$. Ceci montre que le plus petit majorant commun aux éléments de l'ensemble Y' est aussi un majorant à au moins une partie des éléments de l'ensemble Y. Ainsi, le calcul du plus petit majorant commun aux éléments de l'ensemble Y sera égal ou plus grand que le plus petit majorant commun aux éléments de l'ensemble Y', formellement : $\bigsqcap Y' \sqsubseteq \bigsqcap Y$.

Ainsi, la sous-fonction d'instruction est monotone. □

6.4 Génération du tableau des résultats

Lorsque le tableau de résultats de l'analyse est généré, celui-ci doit présenter une copie de la fonction par version de celle-ci. En ce faisant, il devient explicite que les résultats sont indépendants pour chacune des versions de chacune des méthodes. Cette particularité est la seule différence avec le tableau présenté dans la section 5.2. Les figures 6.7, 6.8 et 6.9 présentent respectivement le code, le graphe de flux de contrôle et la structure du tableau de résultat pour un exemple de code simple contenant deux fonctions.

6.5 Déroulement de l'analyse

Puisque la gestion des méthode n'est qu'une fonction de transition parmi les autres de l'analyse, rien ne change pour l'analyse en soit. De plus, le traitement de la récursivité ne demande aucune modification de l'analyse. Ceci est dû au fait que nous analysons tout le programme, plus précisément toutes les versions de toutes les méthodes, en parallèle.

```
fun foo: Int do
1.  x = bar
2.  return x + 2
end

fun bar: Int do
1.  x = ...
    if(...) then
2.    x = ...
    else
3.    return 0
    end
4.  return x
end
```

FIGURE 6.7: Simple code Nit avec plusieurs fonctions

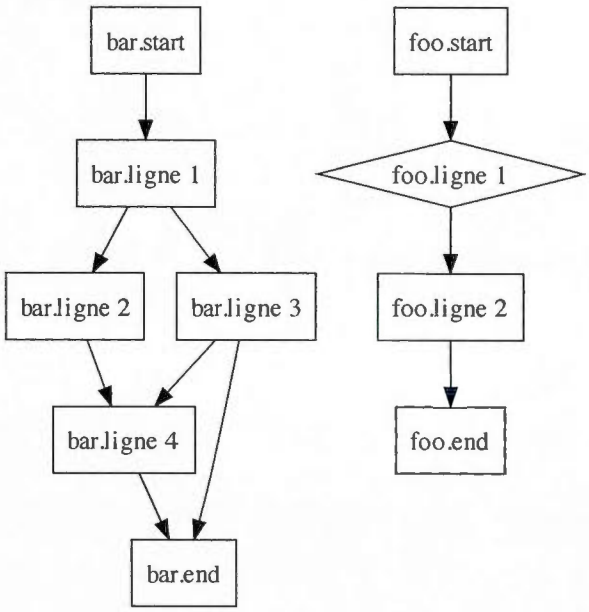

FIGURE 6.8: Représentation graphique d'une analyse de flux pour le code de la figure 6.7

Ligne	Itération 0	Itération 1	Itération 2
foo1.start	{}	{}	...
foo1.ligne1	$\perp$	?	...
foo1.ligne2	$\perp$	?	...
foo1.end	$\perp$	?	...
bar1.start	{}	{}	...
bar1.ligne1	$\perp$	?	...
bar1.ligne2	$\perp$	?	...
bar1.ligne3	$\perp$	?	...
bar1.ligne4	$\perp$	?	...
bar1.end	$\perp$	?	...
foo2.start	$\{self\}$	$\{self\}$	...
foo2.ligne1	$\perp$	?	...
foo2.ligne2	$\perp$	?	...
foo2.end	$\perp$	?	...
bar2.start	$\{self\}$	$\{self\}$	...
bar2.ligne1	$\perp$	?	...
bar2.ligne2	$\perp$	?	...
bar2.ligne3	$\perp$	?	...
bar2.ligne4	$\perp$	?	...
bar2.end	$\perp$	?	...

FIGURE 6.9: Représentation par tableau de l'analyse de la fonction présentée dans le code de la figure 6.7

6.6 Conclusion

Après avoir expliqué le concept de version de méthode, la technique qui permet à nos analyses d'avoir des résultats précis tout en garantissant la terminaison de nos algorithmes, ce chapitre a présenté les particularités des analyses interprocédurales, en rapport avec les analyses intraprocédurales.

Plus précisément, furent abordés :

- le graphe de flux de contrôle ;
- la fonction de transition de l'instruction d'appel de méthode ;
- la génération du tableau de résultats ;
- le déroulement de l'analyse.

La section présentant le graphe de flux de contrôle montre qu'avec les informations contenues dans le graphe d'appel du programme, il est simple de transformer un langage avec appel de méthode en un langage avec appels directs. Par la suite, la monotonie de la nouvelle fonction de transition, celle gérant les appels de méthodes, fut démontrée. La section suivante montre les différences contenues dans le tableau de résultats : le fait qu'il existe des duplicatas de chacune des méthodes, un par version de chacune des méthodes. Avec ces informations, la dernière section a montré que l'analyse n'est pas modifiée pour l'ajout d'appels de méthodes.

CHAPITRE VII

SUPPRESSION DES TESTS DYNAMIQUES

Ce chapitre présente trois analyses statiques que nous proposons. Deux de ces analyses utilisent la notion de références définitives. Dans le contexte de ce document, une référence est dite définitive lorsqu'il n'y a aucun doute quant à l'objet pointé par cette référence. Ainsi, dans l'analyse de référence définitive vers l'objet en construction, une référence est dite définitive vers l'objet en construction lorsqu'il est certain que cette référence référencie l'objet en construction dans tous les contextes d'exécution qui se rendent à la ligne en cours.

Les analyses présentées dans ce chapitre se basent sur la théorie présentée dans les chapitres 5 et 6.

En premier lieu, les éléments communs aux trois analyses seront présentés. Plus précisément, la première section présente l'abstraction de mémoire utilisée et la seconde les fonctions de transitions ainsi que les preuves utilisées par les trois analyses.

L'analyse présentée dans la section 7.3 traite des références définitives vers l'objet en construction. Les résultats de cette analyse seront utilisés par la troisième analyse pour connaître les affectations vers un attribut qui sont effectives pour l'objet en construction.

Celle qui sera présentée dans la section 7.4 traite des références définitives vers un objet qui n'est pas en construction. Les résultats de celle-ci seront utilisés par la

troisième analyse pour savoir si une affectation vers un attribut affecte bien un objet complètement construit ou non.

La dernière analyse, présentée dans la section 7.5, utilise les résultats des deux premières pour trouver les attributs qui sont initialisés en ce qui concerne l'objet en cours de construction. Ainsi, les seules affectations qui sont considérées sont celles qui ont comme receveur l'objet en construction et comme valeur affectée un objet complètement construit.

La section finale de ce chapitre se consacre à l'optimisation du code intermédiaire. L'optimisation qui y est présentée utilise les informations provenant des trois analyses pour supprimer les tests isset .

7.1 Abstraction de l'information

L'abstraction de la mémoire, telle que définie dans la section 5.1.1 utilise des treillis. Ceux-ci sont composés d'un ensemble de valeurs ainsi que d'une relation d'ordre entre ces valeurs. Cette section présente ces deux composantes, (1) l'ensemble de valeurs et (2) la relation d'ordre, pour définir le treillis utilisé par les analyses statiques présentées dans ce chapitre.

De plus, cette section présente la fonction de fusion utilisée en conjonction avec ce treillis.

7.1.1 Ensemble de valeur

L'ensemble de valeurs utilisé pour la construction des treillis utilisés pour les analyses statiques présentées dans cette section est composé de l'ensemble puissance de toutes les références existantes dans le programme. Les types de références que l'on retrouve dans l'ensemble sont les suivants :

- paramètres ;

```
class A
      var attr_A: A
end

class B super A
end

class C super B
      redef var attr_A: B
end

fun foo(x: Int) do
      var bar: Int
end
```

FIGURE 7.1: Code pour l'exemple de treillis, associé au graphique 7.2

- paramètre implicite du receveur courant [1] ;
- variables locales ;
- attributs.

Pour mieux représenter graphiquement les diverses références qui existent à l'intérieur du programme, les règles suivantes sont utilisées dans ce document :

- pour un attribut : le nom est composé du nom de la classe où il est déclaré et de son nom ;
- pour un paramètre : le nom est composé du nom de la classe, du nom de la méthode et du nom du paramètre ;
- pour une variable : le nom est composé du nom de la classe, du nom de la méthode et du nom de la variable. Si plusieurs variables existent avec le même nom dans une même méthode, un numéro séquentiel est ajouté à ceux-ci.

Il est important de comprendre que ces notations ne devraient pas être utilisées dans un algorithme réel puisque certains des noms peuvent être ambigus.

---

1. Selon le langage, ce receveur peut s'appeler *this*, *self*, *me*, ... Ce document utilise le mot clé *self* pour indiquer le receveur courant

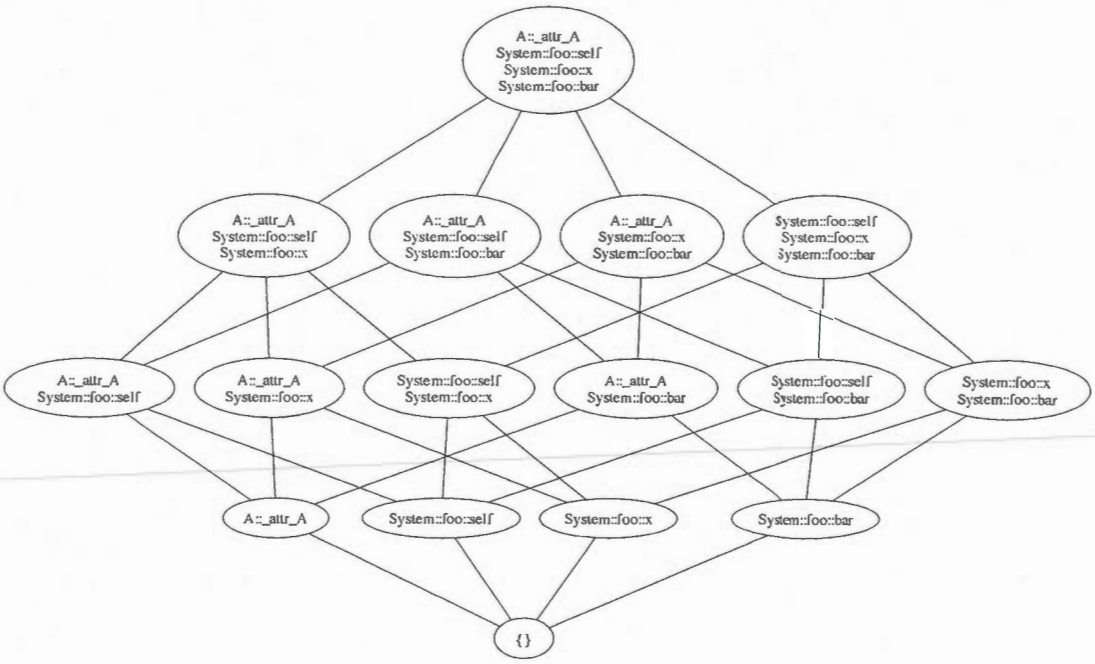

FIGURE 7.2: Exemple de treillis, présenté sous la forme d'un diagramme de Hasse (Birkhoff, 1995), pour le code de la figure 7.1. On y voit un attribut (A::_attr_A), une variable locale (System::foo::bar), un paramètre (System::foo::x) ainsi que le receveur courant (System::foo::self)

Tous ces types de références sont illustrés dans les figures 7.1 et 7.2. Celles-ci présentent un exemple de code ainsi qu'un exemple de treillis pour un attribut (A::_attr_A), une variable locale (System::foo :: bar), un paramètre (System::foo :: x) ainsi que le receveur courant de la fonction System::foo (System::foo :: self).

Puisque la taille du programme est connue, le nombre de références est fini. Ainsi, l'ensemble de valeurs utilisé pour le treillis est fini, ce qui montre que le treillis est aussi fini.

7.1.2 Fonction d'ordre entre les valeurs

Puisque l'ensemble de valeurs utilisé dans le treillis est composé d'un ensemble puissance, l'opération d'inclusion ($\sqsubseteq$) sera utilisée pour la fonction d'ordre. La figure 7.2 présente un exemple de la relation d'ordre sous forme de diagramme de Hasse (Birkhoff, 1995).

7.1.3 Fonction de fusion

La fonction de fusion, telle que présentée dans la section 5.1.2, est utilisée lorsque plusieurs chemins d'exécution peuvent se rendre à un point donné dans le programme. Ceci est courant en programmation, notamment lors de boucles et lors de branchements conditionnels.

Les analyses présentées dans les prochaines sections recherchent des informations sur les références dites définitives dans le programme, que ce soit des références vers un objet en construction ou vers un objet qui n'est pas en construction. Une référence est dite définitive s'il est certain hors de tout doute que celle-ci référencie le type de référence recherché.

Pour garantir de ne conserver que les références définitives, la fonction de fusion doit s'assurer de conserver les informations qui sont présentes dans tous les contextes

fusionnés. Pour ce faire, l'opération d'intersection d'ensemble sera utilisée pour gérer la fusion de valeurs dans le treillis.

Formellement, pour un ensemble X contenant n valeurs membres d'un treillis $(V, \sqsubseteq)$, où $n \geq 1$, la fonction est :

$$Z(X) = \bigcap_{i=1}^{n} X_i$$

Ainsi, la fusion entre les valeurs de treillis $\{A, B, C\}$, $\{C, D, E\}$ et $\{C, E, F\}$ serait $\{C\}$.

Pour garantir la découverte d'un point fixe dans l'analyse, la fonction de fusion doit être monotone. Cette propriété est prouvée par le théorème 7.1.1.

Théorème 7.1.1. *La fonction de fusion* $Z(X) = \cap_{i=1}^{n} X_i$ *où* $n \geq 1$ *et* $\forall_{m=0}^{n} X_m \in W$ *est monotone pour tout treillis* W *fini.*

Démonstration. Soit $X_1, X_2, ...X_n, Z_1, Z_2, ..., Z_n$ des valeurs dans le treillis W, prouvons que

$$X_1 \sqsubseteq Z_1, X_2 \sqsubseteq Z_2, ..., X_n \sqsubseteq Z_n \Rightarrow F(X_1, X_2, ..., X_n) \sqsubseteq F(Z_1, Z_2, ..., Z_n)$$

Nous avons donc :

$$F(X_1, X_2, ..., X_n) = X_1 \cap X_2 \cap ... \cap X_n$$

$$F(Z_1, Z_2, ..., Z_n) = Z_1 \cap Z_2 \cap ... \cap Z_n$$

Nous devons démontrer :

$$X_1 \cap X_2 \cap ... \cap X_n \overset{?}{\sqsubseteq} Z_1 \cap Z_2 \cap ... \cap Z_n$$

Nous savons que :

$$\forall i \in (1..n), X_i \sqsubseteq Z_i$$

Définissons A et B :

$$A = X_1 \cap X_2 \cap ... \cap X_n$$

$$B = Z_1 \cap Z_2 \cap ... \cap Z_n$$

Nous pouvons voir que :

$$\forall a, a \in A \Rightarrow a \in B$$

Puisque nous savons que si a est membre d'un des ensembles X_i, alors il sera membre de l'ensemble correspondant Z_i.

□

7.2 Fonctions de transition

Cette section présente les fonctions de transition qui sont utilisées pour nos analyses. De plus, les preuves de monotonie de chacune de celles-ci seront présentées.

Pour présenter ces fonctions, nous allons utiliser deux valeurs du treillis W. Le treillis W est le treillis tel que défini dans la section 7.1. Les deux valeurs qui seront utilisées tout au long de cette section sont :

- I : la valeur d'entrée de la fonction de transition ;
- O : la valeur de sortie de la fonction de transition.

Les instructions gérées sont : (1) les affectations vers une variable ou un paramètre, (2) les affectations vers un attribut, (3) les instantiations de classes, (4) les appels natifs, (5) les appels de méthodes, ainsi que (6) les autres instructions. Cette dernière catégorie regroupe les branchements conditionnels, les boucles et les tests conditionnels.

7.2.1 Affectation vers une variable ou un paramètre

Une affectation vers une variable ou un paramètre, à partir de tout type de référence (variable, paramètre ou attribut) peut être exprimé comme étant :

```
d = s
```

Où, d est la référence de destination et s est la référence source. La fonction de transfert pour une telle instruction est :

$$f(I) = \begin{cases} I - \{d\} & \text{si } s \notin I \\ I \cup \{d\} & \text{sinon} \end{cases}$$

Lorsque l'expression à la droite de l'affectation est une expression composée, dès qu'une sous-partie de l'expression fait partie de I, alors l'expression complète est considérée comme faisant partie de I pour la fonction $f(I)$ ci-dessus.

Théorème 7.2.1. *Soient I, J, deux valeurs dans le treillis fini W, telles que* $I \sqsubseteq J$, *et*

$$f(I) = \begin{cases} I - \{d\} & \textit{si } s \notin I \\ I \cup \{d\} & \textit{sinon} \end{cases}$$

alors $f(I) \sqsubseteq f(J)$, *donc f est une fonction monotone.*

Démonstration. Soient I, J, deux valeurs dans le treillis fini W, telles que $I \sqsubseteq J$, alors nous introduisons $I' = f(I)$ et $J' = f(J)$. Prouvons que $I' \sqsubseteq J'$.

La relation $I \sqsubseteq J$ montre que $\forall i, i \in I \rightarrow i \in J$.

De par la construction de la fonction de transfert, il y a plusieurs cas à analyser :

Numéro	$s \in I$	$s \in J$	$d \in I$	$d \in J$
(1)	non	non	non	non
(2)	non	non	non	oui
(3)	non	non	oui	oui
(4)	non	non	oui	non
(5)	non	oui	non	non
(6)	non	oui	non	oui
(7)	non	oui	oui	oui
(8)	non	oui	oui	non
(9)	oui	oui	non	non
(10)	oui	oui	non	oui
(11)	oui	oui	oui	oui
(12)	oui	oui	oui	non
(13)	oui	non	non	non
(14)	oui	non	non	oui
(15)	oui	non	oui	oui
(16)	oui	non	oui	non

Cas 1 : $s \notin I$, $s \notin J$, $d \notin I$, $d \notin J$

Dans un tel cas, il faut démontrer que $I - \{d\} \sqsubseteq J - \{d\}$, sachant que d n'est ni dans I, ni dans J. Puisque l'élément d n'est dans aucun des ensembles, nous pouvons enlever les soustractions inutiles, l'équation devient donc : $I \sqsubseteq J$, ce qui montre que la relation reste la même.

Cas 2 : $s \notin I$, $s \notin J$, $d \notin I$, $d \in J$

Dans un tel cas, il faut démontrer que $I \sqsubseteq J - \{d\}$, sachant que $d \notin I$. Puisque l'élément enlevé de l'ensemble J n'était pas présent dans le sous-ensemble I, la relation est préservée.

Cas 3 : $s \notin I$, $s \notin J$, $d \in I$, $d \in J$

Dans un tel cas, il faut démontrer que $I - \{d\} \sqsubseteq J - \{d\}$. Visuellement, nous pouvons voir que toutes les valeurs précédemment dans les ensembles I et J sont préservées par la fonction, sauf la valeur d. Cette dernière n'est dans aucun des deux ensembles.

Cas 5 : $s \notin I$, $s \in J$, $d \notin I$, $d \notin J$

Dans un tel cas, il faut démontrer que $I \sqsubseteq J \cup \{d\}$, sachant que $d \notin I$. L'ajout de l'élément d au super-ensemble J, sachant qu'il n'est pas dans le sous-ensemble, n'invalide pas la relation.

Cas 6 : $s \notin I$, $s \in J$, $d \notin I$, $d \in J$

Dans un tel cas, il faut démontrer que $I - \{d\} \sqsubseteq J \cup \{d\}$, sachant que $d \notin I$ et $d \in J$. Puisque nous savons que d n'est pas dans I, nous pouvons enlever la soustraction dans la partie gauche de l'équation, et puisque d est déjà dans J, nous pouvons enlever l'union dans la partie de droite. Ainsi, l'équation devient $I \sqsubseteq J$, ce montre que la relation reste la même.

Cas 7 : $s \notin I$, $s \in J$, $d \in I$, $d \in J$

Dans un tel cas, il faut démontrer que $I - \{d\} \sqsubseteq J$, or le fait d'enlever un élément dans le sous-ensemble n'invalide pas la relation.

Cas 9 : $s \in I$, $s \in J$, $d \notin I$, $d \notin J$

Dans un tel cas, il faut démontrer que $I \cup \{d\} \sqsubseteq J \cup \{d\}$. Nous pouvons voir visuellement que la relation est maintenue puisque l'élément est ajouté aux deux ensembles.

Cas 10 : $s \in I$, $s \in J$, $d \notin I$, $d \in J$

Dans un tel cas, il faut démontrer que $I \cup \{d\} \sqsubseteq J$, sachant que $d \in J$. Puisque l'élément ajouté dans le sous-ensemble est déjà présent dans l'ensemble J, la relation est préservée.

Cas 11 : $s \in I$, $s \in J$, $d \in I$, $d \in J$

Dans un tel cas, il faut démontrer que $I \cup \{d\} \sqsubseteq J \cup \{d\}$, sachant que d est déjà présent dans les deux ensembles. Puisque d est dans les deux ensembles, nous pouvons enlever les deux unions dans l'équation. Ainsi, l'équation devient $I \sqsubseteq J$, ce montre que la relation reste la même.

Cas 4, 8 et 12 : $d \in I$, $d \notin J$

Puisque $I \sqsubseteq J$, il est impossible que d soit membre de I sans être membre de J. Ces lignes sont invalides, elle ne peuvent pas se produire, et ne nécessitent donc pas de preuve.

Cas 13, 14, 15 et 16 : $s \in I$, $s \notin J$

Puisque $I \sqsubseteq J$, il est impossible que s soit membre de I sans être membre de J. Ces lignes sont invalides, elle ne peuvent pas se produire, et ne nécessitent donc pas de preuve. □

7.2.2 Affectation vers un attribut

Dans l'approche présentée dans ce chapitre, les attributs des diverses instances d'une même classe ne sont pas distingués. Deux instances de deux classes ayant une classe parent commune où un attribut est défini ne sont pas non plus distingués. Par contre, de par la génération des *noms* des attributs (voir section 7.1.1), deux attributs de même nom dans deux instances de classes différentes n'étant pas en relation sont distingués.

Au départ de l'analyse, tous les attributs de toutes les classes autres que celle en cours d'initialisation font partie de l'ensemble de références initialisées. Dès qu'un attribut se voit affecté d'une valeur potentiellement non initialisée, il est impossible pour cet attribut de redevenir initialisé.

Ainsi, une affectation vers un attribut, peu importe la source de l'affectation, est

gérée un peu différemment d'une affectation vers une variable ou un paramètre. On peut représenter une telle instruction par :

```
attr = s
```

Où, s est la référence source et `attr` est un attribut. La fonction de transition est la suivante :

$$f(I) = \begin{cases} I - \{attr\} & \text{si } s \notin I \\ I & \text{sinon} \end{cases}$$

Théorème 7.2.2. *Soient I, J, deux valeurs dans le treillis fini W, telles que* $I \sqsubseteq J$, *et*

$$f(I) = \begin{cases} I - \{attr\} & \textit{si } s \notin I \\ I & \textit{sinon} \end{cases}$$

alors $f(I) \sqsubseteq f(J)$, *donc f est une fonction monotone.*

Démonstration. Soient I, J, deux valeurs dans le treillis fini W, telles que $I \sqsubseteq J$, alors nous introduisons $I' = f(I)$ et $J' = f(J)$.

La relation $I \sqsubseteq J$ montre que $\forall i, i \in I \rightarrow i \in J$.

Prouvons que $I' \sqsubseteq J'$.

De par la construction de la fonction de transfert, il y a plusieurs cas à analyser :

Numéro	$s \in I$	$attr \in J$	$attr \in I$	$attr \in J$
(1)	non	non	non	non
(2)	non	non	non	oui
(3)	non	non	oui	oui
(4)	non	non	oui	non
(5)	non	oui	non	non
(6)	non	oui	non	oui
(7)	non	oui	oui	oui
(8)	non	oui	oui	non
(9)	oui	oui	non	non
(10)	oui	oui	non	oui
(11)	oui	oui	oui	oui
(12)	oui	oui	oui	non
(13)	oui	non	non	non
(14)	oui	non	non	oui
(15)	oui	non	oui	oui
(16)	oui	non	oui	non

Cas 1 : $s \notin I$, $s \notin J$, $attr \notin I$, $attr \notin J$

Dans un tel cas, il faut démontrer que $I - \{attr\} \sqsubseteq J - \{attr\}$, sachant que $attr$ n,est dans aucun des ensembles. Puisque $attr$ n'est dans aucun des ensembles, nous pouvons enlever les deux soustractions tout en conservant la validité de l'équation. Ainsi, l'équation devient $I \sqsubseteq J$, ce qui montre que la relation est préservée.

Cas 2 : $s \notin I$, $s \notin J$, $attr \notin I$, $attr \in J$

Dans un tel cas, il faut démontrer que $I \sqsubseteq J - \{attr\}$, sachant que $attr \notin I$. Puisque l'élément enlevé de l'ensemble J n'était pas présent dans le sous-ensemble I, la relation est préservée.

Cas 3 : $s \notin I$, $s \notin J$, $attr \in I$, $attr \in J$

Dans un tel cas, il faut démontrer que $I - \{attr\} \sqsubseteq J - \{attr\}$. Visuellement, nous pouvons voir que toutes les valeurs précédemment contenues dans les ensembles I et J sont préservées par la fonction, sauf la valeur $attr$. Cette dernière n'est dans aucun des deux ensembles. Ce cas est trivial à démontrer.

Cas 7 : $s \notin I$, $s \in J$, $attr \in I$, $attr \in J$

Dans un tel cas, il faut démontrer que $I - \{attr\} \sqsubseteq J$. Le fait d'enlever un élément dans le sous-ensemble n'invalide pas la relation.

Cas 4, 8 et 12 : $attr \in I$, $attr \notin J$

Puisque $I \sqsubseteq J$, il est impossible que $attr$ soit membre de I sans être membre de J. Ces lignes sont invalides, elle ne peuvent pas se produire, et ne nécessitent donc pas de preuve.

Cas 5 et 6 : $s \notin I$, $s \in J$, $attr \notin I$

Dans un tel cas, il faut prouver que $I - \{attr\} \sqsubseteq J$ sachant que $attr \notin I$. Puisque $attr$ n'est pas dans l'ensemble I, nous pouvons enlever l'union de la partie gauche de l'équation. L'équation devient donc $I \sqsubseteq J$, ce qui montre que la relation est préservée.

Cas 9, 10 et 11 : $s \in I$

De par la fonction de transfert, un tel cas ne va pas modifier les valeurs de retour, elles seront les mêmes que l'ensemble en entrée. Ainsi, $I' = I$ et $J' = J$, ce qui montre que $I' \sqsubseteq J'$.

Cas 13, 14, 15 et 16 : $s \in I$, $s \notin J$

Puisque $I \sqsubseteq J$, il est impossible que s soit membre de I sans être membre de J. Ces lignes sont invalides, elle ne peuvent pas se produire, et ne nécessitent donc pas de preuve. □

7.2.3 Instantiation d'une classe

Dans la majorité des langages, ce type d'instruction correspond à un `new`. Puisque la base que nous présentons ici sera utilisée pour collecter des informations à propos de deux types de références, (1) les références définitives vers l'objet en construction et (2) les références définitives vers un objet construit, il y a deux choix pour la fonction de transition associée à cette instruction : (1) si l'analyse suit les références définitives vers l'objet en construction et (2) si l'analyse suit les références définitives vers un objet qui n'est pas celui en construction. Dans tous les cas, on peut visualiser cette instruction comme étant :

```
d = new Y(A0, ..., An)
```

Où, `d` est la référence de destination, `Y` est la classe à instantier et `A0..An` sont les arguments du constructeur.

Dans un premier cas, lorsque l'analyse collecte des informations en rapport avec les références définitives vers l'objet en construction, il est impossible que la référence retournée par une telle instruction soit celle déjà en construction. Dans le cadre d'une telle instruction, la fonction de transfert est :

$$f(I) = I - \{d\}$$

Dans un second cas, nous savons que la référence retournée par ce type d'instruction sera toujours une référence autre que celle en construction. Ainsi, la fonction de transfert est :

$$f(I) = I \cup \{d\}$$

Théorème 7.2.3. *La fonction $f(I) = I - \{d\}$ est monotone, formellement pour toutes valeurs I, J du treillis fini W, $I \sqsubseteq J \rightarrow f(I) \sqsubseteq f(J)$.*

Démonstration. Soit I, J, I', J' des valeurs dans le treillis W telles que $I' = f(I)$, $J' = f(J)$ et $I \sqsubseteq J$. Prouvons que $I' \sqsubseteq J'$.

Pour prouver cette relation, il faut prouver que $I - \{d\} \sqsubseteq J - \{d\}$, or, il est trivial de voir que cette relation est respectée. □

Théorème 7.2.4. *La fonction $f(I) = I \cup \{d\}$ est monotone, formellement pour toutes valeurs I, J du treillis fini W, $I \sqsubseteq J \rightarrow f(I) \sqsubseteq f(J)$.*

Démonstration. Soit I, J, I', J' des valeurs dans le treillis W telles que $I' = f(I)$, $J' = f(J)$ et $I \sqsubseteq J$. Prouvons que $I' \sqsubseteq J'$.

Pour prouver cette relation, il faut prouver que $I \cup \{d\} \sqsubseteq J \cup \{d\}$, or, il est trivial de voir que cette relation est respectée. □

7.2.4 Appels natifs

Un appel natif est un appel envoyé vers une fonction qui existe en dehors du programme analysé. Les instructions bas niveau (entrée/sortie, *thread*, ...) utilisent généralement ce système. Le point important à propos de ce genre d'appel est que le code source associé à ceux-ci n'est pas disponibles dans le compilateur. Ainsi, il est impossible pour nos analyse de s'assurer de l'absence de déréférencement de nul dans ces fonctions.

Puisque nos analyses ne peuvent pas connaître les effets d'un appel natif, nous n'avons aucune informations sur la valeur de retour de cet appel. On peut visualiser un appel natif comme étant :

```
d = call(A0 .. An)
```

Où, `d` est la référence de destination, `call` est l'appel natif et `A0..An` sont les arguments de la fonction native. Dans le cadre d'une telle instruction, la fonction de transfert est :

$$f(I) = I - \{d\}$$

Cette fonction est déjà prouvée comme étant monotone par le théorème 7.2.3.

7.2.5 Les appels

Rien de spécifique ne doit être effectué dans les analyses présentées dans ce chapitre en ce qui concerne les appels de méthodes. Ainsi, la fonction de transition pour les appels est $f(I) = I$ qui est trivialement monotone.

7.2.6 Les autres instructions

Les instructions restantes (les branchements conditionnels, les boucles et les tests conditionnels) sont gérées lors de la génération du graphe de contrôle et n'influencent pas les données au niveau d'une fonction de transition. On pourrait toutefois voir ces instructions comme étant des fonctions identités, soit :

$$f(I) = I$$

Cette fonction est trivialement monotone.

7.3 Analyse des références définitives vers l'objet en construction

Cette section présente l'analyse qui a pour but de suivre les références définitives vers l'objet en construction. Cette analyse utilise la base présentée dans les chapitres précédents. Les résultats de cette analyse sont stockés pour chacune des lignes où elle est effectuée.

Cette section présente plus précisément les particularités de cette analyse, suivi d'un exemple complet montrant les résultats attendus de cette analyse.

7.3.1 Particularités

Les particularités de cette analyse se divisent en deux parties : (1) valeur de treillis de départ de l'analyse et (2) fonction de transfert de l'instruction d'instantiation.

L'analyse débute dans un constructeur donné. À ce moment, la seule référence

```
class MaClasse
       init do
              var x: MaClasse = part3(self)
       end

       fun part2: MaClasse do
              return self
       end

       fun part3(o: MaClasse): MaClasse do
              return o.part2
       end
end
```

FIGURE 7.3: Exemple complet, analyse de référence définitive vers l'objet en construction

qui existe dans le système qui soit définitivement l'objet en construction est `self`. Pour cette raison, la valeur initiale est $\{self\}$.

Puisque cette analyse cherche à trouver les références définitives vers l'objet en construction, nous allons utiliser la première version de la fonction de transfert de l'instruction d'instantiation. Celle-ci sera donc :

$$f(I) = I - \{d\}$$

7.3.2 Exemple complet

La figure 7.3 présente un extrait de code utilisé pour obtenir les résultats de l'analyse de référence définitive vers l'objet en construction présentés dans la figure 7.4. Par simplicité d'affichage, les noms de variables et paramètres seront tronqués lors de l'affichage de ceux-ci.

7.4 Analyse des références définitives vers un objet qui n'est pas en construction

Cette section présente l'analyse qui a pour but de suivre les références définitives vers un objet qui n'est pas en construction. Cette analyse utilise la base présentée dans

	Itération 0	Itération 1	Itération 2	Itération 3
début de init, version $\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$
var x = part3(self)	{}	{}	{}	{}
fin de init, v. $\{self\}$	-	-	-	-
début de part2, v. {}	{}	{}	{}	{}
return self	{}	{}	{}	{}
fin de part2, v. {}	{}	{}	{}	{}
début de part2, v. $\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$
return self	{}	$\{self\}$	$\{self\}$	$\{self\}$
fin de part2, v. $\{self\}$	{}	{}	$\{self, ret\}$	$\{self, ret\}$
début de part3, v. {}	{}	{}	{}	{}
return o.part2	{}	{}	{}	{}
fin de part3, v. {}	{}	{}	{}	{}
début de part3, v. $\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$
return o.part2	{}	$\{self\}$	$\{self\}$	$\{self\}$
fin de part3, v. {}	{}	{}	$\{self\}$	$\{self\}$
début de part3, v. $\{o\}$	$\{o\}$	$\{o\}$	$\{o\}$	$\{o\}$
return o.part2	{}	$\{o\}$	$\{o\}$	$\{o\}$
fin de part3, v. {}	{}	{}	$\{ret\}$	$\{ret\}$
début de part3, v. $\{self, o\}$	$\{self, o\}$	$\{self, o\}$	$\{self, o\}$	$\{self, o\}$
return o.part2	{}	$\{self, o\}$	$\{self, o\}$	$\{self, o\}$
fin de part3, v. {}	{}	{}	$\{self, o, ret\}$	$\{self, o, ret\}$

FIGURE 7.4: Tableau de résultat, analyse de référence définitive vers l'objet en construction du code de la figure 7.3. Chacune des sections de ce tableau présentent une version d'une méthode. Dans une section donnée, *self* désigne toujours la référence de la méthode de cette section.

```
class AutreClasse
        var attr_x: Object
        init do
                var x: MaClasse = new MaClasse(self)
                attr_x = x
        end
end
class MaClasse
        var y: Object
        init(x: AutreClasse) do
                y = x.attr_x
        end
end
```

FIGURE 7.5: Exemple d'objet non construit passé à un constructeur

les chapitres précédents. Les résultats de cette analyse sont stockés pour chacune des lignes où nos analyses sont effectuées. Nous présenteront plus précisément les particularités de cette analyse, suivi d'un exemple complet montrant les résultats attendus de cette analyse.

7.4.1 Particularités

Les particularités de cette analyse se divisent en deux parties : (1) valeur de treillis de départ de l'analyse et (2) fonction de transfert de l'instruction d'instantiation.

L'analyse débute dans un constructeur donné. À ce moment, nous savons que toutes les références qui existent dans le programme sont des objets complètement construits. Ainsi, tous les attributs existant dans le programme et étant dans une autre classe que celle en cours de construction ainsi que tous les paramètres du constructeur sont considérés comme étant des références vers des objets construits. Il est important de bien comprendre ce point. Bien qu'en réalité il se pourrait que certaines de ces références soient en cours de construction, les erreurs découlant de celles-ci seront détectées lors de l'analyse du constructeur fautif et non de celui que nous analysons présentement. On voit dans la figure 7.5 un exemple de ce type de cas. Lors de l'analyse du constructeur

```
class MaClasse
        init(x: MaClasse) do
                var y: MaClasse = part3(x)
        end

        fun part2: MaClasse do
                return self
        end

        fun part3(o: MaClasse): MaClasse do
                return o.part2
        end
end
```

FIGURE 7.6: Exemple complet, analyse de référence définitive vers un objet qui n'est pas en construction

de `MaClasse`, nous allons supposer que tous les attributs et tous les paramètres sont bien construits, ce qui donne comme résultat de cette analyse que l'instance est complètement construite à la fin du constructeur. Or, lorsque le constructeur de `AutreClasse` est lancé, il va construire une instance de `MaClasse` qui ne sera pas valide. Ce cas est détecté lors de l'analyse du constructeur de `AutreClasse` qui va, lorsqu'il va analyser la ligne `y = x.attr_x`, savoir que `x` est en cours de construction et que l'attribut n'est pas encore initialisé, ce qui va donner une erreur lors de la compilation du programme.

Puisque cette analyse cherche à trouver les références définitives vers les objets complètement construits, nous allons utiliser la seconde version de la fonction de transfert de l'instruction d'instantiation. Celle-ci sera donc :

$$f(I) = I \cup \{d\}$$

7.4.2 Exemple complet

La figure 7.6 présente un extrait de code utilisé pour obtenir les résultats de l'analyse de référence définitive vers les objets complètement construits présentés dans

	Itération 0	Itération 1	Itération 2	Itération 3
début de init, version $\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$
var x = part3(self)	{}	{}	{}	{}
fin de init, v. $\{self\}$	-	-	-	-
début de part2, v. {}	{}	{}	{}	{}
return self	{}	{}	{}	{}
fin de part2, v. {}	{}	{}	{}	{}
début de part2, v. $\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$
return self	{}	$\{self\}$	$\{self\}$	$\{self\}$
fin de part2, v. $\{self\}$	{}	{}	$\{self, ret\}$	$\{self, ret\}$
début de part3, v. {}	{}	{}	{}	{}
return o.part2	{}	{}	{}	{}
fin de part3, v. {}	{}	{}	{}	{}
début de part3, v. $\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$	$\{self\}$
return o.part2	{}	$\{self\}$	$\{self\}$	$\{self\}$
fin de part3, v. {}	{}	{}	$\{self\}$	$\{self\}$
début de part3, v. $\{o\}$	$\{o\}$	$\{o\}$	$\{o\}$	$\{o\}$
return o.part2	{}	$\{o\}$	$\{o\}$	$\{o\}$
fin de part3, v. {}	{}	{}	$\{ret\}$	$\{ret\}$
début de part3, v. $\{self, o\}$	$\{self, o\}$	$\{self, o\}$	$\{self, o\}$	$\{self, o\}$
return o.part2	{}	$\{self, o\}$	$\{self, o\}$	$\{self, o\}$
fin de part3, v. {}	{}	{}	$\{self, o, ret\}$	$\{self, o, ret\}$

FIGURE 7.7: Tableau de résultat, analyse de référence définitive vers l'objet en construction du code de la figure 7.6

la figure 7.7. Cette figure présente les résultats de façon un peu différente du reste de ce document : plutôt que de présenter le résultat directement, les résultats indiqués sont en fait les références qui ne sont pas dans l'ensemble de résultat. En ce faisant, la lecture du tableau devient plus simple. De plus, par simplicité d'affichage, les noms de variables et paramètres seront tronqués lors de l'affichage de ceux-ci.

7.5 Analyse des attributs initialisés

Cette analyse utilise les résultats des deux premières analyses présentées dans ce chapitre. Elle a pour but de détecter, pour chaque instruction atteignable à partir de chacun des constructeurs, quels attributs associés à l'instance en cours de construction sont initialisés. Cette analyse est plus simple que les deux premières présentées dans cette section puisque toutes les informations nécessaires à celle-ci ont déjà été propagées par les deux premières analyses.

Cette analyse va parcourir, à partir de chacun des constructeurs, toutes les lignes accessibles à partir de celui-ci pour propager la liste des attributs initialisés à chacune des instructions parcourues. Formellement, le treillis de valeur possible pour l'analyse est composé de tous les attributs contenus dans la classe courante et est ordonné par la fonction d'inclusion d'ensemble. Ainsi, on peut voir ce treillis comme étant un sous-ensemble du treillis utilisé pour les deux premières analyses. De plus, tel qu'avec les deux analyses précédentes, la fonction de fusion est le concept ensembliste d'intersection.

La seule instruction qui va modifier la valeur de treillis est l'affectation vers un attribut. Pour ces instructions, seules celles dont l'attribut est contenu dans l'objet en construction, tel que défini par la première analyse, et dont la valeur est un objet complètement construit, tel que défini par la seconde analyse, vont utiliser la fonction suivante. Ainsi, pour une instruction du type

```
objet.attribut = valeur
```

```
class AutreClasse
      readable var _attribute: Int
end

class MaClasse
      var first: Object
      var second: Object

      init(x: AutreClasse, y: Int) do
1.          var other: MaClasse = self
2.          first = y
3.          other.second = x.attribute
      end
end
```

FIGURE 7.8: Exemple complet, analyse des attributs initialisés

la fonction de transition est :

$$f(I) = \begin{cases} I - \{attribut\} & \text{si objet} \in K \text{ et valeur} \in L \\ I & \text{sinon} \end{cases}$$

Où K est la valeur associée à l'instruction courante pour l'analyse de références définitives vers l'objet en construction et L est la valeur associée à l'instruction courante pour l'analyse de références définitives vers un objet qui n'est pas en construction. Les preuves pour ce genre de fonction de transfert ont été effectuées dans les sections précédentes.

7.5.1 Exemple complet

On voit dans la figure 7.8 un exemple complet d'analyse des attributs initialisés. Nous allons analyser le constructeur `init`, en considérant que les deux analyses présentées précédemment ont été effectuées et que leurs résultats sont ceux présentés dans la figure 7.9. Les résultats qui sont générés par cette analyse sont présentés dans le tableau 7.10.

Ligne	Résultats de la première analyse	Résultats de la seconde analyse
1	$\{self, other\}$	$\{x, y, AutreClasse :: attribute\}$
2	$\{self, other\}$	$\{x, y, AutreClasse :: attribute\}$
3	$\{self, other\}$	$\{x, y, AutreClasse :: attribute\}$

FIGURE 7.9: Résultats des analyses de références définitives pour le code de la figure 7.8. La première analyse est celle de références définitives vers l'objet en construction alors que la seconde est celle de références définitives vers un objet qui n'est pas en construction

Ligne	Résultats
1	$\{\}$
2	$\{first\}$
3	$\{first, second\}$

FIGURE 7.10: Résultats de l'analyse des attributs initialisés pour le code de la figure 7.8. Les résultats sont les attributs initialisés après l'exécution de la ligne donnée.

7.6 Optimisation du code intermédiaire

Cette section présente l'optimisation de suppression de tests dynamiques, basée sur les résultats des analyses des sections précédentes. C'est avec cette optimisation que le compilateur va enlever les tests dynamiques ajoutés à travers le code par l'approche de types *nullables* présentée au début de ce document. Cette optimisation va analyser chacune des instructions du code pour tenter d'optimiser certaines constructions qui sont présentes dans celui-ci : elle tente de supprimer les lignes semblables à `x = isset(y)` [2].

Plus précisément, cette section présente l'approche utilisée par cette optimisation ainsi qu'un exemple de celle-ci.

7.6.1 Approche

Cette optimisation va analyser tout le code présent dans le programme en ce concentrant sur les instructions de type *isset* [3]. Pour chacun des *isset* découverts, l'optimisation va regarder les différentes informations générées par l'analyse d'attributs initialisés pour la ligne en cours. Trois cas sont possibles pour une instruction de type *isset* :

- chacun des contextes présents dans l'analyse d'attributs initialisés présente l'attribut comme étant initialisé : l'optimisation peut avoir lieu et l'instruction peut être supprimée ;
- aucun contexte n'existe dans l'analyse d'attributs initialisés : ceci indique que l'instruction courante ne peut pas être atteinte à partir d'un constructeur et que l'instruction peut être optimisée ;
- un ou plusieurs contextes présents dans l'analyse d'attributs initialisés présente l'attribut comme n'étant pas initialisé : l'optimisation est impossible. Ainsi, dès

2. Rappel : l'opérateur `isset` est utilisé pour savoir si un attribut est initialisé.

3. L'optimisation ne fait aucune différence entre un *isset* écrit par le programmeur et un *isset* ajouté automatiquement par le compilateur.

Ligne	Résultat
1	`Personne::init`$(\emptyset) = \{nom, telephone, age\}$
2	`Personne::init`$(\emptyset) = \{nom, telephone, age\}$
3	`Personne::init`$(\emptyset) = \{nom, telephone, age\}$
4	$\emptyset$
5	$\emptyset$
6	$\emptyset$

FIGURE 7.11: Résultat de l'analyse d'attributs initialisé appliquée sur le code de la figure 1.2

qu'un contexte présente l'attribut comme n'étant pas initialisé, peu importe ce que les autres contextes apportent comme information, l'optimisation est impossible.

Cette optimisation très simple permet de façon sécuritaire d'enlever les tests dynamiques utilisés sur un attribut que les analyses détectent comme étant toujours initialisé lorsque le programme atteint l'instruction de test dynamique.

7.6.2 Exemple

La figure 1.1 présente une classe simple que nous allons utiliser pour démontrer comment cette optimisation fonctionne. Les résultats de l'analyse de cette classe sont présentés dans le tableau 7.11. Lors de l'optimisation du code contenu dans la figure 1.2, les lignes 1, 2 et 3 sont optimisées puisque le tableau de résultats note qu'à ces lignes, tous les contextes (dans ce cas, le seul contexte) définissent ces attributs comme étant initialisés. Les lignes 4, 5 et 6 sont optimisées puisqu'il n'existe aucun contexte à ces endroits, ce qui signifie qu'il est impossible de se rendre à ces instructions à partir d'un constructeur. Le produit final de l'optimisation est présenté dans la figure 1.3.

7.7 Conclusion

Ce chapitre a présenté les trois analyses proposées par notre approche. En premier lieu, l'abstraction de mémoire utilisée ainsi que les fonctions de transitions des diverses instructions ont été présentées.

La première analyse visitée fut l'analyse de référence définitive vers l'objet en construction. Celle-ci va trouver, pour chaque instruction atteignable à partir de chacun des constructeurs, les différentes références qui *pointent* vers l'objet en construction par le constructeur courant et ce dans tous les contextes d'exécution.

La seconde qui fut présentée est l'analyse de référence définitive vers un objet qui n'est pas en construction. Bien qu'à première vue, celle-ci semble simplement être l'opposée de la première analyse présentée, elle ne va pas simplement prendre le reste des objets non contenus dans la première : elle doit s'assurer qu'une référence *pointe* de façon définitive vers un objet complètement construit. Pour ce faire, elle utilise la même logique que la première analyse, mais a accès à plus d'informations : lors du début de l'analyse, il est certain que toutes les références contenues dans le système sont des références vers un objet complètement construit sauf la référence nommée *self*. Cette information permet à l'analyse d'accéder à des informations plus précises en ce qui concerne les attributs.

La troisième analyse utilise les résultats propagés par les deux premières. Elle se concentre sur les instructions de type affectation vers un attribut, plus précisément celles dont l'attribut affecté est contenu dans l'objet en construction et dont la valeur affectée est une référence vers un objet complètement construit. Cette dernière analyse est vraiment le fondement de notre approche et sera utilisée lors de nos optimisations. Elle n'est simple que parce que les deux premières, sur lesquelles elle repose, sont définies de façon stricte et fournissent des informations très précises.

Ce chapitre a démontré à quel point l'optimisation des tests dynamiques de type `isset` est facile lorsque les informations générées par nos analyses sont accessibles. Plus

précisément, après avoir expliqué l'approche utilisée par l'optimisation, un exemple complet a été effectué.

CHAPITRE VIII

RÉSULTATS, IMPLÉMENTATION ET LIMITATIONS

En premier lieu, ce chapitre présente les résultats expérimentaux ayant ressorti de notre implémentation dans le langage Nit. Par la suite, il montre deux points où l'implémentation diverge de la théorie : (1) l'évaluation paresseuse des méthodes ainsi que (2) l'inlining[1] des accesseurs automatiques. Finalement, deux limitations de l'approche sont présentées : (1) la gestion des appels natifs ainsi que (2) les systèmes d'exceptions.

8.1 Résultats expérimentaux

Cette section présente les résultats obtenus en lançant nos analyses sur le compilateur Nit. La figure 8.1 présente quelques caractéristiques du compilateur. Nous pouvons voir que le nombre de points d'entrée, soit le nombre de constructeurs atteignables, est assez petit (en dehors du *parser*). De plus, le nombre de méthodes atteignables à partir de ces constructeurs est aussi assez restreint. Ces nombres sont dûs au fait que les programmeurs sont habitués de créer des constructeurs aussi simples que possible.

On peut voir dans la figure 8.2 les résultats de nos optimisations : tous les tests ajoutés automatiquement, ainsi que ceux que les programmeurs avaient ajoutés manuellement, ont été enlevés du compilateur. Ceci prouve qu'il est impossible qu'une

1. L'inlining est une optimisation qui permet de copier le code d'une fonction appelée directement au site d'appel.

Méthode et constructeur atteignables	7000
Constructeurs atteignables	
en tout	1576
en dehors du *parser*	164
Méthode et constructeur atteignables à partir d'un constructeur	
en tout	4583
en dehors du *parser*	588
en dehors du *parser* et étant une méthode	424

FIGURE 8.1: Information à propos du compilateur Nit, graphe d'appel généré par RTA (Bacon et Sweeney, 1996).

`isset` (explicites et implicites)	2912	
`isset` optimisés (explicites et implicites)	2912	(100%)
Temps d'exécution avec les tests	1m03	
Temps d'exécution sans les tests	0m58	(-8%)

FIGURE 8.2: Précision des analyses statiques et temps d'exécution nécessaire pour les tests dynamiques dans le compilateur Nit.

exception ajoutée par notre approche de gestion de déréférencement de nul soit lancée lors de l'exécution du compilateur.

8.2 Implémentation

Cette section présente deux points où l'implémentation diverge de la théorie : (1) l'évaluation paresseuse des méthodes ainsi que (2) l'inlining des accesseurs automatiques.

8.2.1 Évaluation paresseuse des méthodes

Lors de la présentation de nos analyses (chapitre 7) ainsi que lors de la présentation des analyses interprocédurales (chapitre 6), il fut indiqué que toutes les versions de toutes les méthodes doivent être analysées systématiquement. En pratique, l'implémentation utilisée présente une approche paresseuse : une version d'une méthode n'est analysée que lorsqu'elle est nécessaire.

8.2.2 Inlining des accesseurs automatiques

Cette section présente une optimisation [2] nécessaire du code intermédiaire utilisé dans le compilateur du langage Nit pour permettre à nos analyses d'avoir de meilleurs résultats. Pour ce faire, l'inlining doit être effectué sur l'ensemble du code intermédiaire généré et ce avant le passage de nos analyses. Pour bien comprendre la raison pour laquelle cette optimisation est nécessaire, un aperçu de la gestion des attributs en Nit sera présenté en premier lieu.

Le langage Nit utilise un concept d'encapsulation d'attributs restreignant l'accès direct aux attributs à l'instance à laquelle appartiennent ces attributs. Pour diminuer le code nécessaire, plus précisément l'écriture manuelle et répétitive d'accesseurs en

---

2. Cette optimisation n'est qu'une triviale optimisation d'inlining qui aide à la distinction entre les diverses versions des méthodes.

```
class MaClasse
	readable var _attr: Int

	init do
		if 2 == 1 then
			print "Attribut: {attr}"
		end
		_attr = 1
	end
end

var a: MaClasse = new MaClasse
for i in [0..100000] do
	print a.attr
end
```

FIGURE 8.3: Utilisation de types nullables résultant en une non-optimisation

lecture ou écriture (*getter* et *setter*), le langage propose au programmeur d'utiliser deux annotations lors de la déclaration de ses attributs : *readable* et *writable*. Ainsi, il existe quatre façons de déclarer un attribut dans le code :

1. `var _myAttr: Int;`
2. `readable var _myAttr: Int;`
3. `writable var _myAttr: Int;`
4. `readable writable var _myAttr: Int.`

La première façon déclare l'attribut, mais ne le rend pas accessible aux autres classes. On pourrait voir cet attribut comme étant *private* en Java. La seconde introduit l'attribut avec un accesseur en lecture (*getter*) automatique. Celui-ci aura la signature : `fun myAttr: Int` et pourra être exécuté par les autres classes. La troisième déclaration créera automatiquement un accesseur en écriture (*setter*), mais pas d'accesseur en lecture. L'accesseur en écriture aura la signature : `fun myAttr=(value: Int)` et sera accessible de l'extérieur de la classe. La dernière déclaration créera les deux accesseurs présentés précédemment.

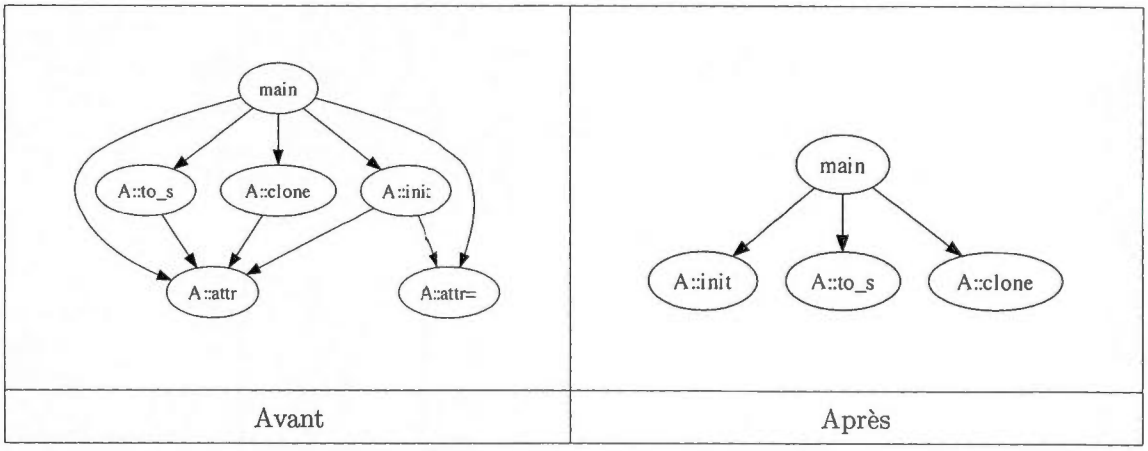

FIGURE 8.4: Graphe d'appel avant et après l'inlining

Le problème résultant de ces fonctions automatiques est visible lors de l'analyse du graphe d'appel (figure 8.4). On voit que ces méthodes sont appelées par un très grand nombre de contextes différents et, de par la nature de nos analyses et de nos optimisations, lorsqu'un de ces contextes présente certaines caractéristiques défavorables, l'optimisation devient impossible. On voit dans l'exemple 8.3 un exemple de code qui pourrait polluer nos optimisations si l'inlining des accesseurs n'était pas effectuée. On y voit l'utilisation d'un attribut non initialisé (à travers l'accesseur automatique) durant la construction de l'objet. Nos analyses vont donc indiquer, pour le contexte d'analyse du constructeur, que l'attribut `_unAttribut` est potentiellement non-initialisé lors de son accès au début du constructeur. Lorsque la phase d'optimisation va s'exécuter sur le code, nous ne pourrons pas enlever le test dynamique ajouté automatiquement dans l'accesseur puisque ce contexte est potentiellement dangereux. Ainsi, tous les accès dans le code à l'attribut `_unAttribut` à travers l'accesseur effectueront le test dynamique alors qu'il n'est jamais vraiment nécessaire. On voit donc qu'une utilisation un peu plus exotique de nos types nullables pourrait potentiellement empêcher l'optimisation d'un accesseur utilisé à travers tout le code.

Bien que le code intermédiaire génère explicitement le code pour les accesseurs

```
class MaClasse
	readable writable var _unAttribut: Int

	init do
		unAttribut = 1
	end
end

var a: MaClasse = new MaClasse
print a.unAttribut
```

FIGURE 8.5: Démonstration de l'inlining, code écrit par le programmeur

```
class MaClass
	var unAttribut: Int

	fun unAttribut: Int do
		var valid: Bool = isset unAttribut
		if not valid then
			abort
		end
		return unAttribut
	end
	fun unAttribut(val: Int) do unAttribut = val

	init do
		unAttribut = 1
	end
end

var a: MaClase = new MaClasse
print a.unAttribut
```

FIGURE 8.6: Démonstration de l'inlining, code intermédiaire généré

```
class MaClass
	var unAttribut: Int

	init do
		unAttribut = 1
	end
end

var a: MaClasse = new MaClasse
var valid: Bool = isset a.unAttribut
if not valid then
	abort
end
print a.unAttribut
```

FIGURE 8.7: Démonstration de l'inlining, code intermédiaire optimisé

automatiques, cette optimisation a pour but de déplacer ce code vers les sites d'appels des accesseurs automatiques. Notre approche pour effectuer l'inlining automatique des accesseurs est simple : modifier les accès aux accesseurs automatiques par des accès directs aux attributs. Cette approche est sécuritaire puisque les restrictions d'accès aux attributs sont garanties par l'analyseur statique.

On voit dans les figures 8.5, 8.6 et 8.7 les différentes phases du code relié aux accesseurs automatiques. La figure 8.5 montre le code écrit par le programmeur. On voit l'utilisation des deux accesseurs automatiques. La figure 8.6 démontre le concept derrière le code intermédiaire généré. On voit que les deux accesseurs qui étaient "cachés" dans la première figure sont maintenant explicites dans le compilateur. Finalement, la figure 8.7 présente le concept derrière le code intermédiaire lorsque l'inlining a été effectué. On voit que les appels de méthodes ont étés remplacés par un accès direct aux attributs, tant en lecture qu'en écriture.

En effectuant cette transformation du code intermédiaire, nos analyses et nos optimisations ne vont plus fusionner les résultats des divers sites d'appels vers les accesseurs en lecture, mais bien les considérer comme étant des sites totalement différents,

permettant l'isolation des contextes et empêchant la propagation d'un contexte invalide à travers tout le programme.

8.3 Limitations

Cette section présente deux limitations de notre approche : (1) la gestion des appels natifs ainsi que (2) les systèmes d'exceptions.

8.3.1 Limitations causées par les appels *natifs*

Cette section présente les limitations qui existent dans plusieurs approches de types nullables en ce qui concerne les appels natifs. En effet, pour le système de type, un appel vers une fonction native est une boîte noire. Un test peut être ajouté au retour de la fonction native pour valider la nullabilité de la valeur de retour. Par contre, il n'existe aucune façon de garantir la "bonne conduite" de la fonction appelée. Le langage *C* est utilisé pour les fonctions natives en Nit. L'interface native Nit est faite sans aucune validation d'appel et la fonction native est exécutée dans le même espace mémoire que le reste du programme. Pour ces deux raisons, la fonction native, lorsqu'exécutée, pourrait accéder à n'importe quoi dans l'espace mémoire de l'application, donc modifier des attributs non-nullables et y insérer la valeur *null*.

Une validation manuelle de toutes les fonctions natives dans le compilateur et dans la librairie standard a été faite pour ainsi s'assurer que les fonctions déjà présentes se comportent correctement. De plus, un autre membre de mon équipe de recherche travaille sur l'interface native de Nit et devrait trouver une solution pour s'assurer de plus de robustesse lors d'appels natifs (Laferrière, 2012).

8.3.2 Limitations causées par les systèmes d'exceptions

Cette section présente les limitations qui existent dans plusieurs approches de types nullables en ce qui concerne les systèmes de gestions d'exceptions.

```
package com.test;

public class MyClass extends RuntimeException
{
    public int x;
    MyClass(){
        throw this;
    }
}

public class App
{
    public static void main( String[] args )
    {
        try {
            new MyClass();
        }catch (MyClass a) {
            System.out.println (a.x);
        }
    }
}
```

FIGURE 8.8: Échappement d'une instance en construction par exception en Java

```
#include <iostream>

class SomeClass{
    public:
    int x;
    SomeClass::SomeClass() {
        throw this;
    }
};

int main(){
    try{
        new SomeClass();
    } catch (SomeClass* s) {
        std::cout << s->x;
        delete s;
    }

    return 0;
}
```

FIGURE 8.9: Échappement d'une instance en construction par exception en C++

Dans plusieurs systèmes de gestion d'exception tels que Java (figure 8.8) et C++ (figure 8.9), une exception peut être lancée lors de la construction d'un objet. En temps normal, l'objet en cours de création ne sera pas accessible dans le code appelant lorsqu'une exception est lancée puisque l'affectation vers la variable cible n'aura jamais lieu. Par contre, il est possible de *lancer* l'objet qui est en construction en tant qu'exception ou, plus simplement, de l'ajouter à une liste (en paramètre au constructeur) ou de l'affecter à une variable *globale*. En ce faisant, la référence de l'objet partiellement construit est accessible et rien ne garantit que les champs de l'instance sont bien initialisés.

Si notre approche de types nullables avait été implémentée dans un tel langage, nous aurions dû valider la bonne initialisation des attributs lors de toutes sorties du constructeur : sortie *normale* à la fin du constructeur, mais aussi lors d'exceptions, ce qui peut être plus complexe puisqu'elles peuvent être soulevées à tous points du constructeur.

Nit n'a pas encore de système de gestion d'exceptions[3], mais certaines idées ont été discutées pour ajouter un système de gestion d'exceptions qui sera compatible avec notre modèle de types nullables.

8.4 Conclusion

En premier lieu, ce chapitre a présenté les résultats de l'utilisation de notre approche sur le compilateur Nit. Plus précisément, ces nombres montrent que dans un programme complexe (un compilateur de plus de 110.000 lignes de code), tous les tests qui furent ajoutés par notre approche de types nullables furent supprimés par nos analyses statiques.

Par la suite, deux points où l'implémentation effectuée et la théorie divergent sont présentés : (1) l'évaluation paresseuse des méthodes ainsi que (2) l'inlining des

3. Présentement, le système implémenté arrête l'exécution du logiciel lorsqu'une exception est lancée

accesseurs automatiques. Ces deux divergences ne sont que des optimisation de vitesse d'exécution du compilateur et ne changent en rien les résultats obtenus.

Finalement, deux limitations de l'approche sont présentées : (1) la gestion des appels natifs ainsi que (2) les systèmes d'exceptions. Bien que ces deux limitations pourraient sembler très importantes, nous avons montré qu'elles ne sont pas applicables à notre situation.

CHAPITRE IX

TRAVAUX CONNEXES

Les références nulles ont toujours causé des problèmes dans les langages de programmation supportant les références ou les pointeurs. Ainsi, depuis l'avènement de la programmation structurée, des outils et des approches, dont (Chalin et James, 2007; Evans, 1994; Evans, 1996; Rutar, Almazan et Foster, 2004; Hubert, 2008; Hovemeyer, Spacco et Pugh, 2006; Hovemeyer et Pugh, 2007; Male et al., 2008; Evans et al., 1994), ont été développés pour détecter et prévenir ce genre d'erreurs.

Ce chapitre présente en premier lieu une approche avec des types dits *nullables*, suivi d'une approche avec des types dits *raw* et ensuite d'une approche des types dits *delayed*. Finalement, quelques ouvrages en lien avec nos approches, analyses ou optimisations sont présentés.

9.1 Types nullables

Cette section présente une méthode de détection et de prévention de déréférencement de nul qui existe depuis longtemps et qui a été créée pour être utilisée dans des langages non-orientés objets. Elle consiste à définir de façon explicite les endroits (variables, paramètres, champs, ...) où la valeur `null` est acceptée. Cette approche a été implémentée de diverses façons à travers les langages, mais les choix les plus courants pour les annotations de conteneurs sont :

- `not nullable`

```
#include <stdio.h>
struct _value {
       int value;
};
struct _object {
       struct _value* first;
       /**nullable**/struct _value* second;
};
struct _object o;
printf("%i", o.first->value); /*Boom!*/
```

FIGURE 9.1: Problème de création d'instance avec types nullables dans une approche où les références nullables sont déclarées.

Un conteneur qui est annoté comme étant `not nullable` ne peut jamais contenir la valeur `null`. Deux types de vérifications sont faites pour garantir ceci :

1. valider qu'il n'y a pas d'affectation directe à partir de `null` ;
2. valider qu'il n'y a pas d'affectation à partir d'un conteneur définie comme étant `nullable` ou n'ayant pas d'information.

- `nullable`

  Un conteneur qui est annoté comme étant `nullable` peut contenir la valeur `null`. Le compilateur ou l'outil peut, avec cette information, avertir le programmeur lorsqu'il essaie de déréférencer un pointeur de ce type ou lorsqu'il essaie de l'affecter dans un endroit qui est défini comme étant `non nullable`.

- `aucune information`

  Selon les implémentations, un conteneur qui n'a aucune information peut être soit : ignoré, considéré comme étant `nullable` ou considéré comme étant `non nullable`. Chalin, P. et James, P.R. (Chalin et James, 2007) ont montré qu'il est préférable, dans les langages orientés-objet, de considérer qu'aucune annotation implique un conteneur `non nullable`.

Ces types d'annotations sont parfaits pour les langages procéduraux où les structures complexes sont absentes. Lorsque l'on commence à regarder les structures (`struct`

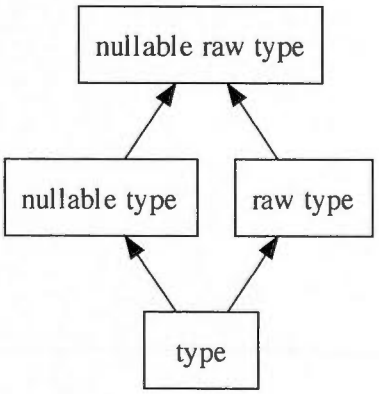

FIGURE 9.2: Relation de type entre types sans annotation, types `nullable`, types `raw` et types `nullable raw`

en *C*) ou les classes, ce procédé devient plus complexe. On voit dans la figure 9.1 le genre de problème qui peut arriver avec l'utilisation de types nullables dans les langages avec structures complexes. On y voit que l'attribut d'une structure, ici la structure `object`, n'est pas validé lors de l'instantiation de celle-ci. Dans l'exemple présenté, on voit que l'accès à l'attribut `value` de l'attribut `first` de la variable `o` causerait un arrêt brutal de l'exécution du programme et que le compilateur, ou l'outil de validation, n'avertirait pas le programmeur.

Nous considérons que trop de cas ne sont pas couverts correctement par les types nullables, tels que présentés dans cette section, et considérons que l'ajout des validations additionnelles que nous proposons valent les ajouts de tests dynamiques.

9.2 Types *raw*

Cette section présente les types `raw`. Le principe derrière les types `raw` (Fähndrich et Leino, 2003) est d'indiquer au système de type les références qui pourraient

```
class A
	var attr: Object

	init do
		print(attr.to_s) # Boom dynamiquement
	end
end
var v: A = new A
```

FIGURE 9.3: Problème de création d'instance avec types nullables

```
class A
	var attr: Int

	init do
		print(attr.to_s)
		#	^^^^ erreur lors de la compilation, attr possiblement null
	end
end
var v: A = new A
```

FIGURE 9.4: Problème de création d'instance avec types nullables, solution avec les types raw

```
class Node
	var next: raw Node
	var previous: raw Node

	var content: Int

	init with_node(other: raw Node, content: Int) do
		self._next = node
		self._previous = node
		self._content = content
	end

	init(content: Int, content2: Int) do
		var other = new Node.with_node(self, content2)
		self._next = other
		self._previous = other
		self._content = content
	end
end
var n: Node = new Node(1, 2)
var second: raw Node = n._next
var content: nullable Int = second.content
if content != null then print "content: {content}"
```

FIGURE 9.5: Structure circulaire, exprimée avec des types raw types

contenir des objets en cours d'initialisation. Le procédé consiste à ajouter une nouvelle annotation à l'approche des types nullables présentée dans la section précédente : l'annotation de type `raw`. Une référence dont le type statique est annoté avec cette annotation est déclaré comme pouvant contenir un objet partiellement initialisé. De par la nature des constructeurs, l'instance courante y est implicitement considérée comme étant `raw`. L'utilisation de l'annotation sur une méthode indique que celle-ci peut être appelée même si l'instance utilisée pour ce faire est définie comme étant `raw`. Plusieurs restrictions s'appliquent aux conteneurs `raw` :

- restriction d'affectation ;
- restriction d'appel de méthode ;
- restriction d'accès aux attributs.

La restriction d'affectation est simple : un conteneur `raw` ne peut être affecté qu'à un autre conteneur `raw`. Cette restriction est en place pour assurer que les instances en construction ne s'échappent pas dans des références qui ne garantissent pas leur bon comportement.

La restriction d'appel de méthode se fait lors d'appels sur une instance déclarée comme étant `raw` : les seules méthodes qui peuvent être appelées sont celles définies explicitement comme étant `raw`.

La restriction d'accès aux attributs est la partie centrale de cette approche : l'accès en lecture à un attribut d'une référence définie comme étant `raw` retourne statiquement un objet de type `nullable`. Ceci est dû au fait que l'instance à laquelle on accède est potentiellement en cours d'initialisation, donc l'attribut accédé est possiblement non initialisé.

Une méthode définie comme étant `raw` implique que l'instance courante (`self`) est `raw`. Toutes les restrictions s'appliquent donc aux appels de méthode, accès aux attributs et affectations faits à partir de cette instance, qu'ils soient explicite ou non.

Bien qu'une référence soit identifiée comme étant `raw`, celle-ci doit tout de même contenir une valeur. Pour permettre à la référence de contenir la valeur `null`, elle doit de

plus être annotée comme étant `nullable`. Les règles d'affectations deviennent alors plus complexes. La figure 9.2 présente la hiérarchie des types normaux, `nullable` et `raw`. Les règles d'affectations y sont représentées du bas vers le haut, ainsi, une référence définie de façon simple, sans annotation, peut être affectée à une référence annotée comme étant `nullable` ou à une référence annotée `raw`. Ensuite, ces deux références peuvent être affectées à une référence `nullable raw`. Cette propriété est transitive, donc une référence sans annotation peut être affectée directement à une autre déclarée comme étant `nullable raw` sans avoir besoin de passer dans les états intermédiaires.

L'exemple 9.3 présente une situation problématique sans la technique des types `raw`. On peut y voir que l'accès à l'attribut `attr` ne va pas causer de problème de compilation. Or, lors de l'exécution, l'exécution de la ligne `print(_attr.to_s)` va causer une erreur de déréférencement de nul.

L'exemple 9.4 montre l'exemple 9.3 mais utilisant la technique de types `raw`. On voit que le système de type va indiquer le déréférencement possible de la valeur non-initialisée.

Les types `raw` ne résolvent pas tous les problèmes. On voit dans la figure 9.5 un des problèmes résultant des types `raw` : le fait que même une fois complètement initialisé, une référence qui était définie comme étant `raw` reste `raw`. Donc tout appel plus tard dans le logiciel devient plus complexe, puisqu'il faudrait un test dynamique pour *prouver* que l'instance est bien initialisée.

Dans certains cas, cette technique demande à ce que des méthodes soient dupliquées : plus précisément, certaines méthodes qui doivent être accessibles autant au moment où l'instance est définie comme *raw* et lorsqu'elle est complètement initialisée. Nous considérons que cette duplication de méthode n'est pas acceptable et que l'ajout de nos tests dynamiques valent la peine.

```
class Node
       var next: Node in delay t
       var previous: Node in delay t

       var content: Int

       init with_node(other: Node in delay t, content: Int)
                     in delay t do
              self.next = node
              self.previous = node
              self.content = content
       end

       init(content: Int, content2: Int) in delay t do
              var other = new Node.with_node(self, content2)
                     in delay t
              self.next = other
              self.previous = other
              self.content = content
       end
end
var n: Node = new Node(1, 2)
var second: Node = n._next
var content: Int = second.content
print "content:_{content}"
```

FIGURE 9.6: Structure circulaire exprimée avec les types *delayed*. Cet exemple utilise une syntaxe adaptée de celle présentée par les auteurs de l'approche pour être plus cohérente avec le reste de notre document.

```
class SomeObject
       var content: Int
       var content_2: Int

       init(content: Int) in delay t do
              self.content = content
       end
end

var n: Node;
in delay x do
       n = new Node(1) in delay x
       n.content_2 = 2
       if n.content != null then
              print "content: {n._content}"
       end
       if n.content_2 != null then
              print "content2: {n.content_2}"
       end
end
print "content: {n._content}"
print "content2: {n._content_2}"
```

FIGURE 9.7: Exemple complexe d'utilisation des types *delayed*. Cet exemple utilise une syntaxe adaptée de celle présentée par les auteurs de l'approche pour être plus cohérente avec le reste de notre document.

9.3 Types *delayed*

Cette section présente une troisième approche (Fähndrich et Xia, 2007) pour la détection et la gestion des déréférencements de nul. Cette approche est basée sur une notion de temporalité : le programmeur peut indiquer au système de type une zone temporelle durant laquelle une ou plusieurs instances peuvent ne pas être complètement construites. Durant ce délai, certaines propriétés des instances attachées à ce délai sont invalidées par le système de type.

Cette approche est basée sur l'approche de types *nullables* ainsi que celle des types *raw* présentées plus tôt dans ce chapitre et vise a régler le problème des structures circulaires laissé ouvert par les types *raw*. On voit dans les figures 9.6 et 9.7 des exemples d'utilisation des types *delayed*. Ces figures présentent les deux types d'annotations : les annotation de types *nullable* et les annotations plus complexes représentées par l'expression *in delay*, suivi d'un identifiant.

L'annotation *nullable* a, dans cette approche, le même effet que dans les approches présentées précédemment : une référence identifiée comme étant *nullable* peut contenir la valeur *null*. Ainsi, le système de type peut détecter tout appel de méthode et affectation relatif à une référence *nullable*.

C'est l'annotation *delayed* qui encapsule toute la magie de cette approche. Elle représente une zone dans le temps durant laquelle certaines propriétés des instances situées dans cette zone sont invalidées par le système de type. On peut retrouver celle-ci associée à une référence à plusieurs endroits dans le code source :

- après une déclaration de fonction [1] ;
- après une déclaration de variable ;
- après une déclaration d'attribut.

Une référence qui est identifiée comme étant *delayed* est potentiellement partiel-

1. Lorsque le délai est attaché à une fonction, il est en fait attaché à l'instance représentée par *self*.

lement initialisée. Le fait qu'une référence soit *delayed* ne permet pas l'affectation à partir d'une référence *nullable*, ni à partir de la valeur *null*. Ainsi un appel de méthode sur une instance *delayed* ne pourrait pas causer de problème de déréférencement de nul lors de l'exécution du programme. Les restrictions appliquées sur une instance *delayed* sont les suivantes :

- tous les attributs sont considérés comme étant *nullables* ;
- les seules méthodes qui peuvent être appelées sur l'instance sont celles qui sont définies comme étant *delayed*.

En plus de l'annotation associée à une référence, cette approche permet deux autres constructions :

- un délai associé à un appel de méthode ;
- un délai associé à la création d'un bloc de code.

Ces deux constructions permettent de lier les délais ensemble et ainsi de représenter des structures complexes où le constructeur d'une instance n'initialise pas toute l'instance, par exemple une structure récursive ou un objet qui doit être initialisé en plusieurs phases. Par défaut, lorsqu'une méthode définie comme étant *delayed* est appelée, une nouvelle zone temporelle est créée pour être associée à l'instance en cours de création. Si l'appel avait été effectué en précisant une zone temporelle déjà existante, alors l'analyse aurait été effectuée liée avec cette zone existante.

Nous jugeons que la syntaxe utilisée par cette approche ainsi que la complexité de celle-ci la rendent peu utilisable pour un programme complexe réel.

9.4 Autres travaux

Une technique est présentée dans (Bond et al., 2007) pour permettre de trouver la source de problème lorsqu'une exception est lancée en raison d'un déréférencement de nul dans les langages tels que *Java*, *C#*, *C* et *C++*. Elle consiste à utiliser un ensemble de valeurs pour représenter la valeur *null* plutôt que le traditionnel 0. Par exemple, une des techniques qui y est présentée est celle d'utiliser les adresses dans la portée

0x00000000-0x07ffffff [2], pour représenter la valeur *null*. Les derniers bits de l'adresse sont utilisés pour indiquer de quel fichier et de quelle ligne la valeur est originaire. L'implémentation que les auteurs ont fait en Java est assez rapide pour être utilisée dans un environnement de production puisqu'elle n'ajoute que 4% de temps d'exécution et l'implémentation qu'ils ont fait avec Memcheck [3] ne ralentit en rien l'exécution de cet outil et permet de trouver les informations d'origine dans 72% des cas en moyenne.

Dans (Loginov et al., 2008), les auteurs présentent une technique pour valider les déréférencement de nul reposant sur une analyse adaptative : lorsque l'analyse intraprocédurale ne suffit pas pour garantir la validité du déréférencement, elle accède aux informations contenues dans un certain nombre de méthodes environnantes dans le graphe d'appel pour tenter de valider le déréférencement. Bien que l'approche soit automatique dans la majorité des cas, elle permet tout de même au programmeur d'ajouter des annotations pour avoir de meilleurs résultats. L'approche permet d'avoir des résultats qui sont presque équivalents à des analyses de programmes complets, sans toutefois nécessiter les même ressources. Des tests ont été effectués sur un grand nombre de logiciels Java et dans 90% des cas, la technique automatique a réussi à garantir les déréférencements.

Le langage *C#* de Microsoft (Hejlsberg, Wiltamuth et Golde, 2006) permet depuis la version 2.0 d'indiquer qu'une référence, dont le type statique est une structure, peut aussi contenir la valeur *null* en utilisant l'annotation de type *?* [4], tel que dans la déclaration : `int? i = 10;`. La valeur contenue dans une référence dont le type est défini comme étant nullable ne peut pas être accédée directement, le programmeur doit passer par la méthode `Value` pour accéder à l'object encapsulé. Le programmeur peut aussi utiliser la méthode `HasValue` pour savoir si une instance est bien encapsulée. De plus, le

---

2. Ces adresses supposent une architecture 32 bits

3. Site web : http ://valgrind.org/

4. En fait, cette annotation est un raccourci pour la forme plus longue `System.Nullable<int>`.

système de typage statique va garantir qu'il n'existe pas d'affectation entre une référence nullable et une référence non-nullable. Cette approche est très similaire à celle présentée plus tôt des annotations *nullable*.

CHAPITRE X

CONCLUSION

Nous décomposons le problème de la gestion de déréférencement de nul en trois dimensions : (1) la simplicité, (2) l'exactitude des résultats et (3) le typage statique. Tel que montré dans le chapitre 9, à notre connaissance, aucune approche couvre les trois dimensions.

Ce document a présenté une nouvelle approche de gestion de déréférencement de nul. Celle-ci consiste en une politique de typage statique augmentée de tests dynamiques ajoutés automatiquement par le compilateur. Ces deux composantes apportent des garanties directes dans toute fonction non atteignable par un constructeur.

Ces deux premières composantes attaquent les dimensions de simplicité et d'exactitude. Par la suite, une optimisation a été présentée pour supprimer les tests dynamiques ajoutés par le compilateur pour un programme de plus de 100.000 lignes de code. Nous avons montré, dans le chapitre 8, que cette optimisation est assez précise pour supprimer tous les tests dynamiques ajoutés par le compilateur. Dans un tel cas, le compilateur garantit de façon statique qu'aucune erreur liée à la gestion de déréférencement de nul ne peut être trouvée lors de l'exécution du programme analysé. Ainsi, la troisième dimension est gérée par cette optimisation.

Cette optimisation repose sur trois analyses statiques interprocédurales. Celles-ci se basent sur une nouvelle technique pour effectuer des analyses statiques inter-

procédurales. La technique utilise une duplication des informations des méthodes analysées pour chacun des contextes d'appels.

Suite à notre analyse de la littérature en analyses statiques, nous avons remarqué le manque d'information explicites en rapport aux preuves de validité des algorithmes communément utilisés. Nous avons donc présenté, dans le quatrième chapitre des preuves de validité des analyses statiques intraprocédurales.

En plus des résultats expérimentaux, le septième chapitre présente certaines limitation de l'approche. En fait, les deux limitations qui y sont présentées ((1) les appels natifs et (2) le système d'exception) ne sont pas de vraies limitations : le langage utilisé pour implémenter notre approche possède des caractéristiques qui font en sorte que ces deux limitations ne sont pas valides.

ANNEXE A

APERÇU DU LANGAGE NIT

Nit est un langage orienté-objet utilisé, jusqu'à ce jour, principalement à des fins de recherche au sein du groupe de recherche Grésil (`http://gresil.org/`). Il présente, entre autre, les caractéristiques suivantes :

- typage statique avec inférence de type légère ;
- types nullables ;
- généricité ;
- héritage multiple ;
- types virtuels ;
- classes ouvertes ;
- et plusieurs autres.

Cette annexe présente un aperçu du langage Nit en focussant sur les aspects du langage qui sont utilisés dans ce document. Pour plus d'information sur le langage Nit, visitez `http://nitlanguage.org/`.

Plus précisément, cette annexe présente en premier lieu la structure générale d'un programme Nit. Par la suite, la structure d'un fichier source Nit sera présentée. La troisième section présente la structure d'une classe Nit et la section suivante présente la structure d'un bloc de code Nit. La dernière section présente quelques classes et méthodes qui font partie de la bibliothèque standard du langage et qui sont utilisées dans ce document.

A.1 Structure d'un programme Nit

Un programme Nit est composé d'un ou plusieurs fichiers source. En Nit, un fichier est aussi appelé module. Chacun des module peut importer d'autres modules, mais il ne doit pas exister de cycles dans la hiérarchie d'importation de modules. On peut donc voir un programme Nit comme étant une arborescence de modules, où la racine est le programme principal (indiqué sur la ligne de commande lors de la compilation) et chacun des liens représente une inclusion de module.

```
1  module principal
2
3  import bibliotheque
4
5  class UserFactory
6          fun create_user(name: String): User do
7                  return new User(name)
8          end
9  end
10
11 fun print_user(user: User) do
12         print "User:"
13         print "\tName: {user.name}"
14 end
15
16 var factory: UserFactory = new UserFactory
17 var user: User = factory.create_user("John Doe")
18 print_user user
```

FIGURE A.1: Exemple de programme Nit, module *principal*

Les figures A.1 et A.2 présentent un exemple de programme contenant deux modules. On y voit le premier module nommé *bibliotheque* dans la figure A.2. Celui-ci contient la définition de la classe *User*. Le second module, nommé *principal* et présenté dans la figure A.1, contient la définition d'une seconde classe, nommée *UserFactory* ainsi que le point d'entrée du programme.

```
1  module bibliotheque
2
3  class User
4          var _name: String
5
6          init (name: String) do
7                  _name = name
8          end
9
10         fun name: String = _name
11 end
```

FIGURE A.2: Exemple de programme Nit, module *bibliothèque*

A.2 Structure d'un fichier source Nit

Cette section présente la structure d'un fichier source Nit. En premier lieu, l'entête de fichier, la définition du nom du module ainsi que l'inclusion de modules, est expliquée. La seconde sous-section présente la définition d'une classe, la troisième présente les fonctions *top-level* et la dernière explique ce à quoi le code inséré directement à la *racine* du document sert.

Entête de fichier

L'entête d'un fichier de source Nit est composé de la déclaration du module (optionnel), suivi des déclarations d'inclusion de modules (aussi optionnel). Si la déclaration du module est présente, elle doit prendre la forme `module NomModule` où *NomModule* doit être le même que le nom du fichier. Dans les exemples A.1 et A.2, les déclarations de noms de module sont la première ligne des fichiers.

L'entête est aussi composée de zéro, une ou plusieurs inclusions de modules. Il existe plusieurs formats pour l'inclusion de module, mais seul le plus simple est présenté ici. Il prends la forme d'une ligne ayant le format `import NomModule`. Dans l'exemple A.1, une seule inclusion est effectuée : celle du module nommé *bibliotheque*.

Définition d'une classe

Les lignes 5 à 9 de la figure A.1 présentent la définition de la classe `UserFactory`. La structure exacte d'une déclaration de classe est présentée dans la section A.3.

Fonctions *top-level*

Les lignes 11 à 14 de la figure A.1 présentent la définition de la fonction *top-level* nommée *print_user*. Ces fonctions sont en quelque sorte des fonctions statiques non attachées à des classes[1]. Ainsi, elles peuvent être appelées directement dans n'importe bloc de code qui les inclus.

Code inséré à même la *racine* du fichier

Dans le cas où le programme compilé ne contient qu'un seul fichier, ce qui est le cas de la majorité des exemples de ce document, les lignes de code insérées à même la *racine* du fichier, telles que les lignes 16 à 18 de l'exemple A.1, sont le point d'entrée du logiciel. Dans les langages tels que *C*, *C++* et *Java*, ces lignes seraient le contenu de la fonction `main`.

Dans le cas où un programme contient plus d'un module, le code inséré à la racine du module principal (celui utilisé en entrée au compilateur) est utilisé comme point d'entrée du logiciel.

A.3 Déclaration d'une classe Nit

La définition d'une classe Nit est composée de quatre parties : (1) les liens d'héritages, (2) les attributs, (3) les méthodes et (4) les constructeurs. Dans cette section, chacune de ces parties sera présentée.

---

1. La réalité est bien plus complexe, mais n'est pas nécessaire à la bonne compréhension de ce mémoire.

```
1  module data
2
3  class Person
4          readable var _name: String
5
6          fun greet do
7                  print "Hi {name}"
8          end
9  end
10
11 class Employee
12         super Person
13
14         readable var _salary: Int
15
16         init with_salary(name: String, salary: Int) do
17                 init(name)
18                 _salary = salary
19         end
20
21         fun salary=(value: Int) do
22                 if _salary < value then
23                         print "Yeah, more $$"
24                         _salary = value
25                 else
26                         print "I don't think so ..."
27                 end
28         end
29 end
```

FIGURE A.3: Exemple de définition de classe en Nit, module *data*

Déclaration d'un lien d'héritage

Les figures A.3 et A.4 présentent 3 classes : *Person*, *Employee* et *Boss*. Les liens hiérarchiques entre ces classes forment une ligne : *Person* est la super-classe directe de *Employee* tandis que *Employee* est la super-classe directe de *Boss*.

Les liens d'héritages sont définis par le mot-clé *super* au début de la déclaration des classes. le nombre de liens d'héritage n'est pas limité. Ainsi, une classe peut avoir

```
1  module program
2
3  import data
4
5  class Boss
6          super Employee
7
8          init (name: String, salary: Int) do
9                  with_salary(name, salary)
10         end
11
12         fun give_raise (employee: Employee) do
13                 print "Humm_..._let's_see_..."
14                 if employee.name == name then
15                         print "I_needed_a_new_boat_!"
16                         employee.salary = employee.salary * 10
17                 else
18                         print "Not_this_year_..."
19                 end
20         end
21 end
22
23 var an_employee: Employee = new Employee.with_salary("Barry_D'Alive", 1000)
24 var a_boss: Boss = new Boss("Al_Gore-Rythim", 10000)
25 an_employee.greet
26 a_boss.give_raise(an_employee)
27 a_boss.give_raise a_boss
```

FIGURE A.4: Exemple de définition de classe en Nit, module *program*

autant de parents qu'elle le souhaite. La seule restriction qui existe pour l'ajout d'un parent est qu'il ne faut pas que cet ajout crée un cycle dans la hiérarchie d'héritage.

Déclaration d'un attribut

Plusieurs attributs sont déclarés dans les classes des figures A.3 et A.4. Une déclaration d'attribut se fait avec le mot-clé *var* dans la déclaration de la classe. Par exemple, la classe *Person* déclare un attribut nommé *name* de type *String*.

Le langage Nit utilise un concept d'encapsulation d'attributs restreignant l'accès

direct aux attributs à l'instance à laquelle appartiennent ces attributs. Pour diminuer le code nécessaire, plus précisément l'écriture manuelle et répétitive d'accesseurs en lecture ou écriture (*getter* et *setter*), le langage propose au programmeur d'utiliser deux annotations lors de la déclaration de ses attributs : *readable* et *writable*. Ainsi, il existe quatre façons de déclarer un attribut dans le code :

1. `var _myAttr: Int;`
2. `readable var _myAttr: Int;`
3. `writable var _myAttr: Int;`
4. `readable writable var _myAttr: Int.`

La première façon déclare l'attribut, mais ne le rend pas accessible aux autres classes. On pourrait voir cet attribut comme étant *private* en Java. La seconde introduit l'attribut avec un accesseur en lecture (*getter*) automatique. Celui-ci aura la signature : `fun myAttr: Int` et pourra être exécuté par les autres classes. La troisième déclaration créera automatiquement un accesseur en écriture (*setter*), mais pas d'accesseur en lecture. La dernière déclaration créera les deux accesseurs présentés précédemment.

La signature de l'accesseur en écriture est un peu complexe. Dans l'exemple de la figure A.3, on voit un tel accesseur déclaré manuellement à la ligne 21. Cette fonction définie l'opérateur = sur l'attribut *salary*. Ainsi, tout appel de l'opérateur = sur l'attribut *salary* d'une instance de la classe *Employee* appellera cette fonction. Par exemple, la ligne 16 de la figure A.4 appellent cette fonction.

Déclaration d'une méthode

Il existe plusieurs façon de déclarer une méthode en Nit. Quelques-unes sont présentées dans les figures A.3 et A.4 et expliquées ici.

La classe *Person* déclare une méthode nommée *greet* aux lignes 6 à 8. Cette méthode ne prend aucun argument et n'a pas de valeur de retour.

La classe *Employee* déclare une méthode nommée *salary=* aux lignes 21 à 28.

Celle-ci définie en fait un accesseur en écriture pour l'attribut *salary*. La méthode reçoit un paramètre de type *Int* et ne retourne pas de valeurs.

La classe *Boss* déclare une méthode nommée *give_raise*. Celle-ci reçoit un attribut de type *Employee* et ne retourne pas de valeurs.

Déclaration d'un constructeur

En Nit, les constructeurs sont déclarés avec le mot clé *init*. Les figures A.3 et A.4 présentent plusieurs constructeurs.

La classe *Boss* déclare un constructeur sans nom. Un constructeur sans nom est le constructeur qui est appelé lors de l'exécution de l'instruction `new` lorsque la le nom de la classe n'est pas suivi par un nom. Par exemple, la ligne 24 de la figure A.4 appelle ce constructeur sans nom. La seule instruction située dans le constructeur, la ligne 9 de la figure A.4, est l'appel vers le constructeur de la super-classe de *Boss*.

La classe *Employee* déclare un constructeur nommé *with_salary*. Ce constructeur appelle le constructeur de la classe parent à la ligne 17 de la figure A.3. Pour appeler directement un constructeur nommé le programmeur doit indiquer le nom du constructeur à l'instruction `new`, un exemple est présenté à la ligne 23 de la figure A.4.

Quand une classe ne déclare pas de constructeur, tel que la classe *Person*, le compilateur crée un constructeur par défaut. Celui-ci va recevoir un argument pour chacun des attributs de la classe et initialisera l'attribut avec l'argument. Ainsi, dans le cas de la classe *Person*, le constructeur créé automatiquement reçoit un seul argument de type *String*.

A.4 Structure d'un bloc de code Nit

Nit prends racine dans plusieurs langages. Les opérateurs Nit ont le même comportement que ceux en *C++*, ainsi une assignation est effectuée avec un symbole d'égalité (=), tandis qu'une comparaison est faite avec deux symboles d'égalité (==).

La syntaxe est un peu plus calquée aux langages tels que *Pascal*, où le programmeur utilise des mots clés tels que `begin`, `end`, `then` et `else` plutôt que des accolades ({ et }) comme délimiteur de blocs de code.

Cette section présente quelques points importants des blocs de code Nit. Plus précisément, la première sous-section montre comment une variable est déclarée en Nit. La seconde présente un aperçu d'une structure de contrôle conditionnelle et la troisième montre quelques points importants des appels de méthode.

Déclaration de variable

Les lignes 23 et 24 de la figure A.4 montrent deux exemples de déclaration de variables. Dans le premier cas, la variable est de type `Employee` et dans le second elle est de type `Boss`. Dans beaucoup de cas, le compilateur Nit va déduire automatiquement le type des variables locales. Ainsi, la ligne 23 aurait pu être écrite : `var an_employee = new Employee.with_salary("Barry_D'Alive", 1000)`. Pour des soucis de clarté et pour être certain que tous les lecteurs de ce document comprennent bien les types des variables locales utilisées, la version longue des déclarations est utilisée.

Structure de contrôle conditionnelle

Les lignes 14 à 19 de la figure A.4 montrent un exemple de structure conditionnelle (`if`). On y voit que la syntaxe diffère de celle de *C++* et les langages qui s'y rapprochent.

Appel de méthode

Un peu partout à travers l'exemple de la figure A.4 se trouvent des appels de méthode. Notamment, notons la petite différence entre l'appel de la ligne 26 et celui de la ligne 27 : celui de la ligne 26 utilise des parenthèse tandis que celui de la ligne 27 n'en a pas. En fait, ces deux syntaxes d'appel sont totalement équivalentes lorsqu'il n'y a pas d'ambiguïté.

A.5 Quelques méthodes utiles en Nit

Cette section présente quelques méthodes Nit qui sont utilisées dans ce document. Celles-ci font partie de la bibliothèque standard de Nit et sont donc disponibles à tout programme Nit.

to_s

La méthode `to_s` retourne une représentation sous forme de chaîne de caractères. Ainsi, `1.to_s` retournera la chaîne de caractères `"1"`. Cette méthode est particulièrement intéressante lorsqu'utilisée avec les '*super-strings*'. Ce que nous appelons '*super-strings*' en Nit est le fait que les chaînes de caractères contenant certains caractères seront automatiquement étendues lors de l'exécution. Les caractères spéciaux utilisés sont simplement les accolades ({ et }). Plus précisément, une chaîne de caractères contenant une accolade ouvrante, suivie d'une instruction Nit, puis d'une accolade fermante contiendra en remplacement des accolades le résultat de l'exécution de la méthode `to_s` sur le résultat l'instruction. Un exemple de ceci est présent dans la figure A.3 à la ligne 7. Lors de l'exécution de cette instruction, le programme affichera la chaîne de caractère `Hi ` , suivi du résultat de l'appel de la fonction `to_s` sur le résultat de l'appel de la fonction `name` (qui est un accesseur en lecture vers l'attribut *name*).

rand

La méthode *rand* permet de retourner un nombre aléatoire. Son utilisation est un peu difficile à comprendre puisqu'il n'existe pas de méthode statique en Nit. Dans un langage où il existe des méthodes statiques, les fonctions pour avoir un nombre aléatoire sont souvent accessible à partir de la classe voulue directement. Ainsi, dans un tel langage, le programmeur écrirait `Int.rand` ou `Float.rand`. En Nit, le programmeur doit lancer l'appel de méthode sur une instance du type voulu, donc `1.rand` pour avoir un entier de façon aléatoire ou `1.0.rand` pour avoir un nombre réel de façon aléatoire.

print

Cette fonction permet d'afficher une chaîne de caractère sur la sortie standard. Par exemple, `print "Hello_World!"` affiche la chaîne `"Hello_World!` sur la sortie standard.

BIBLIOGRAPHIE

Alfred, V., R. Sethi, et D. Jeffrey. 1986. *Compilers : Principles, Techniques and Tools.* Addison-wesley.

Allen, F. E. 1970. « Control flow analysis », *ACM SIGPLAN Notices*, vol. 5, p. 1–19.

Appel, A. et M. Ginsburg. 1998. *Modern Compiler Implementation in C.* Cambridge University Press.

Bacon, D. 1997. « Fast and Effective Optimization of Statically Typed Object-Oriented Languages ». Thèse de Doctorat, University of California, Berkeley.

Bacon, D. F. et P. F. Sweeney. 1996. « Fast static analysis of c++ virtual function calls ». In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, p. 324–341, New York, NY, USA. ACM.

Birkhoff, G. 1967. *Lattice Theory.* American Mathematical Society.

Birkhoff, G. 1995. *Lattice Theory.* American Mathematical Society.

Blyth, T. 2005. *Lattices and ordered algebraic structures.* Springer Verlag.

Bond, M., N. Nethercote, S. Kent, S. Guyer, et K. McKinley. 2007. « Tracking bad apples : reporting the origin of null and undefined value errors ». In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, p. 405 – 422. ACM.

Carré, B. et O. Zendra, éditeurs. 2009. *Actes des journées Langages et Modèles à Objets.* T. RNTI-L-3, série *Revue des Nouvelles Technologies de l'Information*, Nancy. Cépaduès-Editions.

Chalin, P. et P. James. 2007. « Non-null references by default in Java : Alleviating the nullity annotation burden ». In *Proceedings of the 21st European Conference on Object-Oriented Programming.* T. 4609, p. 227–247. Springer.

Cooper, K., T. Harvey, et T. Waterman. 2002. Building a control-flow graph from scheduled assembly code. Rapport, Rice University.

Davey, B. A. et H. A. Priestly. 1990. *Introduction to Lattices and Order.* Cambridge University Press.

Evans, D. 1994. Using specifications to check source code. Rapport, Massachusetts Institute of Technology Cambridge, MA, USA.

Evans, D. 1996. « Static detection of dynamic memory errors », *ACM SIGPLAN Notices*, vol. 31, no. 5, p. 44–53.

Evans, D., J. Guttag, J. Homing, et Y. Tan. 1994. « Lclint : A tool for using specifications to check code ». In *Foundations of Software Engineering : Proceedings of the 2 nd ACM SIGSOFT symposium on Foundations of software engineering : New Orleans, Louisiana, United States*, p. 87–96.

Fähndrich, M. et K. R. M. Leino. 2003. « Declaring and checking non-null types in an object-oriented language », *ACM SIGPLAN Notices*, vol. 38, no. 11, p. 302–312.

Fähndrich, M. et S. Xia. 2007. « Establishing object invariants with delayed types », *ACM SIGPLAN Notices*, vol. 42, no. 10, p. 337–350.

Freese, R. 2004. *Automated Lattice Drawing*. Coll. Eklund, P., éditeur, Coll. « *Concept Lattices* ». T. 2961, série *Lecture Notes in Computer Science*, p. 589–590. Springer Berlin / Heidelberg.

Fuchs, L. 1963. *Partially ordered algebraic systems*. T. 7. Pergamon Press.

Gélinas, J.-S., J. Privat, et E. Gagnon. 2009. « Prévention du déréférencement de références nulles dans un langage à objets ». In Carré, Bernard and Zendra, Olivier, éditeur, *Langages et Modèles à Objets*. T. RNTI-L-3, p. 5–16. Cépaduès-Editions.

Gratzer, G. 1978. *General lattice theory*. Academic Press.

Hejlsberg, A., S. Wiltamuth, et P. Golde. 2006. *The C# programming language*. Addison-Wesley Professional.

Hovemeyer, D. et W. Pugh. 2007. « Finding more null pointer bugs, but not too many ». In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, p. 9–14.

Hovemeyer, D., J. Spacco, et W. Pugh. 2006. « Evaluating and tuning a static analysis to find null pointer bugs », *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1, p. 13–19.

Hubert, L. 2008. « A Non-Null annotation inferencer for Java bytecode ». In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, p. 36–42. ACM.

Kaufmann, A. and Boulaye, G.G. 1978. *Théorie des treillis en vue des applications*. Masson.

Laferrière, A. 2012. « L'interface native de Nit, un langage de programmation à objets ». Mémoire de maîtrise, Université du Québec à Montréal.

Loginov, A., E. Yahav, S. Chandra, S. Fink, N. Rinetzky, et M. Nanda. 2008. « Verifying dereference safety via expanding-scope analysis ». In *Proceedings of the 2008*

international symposium on Software testing and analysis, p. 213–224. ACM.

Male, C., D. Pearce, A. Potanin, et C. Dymnikov. 2008. « Java bytecode verification for @NonNull types », *Lecture Notes in Computer Science*, vol. 4959, p. 229.

Muchnick, S. 1997. *Advanced compiler design and implementation*. Morgan Kaufmann.

Neggers, J. et H. Kim. 1998. *Basic posets*. World Scientific Pub Co Inc.

Privat, J. 2006. « De l'expressivité à l'efficacité, une approche modulaire des langages à objets — Le langage PRM et le compilateur prmc ». Thèse de Doctorat, LIRMM, Université Montpellier II.

Rutar, N., C. Almazan, et J. Foster. 2004. « A comparison of bug finding tools for Java ». In *Proc. 15th IEEE International Symposium on Software Reliability Engineering*, p. 245–256.

Shivers, O. 1988. « Control flow analysis in scheme ». In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, p. 164–174.

Singh, Y. 2006. *Mathematical foundation of computer science*. New Age International.

Stoll, R. 1979. *Set theory and logic*. Dover Pubns.

Sundaresan, V., L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, et C. Godin. 2000. « Practical virtual method call resolution for Java ». In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, p. 264–280.

Tarski, A. 1955. « A lattice-theoretical fixpoint theorem and its applications », *Pacific journal of Mathematics*, vol. 5, no. 2, p. 285–309.

Wilhelm, R., D. Maurer, et S. Wilson. 1995. *Compiler design*. T. 38. Addison-Wesley Cornwall, UK.