REDHA CHORFI

Abstraction et vérification de programmes informatiques

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en Informatique
pour l'obtention du grade de Maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

2008

©Redha Chorfi, 2008

Résumé

Les systèmes informatiques offrent une grande flexibilité aux usagers en leur permettant l'accès, notamment par le biais de réseaux de télécommunication ou de l'Internet, à un vaste éventail de services. Toutefois, certains de ces services sont soumis à de fortes contraintes de sécurité, comme le télépaiement qui est au cœur du commerce électronique. Ainsi, les fournisseurs et les utilisateurs expriment des besoins croissants, et antagonistes, en sécurité. Répondre à ces deux besoins simultanément est un enjeu technologique transversal à de nombreux domaines de l'informatique. L'objectif de ce travail est de développer un mécanisme permettant de garantir la sécurité des systèmes, en s'appuyant sur l'expérience établie dans le domaine de la sécurité et des méthodes formelles. Pour se faire, nous définissons un nouveau cadre de vérification des propriétés de sécurité d'un programme informatique par l'analyse des flots de données et de contrôle construits à partir du code de ce dernier. L'idée principale consiste à définir un modèle pouvant abstraire la trace d'événements et les dépendances de ressources engendrés au moment de l'exécution du programme, et pouvant être soumis à des algorithmes de vérification de modèle *(model-checking)* pour l'analyse de la sûreté du programme vis-à-vis d'une propriété.

Avant-propos

Ma maîtrise est le résultat de mon premier travail de recherche et n'aurait pu aboutir sans le soutien d'un certain nombre de personnes. Je souhaite donc commencer par les remercier chaleureusement.

En premier lieu, je remercie mon directeur de recherche, Béchir KTARI, pour son temps et sa patience. Nos discussions et nos partages d'idées me permettent d'envisager sereinement mon avenir dans le monde de la recherche.

Je tiens également à remercier Danny DUBÉ et Nadia TAWBI, qui ont accepté d'évaluer ce mémoire. Ils sont tous deux professeurs au département d'informatique et de génie logiciel de l'Université Laval à Québec, et plus spécifiquement dans le laboratoire Langages, Sémantiques et Méthodes Formelles (LSFM).

Durant mon séjour à Québec, j'ai eu le plaisir de rencontrer des personnes merveilleuses qui se reconnaîtront ; au sein du laboratoire LSFM, je pense plus particulièrement à ceux qui ont partagé mon quotidien. Je souhaite aussi remercier Lynda, Violaine et Rachel, pour leurs sourires et leur bonne humeur permanente. Je voudrai aussi remercier tous mes amis qui m'ont soutenu tout au long de ma maîtrise et qui ont su me motiver quand j'en avais le plus besoin.

Il me reste à dédier ce mémoire à ma famille qui a toujours été présente dans mon cœur, malgré la distance. Je les remercie de m'avoir toujours accordé leur confiance et de continuer à croire en moi.

Table des matières

Résumé ii

Avant-propos iii

Table des matières iv

Table des figures vi

Liste des tableaux vii

Introduction 1

I État de l'art 9

1 Vérification de programmes par analyse des ressources 10
- 1.1 Analyse des types de ressources . . . 11
- 1.2 Modélisation des séquences d'accès aux ressources . . . 13
- 1.3 Analyse des usages des ressources . . . 15
- 1.4 Système de typage pour l'analyse des usages des ressources . . . 17
  - 1.4.1 Usages et Types . . . 17
  - 1.4.2 Sémantique des usages . . . 18
  - 1.4.3 Sous-typage . . . 20
  - 1.4.4 Environnement de typage . . . 21
  - 1.4.5 Système de typage . . . 22
- 1.5 Conclusion . . . 26

2 Vérification de la politique sécurité par analyse de programmes 27
- 2.1 Un modèle de programme . . . 27
- 2.2 Définition formelle des politiques de sécurité . . . 29
  - 2.2.1 Formalisme des politiques de sécurité . . . 29
  - 2.2.2 Exemples de politiques de sécurité . . . 30
  - 2.2.3 Vérification à états finis . . . 32
- 2.3 Exemple de commerce électronique . . . 35
- 2.4 Conclusion . . . 39

3 Historique d'effets et vérification **41**
3.1 Langage λ_{hist} 41
3.1.1 Syntaxe 42
3.1.2 Sémantique 42
3.2 Système de typage 44
3.3 Interprétation des historiques d'effets 45
3.4 Vérification des historiques d'effets 49
3.4.1 μ-*calcul* modal 49
3.4.2 Historiques d'effets et traces d'exécution 51
3.4.3 Propriétés de vérification 51
3.5 Variation orientée *pile* 52
3.5.1 Syntaxe et Sémantique 53
3.5.2 Transformation d'un historique d'effets en pile (*Stackified*) 53
3.5.3 Pile d'inspection et privilèges paramétrés 55
3.6 Conclusion 59

II Contribution **60**

4 Abstraction et vérification de programmes informatiques **61**
4.1 Introduction 61
4.2 Programmes 62
4.2.1 Syntaxe 63
4.2.2 Sémantique opérationnelle 65
4.2.3 Exemple 66
4.3 Sémantique statique 69
4.3.1 Définitions 69
4.3.2 Trace d'effets 71
4.3.3 Système de typage 73
4.3.4 Algorithme d'inférence de types et d'effets 76
4.4 Vérification de propriétés de sécurité 82
4.4.1 μ-*calcul* modal 85
4.4.2 Extension du μ-*calcul* modal 89
4.5 Conclusion 94

Conclusion Générale **95**

Bibliographie **98**

Table des figures

1 Architecture de notre approche de vérification de programmes. 7

2.1 Graphe dérivé G_{EC}. 40

Liste des tableaux

1.1 Sémantique opérationnelle du langage d'analyse des ressources. 16
1.2 Relation de *classement* ($U \preceq U'$). 18
1.3 Règles de réduction des usages. 20
1.4 Règles de typage. 24
1.5 Exemple d'un arbre de typage. 25

2.1 Langage de définition des politiques de sécurité. 30
2.2 Code système (domaine *Système*). 36
2.3 Code de l'application (domaine *Client*). 36
2.4 Code non certifié (domaine *Inconnu*). 37
2.5 Code du gestionnaire de comptes (domaine *Fournisseur*). 38

3.1 Syntaxe du langage λ_{hist}. 42
3.2 Sémantique du langage λ_{hist}. 43
3.3 Syntaxe du système de typage de λ_{hist}. 44
3.4 Règles de typage de λ_{hist}. 46
3.5 Typage de l'exemple. 47
3.6 Sémantique de la logique μ*-calcul*. 50
3.7 Sémantique du langage λ^S_{hist}. 53
3.8 Algorithme *stackify*. 54
3.9 Définition de $\phi_{demand,r}$ et $\phi_{inspect,r}$. 55
3.10 Preuve de typage de la fonction $checkit$. 57
3.11 Preuve de typage de la fonction $enableit$. 58

4.1 Syntaxe. 63
4.2 Comparaison avec le langage C. 64
4.3 Sémantique opérationnelle. 67
4.4 Exemple d'un programme. 68
4.5 Arbre sémantique du programme. 70
4.6 Exemple d'abstraction d'un programme. 72
4.7 Syntaxe du système de typage. 73
4.8 Règles de typage. 75
4.9 Arbre de typage de l'exemple précédent. 77
4.10 Algorithme d'inférence de types et d'effets (programmes). 82

4.11 Algorithme d'inférence de types et d'effets (fonctions). 82
4.12 Algorithme d'inférence de types et d'effets (expressions (1/3)). 83
4.13 Algorithme d'inférence de types et d'effets (expressions (2/3)). 84
4.14 Algorithme d'inférence de types et d'effets (expressions (3/3)). 85
4.15 Syntaxe du μ-*calcul*. 85
4.16 Syntaxe des actions, des types, des ressources et des étiquettes. 90
4.17 Syntaxe de la logique $\mathfrak{L}_\mu$. 91
4.18 Sémantique de la logique $\mathfrak{L}_d$. 92

Introduction

Introduction

De nos jours, Internet est devenu une partie intégrante de nos vies quotidiennes vu l'étendu des services qu'il offre. En effet, le nombre de logiciels dédiés à l'exploitation du réseau s'accroît d'une manière exponentielle. Toutefois, la propagation des téléchargements de programmes informatiques a donné naissance à un nouveau paradigme informatique nommé «*sécurité du code mobile*». Ses orientations se dirigent vers les réponses à la question : « Quand nous téléchargeons un programme à partir d'une source non fiable, comment pouvons-nous être sûrs que ce dernier ne va pas provoquer des comportements indésirables ? ».

La cryptographie, notamment à travers la signature du code mobile, est la première solution proposée dans la littérature pour répondre à ce problème. Néanmoins, elle n'offre que des mécanismes de sûreté lors de la transmission de données et ne présente aucun moyen de prouver si l'exécution locale du programme reçu est valide ou non : il est seulement possible d'affirmer que le code provient d'une source connue et qu'il n'a pas été modifié au cours de son transfert sur le réseau. Aussi, les recherches se sont orientées vers le développement d'approches formelles pouvant vérifier la sûreté et la sécurité des programmes informatiques. L'idée générale de ces approches consiste à extraire un modèle abstrait du programme puis à vérifier le respect de propriétés de sécurité.

La vérification par le typage est l'une des premières méthodes formelles permettant d'assurer la correction et la cohérence d'un programme au moment de sa compilation. Cette approche permet de détecter l'usage incohérent d'une valeur par rapport à la structure de cette dernière (comme par exemple affecter une valeur entière à une donnée de type « caractère »). Ainsi, aucune erreur d'accès aux données ne peut être causée au moment de l'exécution du programme si ce dernier est bien typé.

L'aspect le plus important de la vérification par le typage réside dans la simplicité de ses règles qui permettent de vérifier la structure d'un programme donné. De plus, cette

approche a démontré son efficacité pour le traitement des programmes définis à partir de langages procéduraux ; toutefois, ses limites sont vite atteintes lors de la vérification des programmes écrits dans des langages dits *impératifs*.

Pour répondre à cette limite, de nouvelles approches, nommées *systèmes à effets*, ont permis de prédire les événements pouvant être produits au moment de l'exécution d'un programme. Dans ce cas, un événement peut être défini comme une action exécutée par le programme (lire un fichier, envoyer une donnée sur le réseau, etc.). Chaque événement est empilé dans l'historique au moment de son apparition dans l'exécution du programme. Cet historique va être confornté à une propriété de sécurité qui ne peut être vérifiée que si ce dernier est bien formé, c'est-à-dire qu'il n'existe aucune séquence dangeureuse pouvant compromettre la sécurité du programme.

L'objectif de ce mémoire est de proposer un cadre permettant de vérifier la sécurité d'un code informatique via les données manipulées par un programme et la trace d'effets pouvant être engendrée au moment de son exécution. L'idée générale est de construire, au moment de la compilation, un modèle suffisamment représentatif de l'exécution réelle d'un programme donné, et le soumettre à un agorithme d'analyse d'assurer la cohérence de ce dernier avec une propriété de sécurité. Pour ce faire, nous procéderons à l'extraction des différentes dépendances reliant les données d'un programme. Par la suite, le programme va être traduit sous forme de *séquence d'événements* représentant les évolutions possibles de l'exécution de ce dernier. Puis, des règles de typage seront appliquées sur ce modèle afin de vérifier la cohérence de ce modèle envers le programme. Enfin, dans le cas où ce modèle est valide, un algorithme de vérification de modèles (*model-checking*) sera appliqué pour déterminer la conformité du modèle précédent par rapport à une propriété de sécurité exprimée sous forme d'une formule d'une logique temporelle étendue.

Mise en contexte

De nos jours, les vérification par le typage semblent être les outils les plus efficaces pour affronter l'analyse des programmes critiques [14]. En effet, les analyseurs statiques permettent de vérifier la sûreté d'un programme informatique par rapport à un ensemble de propriétés de sécurité. De plus, la correction des analyses statiques repose sur une théorie solide : *L'interprétation abstraite* [7]. Grâce à cette théorie, la correction d'une analyse peut être prouvée vis-à-vis de la sémantique d'un programme décrivant formellement le comportement dynamique de ce dernier. Aussi, cette section se propose d'exposer les bases de l'interprétation abstraite et d'introduire ainsi les concepts qui vont être repris lors de la définition de notre approche.

L'intérêt de cette section est d'introduire les concepts utilisés dans les mécanismes de vérification de programme et particulièrement ceux sur lesquels est basée notre approche. Dans un premier temps, nous introduirons les concepts de modèle et de sémantique. Par la suite, nous nous attarderons sur les types d'approches de la vérification de propriétés de sécurité. Enfin, nous étudierons les analyses des flots de données et de contrôle.

Modèles de calculs et sémantique

Un modèle de calcul ou d'exécution est une description mathématique formelle de la suite des opérations effectuées par un ordinateur exécutant un programme et de leurs effets internes à la machine et externes sur l'environnement, dans toutes les conditions possibles [8].

Certaines méthodes donnent la possibilité de raffiner ce modèle abstrait dans un modèle concret. En informatique, ce modèle concret se trouve alors très proche d'une implantation. Il s'agit alors de traduire ce modèle concret dans un langage de programmation. Après traduction, le code obtenu est alors exécutable. Il représente l'implantation formelle du système spécifié, en préservant les propriétés incluses au niveau le plus abstrait.

Selon Cousot [7], la modélisation mathématique a de nombreux avantages dont les principaux sont :

- une précision dans la spécification formelle supérieure à celle fournie par une description informelle en langage naturel qui est souvent ambiguë. C'est la force de la modélisation : lever les ambiguïtés,
- des fondements permettant d'énoncer des propriétés et d'étudier le comportement du système dans son ensemble.

Une sémantique d'un programme informatique est essentiellement un modèle de calcul décrivant les exécutions effectives de ce programme dans tous les environnements possibles [8, 26]. Aussi, on peut voir la sémantique comme une application d'un ensemble de termes vers un domaine de valeurs. Il est donc important que l'abstraction employée permette de concentrer la preuve sur les propriétés pertinentes pour le problème à résoudre [7]. Les informations ainsi calculées permettent alors de prouver statiquement (sans exécuter le programme) des propriétés sur le comportement dynamique du programme (quel que soit son environnement d'exécution).

Le postulat de base de l'interprétation abstraite est que toute sémantique $[[P]]$ d'un programme P peut être exprimée comme un ensemble de valeurs prises dans un domaine D (ensemble d'états, ensemble de traces). De manière générale, le domaine sémantique

est ainsi muni d'une structure de treillis complet [28]. Aussi, notre mécanisme d'abstraction d'un programme va être construit à partir de ce postulat afin d'aboutir à un modèle représentatif de ce dernier. Par la suite, nous devons établir un mécanisme de contrôle permettant de vérifier la cohérence du modèle par rapport à des propriétés de sécurité. Cette relation est construite à partir des concepts définis dans la section suivante.

Vérification de propriétés de sécurité

La spécification d'un programme est un modèle de calcul décrivant les exécutions souhaitées de ce programme dans tous les environnements possibles [22]. Généralement, le comportement d'un programme informatique peut être représenté sous forme d'un graphe dont les nœuds représentent les états du programme et les arcs dénotent les transitions faisant évoluer le programme d'un état à un autre. En d'autres mots, à un instant donné, le programme se trouve à l'état qui indique où l'exécution est rendue. La sémantique du programme peut alors être représentée sous forme d'une séquence incluant tous les états que va rencontrer le programme au moment de son exécution [37]. Dans ce cas, la vérification d'un programme P consiste à prouver que cette sémantique, notée $[[P]]$, satisfait une spécification S : $[[P]] \subseteq [[S]]$.

La vérification des propriétés de sécurité d'un programme informatique s'effectue au moyen d'un outil pouvant analyser et détecter les instructions malicieuses incorporées dans le code du programme. Toutefois, il existe deux courants majeurs pour la définition d'un tel outil :

- *L'approche opérationnelle* permet de détecter les différentes séquences d'actions menant à un état jugé dangereux. Formellement, cette approche consiste à spécifier les propriétés de sécurité en termes de comportements opérationnels. Ces spécifications sont généralement construites sur la base d'automates de sécurité de manière à pouvoir comparer l'automate dérivé du programme et l'automate de sécurité (voir [15, 16, 19, 23]). Cette comparaison se fait par l'intermédiaire de techniques de bissimulation, de simulation, etc.
- *L'approche logique* permet de spécifier les propriétés de sécurité sous forme de formules logiques. Elle offre une expressivité considérable permettant de modéliser un événement pouvant avoir lieu dans le futur. De plus, il existe un grand nombre d'outils pouvant vérifier la validité et la cohérence d'une formule logique donnée tel que les algorithmes de vérification de modèle (*model-checking*) [35, 39].

Dans la suite de ce document, nous allons nous intéresser aux mécanismes de vérification construits à partir de la deuxième approche qui nous semble la plus appropriée pour la définition de notre mécanisme de contrôle vu l'étendu de son expressivité. De plus,

cette approche est largement utilisée dans la littérature pour la définition de mécanismes d'analyse des flots de données et de contrôle [3, 4]. Les sections suivantes reviennent plus en détails sur l'analyse des dépendances de ressources et celle des traces d'événements.

Analyse du flot de données

Rassembler des informations sur les données manipulées par des programmes informatiques est devenu une tâche indispensable des processus de compilation et d'optimisation. Dans ce cas, l'analyse statique peut prendre la forme de prospections générales des propriétés d'un programme, ou bien se spécialiser dans la recherche des *dépendances* liant les différentes variables d'un programme.

L'analyse des dépendances consiste à employer des tests basés sur la recherche d'*alias* entre références [29]. Toutefois, les méthodes générales conduisent à des résultats approximatifs ou incapables de distinguer les différents types de dépendances. D'autres méthodes plus précises recherchent pour chaque cellule mémoire accédée en lecture (*le puits*), l'opération qui l'a produite (*la source*) : on parle alors d'analyse de flot de données (*Data Flow Analysis : DFA*). En d'autres mots, l'analyse va procéder à l'identification de chaque flot de données interférant dans le contenu de chaque variable critique d'un programme donné. Aussi, l'analyse des dépendances entre les variables d'un programme permet d'expliquer l'origine du contenu de chacune d'elles.

Il existe trois types de dépendances selon Pugh et Wonnacott [38] : *directe* (puits → source), *indirecte* (source → puits) et *anti-dépendance* (source → source). Une dépendance directe est une relation pouvant être associée à l'instruction d'écriture de l'instance qui assure la dépendance de flot la plus courte vers l'instruction de lecture. La source de cette instruction de lecture est donc à rechercher parmi les dépendances directes. Une fois les dépendances identifiées, il est aisé d'écrire un algorithme permettant de détecter toutes les affectations dangereuses (ex. affecter une valeur secrète à une variable publique). Pour se faire, il devient indispensable d'extraire le maximum d'informations disponibles statiquement à la compilation pour minimiser le nombre de dépendances superflues.

Analyse du flot de contrôle

Les systèmes de typage et les algorithmes de vérification de modèle sont deux approches permettant de vérifier et de valider les propriétés d'un programme donné. Un certain nombre d'auteurs [2, 4, 13] ont récemment observé que ces deux méthodes peuvent jouer des rôles complémentaires dans la vérification des propriétés de programme à travers

le concept de *trace* défini dans une logique temporelle : les systèmes de typage peuvent être aperçus comme des abstractions de programmes informatiques pouvant être analysées par des algorithmes de vérification de modèle pour valider la sûreté de ces derniers.

Les modèles sont construits à partir de *traces d'événements* générés par le programme, où ces derniers représentent les actions pouvant apparaître au moment de l'exécution des instructions du programme. Les événements sont définis d'une manière abstraite afin qu'ils puissent représenter une variété d'actions –e.g., l'ouverture d'un fichier, l'activation du contrôle d'accès aux privilèges ou les entrées et sorties de zones critiques. Les traces d'événements maintiennent les séquences d'opérations qui se produisent pendant l'exécution du programme. Certains travaux [4, 35, 36] ont démontré que des approximations statiques des traces d'événements peuvent être produites par les systèmes de typage, sous une forme favorable aux techniques de vérification de modèle existantes pour la vérification. Par exemple, l'analyse de traces a permis d'appliquer statiquement un ensemble de propriétés de sécurité sur des programmes de gestion de mémoire [3] et des ressources [4, 36].

Objectifs

L'objectif du document est de présenter un nouveau cadre d'analyse statique pour la vérification de programmes informatiques. La particularité de notre contribution réside dans le fait qu'elle définit un modèle de programmes permettant de procéder à une analyse du flot de données et du flot de contrôle d'un programme donné. Le premier aspect de l'analyse réside dans la vérification des dépendances entre les variables critiques du programme. Le second aspect, quant à lui, est relatif à l'analyse de la trace d'effets définissant le comportement du programme au moment de son exécution. Aussi, le modèle obtenu doit abstraire un programme à partir de ces deux attributs. Ce modèle va être soumis à un ensemble de propriétés de sécurité définies dans une logique temporelle. Notre choix de logique se porte sur le *μ-calcul* qui offre une grande expressivité nous permettant de formuler des propriétés de sécurité sous forme d'expressions logiques.

La figure 1 schématise l'architecture globale de notre approche et reflète les différentes étapes du processus de vérification. Nous reviendrons plus en détails dans le chapitre 4 (page 61) sur les mécanismes d'extraction d'informations, de validation et de vérification du flot de contrôle et celui des données.

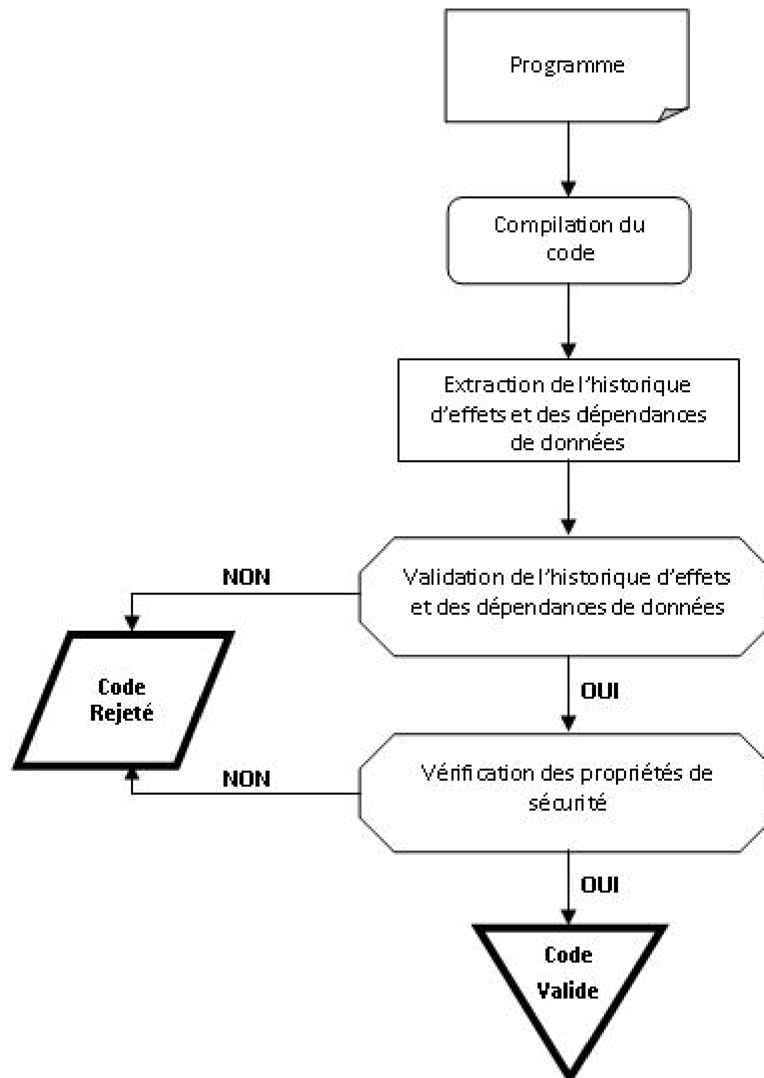

FIGURE 1 – Architecture de notre approche de vérification de programmes.

Structure du document

Ce document est divisé en deux partie principales :

- La première partie consiste en un état de l'art qui présente des approches de vérification de propriétés de sécurité de programmes informatiques. Nous nous sommes penchés sur trois approches qui nous semblaient pertinentes pour notre démarche. La première approche décrit un mécanisme de validation de programmes basé sur l'analyse du *flot de données*. Les deux autres approches représentent les processus de vérification de traces d'effets que nous avons adoptés pour la définition de notre mécanisme de vérification du flot de contrôle. Aussi, nous nous attardons sur le concept d'*historique d'effets* afin d'offrir une lecture commode pour toute personne qui désire s'initier à la vérification de logiciels informatiques.
- La deuxième partie présente notre contribution avec une nouvelle technique de vérification des propriétés de sécurité qui utilise les avantages des techniques présentées dans les parties précédentes, et plus particulièrement celle introduite par Skalka [4, 6]. Cette partie détaille notre contribution d'une manière graduelle afin de mettre en avant les différentes phases de vérification et de validation d'un programme informatique.

Ce mémoire se termine par une conclusion générale qui rappelle le contenu de chaque partie et les grandes lignes suivies au cours de ce travail ; l'intérêt et les limites des travaux sont soulignés, et quelques perspectives d'avenir sont énoncées.

Première partie

État de l'art

Chapitre 1

Vérification de programmes par analyse des ressources

L'accès aux ressources est l'un des critères les plus répandus dans la vérification des propriétés de sécurité des programmes informatiques. Par exemple, chaque fichier ouvert doit être refermé avant la fin de l'exécution du programme. Mais aussi, tout programme bien formé ne doit jamais accéder à une cellule mémoire non allouée. De même, un fichier ouvert doit être fermé avant la fin de l'exécution d'un programme.

Dans ce chapitre, nous présentons une approche proposant un mécanisme de vérification de programmes informatiques basée sur l'analyse des accès aux ressources [2, 3]. La plupart des approches proposées dans la littérature ne traitent que certains types d'accès [31]. Cette approche propose aussi un système de typage permettant de vérifier la validité d'un programme envers un ensemble de propriétés de sécurité exprimées par l'entremise d'une logique temporelle. L'idée principale est d'abstraire les différents accès aux ressources d'un programme par des primitives : initialisation (*init*), lecture (*read*), écriture (*write*), etc. Dans ce cas, un programme est dit sûr si les primitives sont exécutées dans un ordre valide.

Aussi, nous introduirons dans un premier temps le concept d'*usage*et son utilisation dans la vérification de la sécurité de programmes.Par la suite, nous présenterons la syntaxe ainsi que la sémantique opérationnelle d'un modèle pouvant abstraire les usages des ressources définies dans un programme sous forme de séquences d'accès. Puis, nous nous attarderons sur un système de typage permettant d'assurer la validité des séquences d'accès aux ressources d'un programme. Nous conclurons ce chapitre par l'identification de certaines lacunes du mécanisme de vérification de la sécurité.

1.1 Analyse des types de ressources

Afin de renforcer le mécanisme de vérification des propriétés de sécurité, cette approche se propose d'associer le type de chaque ressource avec une information relative à la séquence d'accès à cette dernière. Par exemple, le type ***File*** (fichier) est redéfini comme étant le couple $(\mathbf{File}, U)$, où U, appelé l'usage, dénote l'accès opéré sur le fichier. La syntaxe de l'*usage* peut être définie comme suit [1] :

$$U \quad ::= \quad l \mid U_1; U_2 \mid U_1 \& U_2 \mid \ldots$$

L'usage l dénote qu'une ressource, ou une variable, du programme est exploitée par une primitive, pouvant une instruction ou une fonction du programme, étiquetée par l. La séquence $U_1; U_2$ indique qu'une ressource est accédée successivement par U_1 puis par U_2. L'expression $U_1 \& U_2$ signale qu'une ressource peut être utilisée via un usage U_1 ou bien U_2. Par exemple, chaque fichier pouvant être accédé par une primitive étiquetée par l_1 ou bien par une primitive libellée par l_2 va être associé au type $(\mathbf{File}, l_1 \& l_2)$.

À partir de cette nouvelle définition des types de ressources, les auteurs assurent que les règles de typage du langage λ-*calcul* peuvent être étendues afin d'assimiler les séquences d'accès aux ressources d'un programme. Par exemple, La règle usuelle du typage des expressions polymorphes est définie comme suit :

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma, x : \tau \vdash N : \sigma}{\Gamma \vdash \texttt{let}\ \ x = M\ \ \texttt{in}\ \ N : \sigma}$$

En introduisant l'usage des ressources, cette règle peut être redéfinie comme suit :

$$\frac{\Gamma \vdash M : \tau \qquad \Delta, x : \tau \vdash N : \sigma}{\Gamma; \Delta \vdash \texttt{let}\ \ x = M\ \ \texttt{in}\ \ N : \sigma}$$

Dans ce cas, les environnements de typage Γ et Δ indiquent que les ressources attachées aux variables libres du programme doivent être exploitées successivement selon les usages qu'ils contiennent. En d'autres mots, cette nouvelle règle certifie que l'évaluation de M va précéder celle de N. Par exemple, étant donné les jugements $y : (\mathbf{File}, l_1) \vdash M : \mathbf{bool}$ et $y : (\mathbf{File}, l_2), x : \mathbf{bool} \vdash N : \mathbf{bool}$, alors on peut conclure que : $y : (\mathbf{File}, \mathbf{l_1}; \mathbf{l_2}) \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \mathbf{bool}$. Autrement dit, le dernier jugement indique que la ressource y est un fichier qui va être exploité par une primitive étiquetée par l_1

1. La définition des usages est présentée dans la section 1.2 à la page 13.

puis par une autre ayant l_2 comme label. Toutefois, ce raisonnement peut occasionner un ensemble d'erreurs de typage. Par exemple, nous pouvons obtenir le jugement suivant :

$$\frac{\begin{array}{l} y : (\mathbf{File}, l_w) \vdash y : (\mathbf{File}, l_w) \\ y : (\mathbf{File}, l_r), x : (\mathbf{File}, l_w) \vdash \mathit{fread}^{l_r}(y); \mathit{fwrite}^{l_w}(x) : \mathbf{bool} \end{array}}{y : (\mathbf{File}, l_w; l_r) \vdash \mathsf{let}\ x = y\ \mathsf{in}\ (\mathit{fread}^{l_r}(y); \mathit{fwrite}^{l_w}(x)) : \mathbf{bool}}$$

Dans ce cas, nous pouvons conclure que le fichier y va être modifié (accès en écriture) par une primitive associée au label l_w puis il va être lu (accès en lecture) par une autre primitive étiquetée par l_r, ce qui est incorrect. Cette inconsistance est due au fait que l'accès à la ressource y, représenté dans l'environnement par $y : (\mathbf{File}, l_w)$, ne se produit pas lors de l'évaluation de y mais plutôt au moment de l'évaluation de l'expression $\mathit{fread}^{l_r}(y); \mathit{fwrite}^{l_w}(x)$. Aussi, afin de répondre à cette inconsistance, cette approche introduit l'usage $\Box U$ dénotant qu'une ressource peut être exploitée selon l'usage U *immédiatement* (au moment de l'évaluation de l'expression manipulant cette ressource) ou *ultérieurement* (lors de l'utilisation de la valeur relative à l'expression utilisant cette ressource). Aussi, le jugement précédent peut être corrigé en introduisant l'opérateur $\Box$ comme suit :

$$\frac{\begin{array}{l} y : (\mathbf{File}, \Box l_w) \vdash y : (\mathbf{File}, l_w) \\ y : (\mathbf{File}, l_r), x : (\mathbf{File}, l_w) \vdash \mathit{fread}^{l_r}(y); \mathit{fwrite}^{l_w}(x) : \mathbf{bool} \end{array}}{y : (\mathbf{File}, \Box l_w; l_r) \vdash M : \mathbf{bool}}$$

La prémisse $y : (\mathbf{File}, \Box l_w) \vdash y : (\mathbf{File}, l_w)$ dénote le fait que la ressource y est exploitée uniquement lorsque la valeur de y est utilisée au plus tard (au moment de l'évaluation de la primitive $\mathit{fwrite}^{l_w}(x)$). En d'autres mots, chaque accès en lecture (l_r) du fichier y peut être suivi soit *immédiatement* par un accès en écriture (l_w) soit *ultérieurement*.

Toutefois, l'introduction de cet opérateur ne répond qu'en partie au problème d'inconsistance défini précédemment. Par exemple, si on dispose du jugement $x : (\mathbf{File}, \Box l) \vdash M : \tau$ et si l'évaluation de l'expression M ne comporte aucune référence relative à la ressource x. Alors, on peut conclure que le fichier x est exploité *au moment* de l'évaluation de M. Cette propriété peut être décrite via l'introduction du nouvel opérateur $\blacksquare$ permettant d'annuler l'effet de l'opérateur $\Box$ comme suit :

$$\frac{\Gamma, x : \tau \vdash M : \sigma \qquad x \text{ est directement liée à } M}{\Gamma, x : \blacksquare\tau \vdash M : \sigma}$$

Nous allons revenir plus en détails sur l'utilité des deux opérateurs introduits précédemment dans la suite de ce chapitre.

1.2 Modélisation des séquences d'accès aux ressources

Le mécanisme de vérification introduit par cette approche est construit à partir d'un langage modélisant tous les accès possibles aux ressources d'un programme. Ce langage reprend les différentes formules du λ-*calcul* et introduit de nouvelles expressions représentant la création et l'exploitation de ressources.

Dans ce qui suit, $\mathfrak{L}$ dénote un ensemble infini d'étiquettes représentées par la méta-variable l. $\mathfrak{L}^*$ représente l'ensemble des séquences finies de labels, et $\mathfrak{L}^{*,\downarrow}$ dénote l'ensemble $\mathfrak{L}^* \cup \{s\downarrow \mid s \in \mathfrak{L}^*\}$. Le symbole $\downarrow$ désigne la terminaison du programme. Chaque élément appartenant à $\mathfrak{L}^{*,\downarrow}$ est appelé *trace*, où ϵ représente une trace vide et $s_1 s_2$ dénote la concaténation des deux traces s_1 et s_2.

L'ensemble des traces, noté Φ, comporte toutes les séquences constituant des préfixes pour tous les éléments de $\mathfrak{L}^{*,\downarrow}$, c'est-à-dire si $ss' \in \mathfrak{L}^{*,\downarrow}$ alors $s \in \Phi$. $S^{\#}$ dénote l'ensemble de tous les préfixes des éléments de S : $\{s \in \mathfrak{L}^{*,\downarrow} \mid ss' \in S\}$.

Aussi, chaque terme du langage, noté M, peut être défini comme suit :

$$\begin{array}{lll} M & ::= & \mathsf{true} \mid \mathsf{false} \mid x \mid \lambda^{\Phi}x.M \mid \mathsf{fix}^{\Phi}(f,x,M) \mid \mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \mid \\ & & M_1 @^{\Phi} M_2 \mid \mathsf{new}^{\Phi}() \mid \mathsf{acc}_i^l(M) \mid \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \end{array}$$

Comme on peut le remarquer, ce langage introduit deux nouveaux constructeurs : new^{Φ} dénotant la création d'une nouvelle ressource, et $\mathsf{acc}_i^l(M)$ désignant l'accès à une ressource M. Le deuxième constructeur doit retourner une valeur booléenne indiquant l'état de l'accès : ce constructeur retourne *true* si l'accès est accompli sinon *false*. Cette propriété peut être appliquée à une séquence d'accès $(\mathsf{acc}_1, ..., \mathsf{acc}_n)$. Le terme $\mathsf{fix}^{\Phi}(f,x,M)$ désigne une fonction récursive f qui satisfait la propriété $f = \lambda x.M$. L'expression polymorphe $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2$ est équivalente sémantiquement au terme $(\lambda^{\Phi}x.M_2)@^l M_1$. Il faut noter que l'opérateur $M_1 @^{\Phi} M_2$ correspond à l'application $M_1(M_2)$ soumise aux traces définies dans Φ.

Tout label l doit être associé à chaque occurrence d'une primitive d'accès à une ressource. Cette contrainte permet au système de vérification de construire une trace à partir des différents accès aux ressources d'un programme durant son exécution et de vérifier si cette dernière satisfait celles associées aux différents opérateurs (new, @, fix, ...) par le programmeur. Le même label peut être affilié à plusieurs occurrences d'accès d'une même primitive. De la même manière, l'ensemble des traces Φ doit être attaché à chaque occurrence d'une abstraction λ, et de chaque fonction récursive ($\mathsf{fix}^{\Phi}(f,x,M)$), et de chaque occurrence de la primitive de création de ressources ($\mathsf{new}^{\Phi}()$). Par exemple, la primitive

$\mathsf{new}^{\{l_1 l_2 \downarrow\}^{\#}}()$ crée une ressource qui admet un accès l_1 suivi par un autre accès l_2 avant la terminaison de l'évaluation de l'expression où est introduite la dite ressource.

Exemple Soient init , read , write et reset les primitives permettant, respectivement, d'initialiser, de lire, de modifier, et de libérer une ressource. Le programme suivant permet de créer une ressource r, de l'initialiser, puis appelle une fonction f. La fonction f va alors procéder à la lecture de r, puis cette dernière va être mise à jour avant d'être libérée.

$$\begin{array}{lcl}\mathsf{let}\ f & = & \mathsf{fix}^{\Phi_f}(f, x,\ \mathsf{if}\ \mathsf{read}^{l_R}(x)\ \mathsf{then}\ \mathsf{reset}^{l_F}(x)\ \mathsf{else}\ (\mathsf{write}^{l_W}; f@^{l_1}x))\ \mathsf{in} \\ \mathsf{let}\ r & = & \mathsf{new}^{\Phi_r}()\ \mathsf{in}\ (\mathsf{init}^{l_I}(r); f@^{l_2}r)\end{array}$$

Dans cet exemple, on dispose de deux ensembles de traces, Φ_r et Φ_f, relatifs à la ressource r et à la fonction f :

$$\begin{array}{lcl}\Phi_r & = & (l_I(l_R + l_W)^* l_F \downarrow)^{\#} \\ \Phi_f & = & \mathfrak{L}^{*,\downarrow}\end{array}$$

L'ensemble Φ_r indique que la ressource doit tout d'abord être initialisée (l_I), avant d'être accédée en lecture ou en écriture ($l_R + l_W$), pour être libérée à la fin du programme (l_F). L'ensemble Φ_f indique que la fonction f n'est soumise à aucune contrainte.

La sémantique opérationnelle du langage de vérification fait appel au concept de *tas*. L'intérêt principal d'un tas est la mémorisation des occurrences d'accès d'une ressource. Formellement, un tas peut être défini comme suit :

Définition 1.2.1 *Chaque élément h appartenant à l'ensemble des tas de valeurs est construit à partir de la syntaxe suivante :*

$$h \quad ::= \quad \mathcal{R} \mid \lambda x.M$$

Où $\mathcal{R}$ dénote une ressource créée par la primitive $\mathtt{new}^{\Phi}()$ et $\lambda x.M$ désigne une fonction. Un tas H est une fonction associant à chaque ressource du programme une valeur h et une trace s. En d'autres mots, la détermination de la valeur d'un programme est effectuée à travers l'évaluation de toutes ses ressources à travers le tas H.

L'expression $\{x_1 \mapsto^{\Phi_1} h_1, ..., x_n \mapsto^{\Phi_n} h_n\}$ indique que $dom(H) = \{x_1, ..., x_n\}$ et que $H(x_i) = (h_i, \Phi_i)$. L'expression $H_1 \uplus H_2$ dénote que $dom(H_1) \cap dom(H_2) = \varnothing$. Aussi, si $H = H_1 \uplus H_2$ alors $dom(H) = dom(H_1) \cup dom(H_2)$ et $H(x) = H_i(x)$ si $x \in dom(H_i)$.

Définition 1.2.2 *La notation Φ^{-l} dénote l'ensemble des traces succédant à l'usage l : $\Phi^{-l} = \{s \,|\, ls \in \Phi\}$.*

Définition 1.2.3 (Contexte d'évaluation) *La syntaxe des contextes d'évaluation est définie comme suit :*

$$\mathcal{E} \quad ::= \quad [] \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 \mid \mathcal{E}@^l M \mid v@^l\mathcal{E} \mid \mathsf{acc}_i^l(\mathcal{E}) \mid \mathsf{let}\ x = \mathcal{E}\ \mathsf{in}\ M$$

Où v dénote une variable ou bien une valeur booléenne (`true` ou `false`). La notation $[v/x]$ indique l'instanciation de la variable x par v.

Les règles de la sémantique opérationnelle sont définies à partir de la relation de réduction $(H, M) \rightsquigarrow P$, où P est soit une *erreur* (Error) soit le couple (H', M'). La sémantique opérationnelle de cette approche est décrite dans la table 1.1.

Exemple Soit M le programme ci-dessous obtenu par la suppression de la primitive $\mathsf{init}^{l_1}(r)$ du programme défini précédemment (Exemple 1.2, page 14) :

$$\begin{array}{lcl} \mathsf{let}\ f & = & \mathsf{fix}^{\Phi_f}(f, x,\ \mathsf{if}\ \mathsf{read}^{l_R}(x)\ \mathsf{then}\ \mathsf{reset}^{l_F}(x)\ \mathsf{else}\ (\mathsf{write}^{l_W}; f@^{l_1}x))\ \mathsf{in} \\ \mathsf{let}\ r & = & \mathsf{new}^{\Phi_r}()\ \mathsf{in}\ f@^{l_2}r \end{array}$$

En admettant que $\rightsquigarrow^*$ dénote la fermeture réflexive et transitive de $\rightsquigarrow$, on peut établir que l'évaluation de M va échouer vu que la ressource r va être lue avant d'être initialisée :

$$\begin{array}{ll} & (\{\}, M) \\ \rightsquigarrow^* & (\{x_1 \mapsto^{\mathfrak{L}^{*,\downarrow}} \lambda x.\mathsf{if}\ \mathsf{read}^R(x)\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots, x_2 \mapsto^{\Phi_r} \mathcal{R}\}, x_1@^{l_2}x_2) \\ \rightsquigarrow & (\{x_1 \mapsto^{\mathfrak{L}^{*,\downarrow}} \lambda x.(\ldots), x_2 \mapsto^{(l_I(L_R+l_W)^* l_F\downarrow)^{\#}} \mathcal{R}\}, \mathsf{if}\ \mathsf{read}^{l_R}(x_2)\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots) \\ \rightsquigarrow & \mathbf{Error} \end{array}$$

1.3 Analyse des usages des ressources

L'analyse des usages permet de vérifier la sûreté d'un programme par rapport à l'exploitation de ses ressources. Un programme M est dit *sûr* si son évaluation ne provoque aucun accès dangereux et si toutes ses ressources sont exploitées avant la terminaison de l'exécution de ce dernier.

$$\frac{z \text{ est une nouvelle variable}}{(H, \mathcal{E}[\mathsf{new}^{\Phi}()]) \rightsquigarrow (H \uplus \{z \mapsto^{\Phi} \mathcal{R}\}, \mathcal{E}[z])} \ (\mathsf{R-AlcRes})$$

$$\frac{z \text{ est une nouvelle variable}}{(H, \mathcal{E}[\lambda^{\Phi}x.M]) \rightsquigarrow (H \uplus \{z \mapsto^{\Phi} \lambda x.M\}, \mathcal{E}[z])} \ (\mathsf{R-AlcLam})$$

$$\frac{z \text{ est une nouvelle variable}}{(H, \mathcal{E}[\mathtt{fix}^{\Phi}(f, x, M)]) \rightsquigarrow (H \uplus \{z \mapsto^{\Phi} \lambda x.[z/f]M\}, \mathcal{E}[z])} \ (\mathsf{R-AlcFix})$$

$$\frac{b = \mathsf{true\ ou\ false} \qquad \Phi^{-l} \neq \varnothing}{(H \uplus \{x \mapsto^{\Phi} \mathcal{R}\}, \mathcal{E}[\mathtt{acc}_i^l(x)]) \rightsquigarrow (H \uplus \{x \mapsto^{\Phi^{-l}} \mathcal{R}\}, \mathcal{E}[b])} \ (\mathsf{R-Acc})$$

$$\frac{\phi^{-l} = \varnothing}{(H \uplus \{x \mapsto^{\Phi} \mathcal{R}\}, \mathcal{E}[\mathtt{acc}_i^l(x)]) \rightsquigarrow \mathtt{Error}} \ (\mathsf{R-AccErr})$$

$$\frac{\Phi^{-l} \neq \varnothing}{(H \uplus \{x \mapsto^{\Phi} \lambda y.M\}, \mathcal{E}[x @^l v]) \rightsquigarrow (H \uplus \{x \mapsto^{\Phi^{-l}} \lambda y.M\}, \mathcal{E}[[v/y]M])} \ (\mathsf{R-App})$$

$$\frac{\Phi^{-l} = \varnothing}{(H \uplus \{x \mapsto^{\Phi} \lambda y.M\}, x @^l v) \rightsquigarrow \mathtt{Error}} \ (\mathsf{R-AppErr})$$

$$\frac{\Box}{(H, \mathcal{E}[\mathsf{if\ true\ then}\ M_1\ \mathsf{else}\ M_2]) \rightsquigarrow (H, \mathcal{E}[M_1])} \ (\mathsf{R-ifT})$$

$$\frac{\Box}{(H, \mathcal{E}[\mathsf{if\ false\ then} M_1\ \mathsf{else}\ M_2]) \rightsquigarrow (H, \mathcal{E}[M_2])} \ (\mathsf{R-ifF})$$

TABLE 1.1 – Sémantique opérationnelle du langage d'analyse des ressources.

Définition 1.3.1 *Un programme M est dit sûr si :*
- $(\{\}, M) \not\rightsquigarrow^* \mathtt{Error}$
- *si* $(\{\}, M) \rightsquigarrow^* (H, v)$, *alors pour toute variable* $x \in dom(H)$, $H(x) = (h, \Phi)$ *et* $\downarrow \in \Phi$.

Exemple Nous pouvons démontrer que le programme défini dans l'exemple de la section 1.2(page 14) est *sûr* comme suit :

$$\begin{array}{ll}
 & (\{\}, M) \\
\rightsquigarrow^* & (\{x_1 \mapsto^{\mathfrak{L}^*,\downarrow} \lambda x.\ \mathsf{if\ read}^R(x)\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots, x_2 \mapsto^{((L_R+l_W)^* l_F \downarrow)^\#} \mathcal{R}\}, x_1 @^{l_2} x_2) \\
\rightsquigarrow & (\{x_1 \mapsto^{\mathfrak{L}^*,\downarrow} \lambda x.(\ldots), x_2 \mapsto^{((L_R+l_W)^* l_F \downarrow)^\#} \mathcal{R}\},\ \mathsf{if\ read}^{l_R}(x_2)\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots) \\
\rightsquigarrow^* & (\{x_1 \mapsto^{\mathfrak{L}^*,\downarrow} \lambda x.(\ldots), x_2 \mapsto^{\downarrow^\#} \mathcal{R}\}, [])
\end{array}$$

On remarque qu'à la fin de l'exécution, la ressource x_2 a été exploitée par tous les usages introduits dans la trace $((L_R + l_W)^* l_F \downarrow)^{\#}$: la ressource x_2 doit être exploitée en lecture ou en écriture, puis elle doit être libérée avant la terminaison du programme.

1.4 Système de typage pour l'analyse des usages des ressources

Cette section expose un système de typage permettant que chaque programme *bien-typé* est sûre. Pour se faire, chaque type associé à une ressource est étendu par l'introduction d'un usage exprimant les accès qu'effectue le programme sur cette dernière.

1.4.1 Usages et Types

Un usage U est défini à partir de la grammaire suivante :

$$U \quad ::= \quad \mathbf{0} \mid \alpha \mid l \mid U_1 \& U_2 \mid U_1 ; U_2 \mid U_1 \otimes U_2 \mid \mu\alpha.U \mid \square U \mid \blacksquare U \mid U_1 \odot U_2 \mid U_1 \triangleright U_2$$

$\mathbf{0}$ désigne l'usage d'une ressource qui ne peut pas être exploitée tout au long de l'exécution d'un programme. α dénote un usage variable. Les usages l, $U_1; U_2$, et $U_1 \& U_2$ ont été introduits précédemment dans la section 1.1. $U_1 \otimes U_2$ désigne l'usage d'une ressource qui peut être exploitée par U_1 et U_2 d'une manière intercalée : $(l_1; l_2) \otimes l_3$ est équivalent à l'usage $(l_3; l_1; l_2) \& (l_1; l_3; l_2) \& (l_1; l_2; l_3)$. $\mu\alpha.U$ dénote l'usage récursif d'une ressource tel que $\alpha = U$.Par exemple, $\mu\alpha.(l; \alpha)$ indique que la ressource va être exploitée plusieurs fois selon l'usage l. $\square U$ indique que la ressource peut être exploitée selon l'usage U *immédiatement* ou bien *ultérieurement* : $\square l_1; l_2$ dénote que la ressource peut être exploitée soit par l_1 puis l_2, soit par l_2 puis l_1. $\blacksquare U$ indique que l'usage U va être appliqué *immédiatement* à la ressource : l'utilité de l'opérateur $\blacksquare$ est d'annuler l'effet de $\square$. L'usage $U_1 \odot U_2$ signifie que chaque accès U_2 à une ressource se produit à la suite d'une occurrence de l'accès U_1 à cette même ressource. $U_1 \triangleright U_2$ indique que pour chaque accès l défini dans l'usage U_1 est suivi par l'accès U_2 : $(l_1; l_2) \triangleright U$ est équivalent à l'usage $(l_1; U); (l_2; U)$. Il faut noter ue les opérateurs uniaires $\square$ et $\blacksquare$ sont prioritaires sur les opérateurs binaires : l'expression $\square l_1; l_2$ est équivalente à $(\square l_1); l_2$.

Chaque ressource est associée à un type τ défini comme suit :

$$\tau \quad ::= \quad \texttt{bool} \mid (\tau_1 \to \tau_2, U) \mid (\mathbf{R}, U)$$

$$\frac{\Box}{U\&U' \preceq U} \qquad \frac{\Box}{U\&U' \preceq U'} \qquad \frac{\Box}{\mu\alpha.U \preceq [\mu\alpha.U/\alpha]U}$$

$$\frac{U \preceq U'}{\Box U \preceq \Box U'} \qquad \frac{U \preceq U'}{\blacksquare U \preceq \blacksquare U'}$$

$$\frac{U_1 \preceq U_1' \quad U_2 \preceq U_2'}{U_1;U_2 \preceq U_1';U_2'} \qquad \frac{U_1 \preceq U_1' \quad U_2 \preceq U_2'}{U_1 \otimes U_2 \preceq U_1' \otimes U_2'}$$

$$\frac{U_1 \preceq U_1' \quad U_2 \preceq U_2'}{U_1 \triangleright U_2 \preceq U_1' \triangleright U_2'} \qquad \frac{U_1 \preceq U_1' \quad U_2 \preceq U_2'}{U_1 \odot U_2 \preceq U_1' \odot U_2'}$$

TABLE 1.2 – Relation de *classement* ($U \preceq U'$).

Le type $(\tau_1 \rightarrow \tau_2, U)$ est associé aux fonctions qui vont être appelées selon l'usage U : $(\texttt{bool} \rightarrow \texttt{bool}, l_1; l_2)$ est le type d'une fonction booléenne qui est appelée premièrement selon l'usage l_1 puis selon l'usage l_2. $(\mathbf{R}, U)$ est le type d'une ressource exploitée avec l'usage U.

L'usage d'un type τ est obtenu via la fonction $usage(\tau)$ comme suit :

$$\begin{array}{lcl} usage(\texttt{bool}) & = & \mathbf{0} \\ usage(\tau_1 \rightarrow \tau_2, U) & = & U \\ usage(\mathbf{R}, U) & = & U \end{array}$$

1.4.2 Sémantique des usages

La sémantique des usages est définie à partir d'un système de transitions étiquetées. Dans ce cas, chaque usage dénote un ensemble de traces obtenues à partir des séquences de transitions. De plus, une relation de *classement*, notée $\preceq$, est définie sur l'ensemble des usages, noté $\mathcal{U}$, permettant d'ordonner un usage à partir d'un autre usage : $U_1 \preceq U_2$ signifie que U_1 est plus permissif que U_2. La relation de sous-usage $\preceq$ est définie selon les règles décrites dans la figure 1.2 (page 18).

Définition 1.4.1 *Les relations unaires* $.^{\downarrow}$ *et* $.^{\Downarrow}$ *sont les relations minimales définies sur l'ensemble des usages et satisfaisant les propriétés suivantes :*

- $\mathbf{0}^{\downarrow}$		
- $U^{\downarrow}$	$\Rightarrow$	$((\square U)^{\downarrow} \wedge (\blacksquare U)^{\downarrow} \wedge (U \odot U^{'})^{\downarrow} \wedge (U^{'} \odot U)^{\downarrow} \wedge (U \triangleright U^{'})^{\downarrow})$
- $(U_1^{\downarrow} \wedge U_2^{\downarrow})$	$\Rightarrow$	$(U_1 \otimes U_2)^{\downarrow} \wedge (U_1; U_2)^{\downarrow} \wedge (U_1 \& U_2)^{\downarrow}$
- $([\mu\alpha.U/\alpha]U)^{\downarrow}$	$\Rightarrow$	$(\mu\alpha.U)^{\downarrow}$
- $\mathbf{0}^{\Downarrow}$		
- $(\square U)^{\Downarrow}$		
- $U^{\downarrow}$	$\Rightarrow$	$(\blacksquare U)^{\Downarrow}$
- $U^{\Downarrow}$	$\Rightarrow$	$((U \odot U^{'})^{\Downarrow} \wedge (U^{'} \odot U)^{\Downarrow} \wedge (U \triangleright U^{'})^{\Downarrow})$
- $(U^{\Downarrow} \wedge U^{'\Downarrow})$	$\Rightarrow$	$((U \otimes U^{'})^{\Downarrow} \wedge (U; U^{'})^{\Downarrow} \wedge (U \& U^{'})^{\Downarrow})$
- $([\mu\alpha.U/\alpha]U)^{\Downarrow}$	$\Rightarrow$	$(\mu\alpha.U)^{\Downarrow}$

Où, l'expression $U^{\downarrow}$ indique que la ressource ne peut être accessible qu'après l'évaluation complète du terme dans lequel elle apparaît. La notation $U^{\Downarrow}$ souligne que la ressource n'est pas accessible immédiatement. De plus, si $U \preceq U^{'}$ pour un usage $U^{'}$ donné et $U^{'\downarrow}$ alors on peut écrire $U^{\preceq\downarrow}$. Respectivement, si $U \preceq U^{'}$ pour un usage $U^{'}$ donné et $U^{'\Downarrow}$ alors on peut écrire $U^{\preceq\Downarrow}$.

La notation $\square\mathfrak{L}$ désigne l'ensemble $\{\square l \mid l \in \mathfrak{L}\}$. Tout élément, noté L, appartenant à l'ensemble $\mathfrak{L} \cup \square\mathfrak{L}$ est appelé *label étendu*. Les labels $\square L$ et $\blacksquare L$ sont définis comme suit :

$$\square L = \begin{cases} \square L & \text{si } L \in \mathfrak{L} \\ L & \text{si } L \in \square\mathfrak{L} \end{cases}$$

$$\blacksquare L = \begin{cases} L & \text{si } L \in \mathfrak{L} \\ l & \text{si } L = \square l \end{cases}$$

Si S dénote un ensemble de labels étendus alors $\square S$ et $\blacksquare S$ dénotent respectivement les ensembles $\{\square L \mid L \in S\}$ et $\{\blacksquare L \mid L \in S\}$.

La relation de transition $U \xrightarrow{L} U^{'}$ indique qu'une ressource associée à U peut être exploitée selon l'usage L puis $U^{'}$. La relation de transition est définie à travers des règles de réduction des usages dans la figure 1.3.

Par exemple, l'usage $\square l_1; l_2$ peut être soumis aux transitions suivantes : $\square l_1; l_2 \xrightarrow{\square l_1} \mathbf{0}; l_2 \xrightarrow{l_2} \mathbf{0}; \mathbf{0}$ ou bien $\square l_1; l_2 \xrightarrow{l_2} \square l_1; \mathbf{0} \xrightarrow{\square l_1} \mathbf{0}; \mathbf{0}$. Alors que l'usage $l_1; l_2$ ne peut évoluer qu'à travers une seule séquence de transitions : $l_1; l_2 \xrightarrow{l_1} \mathbf{0}; l_2 \xrightarrow{l_2} \mathbf{0}; \mathbf{0}$.

Chaque usage U est associé à un ensemble de traces, noté $[[U]]$, qui est défini comme suit :

Définition 1.4.2 *Soit U un usage. $[[U]]$ dénote l'ensemble :*

$$\frac{}{l \xrightarrow{l} \mathbf{0}} \ (\mathsf{UR-Zero}) \qquad \frac{U \xrightarrow{L} U'}{\Box U \xrightarrow{\Box L} \Box U'} \ (\mathsf{UR-Box}) \qquad \frac{U \xrightarrow{L} U'}{\blacksquare U \xrightarrow{\blacksquare L} \blacksquare U'} \ (\mathsf{UR-UnBox})$$

$$\frac{U_1 \xrightarrow{L} U_1'}{U_1 \otimes U_2 \xrightarrow{L} U_1' \otimes U_2} \ (\mathsf{UR-ParL}) \qquad \frac{U_2 \xrightarrow{L} U_2'}{U_1 \otimes U_2 \xrightarrow{L} U_1 \otimes U_2'} \ (\mathsf{UR-ParR})$$

$$\frac{U_1 \xrightarrow{L} U_1'}{U_1; U_2 \xrightarrow{L} U_1'; U_2} \ (\mathsf{UR-SeqL}) \qquad \frac{U_2 \xrightarrow{L} U_2' \quad U_1^{\Downarrow}}{U_1; U_2 \xrightarrow{L} U_1; U_2'} \ (\mathsf{UR-SeqR})$$

$$\frac{U_1 \xrightarrow{l} U_1' \quad U_2 \xrightarrow{L} U_2'}{U_1 \odot U_2 \xrightarrow{l \odot L} U_2'; (U_1' \odot U_2)} \ (\mathsf{UR-Mult1}) \qquad \frac{U_1 \xrightarrow{\Box l} U_1' \quad \Box U_2 \xrightarrow{L} U_2'}{U_1 \odot U_2 \xrightarrow{\Box l \odot L} U_2'; (U_1' \odot U_2)} \ (\mathsf{UR-Mult2})$$

$$\frac{U_1 \xrightarrow{l} U_1'}{U_1 \triangleright U_2 \xrightarrow{l} U_2; (U_1' \triangleright U_2)} \ \mathsf{UR-SMult1} \qquad \frac{U_1 \xrightarrow{\Box l} U_1'}{U_1 \triangleright U_2 \xrightarrow{\Box l} \Box U_2; (U_1' \triangleright U_2)} \ \mathsf{UR-SMult2}$$

$$\frac{U \preceq U'' \quad U'' \xrightarrow{L} U'}{U \xrightarrow{L} U'} \ (\mathsf{UR-PCong})$$

TABLE 1.3 – Règles de réduction des usages.

$$\{(\blacksquare L_1) \cdots (\blacksquare L_n) \mid \exists U_1, \ldots, U_n : (U \xrightarrow{L_1} U_1 \xrightarrow{L_2} U_2 \cdots U_{n-1} \xrightarrow{L_n} U_n)\} \cup$$
$$\{(\blacksquare L_1) \cdots (\blacksquare L_n) \downarrow \ \mid \exists U_1, \ldots, U_n : (U \xrightarrow{L_1} U_1 \xrightarrow{L_2} U_2 \cdots U_{n-1} \xrightarrow{L_n} U_n) \wedge U_n^{\preceq\downarrow}\}$$

Où $n \geq 0$ *(*$\epsilon \in [[U]]$ *pour tout usage* U*).*

Aussi, nous pouvons établir que :

- $[[\mathbf{0}]] = \{\epsilon, \downarrow\}$
- $[[\mu\alpha.\alpha]] = \{\epsilon\}$
- $[[\Box(l_1; l_2); l_3]] = \{l_1 l_2 l_3 \downarrow, l_1 l_3 l_2 \downarrow, l_3 l_1 l_2 \downarrow\}^{\#}$
- $[[\mu\alpha.(\mathbf{0} \& (l; \alpha))]] = \{\downarrow, l \downarrow, ll \downarrow, \ldots\}^{\#}$

1.4.3 Sous-typage

Cette approche introduit une relation de *sous-usage*, notée $U_1 \leq U_2$, et une relation de *sous-typage*, notée $\tau_1 \leq \tau_2$. La première relation indique que U_1 caractérise un usage plus

général que celui de U_2, aussi une ressource exploitée selon l'usage U_1 peut l'être aussi avec U_2. La deuxième relation souligne qu'une valeur de type τ_1 peut être aussi manipulée comme étant une valeur de type τ_2.

La relation de sous-usage est construite sur la base des *contextes d'usage*. Formellement, un contexte d'usage, noté C, est l'expression obtenue à partir d'un usage par la substitution d'une occurrence d'une variable libre définie dans ce dernier avec []. Aussi, la notation $C[U]$ désigne l'usage obtenu par la substitution de [] avec U : si $C = \mu\alpha.([];\alpha)$, alors $C[U] = \mu\alpha.(U;\alpha)$.

Définition 1.4.3 *Pour tous les usages $U_1, U_2 \in \mathcal{U}$, si $U_1 \leq U_2$ alors les conditions suivantes sont satisfaites :*
- $C[U_1] \leq C[U_2]$ *pour tout contexte d'usage* C *;*
- *si* $U_2 \xrightarrow{L} U_2'$*, alors il existe un usage* U_1' *tel que* $U_1 \xrightarrow{L'} U_1'$*,* $U_1' \leq U_2'$ *et* $L' = L$ *ou bien* $L' = \Box L$ *;*
- *si* $U_2^{\preceq\downarrow}$ *alors* $U_1^{\preceq\downarrow}$.

La notation $U_1 \cong U_2$ indique que $U_1 \leq U_2$ et $U_2 \leq U_1$.

Définition 1.4.4 *La relation de sous-typage $\leq$ est définie selon les règles décrites ci-dessous :*

$$\frac{\Box}{\texttt{bool} \leq \texttt{bool}}\ (\mathsf{Sub-Bool})$$

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad U \leq U'}{(\tau_1 \rightarrow \tau_2, U) \leq (\tau_1' \rightarrow \tau_2', U')}\ (\mathsf{Sub-Fun})$$

$$\frac{U \leq U'}{(\mathbf{R}, U) \leq (\mathbf{R}, U')}\ (\mathsf{Sub-Res})$$

1.4.4 Environnement de typage

L'environnement de typage permet d'associer chaque variable du programme avec son type. Dans la suite de ce chapitre, les variables Γ et Δ dénotent des envorinnements de typage.De plus, l'expression $\emptyset$ désigne un environnement de typage dont le domaine est vide. Si $x \notin dom(\Gamma)$ alors l'expression $\Gamma, x : \tau$ désigne l'environnement de typage Δ tel que : $dom(\Delta) = dom(\Gamma) \cup \{x\}$, $\Delta(x) = \tau$, et $\forall y \in dom(\Gamma) : \Delta(y) = \Gamma(y)$.

Définition 1.4.5 *Soit C un contexte d'usage. Si l'intersection de l'ensemble des variables libres d'usage apparaissant dans τ ou bien dans Γ et l'ensemble des variables d'usage associées aux contexte C est vide, alors $C[\tau]$ et $C[\Gamma]$ sont définis comme suit :*

$$\begin{array}{lcl} C[\texttt{bool}] & = & \texttt{bool} \\ C[(\tau_1 \rightarrow \tau_2, U)] & = & (\tau_1 \rightarrow \tau_2, C[U]) \\ C[(\mathbf{R}, U)] & = & (\mathbf{R}, C[U]) \\ dom(C[\Gamma]) & = & dom(\Gamma) \\ C[\Gamma](x) & = & C[\Gamma(x)] \end{array}$$

Soit **op** *l'opérateur binaire défini sur l'ensemble des usages comme suit : _; _ ou _&_. En appliquant cet opérateur sur les types et l'environnement de typage d'un programme, on obtient :*

$$\begin{array}{rcl} \texttt{bool}\ \mathbf{op}\ \texttt{bool} & = & \texttt{bool} \\ (\tau_1 \rightarrow \tau_2, U_1)\ \mathbf{op}\ (\tau_1 \rightarrow \tau_2, U_2) & = & (\tau_1 \rightarrow \tau_2, U_1\ \mathbf{op}\ U_2) \\ (\mathbf{R}, U_1)\ \mathbf{op}\ (\mathbf{R}, U_2) & = & (\mathbf{R}, U_1\ \mathbf{op}\ U_2) \\ dom(\Gamma_1\ \mathbf{op}\ \Gamma_2) & = & dom(\Gamma_1) \cup dom(\Gamma_2) \\ (\Gamma_1\ \mathbf{op}\ \Gamma_2)(x) & = & \left\{ \begin{array}{ll} \Gamma_1(x)\ \mathbf{op}\ \Gamma_2(x) & si\ x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & si\ x \in dom(\Gamma_1) \backslash dom(\Gamma_2) \\ \Gamma_2(x) & si\ x \in dom(\Gamma_2) \backslash dom(\Gamma_1) \end{array} \right. \end{array}$$

L'environnement de typage $\blacksquare_x \Gamma$ *est défini comme suit :*

$$\blacksquare_x \Gamma \quad = \quad \left\{ \begin{array}{ll} \Gamma & si\ x \notin dom(\Gamma) \\ \Gamma'', x : \blacksquare \tau_x & si\ \Gamma = \Gamma'', x : \tau_x\ et\ \tau_x \neq \texttt{bool} \end{array} \right.$$

On remarque que l'environnement $\blacksquare_x \Gamma$ *est indéfini si* $\Gamma(x) = \texttt{bool}$.

Si $dom(\Gamma_1) \supseteq dom(\Gamma_2), \forall x \in dom(\Gamma_2) : \Gamma_1(x) \leq \Gamma_2(x)$, et $\forall x \in dom(\Gamma_1) \backslash dom(\Gamma_2) : Use(\Gamma_1(x)) \leq \mathbf{0}$, alors $\Gamma_1 \leq \Gamma_2$. Il faut noter que $Use(\Gamma(x))$ est une fonction qui retourne les usages associés à la variable x dans l'environnement de typage Γ.

Exemple Si $\Gamma = x : (\mathbf{R}, U)$ et $\Delta = x : (\mathbf{R}, U'), y : \texttt{bool}$, alors $\Box\Gamma = x : \Box(\mathbf{R}, U) = x : (\mathbf{R}, \Box U)$, et $\Gamma; \Delta = x : (\mathbf{R}, U); (\mathbf{R}, U'), y : \texttt{bool} = x : (\mathbf{R}, U; U'), y : \texttt{bool}$.

1.4.5 Système de typage

Le système de typage manipule des jugements de la forme $\Gamma \vdash M : \tau$. Chaque jugement relate le type et les différents usages des ressources d'un programme. Par exemple,

si $M = (\mathsf{let}\ y = \mathsf{acc}^{l_1}(x)\mathsf{in}\ x)$ alors le jugement $x : (\mathbf{R}, l_1; \Box l_2) \vdash M : (\mathbf{R}, l_2)$ souligne que si x est une ressource pouvant être exploitée selon les usages l_1 puis l_2, alors le terme M est évalué comme étant une ressource associée à l'usage l_2. De plus, il indique que l'usage l_1 se produit durant l'évaluation de M, alors que l'usage l_2 peut être invoqué pendant l'évaluation de M ou bien lors de l'exploitation de M (lors de l'utilisation de la valeur de M) vu que l'usage l_2 est soumis à l'opérateur $\Box$.

Le système de typage étend la syntaxe définie dans la section 1.2 en introduisant le terme $M^{\{x\}}$ qui souligne que la variable x est directement liée au terme M : le tas de valeurs référé par la variable x n'appratient pas à la valeur de M. En d'autres mots, le programme ne procède à l'évaluation de la variable x qu'au moment où il entreprend l'exploitation du terme M. Cette propriété peut être vérifiée en comparant les types de x et M comme étant des variants dans un système de typage linéaire [3]. Aussi, on peut facilement que cette propriété est vérifiée si M est une ressource et x est une fonction : $x^{\{\lambda x.x\}}$.

Le système de typage est défini à partir des règles décrites dans la table 1.4 (page 24). Dans la règle (T-Var), l'opérateur $\Box$ est appliqué au type associé à la variable x dans l'environnement de typage puisque cette dernière ne peut être exploitée qu'après son évaluation. Dans les règles $(\mathsf{T-New})$, (T-Abs), et (T-fix), la prémisse $[[U]] \subseteq \Phi$ vérifie que les ressources ou les fonctions créées sont accessibles selon les traces définies dans Φ (ces traces reflètent l'intention du programmeur pour exploiter une ressource). Dans la règle (T-Abs), la prémisse $\Gamma, x : \tau_1 \vdash M : \tau_2$ indique que les ressources référées par les variables libres introduites dans l'expression $\lambda^{\Phi} x.M$ sont exploitées selon l'environnement Γ à chaque fois que la fonction est appelée. Vu que la fonction elle-même est appelée selon l'usage U, l'environnement total de typage de la fonction $\lambda^{\Phi} x.M$ est obtenu par la multiplication de l'environnement Γ et l'usage U. L'opérateur $\Box$ est appliqué à l'environnement de typage vu que les ressources ne peuvent être exploitées qu'après que la fonction soit appelée.

Dans la règle (T-Fix), le type de $\mathsf{fix}^{\Phi}(f, x, M)$ souligne que la fonction est appelée selon l'usage U_2. Lors de chaque appel, le terme M est évalué et la fonction est rappelée selon l'usage U_1. Aussi, l'usage total de la fonction est représenté par $U_2 \triangleright (\mu\alpha.U_1 \triangleright \alpha)$. Cet usage doit être inclus dans l'ensemble Φ représentant les intentions du programmeurs pour l'exploitation de la fonction récursive. L'environnement total de typage est alors $U_2 \odot (\mu\alpha.(\Gamma \otimes (U_1 \odot \alpha)))$: la fonction est d'abord appelée selon l'usage U_2, puis le terme M va être évalué et les ressources vont être exploitées selon les usages qui leurs sont attribués dans l'environnement Γ ; par la suite, la fonction f va être associée à l'usage U_1 à chaque appel interne. L'opérateur $\Box$ est appliqué à l'environnement de typage vu que les ressources ne sont pas accessibles qu'après l'évaluation du terme $\mathsf{fix}^{\Phi}(f, x, M)$.

Dans la règle (T-App), les prémisses indiquent que les ressources définies dans M_1 et

$$\frac{c = \texttt{true} \vee c = \texttt{false}}{\varnothing \vdash c : \texttt{bool}} \ (\mathsf{T-Const}) \qquad \frac{\square}{x : \square\tau \vdash x : \tau} \ (\mathsf{T-Var})$$

$$\frac{[[U]] \subseteq \Phi}{\varnothing \vdash \texttt{new}^{\Phi}() : (\mathbf{R}, U)} \ (\mathsf{T-New})$$

$$\frac{\Gamma \vdash M : (\mathbf{R}, l)}{\Gamma \vdash \texttt{acc}_i^l(M) : \texttt{bool}} \ (\mathsf{T-Acc}) \qquad \frac{\Gamma \vdash M : \tau}{\blacksquare_x \Gamma \vdash M^{\{x\}} : \tau} \ (\mathsf{T-Now})$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2 \qquad [[U]] \subseteq \Phi}{\square(U \odot \Gamma) \vdash \lambda^{\Phi} x.M : (\tau_1 \to \tau_2, U)} \ (\mathsf{T-Abs})$$

$$\frac{\Gamma_1 \vdash M_1 : (\tau_1 \to \tau_2, l) \qquad \Gamma_2 \vdash M_2 : \tau_1}{\Gamma_1; \Gamma_2 \vdash M_1 @^l M_2 : \tau_2} \ (\mathsf{T-App})$$

$$\frac{\Gamma, f : (\tau_1 \to \tau_2, U_1), x : \tau_1 \vdash M : \tau_2 \qquad [[U_2 \rhd (\mu\alpha.U_1 \rhd \alpha)]] \subseteq \Phi \quad \alpha \text{ est une variable}}{\square(U_2 \odot (\mu\alpha.(\Gamma \otimes (U_1 \odot \alpha)))) \vdash \texttt{fix}^{\Phi}(f, x, M) : (\tau_1 \to \tau_2, U_2)} \ (\mathsf{T-Fix})$$

$$\frac{\Gamma_1 \vdash M_1 : \texttt{bool} \qquad \Gamma_2 \vdash M_2 : \tau_2 \qquad \Gamma_3 \vdash M_3 : \tau_2}{\Gamma_1; (\Gamma_2 \,\&\, \Gamma_3) \vdash \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 : \tau_2} \ (\mathsf{T-If})$$

$$\frac{\Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma_1; \Gamma_2 \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2 : \tau_2} \ (\mathsf{T-Let})$$

$$\frac{\Gamma' \vdash M : \tau' \qquad \Gamma \leq \Gamma' \qquad \tau' \leq \tau}{\Gamma \vdash M : \tau} \ (\mathsf{T-Sub})$$

TABLE 1.4 – Règles de typage.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\square}{x:(\mathbf{R},\square l_1)\vdash x:(\mathbf{R},\square l_1)}\,(\mathsf{T-Var})}{x:(\mathbf{R},\square l_1)\vdash \mathsf{acc}_i^{l_1}(x):\mathsf{bool}}\,(\mathsf{T-Acc})}{x:(\mathbf{R},\blacksquare\square l_1)\vdash \mathsf{acc}_i^{l_1}(x)^{\{x\}}:\mathtt{bool}}\,(\mathsf{T-Now})}{x:(\mathbf{R},l_1)\vdash \mathsf{acc}_i^{l_1}(x)^{\{x\}}:\mathtt{bool}}\,(\mathsf{T-Sub}) \quad \dfrac{\dfrac{\square}{x:(\mathbf{R},\square l_2)\vdash x:(\mathbf{R},l_2)}\,(\mathsf{T-Var})}{x:(\mathbf{R},\square l_2),y:\mathtt{bool}\vdash x:(\mathbf{R},l_2)}\,(\mathsf{T-Sub})}{x:(\mathbf{R},l_1;\square l_2)\vdash \mathsf{let} y=\mathsf{acc}_i^{l_1}(x)^{\{x\}}\ \mathsf{in}\ x:(\mathbf{R},l_2)}\,(\mathsf{T-Let})$$

TABLE 1.5 – Exemple d'un arbre de typage.

M_2 vont être exploitées selon les usages qui leurs sont attribués dans les environnements Γ_1 et Γ_2 respectivement. L'environnement total de typage va être $\Gamma_1;\Gamma_2$ vu que le terme M_1 va être évalué avant le terme M_2. Dans la règle (T-Acc), la primitive acc est associée à l'usage l qui apparaît lors de l'exploitation de M.

Dans la règle (T-If), après que le terme M_1 soit évalué, seul M_2 ou bien M_3 est évaluée. L'environnement total de typage correspond alors à $\Gamma_1;(\Gamma_2\ \&\ \Gamma_3)$. Dans la règle (T-Now), l'usage véhiculé par le type τ peut être réalisé au moment de l'évaluation du terme M. L'opérateur $\blacksquare$ est introduit pour exprimer cette propriété. La règle (T-Sub) introduit une nouvelle description de la relation de sous-typage.

L'extension de la syntaxe des termes par l'introduction de l'expression $M^{\{x\}}$ provoque l'ajout d'une définition d'un nouveau contexte d'évaluation noté $\mathcal{E}^{\{x\}}$ défini par la règle suivante :

$$\dfrac{x\notin \mathrm{portee}(v,H)}{(H,\mathcal{E}[v^{\{x\}}])\rightsquigarrow(H,\mathcal{E}[v])}\,(\mathsf{Sub-ECheck})$$

Où $portee(v,H)$ dénote l'ensemble des variables qui ne sont pas liées à la valeur v dans le tas H.

Le théorème de sûreté peut alors être défini comme suit :

Théorème 1.4.1 *Si* $\varnothing\vdash M:\tau$ *et* $use(\tau)=\mathbf{0}$ *alors* M *est sûr.*

La table 1.5 décrit l'arbre de typage du jugement :

$$x:(\mathbf{R},l_1;\square l_2)\vdash \mathsf{let}\ y=\mathsf{acc}_i^{l_1}(x)^{\{x\}}\ \mathsf{in}\ x:(\mathbf{R},l_2)$$

1.5 Conclusion

Dans ce chapitre, nous avons présenté une approche qui propose un mécanisme de vérification de programmes par l'analyse des ressources. Le modèle proposé détaille le processus d'exploitation de chaque ressource. Aussi, chaque ressource est associée à une trace qui englobe tous les accès possibles à cette dernière au moment de l'exécution du programme. L'analyse de chaque trace va permettre de détecter toutes les séquences dangereuses d'accès à une ressource (un fichier ne peut être lu avant qu'il ne soit ouvert, le contenu d'une variable ne peut être modifié si cette dernière a été vidée, ...). Cette approche introduit aussi un système de typage permettant de vérifier la validité d'un programme. Dans ce cas, un programme est dit *valide* si son arbre de typage est complet et s'il ne comporte aucun accès illicite à une ressource. L'arbre de typage de chaque programme est construit au moyen d'un algorithme *incomplet* décrit dans l'article de référence [2].

Toutefois, cette approche souffre d'un certain nombre de lacunes :

- La relation $[[U]] \subseteq \Phi$ est indécidable : il est impossible de comparer deux traces infinies ;
- L'exploitation simultanée n'est pas prise en considération : certaines structures de données peuvent être soumises à plusieurs accès simultanés alors que l'approche préconise que chaque ressource n'est exploitée que par un seul accès à tout moment de l'exécution du programme ;

Néanmoins, cette approche peut être considérée comme un point de départ pour la définition de nouvelles approches plus complètes.

Chapitre 2

Vérification de la politique sécurité par analyse de programmes

Dans ce chapitre, nous présentons une approche, introduite par Thomas Thorn dans [36], qui se propose de définir un cadre de vérification de propriétés de sécurité globales au moyen d'un ensemble de contrôle locaux. Le but de cette approche est de définir un modèle général pour la sécurité qui repose sur des mécanismes de langages de programmation. Ce modèle se propose de fournir un ensemble d'outils permettant de vérifier statiquement des propriétés de sécurité d'une application donnée. Pour se faire, l'auteur définit les politiques de sécurité comme des ensembles admissibles de traces d'exécution du modèle abstrait, exprimés dans une logique temporelle.

Aussi, nous présenterons dans un premier temps la syntaxe et la sémantique du modèle qui permet d'abstraire un programme sous forme d'un graphe de contrôle. Par la suite, nous introduirons le formalisme permettant de représenter une propriété de sécurité sous forme d'une formule logique. Puis, nous attarderons sur le mécanisme de vérification de propriétés de sécurité qui se base sur un algorithme de vérification de modèle (*model-checking*). Ensuite, nous étudierons un exemple d'une application de commerce électronique de manière à exposer les différentes étapes du mécanisme de vérification.

2.1 Un modèle de programme

L'objectif premier de ce modèle est de définir un cadre formel de sécurité non spécifique à un langage de programmation. Ce modèle permet de s'abstraire des données de se focaliser uniquement sur les contrôles de sécurité et sur les flots de contrôle, c'est-à-dire les procédures (ou méthodes, ou fonctions) appelées pendant l'exécution ainsi que l'ordre

de leurs appels.

Un programme est représenté par un graphe comportant deux types d'arcs : TG dénote l'ensemble des arcs de transfert (définissant la séquentialité) et CG représente l'ensemble des arcs d'appels (reliant les sites d'appels et les points d'entrée potentiels) :

$$G = (G.nodes, G.instr, G.init, TG, CG)$$

Les signatures des composants du graphe G peuvent être définies comme suit :

$$\begin{array}{lll} G.instr & : & G.nodes \rightarrow \{\text{call}, \text{return}, \text{check}(\phi)\} \\ G.init & : & G.nodes \\ TG & : & G.nodes \rightarrow \mathcal{P}(G.nodes) \\ CG & : & G.nodes \rightarrow \mathcal{P}(G.nodes) \end{array}$$

Les nœuds représentent des points du programme tel que $G.init$ définit le point d'entrée de ce dernier. Si $m \in TG(n)$ (resp. $m \in CG(n)$) alors $n \xrightarrow{TG} m$ (resp. $n \xrightarrow{CG} m$). Cette modélisation comporte trois types de nœuds :

- Les appels ordinaires à une procédure (ou méthode, ou fonction) : $G.instr = \text{call} \Rightarrow CG(n) \neq \varnothing$ où n dénote le nœud d'appel.
- Les nœuds return.
- Les nœuds de vérification $check(\phi)$ permettent d'exprimer une politique de sécurité via la propriété de l'état ϕ. Aussi, une exécution d'un programme qui atteint un nœud de vérification s'arrête si la propriété n'est pas satisfaite.

La sémantique opérationnelle du langage est définie au moyen d'un système de transition disposant d'un état qui représente la pile de contrôle :

$$State = G.nodes^*$$

Les nœuds désignent des points du programme, la pile de contrôle est utilisée pour déterminer la suite du calcul lorsqu'un nœud `return` est exécuté. La sémantique opérationnelle d'un programme G peut être définie à partir des règles introduites ci-dessous :

La relation $\vdash$ détermine si un état satisfait une propriété ϕ (voir prochain paragraphe). La sémantique opérationnelle permet d'associer chaque programme G avec un nœud d'entrée $G.init$ et avec un système de transition ($State, \rhd, \langle G.init \rangle$). La sémantique des traces associées à un graphe G peut alors être définie comme un ensemble de suites d'états accessibles à partir de la configuration initiale :

$$[[G]] = \{T \in State^* \mid T_0 = \langle G.init \rangle \text{ et } \forall k < |T| - 1 : T_k \rhd T_{k+1}\}$$

$$\frac{G.instr(n) = \text{call} \qquad n \xrightarrow{CG} m}{cs : n \rhd cs : n : m}$$

$$\frac{G.instr(m) = \text{return} \qquad n \xrightarrow{TG} n'}{cs : n : m \rhd cs : n'}$$

$$\frac{G.instr(n) = \text{check}(\phi) \qquad cs : n \vdash \phi \qquad n \xrightarrow{TG} n'}{cs : n \rhd cs : n'}$$

2.2 Définition formelle des politiques de sécurité

Comme on a pu le remarquer précédemment, la politique de sécurité est décrite via l'instruction check. Cette instruction modélise un mécanisme du langage de programmation qui effectue un contrôle de sécurité à un point donné de l'exécution. Toutefois, il est bien connu qu'une contrainte de sécurité très forte sur un composant logiciel peut être mise en échec à cause de la coopération associant ce dernier avec un autre composant (la communication entre des éléments logiciels peut être perçue comme dangereuse). De plus, une implantation défensive de la politique de sécurité invoquant des vérifications avant chaque appel de procédure serait extrêmement inefficace. Aussi, cette approche se propose de déterminer statiquement les vérifications redondantes pour mettre en œuvre une politique de sécurité efficace. Pour se faire, l'auteur se propose de définir un formalisme de définition de politiques de sécurité comme des propriétés sur les ensembles des traces d'exécution.

2.2.1 Formalisme des politiques de sécurité

Dans cette approche, la sémantique d'un programme est définie par un ensemble dont les éléments sont des suites de piles de nœuds. Aussi, cette approche dénote trois types différents de propriétés :

1. Les propriétés portant sur les nœuds.
2. Les propriétés portant sur les piles de nœuds.
3. Les propriétés portant sur les suites de piles de nœuds.

Les propriétés des états sont définies au moyen d'une logique temporelle dont la syntaxe et la sémantique sont décrites dans la table 2.1 (page 30).

Comme on peut le remarquer ce langage est minimal car il ne comprend que les connecteurs logiques de conjonction et de négation, les opérateurs *next* **X** et *until* **U** in-

$$\phi ::= NP \mid \mathbf{X}\phi \mid \phi_1 \mathbf{U} \phi_2 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid True$$

$$\begin{array}{lll}
S \vdash \phi & \Leftrightarrow & \forall T \in S : \forall s \in T : s \vdash \phi \\
s \vdash NP & \Leftrightarrow & |s| \geq 1 \text{ et } NP(s_0) \\
s \vdash \mathbf{X}\phi & \Leftrightarrow & |s| > 1 \text{ et } s^1 \vdash \phi \\
s \vdash \phi_1 \mathbf{U} \phi_2 & \Leftrightarrow & \forall i < |s| : s^i \vdash \phi_1 \text{ ou } \exists k : s^k \vdash \phi_2 \text{ et } \forall i < k : s^i \vdash \phi_1 \\
s \vdash \neg\phi & \Leftrightarrow & s \nvdash \phi \\
s \vdash \phi_1 \wedge \phi_2 & \Leftrightarrow & s \vdash \phi_1 \text{ et } s \vdash \phi_2 \\
s \vdash True & \Leftrightarrow & true
\end{array}$$

TABLE 2.1 – Langage de définition des politiques de sécurité.

troduits dans [20](table 2.1 à la page 30). Toutefois, les autres connecteurs peuvent être obtenus à partir de ce langage comme par exemple la disjonction :

$$\begin{array}{lll}
s \vdash \phi_1 \vee \phi_2 & \Leftrightarrow & s \vdash \neg(\neg\phi_1 \wedge \neg\phi_2) \\
 & \Leftrightarrow & s \nvdash \neg\phi_1 \wedge \neg\phi_2 \\
 & \Leftrightarrow & s \nvdash \neg\phi_1 \text{ ou } s \nvdash \neg\phi_2 \\
 & \Leftrightarrow & s \vdash \phi_1 \text{ ou } s \vdash \phi_2
\end{array}$$

De plus, les opérateurs dérivés $\mathbf{G}\phi \equiv \phi \mathbf{U} False$ et $\mathbf{F}\phi \equiv \neg(\mathbf{G}(\neg\phi))$ sont introduits et permettent d'exprimer, respectivement, que ϕ est satisfaite par tous les éléments de la suite, et que ϕ est satisfaite par au moins un des éléments de la suite. Dans la table 2.1, $|s|$ dénote la longueur de la suite s, s_i est le i-ème élément de s et s^i représente le suffixe de s à partir de s_i.

D'après la définition sémantique de $\vdash$, cette relation de satisfaction permet d'associer la propriété de sécurité ϕ avec une pile d'exécution s ou bien l'ensemble des traces S. Aussi, l'expression $s \vdash \phi$ est valide si la pile s satisfait la propriété ϕ. De même, l'expression $S \vdash \phi$ exprime le fait que toutes les piles dans toutes les traces de S vérifient ϕ. Ainsi, la définition d'un programme valide peut être énoncée comme suit :

Définition 2.2.1 *Un programme G satisfait une politique de sécurité ϕ si et seulement si $[[G]] \vdash \phi$. On peut alors écrire $G \vDash \phi$.*

2.2.2 Exemples de politiques de sécurité

Le cadre de cette approche est dédié aux politiques structurelles, c'est-à-dire celles définies à partir de la structure d'appel de l'application. Les auteurs assumentrapportent

que ce dernier est suffisamment général pour formaliser un certain nombre de politiques couramment utilisées. Dans cette section, nous allons approfondir certains exemples de politiques de sécurité présentées dans l'article de référence.

La politique "d'isolement du devoir" ("*Segregation of Duty*") : Cette politique est souvent requise dans les applications financières où la sécurité est assurée en imposant qu'une tâche ne peut être effectuée que si certains sujets se portent garants. Les auteurs rassemblent les sujets dans des grands groupes : *Responsable*, *Comptable*, *etc*. Par exemple, si on impose que le code de la catégorie *Critique* ne peut être exécuté que s'il est précédé par un *Responsable* et un *Comptable*. Cette propriété peut être représentée comme suit :

$$SD = \neg Critique \, \mathbf{U} \, (Responsable \wedge Comptable)$$

La sémantique de la logique (table 2.1) indique que cette politique n'est satisfaite que si toutes les piles d'exécution possibles répondent à ces deux propriétés :

1. Aucun nœud satisfaisant la propriété *Critique* n'apparaît avant un nœud qui vérifie la propriété *Responsable*.
2. Aucun nœud satisfaisant la propriété *Critique* n'apparaît avant un nœud qui vérifie la propriété *Comptable*.

En effet, en se référant à la table 2.1 (page 30), on obtient une nouvelle formulation de la politique SD :

$$\begin{aligned} SD \quad = \quad & \forall T \in S : \forall s \in T : \exists i < |s| : s^i \vdash \neg Critique \\ & \text{ou } \exists k : s^k \vdash Resposable \wedge s^k \vdash Comptable \text{ et } \forall i < k : s^i \nvdash Critique \end{aligned}$$

Comme on peut le remarquer, un programme G peut être considéré comme valide s'il satisfait l'une de ces propriétés :

1. Le programme ne comporte aucun nœud satisfaisant la propriété *Critique* (voir première partie de la politique).
2. Il n'existe aucun nœud satisfaisant la propriété *Critique* qui apparaît avant un nœud assurant la propriété *Responsable* ou bien *Comptable*. Autrement dit, chaque nœud vérifiant la propriété *Critique* doit être précédé par un nœud satisfaisant la propriété *Responsable* et un autre vérifiant la propriété *Comptable* (deuxième partie de la politique).

L'inspection de pile : Cette politique est la base du mécanisme de sécurité de *Java Development Kit* (JDK 1.2). Elle consiste en une inspection descendante de la pile de contrôle, imposant qu'une propriété donnée ϕ soit vérifiée à chaque nœud jusqu'à ce que la propriété $Priv$ soit satisfaite. En d'autres mots, cela revient à ignorer les nœuds traversés par le code avant le dernier nœud satisfaisant $Priv$; la propriété ϕ doit alors être vérifiée par tous les nœuds situés au dessus de ce dernier dans la pile d'exécution. Aussi, cette politique peut être définie de manière à forcer le dernier nœud de la pile assurant la propriété $Priv$ à vérifier la propriété ϕ exprimée comme suit :

$$JDK(\phi) = \mathbf{G}((\mathbf{X}(\mathbf{F}\ Priv)) \vee \phi)$$

Nous allons procéder à une analyse ascendante de la politique $JDK(\phi)$, c'est-à-dire que nous allons étudier toutes les expressions composant cette politique :

- $s \vdash \neg Priv\ \mathbf{U}\ False \Leftrightarrow \forall i < |s| : s^i \nvdash Priv$ ou $\exists k : s^k \vdash False$ et $\forall i < k : s^i \nvdash Priv$: cette expression exprime le fait qu'aucun nœud ne doit satisfaire la propriété $Priv$;
- $s \vdash F(Priv) \Leftrightarrow s \vdash \neg(\neg Priv\ \mathbf{U}\ False) \Leftrightarrow s \nvdash \neg Priv\ \mathbf{U}\ False$: l'expression suivante assure qu'il existe au moins un nœud dans la pile qui satisfait la propriété $Priv$;
- $s \vdash \mathbf{X}(\mathbf{F}(Priv)) \Leftrightarrow |s| > 1 : s^1 \vdash \mathbf{F}(Priv)$: cette expression ne peut être valide sauf si tous les éléments de la pile s vérifient la propriété $\mathbf{F}(Priv)$;
- $s \vdash \mathbf{X}(\mathbf{F}(Priv)) \vee \phi \Leftrightarrow s \vdash \mathbf{X}(\mathbf{F}(Priv))$ ou $s \vdash \phi$: cette expression assure que tous les nœuds de la piles doivent satisfaire la propriété ϕ ou bien $\mathbf{X}(\mathbf{F}(Priv))$;
- $s \vdash \mathbf{G}((\mathbf{X}(\mathbf{F}\ Priv)) \vee \phi) \Leftrightarrow s \vdash (\mathbf{X}(\mathbf{F}(Priv)) \vee \phi)\mathbf{U}\ False$: dans ce cas, une pile s est dite valide si tous ses nœuds vérifient la propriété $(\mathbf{X}(\mathbf{F}(Priv)) \vee \phi)$. En effet, selon les règles sémantiques définies dans la table 2.1, cette expression peut être transformée comme suit :
  $\forall i < |s| : s^i \vdash \mathbf{X}(\mathbf{F}(Priv)) \vee \phi$ ou $\exists k : s^k \vdash False$ et $\forall i < k : s^i \vdash \mathbf{X}(\mathbf{F}(Priv)) \vee \phi$

Aussi, on peut conclure qu'un programme G ne peut être valide que s'il satisfait la propriété : $G \vDash JDK(\phi)$. Autrement dit, pour toute pile s et pour tout nœud n dans s, soit $Priv$ est assurée par un nœud qui suit n dans s (propriété $\mathbf{X}(\mathbf{F}(Priv))$), soit n vérifie ϕ.

2.2.3 Vérification à états finis

La conception d'une méthode automatique de vérification n'est pas directe car la sémantique opérationnelle d'un programme peut conduire à un système de transition à nombre infini d'états. Dans cette approche, l'infini provient de la récursivité dans les programmes qui peut engendrer des piles d'exécution de taille non bornée. Une autre source

de complexité est due à l'instruction check dont l'effet est d'interdire certaines traces d'exécution. Aussi, les auteurs proposent une technique pour réduire un système de transition infini à un système fini équivalent pour une propriété de sécurité donnée tels que les états de ce dernier constituent un sous-ensemble des états du système de départ. Ces états correspondent à ceux du système d'origine dont la taille est inférieure au seuil dérivé à partir du programme et de la propriété de sécurité. Ainsi, la procédure de vérification peut être construit à partir d'un algorithme de vérification de modèle (*model checking*).

La procédure de réduction se base sur le concept de complexité des propriétés de sécurité et de programmes :

Définition 2.2.2 *La complexité $\mathcal{C}(\phi)$ d'une propriété ϕ est définie par :*

$$\begin{array}{lcl} \mathcal{C}(NP) & = & 1 \\ \mathcal{C}(\mathbf{X}(\phi)) & = & 1 + \mathcal{C}(\phi) \\ \mathcal{C}(\phi_1 \ \mathbf{U} \ \phi_2) & = & max(\mathcal{C}(\phi_1), \mathcal{C}(\phi_2)) \\ \mathcal{C}(\neg\phi) & = & \mathcal{C}(\phi) \\ \mathcal{C}(\phi_1 \wedge \phi_2) & = & max(\mathcal{C}(\phi_1), \mathcal{C}(\phi_2)) \\ \mathcal{C}(True) & = & 0 \end{array}$$

La complexité est donc calculée comme étant le nombre maximal d'apparition de l'opérateur $\mathbf{X}$ dans une formule ϕ.

Définition 2.2.3 *La complexité d'un programme G est définie par :*

$$\mathcal{C}(G) = max\{\mathcal{C}(\phi) \mid n \in G.nodes, G.instr(n) = \mathrm{check}(\phi)\}$$

La complexité d'un programme est définie donc comme étant le maximum des complexités des propriétés apparaissant dans les instructions check.

Le seuil délimitant le nouveau système d'états est défini comme étant le nombre d'éléments contigus identiques dans la pile qui représente la *mesure* de cette dernière :

Définition 2.2.4 *Soit s une suite de nœuds. La mesure de s est définie comme suit :*

$$\mathcal{M}(s) = max\{k \mid \exists i : \forall j \in [0, k-1] : s_i = s_{i+j}\}$$

En d'autres, une mesure d'une suite α est le nombre maximum de nœuds définissant une séquence β tel que : $\alpha = \beta^*$.

Afin de définir le nouveau système de transition limité, cette approche introduit le concept de *sémantique limitée* :

Définition 2.2.5 *La sémantique p-limitée d'un programme G est définie comme suit :*

$$\begin{array}{rcl} [[G]]_p & = & \{T \in States^* \mid T_0 = \langle G.init \rangle \ \text{et} \ \forall k < |T| - 1 : T_k \rhd_p T_{k+1}\} \\ & & \text{avec} \qquad s \rhd s^{'} \Leftrightarrow s \rhd s^{'} \ \text{et} \ \mathcal{M}(s^{'}) \leq p \end{array}$$

La sémantique *p-limitée* d'un programme est définie de la même manière que celle décrite dans la table 2.1 sauf qu'elle est basée sur un système de transition p-limité, noté $\rhd_p$, pouvant arrêter un programme quand la mesure de la pile dépasse le seuil p, tel que :

Définition 2.2.6 $G \vDash_p \phi \Leftrightarrow [[G]]_p \vdash \phi$

Théorème 2.2.1 *Si $p \geq \mathcal{C}(G)$ et $p \geq \mathcal{C}(\phi)$ alors $G \vDash \phi \Leftrightarrow G \vDash_p \phi$*

Ce théorème montre qu'il est possible d'assurer la vérification d'une propriété sur un système de transition *p-limitée* sans aucune perte de précision, à condition que p soit supérieur ou égal aux complexités de la propriété et du programme. On obtient alors un système de transition fini pouvant être analysé par des algorithmes de vérification de modèle. En effet, la sémantique *p-limitée* permet d'éliminer les nœuds relatifs au processus d'itération. Autrement dit, il est inutile d'analyser tous les nœuds appartenant à un processus récursif vu qu'ils reflètent le même flux de contrôle (le processus exécute les mêmes instructions pendant un certain nombre d'itérations).

Par exemple, si on voulait comparer les propriétés $\phi_1 \ \mathbf{U} \ \phi_2$ et $\phi_1 \ \mathbf{U} \ (\mathbf{X}(\phi_2))$ selon les les propriétés de base ϕ_1 et ϕ_2 sur les nœuds n_1, n_2 et n_3 qui satisfont respectivement les conditions : $\phi_1 \wedge \neg\phi_2$, $\neg\phi_1 \wedge \neg\phi_2$ et $\phi_1 \wedge \phi_2$. Alors, la propriété $\phi_1 \ \mathbf{U} \ (\mathbf{X}(\phi_2))$ est satisfaite pour la suite $n_1 : n_2 : n_3$, mais elle n'est pas vérifiée pour $n_1 : n_2 : n_2 : n_3$. En effet, cette propriété ne peut être valide que si tous les nœuds vérifient la propriété ϕ_1 ou bien s'il existe un nœud tels que tous les nœuds qui le précédent satisfassent la propriété ϕ_1 et tous les nœuds qui le suivent vérifient la propriété ϕ_2. On remarque que la séquence $n_1 : n_2 : n_2 : n_3$ ne comporte aucun nœud répondant aux conditions précédentes ; alors que dans la séquence $n_1 : n_2 : n_3$, le nœud n_2 assure la deuxième condition. La propriété $\phi_1 \ \mathbf{U} \ \phi_2$ ne peut être satisfaite que si tous les nœuds vérifient la propriété ϕ_1 ou bien s'il existe un nœud vérifiant la propriété ϕ_2 tels que tous ses successeurs assurent la propriété ϕ_2 et tous ses prédécesseurs vérifient la propriété ϕ_1. Ces conditions ne sont pas vérifiées par les séquences de nœuds précédentes. Il n'est donc pas suffisant de vérifier $\phi_1 \ \mathbf{U} \ (\mathbf{X}(\phi_2))$ sur des suites de mesure égale à 1. Cela est en accord avec la définition de la complexité car $\mathcal{C}(\phi_1 \ \mathbf{U} \ (\mathbf{X}(\phi_2))) = 2$ et $\mathcal{C}(\phi_1 \ \mathbf{U} \ \phi_2) = 1$.

Pour démontrer le théorème précédent, les auteurs introduisent la notion de *p-réduction*, notée $\rightsquigarrow_p$, telle que l'expression $s \rightsquigarrow_p s^{'}$ ne peut être valide que si $s^{'}$ peut être obtenue à partir de s en éliminant un élément d'une suite de $p+1$ éléments identiques et contigus.

Définition 2.2.7 *La notion de p-réduction peut être définie comme suit :*

$$s \rightsquigarrow_p s^{'} \quad \Leftrightarrow \quad \begin{array}{l} s = s_0 : ... : s_k : ... : s_{k+p} : ... : s_n \\ s^{'} = s_0 : ... : s_k : ... : s_{k+p-1} : ... : s_n \end{array}$$

avec $\forall i \in [0,p] : s_{k+i} = s_k$

Lemme 2.2.1 *Le lemme principal pour la preuve du théorème est énoncé comme suit :*

$$(p \geq \mathcal{C}(\phi) \textit{ et } s \rightsquigarrow_p^* s^{'}) \Rightarrow (s \vdash \phi \Leftrightarrow s^{'} \vdash \phi)$$

Ce lemme démontre que pour prouver qu'une suite s satisfait une propriété ϕ, il suffit de vérifier que la propriété est valide sur une suite réduite $s^{'}$ qui ne contient aucune sous-suite de plus de p éléments contigus et que p soit supérieur ou égal à la complexité de cette propriété. Ce lemme peut être utilisé pour établir une relation entre la sémantique standard et une sémantique *p-limitée* comme suit [1] :

Lemme 2.2.2 *Pour tout T, s et $s^{'}$,*

$$T \in [[G]], s \rightsquigarrow_p^* s^{'} \textit{ et } p \geq \mathcal{C}(G) \textit{ et } p \geq \mathcal{M}(s^{'}) \Rightarrow \exists T^{'} \in [[G]]_p : s^{'} \in T^{'}$$

Lemme 2.2.3 $[[G]]_p \subseteq [[G]]$

2.3 Exemple de commerce électronique

Dans cette section, nous allons présenter l'exemple introduit par les auteurs pour illustrer l'application de cette approche dans le domaine du commerce électronique. Cet exemple admet quatre domaines de protection correspondant à quatre sujets :

- Le système (domaine de protection *Système*) manipule les variables réelles protégées (voir table 2.2). La classe englobant ces variables propose des méthodes de lecture (read) et d'écriture (write), protégées par un contrôle de permission.
- Le système comporte aussi une application client (domaine de protection *Client*) qui interagit avec le gestionnaire de compte (voir table 2.3). Le client dispose des droits Debit et Canpay.

1. Les preuves de ces lemmes sont décrites dans l'article de référence [36].

```
public class ControlledVar {
        private float var;
        void write(float new){
                AccessController.checkPermission(Write);
                var = new;
        }
        float read(){
                AccessController.checkPermission(Read);
                return var;
        }
}
```

TABLE 2.2 – Code système (domaine *Système*).

```
public void spender(){
        float spend = ...;
        if(account.canpay(spend)){
                account.debit(spend);
        }
        spender();
}
```

TABLE 2.3 – Code de l'application (domaine *Client*).

- L'application non certifiée (domaine de protection *Inconnu*) illustre la gestion des violations des permissions. Elle ne possède aucune permission et tente d'effectuer des opérations non autorisées (voir table 2.4).
- Le fournisseur de service (domaine de protection *Fournisseur*) s'appuie sur la notion de variable protégée pour construire un gestionnaire de compte qui offre une méthode debit pour les retraits et une requête booléenne canpay (voir table 2.5). De plus, le fournisseur dispose des droits Write, Read, Debit et Canpay. Les clients du service ne possèdent pas les permissions Read et Write, ils ne peuvent pas accéder directement à une variable protégée ni en lecture ni en écriture ; toutefois, ils peuvent faire appel aux méthodes debit et canpay du service qui utilisent les méthodes read et write en mode privilégié.

Représentation de l'exemple À partir du code de l'exemple, le graphe de contrôle G_{EC} peut être dérivé selon l'algorithme défini dans la section précédente (voir figure 2.1). Chaque cadre contient tous les nœuds correspondant à des instructions d'une même

```
public void clyde(){
    account.debit(5000000);
    clyde();
}
```

TABLE 2.4 – Code non certifié (domaine *Inconnu*).

méthode. Les cadres sont colorés selon les domaines de protection des codes qu'ils admettent. Les arcs en pointillés sont des arcs de transfert (TG), alors que les arcs pleins sont des arcs d'appels (CG) obtenus par une analyse des classes. Les nœuds encerclés correspondent au code exécuté en mode privilégié.

Les nœuds du graphe peuvent être décomposés selon leurs domaines de protection :

Client	=	n_3, n_4, n_5
Fournisseur	=	$n_8, ..., n_{15}$
Système	=	$n_1, n_2, n_{16}, n_{17}, n_{18}, n_{19}$
Inconnu	=	n_6, n_7

Tels que :

- n_1 : correspond à l'instanciation d'un objet "Client".
- n_2 : détermine l'instanciation d'un objet "Inconnu".
- n_3 : correspond à l'instruction acount.canpay(spend).
- n_4 : est relatif à l'instruction account.debit(spend).
- n_5 : dénote l'instruction spender() dans le code de l'application.
- n_6 : correspond à l'instruction account.debit(5000000).
- n_7 : est relatif à l'instruction clyde().
- n_8 : dénote l'instruction AccessController.checkPermission(Canpay).
- n_9 : correspond à l'instruction balance.read() figurant dans la méthode canpay.
- n_{10} : est associé à l'instruction return figurant dans la méthode canpay.
- n_{11} : dénote l'instruction AccessController.checkPermission(Debit).
- n_{12} : correspond à l'instruction this.canpay(amount) de la méthode debit.
- n_{13} : est associé à l'instruction balance.read() de la méthode debit.
- n_{14} : dénote l'instruction balance.write(...) de la méthode debit.
- n_{15} : dénote l'instruction return associé au type void de la méthode debit. [2]
- n_{16} : correspond à l'instruction AccessController.checkPermission(Read).
- n_{17} : dénote l'instruction return var de la méthode read.
- n_{18} : correspond à l'instruction AccessController.checkPermission(Write).
- n_{19} : dénote l'instruction implicite return de la méthode write.

```
public class AccountMan {
    private ControlleVar balance;
    public boolean canpay(float amount){
        AccessController.checkPermission(Canpay);
        boolean res = false;
        try{
            AccessController.beginPrivileged();
            res = balance.read() > amount;
        }finally{
            AccessController.endPrivilege();
        }
        return res;
    }

    public void debit(float amount){
        AccessController.checkPermission(Debit);
        if(this.canpay(amount)){
            try{
                AccessController.beginPrivileged();
                balance.write(balance.read() − amount);
            }finally{
                AccessController.endPrivilege();
            }
        }else ...
    }
}
```

TABLE 2.5 – Code du gestionnaire de comptes (domaine *Fournisseur*).

Chaque nœud dans le graphe G_{EC} possède une propriété relative à son domaine de protection et la propriété *Priv* si l'appel apparaît dans une section privilégiée. Afin de spécifier les domaines de protection, les auteurs utilisent les notations suivante : un domaine de protection Java "Dom" est dénoté par la propriété D_{Dom}, une permission Java "$Perm$" est exprimée par la propriété P_{Perm} et E_{meth} représentant une propriété qui ne peut être valide que seulement pour les nœuds de la méthode Java $meth$. Les domaines de protection peuvent être notés comme suit :

$$
\begin{aligned}
D_{Client} &= P_{Debit} \wedge P_{Canpay} \\
D_{Fournisseur} &= P_{Debit} \wedge P_{Canpay} \wedge P_{Read} \wedge P_{Write} \\
D_{Systeme} &= P_{Debit} \wedge P_{Canpay} \wedge P_{Read} \wedge P_{Write} \\
D_{Inconnu} &= true
\end{aligned}
$$

De plus, les nœuds n_9, n_{13} et n_{14} sont privilégiés, c'est-à-dire quvils satisfont la propriété $Priv$.

2.4 Conclusion

Dans ce chapitre, nous avons présenté une approche permettant de modéliser un programme sous forme d'un graphe afin de le soumettre à des propriétés de sécurité introduites sous forme de formules d'une logique temporelle. Cette approche a été appliquée comme un cadre de vérification statique pour la pile d'inspection du langage JAVA. Elle fut la première méthode d'analyse statique employée dans la vérification de propriétés de sécurité globale [36].

Toutefois, le mécanisme d'analyse ne procède qu'à la vérification du graphe de contrôle par rapport aux propriétés de sécurité. Ce dernier aspect renferme une lacune envers l'exploitation des ressources dans un programme informatique. Il est donc impossible qu'une valeur de type *entier* soit affectée à une variable dont le type est *fichier*. De plus, cette approche ne définit aucun outil permettant d'assurer qu'une fonction ne comporte pas de codes malicieux.

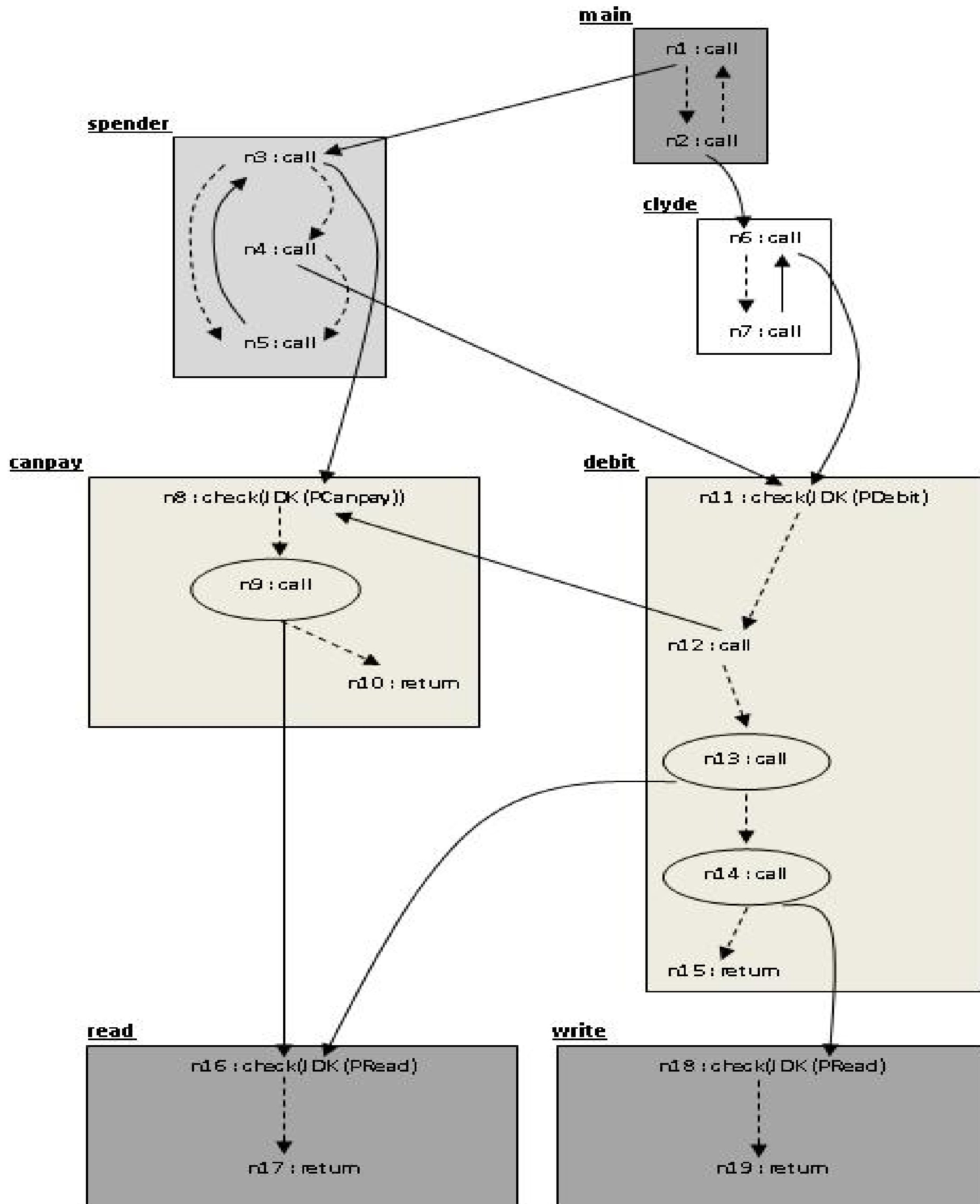

FIGURE 2.1 – Graphe dérivé G_{EC}.

Chapitre 3

Historique d’effets et vérification

La première approche réalisée par Skalka introduit le concept d’*historique d’effets* comme un outil pouvant représenter un modèle d’un programme impératif [4, 6]. Un historique d’effets peut être défini comme l’ensemble des événements produits par un programme durant son exécution. Les événements sont insérés dans l’historique selon l’ordre d’apparition durant l’exécution du programme. L’historique des effets va alors être traduit sous forme de formules de la logique temporelle grâce à un système de transitions étiquetées (*Labelled Transition System : LTS*). Cette transformation permet de vérifier par évaluation la validité d’un programme en utilisant le mécanisme de vérification par évaluation de modèles.

Dans ce chapitre, nous présentons l’architecture générale de cette approche [4, 6]. Tout d’abord, nous introduirons la syntaxe et la sémantique du langage λ_{-hist} sur lequel est construit le mondèle de programme. Par la suite, nous présenterons le système de typage permettant d’assurer la validité d’un historique d’effets envers le programme qu’il abstrait. Puis, nous nous attarderons sur le mécanisme de vérification d’un historique par rapport à une propriété de sécurité exprimée sous forme d’une formule logique modale. Ensuite, nous exposerons une variation du modèle précédent permettant de vérifier des propriétés d’exécution au moment de la compilation d’un programme. Enfin, nous conclurons par l’identification des faiblesses de cette approche.

3.1 Langage λ_{hist}

Dans cette section, nous définissons la syntaxe, la sémantique, et les propriétés de sûreté sur lesquels est construit le modèle de langages λ_{hist}.

c	$\in$	$\mathcal{C}$	Constantes
b	$::=$	$\mathit{true} \mid \mathit{false}$	Valeurs Booléennes
v	$::=$	$x \mid \lambda_z x.e \mid c \mid b \mid \neg \mid \vee \mid \wedge$	Valeurs
e	$::=$	$v \mid e\ e \mid ev(e) \mid \phi(e) \mid \text{if } e \text{ then } e \text{ else } e \mid \text{let } x = v \text{ in } e$	Expressions
η	$::=$	$\epsilon \mid ev(c) \mid \eta; \eta$	Historique
E	$::=$	$[] \mid vE \mid Ee \mid ev(E) \mid \phi(E) \mid \text{if } E \text{ then } e \text{ else } e$	Contexte d'évaluation

TABLE 3.1 – Syntaxe du langage λ_{hist}.

3.1.1 Syntaxe

Le langage λ_{hist} peut être considéré comme une extension du langage λ-*calcul* pouvant représenter le concept d'*historique d'effets*. La syntaxe de ce langage est définie dans la table 3.1 (page 42).

Les valeurs de base correspondent aux valeurs booléennes et la valeur unitaire $()$. Les expressions $\lambda_z x.e$ dénotent les fonctions récursives admettant z comme la seule variable libre. Les programmes polymorphes sont introduits par l'intermédiaire de l'expression `let` $x = v$ `in` e. De plus, le langage admet les équivalences suivantes :

$$(e_1 \wedge e_2) \triangleq (\wedge e_1 e_2) \qquad (e_1 \vee e_2) \triangleq (\vee e_1 e_2) \qquad (\lambda x.e) \triangleq (\lambda_z x.e) \quad (\{z\} \cap fv(e) = \varnothing)$$
$$(\lambda_.e) \triangleq (\lambda x.e) \quad (\{x\} \cap fv(e)^{1} = \varnothing) \qquad (e_1; e_2) \triangleq ((\lambda_.e_2)(e_1))$$

Les événements ev sont toujours paramétrés par une constante $c \in \mathcal{C}$. Un historique η représente une séquence d'événements organisée selon l'ordre d'apparition de ces derniers au moment de l'exécution du programme. L'expression $\hat{\eta}$ dénote la chaîne obtenue à partir de l'historique η en éliminant le symbole «;». Les assertions ϕ permettent de définir l'historique des vérifications, c'est-à-dire, chaque assertion permet de contrôler si l'insertion d'un événement dans l'historique η préserve la cohérence de ce dernier (voir la règle "*check*" dans la table 3.2, page 43).

3.1.2 Sémantique

La sémantique opérationnelle du langage λ_{hist} est décrite dans la table 3.2. Les relations de réduction $\rightarrow$ et $\rightsquigarrow$ sont définies sur les configurations (η, e) ($\rightarrow^*$ représente

1. $fv(e)$ représente l'ensemble des variables libres dans l'expression e.

$\eta, (\lambda_z x.e)v \rightsquigarrow \eta, e[v/x][\lambda_z x.e/z]$		(β)
$\eta, \neg \mathit{true} \rightsquigarrow \eta, \text{false}$		(notT)
$\eta, \neg \mathit{false} \rightsquigarrow \eta, \text{true}$		(notF)
$\eta, \wedge \mathit{true} \rightsquigarrow \eta, \lambda x.x$		(andT)
$\eta, \wedge \mathit{false} \rightsquigarrow \eta, \lambda_.false$		(andF)
$\eta, \vee \mathit{true} \rightsquigarrow \eta, \lambda_.true$		(orT)
$\eta, \vee \mathit{false} \rightsquigarrow \eta, \lambda x.x$		(orF)
$\eta, \mathsf{if}\ \mathit{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow \eta, e_1$		(ifT)
$\eta, \mathsf{if}\ \mathit{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow \eta, e_2$		(ifF)
$\eta, \mathsf{let}\ x = v\ \mathsf{in}\ e \rightsquigarrow \eta, e[v/x]$		(let)
$\eta, ev(c) \rightsquigarrow \eta; ev(c), ()$		(event)
$\eta, \phi(c) \rightsquigarrow \eta; ev_\phi(c), ()$	si $\Pi(\phi(c), \hat{\eta} ev_\phi(c))$	(check)
$\eta, E[e] \rightarrow \eta', E[e']$	si $\eta, e \rightsquigarrow \eta', e'$	(context)

TABLE 3.2 – Sémantique du langage λ_{hist}.

la fermeture réflexive et transitive de la relation $\rightarrow$). La règle *event* assure que chaque événement $ev(c)$ rencontré au cours de l'exécution peut être inséré à la séquence η pour définir une nouvelle trace $(\eta;\ ev(c))$. La fonction Π est introduite dans la règle sémantique $(check)$ afin de vérifier la validité d'un historique η telle que $\Pi(\phi(c), \hat{\eta})$ ne peut être valide que si l'historique η vérifie l'assertion $\phi(c)$. Autrement dit, la règle $check$ permet de spécifier les états d'exécution pouvant contenir la configuration (η, ϕ) : l'événement $ev(c)_\phi$ est inséré dans l'historique η si l'assertion $\phi(c)$ est assurée par η selon la sémantique de la fonction Π. On reviendra plus en détails sur la définition sur la fonction de vérification Π dans la section 3.4.

Exemple Par exemple, soit la fonction f définie comme suit :

$$f \triangleq \lambda_z x.\mathsf{if}\ x\ \mathsf{then}\ ev_1(c)\ \mathsf{else}\ (ev_2(c);\ z(\mathit{true}))$$

En appliquant les règles sémantiques précédentes, on obtient :

$$\begin{array}{ll}
& (\epsilon,\ f(\mathit{false})) \triangleq (\epsilon,\ \lambda_z x.\mathsf{if}\ x\ \mathsf{then}\ ev_1(c)\ \mathsf{else}\ (ev_2(c);\ z(\mathit{true}))\mathit{false}) \\
\xrightarrow{\beta} & (\epsilon, \mathsf{if}\ \mathit{false}\ \mathsf{then}\ ev_1(c)\ \mathsf{else}\ (ev_2(c);\ f(\mathit{true}))) \\
\xrightarrow{ifF} & (\epsilon,\ ev_2(c),\ f(\mathit{true})) \\
\xrightarrow{event} & (\epsilon;\ ev_2(c),\ f(\mathit{true})) \triangleq (\epsilon;\ ev_2(c),\ \lambda_z x.\ \mathsf{if}\ x\ \mathsf{then}\ ev_1(c)\ \mathsf{else}\ (ev_2(c);\ z(\mathit{true}))\mathit{true}) \\
\xrightarrow{ifT} & (\epsilon;\ ev_2(c),\ ev_1(c)) \\
\xrightarrow{event} & (\epsilon;\ ev_2(c);\ ev_1(c),\ ())
\end{array}$$

$\alpha \in \mathcal{V}_s,\ t \in \mathcal{V}_r,\ h \in \mathcal{V}_H,\ \beta \in \mathcal{V}_s \cup \mathcal{V}_r \cup \mathcal{V}_H$	Variables
$s\ ::=\ \alpha \mid c$	Singletons
$\tau\ ::=\ t \mid \{s\} \mid \tau \xrightarrow{H} \tau \mid b \mid unit$	Types
$\sigma\ ::=\ \forall \bar{\beta}.\tau$	Schémas de types
$H\ ::=\ \epsilon \mid h \mid ev(s) \mid H;H \mid H \vert H \mid \mu h.H$	Historique d'effets
$\Gamma\ ::=\ \varnothing \mid \Gamma; x:\sigma$	Environnement

TABLE 3.3 – Syntaxe du système de typage de λ_{hist}.

C'est à dire : $(\epsilon,\ f(false)) \rightarrow^* \epsilon;\ ev_2(c);\ ev_1(c),\ ()$

La propriété de sûreté peut alors être définie comme suit :

Théorème 3.1.1 (Sûreté) *Une configuration (η, e) est dite erronée si et seulement si e n'est pas une valeur (value) et qu'il n'existe aucune autre configuration $(\eta^{'}, e^{'})$ telle que : $(\eta, e) \rightarrow (\eta^{'}, e^{'})$. Si $(\epsilon, e) \rightarrow^* (\eta^{'}, e^{'})$ et que $(\eta^{'}, e^{'})$ est erronée alors l'expression e est dite incorrecte.*

3.2 Système de typage

L'idée générale de cette approche est de construire un historique d'effets H pouvant simuler la séquence d'exécution η [2] de manière que : $\eta \in H$. Autrement dit, l'approche va construire un ensemble d'historiques d'effets H contenant tous les scénarios possibles de l'exécution d'un programme, y compris l'exécution réelle η, puis vérifie que cet historique n'est pas défaillant (si l'ensemble est valide alors chacun de ses éléments l'est aussi). Pour se faire, un ensemble de règles de typage a été construit à partir de la syntaxe décrite dans la table 3.3.

Un historique d'effets peut être un événement $ev(c)$, une concaténation de deux historiques H_1 ;H_2, un choix non-déterministe entre deux historiques $H_1|H_2$ ou bien l'historique $\mu h.H$ définissant l'ensemble des historiques engendrés à partir d'une fonction récursive. Le type $\tau_1 \xrightarrow{H} \tau_2$ dénote que l'historique d'effets H est engendré au moment de l'évaluation de la fonction qui prend un argument de type τ_1 et retourne un résultat de type τ_2.

Afin de préserver la cohérence du système de typage, cette approche préconise que chaque constante c ne peut être associée qu'au type *singleton*. De plus, si on se rapporte

2. η comporte tous les événements produits au moment de l'exécution réelle d'un programme.

à la syntaxe du langage λ_{hist} (table 3.1), on peut remarquer que les événements et les assertions ne peuvent admettre qu'un seul paramètre c ($c \in \mathcal{C}$). En effet, et comme le souligne la première expression de la table 3.3, le système de typage accepte trois sortes de variables de type : les variables régulières ($\mathcal{V}_t$), les singletons ($\mathcal{V}_s$) et les historiques d'effets ($\mathcal{V}_H$). L'expression $\forall\bar{\beta}.\tau$ indique que les variables β vont être associées au type τ ($\beta \in \mathcal{V}_s \cup \mathcal{V}_r \cup \mathcal{V}_H$) dans le cas où $\bar{\beta} \cap fv(\tau) = \varnothing$[3].

Le système de typage manipule des jugements de la forme $\Gamma,\ H \vdash e : \tau$, où Γ dénote l'environnement de typage[4], H représente l'historique d'effets pouvant être produits au moment de l'exécution de e. Les règles de typage sont définies dans la table 3.4 (page 46).

La table 3.5 (page 47) présente l'arbre de dérivation démontrant que : $\varnothing, \epsilon \vdash f : bool \xrightarrow{H} unit$ étant donné un historique d'effet H et un environnement Γ :

$$\begin{array}{lcl} H & \triangleq & \mu h.(ev_1(c)|ev_2(c)); h \\ \Gamma & \triangleq & x : bool,\ z : bool \xrightarrow{H} unit \end{array}$$

3.3 Interprétation des historiques d'effets

Chaque historique H peut être interprété sous forme d'un ensemble de *traces*. Chaque trace représente un scénario d'exécution d'un programme : elle peut être infinie vu que l'exécution du programme peut l'être aussi, sinon elle se termine par le symbole $\downarrow$. Pour se faire, l'historique d'effets est traduit sous forme d'un système de transition étiqueté (*Labelled Transition System : LTS*) construit à partir de l'alphabet suivant :

$$a \quad ::= \quad ev(c) \mid \epsilon \mid \downarrow$$

L'ensemble de ces traces peut être construit à partir d'un historique d'effets en considérant ce dernier comme un programme évoluant dans un environnement non-déterministe. La relation de transition manipulant des historiques d'effets peut être définie via les règles suivantes :

3. Il faut noter que l'expression $[\bar{\tau}/\bar{\beta}]$ spécifie que chaque variable appartenant à l'ensemle β est associée au type τ.

4. Il comporte l'ensemble des couples (x, τ_x) où x dénote une variable du programme et τ_x son type.

$$\frac{\square}{\Gamma,\ \epsilon\ \vdash\ b : bool}\ \textsf{(Bool)} \qquad \frac{\square}{\Gamma,\ \epsilon\ \vdash\ \neg : bool \xrightarrow{\epsilon} bool}\ \textsf{(Not)}$$

$$\frac{\square}{\Gamma,\ \epsilon\ \vdash\ \wedge : bool \xrightarrow{\epsilon} bool \xrightarrow{\epsilon} bool}\ \textsf{(And)}$$

$$\frac{\square}{\Gamma,\ \epsilon\ \vdash\ \vee : bool \xrightarrow{\epsilon} bool \xrightarrow{\epsilon} bool}\ \textsf{(Or)}$$

$$\frac{\square}{\Gamma,\ \epsilon\ \vdash\ () : unit}\ \textsf{(Unit)} \qquad \frac{\square}{\Gamma,\ \epsilon\ \vdash\ c : \{c\}}\ \textsf{(Const)}$$

$$\frac{\Gamma(x)\ =\ \forall\bar{\beta}.\tau}{\Gamma,\ \epsilon\ \vdash\ x : \tau[\bar{\tau}/\bar{\beta}]}\ \textsf{(Var)}$$

$$\frac{\Gamma,\ H\ \vdash\ e : \{s\}}{\Gamma,\ H;\ ev(s) \vdash\ ev(e) : unit}\ \textsf{(Event)}$$

$$\frac{\Gamma,\ H\ \vdash\ e : \{s\}}{\Gamma,\ H;\ ev_{\phi}(s) \vdash\ \phi(e) : unit}\ \textsf{(Check)}$$

$$\frac{\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{H} \tau_2,\ H\ \vdash\ e : \tau_2}{\Gamma,\ \epsilon\ \vdash\ \lambda_z x.e : \tau_1 \xrightarrow{H} \tau_2}\ \textsf{(Abs)} \qquad \frac{\Gamma,\ H\ \vdash\ e : \tau}{\Gamma,\ H|H'\ \vdash\ e : \tau}\ \textsf{(Weaken)}$$

$$\frac{\Gamma,\ H_1\ \vdash\ e_1 : bool \qquad \Gamma,\ H_2\ \vdash\ e_2 : \tau \qquad \Gamma,\ H_2\ \vdash\ e_3 : \tau}{\Gamma,\ H_1; H_2\ \vdash\ \textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 : \tau}\ \textsf{(If)}$$

$$\frac{\Gamma, \epsilon \vdash\ v : \tau_1 \qquad \bar{\beta}\ \cap\ fv(\Gamma) = \varnothing \qquad \Gamma; x : \forall\bar{\beta}.\tau_1,\ H\ \vdash\ e : \tau_2}{\Gamma,\ H\ \vdash\ \textsf{let}\ x = v\ \textsf{in}\ e : \tau_2}\ \textsf{(Let)}$$

$$\frac{\Gamma,\ H_1\ \vdash\ e_1 : \tau_1 \xrightarrow{H_3} \tau_2 \qquad \Gamma,\ H_2\ \vdash\ e_2 : \tau_1}{\Gamma,\ H_1; H_2; H_3\ \vdash\ e_1 e_2 : \tau_2}\ \textsf{(App)}$$

TABLE 3.4 – Règles de typage de λ_{hist}.

$$\dfrac{\dfrac{\dfrac{\Gamma(x) = bool}{\Gamma,\ H \vdash x : bool}\ (\mathsf{Var}) \quad \dfrac{A}{\Gamma,\ H\ \vdash\ ev_1(c) : unit} \quad \dfrac{B}{\Gamma,\ H\ \vdash\ ev_2(c); z(true) : unit}}{\Gamma,\ H\ \vdash\ \mathsf{if}\ x\ \mathsf{then}\ ev_1(c)\ \mathsf{else}\ ev_2(c);\ z(true) : bool}\ (\mathsf{If})}{\varnothing,\ \epsilon\ \vdash\ \lambda_z x.\ \mathsf{if}\ x\ \mathsf{then}\ ev_1(c)\ \mathsf{else}\ ev_2(c); z(true) : bool \xrightarrow{H} unit}\ (\mathsf{Abs})$$

L'arbre A

$$\dfrac{\dfrac{\dfrac{\dfrac{\square}{\Gamma,\ \epsilon\ \vdash\ c : \{c\}}\ (\mathsf{Const})}{\Gamma,\ ev_1(c)\ \vdash\ ev_1(c) : unit}\ (\mathsf{Event})}{\Gamma,\ ev_1(c)|ev_2(c); H\ \vdash\ ev_1(c) : unit}\ (\mathsf{Weaken})}{\Gamma,\ H\ \vdash\ ev_1(c) : unit}\ (\mathsf{H} \xrightarrow{\epsilon} \mathsf{ev_1(c)|ev_2(c); H})$$

L'arbre B

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{D}{\Gamma,\ \epsilon\ \vdash\ \lambda_z x.z(true) : bool \xrightarrow{H} unit} \quad \dfrac{C}{\Gamma,\ ev_1(c)|ev_2(c)\ \vdash\ ev_2(c) : unit}}{\Gamma,\ ev_1(c)|ev_2(c); H\ \vdash\ (\lambda_z x.z(true))ev_2(c) : unit}\ (\mathsf{App})}{\Gamma,\ H\ \vdash\ (\lambda_z x.z(true))ev_2(c) : unit}\ (\mathsf{H} \xrightarrow{\epsilon} \mathsf{ev_1(c)|ev_2(c); H})}{\Gamma,\ H\ \vdash\ (\lambda x.z(true))ev_2(c) : unit}\ (\lambda \mathsf{x.e} \triangleq \lambda_\mathsf{z} \mathsf{x.e})}{\Gamma,\ H\ \vdash\ (\lambda_.z(true))ev_2(c) : unit}\ (\lambda_.\mathsf{e} \triangleq \lambda \mathsf{x.e})}{\Gamma,\ H\ \vdash\ ev_2(c); z(true) : unit}\ (\mathsf{e_1; e_2} \triangleq (\lambda_.\mathsf{e_2})\mathsf{e_1})$$

L'arbre C

$$\dfrac{\dfrac{\dfrac{\square}{\Gamma,\ \epsilon\ \vdash\ c : \{c\}}\ (\mathsf{Const})}{\Gamma,\ ev_2(c)\ \vdash\ ev_2(c) : unit}\ (\mathsf{Event})}{\Gamma,\ ev_1(c)|ev_2(c)\ \vdash\ ev_2(c) : unit}\ (\mathsf{Weaken})$$

L'arbre D

$$\dfrac{\dfrac{\dfrac{\Gamma(z) = bool \xrightarrow{H} unit}{\Gamma,\ \epsilon\ \vdash\ z : bool \xrightarrow{H} unit}\ (\mathsf{Var}) \quad \dfrac{\square}{\Gamma,\ \epsilon\ \vdash\ true : bool}\ (\mathsf{Bool})}{\Gamma,\ H\ \vdash\ z(true) : unit}\ (\mathsf{App})}{\Gamma,\ \epsilon\ \vdash\ \lambda_z x.z(true) : bool \xrightarrow{H} unit}\ (\mathsf{Abs})$$

TABLE 3.5 – Typage de l'exemple.

$$\begin{array}{lcl} ev(c) & \xrightarrow{ev(c)} & \epsilon \\ H_1|H_2 & \xrightarrow{\epsilon} & H_1 \\ H_1|H_2 & \xrightarrow{\epsilon} & H_2 \\ \epsilon; H & \xrightarrow{\epsilon} & H \\ \mu h.H & \xrightarrow{\epsilon} & H[\mu h.H/h] \\ H_1; H_2 & \xrightarrow{a} & H_1^{'}; H_2 \text{ si } H_1 \overset{a}{\rightarrow} H_1^{'} \end{array}$$

Ainsi, l'interprétation LTS d'un historique H, notée $[[H]]$, peut être définie comme suit :

$$[[H]] = \{a_1 a_2 ... a_n | H \overset{a_1}{\rightarrow} ... \overset{a_n}{\rightarrow} H^{'}\} \cup \{a_1 a_2 ... a_n \downarrow | H \overset{a_1}{\rightarrow} ... \overset{a_n}{\rightarrow} \epsilon\}$$

Aussi, l'interprétation $[[H]]$ d'un historique H peut engendrer un nombre important de traces diminuant ainsi la performance du processus de vérification. Pour répondre à ce problème, cette approche introduit un mécanisme de détection des traces redondantes. Pour se faire, l'ensemble des traces, noté t est décomposé en plusieurs sous-ensembles. Chaque sous-ensemble t_i définit un nouvel historique d'effets H_i tel que :

$$[[H_i]] = t_i \quad \text{et} \quad \bigcup_{i=0}^{n} [[H_i]] = t$$

L'idée générale consiste à rechercher les historiques équivalents et d'éliminer les traces redondantes (si l'un des deux historiques est valide alors l'autre l'est aussi). Dans ce cas, deux historiques sont alors dits *similaires* si leurs interprétations sont équivalentes : $H_1 = H_2 \Leftrightarrow [[H_1]] = [[H_2]]$. Toutefois, cette relation n'est pas décidable vu que deux traces peuvent être infinies [4].

De ce fait, la validité d'un historique H est basée sur la validité de chaque assertion composant son interprétation. Autrement dit, chaque point de contrôle (assertion) doit vérifier la validité de la trace θ qui le précède. Cette propriété est définie comme suit :

Définition 3.3.1 *L'historique H est valide si pour toute trace $\theta\ ev_\phi(c) \in [[H]]$, la fonction $\Pi(\phi(c), \theta\ ev_\phi(c))$ doit être vérifiée.*

Cette propriété assure qu'un historique ne peut être valide que si tous les points de contrôle sont satisfaits. Chaque assertion permet de vérifier la validité de la trace qui la précède. Aussi, la validité de toutes les assertions ϕ permet de garantir la cohérence de l'historique H et donc sa validité. En effet, en se référant à la règle sémantique *check* introduite dans la table 3.2 (page 43), chaque événement inséré dans un historique doit

préserver la cohérence de ce dernier. Ainsi, si tous les événements constituant un historique assurent la cohérence de ce dernier alors cet historique est dit *valide*.

Aussi, le jugement $\Gamma, H \vdash e : \tau$ est dit robuste que si et seulement s'il est dérivable et que H est valide. En effet, si ce jugement est dérivable alors $[[H]]$ comporte des scénarios potentiels d'exécution de l'expression e. De plus, la validité de ces derniers est assurée par celle de l'historique H. Alors, $\Gamma, H \vdash e : \tau$ ne peut être que valide.

La propriété de sûreté peut être formalisée sous forme de deux théorèmes :

Théorème 3.3.1 *si* $\Gamma, H \vdash e : \tau$ *est dérivable pour une expression* e *et si* $\epsilon, e \rightarrow^* \eta, e^{'}$ *alors* $\hat{\eta} \in [[H]]$.

La preuve de ce théorème peut être énoncée comme suit : si $\Gamma, H \vdash e : \tau$ est dérivable et si $\epsilon, e \rightarrow^* \eta, e^{'}$ alors, et d'après les fondements de cette approche, $\eta \in H$ et donc $\hat{\eta} \in [[H]]$.

Théorème 3.3.2 *si* $\Gamma, H \vdash e : \tau$ *est valide pour une expression* e *alors elle est correcte.*

Ce théorème peut être prouvé comme suit : si $\Gamma, H \vdash e : \tau$ est robuste alors H est valide et donc η l'est aussi ($\eta \in H$) et donc il existe une configuration valide $(\eta, e^{'})$ telle que $\epsilon, e \rightarrow^* \eta, e^{'}$.

3.4 Vérification des historiques d'effets

3.4.1 μ-*calcul* modal

Pour vérifier un historique, cette approche introduit un mécanisme de validation basé sur la logique temporelle μ-*calcul*. L'idée générale de ce mécanisme est de transformer les assertions d'historique ϕ sous forme de prédicats pouvant être validés grâce aux algorithmes de vérification de modèles (*model-checking*). Le choix de cette logique est motivé du fait de sa puissance d'expression et de son rapprochement syntaxique des historiques des effets (voir table 3.3). De plus, cette logique a la caractéristique de supporter les ensembles infinis. Aussi, les historiques infinis peuvent être vérifiés grâce à cette logique.

De plus, cette approche utilise une variante linéaire du μ-*calcul* car au moment de l'exécution vu que le système ne peut admettre qu'une seule trace d'événements (l'exécution du programme ne peut suivre qu'un seul chemin dans l'ensemble des scénarios

$$
\begin{array}{lll}
[[true]]_V & = & \Theta^\infty \\
[[x]]_V & = & V(x) \\
[[\neg\phi]]_V & = & \Theta^\infty - [[\phi]]_V \\
[[\phi_1 \ \wedge \ \phi_2]]_V & = & [[\phi_1]]_V \ \cap \ [[\phi_2]]_V \\
[[(a)\phi]]_V & = & \{\theta^\infty \in \Theta^\infty \mid \theta^\infty = a; \theta_1^\infty \text{ et } \theta_1^\infty \in [[\phi]]_V\} \\
[[\upsilon x.\phi]]_V & = & \bigcup\{W \subseteq \Theta^\infty | W \subseteq [[\phi]]_{V[x \subseteq W]}\}
\end{array}
$$

TABLE 3.6 – Sémantique de la logique μ-*calcul*.

possibles) [4, 6]. De plus, l'historiques d'effets H et l'historique des événements η ne peuvent être comparés que dans une logique linéaire temporelle.

La syntaxe de cette logique peut être introduite comme suit :

$$\phi \ ::= \ x \mid \textit{true} \mid \textit{false} \mid \mu x.\phi \mid \upsilon x.\phi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid (a)\phi$$

où a représente un argument de transition (ex. $ev(c)$)

La sémantique du μ-*calcul* est définie dans la table 3.6. L'expression θ dénote une trace associée à un historique d'effets, Θ désigne le préfixe d'une trace θ et V représente l'association des variables de la logique μ-*calcul* vers des traces potentiellement infinies notées θ^∞ ($\theta^\infty \in \Theta^\infty$) [5]. Aussi, le l'association vide, noté $\varnothing$, peut être défini comme suit : $[[\phi]] = [[\phi]]_\varnothing$. Cette sémantique peut être enrichie grâce aux règles de base de la logique μ-*calcul*, par exemple :

$$
\begin{array}{lll}
[[\phi_1 \vee \phi_2]] & = & [[\neg(\neg\phi_1 \wedge \neg\phi_2)]] \\
 & = & \Theta^\infty - [[\neg\phi_1 \wedge \neg\phi_2]] \\
 & = & \Theta^\infty - ([[\neg\phi_1]] \cap [[\neg\phi_2]]) \\
 & = & \Theta^\infty - ((\Theta^\infty - [[\phi_1]]) \cap (\Theta^\infty - [[\phi_2]])) \\
 & = & \Theta^\infty - (\Theta^\infty - ([[\phi_1]] \cup [[\phi_2]])) \\
 & = & [[\phi_1]] \cup [[\phi_2]]
\end{array}
$$

L'interprétation d'une assertion ϕ, notée $[[\phi]]$, peut être définie comme un ensemble de préfixes de traces tels que chaque préfixe renferme une séquence d'exécution valide :

$$
\begin{array}{l}
[[\phi]] = \{\theta | \theta \text{ est le préfixe d'une trace infinie } \theta^\infty \in \Theta^\infty\} \cup \\
\qquad \{\theta \downarrow | \theta \text{ est le préfixe d'une trace } \theta \in \Theta^\infty\}
\end{array}
$$

5. Il faut noter que le préfixe d'une trace est aussi une trace.

Une formule ϕ est dite valide pour un historique H, notée $H \Vdash \phi$, si et seulement si $[[H]] \subseteq [[\phi]] \downarrow$. De plus, si toutes les traces d'un historique correspondent à des séquences valides d'exécution alors le programme associé à cet historique l'est aussi.

3.4.2 Historiques d'effets et traces d'exécution

La relation $H \Vdash \phi$ est décidable vu qu'elle peut être vérifiée grâce aux algorithmes de vérification de modèle. Une condition importante de cette logique est que les formules doivent avoir des valeurs réelles étant donné un historique d'effets H et un historique d'exécution η. Aussi, une formule ϕ est associée à l'historique d'événements η par l'intermédiaire de la propriété suivante : $\hat{\eta} \in [[\phi]] \downarrow$. En effet, la trace d'exécution d'un programme valide doit appartenir à l'ensemble des scénarios *finis* spécifiés dans $[[\phi]]$. En se référant au théorème 3.3.1, on peut obtenir la propriété suivante :

$$\text{Si } \hat{\eta} \in [[H]] \text{ et } H \Vdash \phi \text{ alors } \hat{\eta} \in [[\phi]] \downarrow$$

En effet, si $\Gamma, H \vdash e : \tau$ est dérivable et si $\epsilon, e \rightarrow^{*} \eta, e^{'}$ alors $\hat{\eta} \in [[H]]$. De plus, si $H \Vdash \phi$ alors $[[H]] \in [[\phi]] \downarrow$. On peut conclure alors que $\hat{\eta} \in [[\phi]] \downarrow$. Cette propriété schématise le fondement de cette approche. Aussi, l'idée générale est de vérifier l'historique d'effets avec le modèle défini par le programmeur. En effet, au moment de la rédaction du code, le programmeur va insérer des assertions sous forme de *traces* représentant les scénarios possibles d'exécution du programme jusqu'au point de contrôle. De ce fait, un historique d'effets est dit *valide* s'il vérifie le modèle associé à chacun des points de contrôles du programme.

3.4.3 Propriétés de vérification

Une assertion ϕ peut être perçue comme une fonction manipulant une constante c (voir table 3.1, page 42). Aussi, et pour avoir une définition plus générale, la variable $\mathcal{X}$ est introduite comme argument à une assertion telle que cette dernière ne peut être instanciée que par une constante au moment de la vérification. Cette représentation permet de construire un modèle abstrait englobant tous les schémas possibles de vérification associés à un point de contrôle donné. Aussi, chaque événement de contrôle $ev_{\phi}(c)$ ne peut être inséré à un historique d'exécution η que si ce dernier vérifie le modèle associé à l'assertion ϕ. Ainsi, afin de préserver l'abstraction du modèle associé à une assertion ϕ, cette approche introduit le label Now afin de représenter un événement de contrôle ev_{ϕ}. Ce label va être instancié au moment de la vérification d'un historique donné. Chaque formule ϕ peut être une étiquette a tel que :

$$a \quad ::= \quad ev(s) \mid Now$$

Au moment de la rencontre d'un événement $ev_\phi(c)$, l'historique η doit vérifier le modèle associé à l'assertion ϕ. Autrement dit, le processus de vérification va contrôler la validité des traces construites jusqu'à l'événement $ev_\phi(c)$. Pour se faire, ce processus va comparer chaque trace θ avec toutes les séquences valides construites jusqu'au point de contrôle Now, représentant l'interprétation de l'événement $ev_\phi(c)$. Aussi, la propriété de validation d'une trace peut être définie comme suit :

$$\Pi(\phi(c), \theta) \Leftrightarrow \theta \in [[\phi[c/\mathcal{X}][ev_\phi(c)/Now]]] \downarrow \text{ où } (\mathcal{X} \in \mathcal{V}_s)$$

Aussi, on peut dire qu'un historique d'effets H est ***vérifié*** s'il satisfait le modèle de l'assertion ϕ. Autrement dit, si H prévoit l'occurrence d'un événement de contrôle $ev_\phi(c)$ alors H doit absolument vérifier le modèle de l'assertion $\phi(c)$. La propriété de vérification peut alors être définie comme suit :

Définition 3.4.1 *Un historique d'effets H est dit valide si et seulement si pour toutes les occurrences $ev_\phi(c)$ on obtient : $H \Vdash \phi[c/\mathcal{X}][ev_\phi(c)/Now]$.*

En se référant aux propriétés précédentes, on obtient le corollaire suivant :

Corollaire 3.4.1 *Si l'historique H est vérifié alors il est valide.*

3.5 Variation orientée *pile*

La section suivante présente une variante orientée *pile* permettant de vérifier certaines propriétés d'exécution d'un programme au moment de sa compilation. Cette variation ne prend en compte que les événements associés aux appels de fonctions apparaissant dans la pile courante d'exécution d'un programme. Les assertions ϕ sont seulement associées avec les séquences actives d'événements. Les auteurs assurent que cette variation ne nécessite qu'un post-traitement mineur afin d'obtenir un modèle de sécurité orienté *pile* à partir des *historiques d'effets*. Cette variation peut être utile pour vérifier des modèles supportant le concept d'exception : chaque événement succédant à une exception est rejeté.

$S \ ::= \ nil \mid S :: \eta$	Pile d'historiques
$S, \ (\lambda_z x.e)v \rightsquigarrow S :: \epsilon, \ \ \cdot e[v/x][\lambda_z x.e/z]\cdot$	(β)
$S :: \eta, \ ev(c) \rightsquigarrow S :: \eta; \ ev(c), \ ()$	(Event)
$S :: \eta, \ \phi(c) \rightsquigarrow S :: \eta; \ ev_\phi(c), \ ()$	si $\Pi(\phi(c), \ (S;\eta)ev_\phi(c))$(Check)
$S :: \eta, \ \ \cdot v \cdot \rightsquigarrow S, \ v$	(Pop)

TABLE 3.7 – Sémantique du langage λ^S_{hist}.

3.5.1 Syntaxe et Sémantique

La syntaxe du langage λ^S_{hist} définissant la pile d'effets se base sur celle du langage λ_{hist}. De plus, elle introduit l'expression $\cdot e \cdot$ et le contexte $\cdot E \cdot$ définissant la portée d'une fonction. chaque expression e, appartenant à la fonction $\lambda_z x.e$, ne peut comporter aucune sous-expression $\cdot e^{'} \cdot$. La sémantique opérationnelle du langage λ^S_{hist}, définie dans la table 3.7, manipule une configuration (S, e) à travers les relations de réduction $\rightarrow$ et $\rightsquigarrow$, où S représente la pile d'historiques. Aussi, la pile de sécurité peut être obtenue par l'ajout des historiques d'événements d'une manière chronologique ; ceci peut être formalisé à partir de la notation $S;\eta$ comme suit :

$$nil; \ \eta \ = \ \eta \qquad\qquad S :: \eta; \eta^{'} \ = \ S; \eta; \eta^{'}$$

Comme on peut le remarquer, les règles β, $Event$ et $Check$ sont similaires à celles du langage λ_{hist} ; ce qui démontre que les deux langages suivent les mêmes mécanismes pour définir le modèle d'un programme donné (y compris lors de l'interprétation de la fonction Π). Aussi, on peut obtenir d'autres règles sémantiques en remplaçant l'historique η par S dans la table 3.2.

3.5.2 Transformation d'un historique d'effets en pile (*Stackified*)

Même si le langage λ^S_{hist} n'utilise pas d'historiques d'effets, cependant, les règles de typage précédentes peuvent être appliquées pour typer les expressions de ce dernier (voir table 3.4). Pour se faire, les historiques d'effets vont être transformés grâce à l'algorithme *stackify* sous forme d'éléments la pile S. Chaque activation d'une fonction (*resp.* désactivation) va correspondre à un empilement (*resp.* dépilement) dans la pile d'historique. L'algorithme *stackify* est décrit dans la table 3.8.

$$
\begin{array}{rcl}
\textit{stackify}(\epsilon) & = & \epsilon \\
\textit{stackify}(\epsilon, H) & = & \textit{stackify}(h) \\
\textit{stackify}(ev(c), H) & = & ev(c); \textit{stackify}(h) \\
\textit{stackify}(h; H) & = & h|\textit{stackify}(H) \\
\textit{stackify}((\mu h.H_1); H_2) & = & \mu h.\textit{stackify}(H_1)|\textit{stackify}(H_2) \\
\textit{stackify}((H_1|H_2); H) & = & \textit{stackify}(H_1; H)|\textit{stackify}(H_2, H) \\
\textit{stackify}((H_1; H_2); H_3) & = & H_1; (H_2; H_3) \\
\textit{stackify}(H) & = & \textit{stackify}(h,\ \epsilon)
\end{array}
$$

TABLE 3.8 – Algorithme *stackify*.

Comme on peut le remarquer, cet algorithme manipule uniquement des séquences d'historiques $(H_1; H_2)$. La dernière règle permet de réécrire une clause unaire sous forme d'une séquence. On remarque que l'algorithme *stackify* procède à un traitement itératif de l'historique jusqu'à l'obtention d'une forme basique ne comportant aucune récursivité ; la pile est donc un système de transition à états fini pouvant être traité d'une manière plus optimale par les algorithmes de vérification de modèle que les historiques d'effets.

Par exemple, l'expression $\Xi = \textit{stackify}(a; (\mu h.b; c); (\mu h.c; (\epsilon|(d; h; a))))$ peut être traduite comme suit :

$$
\begin{array}{rcl}
\Xi & = & a; \textit{stackify}((\mu h.b; c); (\mu h.c; (\epsilon|(d; h; a)))) \\
& = & a; (\mu h.\textit{stackify}(b; c)|\textit{stackify}(\mu h.c; (\epsilon|(d; h; a)))) \\
& = & a; (\mu h.b; \textit{stackify}(c))|\textit{stackify}(\mu h.c; (\epsilon|(d; h; a))) \\
& = & a; (\mu h.b; c)|\textit{stackify}(\mu h.c; (\epsilon|(d; h; a))) \\
& = & a; (\mu h.b; c)|\textit{stackify}(\mu h.c; \epsilon)|\textit{stackify}((d; h; a)) \\
& = & a; (\mu h.b; c)|\mu h.\textit{stackify}(c; \epsilon)|d; \textit{stackify}(h; a) \\
& = & a; (\mu h.b; c)|\mu h.c; \textit{stackify}(\epsilon)|d; (h|\textit{stackify}(a)) \\
& = & a; (\mu h.b; c)|\mu h.c; \epsilon|d; (h|a) \\
& = & a; ((\mu h.b; c)|\mu h.c; (\epsilon|(d; h)|(d; a)))
\end{array}
$$

La validité du jugement $\Gamma, H \vdash e : \tau$ dans le langage λ^S_{hist} peut être déduite à partir de la conformité de *stackify*(H). Il faut noter que les auteurs introduisent certaines restrictions sur les expressions pour s'assurer qu'une pile vide ne sera jamais dépilée. Aussi, la propriété de sûreté peut alors être décrite d'une manière analogue à celle du théorème 3.3.1 comme suit :

Théorème 3.5.1 *si le jugement $\Gamma, H \vdash e : \tau$ pour une expression e qui ne contient aucune sous-expression de la forme $\cdot e' \cdot$ et si stacktify(H) est valide alors e est correcte.*

En effet, si e ne comporte aucune sous-expression de la forme $\cdot e' \cdot$ alors $\epsilon, e \rightarrow^* \eta, e'$

$(p_{\neg r(c)})\phi$	$\triangleq$	$(\vee\{p \mid r(c) \notin \mathcal{A}(p)\})\phi$	
$\phi_{demand,r}(c)$	$\triangleq$	$\neg((.*)(p_{\neg r(c)})(.*)(Now)\textit{true})$	
$(\bar{p})\phi$	$\triangleq$	$(\vee(\{ev_1()c_1 \ldots, ev_n(c_n)\}\backslash dom(\mathcal{A})))\phi$	
$(\neg ev(c))\phi$	$\triangleq$	$\vee(\{ev_1()c_1 \ldots, ev_n(c_n)\}\backslash\{ev(c)\}))\phi$	
$(a*)\phi$	$\triangleq$	$\mu x.(a)x \vee \phi$	pour $a \in \{\bar{p}, \neg ev(c)\}$
$\phi_{enable-ok,r}(c)$	$\triangleq$	$\neg((.*)(p_{\neg r(c)})(\bar{p}*)(enable(c))\textit{true})$	
$\phi_{inspect-ok,r}(c)$	$\triangleq$	$\neg((\neg enable_r(c)*)(Now)\textit{true})\wedge$ $\neg((.*)(p_{\neg r(c)})(\neg enbale_r(c)*)(Now)\textit{true})$	
$\phi_{inspect,r}(c)$	$\triangleq$	$\phi_{enable-ok,r}(c) \wedge \phi_{inspect-ok,r}(c)$	

TABLE 3.9 – Définition de $\phi_{demand,r}$ et $\phi_{inspect,r}$.

est valide ; et, si $stackify(H)$ est valide alors H l'est aussi (H comporte une séquence d'événements cohérente). Donc, on peut conclure que e est correcte.

3.5.3 Pile d'inspection et privilèges paramétrés

Dans le modèle de sécurité du langage JAVA, chaque principal qui apparaît dans un code JAVA doit absolument expliciter ses ressources. Aussi, lorsqu'une fonction est lancée, son principal est inséré dans la pile et toutes les ressources apparaissant dans son corps. La procédure d'inspection pour une ressource donnée $r(c)$ va alors examiner la pile pour définir les privilèges d'accès à cette dernière. Cette procédure va échouer si un principal réclame une ressource à laquelle il n'a pas accès, ou bien si la pile ne comporte aucune information sur la disponibilité de la ressource $r(c)$.

Afin de présenter la modélisation de la pile d'inspection par le langage λ^S_{his}, les auteurs introduisent un exemple qui fait appel à la fonction suivante :

$$\textit{checkit} \triangleq \lambda x.p : \textit{system}; \phi_{\textit{inspect},r:\textit{filew}}(x)$$

Au moment de son exécution, la fonction $checkit$ va engendrer un événement dénotant qu'elle a été déclenchée à partir du système $(p : system)$ indiquant que le système est le principal auquel elle appartient. La fonction va vérifier si le fichier x est accessible en écriture par l'inspection de la pile par l'intermédiaire de l'assertion $\phi_{\textit{inspect},r:\textit{filew}}(x)$ qui permet de s'assurer que toutes les fonctions incluses dans le principal p ont l'autorisation d'accéder à la ressource r : $\textit{filew}(x)$ tel que $p \in dom(\mathcal{A})$.

L'événement $enable_r(x)$ permet d'indiquer si la ressource $r(x)$ est temporairement

accessible. Afin d'illustrer l'utilisation explicite le concept de "permission", les auteurs introduisent la fonction $enableit$ associée au principal $(p : acct)$ permettant à une fonction f d'écrire dans un fichier x en activant la ressource $(r : \mathit{filew}(x))$:

$$enableit \triangleq \lambda f.p : acct; (\lambda x.p : acct; enable_{r:\mathit{filew}}(x); \mathsf{let}\ y = f(x)\ \mathsf{in}\ y)$$

L'assertion $\phi_{inspect,r}(c)$, définie dans la table 3.9, permet de vérifier si la ressource $r(c)$ est accessible comme suit : tout d'abord, elle s'assure que le principal dispose du droit d'accès à cette ressource $(\phi_{enable-ok,r}(c))$; puis, elle examine la pile d'inspection de cette ressource afin de prospecter la disponibilité de cette dernière $(\phi_{inspect-ok,r}(c))$.

Les tables 3.10 et 3.11 présentent respectivement la preuve de typage des fonctions $checkit$ et $enableit$ basée sur les règles définies dans la table 3.4. On obtient alors :

$$\begin{array}{ll} checkit : \forall\alpha.\{\alpha\} \xrightarrow{H_1} unit & \text{où : } H_1 \triangleq p : system; \phi_{inspect,r:filew(\alpha)} \\ enableit : \forall\alpha ht.(\{\alpha\} \xrightarrow{h} t) \xrightarrow{p:acct} \{\alpha\} \xrightarrow{H_2} t & \text{où : } H_2 \triangleq p : acct; enable_{r:\mathit{filew}}(\alpha); h \end{array}$$

En appliquant l'algorithme $stackify$ sur l'historique engendré par l'application *enableit(checkit(f.pdf))*, on obtient un nouvel historique d'effets H tel que :

$$H = p : acct | p : acct; enbale_r(f.pdf); p : system; \phi_{inspect,r:\mathit{filew}}(f.pdf)$$

La vérification de l'application *enableit(checkit(f.pdf))* va réussir dans le cas où les deux principaux $(p : system)$ et $(p : acct)$ peuvent accéder en écriture au fichier "f.pdf" $(r : \mathit{filew}(f.pdf))$. En effet, les historiques engendrés au moment de l'exécution des fonctions $checkit$ et $enableit$ peuvent être définis comme suit :

$$\begin{array}{lcl} H_{checkit} & \triangleq & p : system; \phi_{inspect,r:\mathit{filew}}(f.pdf) \\ H_{enableit} & \triangleq & \mu h.p : acct; p : acct; enable_{r:\mathit{filew}}(f.pdf); h \end{array}$$

Au moment du lancement de l'application *enableit checkit*, l'historique $H_{checkit}$ va se greffer à l'historique $H_{enableit}$ par l'intermédiaire de la variable h représentant l'historique engendré par la procédure f introduite dans la fonction *enableit*. En effet, la procédure *checkit* va être assimiler à la fonction f dont l'effet h n'apparaît qu'à la fin de l'historique $H_{enableit}$. On obtient alors :

$$\begin{array}{lcl} H_{enableit\ checkit(f.pdf)} & \triangleq & H_{enableit}; H_{checkit} \\ enableit\ checkit(f.pdf) & : & unit \end{array}$$

$$\dfrac{\dfrac{\dfrac{A \quad B}{\Gamma_1, H_0 \vdash (\lambda_{z_1} k_1.\phi_{inspect,r:filew}(x))(p : system) : unit}\ (\mathsf{App})}{\Gamma, x : \forall\alpha.\{\alpha\}; z : \forall\alpha.\{\alpha\} \xrightarrow{H_0} unit, H_0 \vdash p : system; \phi_{inspect,r:filew}(x) : unit}}{\Gamma, \epsilon \vdash \lambda_z x.p : system; \phi_{inspect,r:filew}(x) : \forall\alpha.\{\alpha\} \xrightarrow{H_0} unit}\ (\mathsf{Abs})$$

l'arbre A

$$\dfrac{\dfrac{\dfrac{\Gamma_2(x) = \{\alpha\}}{\Gamma_2, \epsilon \vdash x : \{\alpha\}}\ (\mathsf{Var})}{\Gamma_1; k_1 : unit; z_1 : unit \xrightarrow{H_1} unit, H_1 \vdash \phi_{inspect,r:filew}(x) : unit}\ (\mathsf{Check})}{\Gamma_1, \epsilon \vdash \lambda_{z_1} k_1.\phi_{inspect,r:filew}(x) : unit \xrightarrow{H_1} unit}\ (\mathsf{Abs})$$

L'arbre B

$$\dfrac{\dfrac{\square}{\Gamma_1, \epsilon \vdash () : unit}\ (\mathsf{Unit})}{\Gamma_1, p : system \vdash (p : system) : unit}\ (\mathsf{Event})$$

$$\begin{array}{lcl}
\Gamma_1 & \triangleq & \Gamma, x : \forall\alpha.\{\alpha\}; z : \forall\alpha.\{\alpha\} \xrightarrow{H} unit \\
\Gamma_2 & \triangleq & \Gamma_1; k_1 : unit; z_1 : unit \xrightarrow{H_1} unit \\
H_0 & \triangleq & p : system; \phi_{inspect,r:filew}(\alpha) \\
H_1 & \triangleq & \phi_{inspect,r:filew}(\alpha)
\end{array}$$

TABLE 3.10 – Preuve de typage de la fonction $checkit$.

Dans ce cas, le résultat ne dépend que de la validité de l'événement $enable_{r:filew}(f.pdf)$ (il y aura un échec si l'événement n'apparaît pas dans la pile d'inspection).

$$\dfrac{\dfrac{\dfrac{A \qquad \dfrac{\dfrac{\square}{\Gamma_1, \epsilon \vdash () : unit}}{\Gamma_1, p : acct \vdash (p : acct) : unit}}{\Gamma_1, p : acct \vdash (\lambda_{z_0} k_0.(\lambda_z x.p : acct; enbale_{r:filew}(x);\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y))(p : acct) : \tau_1}}{\Gamma_0; f : \tau_f; z_0 : \tau_0, H_p \vdash p : acct; (\lambda_z x.p : acct; enbale_{r:filew}(x);\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y) : \tau_1}}{\Gamma_0, \epsilon \vdash \lambda_{z_0} f.p : acct; (\lambda_z x.p : acct; enbale_{r:filew}(x);\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y) : \tau_0}$$

L'arbre A

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{B}{\Gamma_3, \epsilon \vdash \lambda_{z_1} k_1.enbale_{r:filew}(x);\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y : unit \xrightarrow{H_1} t} \qquad \dfrac{\dfrac{\square}{\Gamma_3, \epsilon \vdash () : unit}}{\Gamma_3, p : acct \vdash (p : acct) : unit}}{\Gamma_3, H_0 \vdash (\lambda_{z_1} k_1.enbale_{r:filew}(x);\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y)(p : acct) : t}}{\Gamma_2; x : \{\alpha\}; z : \tau_1, H_0 \vdash p : acct; enbale_{r:filew}(x);\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y : t}}{\Gamma_1; k_0 : unit; z_0 : unit \to \tau_1, \epsilon \vdash \lambda_z x.p : acct; enbale_{r:filew}(x);\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y : \tau_1}}{\Gamma_1, \epsilon \vdash \lambda_{z_0} k_0.(\lambda_z x.p : acct; enbale_{r:filew}(x);\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y) : unit \to \tau_1}$$

L'arbre B

$$\dfrac{\dfrac{C \qquad \dfrac{\dfrac{\Gamma_4(x) = \{\alpha\}}{\Gamma_4, \epsilon \vdash x : \{\alpha\}}}{\Gamma_4, enable_{r:filew}(\alpha) \vdash enable_{r:filew}(x) : unit}}{\Gamma_4, H_1 \vdash (\lambda_{z_2} k_2.\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y)(enbale_{r:filew}(x)) : t}}{\Gamma_3; k_1 : unit; z_1 : unit \xrightarrow{H_1} t, H_1 \vdash enbale_{r:filew}(x);\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y : t}$$

L'arbre C

$$\dfrac{\dfrac{\Gamma_5, \epsilon \vdash f(x) : t \qquad t \cap fv(\Gamma_5) = \varnothing \qquad \Gamma_5; y : t, h \vdash y : t}{\Gamma_4; k_2 : unit; z_2 : unit \xrightarrow{h} t, h \vdash\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y : t}}{\Gamma_4, \epsilon \vdash \lambda_{z_2} k_2.\ \mathsf{let}\ y = f(x)\ \mathsf{in}\ y : unit \xrightarrow{h} t}$$

$$\begin{array}{lcl}
H_0 & \triangleq & p : acct; enable_{r:filew}(\alpha); h \\
H_p & \triangleq & p : acct \\
H_1 & \triangleq & enable_{r:filew}(\alpha); h \\
\tau_0 & \triangleq & \forall \alpha h t.(\{\alpha\} \xrightarrow{h} t) \xrightarrow{H_p} \{\alpha\} \xrightarrow{H_0} t \\
\tau_f & \triangleq & \forall \alpha h t.\{\alpha\} \xrightarrow{h} t \\
\tau_1 & \triangleq & \{\alpha\} \xrightarrow{H_0} t \\
\Gamma_1 & \triangleq & \Gamma_0; f : \forall \alpha h t.\{\alpha\} \xrightarrow{h} t; z_0 : \forall \alpha h t.(\{\alpha\} \xrightarrow{h} t) \xrightarrow{H_p} \{\alpha\} \xrightarrow{H_0} t \\
\Gamma_2 & \triangleq & \Gamma_1; k_0 : unit; z_0 : unit \to \{\alpha\} \xrightarrow{H_0} t \\
\Gamma_3 & \triangleq & \Gamma_2; x : \{\alpha\}; z : \{\alpha\} \xrightarrow{H_0} t \\
\Gamma_4 & \triangleq & \Gamma_3; k_1 : unit; z_1 : unit \xrightarrow{H_1} t \\
\Gamma_5 & \triangleq & \Gamma_4; k_2 : unit; z_2 : unit \xrightarrow{h} t
\end{array}$$

TABLE 3.11 – Preuve de typage de la fonction $enableit$.

3.6 Conclusion

Dans cet chapitre, nous avons présenté une approche de vérification et de validation de programmes informatiques basée sur le concept d'*historiques d'effets*. Cette approche introduit des mécanismes permettant de construire un modèle abstrait à partir d'un programme donné. Ce modèle est analysé par des algorithmes pouvant vérifier la cohérence de ce dernier. Ce modèle est construit à partir d'annotations introduites par l'utilisateur au moment de l'écriture du code du programme. Ces annotations correspondent à des assertions définies en logique temporelle μ-*calcul* représentant les séquences valides d'événements les précédents ou bien à des déclaration d'événements.

L'objectif premier de cette approche est de définir un système pouvant modéliser statiquement les paramètres des permissions appartenant à la pile d'inspection JAVA, et assurant la vérification statique des historiques d'exécution suivant les assertions introduites par le programmeur et de renforcer ainsi les politiques d'accès du programme. Cependant, le mécanisme de vérification délaisse les accès aux ressources et peut donc attester qu'un programme, procédant à l'envoi de données secrètes sur un réseau publique, est valide. En effet, cette approche ne définit aucun outil permettant d'abstraire les usages des ressources. De plus, la logique modale utilisée ne peut abstraire que des propriétés de sécurité relatives au flux de contrôle.

Deuxième partie

Contribution

Chapitre 4

Abstraction et vérification de programmes informatiques

4.1 Introduction

Dans ce chapitre, nous présentons un nouveau cadre de vérification de propriétés de sécurité d'un programme informatique à travers l'analyse de ses flots de données et de contrôle. Notre contribution se propose donc de combiner l'analyse des historiques d'effets avec celle des dépendances directes ou indirectes des ressources. Dans ce cas, l'historique d'effets d'un programme dénote le flot de contrôle de ce dernier, alors que les dépendances relatent le flot de données de ce même programme. Notre mécanisme de vérification comprend trois étapes :

- Lors de la première étape, une analyse statique est opérée afin de construire une approximation des opérations définies dans un programme. Pour se faire, cette analyse va déterminer les effets pouvant être engendrés au moment de l'exécution du programme ainsi que les différentes corrélations reliant les variables du dit programme. Cette étape correspond concrètement à la conception d'un algorithme d'inférence de types et d'effets.
- La validation du modèle extrait précédemment est entreprise à la deuxième étape. Cette validation est effectuée au moyen d'un système de typage pouvant assurer la correspondance entre le modèle et le programme. En d'autres mots, cette étape permet d'affirmer qu'un programme est bien typé étant donnés un historique d'effets et un ensemble de dépendances.
- La dernière étape sert à examiner si le modèle obtenu vérifie une propriété de sécurité exprimée à l'aide d'une logique temporelle. Cette vérification résulte en l'une des réponses suivantes : Le modèle satisfait ou viole la propriété de sécurité. Dans

le cas où le modèle violerait la propriété de sécurité, il est souhaitable de disposer d'un mécanisme permettant de préciser l'opération du programme transgressant la politique de sécurité.

La principale contribution de cette approche réside dans le fait qu'elle propose un modèle complet qui offre une abstraction des différentes opérations que réalise le programme durant son exécution. En effet, la plupart des approches de vérification se focalisent sur un seul aspect du programme et délaissent ainsi d'autres aspects qui peuvent contribuer à la violation d'une politique de sécurité donnée. Par exemple, l'approche présentée dans le chapitre 2 (page 27) se concentre uniquement sur l'analyse du flot de contrôle et néglige l'exploitation des ressources alors qu'il est évident que l'envoi du contenu d'un fichier sensible sur un réseau publique renferme une défaillance au niveau de la sécurité d'un programme. Par conséquent, la sécurité d'un programme ne peut être vérifiée qu'en procédant à l'analyse de tous les aspects pouvant provoquer une carence dans la sécurité d'un programme.

Le reste de ce chapitre est organisé comme suit : Dans un premier temps, nous définissons le langage sur lequel opère notre approche. Cette définition est composée d'une syntaxe et d'une sémantique opérationnelle. À la section 4.3, nous introduisons un système de typage permettant de valider les types des variables et les effets associés au programme [1]. Par la suite, nous exposons un algorithme d'extraction des dépendances de données et de l'historique d'effets à partir d'un programme donné. À la section 4.4, nous décrivons un mécanisme de vérification de propriétés de sécurité construit à partir d'un algorithme de vérification par évaluation de modèles.

4.2 Programmes

Cette section permet de définir le contexte d'étude pour la vérification de propriétés de sécurité d'un programme donné via une logique temporelle. Nous présentons d'abord la structure générale d'un programme pouvant être vérifié. Pour se faire, nous exposons la syntaxe et la sémantique des programmes pouvant être analysées par notre mécanisme de vérification.

p	$::=$	let $d_1, \ldots, d_n$ in e
d	$::=$	fun $f\,(x_1 : \tau_1, \ldots, x_m : \tau_m) : \tau = e$
e	$::=$	$c \mid x \mid api \mid e\ op_b\ e' \mid op_u\ e \mid e\,(e_1, \ldots, e_n) \mid x := e \mid e; e' \mid$
		letvar $(x : \tau = e)$ in e' end $\mid$ while e do e $\mid$ if e then e else e
op_b	$::=$	$+ \mid - \mid * \mid / \mid \% \mid < \mid = \mid \&\&$
op_u	$::=$	!

TABLE 4.1 – Syntaxe.

4.2.1 Syntaxe

La table 4.1 introduit la syntaxe des programmes considérés par cette approche. Il s'agit essentiellement de la syntaxe d'un langage de type fonctionnel/impératif (très similaire à la structure des programmes C) :

- Un programme p correspond à un ensemble de declarations de fonctions suivi d'une expression (qui abstrait l'entrée (*main*) d'un programme). L'évaluation d'un programme correspond donc à la valeur résultante de l'évaluation de cette expression.
- Une déclaration d permet de définir une fonction pouvant admettre plusieurs paramètres typés (les types sont introduits dans la table 4.7, page 73).
- Une expression e désigne soit une constante c (entiers, booléens, chaînes de caractères, etc.), soit une variable x, soit l'identifiant d'une API *(Application Programming Interface)*, soit l'application d'une expression à plusieurs autres expressions, $e\,(e_1, \ldots, e_n)$, ce qui résulte en un appel de fonction, soit une affectation $x := e$, soit à une séquence d'expressions $e; e'$, une conditionnelle de la forme if e then e else e ou une boucle de la forme while e do e, soit une expression arithmétique binaire, $e\ op_b\ e$, ou unaire, $op_u\ e$, soit la déclaration d'une variable locale x, de type τ et initialisée par l'expression e, évaluée dans une expression e'.

Il est intéressant de constater la correspondance entre les programmes respectant cette grammaire et les programmes écrits en langage C (table 4.2).

Par ailleurs, pour des raisons de lisibilité, nous utiliserons cette abréviation :

1. Notez que la structure du chapitre ne correspond pas à la séquences d'étapes introduite en introduction de ce chapitre. Il est en effet d'usage de définir un système de types avant de définir l'algorithme d'inférence de types.

	C		
1	*void* swap(*int** x, *int** y)	1	let fun swap(x : *int*, y : *int*) : *void* =
2	{	2	letvar z : *int* = x
3	*int* z = *x ;	3	in
		4	x := y ;
4	*x = *y ;	5	y := z
5	*y = z ;	6	end
6	}		
7	*int* max(*int* x, *int* y)	7	fun max(x : *int*, y : *int*) : *int* =
8	{	8	if (x > y)
9	if (x > y)	9	then
10	x ;	10	x
11	else	11	else
12	y ;	12	y
13	}		
14	*void* main (*int* argc, *char** argv[])	13	in
15	{	14	letvar x : *int* = 1
16	*int* x = 1 ;	15	y : *int* = 10
17	*int* y = 10 ;	16	in
		17	swap(x,y) ;
18	swap(&x,&y) ;	18	max(x,y)
19	return max(*x,*y) ;	19	end
20	}		

TABLE 4.2 – Comparaison avec le langage C.

```
letvar x1 : τ1 = e1                letvar x1 : τ1 = e1
       x2 : τ2 = e2                in
in                          ≜          letvar x2 : τ2 = e2
       e                               in
end                                        e
                                       end
                                   end
```

Aussi, soulignons qu'il est assez facile d'étendre ce langage avec d'autres opérateurs ou constructions syntaxiques :

$$\begin{array}{rcl}
\mathsf{do}\ e'\ \mathsf{while}\ e & \triangleq & e'; \mathsf{while}\ e\ \mathsf{do}\ e' \\
\mathsf{for}\ (e_1, e_2, e_3)\ \mathsf{do}\ e & \triangleq & e_1; \mathsf{while}\ e_2\ \mathsf{do}\ e; e_3 \\
e \mid\mid e' & \triangleq & !(!e\ \&\&\ !e') \\
e \leq e' & \triangleq & e < e' \mid\mid e = e' \\
e \geq e' & \triangleq & !(e < e') \\
e! = e' & \triangleq & !(e = e') \\
e > e' & \triangleq & !(e \leq e') \\
& \vdots &
\end{array}$$

4.2.2 Sémantique opérationnelle

La sémantique opérationnelle comporte un ensemble de règles définissant l'évaluation d'un programme au moment de son exécution. Nous utilisons un environnement dynamique, noté Δ, qui comprend la valeur dynamique des différents identificateurs, y compris les fonctions définies à l'aide du mot-clé «fun». Nous supposons que nous disposons d'un environnement dynamique Δ_a comprenant la valeur des différentes API prédéfiniesainsi que les valeurs de toutes les constantes. De plus, les notations $\lfloor \mathsf{op}_b \rfloor$ et $\lfloor \mathsf{op}_u \rfloor$ désignent respectivement l'opérateur algébrique ou mathématique correspondant à op_b et op_u : $\lfloor + \rfloor = +, \lfloor \% \rfloor = \%, \ldots$

Cette sémantique utilise les jugements suivants :

- $\vdash p \rightarrow v$
  Ce jugement permet de préciser que l'évaluation d'un programme p aboutit à la valeur v.
- $\Delta \vdash d \rightarrow \Delta'$
  Ce jugement permet de préciser en quoi s'évalue une déclaration de fonction. Comme stipulé dans ce jugement, étant donné un environnement dynamique Δ, une déclaration de fonction produit un nouvel environnement Δ'.
- $\Delta \vdash e \rightarrow v, \Delta'$
  En présence de l'affectation, l'évaluation d'une expression produit, en plus d'une valeur, un effet de bord. Cet effet de bord correspond à la mise à jour de l'environnement dynamique (Δ').

Les règles de la sémantique opérationnelle sont représentées dans la table 4.3 :

- La règle (pgm) stipule que l'évaluation d'un programme let $d_1, \ldots, d_m$ in e cor-

respond à la valeur résultante de l'évaluation de l'expression e. Cette dernière est évaluée dans un environnement comprenant l'effet cumulé produit par toutes les déclarations $d_1, \dots, d_m$.

- La règle (dec) spécifie que l'évaluation d'une fonction met à jour l'environnement en associant à l'identifiant f de la fonction un couple $\langle (x_1, \dots, x_n), e \rangle$ où $(x_1, \dots, x_n)$ et e dénotent respectivement les paramètres et le corps de cette fonction.
- Les règles (cst) et (var) sont standards ;
- La règle (api) est similaire à la règle (var).
- La règle (aff) stipule que l'instruction d'affection provoque une mise à jour de l'environnement d'exécution.
- La règle (seq) indique que l'environnement d'exécution d'une séquence d'expressions est mise à jour successivement par l'évaluation des deux expressions composant la séquence.
- La règle (app_1) est standard et souligne que les évaluations des expressions $(e_1, \dots, e_n)$ se font dans un ordre bien défini et mettent à jour progressivement l'environnement dynamique.
- La règle (app_2) est standard et suppose que l'environnement dynamique associe à chaque identificateur d'API un corps de fonction, abstrait dans la règle par λ, qui sera évalué pour produire un résultat.
- Les règles (if_1) et (if_2) décrivent l'évaluation des expressions conditionnelles.
- Les règles (wh_1) et (wh_2) décrivent l'évaluation des expressions itératives.
- Les autres règles sont standard.

4.2.3 Exemple

Pour illustrer l'utilisation des règles de la sémantique opérationnelle, prenons l'exemple de code C proposé à la table 4.4. Cette table propose aussi la version traduite de ce programme.

La table 4.5 présente l'arbre de preuve dynamique. Dans cette arbre, nous utilisons les abréviations suivantes :

$$\frac{\Delta_A \vdash d_1 \rightarrow \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash d_m \rightarrow \Delta_m \quad \Delta_m \vdash e \rightarrow v, \Delta}{\vdash \mathsf{let}\ d_1, \ldots, d_m\ \mathsf{in}\ e \rightarrow v}\ (\mathsf{pgm})$$

$$\frac{\Box}{\Delta \vdash \mathsf{fun}\ f\,(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e \rightarrow \Delta \dagger [f \mapsto \langle (x_1, \ldots, x_n), e \rangle]}\ (\mathsf{dec})$$

$$\frac{\Delta(c) = v}{\Delta \vdash c \rightarrow v, \Delta}\ (\mathsf{cst}) \qquad \frac{\Delta(x) = v}{\Delta \vdash x \rightarrow v, \Delta}\ (\mathsf{var}) \qquad \frac{\Delta(\mathit{api}) = v}{\Delta \vdash \mathit{api} \rightarrow v, \Delta}\ (\mathsf{api})$$

$$\frac{\Delta \vdash e \rightarrow v, \Delta' \quad \Delta' \vdash e' \rightarrow v', \Delta'' \quad v'' = v\ \lfloor \mathsf{op}_b \rfloor\ v'}{\Delta \vdash e\ \mathsf{op}_b\ e' \rightarrow v'', \Delta''}\ (\mathsf{op}_b)$$

$$\frac{\Delta \vdash e \rightarrow v, \Delta' \quad v' = \lfloor \mathsf{op}_u \rfloor\ v}{\Delta \vdash \mathsf{op}_u\ e \rightarrow v', \Delta'}\ (\mathsf{op}_u) \qquad \frac{\Delta \vdash e \rightarrow v, \Delta' \quad \Delta' \vdash e' \rightarrow v', \Delta''}{\Delta \vdash e; e' \rightarrow v', \Delta''}\ (\mathsf{seq})$$

$$\frac{\Delta \vdash e \rightarrow v, \Delta'}{\Delta \vdash \mathit{id} := e \rightarrow (), \Delta' \dagger [\mathit{id} \mapsto v]}\ (\mathsf{aff})$$

$$\frac{\Delta \vdash e \rightarrow \mathit{true}, \Delta' \quad \Delta' \vdash e'' \rightarrow v, \Delta''}{\Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e' \rightarrow v, \Delta''}\ (\mathsf{if}_1)$$

$$\frac{\Delta \vdash e \rightarrow \mathit{false}, \Delta' \quad \Delta' \vdash e'' \rightarrow v, \Delta''}{\Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' \rightarrow v, \Delta''}\ (\mathsf{if}_2)$$

$$\frac{\Delta \vdash e \rightarrow \mathit{false}, \Delta'}{\Delta \vdash \mathsf{while}\ e\ \mathsf{do}\ e' \rightarrow (), \Delta'}\ (\mathsf{wh}_1)$$

$$\frac{\Delta \vdash e \rightarrow \mathit{true}, \Delta' \quad \Delta' \vdash e' \rightarrow v, \Delta'' \quad \Delta'' \vdash \mathsf{while}\ e\ \mathsf{do}\ e' \rightarrow v', \Delta'''}{\Delta \vdash \mathsf{while}\ e\ \mathsf{do}\ e' \rightarrow v', \Delta'''}\ (\mathsf{wh}_2)$$

$$\frac{\begin{array}{c}\Delta \vdash e_1 \rightarrow v_1, \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash e_n \rightarrow v_n, \Delta_n \quad \Delta_n \vdash e \rightarrow \langle (x_1, \ldots, x_n), e' \rangle, \Delta' \\ \Delta' \dagger [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash e' \rightarrow v, \Delta''\end{array}}{\Delta \vdash e\,(e_1, \ldots, e_n) \rightarrow v, \Delta''}\ (\mathsf{app}_1)$$

$$\frac{\begin{array}{ccc}\Delta \vdash e_1 \rightarrow v_1, \Delta_1 & \ldots & \Delta_{n-1} \vdash e_n \rightarrow v_n, \Delta_n \\ \Delta_n \vdash \mathit{api} \rightarrow \lambda, \Delta_n & & v = \lambda(v_1, \ldots, v_n)\end{array}}{\Delta \vdash \mathit{api}\,(e_1, \ldots, e_n) \rightarrow v, \Delta_n}\ (\mathsf{app}_2)$$

$$\frac{\Delta \vdash e \rightarrow v, \Delta' \quad \Delta' \dagger [x \mapsto v] \vdash e' \rightarrow v', \Delta''}{\Delta \vdash \mathsf{letvar}\ x : \tau = e\ \mathsf{in}\ e'\ \mathsf{end} \rightarrow v', \Delta''}\ (\mathsf{let})$$

TABLE 4.3 – Sémantique opérationnelle.

```
1   void fct(handle f, int x, int y)
2   {
3       int z ;
4       if (x > 10) {
5           fprintf("% d", y, f) ;
6       }
7       else {
8           z := x ;
9           fprintf("% d", z, f) ;
10      }
11  }

12  void main (int argc, char* argv[])
13  {
14      int a = 5 ;
15      int b = 8 ;
16      handle c = fopen("password.txt") ;

17      fct(c, a, b) ;
18  }
```

```
1   let fun fct(f : handle^s, x : int^p, y : int^s) : void =
2
3       letvar z : int^p = 0
4       in
5           if (x > 10)
6               fprintf("% d", y, f)
7
8           else
9               z := x ;
10              fprintf("% d", z, f) ;
11
12      end
13

14  in
15
16      letvar a : int^p = 5
17          b : int^s = 8
18          c : handle^s = fopen("password.txt")
19      in
20          fct(c, a, b)
21      end
```

TABLE 4.4 – Exemple d'un programme.

$$
\begin{aligned}
fct_B \quad &= \quad \text{letvar } (z : int^p = 0) \text{ in} \\
&\qquad \text{if } (x > 10) \text{ then fprintf } ("\%d", y, f) \text{ else } z := x; \text{fprintf}("\%d", z, f) \\
&\quad\ \text{end} \\
Dec_{var} \quad &= \quad \text{letvar } a : int^p = 5 \text{ in} \\
&\qquad \text{letvar } b : int^s = 8 \text{ in} \\
&\qquad\quad \text{letvar } c : handle^s = \text{fopen}("password.txt") \\
&\qquad\qquad fct(c, a, b) \\
&\qquad\quad \text{end} \\
&\qquad \text{end} \\
&\quad \text{end}
\end{aligned}
$$

$$
\begin{array}{lcl}
Dec_a & = & \mathsf{letvar}\ b : int^s = 8\ \mathsf{in} \\
 & & \quad \mathsf{letvar}\ c : handle^s = \mathsf{fopen}(\text{"password.txt"})\ \mathsf{in} \\
 & & \quad\quad fct(c, a, b) \\
 & & \quad \mathsf{end} \\
 & & \mathsf{end} \\
Dec_b & = & \mathsf{letvar}\ c : handle^s = \mathsf{fopen}(\text{"password.txt"})\ \mathsf{in} \\
 & & \quad fct(c, a, b) \\
 & & \mathsf{end} \\
Dec_c & = & c : handle^s = \mathsf{fopen}(\text{"password.txt"}) \\
B_z & = & \mathsf{if}\ (x > 10)\ \mathsf{then}\ \mathsf{fprintf}\ (y, f)\ \mathsf{else}\ z := x; \mathsf{fprintf}(z, f) \\
\Delta & = & \Delta_a \dagger [fct \mapsto \langle (f, x, y), fct_B \rangle] \\
\Delta^{'} & = & \Delta \dagger [a \mapsto 5] \\
\Delta^{''} & = & \Delta^{'} \dagger [b \mapsto 8] \\
\Delta_1 & = & \Delta^{''} \dagger [c \mapsto handle_{path}] \\
\Delta_2 & = & \Delta_1 \dagger [f \mapsto handle_p, x \mapsto 5, y \mapsto 8] \\
\Delta_3 & = & \Delta_2 \dagger [z \mapsto 0] \\
\Delta_4 & = & \Delta_3 \dagger [z \mapsto 8] \\
path & = & (\text{"password.txt"})
\end{array}
$$

4.3 Sémantique statique

Les propriétés de sécurité que nous souhaitons vérifiées sont relatives aux actions effectuées par le programme et au flot de données de ce dernier. Nous allons donc définir un ensemble de mécanisme de contrôle du flot de données et du flot d'instructions.

4.3.1 Définitions

Définition 4.3.1 (Dépendance directe) *On dit qu'il existe une dépendance directe entre les variables d'un programme quand le contenu d'une variable x est directement influencé par la valeur d'autres variables du programme. Cette relation est notée $x \leftarrow (E)$, où E représente l'ensemble des variables dont dépend x ($x \notin E$).*

Définition 4.3.2 (Dépendance directe ou indirecte) *Une variable x dépend directement ou indirectement d'un ensemble de variables E si soit elle dépend directement de ces variables, soit elle dépend directement d'un ensembles de variables $E^{'}$ dont l'une des variables dépend directement ou indirectement des variables de l'ensemble E. Formellement, nous avons :*

$$\cfrac{\cfrac{\square}{\Delta_a \vdash \text{fun}\; fct(f : handle^s, x : int^p, y : int^s) = fct_{\mathbf{B}} \rightarrow \Delta}\;(\text{dec}) \quad A}{\vdash \text{let fun}\; fct(f : handle^s, x : int^p, y : int^s) = fct_{\mathbf{B}} \text{ in } Dec_{var} \rightarrow (), \Delta_4}\;(\text{pgm})$$

Arbre A

$$\cfrac{\cfrac{\square}{\Delta \vdash 5 \rightarrow 5, \Delta}\;(\text{cst}) \quad \cfrac{\cfrac{\square}{\Delta' \vdash 8 \rightarrow 8, \Delta}\;(\text{cst}) \quad \cfrac{B \quad C}{\Delta'' \vdash \text{letvar } Dec_c \text{ in } fct(c, a, b) \rightarrow (), \Delta_4}\;(\text{let})}{\Delta \vdash \text{letvar } b : int^s = 8 \text{ in } Dec_a \rightarrow (), \Delta_4}\;(\text{let})}{\Delta \vdash \text{letvar } a : int^p = 5 \text{ in } Dec_a \rightarrow (), \Delta_4}\;(\text{let})$$

Arbre B

$$\cfrac{\cfrac{\square}{\Delta'' \vdash path \rightarrow path, \Delta''}\;(\text{cst}) \quad \cfrac{\Delta(\text{fopen}) = \lambda_o}{\Delta_1 \vdash \text{fopen} \vdash \lambda_{\text{fopen}}, \Delta_1}\;(\text{api}) \quad handle_{path} = \lambda_o(path)}{\Delta'' \vdash \text{fopen}(path) \rightarrow handle_{path}, \Delta''}\;(\text{app}_2)$$

Arbre C

$$\cfrac{\cfrac{\Delta_1(c) = handle_{path}}{\Delta_1 \vdash c \rightarrow handle_{path}, \Delta_1}\;(\text{var}) \quad \cfrac{\Delta_1(a) = 5}{\Delta_1 \vdash a \rightarrow 5, \Delta_1}\;(\text{var}) \quad \cfrac{\Delta_1(b) = 8}{\Delta_1 \vdash b \rightarrow 8, \Delta_1}\;(\text{var}) \quad D \quad E}{\Delta_1 \vdash fct(c, a, b) \rightarrow (), \Delta_2}\;(\text{app}_1)$$

Arbre D

$$\cfrac{\Delta_1(fct) = \langle (f, x, y), fct_{\mathbf{B}} \rangle}{\Delta_1 \vdash fct \rightarrow \langle (f, x, y), fct_{\mathbf{B}} \rangle, \Delta_1}\;(\text{var})$$

Arbre E

$$\cfrac{\cfrac{\square}{\Delta_2 \vdash 0 \rightarrow 0, \Delta_2}\;(\text{cst}) \quad \cfrac{\cfrac{\cfrac{\Delta_3(x) = 5}{\Delta_3 \vdash x \rightarrow 5, \Delta_3}\;(\text{var}) \quad \cfrac{\square}{\Delta_3 \vdash 10 \rightarrow 10, \Delta_3}\;(\text{cst})}{\Delta_3 \vdash x > 10 \rightarrow false, \Delta_3}\;(\text{op}_\text{b}) \quad F}{\Delta_3 \vdash B_z \rightarrow (), \Delta_3}\;(\text{if}_1)}{\Delta_2 \vdash \text{letvar } (z : int^s = 0) \text{ in } B_z \text{ end} \rightarrow (), \Delta_4}\;(\text{let})$$

Arbre F

$$\cfrac{G \quad \cfrac{\cfrac{\square}{\Delta_3 \vdash \text{"}\%d\text{"} \rightarrow \text{"}\%d\text{"}, \Delta_3}\;(\text{cst}) \quad \cfrac{\Delta_4(z) = 5}{\Delta_4 \vdash z \rightarrow 5, \Delta_4}\;(\text{var}) \quad H \quad I \quad () = \lambda_p(\text{"}\%d\text{"}, z, f)}{\Delta_4 \vdash \text{fprintf}(\text{"}\%d\text{"}, z, f) \rightarrow (), \Delta_4}\;(\text{app}_2)}{\Delta_3 \vdash z := x; \text{fprintf}(\text{"}\%d\text{"}, z, f) \rightarrow (), \Delta_4}\;(\text{seq})$$

Arbre G

$$\cfrac{\cfrac{\Delta_3(x) = 5}{\Delta_3 \vdash x \rightarrow 5, \Delta_3}\;(\text{var})}{\Delta_3 \vdash z := x \rightarrow (), \Delta_4}\;(\text{aff})$$

Arbre H

$$\cfrac{\Delta_4(f) = handle_{path}}{\Delta_4 \vdash f \rightarrow handle_{path}, \Delta_4}\;(\text{var})$$

Arbre I

$$\cfrac{\Delta(\text{fprintf}) = \lambda_p}{\Delta_4 \vdash \text{fprintf} \vdash \lambda_p, \Delta_4}\;(\text{api})$$

TABLE 4.5 – Arbre sémantique du programme.

$$x \leftarrow^* (E) \;\Leftrightarrow\; (x \leftarrow (E) \;\vee\; (x \leftarrow (E') \;\wedge\; \exists x' \in E'.x' \leftarrow^* (E)))$$

Définition 4.3.3 (Environnement de dépendances) *Un environnement de dépendances, noté Γ, associe à chaque variable l'ensemble des variables dont elle dépend. On note Γ_p l'environnement des dépendances associées à un programme p. Cet environnement est mis à jour à chaque évaluation d'une expression d'un programme.*

Définition 4.3.4 (Mise à jour d'environnements de dépendances) *Les mises à jour des dépendances d'un programme sont définies au moyen de l'opérateur $\dagger$ qui permet d'actualiser l'ensemble Γ_1 à l'aide des nouvelles dépendances collectées dans Γ_2 comme suit :*

$$\Gamma_1 \dagger \Gamma_2 = \{(x \leftarrow (E)) \in \Gamma_1 \;\wedge\; (x \leftarrow (E')) \notin \Gamma_2\} \cup \Gamma_2.$$

En d'autres mots, la mise à jour des environnements Γ_1 et Γ_2 comporte toutes les dépendances $x \rightarrow (E)$ appartenant à Γ_1 tel qu'il n'existe aucune dépendance $x \leftarrow (E')$ dans l'environnement Γ_2 ainsi que toutes les dépendances de ce dernier : $\Gamma_1 \dagger \Gamma_2$ permet d'injecter les dépendances de Γ_2 dans l'environnement Γ_1.

Par exemple, on peut obtenir les dépendances de la séquence d'expressions ci-dessous comme suit :

$$\begin{array}{ll} x := y - 2; & \Gamma_1 = \{x \leftarrow (y)\} \\ k := y; & \Gamma_2 = \{x \leftarrow (y), k \leftarrow (y)\} = \Gamma_1 \dagger \{k \leftarrow (y)\} \\ x := z & \Gamma_3 = \{x \leftarrow (z), k \leftarrow (y)\} = \Gamma_2 \dagger \{x \leftarrow (z)\} \end{array}$$

4.3.2 Trace d'effets

L'un des objectifs de notre analyse statique est de construire un modèle de contrôle comportant l'ensemble des scénarios possibles d'exécution d'un programme. Aussi, un programme ne peut être valide que si (et seulement si) son modèle ne comporte aucun scénario dangereux. Ce modèle de contrôle, appelé *historique d'effets*, est construit à partir de la grammaire suivante :

$$\begin{array}{lcl} H & ::= & (\varepsilon, \Gamma) \mid (a, \Gamma) \mid H; H \mid (H \mid H) \mid \mu h.H \\ a & ::= & copy(x, y) \mid create(x) \mid delete(x) \end{array}$$

Un historique d'effets peut être un historique passif (ε, Γ), le couple (a, Γ) où a dénote une action critique relative à une instruction du code et Γ représente l'ensemble des dépendances engendrées au moment de l'exécution de cette dernière, la concaténation de

deux historiques d'effets $H_1; H_2$, un choix non-déterministe entre deux historiques d'effets $H_1 \mid H_2$ ou finalement $\mu h.H$ dénotant l'historique d'effets engendré à partir d'un bloc récursif.

L'action $copy(x, y)$ représente un transfert d'information de x vers y. Par exemple, en considérant que x et y abstraient deux zones mémoires distinctes, l'action $copy(x, y)$ souligne l'affectation du contenu de x à y. L'action $create(x)$ spécifie la création d'une ressource nommée x. Par exemple, nous allons décrire la représentation du programme ci-dessous selon les actions introduites précédemment :

Programme	**Abstraction**
File $*f$;	
char$information[25]$;	
$f := fopen$(Fichier);	$create(f)$
fscanf$(f, "\%s", information)$;	$copy(information, f)$
printf$(information)$;	$copy(screen, information)$
$fclose(f)$;	$delete(f)$

TABLE 4.6 – Exemple d'abstraction d'un programme.

Chaque historique d'effets peut être interprété sous forme d'un ensemble de traces d'exécution via un système de transitions étiquetées (*LTS*) adoptant $\downarrow$ comme symbole de terminaison. Il faut noter que certaines traces peuvent être infinies puisqu'un programme peut ne jamais s'arrêter. Aussi, une trace σ peut être construite à partir de la grammaire ci-dessous :

$$\begin{array}{rcl} \alpha & ::= & (\varepsilon, \Gamma) \mid (a, \Gamma) \\ \sigma & ::= & \alpha \mid \sigma\sigma \mid \sigma \downarrow \end{array}$$

Par ailleurs, nous pouvons introduire les propriétés suivantes sur les éléments composants une trace donnée :

$$\sigma(\varepsilon, \varnothing) = \sigma \qquad (\varepsilon, \varnothing)\sigma = \sigma \qquad (\sigma_1\sigma_2)\sigma_3 = \sigma_1(\sigma_2\sigma_3)$$

De plus, nous pouvons définir la relation de transition manipulant des traces d'effets comme suit :

$$(a, \Gamma) \xrightarrow{(a,\Gamma)} (\varepsilon, \varnothing) \qquad H_1|H_2 \xrightarrow{(\varepsilon,\varnothing)} H_1 \qquad H_1|H_2 \xrightarrow{(\varepsilon,\varnothing)} H_2 \qquad \mu h.H \xrightarrow{(\varepsilon,\varnothing)} H[\mu h.H/h]$$

$$(\varepsilon, \varnothing); H \xrightarrow{(\varepsilon,\varnothing)} H \qquad H_1; H_2 \xrightarrow{(a,\Gamma)} H_1'; H_2 \text{ si } H_1 \xrightarrow{(a,\Gamma)} H_1'$$

τ	$::=$	$r^l \mid \tau(x) \mid (\tau_1, \ldots, \tau_n) \xrightarrow{H} \tau \mid \alpha \mid$ *void*	(Types)
r	$::=$	*file* | *dir* | *net* | *host* | ... | *bool* | *int* | *float* | *char* | ρ	(Ressources)
l	$::=$	*secret* | *public* | *coded* | *password* | ... | ω	(Étiquettes)
H	$::=$	$(\varepsilon, \Gamma) \mid (a, \Gamma) \mid H; H \mid (H \mid H) \mid \mu h.H$	(Historiques d'effets)
a	$::=$	$copy(x, y) \mid create(x) \mid delete(x)$	(Actions Critiques)

TABLE 4.7 – Syntaxe du système de typage.

Enfin, nous pouvons définir l'interprétation d'un historique d'effets H, notée $[[H]]$, comme suit :

$$[[H]] = \{\alpha_1 \ldots \alpha_n \mid H \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} H'\} \cup \{\alpha_1 ... \alpha_n \downarrow \mid H \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \varepsilon\}$$

Lemme 4.3.1 *Les historiques d'effets sont soumis aux propriétés d'équivalence définies ci-dessous :*

1. $H \mid H = H$
2. $(\varepsilon, \varnothing); H = H; (\varepsilon, \varnothing) = H$
3. $H_1 \mid H_2 = H_2 \mid H_1$
4. $H_1; (H_2; H_3) = (H_1; H_2); H_3$
5. $H_1 \mid (H_2 \mid H_3) = (H_1 \mid H_2) \mid H_3$
6. $H_1; (H_2 \mid H_3) = (H_1; H_2) \mid (H_1; H_3)$
7. $(H_1 \mid H_2); H_3 = (H_1; H_3) \mid (H_2; H_3)$
8. $\mu h.\ H = H$ *si* $h \notin H$
9. $H[\mu h.\ H/h] = \mu h.\ H$
10. $\mu h.\ H; h = H; \ldots; H; \mu h.\ H; h$

Lemme 4.3.2 *En admettant que* $H \subseteq H'$ *si et seulement si* $[[H]] \subseteq [[H']]$*, nous pouvons définir les propriétés suivantes :*

1. *la relation* $H \subseteq H'$ *est indécidable*
2. *si* $H \subseteq H'$ *alors* $H \mid H'' \subseteq H' \mid H''$
3. *si* $H \subseteq H'$ *alors* $H; H'' \subseteq H'; H''$
4. *si* $H \subseteq H'$ *alors* $H''; H \subseteq H''; H'$
5. *si* $H \subseteq H'$ *alors la validité de l'historique* H' *implique celle de l'historique* H

4.3.3 Système de typage

On dénote Θ_P comme étant l'ensemble des types associés aux variables d'un programme P. Cet ensemble est mis à jour à chaque fois qu'une nouvelle information est

collectée concernant le type d'une donnée. Chaque élément de Θ_P définit l'association d'un type t à une variable var ($var : \tau$).

Afin d'incorporer la vérification du flot d'instructions et celui des données, nous proposons un système de typage pouvant inclure les concepts d'historique d'effets et des dépendances directes introduits précédemment. Aussi, un programme est dit *correct* si les modèles Θ et H assurent la cohérence de ce dernier. Ces modèles sont construits au moment de la compilation du programme. Le système de typage peut alors être construit à partir de la syntaxe définie dans la table 4.7.

Chaque variable d'un programme est associée à un type spécifiant la ressource qu'elle abstrait et le mode d'accès à cette dernière [2]. L'idée est de préciser qu'une variable f est par exemple un fichier publique ; on peut alors écrire : $f : file^p$. Chaque fonction introduite dans un programme, prenant un argument de type τ_1 et retourne un résultat de type τ_2, est associé au type $\tau_1 \xrightarrow{H} \tau_2$ où H dénote l'historique d'effets engendrés au moment de l'exécution de cette dernière.

Le système de typage, défini dans la table 4.8 (page 75), manipule des jugements de la forme $\Theta, H \vdash p : \tau$, où τ dénote le type du programme p étant donné l'environnement de typage Θ, H représente l'historique d'effets engendré au moment de l'évaluation de p. De plus, nous admettons qu'il existe une fonction, Typeof, permettant de déterminer les types associés aux constantes et aux APIs. Cette fonction peut être définie comme suit :

$$\begin{array}{lcl} \mathsf{Typeof}(1) & = & \mathsf{int} \\ \mathsf{Typeof}(\mathsf{true}) & = & \mathsf{bool} \\ & \cdots & \end{array}$$

Formellement, la validité d'un jugement peut être définie comme suit :

Définition 4.3.5 *Un jugement $\Theta, H \vdash p : \tau$ est dit valide s'il est dérivable.*

Nous pouvons ainsi obtenir la proposition suivante en étendant les propriétés introduites dans [4] :

Proposition 4.3.1 *Si le jugement $\Theta, H \vdash p : \tau$ est dérivable pour un programme P alors H est valide.*

À partir des règles de typage précédentes, nous pouvons construire l'arbre de typage, défini dans la table 4.9 (page 77), relatif à l'exemple introduit dans la table 4.4 (page 68) à partir des notations suivantes :

---

2. Il faut noter que chaque constante c d'un programme est associée au type *singleton* $\tau(c)$.

$$\frac{\mathsf{Typeof}(c) = \tau}{\Theta, (\varepsilon, \varnothing) \vdash c : \tau}\ (\mathsf{const}) \qquad \frac{\Theta(x) = \tau}{\Theta, (\varepsilon, \varnothing) \vdash x : \tau}\ (\mathsf{var})$$

$$\frac{\mathsf{Typeof}(api) = \tau}{\Theta, (\varepsilon, \varnothing) \vdash api : \tau}\ (\mathsf{api}) \qquad \frac{\Theta, H \vdash e : \tau}{\Theta, H|H' \vdash e : \tau}\ (\mathsf{weaken})$$

$$\frac{\Theta, H \vdash e : (\tau_1, \ldots, \tau_n) \xrightarrow{H'} \tau \qquad \forall i \in 1 \ldots n : \Theta, H_i \vdash e_i : \tau_i}{\Theta, H; H_1; \ldots; H_n; H' \vdash e(e_1, \ldots, e_n) : \tau}\ (\mathsf{app})$$

$$\frac{\mathsf{Typeof}(op_b) = (\tau \times \tau') \rightarrow \tau'' \qquad \Theta, H \vdash e : \tau \qquad \Theta, H' \vdash e' : \tau'}{\Theta, H; H' \vdash e\ \mathsf{op_b}\ e' : \tau''}\ (\mathsf{op_b})$$

$$\frac{\mathsf{Typeof}(op_u) = \tau \rightarrow \tau' \qquad \Theta, H \vdash e : \tau}{\Theta, H \vdash \mathsf{op_u}\ e : \tau'}\ (\mathsf{op_u})$$

$$\frac{\Theta, H \vdash e : \tau \qquad \Theta, (\varepsilon, \varnothing) \vdash x : \tau}{\Theta, H \vdash x := e : unit}\ (\mathsf{aff})$$

$$\frac{\Theta, H \vdash e : \tau \qquad \Theta, H' \vdash e' : \tau'}{\Theta, H; H' \vdash e; e' : \tau'}\ (\mathsf{seq})$$

$$\frac{\Theta, H \vdash e : bool \qquad \Theta, H' \vdash e' : \tau}{\Theta, \mu h.(H; H'; h) \vdash \mathsf{while}\ e\ \mathsf{do}\ e' : \tau}\ (\mathsf{while})$$

$$\frac{\Theta, H \vdash e : bool \qquad \Theta, H' \vdash e' : \tau \qquad \Theta, H'' \vdash e'' : \tau}{\Theta, H; (H'|H'') \vdash \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : \tau}\ (\mathsf{if})$$

$$\frac{\Theta, H \vdash e : \tau \qquad \Theta \cup \{x : \tau\}, H' \vdash e' : \tau'}{\Theta, H; H' \vdash \mathsf{letvar}\ x : \tau = e\ \mathsf{in}\ e'\ \mathsf{end} : \tau'}\ (\mathsf{let})$$

$$\frac{\Theta \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}, H \vdash e : \tau}{\Theta, (\varepsilon, \varnothing) \vdash \mathsf{fun}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) = e : (\tau_1, \ldots, \tau_n) \xrightarrow{H} \tau}\ (\mathsf{fun})$$

$$\frac{\begin{array}{c}\forall i = 1 \ldots n : \Theta, (\varepsilon, \varnothing) \vdash d_i : \tau_i \xrightarrow{H_i} \tau_i' \\ \Theta \cup \{d_1 : \tau_1 \xrightarrow{H_1} \tau_1', \ldots, d_m : \tau_n \xrightarrow{H_n} \tau_n'\}, H \vdash e : \tau\end{array}}{\varnothing, H \vdash \mathsf{let}\ d_1, \ldots, d_m\ \mathsf{in}\ e : \tau}\ (\mathsf{pgm})$$

TABLE 4.8 – Règles de typage.

$$
\begin{array}{lcl}
path & = & \textit{"password.txt"} \\
D_{fct} & = & \mathsf{letvar}\ (z : \textit{int}^p = 0)\ \mathsf{in} \\
 & & \quad \mathsf{if}\ (x > 10)\ \mathsf{then\ fprintf}\ ("\%d", y, f)\ \mathsf{else}\ z := x; \mathsf{fprintf}("\%d", z, f) \\
 & & \ \mathsf{end} \\
C_{fct} & = & \mathsf{if}\ (x > 10)\ \mathsf{then\ fprintf}\ ("\%d", y, f)\ \mathsf{else}\ z := x; \mathsf{fprintf}("\%d", z, f) \\
D_a & = & a : \textit{int}^p = 5 \\
D_b & = & b : \textit{int}^s = 8 \\
D_c & = & c : \textit{handle}^s = \mathsf{fopen}(\textit{"password.txt"}) \\
D_z & = & z : \textit{int}^p = 0 \\
H & = & (\mathsf{copy}(y, f), \{f \leftarrow y\}) | ((\varepsilon, \{z \leftarrow x\}); (\mathsf{copy}(z, f), \{f \leftarrow y\})) \\
H_1 & = & (\mathsf{copy}(y, f), \{f \leftarrow y\}) \\
H_2 & = & ((\varepsilon, \{z \leftarrow x\}); (\mathsf{copy}(z, f), \{f \leftarrow y\})) \\
H_3 & = & (\varepsilon, \{z \leftarrow x\}) \\
H_4 & = & (\mathsf{copy}(z, f), \{f \leftarrow y\}) \\
H_c & = & (\mathsf{create}(f), \varnothing); H \\
\epsilon & = & (\varepsilon, \varnothing) \\
\Theta & = & \{f : \textit{handle}^s, x : \textit{int}^p, y : \textit{int}^s\} \\
\Theta' & = & \Theta \cup \{z : \textit{int}^p\} \\
\Theta'' & = & \Theta' \cup \{\mathrm{fct} : (\textit{file}^s, \textit{int}^p, \textit{int}^s) \xrightarrow{H} unit\} \\
\Theta_1 & = & \Theta'' \cup \{a : \textit{int}^p\} \\
\Theta_2 & = & \Theta_1 \cup \{b : \textit{int}^s\} \\
\Theta_3 & = & \Theta_2 \cup \{c : \textit{handle}^s\}
\end{array}
$$

Cet arbre démontre que le programme est de type *unit* et que son historique d'effets H_c est valide.

4.3.4 Algorithme d'inférence de types et d'effets

Dans cette section, nous présentons un algorithme d'inférence de types et d'effets permettant d'extraire, automatiquement et de manière correcte, un modèle à partir d'un programme. La correction des résultats fournis par cet algorithme est obtenu grâce aux règles de typage étudiées dans la précédente section.

De manière semi-formelle, étant donné un programme p, l'algorithme d'inférence de types et d'effets génère à la fois un type τ, une trace d'effets H et un environnement de typage Θ. Ce dernier comprend les types des différentes variables utilisées par le programme, notamment celles utilisées dans la trace d'effets générée H. Ces résultats sont alors validés par l'entremise du système de types, notamment en construisant un arbre de preuve du séquent suivant :

$$\Theta, H \vdash p : \tau$$

$$\frac{\dfrac{\dfrac{\dfrac{\text{Typeof}(0)=int}{\Theta,\epsilon\vdash 0:int}\,(\text{const})\quad \dfrac{A\quad B\quad C}{\Theta',H\vdash C_{fct}:unit}\,(\text{if})}{\Theta,H\vdash \text{letvar } D_z \text{ in } C_{fct} \text{ end}:unit}\,(\text{let})}{\varnothing,\epsilon\vdash D_{fct}:(handle^s,int^p,int^s)\xrightarrow{H} unit}\,(\text{fun})\qquad E}{\varnothing,H_c\vdash \text{let } D_{fct} \text{ in let } D_a \text{ in let } D_b \text{ in let } D_c \text{ in } fct(c,a,b):unit}\,(\text{pgm})$$

Arbre A

$$\frac{\text{Typeof}(>)=(int^p,int)\rightarrow bool\quad \dfrac{\Theta'(x)=int^p}{\Theta',\epsilon\vdash x:int^p}\,(\text{var})\quad \dfrac{\text{Typeof}(10)=int}{\Theta,\epsilon\vdash 10:int}\,(\text{const})}{\Theta',\epsilon\vdash x>10:bool}\,(\text{op}_\text{b})$$

Arbre B

$$\frac{\dfrac{\text{Typeof}(\text{printf})=(String,int^s,handle^s)\xrightarrow{H_1} unit}{\Theta',\epsilon\vdash \text{fprintf}:(String,int^s,handle^s)\xrightarrow{H_1} unit}\,(\text{api})\quad B'\quad \dfrac{\Theta'(y)=int^s}{\Theta',\epsilon\vdash y:int^s}\,(\text{var})\quad \dfrac{\Theta'(f)=handle^s}{\Theta',\epsilon\vdash f:handle^s}\,(\text{var})}{\Theta',H_1\vdash \text{fprintf}(\text{"\%d"},y,f):unit}\,(\text{app})$$

Arbre B'

$$\frac{\text{Typeof}(\text{"\%d"})=String}{\Theta',\epsilon\vdash \text{"\%d"}:String}\,(\text{const})$$

Arbre C

$$\frac{\dfrac{\dfrac{\Theta'(x)=int^p}{\Theta',\epsilon\vdash x:int^p}\,(\text{var})\quad \dfrac{\Theta'(z)=int^p}{\Theta',\epsilon\vdash z:int^p}\,(\text{var})}{\Theta',H_3\vdash z:=x:unit}\,(\text{affE})\quad D}{\Theta',H_2\vdash z:=x;\text{fprintf}(\text{"\%d"},z,f):unit}\,(\text{seq})$$

Arbre D

$$\frac{\dfrac{\text{Typeof}(\text{fprintf})=(String,int^p,handle^s)\xrightarrow{H_4} unit}{\Theta',\epsilon\vdash \text{fprintf}:(String,int^p,handle^s)\xrightarrow{H_4} unit}\,(\text{api})\quad B'\quad \dfrac{\Theta'(z)=int^p}{\Theta',\epsilon\vdash z:int^p}\,(\text{var})\quad \dfrac{\Theta'(f)=handle^s}{\Theta',\epsilon\vdash f:handle^s}\,(\text{var})}{\Theta',H_4\vdash \text{fprintf}(\text{"\%d"},z,f):unit}\,(\text{app})$$

Arbre E

$$\frac{\dfrac{\text{Typeof}(5)=int}{\Theta'',\epsilon\vdash 5:int}\,(\text{const})\quad \dfrac{\dfrac{\text{Typeof}(8)=int}{\Theta_1,\epsilon\vdash 8:int}\,(\text{const})\quad \dfrac{F\quad G}{\Theta_2,H_c\vdash \text{ let } D_c \text{ in } fct(c,a,b):unit}\,(\text{let})}{\Theta_1,H_c\vdash \text{ in let } D_b \text{ in let } D_c \text{ in } fct(c,a,b):unit}\,(\text{let})}{\Theta'',H_c\vdash \text{let } D_a \text{ in let } D_b \text{ in let } D_c \text{ in } fct(c,a,b):unit}\,(\text{let})$$

Arbre F

$$\frac{\dfrac{\text{Typeof}(\text{fopen})=String\xrightarrow{H_c} handle^s}{\Theta_2,\epsilon\vdash \text{fopen}:String\xrightarrow{H_c} handle^s}\,(\text{api})\quad \dfrac{\text{Typeof}(path)=String}{\Theta_2,\epsilon\vdash path:String}\,(\text{const})}{\Theta_2,H_c\vdash \text{fopen}(c):unit}\,(\text{app})$$

Arbre G

$$\frac{\dfrac{\Theta_3(fct)=(handle^s,int^p,int^s)\xrightarrow{H} unit}{\Theta_3;\epsilon\vdash fct:(handle^s,int^p,int^s)\xrightarrow{H} unit}\,(\text{var})\quad \dfrac{\Theta_3(c)=handle^s}{\Theta_3,\epsilon\vdash c:file^s}\,(\text{var})\quad \dfrac{\Theta_3(a)=int^p}{\Theta_3,\epsilon\vdash a:int^p}\,(\text{var})\quad H}{\Theta_3,H\vdash fct(c,a,b):unit}\,(\text{app})$$

Arbre H

$$\frac{\Theta_3(b)=int^s}{\Theta_3,\epsilon\vdash b:int^s}\,(\text{var})$$

TABLE 4.9 – Arbre de typage de l'exemple précédent.

Afin de présenter cet algorithme, plusieurs notions doivent préalablement être définies. Cette section suit donc le plan suivant : Dans un premier temps, nous motivons la nécessité de disposer d'une notion de substitution afin de mettre à jour éventuellement les types déjà établis de certaines variables ou fonctions. Par la suite, nous proposons un algorithme de calcul d'un unificateur principal de deux types donnés. Le résultat de cet algorithme est une substitution. Avant de proposer l'algorithme d'inférence, nous motivons le besoin de disposer d'une variable spéciale, nommée it, représentant la dernière expression évaluée (statiquement). Nous y associons un opérateur permettant de retourner les variables dont dépend cette variable it dans un environnement de dépendances bien particulier. Nous terminons cette section par la présentation de l'algorithme et son illustration à travers un exemple.

Substitution

Une substitution μ est une fonction partielle qui associe à des variables de types des expressions de type. Trois fonctions sont définies sur les substitutions :

- $dom(\mu) = \{\alpha \mid (\alpha \mapsto \tau) \in \mu\}$
- $Im(\mu) = \{\tau \mid (\alpha \mapsto \tau) \in \mu\}$
- $\mu \circ \mu' = \mu \dagger [\alpha'_i \mapsto \mu(\tau'_i) \mid (\alpha'_i \mapsto \tau'_i) \in \mu']$
  avec $\mu = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$
  $\mu' = [\alpha'_1 \mapsto \tau'_1, \ldots, \alpha'_m \mapsto \tau'_m]$

Notons que pour un type τ, on a :

$$(\mu \circ \mu')(\tau) \equiv \mu(\mu'(\tau))$$

Par ailleurs, il est possible d'appliquer une substitution μ à un type τ ainsi qu'a un environnement de typage Θ :

- $\mu(\tau)$:
  $\mu(r^l) = (\mu(r))^{\mu(l)}$
  $\mu(\tau(x)) = (\mu(\tau))(x)$
  $\mu(\tau_1 \times \ldots \times \tau_n \xrightarrow{H} \tau) = \mu(\tau) \times \ldots \times \mu(\tau_n) \xrightarrow{H} \mu(\tau)$
  $\mu(\mathit{void}) = \mathit{void}$
  $(\mu \dagger [\alpha \mapsto \tau])(\alpha) = \tau$

- $\mu(r)$:
  $\mu(\mathit{file}) = \mathit{file}$
  $\mu(\mathit{dir}) = \mathit{dir}$

$$\begin{aligned} &\vdots \\ \mu(\mathit{char}) &= \mathit{char} \\ (\mu \dagger [\rho \mapsto r])(\rho) &= r \end{aligned}$$

– $\mu(l)$:

$$\begin{aligned} \mu(\mathit{secret}) &= \mathit{secret} \\ \mu(\mathit{public}) &= \mathit{public} \\ &\vdots \\ (\mu \dagger [\omega \mapsto l])(\omega) &= l \end{aligned}$$

– $\mu(\Theta)$:

$$\mu(\Theta) = \{x_i \mapsto \mu(st_i) \mid (x_i \mapsto st_i) \in \Theta\}$$

Unificateur principal

L'algorithme qui calcule l'unificateur principal utilise d'autre fonctions permettant de calculer l'unificateur principal de respectivement deux types de ressources (r) et deux étiquettes (l).

L'algorithme qui calcule l'unificateur principal de deux étiquettes l et l' est décrit ci-dessous :

$$\begin{array}{lcll} \mathit{mgu}_l(l, l') & = & \textbf{case } (l, l') \textbf{ of} & \\ & & (\omega, l) \mid (l, \omega) & \rightarrow \ [\omega \mapsto l] \\ & & (l, l') & \rightarrow \ [l \mapsto l \sqcup l', l' \mapsto l \sqcup l'] \\ & & \textbf{else} & \rightarrow \ \mathsf{fail} \\ & & \textbf{end} & \end{array}$$

La fonction $\sqcup$ est définie en fonction des étiquettes manipulées. Par exemple, pour les étiquettes *public* et *secret*, cette fonction est définie comme suit :

$$\begin{aligned} \tau_1 \sqcup \tau_2 = \tau_1 \ \text{ si } \tau_2 \sqsubseteq \tau_1 \\ \tau_1 \sqcup \tau_2 = \tau_2 \ \text{ si } \tau_1 \sqsubseteq \tau_2 \end{aligned}$$

avec la relation d'ordre, $\sqsubseteq$, qui classifie les étiquettes, définie comme suit :

$$\begin{array}{lcl} \mathit{public} & \sqsubseteq & \mathit{public} \\ \mathit{public} & \sqsubseteq & \mathit{secret} \\ \mathit{secret} & \sqsubseteq & \mathit{secret} \end{array}$$

L'algorithme qui calcule l'unificateur principal de deux types de ressources r^l et $r'^{l'}$ est

défini comme suit :

$$\begin{array}{rcl}
mgu_r(r^l, r'^{l'}) & = & \textbf{case } (r^l, r'^{l'}) \textbf{ of} \\
(r^l, r^{l'}) & \rightarrow & mgu_l(l, l') \\
(\rho^l, r^{l'}) \mid (r^{l'}, \rho^l) & \rightarrow & [\rho \mapsto r] \circ (mgu_l(l, l')) \\
\textbf{else} & \rightarrow & \mathsf{fail} \\
\textbf{end} & &
\end{array}$$

Finalement, l'algorithme qui permet de calculer l'unificateur principal de deux types τ_1 et τ_2 est défini comme suit :

$$\begin{array}{rcl}
mgu(\tau_1, \tau_2) & = & \textbf{case } (\tau_1, \tau_2) \textbf{ of} \\
(void, void) & \rightarrow & [\,] \\
(\tau(x), \tau(x')) & \rightarrow & \textbf{if } (x = x') \textbf{ then } [\,] \textbf{ else } \mathsf{fail} \\
(r^l, r'^{l'}) & \rightarrow & mgu_r(\tau_1, \tau_2) \\
(\alpha, \alpha') & \rightarrow & [\alpha \mapsto \alpha'] \\
(\alpha, \tau) \mid (\tau, \alpha) & \rightarrow & \textbf{if } \alpha \in fv(\tau) \textbf{ then } \mathsf{fail} \textbf{ else } [\alpha \mapsto \tau] \\
(\tau_1 \times \ldots \times \tau_n \xrightarrow{H} \tau, \tau'_1 \times \ldots \times \tau'_n \xrightarrow{H} \tau') & \rightarrow & \textbf{let } \mu_1 = mgu(\tau_1, \tau'_1) \\
& & \quad \vdots \\
& & \quad \mu_n = mgu(\mu_{n-1}(\tau_n), \mu_{n-1}(\tau'_n)) \\
& & \quad \mu = mgu(\mu_n(\tau), \mu_n(\tau')) \\
& & \textbf{in} \\
& & \quad \mu \circ \mu_n \circ \ldots \circ \mu_1 \\
& & \textbf{end} \\
\textbf{else} & \rightarrow & \mathsf{fail} \\
\textbf{end} & &
\end{array}$$

Notons que pour le calcul de l'unificateur principal de deux types de fonctions, il est requis que ces types aient la même trace d'effets H. Comme travaux futurs, nous pensons traiter le cas où les deux traces d'effets sont différents, mais aussi au cas où on utilise des traces d'effets variables (abstraction de traces d'effets).

Par ailleurs, la fonction *mgu* est naturellement étendue aux environnements de typages Θ.

Variable *it*

Lors de l'appel de fonction, il est nécessaire de disposer d'un mécanisme permettant de déterminer les variables dont dépend le résultat de la fonction. Ainsi, pour l'expression $x := e$ et en considérant que l'expression e correspond à un appel de fonction, il sera possible de déterminer les variables utilisées dans la fonction et qui affecte la valeur de

retour ne cette fonction. Il sera aussi possible de conclure que le contenu de la variable x dépend de ces variables.

Afin de résoudre ce problème, nous intrduisons une variable sémantique *it* dont le rôle est de déterminer toutes les variables dont dépend la valeur de la dernière expression évaluée statiquement.

Par ailleurs, pour accéder au contenu de la variable it dans le dernier environnement de dépendance Γ définie dans une trace d'effets H, nous définissons la fonction suivante :

$$\begin{array}{rcl} _^{\downarrow} & : & \mathit{Trace} \rightarrow \mathit{Vars} \\ \\ (a,\Gamma)^{\downarrow} & = & \Gamma(\mathit{it}) \\ (\epsilon,\Gamma)^{\downarrow} & = & \Gamma(\mathit{it}) \\ (H_1;H_2)^{\downarrow} & = & (H_2^{\downarrow})(\mathit{it}) \\ (H_1 \mid H_2)^{\downarrow} & = & (H_1^{\downarrow} \cup H_2^{\downarrow})(\mathit{it}) \\ (\mu h.H;h)^{\downarrow} & = & H^{\downarrow} \\ (\mu h.h)^{\downarrow} & = & () \end{array}$$

L'union de deux environnements de dépendances Γ_1 et Γ_2 est définie comme suit :

$$\begin{array}{rcl} \Gamma_1 \cup \Gamma_2 & = & \{x \leftarrow E \mid x \in (\mathit{dom}(\Gamma_1) \backslash \mathit{dom}(\Gamma_2))\} \\ & \cup & \{x \leftarrow E \mid x \in (\mathit{dom}(\Gamma_2) \backslash \mathit{dom}(\Gamma_1))\} \\ & \cup & \{x \leftarrow (E \cup E') \mid (x \leftarrow (E)) \in \Gamma_1 \ \wedge \ (x \leftarrow (E')) \in \Gamma_2\} \end{array}$$

Algorithme d'inférence

Dans cette section, nous présentons l'algorithme d'inférence de types et d'effets. Cet algorithme est composé de trois fonctions :

1. $\mathsf{inf}_{\mathsf{p}}$: *Prog* $\rightarrow$ *Type* $\times$ *Env* $\times$ *Trace*
   Cette fonction prend en argument un programme p et retourne en résultat un type τ, un environnement Θ et une trace d'effets H.
2. $\mathsf{inf}_{\mathsf{d}}$: *Env* $\times$ *Dec* $\rightarrow$ *Env*
   Cette fonction prend en argument un environnement Θ et une déclaration de fonction d et retourne en résultat un nouvel environnement Θ'. Ce dernier comprend la signature de la fonction passée en argument.
3. $\mathsf{inf}_{\mathsf{e}}$: *Env* $\times$ *Exp* $\rightarrow$ *Type* $\times$ *Env* $\times$ *Trace* $\times$ *Subs*
   Cette fonction prend en argument un environnement Θ et une expresion e et retourne en résultat un type τ, un environnement Θ', une trace d'effets H et une substitution ρ.

Les tables 4.10, 4.11, 4.12, 4.13 et 4.14 présentent ces fonctions qui définissent l'algorithme d'inférence de types et d'effets.

$$
\begin{array}{l}
\mathsf{inf_p} : \mathit{Prog} \rightarrow \mathit{Type} \times \mathit{Env} \times \mathit{Trace} \\
\\
\mathsf{inf_p}\ (\mathsf{let}\ d_1 \ldots d_n\ \mathsf{in}\ e) = \\
\quad \textbf{let}\ \Theta_1 = \mathsf{inf_d}^{\Theta_a}(d_1) \\
\qquad \vdots \\
\qquad \Theta_n = \mathsf{inf_d}^{\theta_{n-1}}(d_n) \\
\qquad (\tau, \Theta, H, \rho) = \mathsf{inf_e}^{\theta_n}(e) \\
\quad \textbf{in} \\
\qquad (\tau, \Theta, H) \\
\quad \textbf{end}
\end{array}
$$

TABLE 4.10 – Algorithme d'inférence de types et d'effets (programmes).

$$
\begin{array}{l}
\mathsf{inf_d} : \mathit{Env} \times \mathit{Dec} \rightarrow \mathit{Env} \\
\\
\mathsf{inf_d^\Theta}\ (\mathsf{fun} f\ (x_1 : \tau_1 \ldots x_n : \tau_n) : \tau = e) = \\
\quad \textbf{let}\ (\tau', \Theta', H, \rho) = \mathsf{inf_e}^{\Theta \dagger [x_1:\tau_1,\ldots,x_n:\tau_n]}(e) \\
\qquad \rho' = \mathit{mgu}(\tau, \tau') \\
\qquad \rho'' = \rho' \circ \rho \\
\quad \textbf{in} \\
\qquad \rho''(\Theta') \dagger [f : x_1 : \rho''(\tau_1) \times \ldots \times x_n : \rho''(\tau_n) \xrightarrow{H} \rho''(\tau)] \\
\quad \textbf{end}
\end{array}
$$

TABLE 4.11 – Algorithme d'inférence de types et d'effets (fonctions).

L'environnement Θ_a comprend la signature des différentes fonctions API utilisées dans le programme analysé.

4.4 Vérification de propriétés de sécurité

Dans cette section, nous nous intéressons à la vérification du comportement d'un programme relativement à une propriété de sécurité donnée. Plus précisément, nous utilisons

$$\mathsf{inf_e} : \mathit{Env} \times \mathit{Exp} \rightarrow \mathit{Type} \times \mathit{Env} \times \mathit{Trace} \times \mathit{Subs}$$

$$\mathsf{inf}_\mathsf{e}^\Theta(c) = (\mathit{TypeOf}(c), \Theta, (\epsilon, [it \leftarrow ()]))$$

$$\mathsf{inf}_\mathsf{e}^\Theta(\mathit{api}) = (\Theta(\mathit{api}), \Theta, (\epsilon, [it \leftarrow (\mathit{api})]))$$

$$\begin{array}{l}
\mathsf{inf}_\mathsf{e}^\Theta\ (x := e) = \\
\quad \textbf{let}\ (\tau, \Theta, H, \rho) = \mathsf{inf}_\mathsf{e}^\Theta(e) \\
\qquad \rho' = \mathit{mgu}(\tau, \Theta(x)) \\
\qquad \rho'' = \rho' \circ \rho \\
\quad \textbf{in} \\
\qquad (\mathit{unit}, \rho''(\Theta \dagger [x : \tau]), H; (\epsilon, [it \leftarrow (), x \leftarrow H^{\downarrow}]), \rho'') \\
\quad \textbf{end}
\end{array}$$

$$\begin{array}{l}
\mathsf{inf}_\mathsf{e}^\Theta\ (e; e') = \\
\quad \textbf{let}\ (\tau, \Theta, H, \rho) = \mathsf{inf}_\mathsf{e}^\Theta(e) \\
\qquad (\tau', \Theta', H', \rho') = \mathsf{inf}_\mathsf{e}^{\rho(\Theta)}(e') \\
\quad \textbf{in} \\
\qquad (\tau', \Theta', H; H', \rho' \circ \rho) \\
\quad \textbf{end}
\end{array}$$

$$\begin{array}{l}
\mathsf{inf}_\mathsf{e}^\Theta\ (\mathsf{if} e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) = \\
\quad \textbf{let}\ (\tau, \Theta, H, \rho) = \mathsf{inf}_\mathsf{e}^\Theta(e) \\
\qquad (\tau_1, \Theta_1, H_1, \rho_1) = \mathsf{inf}_\mathsf{e}^{\rho(\Theta)}(e_1) \\
\qquad (\tau_2, \Theta_2, H_2, \rho_2) = \mathsf{inf}_\mathsf{e}^{\rho(\Theta)}(e_2) \\
\qquad \rho' = \mathit{mgu}(\tau_1, \tau_2) \\
\qquad \rho'' = \mathit{mgu}(\tau, \mathit{bool}) \\
\qquad \rho''' = \mathit{mgu}(\Theta_1, \Theta_2) \\
\qquad \rho_f = \rho''' \circ \rho'' \circ \rho' \circ \rho \\
\quad \textbf{in} \\
\qquad (\rho_f(\tau_1), \rho_f(\Theta_1), H_b; (H_1 | H_2); (\epsilon, [it \leftarrow H_1^{\downarrow} \cup H_2^{\downarrow}]), \rho_f) \\
\quad \textbf{end}
\end{array}$$

$$\begin{array}{l}
\mathsf{inf}_\mathsf{e}^\Theta\ (\mathsf{while}\ e\ \mathsf{do}\ e') = \\
\quad \textbf{let}\ (\tau, \Theta, H, \rho) = \mathsf{inf}_\mathsf{e}^\Theta(e) \\
\qquad (\tau', \Theta', H', \rho') = \mathsf{inf}_\mathsf{e}^{\rho(\Theta)}(e') \\
\quad \textbf{in} \\
\qquad (\mathit{unit}, \Theta', (\mu h.H; H'; h); (\epsilon, [it \leftarrow ()]), \rho' \circ \rho) \\
\quad \textbf{end}
\end{array}$$

TABLE 4.12 – Algorithme d'inférence de types et d'effets (expressions (1/3)).

$$
\begin{array}{l}
\mathsf{inf}_{\mathsf{e}}^{\Theta}\ (\mathsf{letvar}\ x:\tau=e\ \mathsf{in}\ e'\ \mathsf{end}) = \\
\quad \textbf{let}\ (\tau',\Theta,H,\rho) = \mathsf{inf}_{\mathsf{e}}^{\Theta}(e) \\
\quad\quad \rho' = \textit{mgu}(\tau,\tau') \\
\quad\quad (\tau'',\Theta',H',\rho'') = \mathsf{inf}_{\mathsf{e}}^{(\rho'\circ\rho)(\Theta)\dagger[x:\rho'(\tau)]}(e') \\
\quad\quad \rho_f = \rho''\circ\rho'\circ\rho \\
\quad \textbf{in} \\
\quad\quad (\tau'',\rho_f(\Theta'\dagger[x:\tau]),H;H',\rho_f) \\
\quad \textbf{end} \\
\\
\mathsf{inf}_{\mathsf{e}}^{\Theta}\ (e_1\ \mathsf{op}_b\ e_2) = \\
\quad \textbf{let}\ (\tau_1,\Theta_1,H_1,\rho_1) = \mathsf{inf}_{\mathsf{e}}^{\Theta}(e_1) \\
\quad\quad (\tau_2,\Theta_2,H_2,\rho_2) = \mathsf{inf}_{\mathsf{e}}^{\rho_1(\Theta)}(e_2) \\
\quad\quad \tau = \textit{TypeOf}\,(\mathsf{op}_b) \\
\quad\quad \rho_3 = \textit{mgu}(\tau_1\times\tau_2 \xrightarrow{\ \cdot\ } \alpha,\tau) \\
\quad\quad \rho_f = \rho_3\circ\rho_2\circ\rho_1 \\
\quad \textbf{in} \\
\quad\quad (\rho_f(\alpha),\rho_f(\Theta_2),H_1;H_2;(\epsilon,[it\leftarrow H_1^{\downarrow}\cup H_2^{\downarrow}]),\rho_f) \\
\quad \textbf{end} \\
\\
\mathsf{inf}_{\mathsf{e}}^{\Theta}\ (\mathsf{op}_u\ e) = \\
\quad \textbf{let}\ (\tau,\Theta,H,\rho) = \mathsf{inf}_{\mathsf{e}}^{\Theta}(e) \\
\quad\quad \tau' = \textit{TypeOf}\,(\mathsf{op}_u) \\
\quad\quad \rho' = \textit{mgu}(\tau \xrightarrow{\ \cdot\ } \alpha,\tau') \\
\quad\quad \rho'' = \rho'\circ\rho \\
\quad \textbf{in} \\
\quad\quad (\rho''(\alpha),\rho''(\Theta),H,\rho'') \\
\quad \textbf{end}
\end{array}
$$

TABLE 4.13 – Algorithme d'inférence de types et d'effets (expressions (2/3)).

la vérification par évaluation de modèles (*model checking*). Ainsi, pour vérifier le comportement d'un programme, nous utilisons un modèle de ce dernier ; dans notre cas, il s'agit d'un modèle généré par l'algorithme d'extraction de modèles présenté dans la section précédente. Ce modèle est confronté à une propriété spécifiée par une logique, plus précisément une logique modale (μ-*calcul* modale).

Cette section est organisée comme suit. On rappelle dans un premier temps la syntaxe du μ-*calcul* modale et illustrons son expressivité à travers des exemples. Par la suite, nous présentons la logique que nous avons définie pour la spécification de propriétés. Cette logique est basée sur le μ-*calcul* modale dans lequel les actions manipulent des variables et des types. Nous définissons une sémantique dénotationnelle et nous illustrons l'expressivité de cette logique à travers des exemples.

$$
\begin{array}{l}
\mathsf{inf}_{\mathsf{e}}^{\Theta}\ (e\,(e_1,\ldots,e_n)) = \\
\quad \textbf{let}\ (\tau,\Theta,H,\rho) = \mathsf{inf}_{\mathsf{e}}^{\Theta}(e) \\
\qquad (\tau_1,\Theta_1,H_1,\rho_1) = \mathsf{inf}_{\mathsf{e}}^{\rho(\Theta)}(e_1) \\
\qquad \vdots \\
\qquad (\tau_n,\Theta_n,H_n,\rho_n) = \mathsf{inf}_{\mathsf{e}}^{(\rho\circ\rho_1\circ\cdots\circ\rho_{n-1})(\Theta)}(e_n) \\
\qquad (f_1,\ldots,f_k) = H^{\downarrow} \\
\qquad H_c = (\epsilon,[p_1^1 \leftarrow H_1^{\downarrow},\ldots,p_1^n \leftarrow H_n^{\downarrow}]) \\
\qquad\quad \mid\ \vdots \\
\qquad\quad \mid\ (\epsilon,[p_k^1 \leftarrow H_1^{\downarrow},\ldots,p_k^n \leftarrow H_n^{\downarrow}]) \\
\qquad \rho_f = \rho_n \circ \ldots \circ \rho_1 \circ \rho \\
\quad \textbf{in} \\
\qquad (\rho_f(\tau),\Theta_n,H;H_1;\ldots;H_n;Hc;(\epsilon,[it \leftarrow H_c^{\downarrow}]),\rho_f) \\
\quad \textbf{end} \\
\quad \textbf{where}\ (f_1 : p_1^1 : \tau_1,\ldots,p_1^n : \tau_n \stackrel{H_{f_1}}{\rightarrow} \tau) \in \Theta \\
\qquad\quad \vdots \\
\qquad\quad (f_k : p_k^1 : \tau_1,\ldots,p_k^n : \tau_n \stackrel{H_{f_k}}{\rightarrow} \tau) \in \Theta
\end{array}
$$

TABLE 4.14 – Algorithme d'inférence de types et d'effets (expressions (3/3)).

4.4.1 μ-*calcul* modal

Généralement, le comportement d'un programme informatique peut être représenté sous forme d'un graphe dont les noeuds représentent les états du programme et les arcs dénotent les transitions faisant évoluer le programme d'un état à un autre. Ce graphe correspond alors à une abstraction du programme mettant en avant tous les comportements déterminants de ce dernier. La vérification de la conformité d'un modèle M par rapport à une expression logique ψ est accomplie via des algorithmes de vérification par évaluation de modèles.

Syntaxe

La syntaxe de ce calcul est présentée dans la table 4.15.

$$\psi \quad ::= \quad \mathit{true} \mid X \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \texttt{<}a\texttt{>}\psi \mid \nu X.\psi$$

TABLE 4.15 – Syntaxe du μ-*calcul*.

Par définition, tout programme ou tout état d'un programme satisfait $true$. Aussi, un état satisfait $\psi \wedge \psi'$ s'il satisfait ψ et ψ' ; il satisfait $\neg\psi$ s'il ne satisfait pas ψ. Les formules variables sont représentées par la variable X et sont utilisées dans les formules de points fixes. La formule $<a>\psi$ représente l'opérateur modale de la logique indexé par une action a, et est connu comme l'opérateur de possibilité. Un état satisfait la formule $<a>\psi$ s'il admet au moins une transition immédiate indexée par l'action a qui le mène vers un état satisfaisant la formule ψ. La formule $\nu X.\psi$ permet de décrire des formules récursives et est connue comme l'opérateur du plus grand point fixe.

Notons que pour assurer l'existence du plus grand point fixe, les occurrences de la variable X dans ψ doivent l'être dans un contexte admettant un nombre pair de négations. Grâce à cette contrainte ainsi que l'utilisation de modèles finis, l'opérateur du plus grand point fixe est défini inductivement comme suit :

$$\nu X.\psi = \psi^0 \wedge \psi^1 \wedge \psi^2 \wedge \ldots \quad \text{avec} \quad \psi^0 = true \quad \text{et} \quad \psi^{i+1} = \psi[\psi^i/X]$$

L'expression $\psi[\psi'/X]$ représente la substitution simultanée de toutes les occurrences libres de X dans ψ par ψ'.

Notons que d'autres formules peuvent être définies comme suit :

$$\begin{array}{rcl} false & \equiv & \neg true \\ \psi_1 \vee \psi_2 & \equiv & \neg(\neg\psi_1 \wedge \neg\psi_2) \\ [a]\psi & \equiv & \neg<a>\neg\psi \\ \mu X.\psi & \equiv & \neg\nu X.\neg\psi[\neg X/X] \end{array}$$

Ainsi, à l'opérateur μ, opérateur du plus petit point fixe, lui correspond une définition inductive :

$$\mu X.\psi = \psi^0 \vee \psi^1 \vee \psi^2 \vee \ldots \quad \text{avec} \quad \psi^0 = false \quad \text{et} \quad \psi^{i+1} = \psi[\psi^i/X]$$

Exemples

Dans cette section, nous présentons différents exemples de propriétés exprimées à l'aide du *μ-calcul* modale. Nous débutons par des exemples simples qui expriment des comportements assez élémentaires. Notez que le sens de chaque propriété est relatif à un état courant du modèle.

Utilisation des opérateurs modaux

– <*c*>$true$
  Pour qu'un état satisfasse cette formule, il faut qu'il admette une transition étiquetée par l'action *c* menant vers un état satisfaisant la formule $true$. Puisque tous les états satisfont la formule $true$, on requiert donc que le modèle puisse effectuer l'action *c* à partir de l'état courant.
– [*r*]<*c*>$true$
  Cette formule peut être interprétée comme suit : à partir de l'état courant, toutes les transitions étiquetées par l'action *r* doivent aboutir à des états qui admettent une transition c. Autrement dit, toute action *r* doit être suivie par une action c.

Utilisation des opérateurs de point fixe Les formules présentées dans la section précédente se réfèrent toutes à un état courant. Cependant, pour pouvoir exprimer des propriétés de sécurité usuelles, il est nécessaire de disposer d'opérateurs permettant de spécifier des comportements concernant tout le modèle. Aussi, il serait intéressant de pouvoir définir des propriétés permettant de spécifier des comportements récurrents ou itératifs. C'est dans ce contexte que se situe l'utilisation des opérateurs de point fixe.

$\mu X.$<*b*>$true \vee$ <all>X Cette propriété est de la forme $\mu X.\psi$ dans laquelle la formule ψ correspond à <*b*>$true \vee$<all>X [3]. Pour saisir la signification de cette propriété, il faut calculer les différentes formules ψ^0, ψ^1, ψ^2, ... Par définition, ψ^0 vaut $false$; ψ^1 correspond à ψ, c'est-à-dire dans ce cas <*b*>$true \vee$ <all>X, dans laquelle toutes les occurrences de la variable X sont remplacées par ψ^0, c'est-à-dire $false$, ce qui donne <*b*>$true \vee$ <all>$false$. Le calcul des autres formules ψ^i se fait de la même manière que la formule précédente. Une fois ces formules calculées, il reste à prendre la disjonction de toutes ces formules pour obtenir la signification de la propriété initiale. Ainsi, nous avons :

$$\begin{aligned} \mu X.<b>true \vee <\text{all}>X = \; & false \vee \\ & <b>true \vee <\text{all}>false \vee \\ & <b>true \vee <\text{all}>(<b>true \vee <\text{all}>false) \vee \ldots \end{aligned}$$

Pour qu'un programme respecte cette propriété, il faut qu'il respecte l'une de ses sous-formules puisque, par définition, un programme respecte la formule $\psi_1 \vee \psi_2$ s'il respecte l'une des deux sous-formules ψ_1 et ψ_2. Pour qu'il respecte la formule <*b*>$true \vee$<all>$false$, il faut qu'il respecte soit la sous-formule <*b*>$true$, soit la sous-formule <all>$false$. Aucun programme ne peut satisfaire cette dernière puisqu'elle suggère l'existence d'une transition étiquetée par une action quelconque aboutissant à un état qui satisfait $false$. Or par définition, il n'existe pas d'état qui satisfait cette formule $false$. Pour qu'un programme

3. Pour plus de commodité, nous admettons que le terme «all» désigne toutes les actions possibles.

satisfasse la formule $<b>true$, il suffit qu'il admette au moins une transition immédiate étiquetée par l'action b. De la même manière, pour qu'un programme satisfait la formule $<b>true \vee <\mathsf{all}>(<b>true \vee <\mathsf{all}>false)$, il faut soit qu'il puisse effectuer immédiatement l'action b, soit qu'il effectue une action quelconque qui débouche sur un état à partir duquel il peut effectuer l'action b. En réitérant ce raisonnement, on conclut qu'un programme satisfait la propriété $\mu X.<b>true \vee <\mathsf{all}>X$ s'il admet au moins un état à partir duquel il peut effectuer l'action b.

Abréviations Malgré son expressivité, l'utilisation des opérateurs modaux et de point fixe peut paraître parfois assez complexe. La définition d'abréviations, ou de macros, permet de surmonter ce problème. À titre d'exemples, on présente ci-dessous un ensemble d'abréviations très pratiques qui permettent d'augmenter l'expressivité de la logique sans en augmenter la complexité :

- always$(\psi) = \nu X.\psi \wedge [\mathsf{all}]X$
- eventually$(\psi) = \mu X.\psi \vee <\mathsf{all}>X$
- never$(\psi) = \neg$eventually(ψ)

L'abréviation «always» permet de spécifier qu'une formule ψ passée en argument doit être satisfaite par tous les états futurs du programme. L'abréviation «eventually» permet de spécifier qu'une formule ψ sera inévitablement satisfaite par un état futur du programme. L'abréviation «never» permet de s'assurer qu'il n'existe aucun état futur satisfaisant une formule ψ donnée.

Il est possible d'abréger certains opérateurs existants pour offrir plus de commodité dans leur utilisation, comme par exemple les macros do et doNot.

- do(a) = $<a>true$
- doNot(a) = $[a]false$

En combinant les abréviations et les opérateurs modaux, il est possible d'exprimer des propriétés de sécurité plus complexes et plus usuelles :

- never(do(*f*))
- always([*r*] do(c))
- never(loop(*p*))

La première formule permet d'exprimer que le programme n'admet pas de transition *f*. Autrement dit, le programme ne peut exécuter l'une de ces actions. La deuxième formule permet de s'assurer qu'à chaque fois que l'on exécute une action *r* dans le programme, il est nécessaire par la suite d'exécuter l'action c. Cette propriété est vérifiée en tout point

du programme. La dernière formule permet de spécifier qu'il n'existe pas d'états à partir desquels l'action *p* est exécutée d'une manière infinie.

4.4.2 Extension du μ-*calcul* modal

Dans cette section, nous présentons une description formelle d'une logique offrant la possibilité de définir des propriétés faisant intervenir les données manipulées par un programme. L'idée est d'identifier l'information manipulée par le programme afin de décider s'il y a une violation de sécurité relative à une politique préalablement définie.

La section est organisée comme suit : Nous rappelons le modèle sur lequel s'effectue la vérification. Par la suite, nous présentons la syntaxe ainsi qu'une sémantique dénotationnelle de la logique. Nous illustrons par la suite l'expressivité de la logique à travers une série d'exemples.

Modèle

Au lieu de se limiter aux actions sensibles qu'effectue un programme, le modèle doit tenir compte des accès aux ressources privées, de l'envoi d'information sensible sur le réseau, etc. Comme mentionné dans la section précédente, l'idée est de spécifier, pour chaque action sensible, ses effets sur les ressources et sur l'information. Par exemple, l'action (l'API) *readFile* admet l'effet suivant : Accès au contenu de la ressource *file* et copie d'une partie de son contenu vers une zone mémoire.

De manière à tenir compte de ces différents types d'effets, nous proposons un modèle incluant des variables, des types ainsi qu'une relation de dépendances. Les variables sont utilisées pour pouvoir raisonner sur l'information manipulée à travers les actions sensibles. Les types sont assignés aux variables afin de spécifier les ressources qu'elles abstraient. Par exemple, l'environnement $[f \mapsto \mathit{file}^s]$ spécifie que la variable f abstrait une ressource file contenant de l'information secrète (s). La relation de dépendance est utilisée pour lier les différentes variables : Rappelons qu'une variable x_1 dépend d'une variable x_2 si le contenu de x_1 provient de celui de x_2.

Du modèle extrait à un système de transitions Le modèle généré par l'algorithme d'extraction de traces d'effets est composé d'un historique d'effets qui inclut, pour chaque action effectuée, un environnement de dépendances comprenant l'ensemble des dépendances actives à ce point de la trace. Le modèle inclut aussi un environnement global

τ	$::=$	$r^l \mid \tau(x) \mid \alpha \mid$ *void*	(Types)
r	$::=$	*file* $\mid$ *dir* $\mid$ *net* $\mid$ *host* $\mid$ *user* $\mid \ldots \mid \rho$	(Ressources)
l	$::=$	*secret* $\mid$ *public* $\mid$ *coded* $\mid$ *password* $\mid \ldots \mid \omega$	(Étiquettes)
a_i	$::=$	$\mathsf{copy}(x,y) \mid \mathsf{create}(x) \mid \mathsf{delete}(x)$	(Actions)

TABLE 4.16 – Syntaxe des actions, des types, des ressources et des étiquettes.

associant à toutes les variables manipulées un type. Nous supposons l'utilisation de l'α-conversion pour renommer, en variables fraiches, les différents paramètres d'une fonctions définies à l'aide de l'instruction «fun», ainsi que les différentes variables manipulées dans des blocs d'instructions (corps d'une fonction).

Aussi, nous supposons que seules les variables présentes dans les différentes environnements de dépendances sont définies dans l'environnement de typage Θ (les fonctions sont aussi éliminées de cette environnement). De plus, on suppose qu'en utilisant la transitivité entre les dépendances les variables *it* sont éliminées des différents environnements de dépendances.

À partir de ce modèle, il est possible de générer un système de transitions défini par un quadruplet $\langle \mathcal{S}, \mathcal{A}ct, \rightarrow, s_0 \rangle$, où $\mathcal{S}$ désigne l'ensemble des états, $\mathcal{A}ct$ est l'ensemble des actions $\{\mathsf{copy}(x,y), \mathsf{create}(x), \mathsf{delete}(x)\}$ (actions utilisées dans la trace d'effets), $\rightarrow \subseteq \mathcal{S} \times \mathcal{A}ct \times \mathcal{S}$ est une relation de transition et s_0 est l'état initial du modèle.

La table 4.16 rappelle la syntaxe des actions, des types, des ressources ainsi que des étiquettes associées aux types. Les types abstraient les ressources tandis que les étiquettes précisent le type d'information contenu dans chaque ressource.

À ce système de transitions, M, nous associons deux environnements, Γ et Θ. L'environnement Γ précise, pour chaque état s du modèle (système de transitions), l'ensemble des dépendances actives. L'environnement Θ assigne à chaque variable son type.

Par ailleurs, étant donné qu'un programme accède, en général, à des ressources particulières, un type singleton [**?**], est défini, $r^l(x)$, où la variable x représente une telle ressource. Par exemple, le type *file*(f) représente le fichier f, c'est-à-dire que toute expression ayant comme type *file*(f) doit avoir f comme valeur et doit donc correspondre au fichier f. De la même manière, le type *dir*(`sysdir`) représente le répertoire système (on suppose que `sysdir` est une constante prédéfinie).

L'action $\mathsf{copy}(x,y)$ représente un transfert d'information de x vers y. Par exemple, en considérant que x abstrait une zone mémoire et que y abstrait un port de communication, l'action $\mathsf{copy}(x,y)$ représente le transfert d'information de la mémoire vers le réseau.

Les actions create(x) et delete(x) représentent respectivement la création d'une nouvelle ressource et la suppression d'une ressource abstraite par x.

Logique

Syntaxe La nouvelle syntaxe n'affecte que les actions utilisées dans les modalités. Ainsi, et relativement à la syntaxe présentée à la table 4.15 (page 85), seule la définition de a_i est affectée. La table 4.17 présente la nouvelle définition.

ψ	$::=$	$\mathit{true} \mid \neg\psi \mid \psi \wedge \psi \mid X \mid \texttt{<}a\texttt{>}\psi \mid \nu X.\psi$
a	$::=$	$\mathit{read}(x,\tau) \mid \mathit{write}(x,\tau) \mid \mathit{create}(x,\tau) \mid \mathit{delete}(x,\tau)$

TABLE 4.17 – Syntaxe de la logique $\mathfrak{L}_\mu$.

L'ensemble des formules de cette logique sera désigné par $\mathfrak{L}_d$.

Le terme $\mathit{read}(x,\tau)$ représente l'action de lire une information d'une ressource de type τ. La variable x abstrait cette information. De la même manière, le terme $\mathit{write}(x,\tau)$ représente l'action d'écrire une information, abstraite par x, dans une ressource de type τ. Les termes $\mathit{create}(x,\tau)$ et $\mathit{delete}(x,\tau)$ représentent respectivement les actions de créer et de supprimer une ressource τ abstraite par la variable x.

La variable x permet de lier les informations manipulées par les différentes actions. C'est à travers ce type de variable que la nouvelle syntaxe nous offre la possibilité de définir des propriétés portant sur le flot de données d'un programme. Par ailleurs, notons que lorsqu'une variable x n'est pas requise dans la définition d'une propriété, nous utilisons le symbole « _ ». Par exemple, le terme $\mathit{read}(_,\mathit{file}^s)$ représente l'action de lire une information quelconque d'un fichier secret.

Sémantique dénotationnelle Dans cette section, nous proposons une sémantique dénotationnelle de la logique. Les formules modales sont interprétées dans un modèle de la forme $\mathcal{M} = (\underbrace{\langle \mathcal{S}, \mathcal{A}ct, \rightarrow, s_0\rangle}_{M}, \Gamma, \Theta)$, un environnement $e = [\mathcal{V} \rightarrow 2^{\mathcal{S}}]$ qui associe aux variables de $\mathcal{V}$ des ensembles d'états, et une substitution σ. L'ensemble $\mathcal{S}$ est un ensemble fini d'états, $\mathcal{A}ct$ un ensemble fini d'actions , $\rightarrow \subseteq \mathcal{S} \times \mathcal{A}ct \times \mathcal{S}$ une relation de transition, et $s_0 \in \mathcal{S}$ l'état initial du modèle. Sémantiquement, aux formules de cette logique correspondent les ensembles d'états pour lesquels elles sont vraies.

La fonction sémantique $[\![.]\!]_e^{\mathcal{M},\sigma} : \mathcal{F} \rightarrow 2^{\mathcal{S}}$ est proposée dans la table 4.18. L'ensemble $\mathcal{F}$ correspond à l'ensemble de toutes les formules de la logique $\mathfrak{L}_d$.

$$
\begin{aligned}
[\![true]\!]_e^{\mathcal{M},\sigma} &= \mathcal{S} \\
[\![X]\!]_e^{\mathcal{M},\sigma} &= e(X) \\
[\![\neg\psi]\!]_e^{\mathcal{M},\sigma} &= \overline{[\![\psi]\!]_e^{\mathcal{M},\sigma}} \\
[\![\psi_1 \wedge \psi_2]\!]_e^{\mathcal{M},\sigma} &= [\![\psi_1]\!]_e^{\mathcal{M},\sigma} \cap [\![\psi_2]\!]_e^{\mathcal{M},\sigma} \\
[\![<\!\mathit{read}(x,\tau)\!>\psi]\!]_e^{\mathcal{M},\sigma} &= \{s \mid \exists s', a, r \mid s \xrightarrow{\text{copy}(r,a)} s' \wedge s' \in [\![\psi]\!]_e^{\mathcal{M},\sigma\circ[b/x]_{[b\leftarrow^*(a)]}} \\
&\qquad \wedge (r \leftarrow^* (r')) \in \Gamma_s \\
&\qquad \wedge \Theta(r') \sqsubseteq \sigma(\tau) \\
&\qquad \wedge a \in \sigma(x)\} \\
[\![<\!\mathit{write}(x,\tau)\!>\psi]\!]_e^{\mathcal{M},\sigma} &= \{s \mid \exists s', a, r \mid s \xrightarrow{\text{copy}(a,r)} s' \wedge s' \in [\![\psi]\!]_e^{\mathcal{M},\sigma\circ[b/x]_{[a\leftarrow^*(b)]}} \\
&\qquad \wedge (r' \leftarrow^* (r)) \in \Gamma_s \\
&\qquad \wedge \Theta(r) \sqsubseteq \sigma(\tau) \\
&\qquad \wedge a \in \sigma(x)\} \\
[\![<\!\mathit{create}(x,\tau)\!>\psi]\!]_e^{\mathcal{M},\sigma} &= \{s \mid \exists s', a \mid s \xrightarrow{\text{create}(a)} s' \wedge s' \in [\![\psi]\!]_e^{\mathcal{M},\sigma\circ[a/x]_{[]}} \\
&\qquad \wedge \Theta(a) \sqsubseteq \sigma(\tau) \\
&\qquad \wedge a \in \sigma(x)\} \\
[\![<\!\mathit{delete}(x,\tau)\!>\psi]\!]_e^{\mathcal{M},\sigma} &= \{s \mid \exists s', a \mid s \xrightarrow{\text{delete}(a)} s' \wedge s' \in [\![\psi]\!]_e^{\mathcal{M},\sigma\circ[a/x]_{[]}} \\
&\qquad \wedge \Theta(a) \sqsubseteq \sigma(\tau) \\
&\qquad \wedge a \in \sigma(x)\} \\
[\![\nu X.\psi]\!]_e^{\mathcal{M},\sigma} &= \bigcup\{Q \subseteq \mathcal{S} \mid Q \subseteq [\![\psi]\!]_{e[X\mapsto Q]}^{\mathcal{M},\sigma}\}
\end{aligned}
$$

TABLE 4.18 – Sémantique de la logique $\mathfrak{L}_d$.

Cette sémantique utilise des substitutions de la forme $[t/x]_{\mathcal{C}}$ permettant de substituer une variable x par un terme t qui respecte la contrainte $\mathcal{C}$. Par exemple, $[b/x]_{[b \leftarrow^* (a)]}$ substitue la variable x par un terme b qui dépend directement ou indirectement d'un terme a donné. Par conséquent, appliquée à une variable, une substitution retourne un ensemble de termes en résultat. Plus formellement :

$$\begin{array}{rcl} (\sigma \circ [t/x]_{\mathcal{C}})(x) & = & \{t \mid t \in \mathcal{C}\} \\ \sigma(y) & = & y \quad \text{si } x \notin \mathit{dom}(\sigma) \\ ([t_1/x_1]_{\mathcal{C}_1} \circ \ldots \circ [t_n/x_n]_{\mathcal{C}_n})(_) & = & \bigcup_i^n \{t_i \mid t_i \in \mathcal{C}_i\} \end{array}$$

Aussi, appliquée à un type, nous avons :

$$\begin{array}{rcl} \sigma(\tau(x)) & = & \tau(\sigma(x)) \\ \sigma(\tau) & = & \tau \end{array}$$

La composition de deux substitutions est définie comme suit :

$$[t/x]_{[a \leftarrow^* (b)]} \circ [t'/x']_{[a' \leftarrow^* (b')]} = [t/x, [t/x](t')/x']_{[a \leftarrow^* (b), [t/x](a') \leftarrow^* ([t/x](b'))]}$$

Notons que la sémantique dénotationnelle suppose la définition d'une relation $\sqsubseteq$ définie entre les types et permettant de les ordonner. Cette fonction étend celle définie entre les étiquettes à la section 4.3.4.

Exemples Afin d'illustrer l'expressivité de cette logique, nous présentons dans cette section certains exemples de propriétés. Nous débutons par des propriétés de sécurité usuelles.

1. never(<*read*(x, ρ_1^s)> eventually(<*write*(x, ρ_2^p)>*true*))
   Cette propriété stipule qu'aucune information privée ne peut être transférée dans une zone publique.
2. never(<*read*(x, ρ^s)> eventually(<*write*(x, net^p)>*true*))
   Cette propriété stipule qu'aucune information privée ne peut être envoyée sur un réseau public.
3. never(<*read*$(_, \mathit{file}^s)$>*true*)
   Cette propriété stipule qu'aucune information privée ne peut être lue d'un fichier quelconque.
4. never(<*read*$(_, \mathit{file}^\alpha)$>*true*)
   Cette propriété est plus restrictive que la précédente puisqu'elle interdit tout accès aux fichiers.
5. never(loop(<*create*$(_, \mathit{process}^\alpha)$>*true*))
   Cette propriété s'assure qu'il n'y a pas de création infinie de processus.

Les exemples suivants illustrent l'expression de propriétés de sécurité plus complexes :

1. $\text{never}(\,<\mathit{read}(f, dir^{\alpha}(\texttt{sysdir}))>\text{eventually}(\,<\mathit{read}(x, \mathit{file}^{\alpha}(f))>\text{eventually}(\,<\mathit{write}(x, net^{p})>\mathit{true}\,)\,)\,)$

   Cette propriété stipule qu'aucune information provenant d'un fichier du répertoire système ne peut être envoyée sur un réseau public.
2. $\text{always}(\,[\mathit{write}(f, dir^{\alpha}(\texttt{tempdir}))]\,\text{eventually}(\,<\mathit{delete}(_, \mathit{file}^{\alpha}(f))>\mathit{true}\,)\,)$

   Cette propriété stipule que tout fichier créé dans le répertoire temporaire doit inévitablement être supprimé.

Notons que dans ces deux exemples, la variable f est utilisée à la fois pour abstraire une information et pour abstraire une ressource du système. Ceci permet de définir des propriétés assez complexes.

En conclusion, l'extension tant au niveau du modèle qu'au niveau de la logique, offre à l'utilisateur un langage plus puissant pour spécifier ses propriétés de sécurité et des outils de vérification plus efficaces puisque basés sur des modèles plus précis.

4.5 Conclusion

Dans ce chapitre, nous avons présenté une approche de vérification de propriétés de sécurité d'un programme informatique. La particularité de cette approche réside dans le fait que le mécanisme de vérification combine l'analyse du flot de contrôle et celle du flot de données pour déterminer la cohérence d'un programme par rapport à une propriété de sécurité.

Conclusion Générale

L'objectif de ce travail était de développer un mécanisme permettant de garantir la sécurité des systèmes, en s'appuyant sur l'expérience établie dans le domaine de la sécurité et des méthodes formelles. Notre choix s'est porté sur la définition d'un mécanisme d'analyse du flot de données et du flot contrôle par rapport à un ensemble de propriétés de sécurité décrites sous formes de formules logiques. En étudiant les différentes approches similaires dans la littérature, on remarque que ce type d'analyse permet de modéliser le comportement du programme pour en déduire certaines informations, sans toutefois l'exécuter. L'analyse du flot de contrôle permet de suivre la trace d'exécution du programme, par exemple, en générant un modèle permettant de visualiser les différents chemins possibles d'exécution d'un programme. L'analyse du flot de données permet d'identifier les ressources influant sur la valeur d'une autre variable donnée et qui affecteront l'évaluation d'une instruction.

Vérification de propriétés

Lors de la première partie de ce mémoire, nous avons présenté trois approches de vérification de propriétés de sécurité de programmes informatiques. Ces approches définissent les orientations que nous avons choisies pour construire notre modèle de vérification des propriétés de sécurité.

La première approche, proposée par Atsushi Igarashi dans [2, 3], permet de s'assurer de la validité d'un programme à travers les différents usages des ressources décrites dans le code de ce dernier. L'approche proposée introduit une abstraction pertinente des différentes instructions du programme procédant à l'exploitation d'une ressource donnée. Dans ce cas, la validité d'un programme est liée à la cohérence des usages des ressources introduites dans ce dernier. Toutefois, cette approche souffre de plusieurs lacunes : Exploitation simultanée de ressources, imprécision lors de représentation des structures de données récursives, etc.

La deuxième approche présentée dans ce mémoire introduit le concept d'*historique d'effets* et met en avant son utilité lors de l'abstraction du flot de contrôle. L'auteur présente des exemples d'application de cette approche pour la vérification de plusieurs propriétés de sécurité. L'intérêt de cette approche réside dans sa simplicité ; toutefois, elle ne prend en compte que les appels de méthodes du programme et n'offre donc qu'un modèle abstrait le flot de contrôle de ce dernier [36].

Pour finir, la derniére approche présentée permet de vérifier la validité d'un programme à partir de la trace d'effets qui lui est associée [4]. Cette approche définit un ensemble d'outils permettant de construire une trace d'effets démonstrative du flot de contrôle d'un programme au moment de son exécution. La programme est alors analysé par un ensemble de mécanismes permettant de vérifier la validité de ce dernier par rapport à des propriétés de sécurité. L'objectif de cette approche est de définir un système pouvant modéliser et de vérifier statiquement les politiques d'accès relative à la pile d'inspection JAVA. Toutefois, cette approche ne définit aucun outil permettant de s'assurer de la validité des dépendances de données alors que ces dernières constituent une partie importante de l'architecture générale d'un programme et de son analyse.

Contribution

La deuxième partie de ce mémoire introduit notre contribution, une nouvelle approche de vérification de propriétés à travers une analyse combinée du flot de données et du flot de contrôle d'un programme. L'approche proposée reprend les mécanismes définis dans le chapitre 4 (page 61) en ce qui concerne la validation du flot de contrôle. La validité du flot de données est assurée par des processus d'analyse qui étendent les concepts de dépendances de données [21, 37]. Le modèle obtenu va prendre en compte tous les axes représentatifs de l'architecture générale d'un programme. La vérification d'un programme est alors réalisée de manière statique au moment de la compilation du code.

Le premier apport de cette approche réside dans la définition d'un algorithme d'extraction d'historiques d'effets et des dépendances directes à partir d'un programme. Le deuxième apport propose un cadre pour décrire des propriétés de sécurité sous forme de formules d'une logique temporelle qui étend le μ-*calcul* modal.

Perspectives d'avenir

Le présent mémoire établit les bases d'une nouvelle approche de vérification de propriétés de sécurité. Ce projet de recherche offre l'opportunité d'être concrétisé grâce à un effort de développement et ce afin de profiter d'abord de la solidité de la synthèse élaborée et ensuite de permettre l'extension de ses capacités. Nous allons donc présenter différentes extensions qui nous semble importantes pour ce travail :

- Même si ce travail est construit sur des bases mathématiques, et reprend des concepts introduits dans différentes approches, il serait utile d'implanter une application permettant de mettre en évidence tous les aspects de notre approche. Cette implantation peut constituer un travail pratique à part entière, qui comporte plusieurs modules : la génération automatique du modèle, la preuve de conformité du modèle au programme, l'outil interface qui propose l'écriture de politiques de sécurité, le vérificateur de la conformité de la politique au modèle, etc.
- L'algorithme d'inférence de types et d'effets peut être étendu afin prendre en compte la notion d'*effet variable*. En effet, il est parfois difficile, voire impossible, d'identifier l'effet engendré par une instruction du programme. Pour cet effet, nous pouvons étendre le concept d'effet variable introduit dans [6].
- Le modèle de données de notre approche peut être étendu de manière à inclure plus d'informations relatives aux ressources d'un programmes. Pour se faire, nous pouvons nous inspirer des travaux introduits dans [10].
- Il serait très utile d'adapter l'ensemble des mécanismes de vérification développées dans ce travail aux langages courants (JAVA, $C^{\#}$, DELPHI, etc). Pour se faire, nous pouvons reprendre les concepts introduits dans [5, 32].

Au terme de ce travail, il semble évident que la vérification de propriétés de sécurité reste un domaine de recherche qui nécessite encore une attention particulière. La résolution de ce problème exige inévitablement une synergie des diverses disciplines liées à l'informatique et à la programmation. Nous espérons que ce mémoire contribuera un tant soit peu à la résolution de la détection du code malicieux.

Bibliographie

[1] A. Igarashi, B. C. Pierce, P. W., « Featherweight java : a minimal core calculus for java and gj », *ACM Transactions on Programming Languages and Systems*, 23, Mai 2001.

[2] A. Igarashi, N. K., « Ressource usage analysis », *ACM ISBN*, Février 2002.

[3] A. Igarashi, N. K., « Resource usage analysis », *ACM Transactions on Programming Languages and Systems*, 27, Mars 2005.

[4] C. Skalka, S. S., « History effects and verification », *Asian Programming Languages Symposium*, Novembre 2004.

[5] C. Skalka, S. Smith, D. V. H., « A type and effect system for flexible abstract interpretation of java », *ACM Workshop on Abstract Interpretation of Object Oriented Languages*, Janvier 2005.

[6] C. Skalka, S. Smith, D. V. H., « Types and trace effects of higher order programs », *Journal of Functional Programming*, 2006.

[7] Cousot, P., « Interprétation abstraite », *Technique et science informatiques*, 19, 2000.

[8] Cousot, P., « Verification by abstract interpretation », *International Symposium on Verification*, 2772, 2003.

[9] D. K. Gifford, J. M. L., « Integrating functional and imperative programming », *Proceedings of the 1986 ACM conference on LISP and functional programming*, p. 28–38, 1986.

[10] D. Schmidt, B. S., « Program analysis as model checking of abstract interpretations », *Proceedings of the 5th International Symposium on Static Analysis*, p. 351 – 380, 1998.

[11] D. Volpano, G. Smith, C. I., « A sound type system for secure flow analysis », *Journal of Computer Security*, Juillet 1996.

[12] E. A. Emerson, C. L. L., « Efficient model checking in fragments of the propositional mu-calculs », *Proceedings of the First Annual Symposium on Logic in Computer Science*, 1986.

[13] F. Nielson, H. R. N., « Type and effect systems », *Lecture Notes In Computer Science*, 1710, p. 114 – 136, 1999.

[14] G. J. Holzmann, M. H. S., « Software model checking : Extracting verification models from source code », *FORTE/PSTV Conference*, Octobre 1999.

[15] G. Naumovich, L. A. Clarke, J. M. C., « Using partial order techniques to improve performance of data flow analysis based verification », *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 24, Septembre 1999.

[16] H. Chen, D. W., « Mops : an infrastructure for examining security properties of software », *ACM*, Novembre 2002.

[17] Horn, D. V., *Algorithmic trace effect analysis*, mémoire de maîtrise, University of Vermont, Vermont, USA, Mai 2006.

[18] J. Palsberg, M. I. S., *Object-oriented type systems*, 1994.

[19] K. Marriott, P. J. Stuckey, M. S., « Resource usage verification », *Programming languages and systems*, 2895, p. 212–229, Novembre 2003.

[20] Kozen, D., « Results on the propositional mu-calculs », *Theorical Computer Science*, 27, p. 333–354, 1983.

[21] Ktari, B., *Certification de composantes logicielles*, thèse de doctorat, Université Laval, Québec, CANADA, 2003.

[22] London, R. L., « A view of program verification », *International conference on Reliable software*, 10, Avril 1975.

[23] M. B. Dwyer, L. A. Clarke, J. M. C. G. N., « Flow analysis for verifying properties of concurrent software systems », *ACM Transactions on Software Engineering and Methodology*, 13, Octobre 2004.

[24] M. Müller-Olm, D. Schimdt, B. S., « Model-checking : A tutorial introduction », *Proceedings of the 6th International Symposium on Static Analysis*, 1694, p. 330 – 354, 1999.

[25] Merz, S., « Model checking : a tutorial overview », *Modeling and verification of parallel processes*, p. 3–38, 2001.

[26] N. Rinetzky, J. Bauer, T. R. M. S. R. W., « A semantics for procedure local heaps and its abstractions », *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 40, Janvier 2005.

[27] O. Burkart, B. S., « Model checking the full modal mu-calculus for infinite sequential processes », *Theoretical Computer Science*, 221, p. 251–270, 1999.

[28] Pichardie, D., *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs java certifiéés*, thèse de doctorat, Université de Rennes 1, 2005.

[29] S. Horwitz, J. Prins, T. R., « On the adequacy of program dependence graphs for representing programs », *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Janvier 1988.

[30] S.Berezin, E. Clarke, S. J. W. M., « Model checking algorithms for the mu-calculus », *Proof, language, and interaction : essays in honour of Robin Milner*, p. 309 – 337, 2000.

[31] Schmidt, D. A., « Data flow analysis is model checking of abstract interpretations », *ACM : Principales of Programming Languages*, 1998.

[32] Skalka, C., « Trace effects and object orientation », *ACM Conference on Principles and Practice of Declarative Programming*, 2005.

[33] Sudkamp, T., *Languages and machines*, Addison-Wesley, 1988.

[34] T. Jensen, D. Le Métayer, T. T., *Verification of control flow based security proprieties*, rapport technique 1210, IRISA, Rennes, FRANCE, Octobre 1998.

[35] Talpin, J.-P., *Aspects théoriques et pratiques de l'inférence de type et d'effets*, thèse de doctorat, Université Paris VI, Paris, FRANCE, Mai 1993.

[36] Thorn, T., *Vérification de politiques de sécurité par analyse de programmes*, thèse de doctorat, Université de Rennes 1, Rennes, FRANCE, 1999.

[37] Verbyst, D., *Code incorporant un modèle certifié*, mémoire de maîtrise, Université Laval, Québec, CANADA, 2005.

[38] W. Pugh, D. W., *An exact method for analysis of value-based data dependances*, rapport technique 3196, Université du Maryland, Maryland, USA, Décembre 1993.

[39] Y. Mei Tang, P. J., « Effect systems with subtyping », *ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 1995.