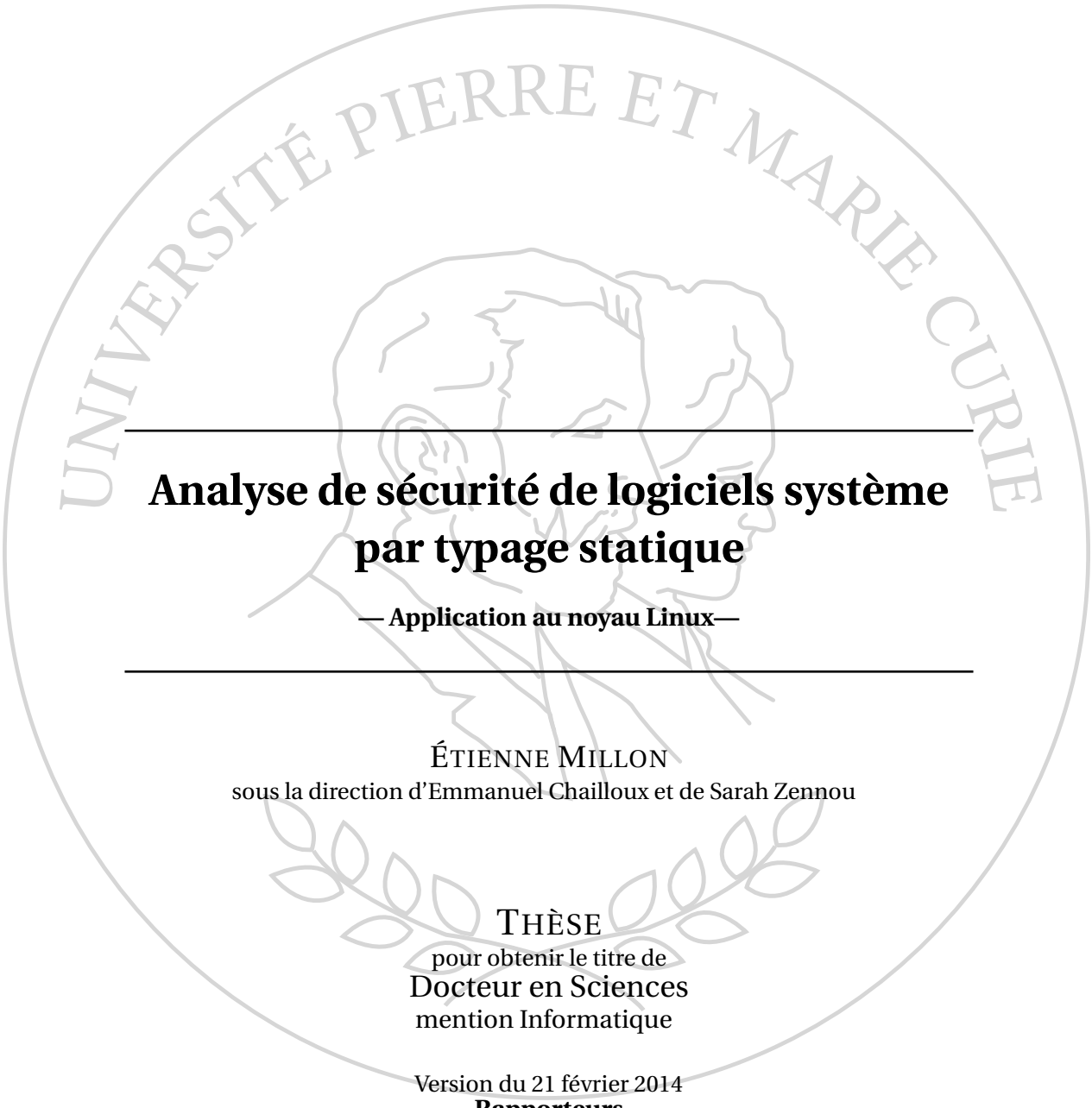
UNIVERSITÉ PIERRE ET MARIE CURIE
ÉCOLE DOCTORALE INFORMATIQUE, TÉLÉCOMMUNICATIONS ET ÉLECTRONIQUE



Analyse de sécurité de logiciels système par typage statique

— Application au noyau Linux—

ÉTIENNE MILLON
sous la direction d'Emmanuel Chailloux et de Sarah Zennou

THÈSE
pour obtenir le titre de
Docteur en Sciences
mention Informatique

Version du 21 février 2014

Rapporteurs
Sandrine Blazy IRISA
Pierre Jouvelot Mines ParisTech
Directeurs
Emmanuel Chailloux Université Pierre et Marie Curie
Sarah Zennou EADS Innovation Works

> “Many C programmers believe that « strong typing » just means pounding extra hard on the keyboard.”
>
> *Peter van den Linden*

REMERCIEMENTS

TABLE DES MATIÈRES

CHAPITRE 1

INTRODUCTION

Communication, audiovisuel, transports, médecine : tous ces domaines se sont transformés dans les dernières décennies, en particulier grâce à la révolution numérique. En effet le plus petit appareil électrique contient maintenant des composants matériels programmables.

En 2014, on pense bien sûr aux téléphones portables dont la fonctionnalité et la complexité les rapprochent des ordinateurs de bureau. Par exemple, le système d'exploitation Android de Google est fondé sur le noyau Linux, destiné à la base aux micro-ordinateurs.

Le noyau d'un système d'exploitation est chargé de faire l'intermédiaire entre le matériel (processeur, mémoire, périphériques, ...) et les applications exécutées sur celui-ci (par exemple un navigateur web, une calculatrice ou un carnet d'adresses).

Il doit aussi garantir la sécurité de celles-ci : en tant qu'intermédiaire de confiance, le noyau a un certain nombre de responsabilités et est le seul à avoir accès à certaines informations sensibles. Il est capital de s'assurer qu'il est bien le seul à pouvoir y accéder. En particulier, il faut pouvoir vérifier que les requêtes faites par l'utilisateur au noyau ne peuvent pas, volontairement ou involontairement, détourner ce dernier et lui faire fuiter des informations confidentielles.

Le problème est que, comme tous les logiciels, les noyaux de système d'exploitation sont écrits par des humains qui ne sont pas parfaits. Les activités de relecture et de débogage ont beau prendre la majeure partie du temps de développement, il est facile de laisser passer des défauts de programmation.

Ces erreurs, ou *bugs*, peuvent avoir des conséquences dramatiques sur le plan matériel ou humain. À titre d'exemple, un Airbus A320 embarque près de 10 millions de lignes de code : il est capital de vérifier que celles-ci ne peuvent pas mettre en danger la sûreté des passagers.

Une technique efficace est de réaliser des tests, c'est-à-dire exécuter le programme sous un environnement contrôlé. On peut alors détecter des comportements non désirés. Mais même avec une grande quantité de tests il n'est pas possible de couvrir tous les cas d'utilisation.

Une autre approche est d'analyser le code source du programme avant de l'exécuter et de refuser de lancer les programmes qui contiennent certaines constructions dangereuses. C'est l'analyse statique de programmes.

Une des techniques d'analyse statique les plus répandues et les plus simples est le typage statique, qui consiste à associer, à chaque morceau de programme, une étiquette décrivant quel genre de valeur sera produite par son évaluation. Par exemple, si n est le nom d'une variable entière, alors $n+2$ produira toujours une valeur entière. Cela permet de savoir si les programmes manipuleront des données incompatibles entre elles.

Pour en revenir aux noyaux de système d'exploitation, ceux-ci manipulent à la fois des données sensibles et des données provenant du monde extérieur, pour lesquelles on n'a aucune garantie. On veut pouvoir distinguer ces deux classes de données.

Plus précisément, un des points cruciaux pour garantir l'isolation d'un noyau de système d'exploitation est de restreindre la manière dont sont traitées les informations provenant des programmes utilisateur.

Le but de cette thèse est de montrer que le typage statique peut être utilisé pour détecter et interdire ces manipulations dangereuses.

1.1 Rôle d'un système d'exploitation

Un ordinateur est constitué de nombreux composants matériels : microprocesseur, mémoire, et divers périphériques. Et au niveau de l'utilisateur, des dizaines de logiciels permettent d'effectuer toutes sortes de calculs et de communications. Le système d'exploitation permet de faire l'interface entre ces deux échelles.

Au cours de l'histoire des systèmes informatiques, la manière de les programmer a beaucoup évolué. Au départ, les programmeurs avaient accès au matériel dans son intégralité : toute la mémoire pouvait être accédée, toutes les instructions pouvaient être utilisées.

Néanmoins c'est un peu restrictif, puisque cela ne permet qu'à une personne d'interagir avec le système. Dans la seconde moitié des années 1960, sont apparus les premiers systèmes « à temps partagé », permettant à plusieurs utilisateurs de travailler en même temps.

Permettre l'exécution de plusieurs programmes en même temps est une idée révolutionnaire, mais elle n'est pas sans difficultés techniques : en effet les ressources de la machine doivent être aussi partagées entre les utilisateurs et les programmes. Par exemple, plusieurs programmes vont utiliser le processeur les uns à la suite des autres ; et chaque programme aura à sa disposition une partie de la mémoire principale, ou du disque dur.

Si plusieurs programmes s'exécutent de manière concurrente sur le même matériel, il faut s'assurer que l'un ne puisse pas écrire dans la mémoire de l'autre, et aussi que les deux n'utilisent pas la carte réseau en même temps. Ce sont des rôles du système d'exploitation.

Ainsi, au lieu d'accéder directement au matériel via des instructions de bas niveau, les programmes communiquent avec le noyau, qui centralise donc les appels au matériel, et abstrait certaines opérations.

Par exemple, comparons ce qui se passe concrètement lors de la copie de données depuis un cédérom ou une clef USB.

- Dans le cas du cédérom, il faut interroger le bus SATA, interroger le lecteur sur la présence d'un disque dans le lecteur, activer le moteur, calculer le numéro de trame des données sur le disque, demander la lecture, puis déclencher une copie de la mémoire.
- Avec une clef, il faut interroger le bus USB, rechercher le bon numéro de périphérique, le bon numéro de canal dans celui-ci, lui appliquer une commande de lecture au bon numéro de bloc, puis copier la mémoire.

Ces deux opérations, bien qu'elles aient la même intention (copier de la mémoire depuis un périphérique amovible), ne sont pas effectuées en extension de la même manière. C'est pourquoi le système d'exploitation fournit les notions de fichier, lecteur, etc : le programmeur n'a plus qu'à utiliser des commandes de haut niveau (« monter un lecteur », « ouvrir un fichier », « lire dans un fichier ») et, selon le type de lecteur, le système d'exploitation effectuera les actions appropriées.

En résumé, un système d'exploitation est l'intermédiaire entre le logiciel et le matériel, et en particulier est responsable de la gestion de la mémoire, des périphériques et des pro-

cessus. Les détails d'implantation ne sont pas présentés à l'utilisateur ; à la place, il manipule des abstractions, comme la notion de fichier. Pour une explication détaillée du concept de système d'exploitation ainsi que des cas d'étude, on pourra se référer à [Tan07].

1.2 Séparation entre espace noyau et espace utilisateur

Puisque le noyau est garant d'une utilisation sûre du matériel, il ne doit pas pouvoir être manipulé directement par l'utilisateur ou les programmes exécutés. Ainsi, il est nécessaire de mettre en place des protections entre les espaces noyau et utilisateur.

Au niveau matériel, on utilise la notion de *niveaux de privilèges* pour déterminer s'il est possible d'exécuter une instruction.

D'une part, le processeur contient un niveau de privilège intrinsèque. D'autre part, chaque zone mémoire contenant du code ou des données possède également un niveau de privilège minimum nécessaire. L'exécution d'une instruction est alors possible si et seulement si le niveau de privilège du processeur est supérieur à celui de l'instruction et des opérandes mémoires qui y sont présentes [1].

Par exemple, supposons qu'un programme utilisateur contienne l'instruction « déplacer le contenu du registre EAX vers l'adresse mémoire a », où a fait partie de l'espace mémoire de l'utilisateur. Alors aucune erreur de protection mémoire n'est déclenchée.

Ainsi, pour une instruction manipulant des données en mémoire, les accès possibles sont décrits dans le tableau suivant. En cas d'impossibilité, une erreur se produit et l'exécution s'arrête. Par exemple, l'avant-dernière ligne indique que, si un programme tente de lire une variable du noyau, celui-ci sera arrêté par une exception.

Mode du processeur	Privilège (code)	Privilège (données)	Accès possible
Noyau	Noyau	Noyau	Oui
Noyau	Noyau	Utilisateur	Oui
Noyau	Utilisateur	Noyau	Oui
Noyau	Utilisateur	Utilisateur	Oui
Utilisateur	Noyau	Noyau	Non
Utilisateur	Noyau	Utilisateur	Non
Utilisateur	Utilisateur	Noyau	Non
Utilisateur	Utilisateur	Utilisateur	Oui

En plus de cette vérification, certains types d'instructions sont explicitement réservés au mode le plus privilégié : par exemple les lectures ou écritures sur des ports matériels, ou celles qui permettent de définir les niveaux de privilèges des différentes zones mémoire.

Comme les programmes utilisateur ne peuvent pas accéder à ces instructions de bas niveau, ils sont très limités dans ce qu'ils peuvent faire. En utilisant seulement les seules instructions non privilégiées, on peut uniquement réaliser des calculs, sans réaliser d'opérations visibles depuis l'extérieur du programme.

Pour utiliser le matériel ou accéder à des abstractions de haut niveau (comme créer un nouveau processus), ils doivent donc passer par l'intermédiaire du noyau. La communication entre le noyau et les programmes utilisateur est constituée du mécanisme des *appels système*.

1. Ici « supérieur » est synonyme de « plus privilégié ». Dans l'implantation d'Intel présentée dans le chapitre 2, les niveaux sont numérotés de 0 à 3, où le niveau 0 est le plus privilégié.

Lors d'un appel système, une fonction du noyau est invoquée (en mode noyau) avec des paramètres provenant de l'utilisateur. Il faut donc être particulièrement précautionneux dans le traitement de ces données.

Par exemple, considérons un appel système de lecture depuis un disque : on passe au noyau les arguments (d, o, n, a) où d est le nom du disque, o (pour *offset*) l'adresse sur le disque où commencer la lecture, n le nombre d'octets à lire et a l'adresse en mémoire où commencer à stocker les résultats.

Dans le cas d'utilisation prévu, le noyau va copier la mémoire lue dans a. Le processeur est en mode noyau, en train d'exécuter une instruction du noyau manipulant des données utilisateur. D'après le tableau de la page 3, aucune erreur ne se produit.

Mais même si ce cas ne produit pas d'erreur à l'exécution, il est tout de même codé de manière incorrecte. En effet, si on passe à l'appel système une adresse a faisant partie de l'espace noyau, que se passe-t-il ?

L'exécution est presque identique : au moment de la copie on est en mode noyau, en train d'exécuter une instruction du noyau manipulant des données noyau. Encore une fois il n'y a pas d'erreur à l'exécution.

On peut donc écrire n'importe où en mémoire. De même, une fonction d'écriture sur un disque (et lisant en mémoire) permettrait de lire de la mémoire du noyau. À partir de ces primitives, on peut accéder aux autres processus exécutés, ou détourner l'exécution vers du code arbitraire. L'isolation est totalement brisée à cause de ces appels système.

La cause de ceci est qu'on a accédé à la mémoire en testant les privilèges du noyau au lieu de tester les privilèges de celui qui a fait la requête (l'utilisateur). Ce problème est connu sous le nom de *confused deputy problem* [Har88].

Pour implanter un appel système, il est donc nécessaire d'interdire le déréférencement direct des pointeurs dont la valeur peut être contrôlée par l'utilisateur. Dans le cas du passage par adresse d'un argument, il aurait fallu vérifier à l'exécution que celui-ci a bien les mêmes privilèges que l'appelant.

Il est facile d'oublier d'ajouter cette vérification, puisque le cas « normal » fonctionne. Avec ce genre d'exemple on voit comment les bugs peuvent arriver si fréquemment et pourquoi il est aussi capital de les détecter avant l'exécution.

1.3 Systèmes de types

La plupart des langages de programmation incorporent la notion de type, dont un des buts est d'empêcher de manipuler des données incompatibles entre elles.

En mémoire, les seules données qu'un ordinateur manipule sont des nombres. Selon les opérations effectuées, ils seront interprétés comme des entiers, des adresses mémoire ou des caractères. Pourtant il est clair que certaines opérations n'ont pas de sens : par exemple, multiplier un nombre par une adresse ou déréférencer le résultat d'une division sont des comportements qu'on voudrait pouvoir empêcher.

En un mot, le but du typage est de classifier les objets et de restreindre les opérations possibles selon la classe d'un objet : en somme, « ne pas ajouter des pommes et des oranges ». Le modèle qui permet cette classification est appelé *système de types* et est en général constitué d'un ensemble de *règles de typage*, comme « un entier plus un entier égale un entier ».

Typage dynamique Dans ce cas, chaque valeur manipulée par le programme est décorée d'une étiquette définissant comment interpréter la valeur en question. Les règles de typage sont alors réalisées à l'exécution. Par exemple, l'opérateur « + » vérifie que ses deux opérandes ont une étiquette « entier », et construit alors une valeur obtenue en faisant l'addition des

deux valeurs, avec une étiquette « entier ». C'est ce qui se passe par exemple dans le langage Python [☞4].

Typage statique Dans ce cas on fait les vérifications à la compilation. Pour vérifier ceci, on donne à chaque fonction un contrat comme « si deux entiers sont passés, et que la fonction renvoie une valeur, alors cette valeur sera un entier ». Cet ensemble de contrats peut être vérifié statiquement par le compilateur, à l'aide d'un système de types statique.

Par exemple, on peut dire que l'opérateur « + » a pour type $(\text{INT}, \text{INT}) \to \text{INT}$. Cela veut dire que, si on lui passe deux entiers (INT, INT), alors la valeur obtenue est également un entier. *A contrario*, si autre chose qu'un entier est passé à cet opérateur, le programme ne compile pas.

Typage fort ou faible Indépendamment du moment où est faite cette analyse, on peut avoir plus ou moins de garanties sur les programmes sans erreurs de typage. En poussant à l'extrême, les systèmes de types forts garantissent que les valeurs ont toujours le type attendu. Avec du typage statique, cela permet d'éliminer totalement les tests de typage à l'exécution. Mais souvent ce n'est pas le cas, car il peut y avoir des constructions au sein du langage qui permettent de contourner le système de types, comme un opérateur de transtypage. On parle alors de typage faible.

Polymorphisme Parfois, il est trop restrictif de donner un unique type à une fonction. Si on considère une fonction ajoutant un élément à une liste, ou une autre triant un tableau en place, leur type doit-il faire intervenir le type des éléments manipulés ?

En première approximation, on peut imaginer fournir une version du code par type de données à manipuler. C'est la solution retenue par le langage C, ou par les premières versions du langage Pascal, ce qui rendait très difficile l'écriture de bibliothèques [Ker81]. On parle alors de monomorphisme.

Une autre manière de procéder est d'autoriser plusieurs fonctions à avoir le même nom, mais avec des types d'arguments différents. Par exemple, on peut définir séparément l'addition entre deux entiers, entre deux flottants, ou entre un entier et un flottant. Selon les informations connues à la compilation, la bonne version sera choisie. C'est ainsi que fonctionnent les opérateurs en C++. On parle de polymorphisme *ad hoc*, ou de surcharge.

Une autre technique est de déterminer la fonction appelée non pas par le type de ses arguments, mais par l'objet sur lequel on l'appelle. Cela permet d'associer le comportement aux données. On parle alors de polymorphisme objet. Dans ce cas, celui-ci repose sur le sous-typage : si A_1 et A_2 sont des sous-types de B, on peut utiliser des valeurs de type A_1 ou A_2 là où une valeur de type B est attendue. Dans ce cas, la fonction correspondante sera appelée.

La dernière possibilité est le polymorphisme paramétrique, qui consiste à utiliser le même code quel que soit le type des arguments. Dans ce cas, on utilise une seule fonction pour traiter une liste d'entiers ou une liste de flottants, par exemple. Au lieu d'associer à chaque fonction un type, dans certains cas on lui associe un type paramétré, instanciable en un type concret. Dans le cas des fonctions de traitement de liste, l'idée est que lorsqu'on ne touche pas aux éléments, alors le traitement est valable quel que soit leur type. Cette technique a été décrite en premier dans [Mil78].

Pour un tour d'horizon de différents systèmes de types statiques, avec en particulier du polymorphisme, on pourra se référer à [Pie02].

1.4 Langages

Le système Unix, développé à partir de 1969, a tout d'abord été développé en assembleur sur un mini-ordinateur PDP-7, puis a été porté sur d'autres architectures matérielles. Pour aider ce portage, il a été nécessaire de créer un « assembleur portable », le langage C [KR88, ISO99]. Son but est de fournir des abstractions au dessus du langage d'assemblage. Les structures de contrôle (`if`, `while`, `for`) permettent d'utiliser la programmation structurée, c'est-à-dire en limitant l'utilisation de l'instruction `goto`. Les types de données sont également abstraits de la machine : ainsi, `int` désigne un entier machine, indépendamment de sa taille concrète. Son système de types, bien que statique (il peut y avoir des erreurs de typage à la compilation), est assez rudimentaire : toutes les formes de transtypage sont acceptées, certaines conversions sont insérées automatiquement par le compilateur, et la plupart des abstractions fournies par le langage sont perméables. Le noyau Linux est écrit dans un dialecte du langage C. Le noyau du système Mac OS X d'Apple est également un dérivé d'Unix, et est donc aussi écrit dans ce langage.

Néanmoins ce langage n'est pas facile à analyser, car il est conçu pour être facilement écrit par des programmeurs humains. Certaines constructions sont ambigües [2], et de nombreux comportements sont implicites [3].

Si on veut analyser des programmes, il est plus pratique de travailler sur une représentation intermédiaire plus simple afin d'avoir moins de traitements dupliqués. Dans ce cas on ajoute une phase préliminaire à l'analyse, qui consiste à convertir le code à étudier vers cette représentation. On présente quelques candidats langages qui peuvent servir ce rôle :

Middle-ends

Les premiers candidats sont bien entendu les représentations intermédiaires utilisées dans les compilateurs C. Elles ont l'avantage d'accepter, en plus du C standard, les diverses extensions (GNU, Microsoft, Plan9) utilisées par la plupart des logiciels. En particulier, le noyau Linux repose fortement sur les extensions GNU.

GCC utilise une représentation interne nommée GIMPLE [Mer03]. Il s'agit d'une structure d'arbre écrite en C, reposant sur de nombreuses macros afin de cacher les détails d'implantation interne pouvant varier entre deux versions. Cette représentation étant réputée difficile à manipuler, le projet MELT [Sta11] permet de générer un *plugin* de compilateur écrit dans un dialecte de Lisp.

LLVM [LA04] est un compilateur développé par la communauté *open-source* puis sponsorisé par Apple. À la différence de GCC, sa base de code est écrite en C++. Il utilise une représentation intermédiaire qui peut être manipulée sous forme d'une structure de données C++, d'un fichier de code-octet compact, ou textuelle.

Cmm est une représentation interne utilisée pour la génération de code lors de la compilation d'OCaml [☞2], et disponible dans les sources du compilateur (il s'agit donc d'une structure de données OCaml). Ce langage a l'avantage d'être très restreint, mais malheureusement il n'existe pas directement de traducteur permettant de compiler C vers Cmm.

2. Selon qu'il existe un type nommé `a`, l'expression `(a)-(b)` sera interprétée comme le transtypage de `-(b)` dans le type `a`, ou la soustraction des deux expressions `(a)` et `(b)`.

3. Par exemple, une fonction acceptant un entier long peut être appelée avec un entier de taille plus petite. Celui-ci sera alors converti implicitement.

C- - [PJNO97] [☞1], dont le nom est inspiré du précédent, est un projet qui visait à unifier les langages intermédiaires utilisés par les compilateurs. L'idée est que, si un front-end peut émettre du C- - (sous forme de texte), il est possible d'obtenir du code machine efficace. Le compilateur Haskell GHC, par exemple, utilise une représentation intermédiaire très similaire à C- -.

Langages intermédiaires ad hoc

Comme le problème de construire une représentation intermédiaire adaptée à une analyse statique n'est pas nouveau, plusieurs projets ont déjà essayé d'y apporter une solution. Puisqu'ils sont développés en parallèle des compilateurs, le support des extensions est en général moins important dans ces langages.

CIL [NMRW02] est une représentation en OCaml d'un programme C, développée depuis 2002. Grâce à un mécanisme de *plugins*, elle permet de prototyper rapidement des analyses statiques de programmes C.

CompCert C, Clight et Cminor sont des langages intermédiaires utilisés dans Compcert, un compilateur certifié pour C [BDL06, AB07]. C'est-à-dire que les transformations sémantiques sont faites de manière prouvée. Ces langages intermédiaires sont utilisés pour les passes de front-end et de middle-end.

1.5 Le projet Penjili

En face du problème théorique et technique décrit dans la section 1.2, il faut mettre en perspective les problématiques industrielles liées à celui-ci. Les travaux présentés ici ont en effet été réalisés dans l'équipe de sécurité et sûreté logicielle d'EADS Innovation Works, dans le cadre d'une convention industrielle de formation par la recherche (CIFRE).

Aujourd'hui, la réussite de missions dépend de logiciels dont la taille est de plus en plus grande. Ainsi en cas de fautes dans ce genre de logiciel, on peut se retrouver face à grands impacts économiques, voire risquer des vies humaines. On comprend bien que les phases de vérification et de certification sont au cœur du cycle de vie des logiciels avioniques. A titre d'exemple, l'échec du premier vol d'Ariane 5 aurait certainement pu être évité si le logiciel de contrôle de vol avait été vérifié plus efficacement [Lan96].

Plusieurs méthodes existent pour éliminer les risques de fautes. En fait, deux approches duales sont nécessaires. La première consiste à mettre le logiciel dans des situations concrètes et à vérifier que la sortie correspond au résultat attendu : c'est la technique des tests. Les tests « boîte noire » consistent à tester en ayant à disposition uniquement les spécifications des modules à plusieurs échelles (par exemple : logiciel, module, classe, méthode). Au contraire, les tests dits « boîte blanche » sont écrits en ayant à disposition l'implémentation. Cela permet par exemple de s'assurer que chaque chemin d'exécution est emprunté. Cette manière de procéder est similaire à la preuve par neuf enseignée aux enfants : il est possible de prouver l'erreur, mais pas que le programme est correct.

L'approche des méthodes formelles, au contraire, permet de s'assurer de l'absence d'erreurs à l'exécution. Par exemple, l'analyse statique par interprétation abstraite permet d'étudier les relations exposées entre les variables afin d'en déduire les ensembles de valeurs dans lesquels elles évoluent. En s'assurant que ceux-ci sont « sûrs », on prouve l'absence d'erreurs de manière automatisée.

L'interprétation abstraite repose sur l'idée suivante : au lieu de considérer que les variables possèdent une valeur, on utilise un domaine abstrait qui permet de voir les variables comme possédant un ensemble de valeurs possibles.

On dit que l'approche est *sound* si l'abstraction d'un ensemble de valeurs est un surensemble de l'ensemble concret. Autrement dit, on réalise une surapproximation.

La zone « sûre » (correspondant aux exécutions sans erreurs) a une forme assez simple compte tenu des erreurs considérées : c'est un produit d'ensembles simples, comme des intervalles. L'ensemble des comportements réels du programme est au contraire d'une forme plus complexe et non calculable.

En calculant une approximation de ce dernier, de forme plus simple, on peut tester plus facilement que les comportements sont dans la zone sûre : le fait que l'analyse soit *sound*, c'est-à-dire que l'approximation ne manque aucun comportement, permet de prouver l'absence d'erreurs.

La figure 1.1 résume cette approche : l'ensemble des valeurs dangereuses est représenté par un ensemble hachuré, l'ensemble des comportements réels du programme est noté par des points, et l'approximation en gris. Plusieurs cas peuvent se produire. Ils correspondent aux cas suivants, de haut en bas puis de gauche à droite. Dans le premier on a prouvé, à la compilation, que le programme ne pourra pas comporter d'erreurs à l'exécution. Dans le deuxième, l'approximation recouvre les cas dangereux : on émet une alarme par manque de précision. Dans la troisième l'approximation n'est pas *sound* (par construction, on évite ce cas). Enfin, dans la quatrième, on émet une alarme à raison car il existe des comportements erronés. Toute la difficulté est donc de construire une surapproximation correcte mais conservant une précision suffisante.

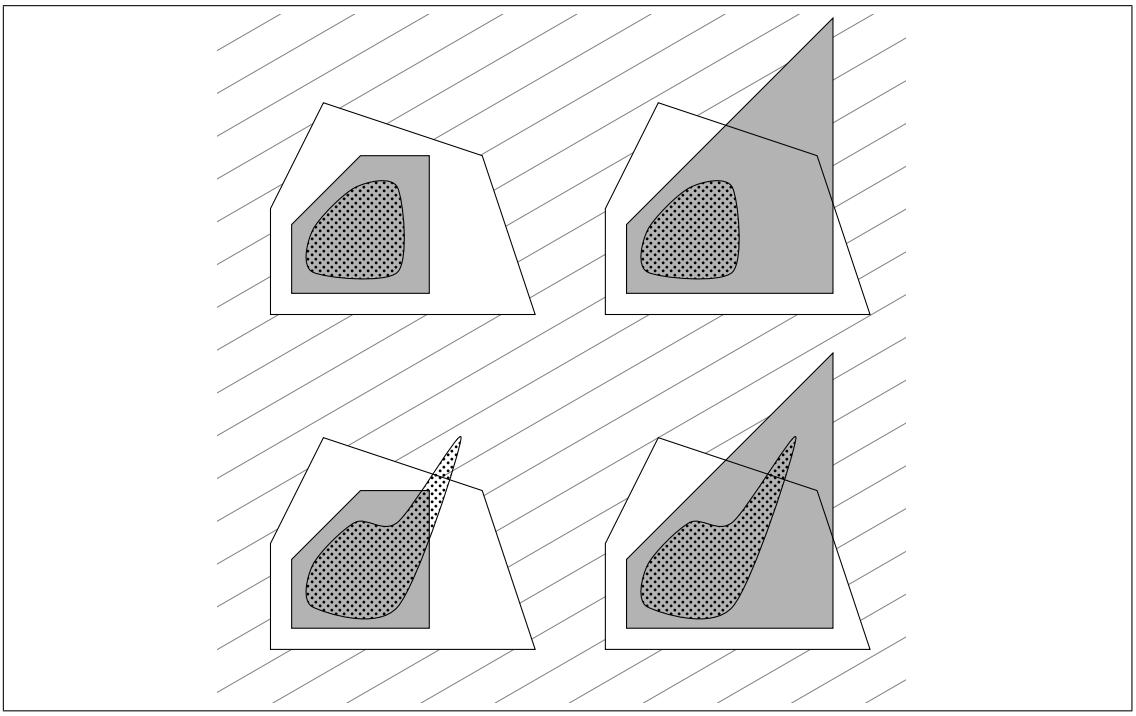


FIGURE 1.1: Surapproximation. L'ensemble des états erronés est hachuré. L'ensemble des états effectifs du programme, noté par des points, est approximé par l'ensemble en gris.

Pour construire cette surapproximation, on peut employer divers outils. Par exemple, un entier pourra être représenté par sa valeur minimale et sa valeur maximale (domaine abstrait des intervalles), et un pointeur sur un tableau peut être représenté par un ensemble de va-

riables associé à un décalage (*offset*) par rapport au début de la zone mémoire (domaine des pointeurs sur tableaux).

Dans ce sens, des outils fondés sur l'interprétation abstraite ont été développés chez EADS Innovation Works dans le cadre du projet Penjili [AH07].

Les analyses statiques ne manipulent pas directement du code C, mais un langage intermédiaire appelé NEWSPEAK [HL08]. Celui-ci est suffisamment expressif pour compiler la plupart des programmes C, y compris de nombreuses extensions GNU utilisées dans le noyau Linux (section 8.1), et des traducteurs automatiques depuis C et Ada existent (section 7.1).

Ensuite, ses instructions sont orthogonales et minimales : il existe en général une seule manière de faire les choses. Par exemple, le flot de contrôle est restreint à la boucle infinie et au saut en avant (« *break* » généralisé).

Enfin, lorsque certaines constructions sont ambigües, un choix est fait. Par exemple, l'évaluation des arguments d'une fonction est faite dans un ordre précis, les tailles des types sont indiquées à chaque déclaration de variable, etc.

Séparer le langage intermédiaire de la phase d'analyse permet de beaucoup simplifier l'analyseur statique. D'une part, les constructions redondantes comme les différents types de boucles ne sont traitées qu'une fois. D'autre part, lorsque le langage source est étendu (en supportant une nouvelle extension de C par exemple), l'analyseur n'a pas besoin d'être modifié.

Le langage NEWSPEAK, ainsi que les outils permettant de le manipuler, sont disponibles sous license libre sur [☞3]. L'analyseur statique Penjili, reposant sur ces outils, a été utilisé pour analyser des logiciels embarqués critiques de plusieurs millions de lignes de code. Ce dernier n'est pour le moment pas *open-source*. Tous ces outils sont écrits dans le langage OCaml [CMP03].

1.6 De l'avionique à l'informatique d'entreprise

Vérifier la sûreté des logiciels avioniques est critique, mais cela présente l'avantage que ceux-ci sont développés avec ces difficultés à l'esprit. Il est plus simple de construire un système sécurisé en connaissant toutes les contraintes d'abord, plutôt que de vérifier *a posteriori* qu'un système existant peut répondre à ces contraintes de sûreté.

Néanmoins cette manière de concevoir des logiciels est très coûteuse. Pour des composants qui sont moins critiques, il peut donc être intéressant de considérer des logiciels ou bibliothèques existantes, en particulier dans le monde de l'*open-source*.

Ces logiciels sont plus difficiles à analyser car ils sont écrits sans contraintes particulières. Non seulement toutes les constructions du langage sont autorisées, mêmes celles qui sont difficiles à traiter (transtypage, allocation dynamique, récursion, accès au système de fichiers, etc), mais aussi des extensions non standards peuvent être utilisées.

Programmes non autosuffisants La grande majorité des programmes ne se suffisent pas à eux-mêmes. En effet, ils interagissent presque toujours avec leur environnement ou appellent des fonctions de bibliothèque.

Cela veut dire qu'un fichier en cours d'analyse peut contenir des appels à des fonctions inconnues. Non seulement on n'a pas accès à leur code source, mais en plus on ne connaît pas *a priori* leur spécification. Une solution peut être de prévoir un traitement particulier pour celles-ci (par exemple en leur attribuant un type prédéfini).

Certaines interagissent directement avec le système d'exploitation, comme les fonctions d'ouverture ou d'écriture dans un fichier. D'autres modifient totalement le mode d'exécution du programme. Par exemple `pthread_create(&t, NULL, f, NULL)` lance l'exécution

de `f(NULL)` tout en continuant l'exécution de la fonction en cours dans un fil d'exécution concurrent.

Extensions du langage Par exemple, la figure 1.2 démontre l'influence de l'attribut `packed` (supporté par GCC) sur la compilation d'une structure. Sans celui-ci, les champs sont alignés de manière à faciliter les accès à la mémoire, par exemple en faisant démarrer les adresses de chaque champ sur un multiple de 4 octets (en gras). Cela nécessite d'introduire des octets de *padding* (en gris) qui ne sont pas utilisés. La taille totale de cette structure est donc de 12 octets.

```
struct s {
    char a;
    int b;
    short c;
};

struct s {
    char a;
    int b;
    short c;
} __attribute__((packed));
```

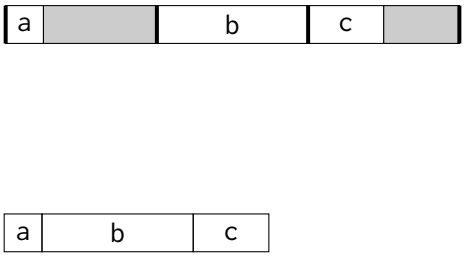


FIGURE 1.2: Utilisation de l'attribut non-standard `packed`

Au contraire, l'utilisation de `packed` supprime totalement le *padding* et permet de diminuer alors la taille de la structure à 7 octets seulement. Puisque `b` et `c` ne sont pas alignés, leur accès sera fait de manière moins efficace.

De manière générale, les compilateurs permettent de personaliser finement le code émis grâce à des extensions. Elles changent parfois le mode d'exécution des programmes d'une manière subtile et pas toujours bien spécifiée ni documentée.

1.7 Objectifs et contributions de la thèse

Le but de ce travail est de définir et d'implanter des analyses statiques « légères »sur le langage C, c'est-à-dire plus simples que les analyses de valeurs par interprétation abstraite. Nous proposons d'étendre NEWSPEAK pour analyser des propriétés de sécurité par typage sur du code non avionique. En effet, les types permettent de modéliser l'environnement d'exécution d'un programme (ici, les paramètres d'appels système) avec un grain assez grand, alors qu'être plus fin est difficile et nécessite de modéliser l'environnement.

Nos contributions sont les suivantes :

- Une première étape est de définir des restrictions sur le langage source. En effet, le langage C permet des conversions non sûres entre données, ce qui empêche le typage d'avoir un intérêt. On définit alors un modèle de langage impératif avec un modèle mémoire de plus haut niveau, interdisant ces constructions : SAFESPEAK.
- Sur ce langage on définit une sémantique opérationnelle, qui permet de raisonner sur l'exécution des programmes. On profite du caractère structuré des états mémoire pour exprimer cette sémantique en terme de lentilles bidirectionnelles, permettant de décrire la modification en profondeur de la mémoire.

- Au cœur de notre travail se trouve un système de types sûrs pour SAFESPEAK, ainsi que deux extensions validant l'approche typage pour résoudre notre problème de base, qui est la vérification des accès aux pointeurs utilisateur.
- Notre formalisation est accompagnée d'un prototype, basé sur NEWSPEAK. Cela permet d'utiliser les règles de typage précédemment définies (en les adaptant), tout en profitant des outils existants développés par EADS, qui utilisent NEWSPEAK. En particulier, cela permet de traiter des programmes écrits en C comme le noyau Linux. Ce prototype est disponible sous une license libre.

1.8 Plan de la thèse

Cette thèse est organisée en trois parties. La première décrit le contexte de ces travaux, ainsi que les solutions existantes. La deuxième expose notre solution, SAFESPEAK, d'un point de vue théorique. La troisième rend compte de la démarche expérimentale : comment la solution a été implantée et en quoi elle est applicable en pratique.

Dans la partie I, on présente tout d'abord le fonctionnement général d'un système d'exploitation. On y introduit aussi les problèmes de manipulation de pointeurs contrôlés par l'utilisateur. Ceux-ci sont centraux puisqu'on désire les restreindre. On fait ensuite un tour d'horizon des techniques existantes permettant de traiter ce problème par analyse statique de code source.

Dans la partie II, on décrit notre solution : le langage SAFESPEAK. Sa syntaxe y est d'abord décrite, puis sa sémantique ainsi qu'un système de types statiques. À ce niveau on a un bon support pour décrire des analyses statiques sur un langage impératif. On propose alors deux extensions du système de types. La première consiste à bien typer les entiers utilisés comme *bitmasks*. La seconde capture les problèmes d'adressage mémoire présents dans les systèmes d'exploitation. Pour ce faire, on ajoute des pointeurs contrôlés par l'utilisateur à la sémantique et au système de types. À chaque étape, c'est-à-dire avant et après ces ajouts, on établit une propriété de sûreté de typage reliant la sémantique d'exécution aux types statiques.

Dans la partie III, on documente la démarche expérimentale associée à ces travaux. L'implantation du système de types sur le langage NEWSPEAK est d'abord décrite, reposant sur l'algorithme W de Damas et Milner. La manière de compiler depuis du code C est également présentée. Ensuite, on applique cette implantation à deux cas d'étude concrets dans le noyau Linux. L'un est un bug ayant touché un pilote de carte graphique, et l'autre un défaut dans l'implantation de la fonction `ptrace`. Dans chaque cas, un pointeur dont la valeur est contrôlée par l'utilisateur crée un problème de sécurité car un utilisateur malveillant peut lire ou écrire dans l'espace mémoire réservé au noyau. En lançant notre prototype, l'analyse de la version non corrigée lève une erreur alors que, dans la version corrigée, un type correct est inféré. On montre ainsi que le système de types capture précisément ce genre d'erreur de programmation.

On conclut enfin en décrivant les possibilités d'extensions autant sur le point théorique qu'expérimental.

Première partie

Méthodes formelles pour la sécurité

Après avoir décrit le contexte général de ces travaux, nous décrivons leurs enjeux.

Le chapitre 2 explore plus en détail le fonctionnement d'un système d'exploitation, y compris la séparation du code en plusieurs niveaux de privilèges. L'architecture Intel 32 bits est prise comme support. En particulier, le mécanisme des appels système est décrit et on montre qu'une implantation naïve de la communication entre espaces utilisateur et noyau casse toute isolation.

Le chapitre 3 consiste en un tour d'horizon des techniques existantes en analyses de programmes. Ces analyses se centrent autour des problèmes liés à la vérification de code système ou embarqué, y compris le problème de manipulation mémoire évoqué dans le chapitre 2.

On conclut en introduisant notre solution : SAFESPEAK, un langage permettant de typer des programmes impératifs, plus précisément en ajoutant des types pointeurs abstraits.

CHAPITRE 2

SYSTÈMES D'EXPLOITATION

Le système d'exploitation est le programme qui permet à un système informatique d'exécuter d'autres programmes. Son rôle est donc capital et ses responsabilités, multiples. Dans ce chapitre, nous allons voir à quoi il sert, et comment il peut être implanté. Pour ce faire, nous présentons ici de quoi est constitué un système Intel 32 bits et ce dont on se sert pour y implanter un système d'exploitation.

2.1 Architecture physique

Un système informatique est principalement constitué d'un processeur (ou CPU pour *Central Processing Unit*), de mémoire principale (ou RAM pour *Random Access Memory*), et de divers périphériques.

Le processeur est constitué de plusieurs registres internes qui permettent d'encoder l'état dans lequel il se trouve : quelle est l'instruction courante (registre EIP), quelle est la hauteur de la pile système (registre ESP), etc. Son fonctionnement peut être vu de la manière la plus simple qui soit comme la suite d'opérations :

- charger depuis la mémoire la prochaine instruction ;
- *(optionnel)* charger depuis la mémoire les données référencées par l'instruction ;
- effectuer l'instruction ;
- *(optionnel)* stocker en la mémoire les données modifiées ;
- continuer avec l'instruction suivante.

Les instructions sont constituées d'un *opcode* (mnémonique indiquant quelle opération faire) et d'un ensemble d'opérandes. La signification des opérandes dépend de l'opcode, mais en général elles permettent de désigner les sources et la destination (on emploiera ici la syntaxe AT&T, celle que comprend l'assembleur GNU). Les opérandes peuvent avoir plusieurs formes : une valeur immédiate ($4), un nom de registre (`%eax`) ou une référence à la mémoire (directement : `addr` ou indirectement : `(%ecx)` [1]). On décrit les opcodes les plus utilisés, permettant de compiler un cœur de langage impératif :

- `mov src, dst` copie le contenu de `src` dans `dst`.
- `add src, dst` calcule la somme des contenus de `src` et `dst` et place ce résultat dans `dst`.

1. Cela consiste à interpréter le contenu du regitre ECX comme une adresse mémoire.

- `push src` place `src` sur la pile, c'est-à-dire que cette instruction enlève au pointeur de pile ESP la taille de `src`, puis place `src` à l'adresse mémoire de la nouvelle valeur ESP.
- `pop src` réalise l'opération inverse : elle charge le contenu de la mémoire à l'adresse ESP dans `src` puis incrémente ESP de la taille correspondante.
- `jmp addr` saute à l'adresse `addr` : c'est l'équivalent de `mov addr, %eip`.
- `call addr` sert aux appels de fonction : cela revient à `push %eip` puis `jmp addr`.
- `ret` sert à revenir d'une fonction : c'est l'équivalent de `pop %eip`.

Certaines de ces instructions font référence à la pile par le biais du registre ESP. Cette zone mémoire n'est pas gérée de manière particulière. Elle permet de gérer la pile des appels de fonction en cours grâce à la manière dont `jmp` et `ret` fonctionnent. Elle sert aussi à stocker les variables locales des fonctions.

À l'aide de ces quelques instructions on peut implanter des algorithmes impératifs. Mais pour faire quelque chose de visible, comme afficher à l'écran ou envoyer un paquet sur le réseau, cela ne suffit pas : il faut parler au reste du matériel.

Pour ceci, il y a deux techniques principales. D'une part, certains périphériques sont dits *memory-mapped* : ils sont associés à un espace mémoire particulier, qui ne permet pas de stocker des informations mais de lire ou d'écrire des données dans le périphérique. Par exemple, écrire à l'adresse `0xB8000` permet d'écrire des caractères à l'écran. L'autre système principal est l'utilisation des ports d'entrée/sortie. Cela correspond à des instructions spéciales `in %ax, port` et `out port, %ax` où `port` est un numéro qui correspond à un périphérique particulier. Par exemple, en écrivant dans le port `0x60`, on peut contrôler l'état des indicateurs lumineux du clavier PS/2.

2.2 Tâches et niveaux de privilèges

Alternance des tâches

Sans mécanisme particulier, le processeur exécuterait uniquement une suite d'instructions à la fois. Pour lui permettre d'exécuter plusieurs tâches, un système de partage du temps existe.

À des intervalles de temps réguliers, le système est programmé pour recevoir une interruption. C'est une condition exceptionnelle (au même titre qu'une division par zéro) qui fait sauter automatiquement le processeur dans une routine de traitement d'interruption. À cet endroit le code peut sauvegarder les registres et restaurer un autre ensemble de registres, ce qui permet d'exécuter plusieurs tâches de manière entrelacée. Si l'alternance est assez rapide, cela peut donner l'illusion que les programmes s'exécutent en même temps. Comme l'interruption peut survenir à tout moment, on parle de multitâche préemptif.

En plus de cet ordonnancement de processus, l'architecture Intel permet d'affecter des niveaux de privilège à ces tâches, en restreignant le type d'instructions exécutables, ou en donnant un accès limité à la mémoire aux tâches de niveaux moins élevés.

Le matérieul permet 4 niveaux de privilèges (nommés aussi *rings*) : le *ring* 0 est le plus privilégié, le *ring* 3, le moins privilégié. Dans l'exemple précédent, on pourrait isoler l'ordonnanceur de processus en le faisant s'exécuter en *ring* 0 alors que les autres tâches seraient en *ring* 3.

Mémoire virtuelle

À partir du moment où plusieurs processus s'exécutent de manière concurrente, un problème d'isolation se pose : si un processus peut lire dans la mémoire d'un autre, des informations peuvent fuiter ; et s'il peut y écrire, il peut en détourner l'exécution.

Le mécanisme de mémoire virtuelle permet de donner à deux tâches une vue différente de la mémoire : c'est-à-dire que vue de tâches différentes, une adresse contiendra une valeur différente (figure 2.1).

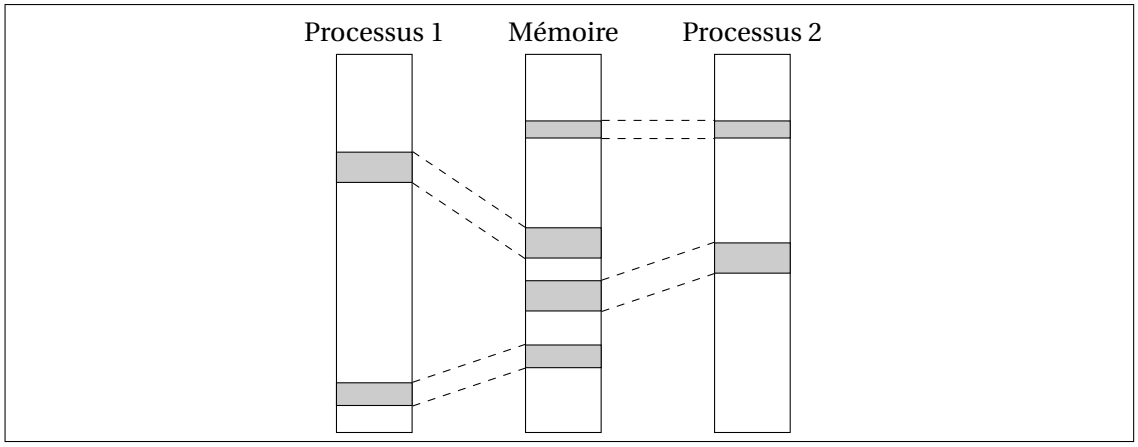


FIGURE 2.1: Mécanisme de mémoire virtuelle.

Ce mécanisme est contrôlé par valeur du registre CR3 : les 10 premiers bits d'une adresse virtuelle sont un index dans le répertoire de pages qui commence à l'adresse contenue dans CR3. À cet index, se trouve l'adresse d'une table de pages. Les 10 bits suivants de l'adresse sont un index dans cette page, donnant l'adresse d'une page de 4 kio (figure 2.2). Plus de détails sur l'utilisation de ce mécanisme seront donnés dans la section 8.2.

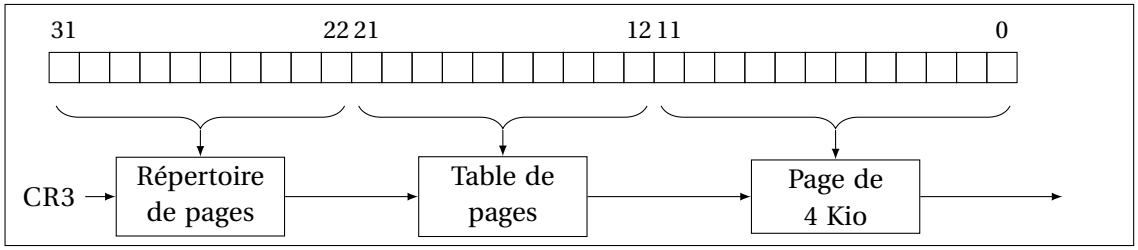


FIGURE 2.2: Implantation de la mémoire virtuelle

2.3 Appels système

Avec une telle isolation, tout le code qui est exécuté en *ring* 3 a une expressivité limitée. Il ne peut pas contenir d'instructions privilégiées comme `in` ou `out`, ni faire référence à des périphériques mappés en mémoire. C'est en effet au noyau d'accéder au matériel, et pas au code utilisateur.

Il est donc nécessaire d'appeler une routine du noyau depuis le code utilisateur. C'est le but des appels système. Cela consiste à coupler une fonction du *ring* 3 à une fonction du *ring* 0 : en appelant la fonction non privilégiée, le flot d'exécution se retrouve dans le noyau avec les bons privilèges.

Bien sûr, il n'est pas possible de faire directement un `call` puisque cela consisterait à faire un saut vers une zone plus privilégiée. Il y a plusieurs manières d'implanter ce mécanisme. Nous décrivons ici la technique historique à l'aide d'interruptions.

Le processeur peut répondre à des interruptions, qui sont des événements extérieurs. Cela permet d'écrire du code asynchrone. Par exemple, une fois qu'un long transfert mémoire est terminé, une interruption est reçue. D'autres interruptions dites logicielles peuvent arriver lorsqu'une erreur se produit. Par exemple, diviser par zéro provoque l'interruption 0, et tenter d'exécuter une instruction privilégiée provoque l'interruption 14. On peut aussi provoquer manuellement une interruption par une instruction `int` dédiée.

Une table globale définit, pour chaque numéro d'interruption, quelle est la routine à appeler pour la traiter, avec quel niveau de privilège, ainsi que le niveau de privilège requis pour pouvoir déclencher celle-ci avec l'instruction `int`.

Il est donc possible de créer une interruption purement logicielle (on utilise en général le numéro 128, soit `0x80`), déclenchable en *ring* 3 et traitée en *ring* 0. Les registres sont préservés, donc on peut les utiliser pour passer un numéro d'appel système (par exemple 3 pour `read()` et 5 pour `open()`) et leurs arguments.

2.4 Le *Confused Deputy Problem*

On a vu que les appels système permettent aux programmes utilisateur d'accéder aux services du noyau. Ils forment donc une interface particulièrement sensible aux problèmes de sécurité.

Comme pour toutes les interfaces, on peut être plus ou moins fin. D'un côté, une interface pas assez fine serait trop restrictive et ne permettrait pas d'implanter tout type de logiciel. De l'autre, une interface trop laxiste (« écrire dans tel registre matériel ») empêche toute isolation. Il faut donc trouver la bonne granularité.

Nous allons présenter ici une difficulté liée à la manipulation de mémoire au sein de certains types d'appels système.

Il y a deux grands types d'appels système. D'une part on trouve ceux qui renvoient un simple entier, comme `getpid` qui renvoie le numéro du processus appelant.

```
pid_t pid = getpid();
printf("%d\n", pid);
```

Ici, pas de difficulté particulière : la communication entre le *ring* 0 et le *ring* 3 est faite uniquement à travers les registres, comme décrit dans la section 8.2.

Mais la plupart des appels système communiquent de l'information de manière indirecte, à travers un pointeur. L'appellant alloue une zone mémoire dans son espace d'adressage et passe un pointeur à l'appel système. Ce mécanisme est utilisé par exemple par la fonction `gettimeofday` (figure 2.3).

Considérons une implantation naïve de cet appel système qui écrirait directement à l'adresse pointée. Si le pointeur fourni est dans l'espace d'adressage du processus, on est dans le cas d'utilisation normal et l'écriture est donc possible.

Si l'utilisateur passe un pointeur dont la valeur correspond à la mémoire réservée au noyau, que se passe-t-il ? Comme le déréférencement est fait dans le code du noyau, il est également fait en *ring* 0, et va pouvoir être réalisé sans erreur : l'écriture se fait et potentiellement une structure importante du noyau est écrasée.

Un utilisateur malveillant peut donc utiliser cet appel système pour écrire à n'importe quelle adresse dans l'espace d'adressage du noyau. Ce problème vient du fait que l'appel

```
struct timeval tv;
struct timezone tz;
int z = gettimeofday(&tv, &tz);
if (z == 0) {
    printf( "tv.tv_sec = %ld\ntv.tv_usec = %ld\n"
            "tz.tz_minuteswest = %d\ntz.tz_dsttime = %d\n",
             tv.tv_sec, tv.tv_usec,
             tz.tz_minuteswest, tz.tz_dsttime
          );
}
```

FIGURE 2.3: Appel de `gettimeofday`

système utilise les privilèges du noyau au lieu de celui qui contrôle la valeur des paramètres sensibles. Cela s'appelle le *Confused Deputy Problem*[Har88].

La bonne solution est de tester dynamiquement la valeur du pointeur : s'il pointe en espace noyau, il faut indiquer une erreur plutôt que d'écrire. Sinon, il peut toujours y avoir une erreur, mais au moins le noyau est protégé.

Dans le noyau, un ensemble de fonctions permet d'effectuer des copies sûres. La fonction `access_ok` réalise le test décrit précédemment. Les fonctions `copy_from_user` et `copy_to_user` réalisent une copie de la mémoire après avoir fait ce test. Ainsi, l'implantation correcte de l'appel système `gettimeofday` fait appel à celle-ci (figure 2.4).

```
SYSCALL_DEFINE2(gettimeofday, struct timeval __user *, tv,
                struct timezone __user *, tz)
{
        if (likely(tv != NULL)) {
                struct timeval ktv;
                do_gettimeofday(&ktv);
                if (copy_to_user(tv, &ktv, sizeof(ktv)))
                        return -EFAULT;
        }
        if (unlikely(tz != NULL)) {
                if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
                        return -EFAULT;
        }
        return 0;
}
```

FIGURE 2.4: Implantation de l'appel système `gettimeofday`

Pour préserver la sécurité du noyau, il est donc nécessaire de vérifier la valeur de tous les pointeurs dont la valeur est contrôlée par l'utilisateur. Cette conclusion est assez contraignante, puisqu'il existe de nombreux endroits dans le noyau où des données proviennent de l'utilisateur. Il est donc raisonnable de vouloir vérifier automatiquement et statiquement l'absence de tels défauts.

CHAPITRE 3

ANALYSES STATIQUES EXISTANTES

Dans ce chapitre, nous présentons un tour d'horizon des techniques existantes permettant d'analyser des programmes. En particulier, on s'intéresse à la propriété d'isolation décrite dans le chapitre 2, mais on ne se limite pas à celle-ci : il est également intéressant de considérer des analyses développées pour d'autres propriétés (comme par exemple s'assurer de l'absence d'erreurs à l'exécution), celles-ci pouvant potentiellement s'adapter.

L'analyse statique de programmes est un sujet de recherche actif depuis l'apparition de l'informatique en tant que science. On commence par en présenter une classification, puis on montrera des exemples pertinents permettant d'analyser du code système ou embarqué.

3.1 Taxonomie

Techniques statiques et dynamiques L'analyse peut être faite au moment de la compilation, ou au moment de l'exécution. En général on peut obtenir des informations plus précises de manière dynamique, mais cela ne prend en compte que les parties du programme qui seront vraiment exécutées. Un autre problème des techniques dynamiques est qu'il est souvent nécessaire d'instrumenter l'environnement d'exécution (ce qui — dans le cas où cela est possible — peut se traduire par un impact en performances). L'approche statique, en revanche, nécessite de construire à l'arrêt une carte mentale du programme, ce qui n'est pas toujours possible dans certains langages.

Les techniques dynamiques sont néanmoins les plus répandues, puisqu'elles sont plus simples à mettre en œuvre et permettent de trouver des erreurs pendant le processus de développement. De plus, on peut considérer qu'un programme avec une forte couverture par les tests a de grandes chances d'être correct pour toutes les entrées. Par exemple, dans l'avionique civile, le processus de développement demande d'être très rigoureux pour les tests fonctionnels et structurels afin de détecter le code ou les branchements non atteints.

Mais pour s'assurer de la correction d'un programme, on ne peut pas s'appuyer uniquement sur les tests — ou de manière générale sur des analyses dynamiques — car il est souvent impossible d'étudier l'ensemble complet de tous les comportements possibles. Par exemple, si un bug se présente lors d'une interaction entre deux composants qui n'a pas été testée, il passera inaperçu durant la phase de tests. Pour cette raison, la plupart des analyses présentées ici sont statiques.

Cohérence et complétude Le but d'une analyse statique est de catégoriser les programmes selon s'ils satisfont ou non un ensemble de propriétés fixées à l'avance. Malheureusement,

cela n'est que rarement possible car l'ensemble des valeurs possibles lors de l'exécution d'un programme quelconque n'est pas un ensemble calculable (théorème de Rice [Ric53]). Autrement dit, il ne peut exister une procédure de décision prenant un programme et le déclarant correct ou incorrect. Un résultat similaire est qu'on ne peut pas écrire une procédure qui détermine si un programme arbitraire boucle indéfiniment ou pas (le problème de l'arrêt).

Il n'est donc pas possible d'écrire un analyseur statique parfait, détectant exactement les problèmes. Toute technique statique va donc de se retrouver dans au moins un des cas suivants :

- un programme valide (pour une propriété donnée) est rejeté : on parle de *faux positif*.
- un programme invalide n'est pas détecté : on parle de *faux négatif*.

En général, et dans notre cas, on préfère s'assurer que les programmes acceptés possèdent la propriété recherchée, quitte à en rejeter certains. C'est l'approche que nous retiendrons. Tolérer les faux négatifs n'est cependant pas toujours une mauvaise idée. Par exemple, si le but est de trouver des constructions dangereuses dans les programmes, on peut signaler certains cas qui empiriquement valent d'être vérifiés manuellement.

Par ailleurs la plupart des techniques ne concernent que les programmes qui terminent. On étudie donc la correction, ou les propriétés des termes convergents. Prouver automatiquement que l'exécution ne boucle pas est une propriété toute autre qui n'est pas ici considérée.

3.2 Méthodes syntaxiques

L'analyse la plus simple consiste à traiter un programme comme du texte, et à y rechercher des motifs dangereux. Ainsi, utiliser des outils comme `grep` permet parfois de trouver un grand nombre de vulnérabilités [Spe05].

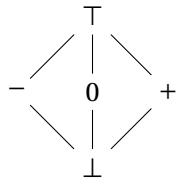
On peut continuer cette approche en recherchant des motifs mais en étant sensible à la syntaxe et au flot de contrôle du programme. Cette notion de *semantic grep* est présente dans l'outil Coccinelle [BDH+09, PTS+11] : on peut définir des *patches sémantiques* pour détecter ou modifier des constructions particulières.

Ces techniques sont utiles parce qu'elles permettent de plonger rapidement dans le code, en identifiant par exemple des appels à des fonctions dangereuses. En revanche, cela n'est possible que lorsque les propriétés que l'on recherche sont plutôt locales. Elles offrent également peu de garantie puisqu'elles ne prennent pas en compte la sémantique d'exécution du langage : il faudra en général vérifier manuellement la sortie de ces analyses.

3.3 Analyse de valeurs et interprétation abstraite

L'interprétation abstraite est une technique d'analyse générique qui permet de simuler statiquement tous les comportements d'un programme [CC77, CC92]. Un exemple d'application est de calculer les bornes de variation des variables pour s'assurer qu'aucun débordement de tableau n'est possible [AH07].

L'idée est d'associer à chaque ensemble concret de valeurs une représentation abstraite. Sur celle-ci, on peut définir des opérations indépendantes de la valeur exacte des données, mais préservant l'abstraction (figure 3.1). Par exemple, les règles comme « − » × « − » = « + » définissent le domaine abstrait des signes arithmétiques. Les domaines ont une structure de treillis, c'est-à-dire qu'ils possèdent les notions d'ordre partiel et d'union de valeurs. En calculant les extrêmes limites d'une variable, on obtient le domaine des intervalles.

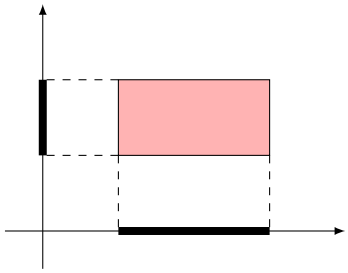


$$\gamma(-) = \mathbb{R}^-$$
$$\gamma(0) = \{0\}$$
$$\gamma(+) = \mathbb{R}^+$$

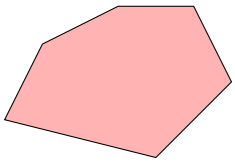
FIGURE 3.1: Domaine des signes

De tels domaines ne capturent aucune relation entre variables. Ils sont dits non relationnels. Lorsque plusieurs variables sont analysées en même temps, utiliser de tels domaines consiste à considérer un produit cartésien d'ensembles abstraits (figure 3.2(a)).

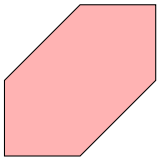
Des domaines abstraits plus précis permettent de retenir celles-ci. Pour ce faire, il faut modéliser l'ensemble des valeurs des variables comme un tout. Parmi les domaines relationnels courants on peut citer : le domaine des polyèdres [CH78], permettant de retenir tous les invariants affines entre variables (figure 3.2(b)) ; le domaine des zones [Min01a], permettant de représenter des relations affines de la forme $v_i - v_j \leq c$ (figure 3.2(c)) ; ou encore le domaine des octogones [Min01b] qui est un compromis entre les polyèdres et les zones. Il permet de représenter les relations $\pm v_i \pm v_j \leq c$ (figure 3.2(d)).



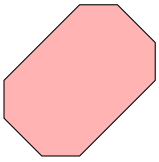
(a) Domaine non relationnel



(b) Domaine des polyèdres



(c) Domaine des zones



(d) Domaine des octogones

FIGURE 3.2: Quelques domaines abstraits

En plus des domaines numériques, il est nécessaire d'employer des domaines spécialisés dans la modélisation de la mémoire. Cela est nécessaire pour pouvoir prendre en compte les pointeurs. Par exemple, on peut représenter un pointeur par un ensemble de variables possiblement pointées et une valeur abstraite représentant le décalage (*offset*) du pointeur par rapport au début de la zone mémoire. Cette valeur peut elle-même être abstraite par un domaine numérique.

Au delà des domaines eux-mêmes, l'analyse se fait sous forme d'un calcul de point fixe. La manière la plus simple est d'utiliser un algorithme de *liste de travail*, décrit par exemple

dans [SRH96]. Les raffinements en revanche sont nombreux.

Dès [CC77] il est remarqué que la terminaison de l'analyse n'est assurée que si le treillis des valeurs abstraites est de hauteur finie, ou qu'un opérateur d'élargissement (*widening*) ∇ est employé. L'idée est qu'une fois qu'on a calculé quelques termes d'une suite croissante, on peut réaliser une projection de celle-ci. Par exemple, dans le domaine des intervalles, $[0;2] \nabla [0;3] = [0;+\infty[$. On atteint alors un point fixe mais qui est plus grand que celui qu'on aurait obtenu sans cette accélération : on perd en précision. Pour en gagner, on peut redescendre sur le treillis des points fixe avec une suite d'itérations décroissantes [Gra92, GGTZ07].

En termes d'ingéniérie logicielle, implanter un analyseur statique est un défi en soi. En plus des domaines abstraits, d'un itérateur, il faut traduire le code source à analyser dans un langage, et traduire les résultats de l'analyse en un ensemble d'alarmes à présenter à l'utilisateur.

Cette technique est très puissante : si un interprète abstrait *sound* (réalisant une surapproximation, c'est-à-dire ne manquant aucun programme incorrect) analyse un programme et ne renvoie pas d'erreur, alors on a prouvé que le programme est correct (par rapport aux propriétés que vérifient les domaines abstraits). Cela a été appliqué avec succès avec les analyseurs Astrée [Mau04, CCF$^+$05, CCF$^+$09] chez Airbus ou CGS [VB04] à la NASA par exemple.

Cependant, ces analyses sont difficiles à mettre en œuvre. Avec des domaines abstraits classiques comme ceux présentés ci-dessus, les premières analyses peuvent remonter un nombre prohibitif de fausses alarmes. Pour « aider » l'analyse, il faut soit annoter le code soit développer des domaines abstraits *ad hoc* au programme à analyser.

Il existe également des analyseurs statiques combinant l'interprétation abstraite avec d'autres techniques et qui ne sont pas *sound*, c'est-à-dire qu'ils peuvent manquer des comportements erronés. Leur approche est plus d'aider le programmeur à détecter certains types de bugs pendant le développement. On citer l'exemple de Coverity [BBC$^+$10], qui publie régulièrement des rapports de qualité sur certains logiciels *open-source*. Néanmoins, de part leur aspect non *sound*, les analyses réalisées ne peuvent pas être assimilées à de la vérification formelle en tant que telle.

Enfin, l'interprétation abstraite n'est pas la seule technique pour analyser finement les valeurs d'un programme. Par exemple, le système Saturn [ABD$^+$07], conçu pour analyser du code système écrit en C, utilise des clauses logiques et un solveur SAT pour manipuler des invariants sur la mémoire. En particulier il traite le problème des pointeurs utilisateur en utilisant une analyse de forme « pointe-sur » [BA08]. Un autre exemple est le *model checking* [CE81], qui consiste à énumérer l'ensemble des états qui peut atteindre un système. Comme l'interprétation abstraite, le but des cette analyse est d'être très précis, au détriment d'un temps de calcul important dans certains cas.

3.4 Typage

Le typage, introduit dans la section 1.3, peut aussi être utilisé pour la vérification de programmes. On peut le voir comme une manière de catégoriser les types de données manipulés par la machine, mais également à plus au niveau comme une manière d'articuler les différents composants d'un programme. Mais on peut aussi programmer avec les types, c'est-à-dire utiliser le compilateur (dans le cas statique) ou l'environnement d'exécution (dans le cas dynamique) pour vérifier des propriétés écrites par le programmeur.

Systèmes ad hoc Les systèmes de types les plus simples expriment des contrats esssentiellement liés à la sûreté d'exécution, pour ne pas utiliser des valeurs de types incompatibles entre eux. Mais il est possible d'étendre le langage avec des annotations plus riches,

par exemple en vérifiant statiquement que des listes ne sont pas vides [KcS07] ou, dans le domaine de la sécurité, d'empêcher des fuites d'information [LZ06].

Qualificateurs de types Dans le cas particulier des vulnérabilités liées à une mauvaise utilisation de la mémoire, les développeurs du noyau Linux ont ajouté un système d'annotations au code source. Un pointeur peut être décoré d'une annotation `__kernel` ou `__user` selon s'il est sûr ou pas. Celles-ci sont ignorées par le compilateur, mais un outil d'analyse statique ad-hoc nommé Sparse [☞6] peut être utilisé pour détecter les cas les plus simples d'erreurs. Il demande aussi au programmeur d'ajouter de nombreuses annotations dans le programme.

Cette solution se rapproche de la solution décrite dans ce manuscrit. Ce système d'annotations sur les types a été formalisé sous le nom de *qualificateurs de types* [FJKA06] : chaque type peut être décoré d'un ensemble de qualificateurs (à la manière de `const`), et des règles de typage permettent d'établir des propriétés sur le programme.

Plus précisément, les jugements de typage de la forme $\Gamma \vdash e : t$ sont remplacés par des jugements de typage qualifiés $\Gamma \vdash e : t\ q$. Les qualificateurs q permettent d'exprimer plusieurs jugements. Par exemple, on peut étudier le fait qu'une variable soit constante ou pas, que sa valeur soit connue à la compilation, ou encore qu'elle puisse être nulle ou pas. La spécificité de ce système est que les qualificateurs sont ordonnés, du plus spécifique au moins spécifique, et que l'on forme alors un treillis à partir de ces informations. Partant des deux caractéristiques précédentes, on forme le treillis de la figure 3.3. Le qualificateur `const` désigne les données dont la valeur ne change pas au cours de l'exécution ; `dynamic` celles qui ne peuvent pas être connues à la compilation ; et `nonzero` celles qui ne peuvent jamais être nulles. Le cube sur lequel se trouvent les qualificateurs correspond à une relation d'ordre, du plus spécifique (en bas) au plus général (en haut). ø correspond à un ensemble vide de qualificateurs.

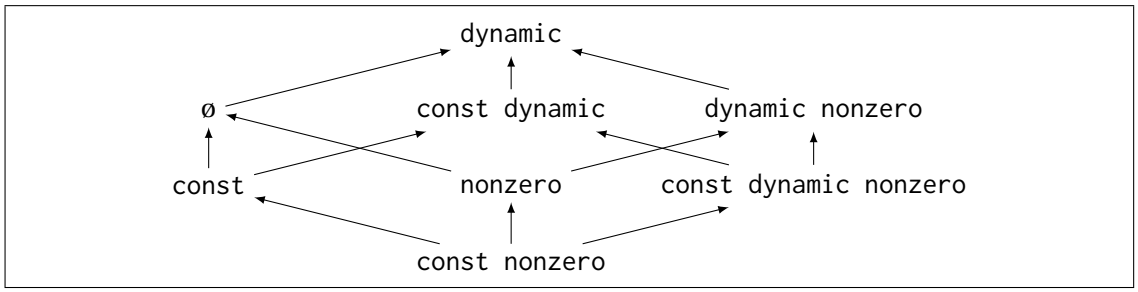


FIGURE 3.3: Treillis de qualificateurs

Cette relation d'ordre $\preceq$ entre qualificateurs induit une relation de sous-typage $\sqsubseteq$ entre les types qualifiés : si $q \preceq q'$, alors $t\ q \sqsubseteq t\ q'$.

Ces analyses ont été implantée dans l'outil CQual. Ce système peut servir à inférer les annotations `const` [FFA99], à l'analyse de souillure pour les chaînes de format [STFW01] (pouvant poser des problèmes de sécurité [New00]) et des propriétés dépendantes du flot de contrôle, comme des invariants sur les verrous [FTA02], à rapprocher du concept de *typestates* [SY86]. Il a également été appliqué à la classe de vulnérabilités sur les pointeurs utilisateurs dont il est ici l'objet [JW04].

Cette approche est assez proche de la nôtre : on donne un type différent aux pointeurs selon leur provenance. Néanmoins cela est très différent. Une première différence est dans le langage considéré. CQual s'applique sur un lambda-calcul à références, alors que pour étudier du code C nous présentons un modèle mémoire avec pile explicite plus proche de la machine. D'autre part, le système de types de CQual est fondamentalement modifié pour prendre en compte ces opérations, alors que dans le nôtre il s'agit d'une simple extension qui

ne nécessite pas de modifier toutes les règles de typage. La conclusion de la partie II, page 95, sera dédiée à une comparaison entre ces solutions.

Le système Flow Caml [Sim03] repose également sur cette approche, en ajoutant une étiquette de sécurité à chaque type. Par exemple, les entiers sont typés `'a int` où `'a` est le niveau de sécurité associé. Couplé à un système d'effets, cela permet de suivre la provenance de chaque expression. Cette technique d'analyse de flot permet d'encoder de nombreuses propriétés de sécurité [SM03].

Ces techniques de typage sont séduisantes parce qu'elles sont en général simples à mettre en place : à l'aide d'un ensemble de règles, on attribue un type à chaque expression. Si le typage se termine sans erreur, alors on est assuré de la correction du programme (par rapport aux propriétés capturées par le système de types).

Le typage statique peut également être implanté de manière efficace. Même si l'inférence peut, dans certains cas, atteindre une complexité exponentielle, la plupart des systèmes de types peuvent être vérifiés en pratique dans un temps linéaire en la taille du programme considéré.

3.5 Langages sûrs

Une autre approche est de concevoir un langage à la fois bas niveau et sûr, permettant d'exprimer des programmes proches de la machine tout en interdisant les constructions dangereuses.

Le langage Cyclone [JMG+02] est conçu comme un C « sûr ». Afin d'apporter plus de sûreté au modèle mémoire de C, des tests dynamiques sont ajoutés, par exemple aux endroits où des conversions implicites peuvent poser problème. Le langage se distingue par le fait qu'il possède plusieurs types de pointeurs : des pointeurs classiques (`int *`), des pointeurs « jamais nuls » (`int @`; un test à l'exécution est alors inséré), et des « pointeurs lourds » (`int ?`; qui contiennent des informations sur la zone mémoire pointée). L'arithmétique des pointeurs n'est autorisée que sur ces derniers, rendant impossibles les débordements de tableaux (ceux-ci étant détectés au pire à l'exécution). Le problème des pointeurs fous [1] est résolu en utilisant un système de régions [GMJ+02], inspiré des travaux de Jouvelot, Talpin et Tofte [TJ92, TT93, TT94]. Cela permet d'interdire statiquement les constructions où l'on déréférence un pointeur faisant référence à une région de mémoire qui n'est plus allouée (par exemple en évitant de retourner l'adresse d'une variable locale).

Le langage Rust [☞5] développé par Mozilla prend une approche similaire en distinguant plusieurs types de pointeurs pour gérer la mémoire de manière plus fine. Les *managed pointers* (notés `@int`) utilisent un ramasse-miettes pour libérer la mémoire allouée lorsqu'ils ne sont plus accessibles. Les *owning pointers* (notés `~int`) décrivent une relation 1 à 1 entre deux objets, comme les `std::unique_ptr` de C++ : la mémoire est libérée lorsque le pointeur l'est. Les *borrowed pointers* (notés `&int`) correspondent aux pointeurs obtenus en prenant l'adresse d'un objet, ou d'un champ d'un objet. Une analyse statique faite lors de la compilation s'assure que la durée de vie de ces pointeurs est plus courte que l'objet pointé, afin d'éviter les pointeurs fous. Cette analyse est également fondée sur les régions. Une fonction qui retourne l'adresse d'une variable locale sera donc rejetée par le compilateur. Enfin, le dernier type est celui des *raw pointers* (notés `*int`), pour lesquels le langage n'apporte aucune

1. Les pointeurs fous, encore appelés pointeurs fantômes ou *dangling pointers*, correspondent à une zone mémoire invalide ou expirée. Il y a deux sources principales de pointeurs fous : les variables de type pointeur non initialisées, et les pointeurs vers des objets dont la mémoire a été libérée. C'est par exemple ce qui arrive aux adresses de variables locales une fois que la fonction dans laquelle elles ont été définies retourne.

garantie (il faut d'ailleurs encapsuler chaque utilisation dans un bloc marqué explicitement `unsafe`). Ils sont équivalents aux pointeurs de C.

Les systèmes de types de ces projets apportent dans le langage différents types de pointeurs. Cela permet de manipuler finement la mémoire, à la manière des *smart pointers* de C++. Ceux-ci sont des types de données abstraits permettant de déterminer quelle partie du code est responsable de la libération de la mémoire associée au pointeur.

De cette approche on retient surtout l'analyse de régions de Rust qui permet de manipuler de manière sûre les adresses des variables locales, et les pointeurs lourds de Cyclone, qui apportent une sûreté à l'arithmétique de pointeurs, au prix d'un test dynamique.

Ces techniques sont utiles pour créer des nouveaux programmes sûrs, mais on ne peut pas les appliquer pour étudier la correction de logiciels existants. Dans cette perspective, le langage CCured [NCH+05] a pour but d'ajouter un système de types forts à C (y compris pour des programmes existants). Dans les cas où il n'est pas possible de prouver que le programme s'exécutera correctement, des vérifications à l'exécution sont ajoutées. Cependant, cela nécessite une instrumentation dynamique qui se paye en performances et interdit la certification, car l'environnement d'exécution doit être inchangé. Le compilateur Fail-Safe C [Oiw09] utilise une approche similaire permettant de garantir la sûreté d'exécution des programmes C tout en respectant la totalité de la norme C89.

3.6 Logique de Hoare

Une technique pour vérifier statiquement des propriétés sur la sémantique d'un programme a été formalisée par Robert Floyd [Flo67] et Tony Hoare [Hoa69].

Elle consiste à écrire les invariants qui sont maintenus à un point donné du programme. Ces propositions sont écrites dans une logique $\mathscr{L}$. Chaque instruction i est annotée d'une pré-condition P et d'une post-condition Q, ce que l'on note $\{P\}\ i\ \{Q\}$. Cela signifie que, si P est vérifiée et que l'exécution de i se termine, alors Q sera vérifiée.

En plus des règles de $\mathscr{L}$, des règles d'inférence traduisent la sémantique du programme ; par exemple la règle de composition est :

$$\frac{\{P\}\ i_1\ \{Q\} \qquad \{Q\}\ i_2\ \{R\}}{\{P\}\ i_1; i_2\ \{R\}}\ (\textsc{Hoare-Seq})$$

Les pré-conditions peuvent être renforcées et les post-conditions relâchées :

$$\frac{\vdash_{\mathscr{L}} P' \Rightarrow P \qquad \{P\}\ i\ \{Q\} \qquad \vdash_{\mathscr{L}} Q \Rightarrow Q'}{\{P'\}\ i\ \{Q'\}}\ (\textsc{Hoare-Consequence})$$

Il est alors possible d'annoter le programme avec ses invariants formalisés de manière explicite dans $\mathscr{L}$. Ceux-ci seront vérifiés à la compilation lorsque c'est possible, sinon à l'exécution.

La règle de conséquence permet de séparer les propriétés du programme lui-même : plusieurs niveaux d'annotations sont possibles, du moins précis au plus précis. En fait, il est même possible d'annoter chaque point de contrôle par l'ensemble d'annotations vide : $\{T\}\ i\ \{T\}$ est toujours vrai.

Augmenter graduellement les pré- et post-conditions est néanmoins assez difficile, puisqu'il peut être nécessaire de modifier l'ensemble des conditions à la fois. Cette difficulté est

mentionnée dans [DRS03], où un système de programmation par contrats est utilisé pour vérifier la correction de routines de manipulation de chaînes en C.

Ce type d'annotations a été implanté par exemple pour le langage Java dans le système JML [LBR06] ou pour le langage C# dans Spec# [BLS05]. Il est aussi possible d'utiliser cette technique pour annoter du code assembleur de bas niveau [MG07].

3.7 Assistants de preuve

Avec un système de types classique, le fait qu'un terme (au sens « expression » ou « instruction ») soit bien typé amène quelques propriétés sur son exécution, par exemple, le fait que seulement un ensemble réduit d'erreurs puisse arriver (comme la division par zéro).

En enrichissant le langage des types, on peut augmenter l'expressivité du typage. Par exemple, on peut former des types « entier pair », « vecteur de n entiers », ou encore « liste triée d'entiers ».

Habituellement, les termes peuvent dépendre d'autres termes (par composition) ou de types (par des annotations). Les types peuvent également dépendre d'autres types (par composition de types : par exemple, un couple de a et de b a pour type $a * b$). Enrichir l'expressivité du typage revient essentiellement à introduire des termes dans les types, comme n dans l'exemple précédent du vecteurs de n entiers. C'est pourquoi on parle de types dépendants. Parmi les langages proposant ces types on peut citer Coq [The04], Agda [BDN09] ou Isabelle [NPW02].

Dans un langage classique, la plupart des types sont habitables, c'est-à-dire qu'il existe des termes ayant ces types. En revanche, avec les types dépendants ce n'est pas toujours vrai : par exemple « vecteur de -1 entiers » n'a pas d'habitants. Ainsi, pouvoir construire un terme d'un type donné est une information en soi.

On peut voir ce phénomène sous un autre angle : les termes sont à leur type ce que les preuves sont à leur théorème. Exhiber un terme ayant un type revient à donner la preuve d'un théorème. À l'aide de cette correspondance, il est possible de voir un algorithme de vérification de typage comme un algorithme de vérification de preuve automatique. Ces preuves ne portent pas forcément sur des programmes. Par exemple, le théorème des 4 couleurs a été prouvé en Coq [Gon07].

Cette technique est très complexe à mettre en œuvre, puisqu'il faut encoder toutes les propriétés voulues dans un formalisme de très bas niveau (du niveau de la théorie des ensembles). De plus, l'inférence de types devient rapidement indécidable.

Conclusion

Il existe de nombreuses techniques pour vérifier du code système ou embarqué. Il y a divers choix à faire entre l'expressivité, l'intégration de tests dynamiques ou la facilité de mise en œuvre.

Pour résoudre le problème des pointeurs utilisateurs dans les noyaux, le typage statique est une solution performante et assez pragmatique, puisqu'elle peut s'appliquer à des programmes existants. Son expressivité limitée nous empêche de reposer entièrement sur elle pour garantir l'absence d'erreur dans les programmes systèmes (par exemple, le typage est mal adapté pour détecter les divisions par zéro). C'est pourquoi nous approchons la sûreté de la manière suivante :

- Tout d'abord, on utilise le typage pour manipuler les données de manière compatible : les types des opérations et fonctions sont vérifiés à la compilation.

- Ensuite, les accès aux tableaux et aux pointeurs sont vérifiés dynamiquement. Dans le cas où une erreur est déclenchée, l'exécution s'arrête plutôt que de corrompre la mémoire. La pile est également nettoyée à chaque retour de fonction afin d'éviter les pointeurs fous.
- Enfin, les pointeurs provenant de l'espace utilisateur sont repérés statiquement afin que leur déréférencement se fasse au sein de fonctions sûres. Cela permet de préserver l'isolation entre le noyau et l'espace utilisateur.

CONCLUSION DE LA PARTIE I

Nous avons montré que l'écriture de noyaux de systèmes d'exploitation nécessite de manipuler des données provenant d'une zone non sûre, l'espace utilisateur. Parmi ces données, il arrive de récupérer des pointeurs qui servent à passer des données par référence à l'appelant, dans certains appels système. Si on déréférence ces pointeurs sans vérifier qu'ils pointent bien vers une zone mémoire également contrôlée par l'appelant, on risque de lire ou d'écrire dans des zones mémoires réservées au noyau seul.

Nous proposons une technique de typage pour détecter ces cas dangereux. Pour ce faire, il faudra tout d'abord définir un langage impératif bien typable que nous appellerons SAFESPEAK. Celui-ci s'appuie sur le langage NEWSPEAK, qui est un langage intermédiaire développé par EADS dans le but de vérifier la sûreté de programmes C embarqués. À ce titre, il existe un compilateur qui est capable de traduire du code C vers NEWSPEAK.

Définir la syntaxe et la sémantique de SAFESPEAK permet d'écrire et d'évaluer des programmes. Mais cela reste trop permissif, car on ne rejette pas les programmes qui manipulent les données de manière incohérente. La première étape est donc de définir un système de types pour classifier les expressions et fonctions selon le type de valeurs que leur évaluation produit.

Une fois SAFESPEAK défini et étendu d'un système de types, nous lui ajoutons des constructions permettant d'écrire du code noyau, et en particulier on lui ajoute des pointeurs utilisateur. il s'agit de pointeurs dont la valeur est contrôlée par un utilisateur interagissant avec le programme via un appel système. Ces pointeurs ont un type distinct des pointeurs habituels.

En résumé, le but de cette thèse est donc de définir un langage intermédiaire proche de C, mais bien typable, et de lui adjoindre un système de types tel que les programmes bien typés manipulent les pointeurs utilisateur sans causer de problèmes de sécurité.

Deuxième partie

Un langage pour l'analyse de code système : SAFESPEAK

Dans cette partie, nous allons présenter un langage impératif modélisant une sous-classe « bien typable » du langage C. Le chapitre 4 décrit sa syntaxe, ainsi que sa sémantique d'exécution. À ce point, de nombreux programmes acceptés peuvent provoquer des erreurs à l'exécution.

Afin de rejeter ces programmes incorrects, on définit ensuite dans le chapitre 5 une sémantique statique s'appuyant sur un système de types simples. Des propriétés de sûreté de typage sont ensuite établies, permettant de catégoriser l'ensemble des erreurs à l'exécution possibles.

Le chapitre 6 commence par étendre notre langage avec une nouvelle classe d'erreurs à l'exécution, modélisant les accès à la mémoire utilisateur catégorisés comme dangereux dans le chapitre 2. Une extension au système de types du chapitre 5 est ensuite établie, et on prouve que les programmes ainsi typés ne peuvent pas atteindre ces cas d'erreur.

Trois types d'erreurs à l'exécution sont possibles :

- les erreurs liées aux valeurs : lorsqu'on tente d'appliquer à une opération des valeurs incompatibles (additionner un entier et une fonction par exemple). L'accès à des variables qui n'existent pas rentre aussi dans cette catégorie.
- les erreurs mémoire, qui résultent d'un débordement de tableau, du déréférencement d'un pointeur invalide ou d'arithmétique de pointeur invalide.
- les erreurs de sécurité, qui consistent en le déréférencement d'un pointeur dont la valeur est contrôlée par l'espace utilisateur. Celles-ci sont uniquement possibles en contexte noyau.

L'introduction des types simples enlève la possibilité de rencontrer le premier cas. Il reste en revanche toujours possible de rencontrer des erreurs mémoire ainsi que des divisions par zéro. Éliminer ces erreurs dépasse le cadre de ce travail.

En présence d'extensions permettant de manipuler des pointeurs utilisateurs, une extension naïve du système de types ne suffit pas à empêcher la présence d'erreurs de sécurité. Celles si sont évitées par l'ajout de règles de typage supplémentaires.

CHAPITRE 4

SYNTAXE ET SÉMANTIQUE D'ÉVALUATION

Dans ce chapitre, on décrit le support de notre travail : un langage impératif nommé SAFESPEAK, sur lequel s'appuieront les analyses de typage des chapitres 5 et 6.

Le langage C [KR88] est un langage impératif, conçu pour être un « assembleur portable ». Ses types de données et les opérations associées sont donc naturellement de très bas niveau.

Ses types de données sont établis pour représenter les mots mémoire manipulables par les processeurs : essentiellement des entiers et flottants de plusieurs tailles. Les types composés correspondent à des zones de mémoire contigües, homogènes (dans le cas des tableaux) ou hétérogènes (dans le cas des structures).

Une des spécificités de C est qu'il expose au programmeur la notion de pointeur, c'est-à-dire des variables qui représentent directement une adresse en mémoire. Les pointeurs peuvent être typés (on garde une indication sur le type de l'objet stocké à cette adresse) ou « non typés ». Dans ce dernier cas, ils ont en fait le type `void *` qui est compatible avec n'importe quel type pointeur.

Son système de types rudimentaire ne permet pas d'avoir beaucoup de garanties sur la sûreté du programme. En effet, aucune vérification n'est effectuée en dehors de celles faites par le programmeur.

Le but ici est de définir SAFESPEAK, un langage plus simple mais qui permettra de raisonner sur une certaine classe de programmes C.

Tout d'abord, on commence par présenter les notations qui accompagneront le reste des chapitres. Cela inclut la notion de lentille, qui est utilisée pour définir les accès profonds à la mémoire. Cela permet de résoudre le problème de mettre à jour une sous-valeur (par exemple un champ de structure) d'une variable, par exemple. Les lentilles permettent de définir de manière déclarative que pour faire cette opération, il faut obtenir l'ancienne valeur de la variable, puis calculer une nouvelle valeur en remplaçant une sous-valeur, avant de replacer cette nouvelle valeur à sa place en mémoire. En pratique, on définira deux lentilles : une qui relie un état mémoire à la valeur d'une variable, et une qui relie une valeur à une de ses sous-valeurs. Avec cette technique, on peut définir en une seule fois les opérations de lecture et d'écriture de sous-valeurs imbriquées.

Ensuite, on présente SAFESPEAK en soi, c'est-à-dire sa syntaxe, ainsi qu'une description de ses caractéristiques principales. En particulier, le modèle mémoire est détaillé, ainsi que les valeurs manipulées par le langage.

Enfin, on décrit une sémantique opérationnelle pour ce langage. Cela permet de définir précisément l'exécution d'un programme SAFESPEAK au niveau de la mémoire.

L'implantation de ces analyses est faite dans le chapitre 7. Puisque SAFESPEAK n'est qu'un modèle, il s'agira d'adapter ces règles de typage sur NEWSPEAK, qui possède un modèle mé-

moire plus bas niveau.

4.1 Notations

Inférence

La sémantique opérationnelle consiste en la définition d'une relation de transition $\cdot \to \cdot$ entre états de l'interprète [1].

Cette relation est définie inductivement sur la syntaxe du programme. Plutôt que de présenter l'induction explicitement, elle est représentée par des jugements logiques et des règles d'inférence, de la forme :

$$\frac{P_1 \quad \dots \quad P_n}{C}\ (\textsc{Nom})$$

Les P_i sont les prémisses, et C la conclusion. Cette règle s'interprète de la manière suivante : si les P_i sont vraies, alors C est vraie.

Certaines règles n'ont pas de prémisse, ce sont des axiomes :

$$\frac{}{A}\ (\textsc{Ax})$$

Compte-tenu de la structure des règles, la dérivation d'une preuve (l'ordre dans lequel les règles sont appliquées) pourra donc être vue sous la forme d'un arbre où les axiomes sont les feuilles, en haut, et la conclusion est la racine, en bas.

$$\dfrac{\dfrac{\dfrac{}{A_1}\ (\textsc{r3}) \quad \dfrac{}{A_2}\ (\textsc{r4})}{B_1}\ (\textsc{r2}) \quad \dfrac{\dfrac{}{A_3}\ (\textsc{r6})}{B_2}\ (\textsc{r5})}{C}\ (\textsc{r1})$$

Listes

X^* est l'ensemble des suites finies de X, indexées à partir de 1. Si $u \in X^*$, on note $|u|$ le nombre d'éléments de u (le cardinal de son domaine de définition). Pour $i \in [1;|u|]$, on note $u_i = u(i)$ le i-ème élément de la suite.

On peut aussi voir les suites comme des listes : on note $[]$ la suite vide, telle que $|[]| = 0$. On définit en outre la construction de suite de la manière suivante : si $x \in X$ et $u \in X^*$, la liste $x :: u \in X^*$ est la liste v telle que :

$$\begin{aligned} v_1 &= x \\ \forall i \in [1;|u|], v_{i+1} &= u_i \end{aligned}$$

Cela signifie que la tête de liste (x dans la liste $x :: u$) est toujours accessible à l'indice 1.

1. Dans le chapitre 5, la relation de typage $\cdot \vdash \cdot : \cdot$ sera définie par la même technique.

Lentilles

Dans la définition de la sémantique de SAFESPEAK, on utilise des *lentilles bidirectionnelles*. Cette notion n'est pas propre à la sémantique des programmes. Il s'agit d'une technique permettant de relier la modification d'un objet à la modification d'un de ses sous-composants. Cela a plusieurs applications possibles. En programmation fonctionnelle pure (sans mutation), on ne peut pas mettre à jour partiellement les valeurs composées comme des enregistrements (*records*). Pour simuler cette opération, on a en général une opération qui permet de définir un nouvel enregistrement dans lequel seul un champ a été mis à jour. C'est ce qui se passe avec le langage Haskell [OGS08] : `r { x = 5 }` représente une valeur enregistrement égale à `r` sur tous les champs, sauf pour le champ `x` où elle vaut 5. Utiliser des lentilles revient à ajouter dans le langage la notion de champ en tant que valeur de première classe. Elles ont l'avantage de pouvoir se composer, c'est-à-dire que si on a un champ nommé x qui contient un champ nommé y, alors on peut modifier le champ du champ automatiquement.

Dans ce cadre, les lentilles ont été popularisées par Van Laarhoven [vL11]. Puisque cela sert à manipuler des données arborescentes, on peut aussi appliquer cet outil aux systèmes de bases de données ou aux documents structurés comme par exemple en XML [FGM+07].

Dans notre cas, cela permettra par exemple de modifier un élément d'un tableau qui est un champ de structure de la variable nommée x dans le 3e cadre de pile.

Définition 4.1 (Lentille)**.** *Étant donnés deux ensembles R et A, une* lentille $\mathscr{L} \in \text{LENS}_{R,A}$ *(ou* accesseur*) est un moyen d'accéder en lecture ou en écriture à une sous-valeur appartenant à A au sein d'une valeur appartenant à R (pour* record*). Elle est consistuée des opérations suivantes :*

- *une fonction de lecture* $\text{get}_{\mathscr{L}} : R \to A$
- *une fonction de mise à jour* $\text{put}_{\mathscr{L}} : (A \times R) \to R$

telles que pour tous $a \in A, a' \in A, r \in R$:

$$\text{put}_{\mathscr{L}}(\text{get}_{\mathscr{L}}(r), r) = r \qquad \text{(GETPUT)}$$
$$\text{get}_{\mathscr{L}}(\text{put}_{\mathscr{L}}(a, r)) = a \qquad \text{(PUTGET)}$$
$$\text{put}_{\mathscr{L}}(a', \text{put}_{\mathscr{L}}(a, r)) = \text{put}_{\mathscr{L}}(a', r) \qquad \text{(PUTPUT)}$$

On note $\mathscr{L} = \langle \text{get}_{\mathscr{L}} | \text{put}_{\mathscr{L}} \rangle$.

GETPUT *signifie que si on lit une valeur puis qu'on la réécrit, l'objet n'est pas modifié ;* PUTGET *décrit l'opération inverse : si on écrit une valeur dans le champ, c'est la valeur qui sera lue ; enfin,* PUTPUT *évoque le fait que chaque écriture est totale : quand deux écritures se suivent, seule la seconde compte.*

Une illustration se trouve dans la figure 4.1.

Exemple 4.1 (Lentilles de tête et de queue de liste)**.** *Soit E un ensemble. On rappelle que E^* désigne l'ensemble des listes d'éléments de E.*

On définit les fonctions suivantes. Notons qu'elles ne sont pas définies sur la liste vide []*, qui pourra être traitée comme un cas d'erreur.*

$$\text{get}_T(t :: q) = t \qquad \text{put}_T(t', t :: q) = t' :: q$$
$$\text{get}_Q(t :: q) = q \qquad \text{put}_Q(q', t :: q) = t :: q'$$

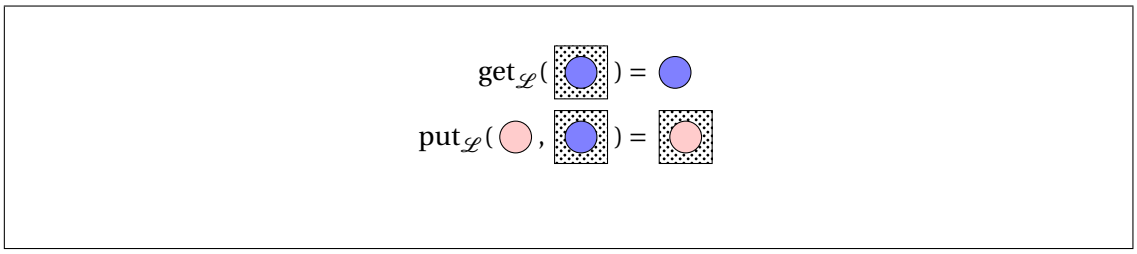


FIGURE 4.1: Fonctionnement d'une lentille

Alors $T = \langle \text{get}_T | \text{put}_T \rangle \in \text{LENS}_{E^*,E}$ *et* $Q = \langle \text{get}_Q | \text{put}_Q \rangle \in \text{LENS}_{E^*,E^*}$.
On a par exemple :

$$\text{get}_T(1 :: 6 :: 1 :: 8 :: []) = 1 \qquad \text{put}_Q(4 :: 2 :: [], 3 :: 6 :: 1 :: 5 :: []) = 3 :: 4 :: 2 :: []$$

Définition 4.2 (Lentille indexée)**.** *Les objets de certains ensembles R sont composés de plusieurs sous-objets accessibles à travers un indice* $i \in I$*. Une lentille indexée est une fonction* Δ *qui associe à un indice i une lentille entre R et un de ses champs* A_i *:*

$$\forall i \in I, \exists A_i, \Delta(i) \in \text{LENS}_{R,A_i}$$

On note alors :

$$r[i]_\Delta \stackrel{\text{def}}{=\!=} \text{get}_{\Delta(i)}(r)$$
$$r[i \leftarrow a]_\Delta \stackrel{\text{def}}{=\!=} \text{put}_{\Delta(i)}(a, r)$$

Un exemple est illustré dans la figure 4.2.

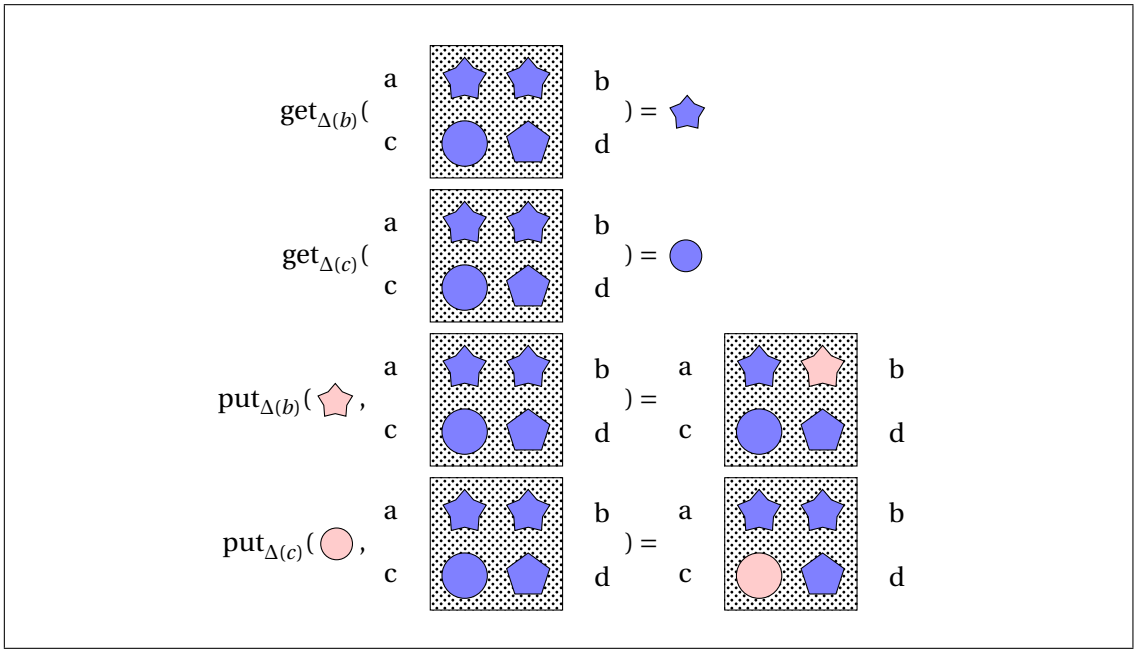


FIGURE 4.2: Fonctionnement d'une lentille indexée

Exemple 4.2 (Lentille « n^{e} élément d'un tuple »)**.** *Soient $n \in \mathbb{N}$, et n ensembles $E_1, \ldots, E_n$.*
Pour tout $i \in [1; n]$, on définit :

$$g_i((x_1, \ldots, x_n)) = x_i$$
$$p_i(y, (x_1, \ldots, x_n)) = (x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_n)$$

Définissons $T(i) = \langle g_i | p_i \rangle$. Alors $T(i) \in \text{LENS}_{(E_1 \times \ldots \times E_n), E_i}$.
Donc T est une lentille indexée, et on a par exemple :

$$\begin{aligned}(3,1,4,1,5)[2]_T &= \text{get}_{T(2)}((3,1,4,1,5)) \\ &= 1\end{aligned}$$

$$\begin{aligned}(9,2,6,5,3)[3 \leftarrow 1]_T &= \text{put}_{T(3)}(1, (9,2,6,5,3)) \\ &= (9,2,1,5,3)\end{aligned}$$

La notation $3 \leftarrow 1$ peut surprendre, mais elle est à interpréter comme « en remplaçant l'élément d'indice 3 par 1 ».

Définition 4.3 (Composition de lentilles)**.** *Soient $\mathscr{L}_1 \in \text{LENS}_{A,B}$ et $\mathscr{L}_2 \in \text{LENS}_{B,C}$.*
La composition de $\mathscr{L}_1$ et $\mathscr{L}_2$ est la lentille $\mathscr{L} \in \text{LENS}_{A,C}$ définie de la manière suivante :

$$\text{get}_{\mathscr{L}}(r) = \text{get}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_1} r)$$
$$\text{put}_{\mathscr{L}}(a, r) = \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1} r), r)$$

On notera alors $\mathscr{L} = \mathscr{L}_1 \ggg \mathscr{L}_2$ (« $\mathscr{L}_1$ flèche $\mathscr{L}_2$ »).

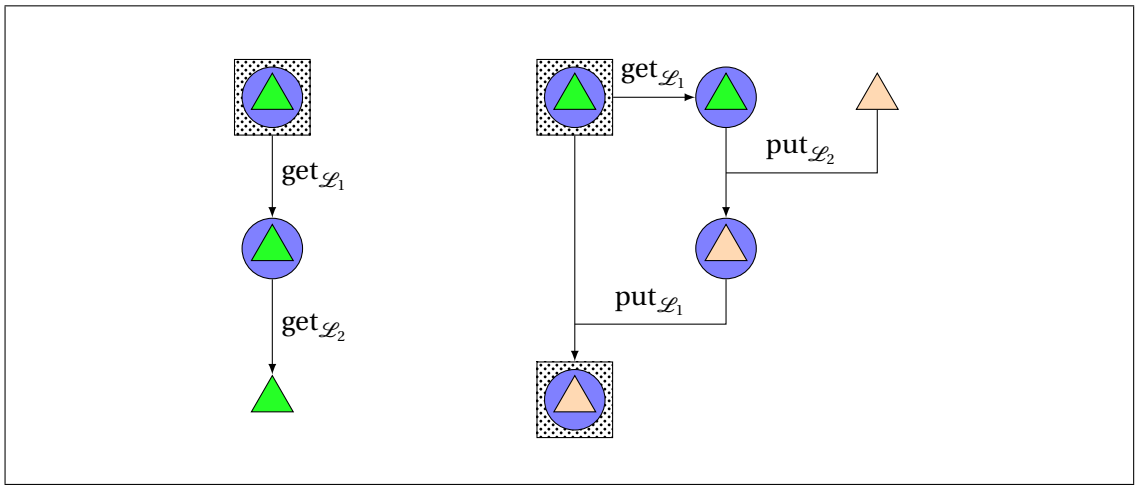


FIGURE 4.3: Composition de lentilles

Cette définition est illustrée dans la figure 4.3. Une preuve que la composition est une lentille est donnée en annexe D.1.

Constantes	$c ::= n$	Entier
	$\mid d$	Flottant
	$\mid$ NULL	Pointeur nul
	$\mid ()$	Valeur unité
Expressions	$e ::= c$	Constante
	$\mid \boxminus\, e$	Opération unaire
	$\mid e \boxplus e$	Opération binaire
	$\mid lv$	Accès mémoire
	$\mid lv \leftarrow e$	Affectation
	$\mid \&lv$	Pointeur
	$\mid f$	Fonction
	$\mid e(e_1, \ldots, e_n)$	Appel de fonction
	$\mid \{l_1 : e_1; \ldots; l_n : e_n\}$	Structure
	$\mid [e_1; \ldots; e_n]$	Tableau
Valeurs gauches	$lv ::= x$	Variable
	$\mid lv.l_S$	Accès à un champ
	$\mid lv[e]$	Accès à un élément
	$\mid *e$	Déréférencement
Fonctions	$f ::= \text{fun}(x_1, \ldots, x_n)\{i\}$	Arguments, corps

FIGURE 4.4: Syntaxe des expressions

4.2 Syntaxe

Les figures 4.4 et 4.5 présentent notre langage intermédiaire. Il contient la plupart des fonctionnalités présentes dans les langages impératifs comme C.

Parmi les expressions, les constantes comportent les entiers et flottants, ainsi que le pointeur NULL qui correspond à une valeur par défaut pour les pointeurs, et la valeur unité () qui pourra être retournée par les fonctions travaillant par effets de bord uniquement.

Les accès mémoire en lecture et écriture se font au travers de valeurs gauches (*left values* ou *lvalues*) : comme en C, elles tiennent leur nom du fait que ce sont ces constructions qui sont à gauche du signe d'affectation. En plus des variables, on obtient une valeur gauche en accédant par nom à un champ ou par indice à un élément d'une valeur gauche, ou encore en appliquant l'opérateur $*$ de déréférencement à une expression. Pour assister le typage, l'accès à un champ doit être décoré du type complet S, mais cette annotation est ignorée lors de l'évaluation. Les valeurs gauches correspondent aussi à l'unité d'adressage : c'est-à-dire que les pointeurs sont construits en prenant l'adresse d'une valeur gauche avec l'opérateur &.

Les fonctions sont des expressions comme les autres, contrairement à C où elles sont forcément déclarées globalement. Cela veut dire qu'on peut affecter une fonction f à une va-

Instructions	$i ::=$ PASS	Instruction vide
	$\mid\ i; i$	Séquence
	$\mid\ e$	Expression
	$\mid$ DECL $x = e$ IN$\{i\}$	Déclaration de variable
	$\mid$ IF$(e)\{i\}$ELSE$\{i\}$	Alternative
	$\mid$ WHILE$(e)\{i\}$	Boucle
	$\mid$ RETURN(e)	Retour de fonction
Phrases	$p ::= x = e$	Variable globale
	$\mid\ e$	Évaluation d'expression
Programme	$P ::= (p_1, \ldots, p_n)$	Phrases

FIGURE 4.5: Syntaxe des instructions

riable x et l'appeller avec $x(a_1, a_2)$. Il est aussi possible de déclarer une fonction au sein d'une fonction. Cependant cela ne respecte pas l'imbrication lexicale : dans la fonction interne il n'est pas possible de faire référence à des variables locales de la fonction externe, seulement à des variables globales. En mémoire les fonctions sont donc uniquement représentées par leur code : il n'y a pas de fermetures.

Enfin, on trouve aussi des expressions permettant de construire des valeurs composées : les structures et les tableaux.

Les instructions sont typiques de la programmation impérative. SAFESPEAK comporte bien sûr l'instruction vide qui ne fait rien et la séquence qui chaîne deux instructions.

Une expression peut être évaluée dans un contexte d'instruction, pour ses effets de bord. Remarquons que l'affectation est une expression, qui renvoie la valeur affectée. Cela permet d'écrire $x \leftarrow (y \leftarrow z)$, comme dans un programme C où on écrirait `x = y = z`.

Il est également possible de déclarer une variable locale avec DECL $x = v$ IN$\{i\}$. x est alors une nouvelle variable visible dans i avec pour valeur initiale v.

L'alternative et la conditionnelle sont classiques ; en revanche, on ne fournit qu'un seul type de boucle et pas de saut (instruction `goto`).

Les opérateurs sont donnés dans la figure 4.6. Ils correspondent à ceux du langage C. La différence principale est que les opérations sur les entiers, flottants et pointeurs sont annotés avec le type de données sur lequel ils travaillent. Par exemple « + » désigne l'addition sur les entiers et « +. » l'addition sur les flottants. Les opérations de test d'égalité, en revanche, sont possibles pour les types numériques, les pointeurs, ainsi que les types composés de types comparables.

4.3 Mémoire et valeurs

L'interprète que nous nous apprêtons à définir manipule des valeurs qui sont associées aux variables du programme.

La mémoire est constituée de variables (toutes mutables), qui contiennent des valeurs. Ces variables sont organisées, d'une part en un ensemble de variables globales, et d'autre

Opérateurs binaires	$\boxplus ::= +, -, \times, /, \%$	Arithmétique entière
	$\mid \; +., -., \times., /.$	Arithmétique flottante
	$\mid \; +_p, -_p$	Arithmétique de pointeurs
	$\mid \; \leq, \geq, <, >$	Comparaison sur les entiers
	$\mid \; \leq ., \geq ., < ., > .$	Comparaison sur les flottants
	$\mid \; =, \neq$	Tests d'égalité
	$\mid \; \&, \mid, \hat{}$	Opérateurs bit à bit
	$\mid \; \&\&, \mid\mid$	Opérateurs logiques
	$\mid \; \ll, \gg$	Décalages
Opérateurs unaires	$\boxminus ::= +, -$	Arithmétique entière
	$\mid \; +., -.$	Arithmétique flottante
	$\mid \; \sim$	Négation bit à bit
	$\mid \; !$	Négation logique

FIGURE 4.6: Syntaxe des opérateurs

part en une pile de contextes d'appel (qu'on appellera donc aussi cadres de pile, ou *stack frames* en anglais). Cette structure empilée permet de représenter les différents contextes à chaque appel de fonction : par exemple, si une fonction s'appelle récursivement, plusieurs instances de ses variables locales sont présentes dans le programme. Le modèle mémoire présenté ici de permet pas l'allocation dynamique sur un tas. Cette limitation sera détaillée dans le chapitre 9.

La structure de pile des variables locales permet de les organiser en niveaux indépendants : à chaque appel de fonction, un nouveau cadre de pile est créé, comprenant ses paramètres et ses variables locales. Au contraire, pour les variables globales il n'y a pas de système d'empilement, puisque ces variables sont accessibles depuis tout point du programme.

Pour identifier de manière non ambigüe une variable, on note simplement x la variable globale nommée x, et (n, x) la variable locale nommée x dans le n^{e} cadre de pile [2].

Les affectations peuvent avoir la forme $x \leftarrow e$ où x est une variable et e est une expression, mais pas seulement. En effet, à gauche de $\leftarrow$ on trouve en général non pas une variable mais une valeur gauche (par définition). Pour représenter quelle partie de la mémoire doit être accédée par cette valeur gauche, on introduit la notion de chemin φ. Un chemin est une valeur gauche évaluée : les cas sont similaires, sauf que tous les indices sont évalués. Par exemple, $\varphi = (5, x).p$ représente le champ « p » de la variable x dans le 5^{e} cadre de pile. C'est à ce moment qu'on évalue les déréférencements qui peuvent apparaître dans une valeur gauche.

Les valeurs, quant à elles, peuvent avoir les formes suivantes (résumées sur la figure 4.7) :

- $\widehat{c}$: une constante. La notation circonflexe permet de distinguer les constructions syntaxique des constructions sémantiques. Par exemple, à la syntaxe 3 correspond la valeur $\widehat{3}$.

2. Les paramètres de fonction sont traités comme des variables locales et se retrouvent dans le cadre correspondant.

Les valeurs entières sont les entiers signés sur 32 bits, c'est-à-dire entre -2^{31} à $2^{31}-1$. Mais ce choix est arbitraire : on aurait pu choisir des nombres à 64 bits, par exemple. Les flottants sont les flottants IEEE 754 de 32 bits [oEE08].

Il n'y a pas de distinction entre procédures et fonctions ; toutes les fonctions doivent renvoyer une valeur. Celles qui ne retournent pas de valeur « intéressante » renvoient alors une valeur d'un type à un seul élément noté $()$, et donc le type sera noté UNIT. Cette notation évoque un n-uplet à 0 composante.

- $\widehat{\&}\ \varphi$: une référence mémoire. Ce chemin correspond à un pointeur sur une valeur gauche. Par exemple, l'expression $\&x$ s'évalue en $\widehat{\&}\ \varphi = \widehat{\&}\ (5, x)$ si x désigne lexicalement une variable dans le 5e cadre de pile.
- $\widehat{[v_1; \ldots; v_n]}$: un tableau. C'est une valeur composée qui contient un certain nombre (connu à la compilation) de valeurs d'un même type, par exemple 100 entiers. On accède à ces valeurs par un indice entier. C'est une erreur (Ω_{array}) d'accéder à un tableau en dehors de ses bornes, c'est-à-dire en dehors de $[0; n-1]$ pour un tableau à n éléments. Pareillement, $\widehat{[\cdot]}$ permet de désigner les valeurs tableau. Par exemple, si x vaut 2 et y vaut 3, l'expression $[x, y]$ s'évaluera en la valeur $\widehat{[2,3]}$
- $\{l_1 : \widehat{v_1; \ldots; l_n} : v_n\}$: une structure. C'est une valeur composée mais hétérogène. Les différents éléments (appelés *champs*) sont désignés par leurs noms l_i (pour *label*). Dans le programme, le nom de champ l_i est décoré de la définition complète de la structure S. Celle-ci n'est pas utilisée dans l'évaluation et sera décrite au chapitre 5. Comme précédemment, on note $\widehat{\{\cdot\}}$ pour dénoter les valeurs.
- $\widehat{f}$: une fonction. On garde en mémoire l'intégralité de la définition de la fonction (liste de paramètres, de variables locales et corps). Même si les fonctions locales sont possibles, il n'est pas possible d'accéder aux variables de la portée entourante depuis la fonction intérieure (il n'y a pas de fermetures). Contrairement à C, les fonctions ne sont pas des cas spéciaux. Par exemple, les fonctions globales sont simplement des variables globales de type fonctionnel, et les « pointeurs sur fonction » de C sont remplacés par des variables de type fonction.
- Ω : une erreur. Par exemple le résultat l'évaluation de $5/0$ est Ω_{div}.

Les erreurs peuvent être classifiées en deux grand groupes : d'une part, Ω_{field}, Ω_{var} et Ω_{typ} sont des erreurs de typage dynamique, qui arrivent lorsqu'on accède dynamiquement à des données qui n'existent pas ou qu'on manipule des types de données incompatibles. D'autre part, Ω_{div}, Ω_{array} et Ω_{ptr} correspondent à des valeurs mal utilisées. Le but du système de types du chapitre 5 sera d'éliminer complètement les erreurs du premier groupe.

4.4 Interprète

La figure 4.8 résume comment ces valeurs sont organisées. Une pile est une liste de cadres de piles, et un cadre de pile est une liste de couples (nom, valeur). Un état mémoire m est un couple (s, g) où s est une pile et g un cadre de pile (qui représente les variables globales). On note $|m| = |s|$ la hauteur de la pile (en nombre de cadres).

Enfin, l'interprétation est définie comme une relation $\cdot \to \cdot$ entre états Ξ ; ces états sont d'une des formes suivantes :

- un couple $\langle e, m\rangle$ où e est une expression et m un état mémoire. m est l'état mémoire sous lequel l'évaluation sera réalisée. Par exemple $\langle 3, ([], [x \mapsto 3])\rangle \to \langle \widehat{3}, ([], [x \mapsto 3])\rangle$ L'évaluation des expressions est détaillée dans la section 4.10.

Valeurs	$v ::= \widehat{c}$	Constante
	$\mid \widehat{\&}\,\varphi$	Référence mémoire
	$\mid \{l_1 : \widehat{v_1; \ldots; l_n} : v_n\}$	Structure
	$\mid [\widehat{v_1; \ldots; v_n}]$	Tableau
	$\mid \widehat{f}$	Fonction
	$\mid \Omega$	Erreur
Chemins	$\varphi ::= a$	Adresse
	$\mid \widehat{\varphi.l}$	Accès à un champ
	$\mid \widehat{\varphi[n]}$	Accès à un élément
Adresses	$a ::= (n, x)$	Variable locale
	$\mid (x)$	Variable globale
Erreur	$\Omega ::= \Omega_{array}$	Débordement de tableau
	$\mid \Omega_{ptr}$	Erreur de pointeur
	$\mid \Omega_{div}$	Division par zéro
	$\mid \Omega_{field}$	Erreur de champ
	$\mid \Omega_{var}$	Variable inconnue
	$\mid \Omega_{typ}$	Données incompatibles

FIGURE 4.7: Valeurs

Pile	$s ::= []$	Pile vide
	$\mid \{x_1 \mapsto v_1; \ldots; x_n \mapsto v_n\} :: s$	Ajout d'un cadre
État mémoire	$m ::= (s, \{x_1 \mapsto v_1; \ldots; x_n \mapsto v_n\})$	Pile, globales
État d'interprète	$\Xi ::= \langle e, m \rangle$	Expression, mémoire
	$\mid \langle i, m \rangle$	Instruction, mémoire
	$\mid \Omega$	Erreur

FIGURE 4.8: Composantes d'un état mémoire

- un couple $\langle i, m\rangle$ où i est une instruction et m un état mémoire. La réduction des instructions est traitée dans la section 4.11. Par exemple, $\langle (x \leftarrow 3; y \leftarrow x), m\rangle \to \langle y \leftarrow x, m[x \mapsto \widehat{3}]\rangle \to \langle \text{PASS}, m[x \mapsto \widehat{3}][y \mapsto \widehat{3}]\rangle$.

  Dans le cas général, utiliser des instructions pour représenter l'état des calculs ne suffit pas ; il faut utiliser une continuation. C'est ce qui est fait par exemple dans la sémantique de CMinor [AB07]. Ici, le flot de contrôle est plus simple et on peut se contenter de retenir une simple instruction, ce qui simplifie la présentation.
- un couple $\langle l\,v, m\rangle$ où $l\,v$ est une valeur gauche et m un état mémoire. L'évaluation des valeurs gauches est décrite en section 4.9.
- une erreur Ω. La propagation des erreurs est détaillée dans la section 4.12.

L'évaluation des expressions, valeurs gauches et instructions se fait à petits pas. C'est-à-dire qu'on simplifie d'étape en étape leur forme, jusqu'à arriver à un cas de base :

- pour les expressions, une valeur v ;
- pour les instructions, l'instruction PASS ou RETURN(v) où v est une valeur ;
- pour les valeurs gauches un chemin φ.

On considère en fait la clôture transitive de cette relation. Cela revient à ajouter une règle :

$$\frac{\Xi_1 \to \Xi_2 \qquad \Xi_2 \to \Xi_3}{\Xi_1 \to \Xi_3}\ (\text{TRANS})$$

4.5 Opérations sur les valeurs

Un certain nombre d'opérations est possible sur les valeurs (figure 4.6) :

- les opérations arithmétiques $+$, $-$, $\times$, / et % sur les entiers. L'opérateur % correspond au modulo (reste de la division euclidienne). En cas de division par zéro, l'erreur Ω_{div} est levée.
- les versions « pointées » $+.$, $-.$, $\times.$ et $/.$ sur les flottants.
- les opérations d'arithmétique de pointeur $+_p$ et $-_p$ qui à un chemin mémoire et un entier associent un chemin mémoire.
- les opérations d'égalité $=$ et $\neq$. L'égalité entre entiers ou entre flottants est immédiate. Deux valeurs composées (tableaux ou structures) sont égales si elles ont la même forme (même taille pour les tableaux, ou même champs pour les structures) et que toutes leurs sous-valeurs sont égales deux à deux. Deux références mémoire sont égales lorsque les chemins qu'elles décrivent sont syntaxiquement égaux.
- les opérations de comparaison $\leq, \geq, <, >$ sont définies avec leur sémantique habituelle sur les entiers et les flottants. Sur les références mémoires, elles sont définies dans le cas où les deux opérandes sont de la forme $\varphi[\cdot]$ par : $\varphi[n] \boxplus \varphi[m] \stackrel{\text{def}}{=\!=} n \boxplus m$. Dans les autres cas, l'erreur Ω_{ptr} est renvoyée. Notamment, il n'est pas possible de comparer deux fonctions, deux tableaux ou deux structures.
- les opérateurs bit à bit sont définis sur les entiers. &, | et ^ représentent respectivement la conjonction, la disjonction et la disjonction exclusive (XOR).
- des versions logiques de la conjonction (&&) et de la disjonction (||) sont également présentes. Leur sémantique est donnée par le tableau suivant :

n	m	n && m	n || m
0	0	0	0
0	$\neq 0$	0	1
$\neq 0$	0	0	1
$\neq 0$	$\neq 0$	1	1

- des opérateurs de décalage à gauche ($\ll$) et à droite ($\gg$) sont présents. Eux aussi ne s'appliquent qu'aux entiers.
- les opérateurs arithmétiques unaires $+$, $-$, $+.$ et $-.$ sont équivalents par définition à l'opération binaire correspondante. Par exemple $-4 \stackrel{\text{def}}{=\!=} 0-4$.
- $\sim$ inverse tous les bits de son opérande. ! est une version logique, c'est-à-dire que $!0 = 1$ et si $n \neq 0$, $!n = 0$.

Si ces opérateurs sémantiques reçoivent des données incompatibles (par exemple si on tente d'ajouter une fonction et un entier), l'erreur spéciale Ω_{typ} est renvoyée.

4.6 Opérations sur les états mémoire

Définition 4.4 (Recherche de variable)**.** *La recherche de variable permet d'associer à une variable x une adresse a.*

Chaque fonction peut accéder aux variables locales de la fonction en cours, ainsi qu'aux variables globales.

Remarque : le cadre de variables locales le plus récent a toujours l'indice 1.

$$\text{Lookup}((s,g),x) = \begin{cases} (|s|,x) & si\ |s| > 0\ et\ (x \mapsto v) \in s_1 \\ x & si\ (x \mapsto v) \in g \\ \Omega_{var} & sinon \end{cases}$$

En entrant dans une fonction, on rajoutera un cadre de pile qui contient les paramètres de la fonction ainsi que ses variables locales. En retournant à l'appelant, il faudra supprimer ce cadre de pile.

Définition 4.5 (Manipulations de pile)**.** *On définit l'empilement d'un cadre de pile $c = ((x_1 \mapsto v_1), \ldots, (x_n \mapsto v_n))$ sur un état mémoire $m = (s,g)$ (figure 4.9(a)) :*

$$\text{Push}((s,g),c) = (c :: s, g)$$

On définit aussi l'extension du dernier cadre de pile, qui sert aux déclarations de variables locales (figure 4.9(b)) :

$$\text{Extend}((c :: s, g), x \mapsto v) = ((x \mapsto v :: c :: s), g)$$

L'opération inverse de Extend$(\cdot, \cdot \mapsto \cdot)$ *sera simplement notée « − » : $m - x$, par exemple.*
De même on définit le dépilement (figure 4.9(c)) :

$$\text{Pop}((c :: s, g)) = (s, g)$$

Définition 4.6 (Hauteur d'une valeur)**.** *Une valeur peut contenir une référence vers une variable de la pile. La hauteur d'une valeur est l'indice du plus haut cadre qu'elle référence, ou* -1 *sinon.*

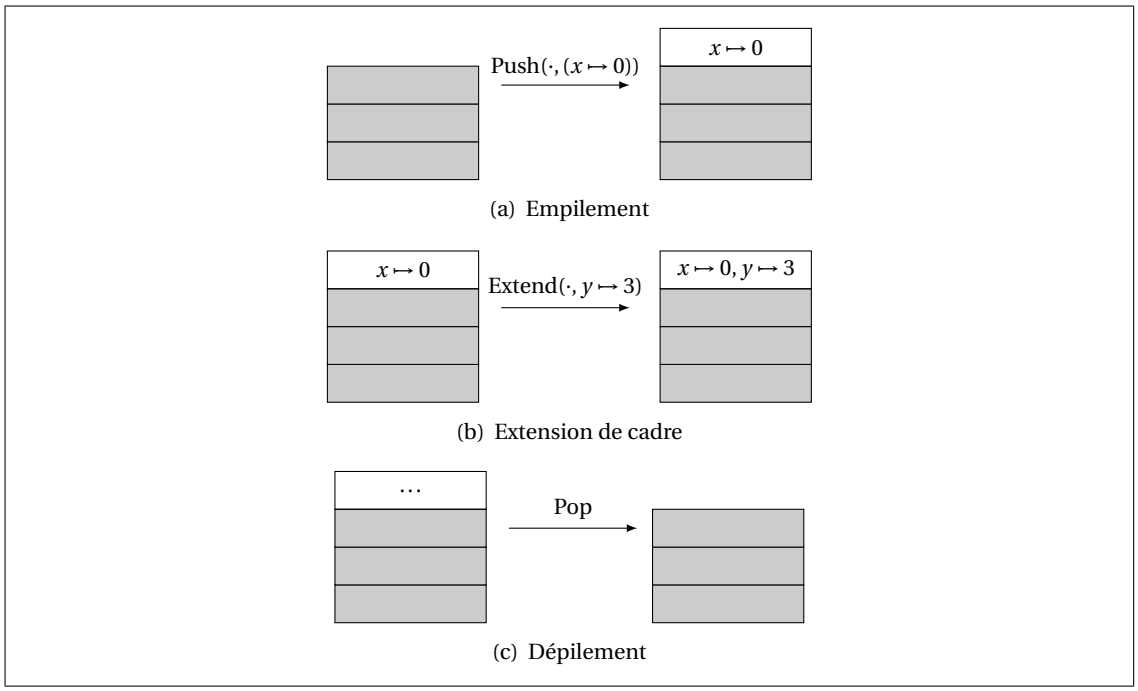


FIGURE 4.9: Opérations de pile

$$
\begin{aligned}
\mathscr{H}(\widehat{c}) &= -1 \\
\mathscr{H}(\widehat{f}) &= -1 \\
\mathscr{H}(\widehat{\&}\,\varphi) &= \mathscr{H}_\Phi(\varphi) \\
\mathscr{H}(\{l_1 : \widehat{v_1;\ldots;l_n} : v_n\}) &= \max_{i\in[1;n]} \mathscr{H}(v_i) \\
\mathscr{H}(\widehat{[v_1,\ldots,v_n]}) &= \max_{i\in[1;n]} \mathscr{H}(v_i)
\end{aligned}
\qquad \textit{où :} \qquad
\begin{aligned}
\mathscr{H}_\Phi((x)) &= -1 \\
\mathscr{H}_\Phi((n,x)) &= n \\
\mathscr{H}_\Phi(\varphi\widehat{.l}) &= \mathscr{H}_\Phi(\varphi) \\
\mathscr{H}_\Phi(\varphi\widehat{[n]}) &= \mathscr{H}_\Phi(\varphi)
\end{aligned}
$$

Les opérations Extend et Pop ne sont définies que pour une pile non vide. Néanmoins cela ne pose pas de problème, puisque, lors de l'exécution, la pile n'est vide que lors de l'évaluation d'expressions dans les phrases de programme. À cet endroit, seules des expressions peuvent apparaître, et leur évaluation ne manipule jamais la pile avec ces opérations.

On définit aussi une opération de nettoyage de pile, qui sera utile pour les retours de fonction.

En effet, si une référence au dernier cadre est toujours présente après le retour d'une fonction, cela peut casser le typage.

Par exemple, dans la figure 4.10, l'exécution de $h(\,)$ donne à p la valeur $(1,x)$. Puis en arrivant dans g, le déréférencement de p va modifier x qui va avoir la valeur 1. x, variable flottante, contient donc un entier. Dans la ligne marquée `(*)`, on réalise donc l'addition d'un entier (contenu dans x malgré le type de la variable) et d'un flottant. Cette opération est bien typée dans le programme mais provoquera une erreur Ω_{typ} à l'exécution.

Pour empêcher cela, on instrumente donc le retour de la fonction `f` pour que `p` soit remplacé par NULL. Alors dans `h`, le déréférencement provoquera une erreur et empêchera la violation du typage.

Pour définir l'opération de nettoyage, on commence par définir une opération de nettoyage selon un prédicat sur les chemins :

```
f = Fun () {
    Decl x = 0 in
    return (&x);
}

g = Fun (p) {
    Decl x = 0.0 in
    *p = 1;
    x <- x + 2.0; // (*)
}

h = Fun () {
    Decl p = f() in
    g(p);
}
```

FIGURE 4.10: Cassage du typage par un pointeur fou

$$\begin{aligned}
\text{CleanVP}(p, \widehat{c}) &= \widehat{c} \\
\text{CleanVP}(p, \widehat{f}) &= \widehat{f} \\
\text{CleanVP}(p, \widehat{\&}\,\varphi) &= \begin{cases} \text{NULL} & \text{si } p(\varphi) \\ \widehat{\&}\,\varphi & \text{sinon} \end{cases} \\
\text{CleanVP}(p, \{l_1 : \widehat{v_1}, \ldots, l_n : v_n\}) &= \{l_1 : \text{CleanVP}(p, \widehat{v_1}), \ldots, l_n : \text{CleanVP}(p, v_n)\} \\
\text{CleanVP}(p, [\widehat{v_1}, \ldots, \widehat{v_n}]) &= [\text{CleanVP}(p, \widehat{v_1}), \ldots, \widehat{\text{C}}\text{leanVP}(p, v_n)]
\end{aligned}$$

On l'étend ensuite aux cadres de pile, puis aux états mémoire :

$$\text{CleanLP}(p, (x_1 \mapsto v_1, \ldots, x_n \mapsto v_n)) = (x_1 \mapsto \text{CleanVP}(p, v_1), \ldots, x_n \mapsto \text{CleanVP}(p, v_n))$$

$$\begin{aligned}
\text{CleanP}(p, (s, g)) &= (s', g') \\
\text{où } s' &= (\text{CleanLP}(p, s_1), \ldots, \text{CleanLP}(p, s_{|s|})) \\
g' &= \text{CleanLP}(p, g)
\end{aligned}$$

À l'aide de ces fonctions, on définit quatre opérations, permettant de nettoyer des états mémoire ou des valeurs en enlevant tout un niveau de pile ou seulement une variable :

$$\text{Cleanup}(m) = \text{CleanP}(\lambda\varphi.\mathscr{H}_\Phi(\varphi) > |m|, m) \qquad \text{CleanVar}(m, a) = \text{CleanP}(\lambda\varphi.\varphi = a, m)$$

$$\text{CleanV}_p(v) = \text{CleanVP}(\lambda\varphi.\mathscr{H}_\Phi(\varphi) > |m|, v) \qquad \text{CleanVarV}(v, a) = \text{CleanVP}(\lambda\varphi.\varphi = a, v)$$

Ces 4 fonctions seront utilisées dans plusieurs règles dans la suite de ce chapitre.

Remarques Ces opérations ne sont pas toujours bien définies. Par exemple, $\text{Extend}(\cdot, \cdot \mapsto \cdot)$ ne peut pas s'appliquer à une pile vide, et $m - x$ n'est défini que si une variable x existe au sommet de la pile de m. Ce caractère partiel ne pose pas de problème de par la structure

des règles qui vont utiliser ces constructions. Par exemple, à chaque empilement correspond exactement un dépilement. De plus, les phrases d'un programme ne peuvent pas faire intervenir de déclaration de variable (une instruction est forcément dans une fonction), donc Extend$(\cdot, \cdot \mapsto \cdot)$ réussit toujours.

Un autre problème se pose si deux variables ont le même nom dans un cadre. Elles ne peuvent pas être distinguées. On interdit donc ce cas en demandant aux programmes d'être bien formés : au sein d'une fonction, les paramètres ainsi que l'ensemble des locales déclarées doivent être de noms différents. En pratique, une phase préalable d'α-conversion peut renommer les variables problématiques.

De plus, le fait d'ajouter cette étape de nettoyage à chaque retour de fonction peut être assez coûteux. C'est un compromis : si on considère que les programmes se comportent bien et ne créent pas de pointeurs fous (pointant au-dessus de la pile), alors cette phase est inutile et peut être remplacée par l'identité. Autrement dit, il s'agit seulement d'une technique pour s'assurer de ne pas avoir d'erreurs dans la sémantique. L'ajout d'un ramasse-miette, ou une vérification préalable par un système de régions [TJ92] peut garantir qu'il n'y a pas de telles constructions dangereuses.

4.7 Accesseurs

Le but de cette section est de définir rigoureusement les accès à la mémoire. À partir d'un état mémoire m et d'une valeur gauche φ, on veut pouvoir obtenir :

- la valeur accessible au chemin φ : $m[\varphi]_\Phi$
- l'état mémoire obtenu en remplaçant celle-ci par une nouvelle valeur v' : $m[\varphi \leftarrow v']_\Phi$

Pour définir cette lentille indexée Φ, on commence par définir des lentilles élémentaires, et on les compose pour pouvoir définir des lentilles entre valeurs.

On commence par définir deux lentilles I et L pour accéder aux structures de listes. I accède par indice et L par clef (dans une liste d'association, donc).

Cela permet ensuite de définir A, qui extrait une valeur à partir d'un nom de variable et d'une éventuelle hauteur de pile. Pour cela, on compose les lentilles I et L.

Les autres travaillent sur des valeurs composées, c'est-à-dire sur les structures et tableaux. La lentille F extrait une sous-valeur correspondant au champ d'une structure. Le fonctionnement est similaire à la lentille L puisqu'on accède par nom à une sous-structure. La lentille T, quant à elle, permet d'accéder au n^{e} élement d'un tableau. De ce point de vue, elle est similaire à I mais en travaillant sur les valeurs.

Enfin, on définit Φ pour accéder à n'importe quelle sous-valeur d'une variable dans la mémoire. Cela utilise A, F et T précédemment définis.

La figure 4.11 résume ces dépendances. Les lignes pleines indiquent quelles sont les définitions utilisées, et les pointillés relient les lentilles similaires. À droite, on donne un exemple des lentilles de base. La valeur entourée correspond au « curseur » de la lentille, c'est-à-dire la valeur qui peut être renvoyée ou mise à jour.

Accès à une liste par indice : I

On définit une lentille indexée $I : \mathbb{N} \to \text{LENS}_{\alpha^*,\alpha}$ permettant d'accéder aux éléments d'une liste par leur indice. On rappelle que les listes sont des suites finies, définies page 36. En outre, I n'est définie que pour $n \in [1; |l|]$.

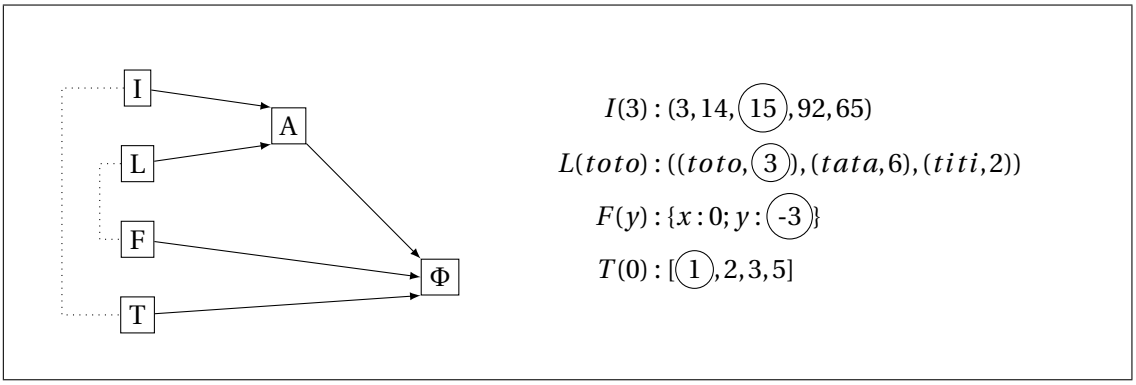


FIGURE 4.11: Dépendances entre les lentilles

$$
\begin{aligned}
l[n]_I &= l_n \text{ si } n \in [1;|l|] \\
l[n \leftarrow x]_I &= l' \\
&\text{où } l'_n = x \\
&\quad \forall i \neq n, l'_i = l_i
\end{aligned}
$$

Accès à une liste d'associations : L

Une liste d'association est une liste de paires (clef, valeur) avec l'invariant supplémentaire que les clefs sont uniques. Il est donc possible de trouver au plus une valeur associée à une clef donnée. L'écriture est également possible, en remplaçant un couple par un couple avec une valeur différente.

$$
\begin{aligned}
l[x]_L &= \begin{cases} v & \text{si } \exists! n \in [1;|l|], \exists v, l_n = (x \mapsto v) \\ \Omega_{var} & \text{sinon} \end{cases} \\
l[x \leftarrow v]_L &= \begin{cases} l[n \leftarrow (x \mapsto v')]_I & \text{si } \exists! n \in [1;|l|], \exists v, l_n = (x \mapsto v) \\ \Omega_{var} & \text{sinon} \end{cases}
\end{aligned}
$$

Accès par adresse : A

Les états mémoire sont constitués des listes d'association (nom, valeur).

L'accesseur par adresse $[\cdot]_A$ permet de généraliser l'accès à ces valeurs en utilisant comme clef non pas un nom mais une adresse.

Selon cette adresse, on accède soit à la liste des globales, soit à une des listes de la pile des locales.

On pose $m = (s, g)$.

Les accès aux globales se font de la manière suivante. Si la variable n'existe pas, notons que L retourne Ω_{var}.

$$A((x)) = \text{Snd} \ggg L(x)$$

Snd désigne la lentille entre un couple et sa deuxième composante. Ainsi, par exemple $m[(x) \leftarrow v]_A = (s, g[x \leftarrow v]_L)$.

Les accès aux locales reviennent à accéder à la bonne variable du bon cadre de pile. Cela revient naturellement à composer les lentilles L et I. On définit donc une lentille $\mathscr{L}_{n,x} = I(|s|-n+1) \ggg L(x)$ qui accède à la variable x du n^{e} cadre de pile.

$$m[(n,x)]_A = \begin{cases} \text{get}_{\mathscr{L}_{n,x}}(s) & \text{si } n \in [1;|s|] \\ \Omega_{var} & \text{sinon} \end{cases}$$

$$m[(n,x) \leftarrow v]_A = \begin{cases} (\text{put}_{\mathscr{L}_{n,x}}(v,s), g) & \text{si } n \in [1;|s|] \\ \Omega_{var} & \text{sinon} \end{cases}$$

Les numéros de cadre qui permettent d'identifier les locales (le n dans (n,x)) croissent avec la pile. D'autre part, l'empilement se fait en tête de liste (près de l'indice 1). Donc pour accéder aux plus vieilles locales (numérotées 1), il faut accéder au dernier élément de la liste. Ceci explique pourquoi un indice $|s|-n+1$ apparaît dans la définition précédente.

Accès par champ : F

Les valeurs qui sont des structures possèdent des sous-valeurs, associées à des noms de champ.

L'accesseur $[\cdot]_F$ permet de lire et de modifier un champ de ces valeurs.

L'erreur Ω_{field} est levée si on accède à un champ non existant.

$$\begin{aligned} \{l_1 : v_1; \ldots; l_n : v_n\}[l]_F &= v_i \text{ si } \exists i \in [1;n], l = l_i \\ \{l_1 : v_1; \ldots; l_n : v_n\}[l]_F &= \Omega_{field} \text{ sinon} \\ \{l_1 : v_1; \ldots; l_n : v_n\}[l \leftarrow v]_F &= \{l_1 : v_1 \\ &\quad ; \ldots \\ &\quad ; l_{p-1} : v_{p-1} \\ &\quad ; l_p : v \\ &\quad ; l_{p+1} : v_{p+1} \\ &\quad ; \ldots \\ &\quad ; l_n : v_n\} \text{ si } \exists p \in [1;n], l = l_p \\ \{l_1 : v_1; \ldots; l_n : v_n\}[l \leftarrow v]_F &= \Omega_{field} \text{ sinon} \end{aligned}$$

Accès par indice de tableau : T

On définit de même un accesseur $[\cdot]_T$ pour les accès par indice à des valeurs tableaux. Néanmoins le paramètre indice est toujours un entier et pas une expression arbitraire. Notons que les accès sont vérifiés dynamiquement : il ne peut pas y avoir de débordement de tableau.

$$
\begin{aligned}
[v_1;\ldots;v_n][i]_T &= v_{i+1} \text{ si } i \in [0;n-1] \\
[v_1;\ldots;v_n][i]_T &= \Omega_{array} \text{ sinon} \\
[v_1;\ldots;v_n][i \leftarrow v]_T &= [v'_1;\ldots;v'_n] \text{ si } i \in [0;n-1] \\
&\text{où } \begin{cases} v'_i = v \\ \forall\, j \neq i, v'_j = v_j \end{cases} \\
[v_1;\ldots;v_n][i \leftarrow v]_T &= \Omega_{array} \text{ sinon}
\end{aligned}
$$

Accès par chemin : Φ

L'accès par chemin Φ permet de lire et de modifier la mémoire en profondeur.

On peut accéder directement à une variable, et les accès à des sous-valeurs se font en composant les accesseurs (définition 4.3, page 39) :

$$
\begin{aligned}
\Phi(a) &= A(a) \\
\Phi(\varphi.l) &= \Phi(\varphi) \ggg F(l) \\
\Phi(\varphi[i]) &= \Phi(\varphi) \ggg T(i)
\end{aligned}
$$

Remarque Dans toute la suite, lorsque ce n'est pas ambigü, on emploiera la notation $m[\varphi]$ pour désigner $m[\varphi]_\Phi$. Il est important de remarquer que m désigne un état particulier et φ un chemin particulier, mais que Φ est la lentille indexée globale définie page 52.

4.8 Contextes d'évaluation

L'évaluation des expressions repose sur la notion de contextes d'évaluation. L'idée est que, si on peut évaluer une expression, alors on peut évaluer une expression qui contient celle-ci.

Par exemple, supposons que $\langle f(3), m\rangle \rightarrow \langle 2, m\rangle$. Alors on peut ajouter la constante 1 à gauche de chaque expression sans changer le résultat : $\langle 1 + f(3), m\rangle \rightarrow \langle 1 + 2, m\rangle$. On a utilisé le même contexte $C = 1 + \bullet$.

Pour pouvoir raisonner en termes de contextes, 3 points sont nécessaires :

- comment découper une expression selon un contexte ;
- comment appliquer une règle d'évaluation sous un contexte ;
- comment regrouper une expression et un contexte.

Le premier point consiste à définir les contextes eux-mêmes (figure 4.12).

Dans cette définition, chaque cas hormis le cas de base fait apparaître exactement un « C ». Chaque contexte est donc constitué d'exactement une occurrence de $\bullet$ (une dérivation de C est toujours linéaire). L'opération de substitution consiste à remplacer ce trou : $C(\!|X|\!)$ est l'objet syntaxique (instruction, expression ou valeur gauche) obtenu en remplaçant l'unique $\bullet$ dans C par X. Par exemple, DECL $x = 2 + \bullet$ IN{PASS}$(\!|5|\!)$ est DECL $x = 2 + 5$ IN{PASS}

À titre d'illustration, décomposons l'évaluation de $e_1 \boxplus e_2$ en $v = v_1 \,\widehat{\boxplus}\, v_2$ depuis un état mémoire m :

Contextes

$$
\begin{aligned}
C ::= {} & \bullet \\
| \;\; & C \boxplus e \\
| \;\; & v \boxplus C \\
| \;\; & \boxminus C \\
| \;\; & \& C \\
| \;\; & C \leftarrow e \\
| \;\; & \varphi \leftarrow C \\
| \;\; & \{l_1 : v_1; \ldots; l_i : C; \ldots; l_n : e_n\} \\
| \;\; & [v_1; \ldots; C; \ldots; e_n] \\
| \;\; & C(e_1, \ldots, e_n) \\
| \;\; & f(v_1, \ldots, C, \ldots, e_n) \\
| \;\; & C.l_S \\
| \;\; & C[e] \\
| \;\; & \varphi[C] \\
| \;\; & * \; C \\
| \;\; & C; i \\
| \;\; & \textsc{If}(C)\{i_1\}\textsc{Else}\{i_2\} \\
| \;\; & \textsc{Return}(C) \\
| \;\; & \textsc{Decl}\; x = C \;\textsc{in}\{i\}
\end{aligned}
$$

FIGURE 4.12: Contextes d'évaluation

1. on commence par évaluer l'expression e_1 en une valeur v_1. Le nouvel état mémoire est noté m'. Soit donc $\langle e_1, m\rangle \to \langle v_1, m'\rangle$.
2. En appliquant la règle CTX (définie page suivante) avec $C = \bullet \boxplus e_2$ (qui est une des formes possibles pour un contexte d'évaluation), on déduit de 1. que $\langle e_1 \boxplus e_2, m\rangle \to \langle v_1 \boxplus e_2, m'\rangle$
3. D'autre part, on évalue e_2 depuis m'. En supposant encore que l'évaluation converge, notons v_2 la valeur calculée et m'' l'état mémoire résultant : $\langle e_2, m'\rangle \to \langle v_2, m''\rangle$.
4. Appliquons la règle CTX à 3. avec $C = v_1 \boxplus \bullet$. On obtient $\langle v_1 \boxplus e_2, m\rangle \to \langle v_1 \boxplus v_2, m'\rangle$.
5. En combinant les résultats de 2. et 4. on en déduit que $\langle e_1 \boxplus e_2, m\rangle \to \langle v_1 \boxplus v_2, m''\rangle$.
6. D'après la règle EXP-BINOP (page 55), $\langle v_1 \boxplus v_2, m''\rangle \to \langle v_1 \mathbin{\widehat{\boxplus}} v_2, m''\rangle$
7. D'après 5. et 6., on a par combinaison $\langle e_1 \boxplus e_2, m\rangle \to \langle v, m''\rangle$ en posant $v = v_1 \mathbin{\widehat{\boxplus}} v_2$.

Le deuxième point sera résolu par la règle d'inférence suivante.

$$\frac{\langle i, m\rangle \to \langle i', m'\rangle}{\langle C(\!|i|\!), m\rangle \to \langle C(\!|i'|\!), m'\rangle}\ (\textsc{Ctx})$$

Enfin, le troisième revient à définir l'opérateur de substitution $\cdot(\!|\cdot|\!)$ présent dans la règle précédente. Notons que puisque $i ::= e$ et $e ::= l\,v$, on peut aussi l'appliquer aux expressions et aux valeurs gauches : l'opération $\cdot(\!|\cdot|\!)$ est purement syntaxique.

4.9 Valeurs gauches

Obtenir un chemin à partir d'un nom de variable revient à résoudre le nom de cette variable : est-elle accessible ? Le nom désigne-t-il une variable locale ou une variable globale ?

$$\frac{a = \text{Lookup}(x, m)}{\langle x, m\rangle \to \langle a, m\rangle} \text{ (PHI-VAR)}$$

Les règles portant sur le déréférencement et l'accès à un champ de structure sont similaires : on commence par évaluer la valeur gauche sur laquelle porte ce modificateur, et on place le même modificateur sur le chemin résultant. Dans le cas des champs de structure, l'annotation de structure S n'est pas prise en compte pour l'évaluation : elle servira uniquement au typage.

$$\frac{}{\langle \varphi.l_S, m\rangle \to \langle \widehat{\varphi.l}, m\rangle} \text{ (PHI-STRUCT)}$$

Enfin, pour évaluer un chemin dans un tableau, on commence par procéder comme précédemment, c'est-à-dire en évaluant la valeur gauche sur laquelle porte l'opération d'indexation. Puis on évalue l'expression d'indice en une valeur qui permet de construire le chemin résultant.

$$\frac{}{\langle \varphi[n], m\rangle \to \langle \widehat{\varphi[n]}, m\rangle} \text{ (PHI-ARRAY)}$$

Notons qu'en procédant ainsi, on évalue les valeurs gauches en allant de gauche à droite : dans l'expression $x[e_1][e_2][e_3]$, e_1 est évalué en premier, puis e_2, puis e_3.

La règle portant sur le déréférencement est particulière. On peut penser que la bonne définition de φ consiste à se calquer sur la définition de lv, en remplaçant les noms de variable par leur adresse résolue et en évaluant les indices de tableau, et à ajouter une règle qui transforme $*\varphi$ en $\widehat{*}\varphi$.

Cela ne fonctionne pas, car alors les déréférencements sont évalués trop tard : au moment de l'affectation dans la valeur gauche plutôt qu'à sa définition. La figure 4.13 illustre ce problème.

```
Decl s0 = { .f : 0 } in
Decl s1 = { .f : 1 } in
Decl x  = & s0 in
Decl p = & ((*x).f) in
/* (a) */
x <- & s1
/* (b) */
```

FIGURE 4.13: Évaluation stricte ou paresseuse des valeurs gauches

On s'intéresse à l'évaluation de l'expression `*p` aux points `(a)` et `(b)`. Avec une sémantique paresseuse (en ajoutant un $\widehat{*}\varphi$), la valeur de `p` est $\widehat{\&}\ ((*(1,x)).f)$, donc `*p` est évalué à 0 en `(a)` et 1 en `(b)`. Au contraire, avec une sémantique stricte (correcte), `p` vaut $\widehat{\&}\ (((1,s0).f)$ et donc `*p` est évalué à 0 en `(a)` et en `(b)`.

Dans le cas où la valeur référencée n'a pas la forme $\&\varphi$ ou $\widehat{\text{NULL}}$, aucune règle ne peut s'appliquer (comme lorsqu'on cherche à réduire l'addition d'une fonction et d'un entier, par

exemple). Cela est préférable à renvoyer Ω_{ptr} car on montrera que ce cas est toujours évité dans les programmes typés (théorème 5.1).

$$\frac{v = \widehat{\&}\,\varphi}{\langle *\ v, m\rangle \to \langle \varphi, m\rangle}\ (\textsc{Phi-Deref}) \qquad \frac{v = \widehat{\textsc{Null}}}{\langle *\ v, m\rangle \to \Omega_{ptr}}\ (\textsc{Phi-Deref-Null})$$

Par exemple, $l\,v = x.l_S[2 * n].g_T$ pourra s'évaluer en $\varphi = (2, x).l[4].g$.

4.10 Expressions

Évaluer une constante est le cas le plus simple, puisqu'en quelque sorte celle-ci est déjà évaluée. À chaque constante syntaxique c, on peut associer une valeur sémantique $\widehat{c}$. Par exemple, au chiffre (symbole) 3, on associe le nombre (entier) $\widehat{3}$.

$$\frac{}{\langle c, m\rangle \to \langle \widehat{c}, m\rangle}\ (\textsc{Exp-Cst})$$

De même, une fonction est déjà évaluée :

$$\frac{}{\langle f, m\rangle \to \langle \widehat{f}, m\rangle}\ (\textsc{Exp-Fun})$$

Pour lire le contenu d'un emplacement mémoire (valeur gauche), il faut tout d'abord l'évaluer en un chemin.

$$\frac{}{\langle \varphi, m\rangle \to \langle m[\varphi]_\Phi, m\rangle}\ (\textsc{Exp-Lv})$$

Pour évaluer une expression constituée d'un opérateur, on évalue une sous-expression, puis l'autre (l'ordre d'évaluation, est encore imposé : de gauche à droite). À chaque opérateur $\boxplus$, correspond un opérateur sémantique $\widehat{\boxplus}$ qui agit sur les valeurs. Par exemple, l'opérateur $\widehat{+}$ est l'addition entre entiers machine (page 42). Comme précisé dans la section 4.5, la division par zéro via /, % ou /. provoque l'erreur Ω_{div}.

$$\frac{}{\langle \boxminus\ v, m\rangle \to \langle \widehat{\boxminus}\ v, m\rangle}\ (\textsc{Exp-UnOp}) \qquad \frac{}{\langle v_1 \boxplus v_2, m\rangle \to \langle v_1 \mathbin{\widehat{\boxplus}} v_2, m\rangle}\ (\textsc{Exp-BinOp})$$

Il est nécessaire de dire un mot sur les opérations $\widehat{+_p}$ et $\widehat{-_p}$ définissant l'arithmétique des pointeurs. Celles-ci sont uniquement définies pour les références mémoire à un tableau, c'est-à-dire celles qui ont la forme $\widehat{\&}\,\varphi[n]$. On a alors :

$$\widehat{\&}\,\varphi[n] \mathbin{\widehat{+}_p} i = \widehat{\&}\,\varphi[n \mathbin{\widehat{+}} i]$$
$$\widehat{\&}\,\varphi[n]\ ] \mathbin{\widehat{-}_p} i = \widehat{\&}\,\varphi[n \mathbin{\widehat{-}} i]$$

Cela implique qu'on ne peut pas faire faire d'arithmétique de pointeurs au sein d'une même structure. Autrement c'est une erreur de manipulation de pointeurs [3] et l'opérateur $\widehat{\boxplus}$ renvoie Ω_{ptr}.

3. Cela est cohérent avec la norme C99 : « If the pointer operand points to an element of an array object, and the array is large enough, [...] ; otherwise, the behavior is undefined. » [ISO99, 6.5.6 §8]

Si l'indice calculé ($n \mathbin{\widehat{+}} i$ ou $n \mathbin{\widehat{-}} i$) sort de l'espace alloué, alors l'erreur sera faite au moment de l'accès : la lentille T renverra Ω_{array} (page 51).

Une left-value s'évalue en le chemin correspondant.

$$\frac{}{\langle \& \varphi, m\rangle \to \langle \widehat{\&} \varphi, m\rangle} \text{(EXP-ADDR)}$$

L'affectation se déroule en 3 étapes. D'abord, l'expression est évaluée en une valeur v. Ensuite, la valeur gauche est évaluée en un chemin φ. Enfin, un nouvel état mémoire est construit, où la valeur accessible par φ est remplacée par v. Comme dans le langage C, l'expression d'affectation produit une valeur, qui est celle qui a été affectée.

$$\frac{}{\langle \varphi \leftarrow v, m\rangle \to \langle v, m[\varphi \leftarrow v]_\Phi\rangle} \text{(EXP-SET)}$$

Expressions composées

Les littéraux de structures sont évalués en leurs constructions syntaxiques respectives. Puisque les contextes d'évaluation sont de la forme $[v_1;\ldots;C;\ldots;e_n]$, l'évaluation se fait toujours de gauche à droite.

$$\frac{}{\langle \{l_1 : v_1;\ldots;l_n : v_n\}, m\rangle \to \langle \{l_1 : \widehat{v_1;\ldots;l_n} : v_n\}, m\rangle} \text{(EXP-STRUCT)}$$

$$\frac{}{\langle [v_1,\ldots,v_n], m\rangle \to \langle [\widehat{v_1,\ldots,v_n}], m\rangle} \text{(EXP-ARRAY)}$$

L'appel de fonction est traité de la manière suivante. On ne peut pas facilement relier un pas d'évaluation de i à un pas d'évaluation de $\text{fun}(a)\{i\}(v_1,\ldots,v_n)$, et donc un contexte $C ::= \text{fun}(a)\{\bullet\}(v_1,\ldots,v_n)$ n'est pas à considérer. En effet, l'empilement suivi du dépilement modifie la mémoire.

On emploie donc une règle EXP-CALL-CTX qui relie un pas interne $\langle i, m_1\rangle \to \langle i', m_2\rangle$ à un pas externe. Une fois l'instruction interne réduite d'un pas, on évalue les arguments en des valeurs v'_i. Ils correspondent aux nouvelles valeurs à passer à la fonction.

Les autres fonctions permettent de transférer le flot de contrôle : en retournant la même instruction pour une instruction terminale, ou en propageant une erreur. Dans le cas où on retourne de la fonction par $i = \text{RETURN}(v)$, il faut alors supprimer les références aux variables qui ont disparu grâce aux opérateurs Cleanup$(\cdot)$ et CleanV.$(\cdot)$.

On suppose deux choses sur chaque fonction : d'une part, les noms de ses arguments sont deux à deux différents et, d'autre part, son corps se termine par une instruction RETURN$(\cdot)$. Cela veut dire que la dernière instruction doit être soit de cette forme, soit par exemple une alternative dans laquelle les deux branches se terminent par un RETURN$(\cdot)$. C'est une propriété qui peut être détectée statiquement avant l'exécution. Néanmoins, dans la syntaxe concrète, on peut supposer qu'un RETURN$(())$ est inséré automatiquement en fin de fonction lorsqu'aucun RETURN$(\cdot)$ n'est présent dans son corps.

$$\frac{\begin{array}{c} m_1 = \text{Push}(m_0, ((a_1 \mapsto v_1), \ldots, (a_n \mapsto v_n))) \\ \langle i, m_1\rangle \to \langle i', m_2\rangle \qquad \forall\, i \in [1;n], v'_i = m_2[(|m_2|, a_i)]_A \qquad m_3 = \text{Pop}(m_2) \end{array}}{\langle \text{fun}(a_1,\ldots,a_n)\{i\}(v_1,\ldots,v_n), m_0\rangle \to \langle \text{fun}(a_1,\ldots,a_n)\{i'\}(v'_1,\ldots,v'_n), m_3\rangle}\ (\text{EXP-CALL-CTX})$$

$$\frac{m' = \text{Push}(m, ((a_1 \mapsto v_1), \ldots, (a_n \mapsto v_n))) \qquad \langle i, m'\rangle \to \Omega}{\langle \text{fun}(a_1,\ldots,a_n)\{i\}(v_1,\ldots,v_n), m\rangle \to \Omega}\ (\text{EXP-CALL-ERR})$$

$$\frac{m' = \text{Cleanup}(m) \qquad v' = \text{CleanV}_{|m|}(v)}{\langle \text{fun}(a_1,\ldots,a_n)\{\text{RETURN}(v)\}(v_1,\ldots,v_n), m\rangle \to \langle v', m'\rangle}\ (\text{EXP-CALL-RETURN})$$

4.11 Instructions

Les cas de la séquence et de l'évaluation d'une expression sont sans surprise.

$$\frac{}{\langle (\text{PASS}; i), m\rangle \to \langle i, m\rangle}\ (\text{SEQ}) \qquad \frac{}{\langle v, m\rangle \to \langle \text{PASS}, m\rangle}\ (\text{EXP})$$

L'évaluation de DECL $x = v$ IN$\{i\}$ sous m se fait de la manière suivante, similaire à l'appel de fonction. La règle principale est DECL-CTX qui relie un pas d'évaluation sous une déclaration à un pas d'évaluation externe : pour ce faire, on étend l'état mémoire en ajoutant x, on effectue le pas, puis on enlève x. L'instruction résultante est la déclaration de x avec la nouvelle valeur v' de x après le pas d'exécution [4].

On suppose qu'il n'y a pas de masquage au sein d'une fonction, c'est-à-dire que le nom d'une variable déclarée n'est jamais dans l'environnement avant cette déclaration.

Si i est terminale (PASS ou RETURN(v)), alors on peut s'évaluer en i en nettoyant l'espace mémoire des références à x qui peuvent subsister.

Enfin, si une erreur se produit elle est propagée.

$$\frac{m' = \text{CleanVar}(m - x, (|m|, x))}{\langle \text{DECL } x = v \text{ IN}\{\text{PASS}\}, m\rangle \to \langle \text{PASS}, m'\rangle}\ (\text{DECL-PASS})$$

$$\frac{m' = \text{CleanVar}(m - x, (|m|, x)) \qquad v'' = \text{CleanVarV}(v', (|m|, x))}{\langle \text{DECL } x = v \text{ IN}\{\text{RETURN}(v')\}, m\rangle \to \langle \text{RETURN}(v''), m'\rangle}\ (\text{DECL-RETURN})$$

$$\frac{\begin{array}{c} m' = \text{Extend}(m, x \mapsto v) \\ \langle i, m'\rangle \to \langle i', m''\rangle \qquad v' = m''[(|m''|, x)]_A \qquad m''' = m'' - x \end{array}}{\langle \text{DECL } x = v \text{ IN}\{i\}, m\rangle \to \langle \text{DECL } x = v' \text{ IN}\{i'\}, m'''\rangle}\ (\text{DECL-CTX})$$

$$\frac{\langle i, m\rangle \to \Omega}{\langle \text{DECL } x = v \text{ IN}\{i\}, m\rangle \to \Omega}\ (\text{DECL-ERR})$$

Pour traiter l'alternative, on a besoin de 2 règles. Elles commencent de la même manière, en évaluant la condition. Si le résultat est 0 (et seulement dans ce cas), c'est la règle IF-FALSE

4. On peut remarquer qu'il est impossible de définir un contexte d'évaluation $C ::=$ DECL $x = v$ IN$\{C\}$. En effet, puisque celui-ci nécessiterait d'ajouter une variable, il ne préserve pas la mémoire.

qui est appliquée et l'instruction revient à évaluer la branche « *else* ». Dans les autres cas, c'est la règle IF-TRUE qui s'applique et la branche « *then* » qui est prise.

$$\frac{}{\langle \text{IF}(0)\{i_t\}\text{ELSE}\{i_f\}, m\rangle \to \langle i_f, m\rangle}\ (\text{IF-FALSE}) \qquad \frac{v \neq 0}{\langle \text{IF}(v)\{i_t\}\text{ELSE}\{i_f\}, m\rangle \to \langle i_t, m\rangle}\ (\text{IF-TRUE})$$

On exprime la sémantique de la boucle comme une simple règle de réécriture :

$$\frac{}{\langle \text{WHILE}(e)\{i\}, m\rangle \to \langle \text{IF}(e)\{i; \text{WHILE}(e)\{i\}\}, m\rangle}\ (\text{WHILE})$$

Enfin, si un RETURN(·) apparaît dans une séquence, on peut supprimer la suite :

$$\frac{}{\langle \text{RETURN}(v); i, m\rangle \to \langle \text{RETURN}(v), m\rangle}\ (\text{RETURN})$$

4.12 Erreurs

Les erreurs se propagent des données vers l'interprète ; c'est-à-dire que si une expression ou instruction est réduite en une valeur d'erreur Ω, alors une transition est faite vers cet état d'erreur.

Cela est aussi vrai d'une sous-expression ou sous-instruction : si l'évaluation de e_1 provoque une erreur, l'évaluation de $e_1 + e_2$ également. La notion de sous-expression ou sous instruction est définie en fonction des contextes C. Notons que, dans EVAL-ERR, $C(\!|e|\!)$ peut être une expression ou une instruction.

$$\frac{}{\langle \Omega, m\rangle \to \Omega}\ (\text{EXP-ERR}) \qquad \frac{\langle e, m\rangle \to \Omega}{\langle C(\!|e|\!), m\rangle \to \Omega}\ (\text{EVAL-ERR})$$

4.13 Phrases et exécution d'un programme

Un programme est constitué d'une suite de phrases : déclarations de fonctions, de variables et de types, et évaluation d'expressions.

Donc l'évaluation d'une phrase p fait passer d'un état mémoire m à un autre m', ce que l'on note $m \Vdash p \to m'$.

L'évaluation d'une expression est uniquement faite pour ses effets de bord. Par exemple, après avoir défini les fonctions du programme, on pourra appeler `main()`. La déclaration d'une variable globale, quant à elle, consiste à évaluer sa valeur initiale et à étendre l'état mémoire avec ce couple (variable, valeur). On suppose que les variables globales ont toutes des noms différents. Notons que ces évaluations se font à grands pas.

Enfin, l'exécution d'un programme est sans surprise l'exécution de ses phrases, les unes à la suite des autres.

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle}{m \Vdash e \to m'} \text{ (ET-EXP)}$$

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle \qquad m' = (s, g) \qquad m'' = (s, (x \mapsto v) :: g)}{m \Vdash x = e \to m''} \text{ (ET-VAR)}$$

$$\frac{([\,], [\,]) \Vdash p_1 \to m_1 \qquad m_1 \Vdash p_2 \to m_2 \qquad \dots \qquad m_{n-1} \Vdash p_n \to m_n}{\Vdash p_1, \dots, p_n \to^* m} \text{ (PROG)}$$

4.14 Exemple

Considérons le programme suivant :

```
x = 0
f = fun(q) {
    *q <- 1
}
f(&x)
```

Ce programme est constitué des phrases p_1, p_2 et p_3. On rappelle que par rapport à la syntaxe concrète, un RETURN(()) est inséré automatiquement, donc $p_2 = f = \text{fun}(q)\{*q \leftarrow 1; \text{RETURN}(())\}$

D'après PROG, l'évaluer va revenir à évaluer à la suite ces 4 phrases.

Déclaration de x D'après PROG, on part d'un état mémoire $m_0 = ([\,], [\,])$. Pour trouver m_1 tel que $m_0 \Vdash x = 0 \to m_1$, il faut appliquer la règle ET-VAR.

Celle-ci va étendre l'ensemble (vide) des globales mais demande d'évaluer l'expression 0. D'après EXP-CST, $\langle 0, m_0\rangle \to \langle \widehat{0}, m_0\rangle$.

Donc $m_0 \Vdash x = 0 \to m_1$ en posant $m_1 = ([\,], [x \mapsto \widehat{0}])$.

Déclaration de f On se trouve encore dans le cas de la déclaration d'une variable globale. Il faut comme auparavant évaluer l'expression. C'est la règle EXP-FUN qui s'applique : $\langle \text{fun}(q)\{*q \leftarrow 1; \text{RETURN}(())\}, m_1\rangle \to \langle \text{fun}(q)\{\widehat{*q \leftarrow 1; \text{RETURN}(())}\}, m_1\rangle$ (ce qui revient à dire que la fonction est directement placée en mémoire).

Ainsi $m_1 \Vdash f = \text{fun}(q)\{*q \leftarrow 1; \text{RETURN}(())\} \to m_2$ où :

$$m_2 = ([\,], [f \mapsto \text{fun}(q)\{\widehat{*q \leftarrow 1; \text{RETURN}(())}\}; x \mapsto \widehat{0}])$$

Appel de f Ici, on évalue une expression pour ses effets de bords. La règle à appliquer est ET-EXP, qui a comme prémisse $\langle f(\&x), m_2\rangle \to \langle v, m_3\rangle$.

D'après la forme de l'expression, la règle à appliquer va être EXP-CALL-RETURN. Mais il va falloir d'abord réécrire l'expression à l'aide de CTX (pour que l'expression appellée ait la forme $\widehat{f}$ et l'argument soit évalué) et EXP-CALL-CTX (pour que le corps de la fonction ait pour forme RETURN(())).

Tout d'abord on applique donc CTX avec $C = \bullet(\&x)$. Comme on a via PHI-VAR puis EXP-LV :

$$\langle f, m_2\rangle \to \langle \text{fun}(q)\{\widehat{*q \leftarrow 1; \text{RETURN}(())}\}, m_2\rangle$$

On en déduit que :

$$\langle f(\&x), m_2\rangle \to \langle \text{fun}(q)\{\widehat{*q \leftarrow 1; \text{RETURN}(())}\}(\&x), m_2\rangle$$

Une application supplémentaire de CTX permet d'en arriver à la ligne suivante (en évaluant $\&x$ en $\widehat{\&}(x)$ via EXP-ADDR) :

$$\langle f(\&x), m_2\rangle \to \langle \text{fun}(q)\{\widehat{*q \leftarrow 1; \text{RETURN}(())}\}(\widehat{\&}(x)), m_2\rangle$$

La fonction et son argument sont évalués, donc on peut appliquer EXP-CALL-CTX. En posant $m_2' = \text{Push}(m_2, (q \mapsto \widehat{\&}(x)))$, le but est de trouver m_2'' et v tels que :

$$\langle *q \leftarrow 1; \text{RETURN}(()), m_2'\rangle \to \langle \text{RETURN}(v), m_2''\rangle$$

Puisque l'instruction est une séquence, on va appliquer SEQ. La première partie n'étant pas PASS, il faut l'évaluer grâce à la règle CTX avec $C = \bullet; \text{RETURN}(())$.

Le nouveau but est de trouver un m_2'' tel que $\langle *q \leftarrow 1, m_2'\rangle \to \langle \text{PASS}, m_2'\rangle$.

En appliquant PHI-DEREF sous $C = *\bullet \leftarrow 1$, on obtient $\langle *q \leftarrow 1, m_2'\rangle \to \langle (x) \leftarrow 1, m_2'\rangle$.

Puis on applique EXP-CST sous $C = (x) \leftarrow \bullet$ et $\langle *q \leftarrow 1, m_2'\rangle \to \langle (x) \leftarrow \widehat{1}, m_2'\rangle$.

Maintenant que les deux côtés de $\leftarrow$ sont évalués, on peut appliquer EXP-SET, et $\langle *q \leftarrow 1, m_2'\rangle \to \langle \text{PASS}, m_2''\rangle$ où :

$$m_2'' = ([[q \mapsto (x)]], [f \mapsto \text{fun}(q)\{\widehat{*q \leftarrow 1; \text{RETURN}(())}\}; x \mapsto \widehat{1}])$$

Alors, d'après SEQ, $\langle *q \leftarrow 1, m_2'\rangle \to \langle \text{RETURN}(()), m_2''\rangle$. Avec EXP-CST sous $C = \text{RETURN}(\bullet)$, on a donc $\langle *q \leftarrow 1, m_2'\rangle \to \langle \text{RETURN}(\widehat{()}), m_2''\rangle$.

On peut enfin appliquer EXP-CALL-CTX pour en déduire que :

$$\langle f(\&x), m_2\rangle \to \langle \text{fun}(q)\{\text{RETURN}(\widehat{()})\}(\widehat{\&}(x)), m_2'''\rangle$$

Donc d'après EXP-CALL-RETURN (car on a $m_2''' = \text{Cleanup}(m_2''')$ et $\widehat{()} = \text{CleanV}_{|m_2'''|}(\widehat{()})$) :

$$\langle f(\&x), m_2\rangle \to \langle \widehat{()}, m_2'''\rangle$$

En posant $m_3 = m_2'''$, on a $m_2 \Vdash f(\&x) \to m_3$.

Donc pour conclure (grâce à PROG), on a $\Vdash [p_1, p_2, p_3] \to^* m_3$.

Conclusion

On vient de définir un langage impératif, SAFESPEAK. Le but est que celui-ci serve de support à des analyses statiques, afin notamment de montrer une propriété de sécurité sur les pointeurs. Pour le moment, on a seulement défini ce que sont les programmes (leur syntaxe) et comment ils s'exécutent (leur sémantique). Sur ces deux points, on note que nous sommes restés suffisamment proches de C, tout en utilisant pour la mémoire un modèle plus structuré qu'une simple suite d'octets. Les définitions de la syntaxe ainsi que de la sémantique sont rappelées dans l'annexe B (sections B.1 à B.7).

Afin de manipuler les états mémoire dans la sémantique d'évaluation, nous avons utilisé le concept des lentilles, qui permettent de chaîner des accesseurs entre eux et d'accéder simplement à des valeurs profondes de la mémoire, en utilisant le même outil pour la lecture et l'écriture.

Pour le moment, on ne peut rien présager de l'exécution d'un programme bien formé syntaxiquement. Pour la grande majorité des programmes bien formés (à la syntaxe correcte), l'évaluation s'arrêtera soit par une erreur, soit parce qu'aucune règle d'évaluation ne peut s'appliquer. Dans les chapitres 5 et 6, nous allons donc définir un système de types qui permet de rejeter ces programmes se comportant mal à l'exécution.

CHAPITRE 5

TYPAGE

Dans ce chapitre, nous enrichissons le langage défini dans le chapitre 4 d'un système de types. Celui-ci permet de séparer les programmes bien formés, comme celui de la figure 5.1(a), des programmes mal formés, comme celui de la figure 5.1(b). Intuitivement, le programme mal formé provoquera des erreurs à l'exécution car il manipule des données de manière incohérente : la variable x reçoit 1, donc elle se comporte comme un entier, puis est déférencée, se comportant comme un pointeur.

```
f = Fun() {
  Decl x = 0 in
  x <- 1
  return x
}
```

(a) Programme bien formé

```
f = Fun() {
  Decl x = 0 in
  x <- 1
  return (*x)
}
```

(b) Programme mal formé

FIGURE 5.1: Programmes bien et mal formés

Le but d'un tel système de types est de rejeter les programmes pour lesquels on peut facilement déterminer qu'ils sont faux, c'est-à-dire dont on peut prouver qu'ils provoqueraient des erreurs à l'exécution dues à une incompatibilité entre valeurs. En ajoutant cette étape, on restreint la classe d'erreurs qui pourraient bloquer la sémantique.

On emploie un système de types monomorphe : à chaque expression, on associe un unique type. En plus des types de base INT, FLOAT et UNIT, on peut construire des types composés : pointeurs, tableaux, structures et fonctions.

Pour typer les structures, on suppose avoir à notre disposition (par un prétraitement, par exemple) le type complet de la structure à chaque accès d'un champ. Cela alourdit la présentation, mais permet d'éviter le sous-typage qui rend l'inférence indécidable. En pratique, dans l'implantation décrite dans le chapitre 7, on utilise une variante du polymorphisme de rangée (*row polymorphism*) [RV98] présent dans OCaml pour unifier deux types structures partiellement connus.

Le principe du typage est d'associer à chaque construction syntaxique une étiquette représentant le genre de valeurs qu'elle produira. Dans le programme de la figure 5.1(a), la variable x est initialisée avec la valeur 0 ; c'est donc un entier. Cela signifie que, dans tout le programme, toutes les instances de cette variable [1] porteront ce type. La première instruction

1. Deux variables peuvent avoir le même nom dans deux fonctions différentes, par exemple. Dans ce cas il n'y a aucune contrainte particulière entre ces deux variables. L'analyse de typage se fait toujours dans un contexte

est l'affectation de la constante 1 (entière) à x dont on sait qu'elle porte des valeurs entières, ce qui est donc correct. Le fait de rencontrer RETURN(x) permet de conclure que le type de la fonction est ( ) → INT (c'est-à-dire qu'elle n'a pas d'arguments et qu'elle retourne un INT).

Dans la seconde fonction, au contraire, l'opérateur $*$ est appliqué à x (le début de l'analyse est identique et permet de conclure que x porte des valeurs entières). Or cet opérateur prend un argument d'un type pointeur de la forme $t*$ et renvoie alors une valeur de type t. Ceci est valable pour tout t (INT, FLOAT ou même $t'*$: le déréférencement d'un pointeur sur pointeur donne un pointeur), mais le type de x, INT, n'est pas de cette forme. Ce programme est donc mal typé.

Dans ce chapitre, on commence par poser les notations qui vont servir à définir la relation de typage. Ensuite, on explique les différentes règles de typage sur les composantes de SAFESPEAK : expressions, instructions et phrases. Enfin, dans le reste du chapitre on établit des propriétés qui sont respectées par les programmes bien typés. On conclut par les théorèmes de progrès et de préservation qui établissent la sûreté du typage.

5.1 Environnements et notations

Les types associés aux expressions sont décrits dans la figure 5.2. Tous sont des types concrets : il n'y a pas de polymorphisme.

Type	$t ::=$ INT	Entier
	| FLOAT	Flottant
	| UNIT	Unité
	| $t*$	Pointeur
	| t []	Tableau
	| S	Structure
	| $(t_1, \ldots, t_n) \rightarrow t$	Fonction
Structure	$S ::= \{l_1 : t_1; \ldots; l_n : t_n\}$	

FIGURE 5.2: Types et environnements de typage

Pour maintenir les contextes de typage, un environnement Γ associe un type à un ensemble de variables.

Plus précisément, un environnement Γ est composé de deux listes de couples (variable, type) : une pour les variables locales, et une pour les variables globales. Cette distinction est nécessaire pour les définitions de fonctions : on remplace la liste des variables locales, mais on conserve le type des variables globales.

Si $\Gamma = (\Gamma_G, \Gamma_L) = ((\gamma_i)_{i\in[1;n]}, (\eta_i)_{i\in[1;m]})$, avec $\gamma_i = (g_i, t_i)$ et $\eta_i = (l_i, u_i)$, on utilise les notations suivantes :

---

précis.

$$x : t \in \Gamma \stackrel{\text{def}}{=\!=} \exists i \in [1;n], \gamma_i = (x,t) \vee \exists i \in [1;m], \eta_i = (x,t)$$

$$\text{dom}(\Gamma_G) \stackrel{\text{def}}{=\!=} \{g_i / i \in [1;n]\}$$

$$\text{dom}(\Gamma_L) \stackrel{\text{def}}{=\!=} \{l_i / i \in [1;m]\}$$

$$\text{dom}(\Gamma) \stackrel{\text{def}}{=\!=} \text{dom}(\Gamma_G) \cup \text{dom}(\Gamma_L)$$

$$\Gamma, \text{global } x : t \stackrel{\text{def}}{=\!=} ((\gamma'_i)_{i\in[1;n+1]}, \Gamma_L) \text{ tel que } \begin{cases} \forall i \in [1;n], \gamma'_i = \gamma_i \\ \gamma_{n+1} = (x,t) \end{cases}$$

$$\Gamma, \text{local } x : t \stackrel{\text{def}}{=\!=} (\Gamma_G, (\eta'_i)_{i\in[1;m+1]}) \text{ tel que } \begin{cases} \forall i \in [1;m], \eta'_i = \eta_i \\ \eta_{n+1} = (x,t) \end{cases}$$

Le type des fonctions semble faire apparaître un n-uplet $(t_1, \ldots, t_n)$ mais ce n'est qu'une notation : il n'y a pas de n-uplets de première classe ; ils sont toujours présents dans un type fonctionnel.

Le typage correspond à la définition des trois jugements suivants. Les deux premiers sont mutuellement récursifs car une instruction peut consister en l'évaluation d'une expression, et la définition d'une fonction repose sur le typage de son corps.

Typage d'une expression : on note de la manière suivante le fait qu'une expression e (telle que définie dans la figure 4.4) ait pour type t dans le contexte Γ.

$$\Gamma \vdash e : t$$

Typage d'une instruction : les instructions n'ont en revanche pas de type. Mais il est tout de même nécessaire de vérifier que toutes les sous-expressions apparaissant dans une instruction sont cohérentes ensemble.

On note de la manière suivante le fait que sous l'environnement Γ l'instruction i est bien typée :

$$\Gamma \vdash i$$

Typage d'une phrase : De par leur nature séquentielle, les phrases qui composent un programme altèrent l'environnement de typage. Par exemple, la déclaration d'une variable globale ajoute une valeur dans l'environnement.

On note de la manière suivante le fait que le typage de la phrase p transforme l'environnement Γ en Γ' :

$$\Gamma \vdash p \rightarrow \Gamma'$$

On étend cette notation aux suites de phrases, ce qui définit le typage d'un programme, ce que l'on note $\vdash P$.

5.2 Expressions

Littéraux

Le typage des littéraux numériques ne dépend pas de l'environnement de typage : ce sont toujours des entiers ou des flottants.

$$\frac{}{\Gamma \vdash n : \text{INT}}\ (\text{CST-INT}) \qquad \frac{}{\Gamma \vdash d : \text{FLOAT}}\ (\text{CST-FLOAT})$$

Le pointeur nul, quant à lui, est compatible avec tous les types pointeur. Cependant, il conserve bien un type monomorphe : le type t n'est pas généralisé.

$$\frac{}{\Gamma \vdash \text{NULL} : t*}\ (\text{CST-NULL})$$

Enfin, le littéral unité a le type UNIT.

$$\frac{}{\Gamma \vdash () : \text{UNIT}}\ (\text{CST-UNIT})$$

Valeurs gauches

Rappelons que l'environnement de typage Γ contient le type des variables accessibles du programme. Le cas où la valeur gauche à typer est une variable est donc direct : il suffit de retrouver son type dans l'environnement.

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t}\ (\text{LV-VAR})$$

Dans le cas d'un déréférencement, on commence par typer la valeur gauche déréférencée. Si elle a un type pointeur, la valeur déréférencée est du type pointé.

$$\frac{\Gamma \vdash e : t*}{\Gamma \vdash *e : t}\ (\text{LV-DEREF})$$

Pour une valeur gauche indexée (l'accès à tableau), on s'assure que l'indice soit entier, et que la valeur gauche a un type tableau : le type de l'élement est encore une fois le type de base du type tableau.

$$\frac{\Gamma \vdash e : \text{INT} \qquad \Gamma \vdash lv : t[\,]}{\Gamma \vdash lv[e] : t}\ (\text{LV-INDEX})$$

Le typage de l'accès à un champ est facilité par le fait que dans le programme, le type complet de la structure est accessible sur chaque accès.

Dans la définition de cette règle on utilise la notation :

$$(l,t) \in \{l_1 : t_1; \ldots; l_n : t_n\} \stackrel{\text{def}}{=\!=} \exists i \in [1;n], l = l_i \wedge t = t_i$$

$$\frac{(l,t) \in S \qquad \Gamma \vdash lv : S}{\Gamma \vdash lv.l_S : t}\ (\text{LV-FIELD})$$

Opérateurs

Un certain nombre d'opérations est possible sur le type INT.

$$\frac{\boxplus \in \{+,-,\times,/,\&,|,\hat{},\&\&,||,\ll,\gg,\leq,\geq,<,>\} \qquad \Gamma \vdash e_1 : \text{INT} \qquad \Gamma \vdash e_2 : \text{INT}}{\Gamma \vdash e_1 \boxplus e_2 : \text{INT}} \text{ (OP-INT)}$$

De même sur FLOAT.

$$\frac{\boxplus \in \{+.,-.,\times.,/.,\leq.,\geq.,<.,>.\} \qquad \Gamma \vdash e_1 : \text{FLOAT} \qquad \Gamma \vdash e_2 : \text{FLOAT}}{\Gamma \vdash e_1 \boxplus e_2 : \text{FLOAT}} \text{ (OP-FLOAT)}$$

Les opérateurs de comparaison peuvent s'appliquer à deux opérandes qui sont d'un type qui supporte l'égalité. Ceci est représenté par un jugement $\text{EQ}(t)$ qui est vrai pour les types INT, FLOAT et pointeurs, ainsi que les types composés si les types de leurs composantes le supportent (figure 5.3). Les opérateurs = et $\neq$ renvoient alors un INT :

$$\frac{\boxplus \in \{=,\neq\} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \qquad \text{EQ}(t)}{\Gamma \vdash e_1 \boxplus e_2 : \text{INT}} \text{ (OP-EQ)}$$

$\boxed{\text{EQ}(t)}$

$$\frac{t \in \{\text{INT}, \text{FLOAT}\}}{\text{EQ}(t)} \text{ (EQ-NUM)} \qquad \frac{}{\text{EQ}(t*)} \text{ (EQ-PTR)} \qquad \frac{\text{EQ}(t)}{\text{EQ}(t[\,])} \text{ (EQ-ARRAY)}$$

$$\frac{\forall i \in [1;n].\text{EQ}(t_i)}{\text{EQ}(\{l_1 : t_1; \ldots l_n : t_n\})} \text{ (EQ-STRUCT)}$$

FIGURE 5.3: Jugements d'égalité sur les types

Les opérateurs unaires « + » et « − » appliquent aux entiers, et leurs équivalents « +. » et « −. » aux flottants.

$$\frac{\Gamma \vdash e : \text{INT}}{\Gamma \vdash +e : \text{INT}} \text{ (UNOP-PLUS-INT)} \qquad \frac{\Gamma \vdash e : \text{FLOAT}}{\Gamma \vdash +.e : \text{FLOAT}} \text{ (UNOP-PLUS-FLOAT)}$$

$$\frac{\Gamma \vdash e : \text{INT}}{\Gamma \vdash -e : \text{INT}} \text{ (UNOP-MINUS-INT)} \qquad \frac{\Gamma \vdash e : \text{FLOAT}}{\Gamma \vdash -.e : \text{FLOAT}} \text{ (UNOP-MINUS-FLOAT)}$$

Les opérateurs de négation unaires, en revanche, ne s'appliquent qu'aux entiers.

$$\frac{\boxminus \in \{\sim,!\} \qquad \Gamma \vdash e : \text{INT}}{\Gamma \vdash \boxminus e : \text{INT}} \text{ (UNOP-NOT)}$$

L'arithmétique de pointeurs préserve le type des pointeurs.

$$\frac{\boxplus \in \{+_p, -_p\} \qquad \Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \text{INT}}{\Gamma \vdash e_1 \boxplus e_2 : t*} \text{(PTR-ARITH)}$$

Autres expressions

Prendre l'adresse d'une valeur gauche rend un type pointeur sur le type de celle-ci.

$$\frac{\Gamma \vdash lv : t}{\Gamma \vdash \&lv : t*} \text{(ADDR)}$$

Pour typer une affectation, on vérifie que la valeur gauche (à gauche) et l'expression (à droite) ont le même type. C'est alors le type résultat de l'expression d'affectation.

$$\frac{\Gamma \vdash lv : t \qquad \Gamma \vdash e : t}{\Gamma \vdash lv \leftarrow e : t} \text{(SET)}$$

Un littéral tableau a pour type $t[]$ où t est le type de chacun de ses éléments.

$$\frac{\forall i \in [1;n], \Gamma \vdash e_i : t}{\Gamma \vdash [e_1;\ldots;e_n] : t[\,]} \text{(ARRAY)}$$

Un littéral de structure est bien typé si ses champs sont bien typés.

$$\frac{\forall i \in [1;n], \Gamma \vdash e_i : t_i}{\Gamma \vdash \{l_1 : e_1;\ldots;l_n : e_n\} : \{l_1 : t_1;\ldots;l_n : t_n\}} \text{(STRUCT)}$$

Pour typer un appel de fonction, on s'assure que la fonction a bien un type fonctionnel. On type alors chacun des arguments avec le type attendu. Le résultat est du type de retour de la fonction.

$$\frac{\Gamma \vdash e : (t_1,\ldots,t_n) \to t \qquad \forall i \in [1;n], \Gamma \vdash e_i : t_i}{\Gamma \vdash e(e_1,\ldots,e_n) : t} \text{(CALL)}$$

5.3 Instructions

La séquence est simple à traiter : l'instruction vide est toujours bien typée, et la suite de deux instructions est bien typée si celles-ci le sont également.

$$\frac{}{\Gamma \vdash \text{PASS}} \text{(PASS)} \qquad\qquad \frac{\Gamma \vdash i_1 \qquad \Gamma \vdash i_2}{\Gamma \vdash i_1; i_2} \text{(SEQ)}$$

Une instruction constituée d'une expression est bien typée si celle-ci peut être typée dans ce même contexte.

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e} \text{ (EXP)}$$

Une déclaration de variable est bien typée si son bloc interne est bien typé quand on ajoute à l'environnement la variable avec le type de sa valeur initiale.

$$\frac{\Gamma \vdash e : t \qquad \Gamma, \text{local } x : t \vdash i}{\Gamma \vdash \text{DECL } x = e \text{ IN}\{i\}} \text{ (DECL)}$$

Les constructions de contrôle sont bien typées si leurs sous-instructions sont bien typées, et si la condition est d'un type entier.

$$\frac{\Gamma \vdash e : \text{INT} \qquad \Gamma \vdash i_1 \qquad \Gamma \vdash i_2}{\Gamma \vdash \text{IF}(e)\{i_1\}\text{ELSE}\{i_2\}} \text{ (IF)} \qquad \frac{\Gamma \vdash e : \text{INT} \qquad \Gamma \vdash i}{\Gamma \vdash \text{WHILE}(e)\{i\}} \text{ (WHILE)}$$

5.4 Fonctions

Le typage des fonctions fait intervenir une variable virtuelle $\underline{R}$. Cela revient à typer l'instruction RETURN(e) comme $\underline{R} \leftarrow e$. Cela rappelle le langage Pascal, où pour retourner une valeur on l'affecte à une variable nommée comme la fonction courante [2].

$$\frac{\underline{R} : t \in \Gamma \qquad \Gamma \vdash e : t}{\Gamma \vdash \text{RETURN}(e)} \text{ (RETURN)}$$

Pour typer une définition de fonction, on commence par créer un nouvel environnement de typage Γ' obtenu par la suite d'opérations suivantes :

- on enlève l'ensemble des locales. Cela inclut le couple $\underline{R} : t_f$ correspondant à la valeur de retour de la fonction appelante.
- on ajoute les types des arguments $a_i : t_i$
- on ajoute le type de la valeur de retour de la fonction appelée, $\underline{R} : t$

Si le corps de la fonction est bien typé sous Γ', alors la fonction est typable en $(t_1, \ldots, t_n) \to t$ sous Γ.

$$\frac{\Gamma = (\Gamma_G, \Gamma_L) \qquad \Gamma' = (\Gamma_G, [a_1 : t_1, \ldots, a_n : t_n, \underline{R} : t]) \qquad \Gamma' \vdash i}{\Gamma \vdash \text{fun}(a_1, \ldots, a_n)\{i\} : (t_1, \ldots, t_n) \to t} \text{ (FUN)}$$

5.5 Phrases

Le typage des phrases est détaillé dans la figure 5.4. Le typage d'une expression est le cas le plus simple. En effet, il y a juste à vérifier que celle-ci est bien typable (avec ce type) dans l'environnement de départ : l'environnement n'est pas modifié. En revanche, la déclaration d'une variable globale commence de la même manière, mais on enrichit l'environnement de typage des globales de cette nouvelle association.

2. Si on n'avait pas introduit la restriction que chaque fonction doit terminer par un RETURN(·) (page 56), alors le type de $\underline{R}$ pourrait rester inconnu. En pratique cela veut dire que la valeur de retour d'une telle fonction serait compatible avec n'importe quel type t, ce qui briserait la sûreté du typage.

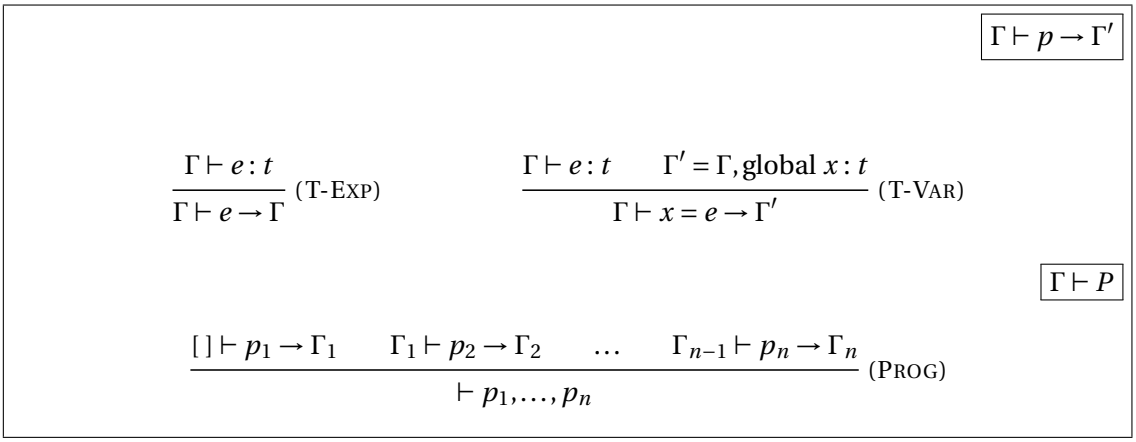


FIGURE 5.4: Typage des phrases et programmes

5.6 Sûreté du typage

Comme nous l'évoquions au début de ce chapitre, le but du typage est de rejeter certains programmes afin de ne garder que ceux qui ne provoquent pas un certain type d'erreurs à l'exécution.

Dans la suite, nous donnons des propriétés que respectent tous les programmes bien typés. Il est traditionnel de rappeler l'adage de Robin Milner :

> Well-typed programs don't go wrong.

To go wrong reste bien sûr à définir ! Cette sûreté du typage repose sur deux théorèmes :

- progrès : si un terme est bien typé, il y a toujours une règle d'évaluation qui s'applique.
- préservation (ou *subject reduction*) : l'évaluation transforme un terme bien typé en un terme du même type.

5.7 Typage des valeurs

Puisque nous allons manipuler les propriétés statiques et dynamiques des programmes, nous allons avoir à traiter des environnements de typage Γ et des états mémoires m. La première chose à faire est donc d'établir une correspondance entre ces deux mondes.

Étant donné un état mémoire m, on associe un type de valeur τ aux valeurs v. Cela est fait sous la forme d'un jugement $m \vDash v : \tau$.

Ces types de valeurs ne sont pas exactement les mêmes que les types statiques. Pour les calculer, on n'a pas accès au code du programme, seulement à ses données. Il est par exemple possible de reconnaître le type des constantes, mais pas celui des fonctions. Celles-ci sont en fait le seul cas qu'il est impossible de déterminer à l'exécution. On le remplace donc par un cas plus simple où seul l'arité est conservée. L'alternative serait de garder le type à l'exécution, ce qui éloigne d'un langage à la C.

Remarque Le fait d'effacer les types à l'exécution est un choix permettant d'alléger les valeurs en mémoire. Il serait aussi possible de conserver les types complets à l'exécution, afin de permettre une introspection dynamique des valeurs.

Le cas des références (règle S-PTR) utilise le typage des valeurs gauches, codéfini par :

$$m \vDash_\Phi \varphi : \tau \stackrel{\text{def}}{=\!=} m \vDash m[\varphi] : \tau$$

Les règles de définition du typage des valeurs sont données dans la figure 5.5. On rappelle que Φ est la lentille indexée définie page 52.

Type de valeur	$\tau ::= \text{INT}$	Entier
	$\mid \text{FLOAT}$	Flottant
	$\mid \text{UNIT}$	Unité
	$\mid \tau *$	Pointeur
	$\mid \tau[\,]$	Tableau
	$\mid \{l_1 : \tau_1; \ldots; l_n : \tau_n\}$	Structure
	$\mid \text{FUN}_n$	Fonction

FIGURE 5.5: Types de valeurs

Les règles sont détaillées dans la figure 5.6 : les types des constantes sont simples à retrouver car il y a assez d'information en mémoire. Pour les références, ce qui peut être déréférencé en une valeur de type τ est un $\tau *$. Le typage des valeurs composées se fait en profondeur. Enfin, la seule information restant à l'exécution sur les fonctions est son arité.

$\boxed{m \vDash v : \tau}$

$$\frac{}{m \vDash n : \text{INT}}\ (\text{S-INT}) \qquad \frac{}{m \vDash d : \text{FLOAT}}\ (\text{S-FLOAT}) \qquad \frac{}{m \vDash () : \text{UNIT}}\ (\text{S-UNIT})$$

$$\frac{}{m \vDash \text{NULL} : \tau *}\ (\text{S-NULL}) \qquad \frac{m \vDash_\Phi \varphi : \tau}{m \vDash \widehat{\&}\,\varphi : \tau *}\ (\text{S-PTR}) \qquad \frac{\forall i \in [1;n].m \vDash v_i : \tau}{m \vDash \widehat{[v_1;\ldots;v_n]} : \tau[\,]}\ (\text{S-ARRAY})$$

$$\frac{\forall i \in [1;n].m \vDash v_i : \tau_i}{m \vDash \{l_1 : \widehat{v_1;\ldots;l_n} : v_n\} : \{l_1 : \tau_1;\ldots;l_n : \tau_n\}}\ (\text{S-STRUCT})$$

$$\frac{}{m \vDash \text{fun}(x_1,\ldots,x_n)\{i\} : \text{FUN}_n}\ (\text{S-FUN})$$

FIGURE 5.6: Règles de typage des valeurs

La prochaine étape est de définir une relation de compatibilité entre les types de valeurs τ et statiques t. Nous noterons ceci sous la forme d'un jugement $\tau \triangleright t$. Les règles sont décrites dans la figure 5.7, la règle importante étant COMP-FUN. Notons qu'on garde le même nom pour les types de base, et que par exemple INT peut être vu soit comme un type statique, soit comme un type de valeur. Il y a donc un abus de notation dans la règle COMP-GROUND : quand on note $\text{INT} \triangleright \text{INT}$, le premier désigne le type des valeurs à l'exécution, et le second le type statique.

On définit enfin la notion d'état mémoire bien typé. On dit qu'un état mémoire m est bien typé sous un environnement Γ, ce que l'on note $\Gamma \vDash m$, si le types des valeurs à l'exécution présent dans m est « compatible » avec les types présents dans Γ.

Cela se fait par induction sur la forme de Γ et m. Fonctionnellement, cela implique que les accès à la mémoire retournent des valeurs en accord avec le type statique (lemme 5.6). Les

$\boxed{\tau \rhd t}$

$$\frac{t \in \{\text{INT}, \text{FLOAT}, \text{UNIT}\}}{t \rhd t}\ (\text{COMP-GROUND}) \qquad \frac{\tau \rhd t}{\tau\ * \rhd t\ *}\ (\text{COMP-PTR}) \qquad \frac{\tau \rhd t}{\tau[\,] \rhd t[\,]}\ (\text{COMP-ARRAY})$$

$$\frac{\forall i \in [1;n].\tau_i \rhd t_i}{\{l_1 : \tau_1; \ldots; l_n : \tau_n\} \rhd \{l_1 : t_1; \ldots; l_n : t_n\}}\ (\text{COMP-STRUCT}) \qquad \frac{}{\text{FUN}_n \rhd (t_1, \ldots, t_n) \to t}\ (\text{COMP-FUN})$$

FIGURE 5.7: Compatibilité entre types de valeurs et statiques

$$\frac{}{[\,] \vDash ([\,], [\,])}\ (\text{M-EMPTY}) \qquad \frac{\Gamma \vDash (s,g) \qquad (s,g) \vDash v : \tau \qquad \tau \rhd t}{\Gamma, \text{global } x : t \vDash (s, ((x \mapsto v) :: g))}\ (\text{M-GLOBAL})$$

$$\frac{\begin{array}{c}\Gamma \vDash m \qquad \Gamma = (\Gamma_G, \Gamma_L) \qquad \Gamma' = (\Gamma_G, [x_1 : t_1, \ldots, x_n : t_n, \underline{R} : t]) \\ m \vDash v_1 : \tau_1 \qquad \tau_1 \rhd t_1 \qquad \ldots \qquad m \vDash v_n : \tau_n \qquad \tau_n \rhd t_n\end{array}}{\Gamma' \vDash \text{Push}(m, ((x_1 \mapsto v_1), \ldots, (x_n \mapsto v_n)))}\ (\text{M-PUSH})$$

$$\frac{\begin{array}{c}\Gamma = (\Gamma_G, \Gamma_L) \qquad \Gamma \vDash m \\ m' = \text{Push}(m, ((x_1 \mapsto v_1), \ldots, (x_n \mapsto v_n))) \qquad (\Gamma_G, [x_1 : t_1, \ldots, x_n : t_n, \underline{R} : t]) \vDash m'\end{array}}{\Gamma \vDash \text{Cleanup}(\text{Pop}(m'))}\ (\text{M-POP})$$

$$\frac{\Gamma \vDash m \qquad m \vDash v : \tau \qquad \tau \rhd t \qquad x \notin \Gamma}{\Gamma, \text{local } x : t \vDash \text{Extend}(m, x \mapsto v)}\ (\text{M-DECL}) \qquad \frac{\Gamma, \text{local } x : t \vDash m \qquad x \notin \Gamma}{\Gamma \vDash \text{CleanVar}(m - x, x)}\ (\text{M-DECLCLEAN})$$

$$\frac{\Gamma \vDash m \qquad \Gamma \vdash \varphi : t \qquad m \vDash v : \tau \qquad \tau \rhd t \qquad m' = m[\varphi \leftarrow v]}{\Gamma \vDash m'}\ (\text{M-WRITE})$$

FIGURE 5.8: Compatibilité entre états mémoire et environnements de typage

règles définissant cette relation sont données dans la figure 5.8.

5.8 Propriétés du typage

On commence par énoncer quelques lemmes utiles dans la démonstration de ces théorèmes. Certaines démonstrations sont des analyses de cas laborieuses et sans difficultés ; dans ce cas on n'en donne que des esquisses.

Lemme 5.1 (Inversion)**.** *À partir d'un jugement de typage, on peut en déduire des informations sur les types de ses sous-expressions.*

- *Constantes*
  - *si* $\Gamma \vdash n : t$, *alors* $t = \text{INT}$
  - *si* $\Gamma \vdash d : t$, *alors* $t = \text{FLOAT}$
  - *si* $\Gamma \vdash \text{NULL} : t$, *alors* $\exists t', t = t'*$

  - *si* $\Gamma \vdash () : t$, *alors* $t = \text{UNIT}$
- *Références mémoire :*
  - *si* $\Gamma \vdash x : t$, *alors* $x : t \in \Gamma$
  - *si* $\Gamma \vdash *e : t$, *alors* $\Gamma \vdash e : t*$
  - *si* $\Gamma \vdash lv[e] : t$, *alors* $\Gamma \vdash lv : t[]$ *et* $\Gamma \vdash e : \text{INT}$
  - *si* $\Gamma \vdash lv.l_S : t$, *alors* $\Gamma \vdash lv : S$
- *Opérations :*
  - *si* $\Gamma \vdash \boxminus e : t$, *alors on est dans un des cas suivants :*
    - $\boxminus \in \{+, -, \sim, !\}$, $t = \text{INT}$, $\Gamma \vdash e : \text{INT}$
    - $\boxminus \in \{+., -.\}$, $t = \text{FLOAT}$, $\Gamma \vdash e : \text{FLOAT}$
  - *si* $\Gamma \vdash e_1 \boxplus e_2 : t$, *un des cas suivants se présente :*
    - $\boxplus \in \{+, -, \times, /, \&, |, \wedge, \&\&, ||, \ll, \gg, \leq, \geq, <, >\}$, $\Gamma \vdash e_1 : \text{INT}$, $\Gamma \vdash e_2 : \text{INT}$, $t = \text{INT}$
    - $\boxplus \in \{+., -., \times., /., \leq ., \geq ., < ., > .\}$, $\Gamma \vdash e_1 : \text{FLOAT}$, $\Gamma \vdash e_2 : \text{FLOAT}$, $t = \text{FLOAT}$
    - $\boxplus \in \{=, \neq\}$, $\Gamma \vdash e_1 : t'$, $\Gamma \vdash e_2 : t'$, $\text{EQ}(t')$, $t = \text{INT}$
    - $\boxplus \in \{\leq, \geq, <, >\}$, $t = \text{INT}$, $\Gamma \vdash e_1 : t'$, $\Gamma \vdash e_2 : t'$, $t' \in \{\text{INT}, \text{FLOAT}\}$
    - $\boxplus \in \{+_p, -_p\}$, $\exists t'$, $t = t'*$, $\Gamma \vdash e_1 : t'*$, $\Gamma \vdash e_2 : \text{INT}$
- *Appel de fonction : si* $\Gamma \vdash e(e_1, \ldots, e_n) : t$, *il existe* $(t_1, \ldots, t_n)$ *tels que :*

$$\begin{cases} \Gamma \vdash e : (t_1, \ldots, t_n) \to t \\ \forall i \in [1; n], \Gamma \vdash e_i : t_i \end{cases}$$

- *Fonction : si* $(\Gamma_G, \Gamma_L) \vdash \text{fun}(a_1, \ldots, a_n)\{i\} : t$, *alors il existe* $(t_1, \ldots, t_n)$ *et* t' *tels que :*

$$\begin{cases} t' = (t_1, \ldots, t_n) \to t \\ (\Gamma_G, [a_1 : t_1, \ldots, a_n : t_n, \underline{R} : t]) \vdash i \end{cases}$$

- *Si* $\Gamma \vdash lv \leftarrow e : t$, *alors* $\Gamma \vdash lv : t$ *et* $\Gamma \vdash e : t$.
- *Si* $\Gamma \vdash \& lv : t$, *alors il existe* t' *tel que* $\Gamma \vdash lv : t'$ *et* $t = t'*$.
- *Instructions :*
  - *Si* $\Gamma \vdash i_1; i_2$, *alors* $\Gamma \vdash i_1$ *et* $\Gamma \vdash i_2$.
  - *Si* $\Gamma \vdash e$, *alors il existe* t *tel que* $\Gamma \vdash e : t$.
  - *Si* $\Gamma \vdash \text{DECL } x = e \text{ IN}\{i\}$, *alors il existe* t *tel que* $\Gamma \vdash e : t$ *et* $\Gamma, \text{local } x : t \vdash i$.
  - *Si* $\Gamma \vdash \text{IF}(e)\{i_t\}\text{ELSE}\{i_f\}$, *alors* $\Gamma \vdash e : \text{INT}$, $\Gamma \vdash i_t$ *et* $\Gamma \vdash i_f$.
  - *Si* $\Gamma \vdash \text{WHILE}(e)\{i\}$, *alors* $\Gamma \vdash e : \text{INT}$ *et* $\Gamma \vdash i$.
  - *Si* $\Gamma \vdash \text{RETURN}(e)$, *alors il existe* t *tel que* $\Gamma \vdash e : t$ *et* $\Gamma \vdash \underline{R} : t$.

Démonstration (esquisse). Pour chaque forme de jugement de typage, on liste les règles qui peuvent amener à cette conclusion. □

Il est aussi possible de réaliser l'opération inverse : à partir du type d'une valeur, on peut déterminer sa forme syntaxique. C'est bien sûr uniquement possible pour les valeurs, pas pour n'importe quelle expression (par exemple l'expression x (variable) peut avoir n'importe quel type t dans le contexte $\Gamma = x : t$).

Lemme 5.2 (Formes canoniques)**.** *Il est possible de déterminer la forme syntaxique d'une valeur étant donné son type, comme décrit dans le tableau suivant. Par exemple, d'après la première ligne, si* $\Gamma \vdash v : \text{INT}$*, alors* v *est de la forme* $\widehat{n}$ *(cf. figure 4.7, page 44 pour la définition des valeurs).*

Type de v	*Forme de* v
INT	n
FLOAT	d
UNIT	()
$t*$	$\widehat{\&}\,\varphi$ *ou* NULL
$t[\,]$	$[v_1;\ldots;v_n]$
$\{l_1 : t_1;\ldots;l_n : t_n\}$	$\{l_1 : v_1;\ldots;l_n : v_n\}$
$(t_1,\ldots,t_n) \to t$	$\text{fun}(a_1,\ldots,a_n)\{i\}$

Démonstration (esquisse). On fait comme pour le lemme d'inversion : Pour chaque forme syntaxique, on fait l'inventaire des règles pouvant arriver à cette dérivation. □

Lemme 5.3 (Représentabilité)**.** *On définit un opérateur de représentation d'un type statique à l'exécution :*

$$\begin{aligned}
\text{Repr}(\text{INT}) &= \text{INT}\\
\text{Repr}(\text{FLOAT}) &= \text{FLOAT}\\
\text{Repr}(\text{UNIT}) &= \text{UNIT}\\
\text{Repr}(t'*) &= \text{Repr}(t')*\\
\text{Repr}(t'[]) &= \text{Repr}(t')[]\\
\text{Repr}(\{l_1 : t_1;\ldots;l_n : t_n\}) &= \{l_1 : \text{Repr}(t_1);\ldots;l_n : \text{Repr}(t_n)\}\\
\text{Repr}((t_1,\ldots,t_n) \to t) &= \text{FUN}_n
\end{aligned}$$

Supposons que $\Gamma \vdash v : t$ *et* $\Gamma \vDash m$*. On pose* $\tau = \text{Repr}(t)$*. Alors* $m \vDash v : \tau$ *et* $\tau \rhd t$*.*

Démonstration. On procède par induction sur la forme de t.

- INT : D'après le lemme des formes canoniques, $v = n$. On conclut avec S-INT et COMP-GROUND.
- FLOAT : Idem avec $v = d$ et S-FLOAT.
- UNIT : Idem avec $v = ()$ et S-UNIT.
- $t = t'*$: Soient $\tau' = \text{Repr}(t')$ et $\tau = \tau'*$. D'après le lemme des formes canoniques, deux cas sont possibles :
  - $v = \widehat{\&}\,\varphi$:
    Par inversion (lemme 5.1), $\Gamma \vdash \varphi : t'$.
    Puisque $\Gamma \vDash m$ et $\Gamma \vdash \varphi : t'$, on obtient par le lemme 5.6 que $m[\varphi]$ est une valeur telle que $m \vDash m[\varphi] : \tau'$ où $\tau' \rhd t'$. D'après S-PTR, $m \vDash \widehat{\&}\,\varphi : \tau$.
    De plus par COMP-PTR $\tau \rhd t$.
  - $v = \text{NULL}$:
    Par induction, $\tau' \rhd t'$. Alors par COMP-PTR, $\tau\ * \rhd t\ *$.
    En outre, grâce à S-NULL, on obtient $m \vDash \text{NULL} : \tau\ *$.

- $t = t'[]$: Par le lemme des formes canoniques, $v = [\widehat{v_1, \ldots, v_n}]$. Par inversion on obtient que $\forall i, \Gamma \vdash v_i : t'$.
  Soient $\tau' = \text{Repr}(t')$ et $\tau = \tau'\ []$.
  Alors par induction $\forall i, m \vDash v_i : \tau'$ et $\tau' \rhd t'$. De la première propriété il vient (via S-ARRAY) $m \vDash v : \tau$, et de la seconde (via COMP-ARRAY) $\tau \rhd t$.
- $\{l_1 : t_1; \ldots; l_n : t_n\}$: Par le lemme des formes canoniques, $v = \{l_1 : \widehat{v_1; \ldots; l_n} : v_n\}$. Et par inversion, $\forall i, \Gamma \vdash v_i : t_i$.
  Soient $\tau_i = \text{Repr}(t_i)$ et $\tau = \{l_1 : \tau_1; \ldots; l_n : \tau_n\}$.
  Alors par induction, $\forall i, m \vDash v_i : \tau_i$ et $\tau_i \rhd t_i$.
  On déduit de S-STRUCT que $m \vDash v : \tau$, et de COMP-STRUCT que $\tau \rhd t$.
- $t = (t_1, \ldots, t_n) \to t'$: Par formes canoniques, on a $v = \text{fun}(\widehat{a_1, \ldots, a_n})\{i\}$.
  Soit $\tau = \text{FUN}_n$: par S-FUN on obtient que $m \vDash v : \tau$. On conclut d'autre part que $\tau \rhd t$ grâce à COMP-FUN.

□

Lemme 5.4 (Hauteur des chemins typés)**.** *Une valeur typée ne peut jamais pointer au dessus du niveau courant de pile. ($\mathcal{H}(\cdot)$ provient de la définition 4.6, page 46).*
Si $m \vDash v : \tau$, alors $\mathcal{H}(v) \leq |m|$.

Démonstration. On procède par induction sur la forme de v.

$\widehat{c}$**:** Alors $\mathcal{H}(v) = -1$. Comme $|m| \geq 0$, ce cas est établi.

$\widehat{f}$**:** Idem.

$\widehat{\&}\, a$**:** On distingue selon la forme de a. Si $a = (x)$, c'est immédiat. Si $a = (n, x)$, alors d'après la forme de v, la dernière règle appliquée dans la dérivation de $m \vDash v : \tau$ est S-PTR, et donc $m[(n, x)]$ est une valeur. D'après la définition de φ, $n \leq |m|$.

$\widehat{\&}\, \varphi.l$**:** On procède par induction sur $v' = \widehat{\&}\, \varphi$. Comme $\mathcal{H}(\widehat{\&}\, \varphi) \leq |m|$ et $\mathcal{H}(\widehat{\&}\, \varphi.l) = \mathcal{H}(\widehat{\&}\, \varphi)$, on en déduit que $\mathcal{H}(\widehat{\&}\, \varphi.l) \leq |m|$.

$\widehat{\&}\, \varphi[n]$**:** Idem.

$\{l_1 : \widehat{v_1; \ldots; l_n} : v_n\}$**:** Par induction, $\forall i \in [1; n], \mathcal{H}(v_i) \leq |m|$. Donc il en est de même pour leur maximum, et $\mathcal{H}(v) \leq |m|$.

$[\widehat{v_1, \ldots, v_n}]$**:** Idem.

□

Lemme 5.5 (Accès à des variables bien typées)**.** *Soit* Adr(φ) *l'adresse de la variable qui apparaît dans φ :*

$$\begin{aligned} \text{Adr}(a) &= a \\ \text{Adr}(\varphi.l) &= \text{Adr}(\varphi) \\ \text{Adr}(\varphi[n]) &= \text{Adr}(\varphi) \end{aligned}$$

Alors, si $\Gamma \vDash m$ *et* $\Gamma \vdash \varphi : t$*, alors* $\mathrm{Adr}(\varphi)$ *est soit une variable globale* (x) *avec* $x \in \mathrm{dom}(\Gamma_G)$*, soit une variable locale* $(|m|, x)$ *du plus haut cadre de pile avec* $x \in \mathrm{dom}(\Gamma_L)$.

Démonstration (esquisse). On procède par induction sur une dérivation de $\Gamma \vDash m$. □

Lemme 5.6 (Accès à une mémoire bien typée)**.** *Si* $\Gamma \vDash m$ *et* $\Gamma \vdash \varphi : t$*, alors* $m[\varphi]$ *est une valeur* v *et* $m \vDash v : \tau$ *où* $\tau \triangleright t$.

Démonstration. À partir du lemme 5.5, on prouve celui-ci par induction sur une dérivation de $\Gamma \vDash m$.

M-EMPTY : $\Gamma_G = \Gamma_L = [\,]$, la prémisse $\Gamma \vdash \varphi : t$ est donc impossible à satisfaire.

M-GLOBAL : Soient φ tel que $\Gamma, \text{global } x : t' \vdash \varphi : t$ et $m' = (s, ((x \mapsto v) :: g))$. Alors la variable référencée par φ est soit (x), soit (y) avec $y \in \mathrm{dom}(\Gamma_G)$, soit $(|m|, y)$ avec $y \in \mathrm{dom}(\Gamma_L)$.

Dans le premier cas, $m'[\varphi] = v$, ce qui permet de conclure.

Dans les autres cas, $m'[\varphi] = m[\varphi]$, ce qui nous permet de conclure grâce à l'hypothèse d'induction.

M-DECL : On part de $\Gamma \vdash \varphi : t'$. Alors $\mathrm{Adr}(\varphi)$ est soit la locale x, soit une autre variable locale, soit une globale. Dans le premier cas, $m'[\varphi] = v$ et les prémisses nous permettent de conclure. Dans tous les autres cas, $m'[\varphi] = m[\varphi]$ et on applique l'hypothèse d'induction.

M-DECLCLEAN : On suppose que $\Gamma \vdash \varphi : t'$. Alors $\Gamma, \text{local } x : t \vdash \varphi : t'$ par affaiblissement. On peut donc appliquer l'hypothèse d'induction : $m[\varphi] = v$ où $m \vDash v : \tau'$ avec $\tau' \triangleright t'$. On distingue alors selon la forme de v. Si $v = \widehat{\&}\,\varphi'$ où $\mathrm{Adr}(\varphi') = (|m|, x)$, alors $m'[\varphi] = \text{NULL}$ par l'opération $\mathrm{CleanVar}(\cdot,\cdot)$. Le type τ' étant un type pointeur par le lemme 5.2, on peut conclure. Dans les autres cas $m[\varphi] = m'[\varphi]$ ce qui termine ce cas.

M-PUSH : On procède d'une manière similaire. φ peut faire référence soit à un des x_i, auquel cas la valeur v_i convient, soit à une variable globale, auquel cas on applique l'hypothèse de récurrence.

M-POP : On part de $\Gamma \vDash m''$ où $m'' = \mathrm{Cleanup}(\mathrm{Pop}(m'))$. Deux cas se produisent selon la forme de $\mathrm{Adr}(\varphi)$.

- Soit $\mathrm{Adr}(\varphi) = (x)$ avec $x \in \mathrm{dom}(\Gamma_G)$, alors $\Gamma' \vdash \varphi : t$ où $\Gamma' = (\Gamma_G, [x_1 : t_1, \ldots, x_n : t_n, \underline{R} : t])$. On applique alors l'hypothèse de récurrence en partant du jugement $\Gamma \vDash m'$: il vient que $m'[\varphi] = v$ où $m \vDash v : \tau'$ avec $\tau' \triangleright t'$. Comme $\mathscr{H}(v) \leq |m'|$ (lemme 5.4), deux cas peuvent se produire :
  - Si $\mathscr{H}(v) = |m'|$, alors $m''[\varphi] = \text{NULL}$ et on a bien la compatibilité mémoire (l'argument est similaire au cas DECL-CLEAN).
  - Sinon, $m''[\varphi] = m'[\varphi]$ et on conclut directement.
- Soit $\mathrm{Adr}(\varphi) = (|m''|, x)$. On procède alors de la même manière sauf qu'on invoque alors le cas d'induction sur $\Gamma \vDash m$.

M-WRITE : On part de $\Gamma \vDash m'$ où $m' = m[\varphi \leftarrow v]$, et on suppose que $\Gamma \vdash \varphi' : t'$.

Si $\varphi = \varphi'$: alors il suffit d'appliquer GETPUT à la lentille $\Phi : m[\varphi \leftarrow m[\varphi]] = m$, ce qui donne directement la conclusion.

Si $\varphi \neq \varphi' : \Gamma \vdash \varphi' : t'$ donc $\mathrm{Adr}(\varphi')$ est soit une locale soit une globale de m'. Donc $m'[\varphi'] = m[\varphi']$ et on conclut grâce à l'hypothèse d'induction. □

5.9 Progrès et préservation

Ces lemmes étant établis, on énonce maintenant le théorème de progrès. Contrairement aux langages où tout est expression, il faut traiter séparément les trois constructions principales de SAFESPEAK : les expressions, les valeurs gauches et les instructions. Celles-ci sont mutuellement dépendantes car :

- la définition d'une fonction par un bloc est une expression ;
- une expression est un cas particulier d'instruction ;
- une valeur gauche peut convenir une expression en indice de tableau ;
- une valeur gauche est un cas particulier d'expression.

Théorème 5.1 (Progrès)**.** *Supposons que* $\Gamma \vdash i$. *Soit* m *un état mémoire tel que* $\Gamma \vDash m$.
Alors l'un des cas suivants est vrai :

- $i = \text{PASS}$
- $\exists v, i = \text{RETURN}(v)$
- $\exists (i', m'), \langle i, m\rangle \to \langle i', m'\rangle$
- $\exists \Omega \in \{\Omega_{div}, \Omega_{array}, \Omega_{ptr}\}, \langle i, m\rangle \to \Omega$

✧ ✧ ✧

Supposons que $\Gamma \vdash e : t$. *Soit* m *un état mémoire tel que* $\Gamma \vDash m$. *Alors l'un des cas suivants est vrai :*

- $\exists v \neq \Omega, e = v$
- $\exists (e', m'), \langle e, m\rangle \to \langle e', m'\rangle$
- $\exists \Omega \in \{\Omega_{div}, \Omega_{array}, \Omega_{ptr}\}, \langle e, m\rangle \to \Omega$

✧ ✧ ✧

Supposons que $\Gamma \vdash l\,v : t$. *Soit* m *un état mémoire tel que* $\Gamma \vDash m$.
Alors l'un des cas suivants est vrai :

- $\exists \varphi, l\,v = \varphi$
- $\exists (l\,v', m'), \langle l\,v, m\rangle \to \langle l\,v', m'\rangle$
- $\exists \Omega \in \{\Omega_{div}, \Omega_{array}, \Omega_{ptr}\}, \langle l\,v, m\rangle \to \Omega$

C'est-à-dire, soit :

- l'entité (instruction, expression ou valeur gauche) est complètement évaluée.
- un pas d'évaluation est possible.
- une erreur de division, tableau ou pointeur se produit.

La preuve du théorème 5.1 se trouve en annexe D.2.

Théorème 5.2 (Préservation)**.** *Soient* Γ *un environnement de typage, et* m *un état mémoire tels que* $\Gamma \vDash m$.

Alors :

- *Si* $\Gamma \vdash l\,v : t$ *et* $\langle l\,v, m\rangle \to \langle \varphi, m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $m' \vDash_\Phi \varphi : \tau$ *où* $\tau \rhd t$.
- *Si* $\Gamma \vdash l\,v : t$ *et* $\langle l\,v, m\rangle \to \langle l\,v', m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $\Gamma \vdash l\,v' : t$.
- *Si* $\Gamma \vdash e : t$ *et* $\langle e, m\rangle \to \langle v, m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $m' \vDash v : \tau$ *où* $\tau \rhd t$.
- *Si* $\Gamma \vdash e : t$ *et* $\langle e, m\rangle \to \langle e', m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $\Gamma \vdash e' : t$.
- *Si* $\Gamma \vdash i$ *et* $\langle i, m\rangle \to \langle i', m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $\Gamma \vdash i'$.

Autrement dit, si une construction est typable, alors un pas d'évaluation ne modifie pas son type et préserve le typage de la mémoire.

Remarque Dans la formulation classique de ce théorème, on indique que $\Gamma \vDash m$ implique $\Gamma \vDash m'$. Ici, la conclusion est moins forte en indiquant seulement que $\Gamma \vDash \text{Cleanup}(m')$. Cela indique que la compatibilité mémoire est établie mais peut localement introduire des pointeurs fous. En fait, comme une étape de Cleanup$(\cdot)$ est faite après chaque appel de fonction et chaque déclaration, la propriété classique est vraie mais uniquement sur un plus grand pas d'exécution.

La preuve de ce théorème se trouve en annexe D.3.

Cela établit qu'aucun terme ne reste « bloqué » parce qu'aucune règle ne s'applique, et que la sémantique respecte le typage. En quelque sorte, les types sont un contrat entre les expressions et les fonctions : si leur évaluation converge, alors une valeur du type inféré sera produite.

Enfin, on donne une version de ces propriétés pour les phrases de programme.

Théorème 5.3 (Progrès pour les phrases)**.** *Soient* Γ *un environnement de typage,* m *un état mémoire et* p *une phrase de programme. Supposons que* $\Gamma \vdash p \to \Gamma'$ *et* $\Gamma \vDash m$.

On suppose en outre que l'évaluation de p *termine.*

Alors $\exists m'.\, m \Vdash p \to m'$.

Démonstration. Ici il n'y a pas de difficulté puisque la contrainte (forte) de terminaison se lit $\langle e, m\rangle \to \langle v', m'\rangle$ où e est l'expression apparaissant dans p.

Selon la forme de p, il suffit alors d'appliquer la règle ET-EXP ou ET-VAR.

□

Théorème 5.4 (Préservation pour les phrases)**.** *On suppose que les trois propriétés suivantes sont vérifiées :*

$$\begin{cases} \Gamma \vDash m \\ \Gamma \vdash p \to \Gamma' \\ m \Vdash p \to m' \end{cases}$$

Alors $\Gamma' \vDash m'$.

Démonstration. On distingue selon la dernière règle appliquée dans la dérivation de $m \Vdash p \to m'$.

ET-EXP : La dernière règle appliquée pour dériver $\Gamma \vdash p \to \Gamma'$ est donc T-EXP. D'après les prémisses de ces deux règles, on a donc $\Gamma \vdash e : t$ et $\langle e, m\rangle \to \langle v, m'\rangle$. Alors, d'après le théorème de préservation, $\Gamma \vDash m'$.

ET-VAR : Ici, la dérivation de $\Gamma \vdash p \to \Gamma'$ termine par T-VAR. D'après leurs prémisses, on a donc : $\Gamma \vdash e : t$, $\Gamma' = \Gamma, \text{global } x : t$, et $m'' = (s, (x \mapsto v) :: g)$ où $(s, g) = m'$ (on cherche à prouver que $\Gamma' \vDash m''$).

En appliquant le théorème de préservation, on obtient que $\Gamma \vdash v : t$ et $\Gamma \vDash m'$. D'après le lemme 5.3, il existe τ tel que $m' \vDash v : \tau$ où $\tau \rhd t$. On peut alors appliquer M-GLOBAL qui nous donne que $\Gamma' \vDash m''$.

□

Conclusion

En ajoutant un système de types statiques à SAFESPEAK, on peut calculer à la compilation la forme des valeurs produite par chaque expression. Pour ce faire, on a défini un ensemble de règles de typage (regroupées dans l'annexe C) à appliquer selon la forme de celle-ci.

Si on considère des programmes qui sont seulement syntaxiquement corrects, on ne peut rien prédire sur leur exécution. Par exemple, fun(x){PASS}+1 est une expression correcte mais pour laquelle il n'y a pas de règle d'évaluation qui s'applique. En ajoutant un système de types, les propriétés de sûreté établies dans ce chapitre assurent que les termes peuvent être évalués, et que les valeurs produites sont en accord avec les types donnés aux différentes parties du programme. Cela permet surtout de s'assurer que les programmes ne peuvent provoquer une erreur d'exécution que dans certains cas particuliers, comme les divisions ou les accès aux tableaux.

À l'issue de ce chapitre, on a donc un langage impératif sain pour bâtir des analyses de typage, ce que nous allons faire dans le chapitre suivant.

CHAPITRE 6

EXTENSIONS DE TYPAGE

Nous venons de définir un système de types sûrs dans le chapitre 5. Cela permet de mettre en relation les types des expressions avec les valeurs qui leur seront associées. Cela permet une forme d'analyse de flot : si peu de constructions permettent de créer des valeurs d'un type t, alors toutes les valeurs de type t proviennent de ces « sources ».

On se propose ici d'enrichir le système de types de plusieurs extensions permettant d'explorer cette idée, en ajoutant de la « signification » dans les types de données des programmes. Ces extensions permettront de détecter des erreurs de programmation communes, appuyées sur des exemples réels.

Cela revient à introduire une séparation entre le type des données et sa représentation, c'est-à-dire définir un type abstrait. Les pointeurs utilisateur sont en fait des pointeurs classiques déguisés, pour lesquels on interdit l'opérateur de déréférencement.

Cette technique est en fait générique : on peut également l'appliquer à certains types d'entiers. En C, il est commun d'utiliser des `int` pour tout et n'importe quoi : pour des entiers bien sûr (au sens de $\mathbb{Z}$), mais aussi comme identificateurs pour lesquels les opérations usuelles comme l'addition n'ont pas de sens. Par exemple, sous Linux, l'opération d'ouverture de fichier renvoie un entier, dit *descripteur de fichier*, qui identifie ce fichier pour ce processus. Le langage autorise donc par exemple de multiplier entre eux deux descripteurs de fichiers, mais le résultat n'a pas de raison *a priori* d'être un descripteur de fichier valide.

En n'offrant pas cette distinction, le langage C permet d'écrire du code qui peut s'exécuter mais dont la sémantique n'est, quelque part, pas bien fondée. En effet, le système de types de C est trop primitif pour pouvoir garantir une véritable isolation entre deux types de même représentation : il n'y a pas de types abstraits. Certes, `typedef` permet d'introduire un nouveau nom pour un type, mais ce n'est qu'un raccourci syntaxique. Le compilateur ne peut en effet pas considérer un programme sans avoir la définition quasi-complète des types qui y apparaissent. La seule exception concerne les pointeurs sur structures : si on ne fait que les affecter, il n'est pas nécessaire de connaître la taille ni la disposition de la structure, donc la définition peut ne pas être visible. Cette technique, connue sous le nom de *pointeurs opaques*, n'est pas applicable aux autres types.

En ajoutant une couche de typage, on interdit ces opérations à la compilation. Cela permet deux choses : pour le code déjà écrit, de détecter et corriger les manipulations dangereuses ; et, pour le nouveau code, de s'assurer qu'il est correct. Par exemple, si on écrit un éditeur de texte, on peut éviter de nombreuses erreurs de programmation en définissant un type « indice de ligne » et un type « indice de colonne » incompatibles entre eux.

Un premier exemple permet de distinguer plusieurs utilisations des entiers, selon s'ils sont utilisés comme entiers arithmétiques ou ensemble de bits. Cela permet de détecter une

erreur courante qui consiste à mélanger les opérateurs logiques et bit à bit.

Ensuite, on étend de manière indépendante le système de types, cette fois au niveau des pointeurs. Plus précisément, dans le contexte des systèmes d'exploitation, introduit une différence entre les pointeurs dont la valeur est contrôlée par l'utilisateur et les autres.

6.1 Exemple préliminaire : les entiers utilisés comme bitmasks

Dans le langage C, les types de données décrivent uniquement l'agencement en mémoire des valeurs. Ils n'ont pas de signification plus sémantique permettant d'exprimer ce que les données représentent. Par exemple, dans un programme manipulant des dates, on sera amené à manipuler des numéros de mois et d'années, représentés par des types entiers. Le langage C permet de définir des nouveaux types :

```
typedef int month_t;
typedef int year_t;
```

Cependant, rien ne distingue le nouveau type de l'ancien. Il ne s'agit que d'une aide à la documentation. Dans cet exemple, `month_t` et `year_t` sont tous les deux des nouveaux noms pour le type `int`, donc ils sont en fait compatibles. Le compilateur ne peut donc pas détecter qu'on utilise un numéro de mois là où un numéro d'année était attendu (ou *vice versa*).

Cet idiome est commun en C. On manipule notamment certaines données abstraites par des clés entières, et un `typedef` particulier permet de désigner celles-ci. Par exemple sous Linux, les numéros de processus sont des indices dans la table de processus interne au noyau, et on y accède par une valeur de type `pid_t`. De même, les utilisateurs sont représentés par un nombre entier du type `uid_t`.

Un autre idiome est répandu : l'utilisation d'entiers comme représentation d'un ensemble de booléens. En effet, un nombre $a = \sum_{i=0}^{N-1} a_i 2^i$ peut s'interpréter comme l'ensemble de ses bits égaux à 1 : $\{i \in [0; N-1] / a_i = 1\}$. Un entier de 32 bits peut donc représenter une combinaison de 32 options indépendantes.

C'est de cette manière que fonctionne l'interface qui permet d'ouvrir un fichier sous Unix (figure 6.1). Le paramètre `flags` est un entier qui encode les options liées à l'ouverture du fichier. On précise son mode (lecture, écriture ou les deux) par les bits 1 et 2, s'il faut créer le fichier ou non s'il n'existe pas par le bit 7, si dans ce cas il doit être effacé par le bit 8, etc. On obtient le paramètre complet en réalisant un « ou » bit à bit entre des constantes. Le paramètre `mode` encode de la même manière les permissions que doit avoir le fichier créé, le cas échéant (`mode_t` désigne en fait `unsigned int`).

```
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

FIGURE 6.1: Interface permettant d'ouvrir un fichier sous Unix

Ces fonctions retournent un entier, qui est un *descripteur de fichier*. Il correspond à un indice numérique dans une table interne au processus. Par exemple, 0 désigne son entrée standard, 1 sa sortie standard, et 2 son flux d'erreur standard.

On identifie donc au moins trois utilisations du type `int` :

- entier : c'est l'utilisation classique pour représenter des valeurs numériques. Toutes les opérations sont possibles.

- bitmask : on utilise un entier comme ensemble de bits. Seules les opérations bit à bit ont du sens.
- entier opaque : on utilise un entier de manière purement abstraite. C'est l'exemple des descripteurs de fichier.

Ces utilisations du type n'ont rien à voir ; il faudrait donc empêcher d'utiliser un descripteur de fichier comme un mode, et vice-versa. De même, aucun opérateur n'a de sens sur les descripteurs de fichier, mais l'opérateur `|` du « ou » bit à bit doit rester possible pour les modes.

On décrit ici une technique de typage pour détecter et interdire ces mauvaises utilisations en proposant une version « bien typée » de la fonction `open`. Plus précisément, on donne à ses deuxième et troisième arguments (respectivement `flags` et `mode`) le nouveau type BITS qui correspond aux entiers utilisés comme bitmasks. Le type de retour n'est pas modifié (il reste INT), mais on décrit comment il est possible de rendre ce type opaque.

6.1.1 Modifications

On commence par ajouter deux types : d'une part BITS bien sûr, mais également CHAR qui apparaît dans les chaînes de caractères. On ne spécifie pas plus ce dernier mais on suppose qu'il existe des littéraux de chaînes qui retournent un pointeur vers le premier élément d'un tableaux de caractères. Pour rester compatible avec C, on suppose qu'un caractère nul `'\0'` est inséré à la fin de la chaîne. On ajoute ces chaînes uniquement dans le but de pouvoir représenter les noms de fichiers.

Au niveau des valeurs, les entiers utilisés comme bitmasks sont représentés par des valeurs entières classiques $\widehat{n}$. En particulier, on n'ajoute pas de nouveau type sémantique, mais on ajoute une règle de compatibilité entre le type de valeurs INT et le type statique BITS (cela signifie qu'une valeur de type BITS est représentée par un INT en mémoire, figure 6.2). Par ailleurs, on change le type des « constructeurs » (`O_RDONLY`, `O_RDWR`, `O_APPEND`, ...) et du « consommateur » `open` (figure 6.3).

Type

$t ::= \ldots$

|	CHAR	Caractère
|	BITS	Entier utilisé comme *bitmask*

$$\frac{}{\text{INT} \rhd \text{BITS}}\ (\text{COMP-BITS})$$

FIGURE 6.2: Ajouts liés aux entiers utilisés comme bitmasks

Pour que les opérations bit à bit puissent s'appliquer aux bitmasks, on remplace aux règles s'appliquant à INT les règles suivantes. Cela revient à surcharger le typage de l'opérateur ~, mais la sémantique est la même quelque soit le type car BITS et INT sont représentés de la même manière.

$$\frac{\boxplus \in \{ \mid , \&, \hat{} \} \qquad \Gamma \vdash e_1 : \text{BITS} \qquad \Gamma \vdash e_2 : \text{BITS}}{\Gamma \vdash e_1 \boxplus e_2 : \text{BITS}}\ (\text{OP-BITS}) \qquad \frac{\Gamma \vdash e : \text{BITS}}{\Gamma \vdash \sim e : \text{BITS}}\ (\text{NOT-BITS})$$

Il reste à permettre d'utiliser les bitmasks dans les contextes où on attend un entier. Par exemple, pour écrire IF(x & `0x80`){...}ELSE{...} (test du bit numéro 7). On veut donc exprimer

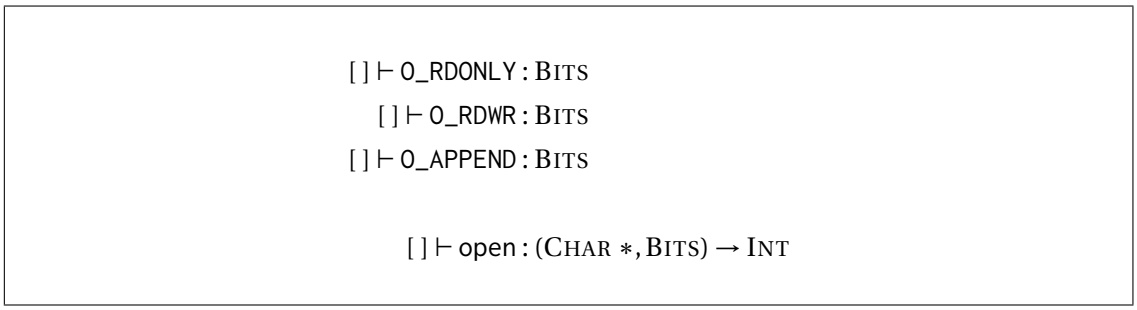


FIGURE 6.3: Nouvelles valeurs liées aux bitmasks

que « un BITS est un INT ». Cette relation entre différents types d'entier correspond à un cas particulier de sous-typage.

On ajoute la règle suivante. Elle permet d'utiliser une expression de type BITS là où une expression de type INT est attendue.

$$\frac{\Gamma \vdash e : \text{BITS}}{\Gamma \vdash e : \text{INT}} \text{ (SUB-BITSINT)}$$

Cela modifie légèrement l'implantation de l'inférence de types. Le type d'une expression utilisée comme opérande de l'opérateur + n'est donc pas *a priori* de type INT, mais BITS ou INT. Cela implique aussi qu'on peut additionner un BITS et un INT pour obtenir un INT.

Ainsi, si $\Gamma \vdash e$: BITS, on a par exemple $\Gamma \vdash !\, e$: INT. On rappelle que la règle permettant de typer ! est inchangée et reste :

$$\frac{\boxminus \in \{\sim, !\} \qquad \Gamma \vdash e : \text{INT}}{\Gamma \vdash \boxminus\, e : \text{INT}} \text{ (UNOP-NOT)}$$

6.1.2 Exemple : ! x & y

Les nombreux opérateurs de C (repris en SAFESPEAK) posent plusieurs problèmes :

- il sont nombreux et il est facile de confondre && avec &, ou ! avec ~ ;
- il y a un opérateur « ou exclusif » bit à bit (^) mais pas d'équivalent logique ;
- la priorité des opérateurs semble parfois arbitraire. Par exemple, les opérateurs de décalage sont plus prioritaires que les additions, donc $x \ll 2 + 1$ est interprété comme $(x \ll 2) + 1$.

Le premier et le dernier point permettent d'expliquer une erreur courante : celle qui consiste à écrire ! x & y au lieu de ! (x & y).

En effet, la première expression est équivalente à (! x) & y. Comme ! x vaut 0 ou 1, l'expression résultante vaut y & 1 si $x = 0$, ou 0 sinon. Il s'agit probablement d'une erreur de programmation. L'alternative ! (x & y) a plus de sens : elle vaut 0 si x et y ont un bit en commun, 1 sinon.

On vérifie enfin que la première n'est pas bien typée alors que la seconde l'est. Dans les deux cas suivants on se place dans un environnement Γ comportant deux variables globales x et y de type BITS. Alors (! x)&y n'est pas bien typée. En effet, $\Gamma \vdash !\, x$: INT et la seule règle qui s'applique à l'opérateur & ne peut pas s'appliquer. En revanche la seconde est bien typée (figure 6.4).

$$\Gamma = ([x \mapsto \textsc{Bits}, y \mapsto \textsc{Bits}], [\,])$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x : \textsc{Bits} \qquad \Gamma \vdash y : \textsc{Bits}}{\Gamma \vdash x \,\&\, y : \textsc{Bits}}\ (\textsc{Op-Bits})}{\Gamma \vdash x \,\&\, y : \textsc{Int}}\ (\textsc{Sub-BitsInt})}{\Gamma \vdash !\,(x \,\&\, y) : \textsc{Int}}\ (\textsc{Unop-Not})$$

FIGURE 6.4: Dérivation montrant que $!\,(x \,\&\, y)$ est bien typée

Cet exemple préliminaire permet de voir en quoi SAFESPEAK est adapté à des analyses de typage légères. Puisque le typage est sûr, on en déduit que les valeurs d'un certain type ne peuvent être créées que par un certain nombre de constructeurs. Par exemple ici les bitmasks ne proviennent que de combinaisons de constantes. C'est précisément cette idée de détection de source qui est au cœur de l'analyse suivante.

6.2 Analyse de provenance de pointeurs

Jusqu'ici SAFESPEAK est un langage impératif généraliste, ne prenant pas en compte les spécificités de l'adressage utilisé dans les systèmes d'exploitation.

Dans cette section, on commence par l'étendre en ajoutant des constructions modélisant les variables présentes dans l'espace utilisateur (cf. chapitre 2). Pour accéder à celles-ci, on ajoute un opérateur de déréférencement sûr qui vérifie à l'exécution que l'invariant suivant est respecté :

> Les pointeurs dont la valeur est contrôlée par l'utilisateur pointent vers l'espace utilisateur.

La terminologie mérite d'être détaillée :

Un pointeur contrôlé par l'utilisateur, ou *pointeur utilisateur*, est une référence mémoire dont la valeur est modifiable par le code utilisateur (opposé au code noyau, que nous analysons ici). Ceci correspond à des données provenant de l'extérieur du système vérifié. C'est une propriété statique, qui peut être déterminée à la compilation à partir de considérations syntaxiques. Par exemple, l'adresse d'une variable locale au sein de code noyau est toujours considérée comme étant contrôlée par le noyau.

Un pointeur pointant vers l'espace utilisateur fait référence à une variable allouée en espace utilisateur. Cela veut dire qu'y accéder ne risque pas de mettre en péril l'isolation du noyau en faisant fuiter des informations confidentielles ou en déjouant son intégrité. Cette propriété est dynamique : un pointeur utilisateur peut *a priori* pointer vers l'espace utilisateur, ou non.

Pour prouver que l'invariant précédent est bien respecté, on procède en plusieurs étapes.

Tout d'abord, on définit une nouvelle erreur Ω_{sec} (pour « sécurité »), déclenchée lorsqu'un pointeur contrôlé par l'utilisateur et pointant vers le noyau est déréférencé (le cas que l'on cherche à éviter). Il est important de noter que ce cas d'erreur est « virtuel », c'est-à-dire qu'on l'ajoute à la sémantique pour pouvoir le détecter facilement comme un cas d'erreur, mais dans une sémantique de plus bas niveau, comme en C, l'erreur ne serait pas déclenchée. D'un point de vue opérationnel, cela équivaut à ajouter un test dynamique à chaque déréférencement, ce qui est sûr mais se paye en performances. Ajouter ce cas d'erreur virtuel

dans la sémantique d'évaluation permet de transformer un problème de sécurité (empêcher les fuites d'information) en problème de sûreté (empêcher les erreurs à l'exécution).

Ensuite, on montre qu'avec cet ajout, si on étend naïvement le système de types en donnant le même type aux pointeurs contrôlés par l'utilisateur et le noyau, le théorème de progrès (5.1) n'est plus valable. Cela signifie que le système de types classique présenté dans le chapitre 5 ne suffit pas à capturer les propriétés de sécurité que nous voulons interdire.

L'étape suivante est d'étendre, à son tour, le système de types de SAFESPEAK en distinguant les types des pointeurs contrôlés par l'utilisateur des pointeurs contrôlés par le noyau. Puisqu'on veut interdire le déréférencement des premiers par l'opérateur $*$, on introduit les constructions copy_from_user et copy_to_user qui réaliseront le déréférencement sûr de ces pointeurs.

Enfin, une fois ces modifications faites, on prouve que les propriétés de progrès et de préservation sont rétablies.

6.2.1 Extensions noyau pour SAFESPEAK

On ajoute à SAFESPEAK la notion de valeur provenant de l'espace utilisateur. Pour marquer la séparation entre les deux espaces d'adressage, on ajoute une construction $\varphi ::= \widehat{\Diamond}\,\varphi'$. Le chemin interne φ' désigne une variable classique (un pointeur noyau) et l'opérateur $\widehat{\Diamond}\,\cdot$ permet de l'interpréter comme un pointeur vers l'espace utilisateur. En quelque sorte, on ne classifie pas les valeurs selon la variable pointée mais selon la construction du pointeur.

Remarquons qu'on n'introduit pas de sous-typage : les pointeurs noyau ne peuvent être utilisés qu'en tant que pointeurs noyau, et les pointeurs utilisateurs qu'en tant que pointeurs utilisateurs.

En plus du déréférencement par $*$ (qui devra donc renvoyer Ω_{sec} pour les valeurs de la forme $\widehat{\Diamond}\,\varphi'$), il faut aussi ajouter des constructions de lecture et d'écriture à travers les pointeurs utilisateurs. Ceci sera fait sous forme de deux fonctions, copy_from_user et copy_to_user. Celles-ci prennent deux pointeurs en paramètre et renvoient un booléen indiquant si la copie a pu être faite (si le paramètre contrôlé par l'utilisateur pointe en espace noyau, les fonctions ne font pas la copie et signalent l'erreur).

Illustrons ceci par un exemple. Imaginons un appel système fictif qui renvoie la version du noyau, en remplissant par pointeur une structure contenant les champs entiers `major`, `minor` et `patch` (un équivalent dans Linux est l'appel système `uname()`). Celui-ci peut être alors écrit comme dans la figure 6.5. Une fois la structure noyau v remplie, il faut la copier vers l'espace utilisateur. La fonction copy_to_user va réaliser cette copie (de la même manière qu'avec un `memcpy()`), mais après avoir testé dynamiquement que p pointe en espace utilisateur (dans le cas contraire, la copie n'est pas faite).

```
sys_getver = fun(p){
        DECL v = { major : 3; minor : 14; patch : 15 } IN
        copy_to_user(p, &v)
}
```

FIGURE 6.5: Implantation d'un appel système qui remplit une structure par pointeur

On peut remarquer que, contrairement aux fonctions présentes dans le noyau Linux, les fonctions copy_from_user et copy_to_user n'ont pas de paramètre indiquant la taille à copier.

Expressions	$e ::= \ldots$	
	$\mid \diamondsuit\, l\, v$	Adresse utilisateur
	$\mid$ copy_from_user(e_d, e_s)	Lecture depuis l'espace utilisateur
	$\mid$ copy_to_user(e_d, e_s)	Écriture vers l'espace utilisateur
Contextes	$C ::= \ldots$	
	$\mid \diamondsuit\, C$	
	$\mid$ copy_from_user(C, e)	
	$\mid$ copy_from_user(v, C)	
	$\mid$ copy_to_user(C, e)	
	$\mid$ copy_to_user(v, C)	
Chemins	$\varphi ::= \ldots$	
	$\mid \widehat{\diamondsuit}\, \varphi$	Pointeur utilisateur
Erreurs	$\Omega ::= \ldots$	
	$\mid \Omega_{sec}$	Erreur de sécurité

FIGURE 6.6: Ajouts liés aux pointeurs utilisateurs (par rapport à l'interprète du chapitre 4)

Cela est dû au fait que le modèle mémoire de SAFESPEAK est de plus haut niveau. L'information de taille est déjà présente dans chaque valeur.

Une autre remarque à faire est qu'il n'y a pas de manière de copier des données de l'espace utilisateur vers l'espace utilisateur. Il est nécessaire de passer par l'espace noyau. La raison est que, puisqu'il faut réaliser deux tests dynamiques, les erreurs peuvent arriver à ces deux endroits. Plutôt que de proposer un opérateur qui réalise cette copie, on laisse le programmeur faire les deux copies manuellement.

On commence donc par ajouter aux instructions des constructions copy_to_user$(\cdot,\cdot)$ et copy_from_user$(\cdot,\cdot)$ de copie sûre. copy_from_user(p_k, p_u) copie la valeur par pointée par p_u (qui se trouve en espace utilisateur) à l'emplacement mémoire pointé par p_k (en espace noyau). copy_to_user(p_u, p_k) réalise l'opération inverse, en copiant la valeur par pointée par p_k (en espace noyau) à l'emplacement mémoire pointé par p_u (en espace utilisateur).

Afin de leur donner une sémantique, il faut étendre l'ensemble des valeurs pointeur φ aux constructions de la forme $\widehat{\diamondsuit}\, \varphi'$. Pour créer des termes s'évaluant en de telles valeurs, il faut une construction syntaxique $\diamondsuit\, e$ telle que, si e s'évalue en $\widehat{\&}\, \varphi$, $\diamondsuit\, e$ s'évalue en $\widehat{\&}\, \widehat{\diamondsuit}\, \varphi$. Cela demande 2 autres ajouts : un nouveau contexte d'évaluation $\diamondsuit\, C$ et une règle d'évaluation. Enfin, on ajoute une nouvelle erreur Ω_{sec} à déclencher lorsqu'on déréférence directement un pointeur utilisateur. Ces étapes sont résumées dans la figure 6.6.

En résumé, on a deux constructions pour créer des pointeurs à partir d'une valeur gauche : $\& \cdot$ crée un pointeur noyau, et $\diamondsuit\, \cdot$ crée un pointeur utilisateur. Seule la première est faite pour être utilisée dans le code à analyser. La seconde sert uniquement à modéliser les points d'entrée du noyau. Par exemple, la fonction sys_getver de la figure 6.5 peut être appelée par un utilisateur de la manière décrite dans la figure 6.7.

DECL $v = \{$ `major:0;minor:0;patch:0` $\}$ IN
sys_getver($\diamondsuit\ v$)

FIGURE 6.7: Appel de la fonction sys_getver de la figure 6.5

6.2.2 Extensions sémantiques

En ce qui concerne l'évaluation des expressions $\diamondsuit\ \cdot$, on ajoute la règle suivante :

$$\frac{}{\langle \diamondsuit\ \varphi, m\rangle \rightarrow \langle \widehat{\&}\ (\widehat{\diamondsuit}\ \varphi), m\rangle}\ \text{(PHI-USER)}$$

Dans $\widehat{\&}\ (\widehat{\diamondsuit}\ \varphi)$, l'opérateur $\widehat{\&}\ \cdot$ indique que la valeur créée est une référence mémoire. Cette référence mémoire, $\widehat{\diamondsuit}\ \varphi$, est contrôlée par l'utilisateur. C'est ce qu'indique le constructeur $\widehat{\diamondsuit}\ \cdot$

Ensuite, il est nécessaire d'adapter les règles d'accès à la mémoire pour déclencher une erreur Ω_{sec} en cas de déréférencement d'un pointeur utilisateur. Les accès mémoire en lecture proviennent de la règle EXP-LV et ceux en lecture, de la règle EXP-SET, rappellées ici :

$$\frac{}{\langle \varphi, m\rangle \rightarrow \langle m[\varphi]_\Phi, m\rangle}\ \text{(EXP-LV)} \qquad \frac{}{\langle \varphi \leftarrow v, m\rangle \rightarrow \langle v, m[\varphi \leftarrow v]_\Phi\rangle}\ \text{(EXP-SET)}$$

Les accès à la mémoire sont en effet faits par le biais de la lentille Φ. Il suffit donc d'adapter sa définition (page 52) de celle-ci en rajoutant les cas suivant :

$$\text{get}_\Phi(\widehat{\diamondsuit}\ \varphi) = \Omega_{sec}$$
$$\text{put}_\Phi(\widehat{\diamondsuit}\ \varphi, v) = \Omega_{sec}$$

Enfin, il est nécessaire de donner une sémantique aux fonctions copy_from_user et copy_to_user. L'idée est que celles-ci testent dynamiquement la *valeur* du paramètre contrôlé par l'utilisateur afin de vérifier que celui-ci pointe vers l'espace utilisateur (c'est-à-dire, qu'il est de la forme $\widehat{\diamondsuit}\ \varphi$).

Deux cas peuvent se produire. Soit la partie à vérifier a la forme $\widehat{\diamondsuit}\ \varphi'$, soit non (et dans ce cas $\nexists \varphi', \varphi = \widehat{\diamondsuit}\ \varphi'$). Dans le premier cas (règles USER-*-OK), alors la copie est faite et l'opération de copie retourne la valeur entière 0. Dans le second (règles USER-*-ERR), aucune copie n'est faite et la valeur -1 est retournée. Ce comportement est calé celui des fonctions `copy_{from,to}_user` du noyau Linux : en cas de succès elles renvoient 0, et en cas d'erreur `-EFAULT` ($= -14$).

$$\frac{v = m[\varphi_s]_\Phi \qquad m' = m[\varphi_d \leftarrow v]_\Phi}{\langle \text{copy\_from\_user}(\widehat{\&}\ \varphi_d, \widehat{\&}\ (\widehat{\diamondsuit}\ \varphi_s)), m\rangle \rightarrow \langle 0, m'\rangle}\ \text{(USER-GET-OK)}$$

$$\frac{\nexists\ \varphi_s.\varphi = \widehat{\diamondsuit}\ \varphi_s}{\langle \text{copy\_from\_user}(\widehat{\&}\ \varphi_d, \widehat{\&}\ \varphi), m\rangle \rightarrow \langle -14, m\rangle}\ \text{(USER-GET-ERR)}$$

$$\frac{v = m[\varphi_s]_\Phi \qquad m' = m[\varphi_d \leftarrow v]_\Phi}{\langle \text{copy\_to\_user}(\widehat{\&}\ (\widehat{\diamondsuit}\ \varphi_d), \widehat{\&}\ \varphi_s), m\rangle \rightarrow \langle 0, m'\rangle}\ \text{(USER-PUT-OK)}$$

$$\frac{\nexists\ \varphi_d.\varphi = \widehat{\diamondsuit}\ \varphi_d}{\langle \text{copy\_to\_user}(\widehat{\&}\ \varphi, \widehat{\&}\ \varphi_s), m\rangle \rightarrow \langle -14, m\rangle}\ \text{(USER-PUT-ERR)}$$

Ces règles sont à appliquer en priorité de la règle d'appel de fonction classique, puisqu'il s'agit d'élements de syntaxe différents. En effet ces « fonctions » ne sont pas implantables directement en SAFESPEAK, puisqu'il n'y a pas par exemple d'opérateur permettant d'extraire φ depuis une valeur $\widehat{\diamondsuit}\,\varphi$. L'opération en « boîte noire » de ces deux fonctions permet d'assurer que l'accès à l'espace utilisateur est toujours couplé à un test dynamique.

6.2.3 Insuffisance des types simples

Étant donné SAFESPEAK augmenté de cette extension sémantique, on peut étendre naïvement le système de types avec la règle suivante :

$$\frac{\Gamma \vdash lv : t}{\Gamma \vdash \diamondsuit\, lv : t*}\ \text{(ADDR-USER-IGNORE)}$$

Cette règle est compatible avec l'extension, sauf qu'elle introduit des termes qui sont bien typables mais dont l'évaluation provoque une erreur $\Omega_{sec} \notin \{\Omega_{div}, \Omega_{array}, \Omega_{ptr}\}$, violant ainsi le théorème 5.1. Posons :

$$\begin{cases} e = *\diamondsuit\, x \\ \Gamma = x : \text{INT} \\ m = ([[x \mapsto 0]], [\,]) \end{cases}$$

Les hypothèses du théorème de progrès sont bien vérifiées, mais cependant la conclusion n'est pas vraie :

- On a bien $\Gamma \vDash m$. En effet :

$$\dfrac{\dfrac{}{[\,] \vDash ([\,], [\,])}\ \text{(M-EMPTY)} \qquad [\,] \vDash 0 : \text{INT} \qquad \text{INT} \rhd \text{INT}}{x : \text{INT} \vDash ([[x \mapsto 0]], [\,])}\ \text{(M-PUSH)}$$

- e est bien typée sous Γ :

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \text{INT} \in \Gamma}{\Gamma \vdash x : \text{INT}}\ \text{(LV-VAR)}}{\Gamma \vdash \&x : \text{INT}*}\ \text{(LV-DEREF)}}{\Gamma \vdash \diamondsuit\, x : \text{INT}*}\ \text{(ADDR-USER-IGNORE)}}{\Gamma \vdash *\diamondsuit\, x : \text{INT}}\ \text{(LV-DEREF)}$$

- L'évaluation de e sous m provoque une erreur différente de Ω_{div}, Ω_{array}, ou Ω_{ptr} :

$$\dfrac{\dfrac{\overline{m[\widehat{\diamondsuit}\, x] = \Omega_{sec}}}{\langle *\diamondsuit\, x, m\rangle \to \langle \Omega_{sec}, m\rangle}\ \text{(EXP-LV)} \qquad \dfrac{}{\langle \Omega_{sec}, m\rangle \to \Omega_{sec}}\ \text{(EVAL-ERR)}}{\langle e, m\rangle \to \Omega_{sec}}$$

Cela montre que le typage n'apporte plus de garantie de sûreté sur l'exécution : le système de types naïvement étendu par une règle comme ADDR-USER-IGNORE n'est pas en adéquation avec les extensions présentées dans la section 6.2.1. Il faut donc raffiner les règles de typage pour interdire ce cas.

6.2.4 Extensions du système de types

On présente ici un système de types plus expressif permettant de capturer les extensions de sémantique. *In fine*, cela permettra de prouver le théorème 6.1 qui est l'équivalent du théorème 5.1 mais pour le nouveau jugement de typage.

Définir un nouveau système de types revient à étendre le jugement de typage $\cdot \vdash \cdot : \cdot$, en modifiant certaines règles et en en ajoutant d'autres. Naturellement, la plupart des différences porteront sur le traitement des pointeurs.

Pointeurs utilisateurs

Le changement clef est l'ajout de *pointeurs utilisateur*. En plus des types pointeurs habituels $t*$, on ajoute des types pointeurs utilisateur $t@$. La différence entre les deux représente *qui* contrôle leur valeur (section 2.4).

Les différences sont les suivantes (figure 6.8) :

- Les types « $t*$ » s'appliquent aux pointeurs contrôlés par le noyau. Par exemple, prendre l'adresse d'un objet de la pile noyau donne un pointeur noyau.
- Les types « $t@$ », quant à eux, s'appliquent aux pointeurs qui proviennent de l'espace utilisateur. Ces pointeurs proviennent toujours d'interfaces particulières, comme les appels système ou les paramètres passés aux implantations de la fonction `ioctl`.

L'ensemble des notations est résumé dans le tableau suivant :

	Noyau	Utilisateur
Syntaxe	$\& x$	$\diamondsuit x$
Valeur	$\widehat{\&}\,(x)$	$\widehat{\&}\,\widehat{\diamondsuit}\,(x)$
Type	$t*$	$t@$
Accès	$*x$	copy_*_user

Puisqu'on s'intéresse à la provenance des pointeurs, détaillons les règles qui créent, manipulent et utilisent des pointeurs.

Sources de pointeurs

La source principale de pointeurs est l'opérateur & qui prend l'adresse d'une variable. Celle-ci est bien entendue contrôlée par le noyau (dans le sens où son déréférencement est toujours sûr). Cette construction crée donc des pointeurs noyau, et on maintient la règle suivante :

Type	$t ::= \ldots$	
	$\mid\ t@$	Pointeur utilisateur
Type de valeur	$\tau ::= \ldots$	
	$\mid\ \tau@$	Pointeur utilisateur

FIGURE 6.8: Ajouts liés aux pointeurs utilisateur (par rapport aux figures 5.2 et 5.5)

$$\frac{\Gamma \vdash lv : t}{\Gamma \vdash \&lv : t\,*}\ (\textsc{Addr})$$

Manipulations de pointeurs

L'avantage du typage est que celui-ci suit le flot de données : si à un endroit une valeur de type t est affectée à une variable, que le contenu de cette variable est placé puis retiré d'une structure de données, il conserve ce type t. En particulier un pointeur utilisateur reste un pointeur utilisateur.

Une seule règle consomme un pointeur et en retourne un. Elle concerne l'arithmétique des pointeurs. On ne l'étend pas aux pointeurs utilisateur, car pour effectuer de l'arithmérique, il faut observer la forme du pointeur sous-jacent. Si on veut laisser $\widehat{\diamondsuit}\,\cdot$ opaque, il faut donc interdire l'arithmétique sur les pointeurs utilisateur.

Utilisations de pointeurs

La principale restriction est que seuls les pointeurs noyau peuvent être déréférencés de manière sûre. La règle capitale est donc la suivante (déjà introduite dans le chapitre 5) :

$$\frac{\Gamma \vdash e : t\,*}{\Gamma \vdash *e : t}\ (\textsc{Lv-Deref})$$

Ainsi, on interdit le déréférencement des expressions de type $t\,@$ à la compilation.

L'opérateur $\diamondsuit\,\cdot$ transforme un pointeur selon la règle suivante :

$$\frac{\Gamma \vdash lv : t}{\Gamma \vdash \diamondsuit\ lv : t\,@}\ (\textsc{Addr-User})$$

Les « fonctions » copy_from_user et copy_to_user sont typées de la manière suivante. Il est à remarquer que ce ne sont pas vraiment des fonctions et qu'elles n'ont pas un type en $(t_1, t_2) \to t$, car il faudrait un type polymorphe pour pouvoir les appliquer à n'importe quel type de pointeurs. Leur typage est donc plus proche de celui d'un opérateur.

$$\frac{\Gamma \vdash e_d : t\,* \qquad \Gamma \vdash e_s : t\,@}{\Gamma \vdash \text{copy\_from\_user}(e_d, e_s) : \textsc{Int}}\ (\textsc{Uset-Get}) \qquad \frac{\Gamma \vdash e_d : t\,@ \qquad \Gamma \vdash e_s : t\,*}{\Gamma \vdash \text{copy\_to\_user}(e_d, e_s) : \textsc{Int}}\ (\textsc{User-Put})$$

6.2.5 Sûreté du typage

Typage sémantique

La définition du typage sémantique doit aussi être étendue au cas $\varphi = \widehat{\diamondsuit}\,\varphi'$. En essence, S-UserPtr énonce que traverser un constructeur $\widehat{\diamondsuit}\,\cdot$ transforme un pointeur en pointeur utilisateur.

$$\frac{m \vDash \widehat{\&}\,\varphi : \tau\,*}{m \vDash \widehat{\&}\,\widehat{\diamondsuit}\,\varphi : \tau\,@}\ (\textsc{S-UserPtr}) \qquad \frac{\tau \rhd t}{\tau\,@ \rhd t\,@}\ (\textsc{Comp-Ptr})$$

Propriétés du typage

Lemme 6.1 (Inversion du typage)**.** *En plus des cas présentés dans le lemme 5.1, les cas suivants permettent de remonter un jugement de typage.*

- *Si* $\Gamma \vdash \diamondsuit\ e : t$*, alors il existe* t' *tel que* $t = t'$ @ *et* $\Gamma \vdash e : t'$.
- *Si* $\Gamma \vdash$ *copy_from_user*$(e_d, e_s) : t$*, alors* $t =$ INT *et il existe* t' *tel que* $\Gamma \vdash e_d : t*$ *et* $\Gamma \vdash e_s : t$ @.
- *Si* $\Gamma \vdash$ *copy_to_user*$(e_d, e_s) : t$*, alors* $t =$ INT *et il existe* t' *tel que* $\Gamma \vdash e_d : t$ @ *et* $\Gamma \vdash e_s : t*$.

Démonstration. Pour chaque forme syntaxique, on liste les règles qui ont comme conclusion un jugement de typage portant sur celle-ci. Comme aucune autre règle ne convient, on peut en déduire que c'est l'une de celles-ci qui a été appliquée, et donc qu'une des prémisses est vraie. □

Progrès et préservation

La propriété que nous cherchons à prouver est que le déréférencement d'un pointeur dont la valeur est contrôlée par l'utilisateur ne peut se faire qu'à travers une fonction qui vérifie la sûreté de celui-ci.

En fait il s'agit des théorèmes de sûreté du chapitre précédent.

Théorème 6.1 (Progrès pour les extensions noyau)**.** *Le théorème 5.1 reste valable avec les extensions de ce chapitre.*

La preuve de ce théorème est en annexe D.4.

Théorème 6.2 (Préservation pour les extensions noyau)**.** *Le théorème 5.2 reste valable avec les extensions de ce chapitre.*

La preuve de ce théorème est en annexe D.5.

Ces extensions ne modifient pas les théorème de progrès et préservation sur les phrases (théorèmes 5.3 et 5.4).

La sûreté du typage étant à nouveau établie, on a montré que l'ajout de types pointeurs utilisateur suffit pour avoir une adéquation entre les extensions de sémantique de la section 6.2.1 et les extensions du système de type de la section 6.2.4.

Conclusion

En partant de SAFESPEAK tel que décrit dans les chapitres 4 et 5, on décrit une extension de sa syntaxe et de sa sémantique. Cela permet d'exprimer les pointeurs vers l'espace utilisateur, qui sont utilisés pour l'implantation d'appels système (chapitre 2).

Une première idée pour le typage de ces nouveaux pointeurs est de leur donner le même type que les pointeurs classiques. On a montré ensuite que ce typage naïf ne suffit pas : il permet en effet de faire fuiter de l'information, ce qu'on note par un cas d'erreur Ω_{sec}. En termes de systèmes de types, cela signifie que le théorème de progrès (théorème 5.1, page 77) n'est plus vérifié.

Le langage des types est donc enrichi pour séparer les pointeurs utilisateur des pointeurs noyau : les premiers sont explicitement construits par un ensemble de sources bien déterminé, et les autres sont créés par exemple en prenant l'adresse d'une variable. La règle de typage LV-DEREF assure que seuls les pointeurs noyau peuvent être déréférencés. Pour accéder aux pointeurs utilisateur, il faut appeler les constructions copy_to_user et copy_from_user,

qui sont typées adéquatement et vérifient dynamiquement que les pointeurs dont la valeur est contrôlée par l'utilisateur pointent vers l'espace utilisateur.

CONCLUSION DE LA PARTIE II

On vient de décrire en détail un langage impératif, SAFESPEAK : tout d'abord sa syntaxe et sa sémantique d'évaluation dans le chapitre 4. Une des spécificités de cette sémantique est l'utilisation de lentilles pour modifier les valeurs composées en profondeur.

Il y a plusieurs alternatives à cette présentation. La première est la solution classique qui consiste à décrire les modifications de la mémoire en extension. C'est en général long et laborieux puisqu'il faut définir les accès en lecture et écriture à chaque étape (avec des lentilles on décrit ces deux opérations uniquement sur les briques du calcul, et la composition fait le reste). La seconde solution est d'employer une sémantique monadique. Les transisitions sont alors encodées comme des actions monadiques qui représentent les modifications de la mémoire. Un des avantages de cette solution est qu'elle est très extensible. Par exemple, la propagation des erreurs ou l'ajout de continuations légères (c'est-à-dire le support des fonctions `setjmp` et `longjmp`) peuvent facilement être exprimés dans un formalisme monadique. Nous avons préféré une présentation classique qui reste plus accessible une audience habituée à C, et suffisante compte tenu de la simplicité des constructions à interpréter dans le langage.

Ensuite, dans le chapitre 5 nous avons ajouté un système de types à SAFESPEAK. Le but est de restreindre le genre d'erreurs qui peuvent arriver lors de l'évaluation d'un programme. Par le théorème de progrès (théorème 5.1 , page 77), on interdit les erreurs qui signalent une manipulation de valeurs incompatibles, l'accès à un champ de structure inconnu, et l'accès à une variable inexistante. Et le théorème de préservation (théorème 5.2, page 78) formalise le résultat classique qu'une étape d'évaluation ne modifier pas le typage. Une particularité de SAFESPEAK est que son état mémoire est structuré, avec une pile de variables locales explicite. On retrouve donc cette distinction dans le typage : les variables globales et les variable locales sont séparées dans les environnements de typage Γ (page 64).

Enfin, dans le chapitre 6, on a étendu le langage pour exprimer la notion de pointeurs utilisateur. Cela permet d'écrire des fonctions qui se comportent comme des appels système. On a commencé par montrer qu'une extension naïve du système de types ne suffit pas, car le théorème de progrès est alors invalidé. On ajoute donc un type dédié aux pointeurs utilisateur. Les valeurs de ce type sont créées explicitement et passées aux appels système. La règle de typage du déréférencement est restreinte aux pointeurs noyau, ce qui permet de ré-établir les théorèmes de progrès et préservation.

Notre technique de typage permet donc d'exprimer correctement les problèmes liés à la manipulation mémoire lors des appels système, ainsi que décrits dans le chapitre 2 : c'est une méthode simple pour détecter et empêcher les problèmes de sécurité qui proviennent des pointeurs utilisateur.

Comme nous l'avons fait remarquer dans le chapitre 3, utiliser une technique de typage pour étudier des propriétés sur les données a déjà été explorée dans l'outil CQual [FFA99], en particulier sur les problèmes de pointeurs utilisateurs [JW04].

En effet, si on remplace « $t*$ » par « KERNEL $t*$ » et « $t@$ » par « USER $t*$ », on obtient un début de système de types qualifiés.

En revanche, il y a une différence importante : CQual modifie fondamentalement l'ensemble du système de types, pas SAFESPEAK. Le jugement de typage de CQual a pour forme générale $\Gamma \vdash e : q\ t$ (où Γ est un environnement de typage, e une expression, q un qualifica-

teur et t un type), alors que le nôtre a la forme plus classique $\Gamma \vdash e : t$.

En intégrant q à la relation de typage, on ajoute un qualificateur à chaque type, même les expressions pour lesquels il n'est pas directement pertinent de déterminer qui les contrôle (comme par exemple, un entier). Dans CQual, ceci permet de traiter de manière correcte le transtypage. Par exemple, si e a pour type qualifié USER INT, alors (FLOAT $*$) e aura pour type qualifié USER FLOAT $*$, et déréférencer cette expression produira une erreur de typage. SAFESPEAK, dans son état actuel, ne permet pas de traiter les conversions de type et ne permet donc pas de traiter ce cas.

Nous prenons, au contraire, l'approche de ne modifier le système de types que là où cela est nécessaire, c'est-à-dire sur les types pointeurs. Cela permet de ne pas avoir à modifier en profondeur un système de types existant.

Le modèle d'exécution est aussi très différent. CQual s'appuie sur un langage proche de ML : un noyau de lambda-calcul avec des références. Le système de types sous-jacent est proche de celui d'OCaml : du polymorphisme de premier ordre (avec la restriction habituelle de généralisation des références) et du sous-typage structurel. En outre, leur approche repose sur une gestion automatique de la mémoire. De notre côté, nous nous appuyons sur un modèle mémoire plus proche de C, reposant sur une pile de variables et des pointeurs manipulés à la main.

Une autre différence fondamentale est que le système de types de CQual fait intervenir une relation de sous-typage. Le cas particulier du problème de déréférencement des pointeurs utilisateur peut être traité dans ce cadre en posant KERNEL $\preceq$ USER pour restreindre certaines opérations aux pointeurs KERNEL.

Notre approche, au contraire, n'utilise pas de sous-typage, mais consiste à définir un type abstrait t @ partageant certaines propriétés avec $t*$ (comme la taille et la représentation) mais incompatible avec certaines opérations. C'est à rapprocher des types abstraits dans les langages comme OCaml et Haskell.

Les perspectives de travaux futurs sont également très différentes. Dans le cas des pointeurs, même si le noyau Linux (et la plupart des systèmes d'exploitation) ne comportent que deux espaces d'adressage, il est commun dans les systèmes embarqués de manipuler des pointeurs provenant d'espaces mémoire indépendants : par exemple, de la mémoire flash, de la RAM, ou une EEPROM de configuration. Ces différentes mémoires possèdent des adresses, et un pointeur est interprété comme faisant référence à une ou l'autre selon le code dont il est tiré. Lorsqu'il y a plus de deux espaces mémoire, il devient impossible d'encoder cette propriété dans un qualificateur, qui n'a que deux valeurs possibles.

Troisième partie

Expérimentation

Après avoir décrit notre solution dans la partie II, on présente ici son implantation.

Le chapitre 7 décrit l'implantation en elle-même : un prototype d'analyseur de types, distribué avec le langage NEWSPEAK sur [☞3]. Il s'agit d'un logiciel libre, distribué sous la license LGPL. La compilation depuis C est réalisée par l'utilitaire C2NEWSPEAK. Celui-ci, tout comme le langage NEWSPEAK, proviennent d'EADS et sont antérieurs à ce projet, mais le support de plusieurs extensions GNU C a été développé spécialement pour pouvoir analyser le code du noyau Linux.

L'analyse en elle-même est implantée de la manière classique avec une variation de l'algorithme W de Damas et Milner. Pour des raisons de simplicité et d'efficacité, l'unification est faite en utilisant le partage de références plutôt que des substitutions. L'algorithme d'inférence ne pose pas de problèmes de performance.

Ensuite, dans le chapitre 8, on évalue cette implantation sur le noyau Linux. On commence par décrire comment fonctionnennt les appels système sous ce noyau, et comment le *confused deputy problem* évoqué dans le chapitre 2 peut arriver dans ce contexte. Dans une deuxième partie, on décrit le cas de deux *bugs* dans le noyau Linux. On montre que, pour chacun, les analyses précédentes permettent de distinguer statiquement le cas incorrect du cas corrigé.

CHAPITRE 7

IMPLANTATION

Dans ce chapitre, nous décrivons la mise en œuvre des analyses statiques précédentes. Celles-ci ont été décrites sur SAFESPEAK, qui permet de modéliser des programmes C bien typables.

Notre but est d'utiliser la représentation intermédiaire NEWSPEAK, développée par EADS. Cela permet de profiter des nombreux outils existant déjà autour de ce langage, notamment un compilateur depuis C et un analyseur statique par interprétation abstraite.

Mais cette représentation utilise un modèle mémoire différent. En effet il colle finement à celui de C, où des constructions comme les unions empêchent la sûreté du typage. Définir SAFESPEAK a précisement pour but de définir un langage inspiré de C mais sur lequel le typage peut être sûr. Il faudra donc adapter les règles de typage les chapitres 5 et 6. On reviendra sur cette distinction entre les deux niveaux de sémantique dans la conclusion de la partie III, page 119.

On commence par décrire le langage NEWSPEAK. Ensuite, nous décrivons la phase de compilation, de C à NEWSPEAK, auquel on rajoute ensuite des étiquettes de types. Celles-ci sont calculées par un algorithme d'inférence de types à la Hindley-Milner, reposant sur l'unification et le partage de références. Toutes ces étapes sont implantées dans le langage OCaml [CMP03].

Le prototype décrit ici est disponible sur [☞3] sous une license libre, la *GNU Lesser General Public License*.

7.1 NEWSPEAK et chaîne de compilation

NEWSPEAK est un langage intermédiaire conçu pour être un bon support d'analyses statiques, contrairement à des langages conçus pour les programmeurs comme C. Sa sémantique d'exécution (ainsi qu'une partie des étapes de compilation) est décrite dans [HL08]. Sa syntaxe est donnée dans la figure 7.1.

La traduction depuis C est faite en trois étapes : prétraitement du code source par un outil externe, compilation séparée de C prétraité vers des objets NEWSPEAK, puis liaison de ces différentes unités de compilation. Il est aussi possible de compiler directement du code Ada vers un objet NEWSPEAK.

La première étape consiste à prétraiter les fichiers C source avec le logiciel cpp, comme pour une compilation normale. Cette étape interprète les directives de prétraitement comme `#include`, `#ifdef`. À cet étape, les commentaires sont aussi supprimés.

Instruction	$s ::=$ Set(lv, e, st)	Affectation
	| Copy(lv, lv, n)	Copie
	| Guard(e)	Garde
	| Decl(var, t, blk)	Déclaration
	| Select(blk, blk)	Branchement
	| InfLoop(blk)	Boucle infinie
	| DoWith(blk, x)	Nommage de bloc
	| Goto(x)	Saut
	| Call$([(e_i, t_i)], f, [(lv_i, t_i)])$	Appel de fonction
Bloc	$blk ::= [s_i]$	Liste d'instructions
Valeur gauche	$lv ::=$ Local(x)	Locale
	| Global(x)	Globale
	| Deref(e, n)	Déréférencement
	| Shift(lv, e)	Décalage
Expression	$e ::=$ CInt(n)	Entier
	| CFloat(d)	Flottant
	| Nil	Pointeur nul
	| Lval(lv, t)	Accès mémoire
	| AddrOf(lv)	Adresse de variable
	| AddrOfFun$(x, [t_i], [t_i])$	Adresse de fonction
	| UnOp$(unop, e)$	Opérateur unaire
	| BinOp$(binop, e_1, e_2)$	Opérateur binaire
Fonction	$f ::=$ FunId(x)	Appel par nom
	| FunDeref(e)	Appel par pointeur
Type	$t ::=$ Scalar(st)	Type scalaire
	| Array(t, n)	Tableau
	| Region$([(n_i, t_i)], n')$	Structure/union
Type scalaire	$st ::=$ Int(n)	Entier
	| Float(n)	Flottant
	| Ptr	Pointeur sur données
	| FunPtr	Pointeur sur fonction

FIGURE 7.1: Syntaxe simplifiée de NEWSPEAK

Une fois cette passe effectuée, le résultat est un ensemble de fichiers C prétraités ; c'est-à-dire des unités de compilation.

Sur cette représentation (du C prétraité), il est possible d'ajouter des annotations de la forme `/*!npk [...] */` qui pourront être accessibles dans l'arbre de syntaxe abstraite des passes suivantes.

À ce niveau, les fichiers sont passés à l'outil C2NEWSPEAK qui les traduit vers NEWSPEAK. Comme il sera décrit dans la section 8.1, la plupart des extensions GNU C sont acceptées en plus du C ANSI. Dans cette étape, les types et les noms sont résolus, et le programme est annoté de manière à rendre les prochaines étapes indépendantes du contexte. Par exemple, chaque déclaration de variable est adjointe d'une description complète du type.

Lors de cette étape, le flot de contrôle est également simplifié (figure 7.2). De plus, les constructions ambigües en C comme `i = i++` sont transformées pour que leur évaluation se fasse dans dans un ordre explicite.

Au contraire, NEWSPEAK propose un nombre réduit de constructions. Rappelons que le but de ce langage est de faciliter l'analyse statique : des constructions orthogonales permettent donc d'éviter la duplication de règles sémantiques, ou de code, lors de l'implantation d'un analyseur.

Par exemple, plutôt que de fournir une boucle *while*, une boucle *do/while* et une boucle *for*, NEWSPEAK fournit une unique boucle WHILE(1){·}. La sortie de boucle est compilée vers un GOTO [EH94], qui est toujours un saut vers l'avant (similaire à un « *break* » généralisé).

NEWSPEAK est conçu pour l'analyse statique par interprétation abstraite. Il a donc une vue de bas niveau sur les programmes. Par exemple, aucune distinction n'est faite entre l'accès à un champ et l'accès à un élément d'un tableau (tous deux sont traduits par un décalage numérique depuis le début de la zone mémoire). De plus, les unions et les structures sont regroupées sous forme des types « régions » qui associent à un décalage un type de champ. Pour supprimer ces ambiguïtés, il faut s'interfacer dans les structures internes de C2NEWSPEAK, où les informations nécessaires sont encore présentes.

Ensuite, les différents fichiers sont liés ensemble. Cette étape consiste principalement à s'assurer que les hypothèses faites par les différentes unités de compilation sont cohérentes entre elles. Les objets marqués `static`, invisibles à l'extérieur de leur unité de compilation, sont renommés afin qu'ils aient un nom globalement unique. Cette étape se conclut par la

```
                    int32 x;
                    x =(int32) 0;
                    do {
                        while (1) {
                            choose {
int x;                          -->
x = 0;                              guard((10 > x_int32));
while (x < 10) {                -->
    x++;                            guard(! (10 > x_int32));
}                                   goto lbl1;
                            }
                            x =(int32) coerce[-2**31,2**31-1] (x_int32 + 1);
                        }
                    } with lbl1: {
                    }
```

FIGURE 7.2: Compilation du flot de contrôle en NEWSPEAK

création d'un fichier NEWSPEAK.

La dernière étape est réalisée dans un autre outil nommé `ptrtype`. Elle consiste en l'implantation d'un algorithme d'inférence pour les systèmes de types décrits dans les chapitres 5 et 6 est assez simple. Puisqu'ils sont suffisamment proches du lambda calcul simplement typé, on peut utiliser une variante de l'algorithme W de Damas et Milner [DM82].

Cela repose sur l'unification : on dispose d'une fonction permettant de créer des inconnues de type, et d'une fonction pour unifier deux types partiellement inconnus. Cet algorithme sera décrit dans la section 7.2. En pratique, on utilise l'optimisation classique qui consiste à se reposer sur le partage de références pour réaliser l'unification, plutôt que de faire des substitutions explicites. Puisque ces systèmes de types sont monomorphes, on présente une erreur si des variable de type libres sont présentes.

À la fin de cette étape, on obtient soit un programme complètement annoté, soit une erreur de type.

7.2 Algorithme d'unification

On présente ici la fonction `unify`. Celle-ci prend en entrée deux représentations de types pouvant contenir des inconnues de la forme `Var n`, et retourne une liste de couples (n, t) indiquant les substitutions à faire.

Cet algorithme (décrit en pseudo-code ML en figure 7.3) prend un chemin différent selon la forme des deux types d'entrée :

- si les deux types sont inconnus (de la forme `Var n`), on substitue l'un par l'autre.
- si un type est inconnu et pas l'autre, il faut de la même manière faire une substitution. Mais en faisant ça inconditionnellement, cela peut poser problème : par exemple en tentant d'unifier `a` avec `Ptr(a)` on pourrait créer une substitution cyclique. Pour éviter cette situation, il suffit de s'assurer que le type inconnu n'est pas présent dans le type à affecter. C'est le but de la fonction `occurs(n, t)` qui calcule si `Var n` apparaît dans `t`.
- si les deux types sont des types de base (comme INT ou FLOAT) égaux, on ne fait rien.
- si les deux types sont des constructeurs de type, il faut que les constructeurs soient égaux. On unifie en outre leurs arguments deux à deux.
- dans les autres cas, l'algorithme échoue.

Le traitement des types structures est géré dans l'implantation d'une manière différente de la présentation du chapitre 4. C'est pourquoi elle n'apparaît pas dans la figure 7.3. Au lieu d'accéder directement au type complet S à chaque accès $x.l_S$, on n'obtient qu'un nom de champ à chaque accès. C'est-à-dire qu'on va par exemple inférer le type $\{l : \text{INT}; \ldots X\}$ où $\ldots X$ désigne l'ensemble des champs inconnus.

Plus précisément, si on cherche à unifier les types structures $A = \{a_1 : t_1; \ldots; a_n : t_n; \ldots X_a\}$ et $B = \{b_1 : s_1; \ldots; b_m : u_m; \ldots X_b\}$, il faut partitionner l'ensemble des champs en 3 : ceux qui apparaissent dans les deux structures, ceux qui apparaissent dans A mais pas dans B, et ceux qui apparaissent dans B mais pas dans A.

- Pour tous les champs l tels que $l : a_i \in A$ et $l : b_j \in B$, on unifie a_i et b_j.
- Pour les champs l qui sont dans A mais pas dans B : on ajoute l à X_b.
- Et symétriquement dans X_a.

Cela se rapproche du polymorphisme de rangée [RV98] présent dans les langages comme OCaml. À la fin de l'inférence, on considère que la variable de rangée « $\ldots X$ » est vide. Elle n'apparaît donc pas dans les types.

```
1: function UNIFY(t_a, t_b)
2:     match (t_a, t_b) with
3:         | VAR n_a, VAR n_b ⇒
4:             if n_a = n_b then
5:                 return [ ]
6:             else
7:                 return [(n_a, t_b)]
8:             end if
9:         | VAR n_a, t_b ⇒
10:            if OCCURS(n_a, t_b) then
11:                erreur
12:            end if
13:            return [(n_a, t_b)]
14:        | t_a, VAR n_b ⇒ return UNIFY(t_b, t_a)
15:        | INT, INT ⇒ return [ ]
16:        | FLOAT, FLOAT ⇒ return [ ]
17:        | a[ ], b[ ] ⇒ return UNIFY(a, b)
18:        | a *, b * ⇒ return UNIFY(a, b)
19:        | a @, b @ ⇒ return UNIFY(a, b)
20:        | (l_a) → r_a, (l_b) → r_b ⇒
21:            r ← UNIFY(r_a, r_b)
22:            n ← LENGTH(l_a)
23:            if LENGTH(l_b) ≠ n then
24:                erreur
25:            end if
26:            for i = 0 to n − 1 do
27:                r ← r ∪ UNIFY(l_a[i], l_b[i])
28:            end for
29:            return r
30:        | _ ⇒ erreur
31: end function
```

FIGURE 7.3: Algorithme d'unification

7.3 Architecture de `ptrtype`

Cette analyse est implantée dans un outil du nom de `ptrtype`, d'environ 1600 lignes de code OCaml. Il lit un programme NEWSPEAK (ou un fichier C), et réalise l'inférence de types. En sortie, il affiche soit le programme complètement annoté, soit une erreur.

Si l'argument passé à `ptrtype` est un fichier C, il est tout d'abord compilé vers NEWSPEAK grâce à l'utilitaire C2NEWSPEAK. Ensuite, les autres passes travaillent sur une représentation intermédiaire proche de NEWSPEAK, mais où des étiquettes de type supplémentaires sont ajoutées. Ce type de représentation intermédiaire est noté `'a Tyspeak.t` (pour *typed* NEWSPEAK), où `'a` est le type des étiquettes.

Le reste de l'outil est résumé dans la fonction `process_npk` (figure 7.4) :

- Grâce à la fonction `convert_unit : Newspeak.t -> unit Tyspeak.t`, on ajoute des étiquettes « vides » (toutes égales à `() : unit`).
- L'ensemble des fonctions du programme est trié topologiquement selon la relation $\preceq$

```
let process_npk npk =
  let tpk = Npk2tpk.convert_unit npk in
  let order = Topological.topological_sort (Topological.make_graph npk) in

  let function_is_defined f =
    Hashtbl.mem tpk.Tyspeak.fundecs f
  in

  let (internal_funcs, external_funcs) =
    List.partition function_is_defined order
  in

  let exttbl = Printer.parse_external_type_annotations tpk in

  let env =
    env_add_external_fundecs exttbl external_funcs Env.empty
  in
  let s = Infer.infer internal_funcs env tpk in
  begin
    if !Options.do_checks then
      Check.check env s
  end;
  Printer.dump s
```

FIGURE 7.4: Fonction principale de `ptrtype`

définie par $f \preceq g \stackrel{\text{def}}{=}$ « g apparaît dans la définition de f ». Cela est fait en construisant une représentation de $\preceq$ sous forme de graphe, puis en faisant un parcours en largeur de celui-ci. Pour le moment, les fonctions récursives et mutellement récursives ne sont pas supportées.

- Les annotations extérieures sont alors lues (variable `exttbl`), ce qui permet de créer un environnement initial. Celles-ci permettent par exemple de spécifier le type d'une fonction inconnue.
- Les types de chaque fonction sont inférés, par le biais de la fonction suivante :

```
val infer : Newspeak.fid list   (* liste triée de fonctions à typer *)
         -> Types.simple Env.t  (* environnement initial *)
         -> 'a Tyspeak.t        (* programme à analyser *)
         -> Types.simple Tyspeak.t
```

- S'il n'y a pas d'erreurs, le programme obtenu, de type `Types.simple Tyspeak.t`, est affiché sur le terminal.

La fonction appelée `unify`, appelée dans toutes les fonctions d'inférence, peut retarder l'unification (figure 7.5). Dans ce cas, la paire de types à unifier est mise dans une liste d'attente qui sera unifiée après le parcours du programme. Le but est d'instrumenter l'inférence de types afin de pouvoir en faire une exécution « pas à pas ».

```
let unify a b =
  if !Options.lazy_unification then
    Queue.add (Unify (a, b)) unify_queue
  else
    unify_now a b
```

FIGURE 7.5: Unification directe ou retardée

7.4 Inférence de types

L'inférence de types consiste à remplacer les étiquettes de type `unit` par des étiquettes de type `simple` (autrement dit de vraies représentations de types).

Cette étape se fait de manière impérative. Cela permet de ne pas avoir à réaliser de substitutions explicites. À la place, on repose sur le partage et les références, qui représentent les inconnues de type. Lorsque celles-ci sont résolues, il suffit de muter une seule fois la référence, et le partage fait que ce changement sera visible partout. Plus précisément, on peut créer de nouveaux types avec la fonction `new_unknown` et unifier deux types avec la fonction `unify`. Leurs types sont :

```
val new_unknown : unit -> Types.simple
val unify : Types.simple -> Types.simple -> unit
```

La fonction `infer` s'appuie sur un ensemble de fonctions récursivement définies portant sur chaque type de fragment : `infer_fdec` pour les déclarations de fonction, `infer_exp` pour les expressions, `infer_stmtkind` pour les instructions, etc. Grâce au lemme 5.1, on sait quelle règle appliquer en fonction de l'expression ou instruction considérée. Notons que, même si le programme NEWSPEAK est décoré d'informations de types (celles qui existent dans le programme C), elles ne sont pas utilisées.

Les règles de typage sont implantées par `new_unknown` et `unify`. Par exemple, pour typer une déclaration (qui n'a pas de valeur initiale en NEWSPEAK), on crée un nouveau type `t0`. On étend l'environnement courant avec cette nouvelle association et, sous ce nouvel environnement, on type le bloc de portée de la déclaration (figure 7.6).

De même, pour typer un appel de fonction, on infère le type de ses arguments et valeurs gauches de retour. On obtient également le type de la fonction (à partir du type de la fonction présent dans l'environnement, ou du type du pointeur de fonction qui est déréférencé), et on unifie ces deux informations.

On peut donner quelques autres exemples. Pour additionner deux flottants, on unifie leur type avec FLOAT. Le résultat est également de type FLOAT. Cela correspond à la règle OP-FLOAT.

```
let infer_binop op (_, a) (_, b) =
  match op with
    (* [...] *)
    | N.PlusF _ ->
        unify a Float;
        unify b Float;
        Float
```

Pour prendre l'adresse d'une variable, la règle ADDR s'applique : on prend le type de la valeur gauche et on construit un pointeur noyau à partir de lui.

```
let rec infer_stmtkind env sk =
  match sk with
  (* [...] *)
  | T.Decl (n, nty, _ty, blk) ->
      let var = T.Local n in
      let t0 = new_unknown () in
      let new_env = Env.add (VLocal n) (Some nty) t0 env in
      let blk' = infer_blk new_env blk in
      let ty = lval_type new_env var in
      T.Decl (n, nty, ty, blk')
  (* [...] *)
  | T.Call (args, fexp, rets) ->
      let infer_arg (e, nt) =
        let et = infer_exp env e in
        (et, nt)
      in

      let infer_ret (lv, nt) =
        (infer_lv env lv, nt)
      in

      let args' = List.map infer_arg args in
      let rets' = List.map infer_ret rets in

      let t_args = List.map (fun ((_, t), _) -> t) args' in
      let t_rets = List.map (fun (lv, _) -> lval_type env lv) rets' in

      let (fexp', tf) = infer_funexp env fexp in
      let call_type = Fun (t_args, t_rets) in
      unify tf call_type;
      T.Call (args', fexp', rets')
```

FIGURE 7.6: Inférence des déclarations de variable et appels de fonction

```
| T.AddrOf lv ->
    let lv' = infer_lv env lv in
    let ty = lval_type env lv in
    (T.AddrOf lv', Ptr (Kernel, ty))
```

Enfin, pour déréférencer une expression, on unifie tout d'abord son type avec le type d'un pointeur noyau.

```
| T.Deref(e, _sz) ->
    let (_, te) = infer_exp env e in
    let t = new_unknown () in
    unify (Ptr (Kernel, t)) te;
    t
```

7.5 Exemple

Lançons l'analyse sur un petit exemple :

```
int f(int *x) { return (*x + 1); }
```

L'exécution de notre analyseur affiche un programme complètement annoté :

```
 % ptrtype example.c
     1 f : (KPtr (Int)) -> (Int)
     2 Int (example.c:1#4)^f(KPtr (Int) x) {
     3   (.c:3#4)^!return =(int32)
     4     (coerce[-2147483648,2147483647]
     5       ( ( ([(x_KPtr (Int) : KPtr (Int))]32_Int
     6             : Int
     7           )
     8           + (1 : Int)
     9         ) : Int
    10       ) : Int
    11     );
    12 }
```

- Ligne 1 : le type inféré de la fonction `f` est affiché. Il est calculé entièrement en fonction des opérations effectuées, on n'utilise pas les étiquettes de type du programme.
- Ligne 2 : le code de la fonction est affiché. Les indications de la forme `(F:L#C)^X` correspondent à la déclaration d'une variable X, dans le fichier F, ligne L et colonne C.
- Ligne 3 : en NEWSPEAK, la valeur de retour est une variable qui est affectée. On sépare ainsi le flot de données (définir la valeur de retour) du flot de contrôle (sortir de la fonction). C'est un équivalent de la variable $\underline{R}$ introduite pour le typage des fonctions (page 69). L'affectation est notée `=(int32)` car en NEWSPEAK elle est décorée du type des opérandes. Cette information n'est pas utilisée dans l'inférence de types.
- Ligne 4 : l'opérateur `coerce[a,b]` sert à détecter les débordements d'entiers lors d'une analyse de valeurs par interprétation abstraite. Dans le cas de notre analyse, les valeurs ne sont pas pertinentes et cet opérateur peut être vu que comme l'identité.
- Ligne 5 : le déréférencement d'une valeur gauche `e` est noté `[e]_n`. Il est annoté par la taille de l'opérande (32 bits ici). De plus, l'accès à une valeur gauche (pour la transformer en expression) est annoté par son type, ce qui explique la verbosité de cette ligne.
- les autres lignes sont des étiquettes de type inférées sur les expressions 1, $*x$, 1, $*x+1$ et la valeur de retour coerce$[-2^{31}, 2^{31}-1](*x+1)$.

Un exemple de détection d'erreur sera décrit dans la section 8.6.

7.6 Performance

Même s'il est simple en apparence, le problème de l'inférence de types par l'algorithme W est EXP-complet [Mai90], c'est-à-dire que les algorithmes efficaces ont une complexité exponentielle en la taille du programme. Cependant, lorsqu'on borne la « taille » des types, celle-ci devient quasi-linéaire [McA03], ce qui signifie qu'il n'y a pas de problème de performance à attendre.

Dans notre cas, on utilise une variante de l'algorithme W pour un langage particulièrement simple. En particulier il n'y a pas de polymorphisme, ni de fonctions imbriquées, et les types des valeurs globales sont écrites par le programmeur, ce qui permet de borner d. En effet, sur les exemples testés nous n'avons pas noté de délai d'exécution notable.

En revanche, la compilation de C vers NEWSPEAK peut être plus coûteuse, notamment lorsque le fichier d'entrée est de taille importante. Le temps de traitement est plus long que celui d'un compilateur comme `gcc` ou `clang`, mais reste de l'ordre de quelques secondes. C2NEWSPEAK a été utilisé pour compiler des projets de l'ordre du million de lignes de code source prétraité, et son exécution ne prenait pas plus de quelques minutes.

Les structures internes de C2NEWSPEAK ont déjà été améliorées, et d'autres optimisations sont certainement possibles, mais la performance n'est pas bloquante pour le moment : une fois que le code est compilé, on peut réutiliser le fichier objet NEWSPEAK pour d'autres analyses. La compilation est donc relativement rare.

Conclusion

Les analyses de typage correspondant aux chapitres 5 et 6 ont été implantées sous forme d'un prototype utilisant le langage NEWSPEAK développé par EADS. Cela permet de réutiliser les phases de compilation déjà implantées, et d'exprimer les règles de typage sur un langage suffisament simple.

On utilise un algorithme par unification, qui donne une forme simple au programme d'inférence. Pour chaque expression ou instruction à typer, on détermine grâce au lemme 5.1 quelle règle il faut appliquer. Ensuite, on génère les inconnues de type nécessaires pour appliquer cette règle et on indique les contraintes en appelant la fonction d'unification.

Ce prototype comporte environ 1600 lignes de code OCaml. Il est disponible sous license libre sur [☞3]. Il a été pensé pour traiter un type de code particulier, à savoir le noyau Linux. On montre dans le chapitre suivant que cet objectif est atteint, puisqu'il permet de détecter plusieurs bugs.

CHAPITRE 8

ÉTUDE DE CAS : LE NOYAU LINUX

Le noyau Linux, abordé dans le chapitre 2, est un noyau de système d'exploitation développé depuis le début des années 90 et « figure de proue » du mouvement *open-source*. Au départ écrit par Linus Torvalds sur son ordinateur personnel, il a été porté au fil des années sur de nombreuses architectures et s'est enrichi de nombreux pilotes de périphériques. Dans la version 3.13.1 (2014), son code source comporte 12 millions de lignes de code (en grande majorité du C) dont 58% de pilotes.

Même si le noyau est monolithique (la majeure partie des traitements s'effectue au sein d'un même fichier objet), les sous-systèmes sont indépendants. C'est ce qui permet d'écrire des pilotes de périphériques et des modules.

Ces pilotes manipulent des données provenant de l'utilisateur, notamment par pointeur. Comme on l'a vu, cela peut poser des problèmes de sécurité si on déréférence ces pointeurs sans vérification.

Dans ce chapitre, on met en œuvre sur le noyau Linux le système de types décrit dans le chapitre 6, ou plus précisément l'outil `ptrtype` du chapitre 7.

Pour montrer que le système de types capture cette propriété et que l'implantation est utilisable, on étudie les cas de deux bugs qui ont touché le noyau Linux. À chaque fois, dans une routine correspondant à un appel système, un pointeur utilisateur est déréférencé directement, pouvant provoquer une fuite d'informations confidentielles dans le noyau.

On commence par décrire les difficultés rencontrées pour analyser le code du noyau Linux. On décrit ensuite l'implantation du mécanisme d'appels système dans ce noyau, et en quoi cela peut poser des problèmes. On détaille enfin les bugs étudié, et comment les adapter pour traiter le code en question.

8.1 GNU C

Linux est écrit dans le langage C, mais pas dans la version qui correspond à la norme. Il utilise le dialecte GNU C qui est celui que supporte GCC. Une première difficulté pour traiter le code du noyau est donc de le compiler.

Pour traduire ce dialecte, il a été nécessaire d'adapter C2NEWSPEAK. La principale particularité est la notation `__attribute__((...))` qui peut décorer les déclarations de fonctions, de variables ou de types.

Par exemple, il est possible de manipuler des étiquettes de première classe : si « `lbl:` » est présent avant une instruction, on peut capturer l'adresse de celle-ci avec `void *p = &&lbl` et y sauter indirectement avec `goto *p`.

Une autre fonctionnalité est le concept d'instruction-expression : `({bloc})` est une expression, dont la valeur est celle de la dernière expression évaluée lors de `bloc`.

Les attributs, quant à eux, rentrent dans trois catégories :

- les annotations de compilation ; par exemple, `used` désactive l'avertissement « cette variable n'est pas utilisée ».
- les optimisations ; par exemple, les objets marqués `hot` sont groupés de telle manière qu'ils se retrouvent en cache ensemble.
- les annotations de bas niveau ; par exemple, `aligned(n)` spécifie qu'un objet doit être aligné sur au moins `n` bits.

Dans notre cas, toutes ces annotations peuvent être ignorées, mais il faut tout de même adapter l'analyse syntaxique pour les ignorer. En particulier, pour le traitement du noyau Linux, il a fallu traiter certaines formes de la construction `typeof` qui n'étaient pas supportées.

De plus, pour que le code noyau soit compilable, il est nécessaire de définir certaines macros. En particulier, le système de configuration de Linux utilise des macros nommées `CONFIG_*` pour inclure ou non certaines fonctionnalités. Il a donc fallu faire un choix ; nous avons choisi la configuration par défaut. Pour analyser des morceaux plus importants du noyau, il faudrait définir un fichier de configuration plus important.

8.2 Appels système sous Linux

Dans cette section, nous allons voir comment ces mécanismes sont implantés dans le noyau Linux. Une description plus détaillée pourra être trouvée dans [BC05], ou, pour le cas de la mémoire virtuelle, dans [Gor04].

Deux rings sont utilisés : en *ring* 0, le code noyau et, en *ring* 3, le code utilisateur.

Une notion de tâche similaire à celle décrite dans la section 2.2 existe : elles s'exécutent l'une après l'autre, le changement s'effectuant sur interruptions.

Pour faire appel aux services du noyau, le code utilisateur doit faire appel à des appels système, qui sont des fonctions exécutées par le noyau. Chaque tâche doit donc avoir deux piles : une pile « utilisateur », qui sert pour l'application elle-même, et une pile « noyau », qui sert aux appels système.

Grâce à la mémoire virtuelle, chaque processus possède sa propre vue de la mémoire dans son espace d'adressage (figure 8.1), et donc chacun gère un ensemble de tables de pages et une valeur de CR3 associée (ce mécanisme a été abordé page 17). Au moment de changer le processus en cours, l'ordonnanceur charge donc le CR3 du nouveau processus.

Les adresses basses (inférieures à `PAGE_OFFSET` = 3 Gio = `0xc0000000`) sont réservées à l'utilisateur. On y trouvera par exemple : le code du programme, les données du programme (variables globales), la pile utilisateur, le tas (mémoire allouée par `malloc` et fonctions similaires), ou encore les bibliothèques partagées.

Au dessus de `PAGE_OFFSET`, se trouve la mémoire réservée au noyau. Cette zone contient le code du noyau, les piles noyau des processus, etc.

FIGURE 8.1: L'espace d'adressage d'un processus. En gris clair, les zones accessibles à tous les niveaux de privilèges : code du programme, bibliothèques, tas, pile. En gris foncé, la mémoire du noyau, réservée au mode privilégié.

Les programmes utilisateur s'exécutant en *ring* 3, ils ne peuvent pas contenir d'instructions privilégiées, et donc ne peuvent pas accéder directement au matériel. Pour que ces programmes puissent interagir avec le système (afficher une sortie, écrire sur le disque...) le mécanisme des appels système est nécessaire. Il s'agit d'une interface de haut niveau entre les *rings* 3 et 0. Du point de vue du programmeur, il s'agit d'un ensemble de fonctions C « magiques » qui font appel au système d'exploitation pour effectuer des opérations.

Par exemple, le programmeur peut appeller la fonction `getpid` pour connaître le numéro du processus courant. Cela passe par une fonction `getpid` dans la bibliothèque C, en espace utilisateur. Celle-ci va invoquer (via un mécanisme non pertinent ici) la fonction `sys_getpid` du noyau (figure 8.2).

Comme les piles sont différentes entre les espaces, la convention d'appel est différente : les arguments sont copiés directement par les registres.

```
SYSCALL_DEFINE0(getpid)
{
        return task_tgid_vnr(current);
}
```

FIGURE 8.2: Fonction de définition d'un appel système

La macro `SYSCALL_DEFINE0` permet de nommer la fonction `sys_getpid`, et définit entre autres des points d'entrée pour les fonctionnalités de débogage du noyau. Le corps de la fonction fait directement référence aux structures de données internes du noyau pour retourner le résultat voulu.

8.3 Risques

Ainsi que décrit dans la section 2.4, cela peut poser un problème de manipuler des pointeurs contrôlés par l'utilisateur au sein d'une routine de traitement d'appel système.

Si le déréférencement est fait sans vérification, un utilisateur mal intentionné peut forger un pointeur vers le noyau (en déterminant des adresses valides dans l'espace noyau entre `0xc0000000` et `0xffffffff`). En provoquant une lecture sur ce pointeur, des informations confidentielles peuvent fuiter ; et, en forçant une écriture, il est possible d'augmenter ses privilèges, par exemple en devenant super-utilisateur (*root*). En pratique, il n'est pas toujours possible d'accéder à la mémoire. La mémoire utilisateur peut par exemple avoir été placée en zone d'échange sur le disque, ou *swap*. À ce moment là, l'erreur provoquera tout de même un déni de service. Plus de détails sur ce mécanisme, et le fonctionnement de la mémoire virtuelle dans Linux, peuvent être trouvés dans [Jon10].

8.4 Premier exemple de bug : pilote Radeon KMS

On décrit le cas d'un pilote vidéo qui contenait un bug de pointeur utilisateur. Il est répertorié sur `http://freedesktop.org` en tant que bug #29340.

Pour changer de mode graphique, les pilotes de GPU peuvent supporter le *Kernel Mode Setting* (KMS).

Pour configurer un périphérique, l'utilisateur communique avec le pilote noyau avec le mécanisme d'*ioctls* (pour *Input/Output Control*). Ils sont similaires à des appels système, mais spécifiques à un périphérique particulier. Le transfert de contrôle est similaire à ce qui a été décrit dans la section précédente : les applications utilisateurs appellent la fonction

`ioctl()` de la bibliothèque standard, qui provoque une interruption. Celle-ci est traitée par la fonction `sys_ioctl()` qui appelle la routine de traitement dans le bon pilote de périphérique.

Les fonctions du noyau implantant un *ioctl* sont donc vulnérables à la même classe d'attaques que les appels système, et donc doivent être écrits avec une attention particulière.

Le code de la figure 8.3 est présent dans le pilote KMS pour les GPU AMD Radeon.

```
/* drivers/gpu/drm/radeon/radeon_kms.c */
int radeon_info_ioctl(struct drm_device *dev, void *data,
                      struct drm_file *filp) {
        struct radeon_device *rdev = dev->dev_private;
        struct drm_radeon_info *info;
        struct radeon_mode_info *minfo = &rdev->mode_info;
        uint32_t *value_ptr;
        uint32_t value;
        struct drm_crtc *crtc;
        int i, found;

        info = data;
        value_ptr = (uint32_t *) ((unsigned long)info->value);
/*=>*/  value = *value_ptr;
        /* ... */
}
```

FIGURE 8.3: Code de la fonction `radeon_info_ioctl`

On peut voir que l'argument `data` est converti en un `struct drm_radeon_info *`. Un pointeur `value_ptr` est extrait de son champ `value`, et finalement ce pointeur est déréférencé.

Cependant, l'argument `data` est un pointeur vers une structure (allouée en espace noyau) du type donné dans la figure 8.4, dont les champs proviennent d'un appel utilisateur de `ioctl()`.

```
/* from include/drm/radeon_drm.h */
struct drm_radeon_info {
  uint32_t    request;
  uint32_t    pad;
  uint64_t    value;
};
```

FIGURE 8.4: Définition de `struct drm_radeon_info`

Pour mettre ce problème en évidence, nous avons annoté la fonction `radeon_info_ioctl` de telle manière que son second paramètre soit un pointeur noyau vers une structure contenant un champ contrôlé par l'utilisateur, `value`.

L'intégralité de ce code peut être trouvée en annexe A.

La bonne manière de faire a été publiée avec le numéro de *commit* `d8ab3557` (figure 8.5) (`DRM_COPY_FROM_USER` étant une simple macro pour `copy_from_user`). Dans ce cas, on n'obtient pas d'erreur de typage.

```
--- a/drivers/gpu/drm/radeon/radeon_kms.c
+++ b/drivers/gpu/drm/radeon/radeon_kms.c
@@ -112,7 +112,9 @@

        info = data;
        value_ptr = (uint32_t *)((unsigned long)info->value);
-       value = *value_ptr;
+       if (DRM_COPY_FROM_USER(&value, value_ptr, sizeof(value)))
+               return -EFAULT;
+
        switch (info->request) {
        case RADEON_INFO_DEVICE_ID:
                value = dev->pci_device;
```

FIGURE 8.5: Patch résolvant le problème de pointeur utilisateur. La ligne précédée par un signe – est supprimée et remplacée par les lignes précédées par un signe +.

8.5 Second exemple : `ptrace` sur architecture Blackfin

Le noyau Linux peut s'exécuter sur l'architecture Blackfin, qui est spécialisée dans le traitement du signal. Le problème de manipulation des pointeurs utilisateur auquel nous nous intéressons peut également s'y produire.

En particulier nous nous intéressons à l'appel système `ptrace`. Il permet à un processus d'accéder à la mémoire et de contrôler l'exécution d'un autre processus, par exemple à des fins de débogage. Ainsi, `ptrace(PTRACE_PEEKDATA, p, addr)` renvoie la valeur du mot mémoire à l'adresse `addr` dans l'espace d'adressage du processus `p`.

Comme pour la plupart des appels système, la fonction `ptrace` est dépendante de l'architecture. Le deuxième exemple que nous présentons concerne l'implantation de celle-ci pour les processeurs Blackfin, figure 8.6.

Dans d'anciennes versions de Linux[1], cette fonction appelle `memcpy` au lieu de `copy_from_user` pour lire dans la mémoire du processus. La ligne problématique est préfixée par `/*=>*/`. En théorie, si un utilisateur passe un pointeur vers une adresse du noyau à la fonction `ptrace`, il pourra lire des données du noyau. L'appel `ptrace (PTRACE_PEEKDATA, p, addr)` permet ainsi non seulement de lire les variables du processus `p` si `addr` est une adresse dans l'espace utilisateur (ce qui est le comportement attendu), mais aussi de lire dans l'espace noyau si `addr` y pointe (ce qui est un bug de sécurité).

On peut repérer ce bug par simple relecture pour commencer. On commence par remarquer que l'argument `addr`, malgré son type `long`, est en réalité un `void *` provenant directement de l'espace utilisateur. C'est en effet le même argument `addr` de l'appel système `ptrace`. Cet argument correspond à l'adresse à lire dans l'espace mémoire du processus. Comme il est passé à `memcpy`, aucune vérification n'est faite avant la copie. La valeur pointée par `addr` sera copiée, même si elle est en espace noyau.

En annotant correctement les types, on peut donc détecter ce bug : le type correct de `addr` est INT @, et celui de `memcpy` est (INT *, INT *, INT) → INT *. Il est donc impossible de lui passer cet argument. Remarquons que le type de `memcpy` en C utilise des pointeurs de type

1. Jusqu'à la version 2.6.28 — ce bug a été corrigé dans le *commit* `7786ce82` en remplaçant l'appel à `memcpy` par un appel à `copy_from_user_page`.

```
/* kernel/ptrace.c */
SYSCALL_DEFINE4(ptrace, long, request, long, pid, unsigned long, addr,
                unsigned long, data)
{
        struct task_struct *child = ptrace_get_task_struct(pid);
        /* ... */
        long ret = arch_ptrace(child, request, addr, data);
        /* ... */
        return ret;
}

/* arch/blackfin/kernel/ptrace.c */
long arch_ptrace(struct task_struct *child, long request,
                 long addr, long data)
{
        int ret;
        unsigned long __user *datap = (unsigned long __user *)data;

        switch (request) {
        /* ... */
        case PTRACE_PEEKTEXT: {
                unsigned long tmp = 0;
                int copied;

                ret = -EIO;

                /* ... */
                if (addr >= FIXED_CODE_START
                    && addr + sizeof(tmp) <= FIXED_CODE_END) {
/*=>*/                  memcpy(&tmp, (const void *)(addr), sizeof(tmp));
                        copied = sizeof(tmp);
                }
                /* ... */
                ret = put_user(tmp, datap);
                break;
        /* ... */
        }

        return ret;
}
```

FIGURE 8.6: Implantation de `ptrace` sur architecture Blackfin

`void *`. Pour les traiter correctement on pourrait utiliser du polymorphisme, mais dans ce cas précis utiliser le type INT * est suffisant.

Remarque En pratique, le problème de sécurité n'est pas si important. En effet, la copie se fait sous un test forçant `addr` à être entre `FIXED_CODE_START` et `FIXED_CODE_END`. Cette zone est incluse en espace utilisateur, cela empêche donc le problème de fuite de données.

Mais cela reste un problème de sécurité : contrairement à `copy_from_user`, la fonction `memcpy` ne vérifie pas que l'espace utilisateur est chargé en mémoire. Si ce n'est pas le cas, une faute mémoire sera provoquée dans le noyau. Il s'agit alors d'un déni de service (section 8.3), qui est tout de même un comportement à empêcher.

8.6 Procédure expérimentale

Pour utiliser notre système de types, plusieurs étapes sont nécessaires en plus de traduire le noyau Linux en SAFESPEAK.

Afin de réaliser l'analyse, il faut annoter les sources pour créer un environnement initial (via la variable `exttbl` décrite en section 7.3). Plus précisément, pour chaque source de pointeurs utilisateur, on ajoute un commentaire spécial `!npk userptr_fieldp x f`, qui indique que `x` est un pointeur vers une structure contenant un pointeur utilisateur dans le champ `f`. En fait, il unifie le type de `x` avec $\{f : t\ @; \ldots\} *$ où t est une inconnue de type. Cette annotation est nécessaire car c'est le moyen d'indiquer que la structure contient un pointeur utilisateur.

Par rapport au code complet présent dans l'annexe A, l'expression calculant `value_ptr` est également simplifiée. Dans le code d'origine, `info->value` est transtypé en `unsigned long` puis en `uint32_t *`. En NEWSPEAK, cela correspond à des opérateurs `PtrToInt` et `IntToPtr` mais, si on les autorise, on casse le typage puisqu'il est alors possible de transformer n'importe quel type en un autre. De plus, on modifie la définition du type `struct drm_radeon_info` pour que son champ `value` ait pour type `uint32_t *` plutôt que `uint64_t`. En effet, dans ce cas d'étude, cet entier est uniquement utilisé en tant que pointeur au cours de toute l'exécution.

En ce qui concerne les fonctions de manipulation de pointeurs fournies par le noyau (`get_user`, `put_user`, `copy_from_user`, `copy_to_user`, etc.), on ajoute à l'environnement global leur type correct.

Enfin, on peut lancer l'inférence de type. Ainsi, sur l'exemple de la figure 8.7, on obtient la sortie suivante :

```
05-drm.c:18#8 - Type clash between :
  KPtr (_a15)
  UPtr (_a8)
```

Cela indique qu'on a essayé d'unifier un type de la forme $t *$ avec un type de la forme $t\ @$, en précisant l'emplacement où la dernière unification a échoué (les `_aN` correspondent à des inconnues de type).

La version correcte minimisée correspond à la figure 8.8. Pour celle-ci, l'inférence se fait sans erreur.

```
typedef unsigned long uint32_t;

struct drm_radeon_info {
    uint32_t *value;
};

int radeon_info_ioctl(struct drm_device *d, void *data, struct drm_file *f)
{
        /*!npk userptr_fieldp data value*/
        struct drm_radeon_info *info;
        uint32_t *value_ptr;
        uint32_t value;
        struct drm_crtc *crtc;
        int i, found;

        info = data;
        value_ptr = info->value;
        value = *value_ptr;
        return 0;
}
```

FIGURE 8.7: Cas d'étude minimisé et annoté

```
typedef unsigned long uint32_t;

struct drm_radeon_info {
    uint32_t *value;
};

int radeon_info_ioctl(struct drm_device *d, void *data, struct drm_file *f)
{
        /*!npk userptr_fieldp data value*/
        struct drm_radeon_info *info;
        uint32_t *value_ptr;
        uint32_t value;
        struct drm_crtc *crtc;
        int i, found;

        info = data;
        value_ptr = info->value;
        if (copy_from_user(&value, value_ptr, sizeof(value)))
            return -14;
        value = *value_ptr;
        return 0;
}
```

FIGURE 8.8: Cas d'étude minimisé et annoté – version correcte

Conclusion

Après voir décrit l'implantation de notre solution, on a montré comment celle-ci peut s'appliquer à détecter un bug dans le noyau Linux. La première difficulté est de traduire en NEWSPEAK le code source écrit dans le dialecte GNU C.

En partant d'un bug existant, on montre que la version originale du code (incluant une erreur de programmation) ne peut pas être typée, alors que sur la version corrigée on peut inférer des types compatibles.

Le prototype décrit dans le chapitre 7 peut donc s'adapter à détecter des bugs dans le noyau Linux. Pour le moment, il nécessite du code annoté, mais des travaux sont en cours pour permettre de passer automatiquement des portions plus importantes du noyau Linux. Le principal obstacle est de devoir réécrire certaines parties du code pour supprimer les constructions non typables.

CONCLUSION DE LA PARTIE III

Après avoir décrit notre solution théorique dans la partie II, nous avons présenté ici notre démarche expérimentale. Dans le chapitre 7, nous avons détaillé l'implantation de notre prototype. Pour ce faire, nous avons ajouté des étiquettes de type au langage NEWSPEAK et adapté l'algorithme W de Damas et Milner pour réaliser l'inférence de types. Ce prototype est distribué sur [☞3] sous le nom de `ptrtype`. Il est constitué d'environ 1600 lignes d'OCaml.

Ensuite, le but du chapitre 8 est d'appliquer notre analyse (à l'aide de ce prototype) sur le noyau Linux. Après avoir décrit le fonctionnement des appels système sur ce noyau, on présente deux bugs qui ont touché respectivement un pilote de carte graphique et l'implantation d'un appel système. Ils sont la manifestation d'un problème de pointeur utilisateur mal déréférencé dans le noyau, ainsi que décrit dans le chapitre 2. En lançant notre analyse sur le code présentant un problème, l'erreur est détectée. Au contraire, en la lançant sur le code après application du correctif, aucune erreur n'est trouvée.

En s'appuyant sur le langage NEWSPEAK, on gagne beaucoup par rapport à d'autres représentations intermédiaires. Le fait d'avoir un langage avec peu de constructions permet de ne pas avoir à exprimer plusieurs fois la même règle (par exemple, une fois sur la boucle *for* et une autre sur la boucle *while*).

Un des inconvénients est que le modèle mémoire utilisé par NEWSPEAK est assez différent de celui de SAFESPEAK (ainsi que décrit dans le chapitre 4). NEWSPEAK est en effet prévu pour implanter des analyses précises de valeur reposant sur l'interprétation abstraite, et nécessite donc un modèle mémoire de plus bas niveau (où on peut créer des valeurs à partir d'une suite d'octets, par exemple).

Le prototype d'implantation peut évoluer dans deux directions : d'une part, en continuant à s'appuyer sur NEWSPEAK, on peut réaliser des pré-analyses de typage qui permettent de guider une analyse de valeurs plus précise, par exemple en choisissant un domaine abstrait différent en fonction des types de données rencontrés. D'autre part, il est possible de faire une implantation plus fidèle à SAFESPEAK, qui permette d'ajouter de nouvelles fonctionnalités plus éloignées de C. Par exemple, un système de régions comme [TJ92] permettrait de simplifier l'environnement d'exécution en enlevant l'opération de nettoyage mémoire Cleanup(·). Le système de types peut également être enrichi, pour ajouter par exemple du polymorphisme. Cela rapprocherait le langage source de Rust. Le chapitre 9 présente quelques unes de ces extensions possibles.

L'expérimentation, quant à elle, est pour le moment limitée, mais on peut l'étendre à des domaines de plus en plus importants dans le noyau Linux. Tout d'abord, le module graphique définit d'autres fonctions implantant des *ioctls*. Celles-ci reçoivent donc également des pointeurs utilisateur et sont susceptibles d'être vulnérables à ce genre d'erreurs de programmation. Ensuite, d'autres modules exposent une interface similaire, à commencer par les autres pilotes de cartes graphiques. Ceux-ci sont également un terrain sur lequel appliquer cette analyse.

De manière générale, toutes les interfaces du noyau manipulant des pointeurs utilisateur gagnent à être analysées. Outre les implantations des *ioctls* dans chaque pilote et les appels système, les systèmes de fichiers manipulent aussi de tels pointeurs via leurs opérations de lecture et d'écriture.

Quatrième partie

Conclusion

CHAPITRE 9

CONCLUSION

On fait ici un résumé des travaux présentés, en commençant par un bilan des contributions réalisées. On fait ensuite un tour des aspects posant problème, ou traités de manière incomplète, en évoquant les travaux possibles pour enrichir l'expressivité de ce système.

9.1 Contributions

Cette thèse comporte 4 contributions principales.

Un langage impératif bien typé Le système de types de C est trop rudimentaire pour permettre d'obtenir des garanties sur l'exécution des programmes bien typés. En interdisant certaines constructions dangereuses et en annotant certaines autres, nous avons isolé un langage impératif bien typable, SAFESPEAK, pour lequel on peut définir un système de types sûr.

Une sémantique basée sur les lentilles Une des particularités de SAFESPEAK est qu'il utilise un état mémoire structuré, modélisant les cadres de piles présents dans le langage. Pour décrire la sémantique des accès mémoire, nous utilisons le concept de lentilles issues de la programmation fonctionnelle et des systèmes de bases de données. Cela permet de définir de manière déclarative la modification en profondeur de valeurs dans la mémoire, sans avoir à distinguer le cas de la lecture et celui de l'écriture.

Un système de types abstraits En partant de ce système de types, on a décrit une extension permettant de créer des pointeurs pour lesquels l'opération de déréférencement est restreinte à certaines fonctions. Dans le contexte d'un noyau de système d'exploitation, cette restriction permet de vérifier statiquement qu'à aucun moment le noyau ne déréférence un pointeur dont la valeur est contrôlée par l'espace utilisateur, évitant ainsi un problème de sécurité. Cette approche peut s'étendre à d'autres classes de problèmes comme par exemple éviter l'utilisation de certaines opérations sur les types entiers lorsqu'ils sont utilisés comme identificateurs ou masque de bits.

Un prototype d'analyseur statique Les analyses de typage ici décrites ont été implantées sous forme d'un prototype d'analyseur statique distribué avec le langage NEWSPEAK, développé par EADS. Le choix de NEWSPEAK pour l'implantation demande d'adapter les règles de typage, mais il permet de réutiliser un traducteur existant, et de contribuer les résultats

à l'entreprise. Ce prototype permet d'une part de vérifier la propriété d'isolation des appels système sur du code C existant, et d'autre part fournit une base saine pour implanter d'autres analyses de typage sur le langage NEWSPEAK. Ce prototype a été utilisé pour confirmer l'existence de deux bugs dans le noyau Linux, ce qui permet de valider l'approche : il est possible de vérifier du code de production à l'aide de techniques de typage. Des travaux d'expérimentation sont en cours afin d'analyser de plus grandes parties du noyau.

9.2 Différences avec C

SAFESPEAK a été construit pour pouvoir ajouter un système de types à un langage proche de C. Ces deux langages diffèrent donc sur certains points. On détaille ici ces différences et selon les cas, comment les combler ou pourquoi cela est impossible de manière inhérente.

Types numériques En C, on dispose de plusieurs types entiers, pouvant avoir plusieurs tailles et être signés ou non signés, ainsi que des types flottants qui diffèrent par leur taille. Au contraire, en SAFESPEAK on ne conserve qu'un seul type d'entier et un seul type de flottant. La raison pour cela est que nous ne nous intéressons pas du tout aux problématiques de sémantique arithmétique : les débordements, dénormalisations, etc, sont supposés ne pas arriver.

Il est possible d'étendre le système de types de SAFESPEAK pour ajouter tous ces nouveaux types. La traduction depuis NEWSPEAK insère déjà des opérateurs de transtypage pour lesquels il est facile de donner une sémantique (pouvant lever une erreur en cas de débordement, comme en Ada) et un typage. Les littéraux numériques peuvent poser problème, puisqu'ils deviennent alors polymorphes. Une solution peut être de leur donner le plus grand type entier et d'insérer un opérateur de transtypage à chaque littéral. Haskell utilise une solution similaire : les littéraux entiers ont le type de précision arbitraire `Integer` et sont convertis dans le bon type en appelant la fonction `fromInteger` du type synthétisé à partir de l'environnement.

Transtypage et unions Puisque l'approche retenue est basée sur le typage statique, il est impossible de capturer de nombreuses constructions qui sont permises, ou même idiomatiques en C : les unions, les conversions de types (explicites ou implicites) et le *type punning* (défini ci-dessous). Les deux premières sont équivalentes. Bien qu'on puisse remplacer chaque conversion explicite d'un type t_1 vers un type t_2 par l'appel à une fonction cast_{t_1,t_2}, on ajoute alors un « trou » dans le système de types. Cette fonction devrait en effet être typée $(t_1) \to t_2$, autrement dit le type « maudit » $\alpha \to \beta$ de `Obj.magic` en OCaml ou `unsafeCoerce` en Haskell.

Le *type punning* consiste à modifier directement la suite de bits de certaines données pour la manipuler d'une manière efficace. Par exemple, il est commun de définir un ensemble de macros pour accéder à la mantisse et à l'exposant de flottants IEEE754. Ceci peut être fait avec des unions ou des masques de bits.

Dans de tels cas, le typage statique est bien sûr impossible. Pour traiter ces cas, il faudrait encapsuler la manipulation dans une fonction et y ajouter une information de type explicite, comme `float_exponent` : (FLOAT) → INT.

Pour ces conversions de types, on distingue en fait plusieurs cas : les conversions entre types numériques, entre types pointeurs, ou entre un type entier et un type pointeur.

Le premier ne pose pas de problème : il est toujours possible de donner une sémantique à une conversion entre deux types numériques, quitte à détecter les cas où il faut signaler une erreur à l'exécution (comme en cas de débordement).

Le deuxième non plus n'est pas un problème en soi : une conversion entre deux types pointeurs revient à convertir entre les types pointés (il faut bien sûr interdire les conversions entre pointeurs noyau et utilisateur).

Le vrai problème provient des conversions entre entiers et pointeurs, qui sont des données fondamentalement différentes. Le même problème se pose d'ailleurs si on cherche à convertir une fonction en entier ou en pointeur, même si les raisons valables pour faire cela sont moins nombreuses. Si on s'en tient aux conversions entre entiers et pointeurs, une manière naïve de typer ces opérations est :

$$\frac{\Gamma \vdash e : t*}{\Gamma \vdash (\textsc{Int})\ e : \textsc{Int}}\ (\textsc{PtrInt-Bad}) \qquad \frac{\Gamma \vdash e : \textsc{Int}}{\Gamma \vdash (\textsc{Ptr})\ e : t*}\ (\textsc{IntPtr-Bad})$$

Tout d'abord, cela pose problème car il est alors possible de créer une fonction pouvant convertir n'importe quel type pointeur en n'importe quel autre type pointeur :

$$\vdash \text{fun}(p)\{\textsc{Return}((\textsc{Ptr})\ (\textsc{Int})\ p)\} : (t_a*) \to t_b*$$

Si on crée une variable du type t_a, prend son adresse, la convertit à l'aide de cette fonction, puis déréférence le résultat, on obtient une valeur du type t_b (remarquons que ce genre d'opération est tout à fait possible en C).

Outre ce problème de typage, il faudrait pouvoir donner une sémantique à ces opérations. Convertir un pointeur en entier revient à spécifier l'environnement d'exécution, c'est-à-dire qu'il faut une fonction de placement en mémoire beaucoup plus précise que notre modèle mémoire actuel. Celle-ci dépend de beaucoup de paramètres : dans quel sens croit la pile, quelle est la taille des types, etc.

La conversion dans le sens inverse, d'entier vers pointeur, est encore plus complexe. Entre autres, cela suppose qu'on puisse retrouver la taille des valeurs à partir de leur adresse. Dans de nombreux langages, on résout ce problème en stockant la taille de chaque valeur avec elle.

Mais cela fait s'éloigner du modèle mémoire de C, où le déréférencement porte sur une adresse mais également sur une taille (portée implicitement par le type du pointeur). Le langage NEWSPEAK conserve d'ailleurs cette distinction, que nous avons éliminée dans SAFESPEAK. Il y a une incompatibilité entre ces deux approches : dans le cas de C (et de NEWSPEAK), on laisse le programmeur gérer l'organisation de la mémoire alors qu'avec SAFESPEAK ces choix sont faits par le langage. En contrepartie, cela permet d'avoir d'assurer la sûreté du typage.

Environnement d'exécution La sémantique opérationnelle utilise un environnement d'exécution pour certains cas. Contrairement à C, les débordements de tampon et les déréférencements de pointeurs sont vérifiés dynamiquement. Mais ce n'est pas une caractéristique cruciale de cette approche : en effet, si on suppose que les programmes que l'on analyse ne comportent pas de telles erreurs de programmation, on peut désactiver ces vérifications et le reste des propriétés est toujours valable.

On repose sur l'environnement d'exécution à un endroit plus problématique. À la sortie de chaque portée (au retour d'une fonction et après la portée d'une variable locale déclarée), on parcourt la mémoire à la recherche des pointeurs référençant les variables qui ne sont plus valides. Supprimer ce test rend l'analyse incorrecte, car il est alors possible de faire référence à une variable avec un type différent.

Si on peut avoir une garantie statique que les adresses des variables locales ne seront plus accessibles au retour d'une fonction, alors on peut supprimer cette étape de nettoyage. Cette

garantie peut être obtenue avec une analyse statique préalable. Par exemple les régions [TJ92] peuvent être utilisées à cet effet : en plus de donner un type à chaque expression, on calcule statiquement la zone mémoire dans laquelle cette valeur sera allouée. Cela correspond à un ramasse-miette réalisé statiquement.

Flot de contrôle Dans le langage C, en plus des boucles et de l'alternative, on peut sauter au sein d'une fonction à l'aide de la construction `goto`. Pour pouvoir traiter ces cas, il est possible de transformer ces sauts d'un programme vers des simples boucles. Cette réécriture peut être coûteuse puisqu'elle peut introduire des variables booléennes et dupliquer du code. En pratique, c'est d'ailleurs ce qui est fait dans l'implantation puisque cette transformation est réalisée par C2NEWSPEAK.

Dans le noyau Linux, il est courant d'utiliser les sauts pour factoriser la libération de ressources à la fin d'une fonction. Il est d'ailleurs possible d'utiliser l'outil Coccinelle pour donner cette forme à du code utilisant un autre style de structures de contrôle [SLM11]. On peut imaginer qu'il est possible de l'utiliser pour faire la conversion inverse.

En plus de ces sauts locaux, le langage C contient une manière de sauvegarder un état d'exécution et d'y sauter, même entre deux fonctions : ce sont respectivement les constructions `setjmp` et `longjmp`. Elles sont très puissantes puisqu'elles permettent d'exprimer de nouvelles structures de contrôle. Il s'agit de formes légères de continuations où la pile reste commune. Cette fonctionnalité peut servir par exemple à implanter des exceptions, ou des coroutines.

Avec l'interprète du chapitre 4, il n'est pas possible de donner une sémantique à ces constructions. Une des manières de faire est de modifier les états de l'interprète : au lieu de retenir l'instruction à évaluer avec $\langle i, m\rangle$, on retient la continuation complète : $\langle k, m\rangle$. Pour faciliter ce changement, on peut tout d'abord passer à une sémantique monadique (ainsi qu'évoqué dans la conclusion de la partie II, page 95) puis ajouter les continuations à la monade sous-jacente.

En pratique, il est rare de trouver ces constructions plus avancées dans du code noyau ou embarqué, donc ce manque n'a pas beaucoup d'impact. De plus, cela permet une présentation plus simple et accessible.

Allocation dynamique La plupart des programmes, et le noyau Linux en particulier, utilisent la notion d'allocation dynamique de mémoire. C'est une manière de créer dynamiquement une zone de mémoire qui restera accessible après l'exécution de la fonction courante. Cette mémoire pourra être libérée à l'aide d'une fonction dédiée. Dans l'espace utilisateur, les programmes peuvent utiliser les fonctions `malloc()`, `calloc()`, et `realloc()` pour allouer des zones de mémoire et `free()` pour les libérer. Dans le noyau Linux, ces fonctions existent sous la forme de `kmalloc()`, `kfree()`, etc. Une explication détaillée de ces mécanismes peut être trouvée dans [Gor04].

Ces fonctions manipulent les données en tant que zones mémoires opaques, en renvoyant un pointeur vers une zone mémoire d'un nombre d'octets donnés. Cela présuppose un modèle mémoire plus bas niveau. Pour se rapprocher de la sémantique de SAFESPEAK, une manière de faire est de définir un opérateur de plus haut niveau prenant une expression et retournant l'adresse d'une cellule mémoire contenant cette valeur (la taille de chaque valeur fait partie de celle-ci). Le typage est alors direct :

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \textsc{New}(e) : t*}\ (\textsc{New}) \qquad \frac{\Gamma \vdash e : t*}{\Gamma \vdash \textsc{Free}(e)}\ (\textsc{Free})$$

En ce qui concerne l'exécution, on peut ajouter une troisième composante aux états mémoire : $m = (s, g, h)$ où h est une liste d'association entre des entiers et des valeurs. Chaque allocation dynamique crée une nouvelle clef entière et met à jour h. La libération de mémoire est en revanche problématique parce qu'il faut faire confiance au programmeur pour ne pas accéder aux zones mémoires libérées, ni libérer deux fois la même zone mémoire. Il est aussi possible d'obtenir cette garantie avec une analyse préalable. Par exemple, il est possible ici encore d'utiliser une analyse basée sur les régions pour vérifier l'absence de pointeurs fous [DDMP10].

Conclusion

L'importance des logiciels grandit par deux effets : d'une part, ils sont présents dans de plus en plus d'appareils, et d'autre part, leur taille est de plus en plus importante. En une journée, entre les appareils dédiés au calcul, à la communication, au multimédia, et au transport, on est facilement exposé au fonctionnement de plus d'une dizaine de millions de lignes de code. Il donc primordial de vérifier que ces logiciels ne peuvent pas être détournés de leur utilisation prévue. Les conséquences peuvent en effet être catastrophiques dans un contexte avionique ou militaire.

Au cœur de la plupart de ces systèmes informatiques, se trouve un noyau qui abstrait les détails du matériel pour fournir aux programmes des abstractions sûres, permettant de protéger les données sensibles contenues dans ce système. Si une simple erreur de programmation peut briser cette isolation, on voit pourquoi la vérification est si importante.

Dans ce but, les systèmes de types sont des outils bien connus de programmeurs. Même dans les langages peu typés comme C, les compilateurs aident de plus en plus les programmeurs à trouver des erreurs de programmation. De très nombreuses analyses peuvent être faites rien qu'en classifiant les expressions selon le genre de valeurs qu'elles créent à l'exécution — c'est la définition que donne Benjamin C. Pierce d'un système de types [Pie02, p. 1].

L'utilisation d'un système de types comme analyseur statique léger est donc efficace. Pour des propriétés qui ne dépendent pas de la valeur des expressions, mais uniquement de leur forme, c'est d'ailleurs la solution à préférer. En effet, nous avons montré qu'elle est simple à mettre en œuvre et rapide à exécuter.

On peut se poser la question suivante : pourquoi a-t-on besoin d'une analyse statique dédiée, plutôt que de passer par le langage C lui-même ? Le problème vient du fait que celui-ci considère que les types définissent une représentation en mémoire sans guère plus d'information. On peut définir des nouveaux noms pour un type, mais l'ancien et le nouveau sont alors compatibles. En un mot il est impossible de distinguer le rôle d'un type (son intention) de sa représentation (son extension).

Ici, nous avons amené une solution au problème de pointeurs utilisateur en introduisant un type ayant la même représentation que les pointeurs classique, mais pour lequel l'ensemble des opérations est différents : c'est un type opaque. Cela suffit déjà à détecter des erreurs de programmation.

Si on ajoutait cette construction au langage, on pourrait définir des nouveaux types partageant la représentation d'un type C existant, mais qui ne soit pas compatible avec le type d'origine, et pour lequel l'ensemble des opérations est restreint. Avec cette fonctionnalité dans un langage, non seulement on peut détecter d'autres classes de problèmes, mais surtout on laisse le programmeur définir de nouvelles analyses lui-même.

Annexes

ANNEXE A

MODULE RADEON KMS

On inclut ici le code analysé dans le chapitre 8. On inclut à la suite le contexte nécessaire pour comprendre ce code.

```
/* from drivers/gpu/drm/radeon/radeon_kms.c */
int radeon_info_ioctl(struct drm_device *dev, void *data, struct drm_file *filp)
{
        struct radeon_device *rdev = dev->dev_private;
        struct drm_radeon_info *info;
        struct radeon_mode_info *minfo = &rdev->mode_info;
        uint32_t *value_ptr;
        uint32_t value;
        struct drm_crtc *crtc;
        int i, found;

        info = data;
        value_ptr = (uint32_t *)((unsigned long)info->value);
        value = *value_ptr;
        switch (info->request) {
        case RADEON_INFO_DEVICE_ID:
                value = dev->pci_device;
                break;
        case RADEON_INFO_NUM_GB_PIPES:
                value = rdev->num_gb_pipes;
                break;
        case RADEON_INFO_NUM_Z_PIPES:
                value = rdev->num_z_pipes;
                break;
        case RADEON_INFO_ACCEL_WORKING:
                /* xf86-video-ati 6.13.0 relies on this being false for evergreen */
                if ((rdev->family >= CHIP_CEDAR) && (rdev->family <= CHIP_HEMLOCK))
                        value = false;
                else
                        value = rdev->accel_working;
                break;
        case RADEON_INFO_CRTC_FROM_ID:
                for (i = 0, found = 0; i < rdev->num_crtc; i++) {
                        crtc = (struct drm_crtc *)minfo->crtcs[i];
                        if (crtc && crtc->base.id == value) {
                                struct radeon_crtc *radeon_crtc = to_radeon_crtc(crtc);
                                value = radeon_crtc->crtc_id;
```

```
                                found = 1;
                                break;
                        }
                }
                if (!found) {
                        DRM_DEBUG_KMS("unknown crtc id %d\n", value);
                        return -EINVAL;
                }
                break;
        case RADEON_INFO_ACCEL_WORKING2:
                value = rdev->accel_working;
                break;
        case RADEON_INFO_TILING_CONFIG:
                if (rdev->family >= CHIP_CEDAR)
                        value = rdev->config.evergreen.tile_config;
                else if (rdev->family >= CHIP_RV770)
                        value = rdev->config.rv770.tile_config;
                else if (rdev->family >= CHIP_R600)
                        value = rdev->config.r600.tile_config;
                else {
                        DRM_DEBUG_KMS("tiling config is r6xx+ only!\n");
                        return -EINVAL;
                }
        case RADEON_INFO_WANT_HYPERZ:
                mutex_lock(&dev->struct_mutex);
                if (rdev->hyperz_filp)
                        value = 0;
                else {
                        rdev->hyperz_filp = filp;
                        value = 1;
                }
                mutex_unlock(&dev->struct_mutex);
                break;
        default:
                DRM_DEBUG_KMS("Invalid request %d\n", info->request);
                return -EINVAL;
        }
        if (DRM_COPY_TO_USER(value_ptr, &value, sizeof(uint32_t))) {
                DRM_ERROR("copy_to_user\n");
                return -EFAULT;
        }
        return 0;
}

/* from include/drm/radeon_drm.h */
struct drm_radeon_info {
        uint32_t                request;
        uint32_t                pad;
        uint64_t                value;
};

/* from drivers/gpu/drm/radeon/radeon_kms.c */
struct drm_ioctl_desc radeon_ioctls_kms[] = {
        /* KMS */
        DRM_IOCTL_DEF(DRM_RADEON_INFO, radeon_info_ioctl, DRM_AUTH|DRM_UNLOCKED)
```

```
};

/* from drivers/gpu/drm/radeon/radeon_drv.c */

static struct drm_driver kms_driver = {
        .driver_features =
            DRIVER_USE_AGP | DRIVER_USE_MTRR | DRIVER_PCI_DMA | DRIVER_SG |
            DRIVER_HAVE_IRQ | DRIVER_HAVE_DMA | DRIVER_IRQ_SHARED | DRIVER_GEM,
        .dev_priv_size = 0,
        .ioctls = radeon_ioctls_kms,
        .name = "radeon",
        .desc = "ATI Radeon",
        .date = "20080528",
        .major = 2,
        .minor = 6,
        .patchlevel = 0,
};

/* from drivers/gpu/drm/drm_drv.c */
int drm_init(struct drm_driver *driver)
{
        DRM_DEBUG("\n");
        INIT_LIST_HEAD(&driver->device_list);

        if (driver->driver_features & DRIVER_USE_PLATFORM_DEVICE)
                return drm_platform_init(driver);
        else
                return drm_pci_init(driver);
}
```

ANNEXE B

SYNTAXE ET RÈGLES D'ÉVALUATION

On rappelle ici pour référence la syntaxe de SAFESPEAK, ainsi que sa sémantique d'évaluation. Les règles sont décrites dans les chapitres 4 et 6.

B.1 Syntaxe des expressions

Constantes	$c ::= n$	Entier
	$\mid d$	Flottant
	$\mid$ NULL	Pointeur nul
	$\mid ()$	Valeur unité
Expressions	$e ::= c$	Constante
	$\mid \boxminus\, e$	Opération unaire
	$\mid e \boxplus e$	Opération binaire
	$\mid lv$	Accès mémoire
	$\mid lv \leftarrow e$	Affectation
	$\mid \&lv$	Pointeur
	$\mid f$	Fonction
	$\mid e(e_1, \ldots, e_n)$	Appel de fonction
	$\mid \{l_1 : e_1; \ldots; l_n : e_n\}$	Structure
	$\mid [e_1; \ldots; e_n]$	Tableau
Valeurs gauches	$lv ::= x$	Variable
	$\mid lv.l_S$	Accès à un champ
	$\mid lv[e]$	Accès à un élément
	$\mid *e$	Déréférencement
Fonctions	$f ::= \text{fun}(x_1, \ldots, x_n)\{i\}$	Arguments, corps

B.2 Syntaxe des instructions

Instructions	$i ::=$ PASS	Instruction vide
	$\mid\ i ; i$	Séquence
	$\mid\ e$	Expression
	$\mid$ DECL $x = e$ IN$\{i\}$	Déclaration de variable
	$\mid$ IF$(e)\{i\}$ELSE$\{i\}$	Alternative
	$\mid$ WHILE$(e)\{i\}$	Boucle
	$\mid$ RETURN(e)	Retour de fonction
Phrases	$p ::= x = e$	Variable globale
	$\mid\ e$	Évaluation d'expression
Programme	$P ::= (p_1, \ldots, p_n)$	Phrases

B.3 Syntaxe des opérateurs

Opérateurs binaires	$\boxplus ::= +, -, \times, /, \%$	Arithmétique entière
	$\mid\ +., -., \times., /.$	Arithmétique flottante
	$\mid\ +_p, -_p$	Arithmétique de pointeurs
	$\mid\ \leq, \geq, <, >$	Comparaison sur les entiers
	$\mid\ \leq., \geq., <., >.$	Comparaison sur les flottants
	$\mid\ =, \neq$	Tests d'égalité
	$\mid$ &, |, ^	Opérateurs bit à bit
	$\mid$ &&, ||	Opérateurs logiques
	$\mid\ \ll, \gg$	Décalages
Opérateurs unaires	$\boxminus ::= +, -$	Arithmétique entière
	$\mid\ +., -.$	Arithmétique flottante
	$\mid\ \sim$	Négation bit à bit
	$\mid$!	Négation logique

B.4 Contextes d'évaluation

Contextes

$$
\begin{aligned}
C ::= {} & \bullet \\
| \;\; & C \boxplus e \\
| \;\; & v \boxplus C \\
| \;\; & \boxminus C \\
| \;\; & \& C \\
| \;\; & C \leftarrow e \\
| \;\; & \varphi \leftarrow C \\
| \;\; & \{l_1 : v_1; \ldots; l_i : C; \ldots; l_n : e_n\} \\
| \;\; & [v_1; \ldots; C; \ldots; e_n] \\
| \;\; & C(e_1, \ldots, e_n) \\
| \;\; & f(v_1, \ldots, C, \ldots, e_n) \\
| \;\; & C.l_S \\
| \;\; & C[e] \\
| \;\; & \varphi[C] \\
| \;\; & * \; C \\
| \;\; & C; i \\
| \;\; & \textsc{If}(C)\{i_1\}\textsc{Else}\{i_2\} \\
| \;\; & \textsc{Return}(C) \\
| \;\; & \textsc{Decl}\; x = C \;\textsc{in}\{i\}
\end{aligned}
$$

B.5 Règles d'évaluation des erreurs

$\boxed{\Xi \to \Omega}$

$$\frac{}{\langle \Omega, m \rangle \to \Omega} \text{ (Exp-Err)} \qquad \frac{\langle e, m \rangle \to \Omega}{\langle C(\!|e|\!), m \rangle \to \Omega} \text{ (Eval-Err)}$$

$$\frac{\langle i, m \rangle \to \Omega}{\langle \textsc{Decl}\; x = v \;\textsc{in}\{i\}, m \rangle \to \Omega} \text{ (Decl-Err)}$$

$$\frac{m' = \text{Push}(m, ((a_1 \mapsto v_1), \ldots, (a_n \mapsto v_n))) \qquad \langle i, m' \rangle \to \Omega}{\langle \text{fun}(a_1, \ldots, a_n)\{i\}(v_1, \ldots, v_n), m \rangle \to \Omega} \text{ (Exp-Call-Err)}$$

B.6 Règles d'évaluation des valeurs gauches et expressions

$\boxed{\langle e, m\rangle \to \langle e', m'\rangle}$

$$\frac{\langle i, m\rangle \to \langle i', m'\rangle}{\langle C(\!|i|\!), m\rangle \to \langle C(\!|i'|\!), m'\rangle}\ (\textsc{Ctx})$$

$$\frac{\begin{array}{c} m_1 = \text{Push}(m_0, ((a_1 \mapsto v_1), \ldots, (a_n \mapsto v_n))) \\ \langle i, m_1\rangle \to \langle i', m_2\rangle \qquad \forall\, i \in [1; n], v'_i = m_2[(|m_2|, a_i)]_A \qquad m_3 = \text{Pop}(m_2) \end{array}}{\langle \text{fun}(a_1, \ldots, a_n)\{i\}(v_1, \ldots, v_n), m_0\rangle \to \langle \text{fun}(a_1, \ldots, a_n)\{i'\}(v'_1, \ldots, v'_n), m_3\rangle}\ (\textsc{Exp-Call-Ctx})$$

$\boxed{\langle l\, v, m\rangle \to \langle \varphi, m\rangle}$

$$\frac{a = \text{Lookup}(x, m)}{\langle x, m\rangle \to \langle a, m\rangle}\ (\textsc{Phi-Var}) \qquad \frac{v = \widehat{\&}\ \varphi}{\langle *\ v, m\rangle \to \langle \varphi, m\rangle}\ (\textsc{Phi-Deref})$$

$$\frac{v = \widehat{\text{NULL}}}{\langle *\ v, m\rangle \to \Omega_{ptr}}\ (\textsc{Phi-Deref-Null}) \qquad \frac{}{\langle \varphi.l_S, m\rangle \to \langle \widehat{\varphi.l}, m\rangle}\ (\textsc{Phi-Struct})$$

$$\frac{}{\langle \varphi[n], m\rangle \to \langle \widehat{\varphi[n]}, m\rangle}\ (\textsc{Phi-Array})$$

$\boxed{\langle e, m\rangle \to \langle v, m\rangle}$

$$\frac{}{\langle c, m\rangle \to \langle \widehat{c}, m\rangle}\ (\textsc{Exp-Cst}) \qquad \frac{}{\langle f, m\rangle \to \langle \widehat{f}, m\rangle}\ (\textsc{Exp-Fun}) \qquad \frac{}{\langle \varphi, m\rangle \to \langle m[\varphi]_\Phi, m\rangle}\ (\textsc{Exp-Lv})$$

$$\frac{}{\langle \boxminus\ v, m\rangle \to \langle \widehat{\boxminus}\ v, m\rangle}\ (\textsc{Exp-UnOp}) \qquad \frac{}{\langle v_1 \boxplus v_2, m\rangle \to \langle v_1 \widehat{\boxplus} v_2, m\rangle}\ (\textsc{Exp-BinOp})$$

$$\frac{}{\langle \&\, \varphi, m\rangle \to \langle \widehat{\&}\, \varphi, m\rangle}\ (\textsc{Exp-Addr}) \qquad \frac{}{\langle \varphi \leftarrow v, m\rangle \to \langle v, m[\varphi \leftarrow v]_\Phi\rangle}\ (\textsc{Exp-Set})$$

$$\frac{}{\langle \{l_1 : v_1; \ldots; l_n : v_n\}, m\rangle \to \langle \{l_1 : \widehat{v_1; \ldots; l_n} : v_n\}, m\rangle}\ (\textsc{Exp-Struct})$$

$$\frac{}{\langle [v_1, \ldots, v_n], m\rangle \to \langle [\widehat{v_1, \ldots, v_n}], m\rangle}\ (\textsc{Exp-Array})$$

$$\frac{m' = \text{Cleanup}(m) \qquad v' = \text{CleanV}_{|m|}(v)}{\langle \text{fun}(a_1, \ldots, a_n)\{\textsc{Return}(v)\}(v_1, \ldots, v_n), m\rangle \to \langle v', m'\rangle}\ (\textsc{Exp-Call-Return})$$

B.7 Règles d'évaluation des instructions, phrases et programmes

$$\boxed{\langle i, m\rangle \to \langle i', m\rangle}$$

$$\frac{}{\langle(\textsc{Pass}; i), m\rangle \to \langle i, m\rangle}\ (\textsc{Seq}) \qquad \frac{}{\langle v, m\rangle \to \langle \textsc{Pass}, m\rangle}\ (\textsc{Exp})$$

$$\frac{m' = \text{CleanVar}(m - x, (|m|, x))}{\langle \textsc{Decl}\ x = v\ \textsc{in}\{\textsc{Pass}\}, m\rangle \to \langle \textsc{Pass}, m'\rangle}\ (\textsc{Decl-Pass})$$

$$\frac{m' = \text{CleanVar}(m - x, (|m|, x)) \qquad v'' = \text{CleanVarV}(v', (|m|, x))}{\langle \textsc{Decl}\ x = v\ \textsc{in}\{\textsc{Return}(v')\}, m\rangle \to \langle \textsc{Return}(v''), m'\rangle}\ (\textsc{Decl-Return})$$

$$\frac{\begin{array}{c} m' = \text{Extend}(m, x \mapsto v) \\ \langle i, m'\rangle \to \langle i', m''\rangle \qquad v' = m''[(|m''|, x)]_A \qquad m''' = m'' - x \end{array}}{\langle \textsc{Decl}\ x = v\ \textsc{in}\{i\}, m\rangle \to \langle \textsc{Decl}\ x = v'\ \textsc{in}\{i'\}, m'''\rangle}\ (\textsc{Decl-Ctx})$$

$$\frac{}{\langle \textsc{If}(0)\{i_t\}\textsc{Else}\{i_f\}, m\rangle \to \langle i_f, m\rangle}\ (\textsc{If-False}) \qquad \frac{v \neq 0}{\langle \textsc{If}(v)\{i_t\}\textsc{Else}\{i_f\}, m\rangle \to \langle i_t, m\rangle}\ (\textsc{If-True})$$

$$\frac{}{\langle \textsc{While}(e)\{i\}, m\rangle \to \langle \textsc{If}(e)\{i; \textsc{While}(e)\{i\}\}, m\rangle}\ (\textsc{While})$$

$$\frac{}{\langle \textsc{Return}(v); i, m\rangle \to \langle \textsc{Return}(v), m\rangle}\ (\textsc{Return})$$

$$\boxed{m \Vdash p \to m'}$$

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle}{m \Vdash e \to m'}\ (\text{ET-Exp})$$

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle \qquad m' = (s, g) \qquad m'' = (s, (x \mapsto v) :: g)}{m \Vdash x = e \to m''}\ (\text{ET-Var})$$

$$\boxed{\Vdash P \to^* m}$$

$$\frac{([\,], [\,]) \Vdash p_1 \to m_1 \qquad m_1 \Vdash p_2 \to m_2 \qquad \ldots \qquad m_{n-1} \Vdash p_n \to m_n}{\Vdash p_1, \ldots, p_n \to^* m}\ (\textsc{Prog})$$

B.8 Règles d'évaluation des extensions noyau

$$\frac{}{\langle \Diamond\ \varphi, m\rangle \rightarrow \langle \widehat{\&}\ (\widehat{\Diamond}\ \varphi), m\rangle}\ (\textsc{Phi-User})$$

$$\frac{v = m[\varphi_s]_\Phi \qquad m' = m[\varphi_d \leftarrow v]_\Phi}{\langle \text{copy\_from\_user}(\widehat{\&}\ \varphi_d, \widehat{\&}\ (\widehat{\Diamond}\ \varphi_s)), m\rangle \rightarrow \langle 0, m'\rangle}\ (\textsc{User-Get-OK})$$

$$\frac{\nexists\ \varphi_s.\varphi = \widehat{\Diamond}\ \varphi_s}{\langle \text{copy\_from\_user}(\widehat{\&}\ \varphi_d, \widehat{\&}\ \varphi), m\rangle \rightarrow \langle -14, m\rangle}\ (\textsc{User-Get-Err})$$

$$\frac{v = m[\varphi_s]_\Phi \qquad m' = m[\varphi_d \leftarrow v]_\Phi}{\langle \text{copy\_to\_user}(\widehat{\&}\ (\widehat{\Diamond}\ \varphi_d), \widehat{\&}\ \varphi_s), m\rangle \rightarrow \langle 0, m'\rangle}\ (\textsc{User-Put-OK})$$

$$\frac{\nexists\ \varphi_d.\varphi = \widehat{\Diamond}\ \varphi_d}{\langle \text{copy\_to\_user}(\widehat{\&}\ \varphi, \widehat{\&}\ \varphi_s), m\rangle \rightarrow \langle -14, m\rangle}\ (\textsc{User-Put-Err})$$

ANNEXE C

RÈGLES DE TYPAGE

On rappelle ici l'ensemble des règles de typage décrites dans les chapitres 5 et 6.

C.1 Règles de typage des constantes et valeurs gauches

$\boxed{\Gamma \vdash c : t}$

$$\frac{}{\Gamma \vdash n : \text{INT}}\ (\text{CST-INT}) \qquad \frac{}{\Gamma \vdash d : \text{FLOAT}}\ (\text{CST-FLOAT}) \qquad \frac{}{\Gamma \vdash \text{NULL} : t*}\ (\text{CST-NULL})$$

$$\frac{}{\Gamma \vdash () : \text{UNIT}}\ (\text{CST-UNIT})$$

$\boxed{\Gamma \vdash lv : t}$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t}\ (\text{LV-VAR}) \qquad \frac{\Gamma \vdash e : t*}{\Gamma \vdash *e : t}\ (\text{LV-DEREF}) \qquad \frac{\Gamma \vdash e : \text{INT} \qquad \Gamma \vdash lv : t[\,]}{\Gamma \vdash lv[e] : t}\ (\text{LV-INDEX})$$

$$\frac{(l,t) \in S \qquad \Gamma \vdash lv : S}{\Gamma \vdash lv.l_S : t}\ (\text{LV-FIELD})$$

C.2 Règles de typage des opérateurs

$\boxed{\Gamma \vdash \boxminus\, e : t}$

$$\frac{\Gamma \vdash e : \textsc{Int}}{\Gamma \vdash +e : \textsc{Int}}\ (\textsc{Unop-Plus-Int}) \qquad \frac{\Gamma \vdash e : \textsc{Float}}{\Gamma \vdash +.e : \textsc{Float}}\ (\textsc{Unop-Plus-Float})$$

$$\frac{\Gamma \vdash e : \textsc{Int}}{\Gamma \vdash -e : \textsc{Int}}\ (\textsc{Unop-Minus-Int}) \qquad \frac{\Gamma \vdash e : \textsc{Float}}{\Gamma \vdash -.e : \textsc{Float}}\ (\textsc{Unop-Minus-Float})$$

$$\frac{\boxminus \in \{\sim, !\} \qquad \Gamma \vdash e : \textsc{Int}}{\Gamma \vdash \boxminus\, e : \textsc{Int}}\ (\textsc{Unop-Not})$$

$\boxed{\Gamma \vdash e_1 \boxplus e_2 : t}$

$$\frac{\boxplus \in \{+, -, \times, /, \&, |, \hat{}, \&\&, ||, \ll, \gg, \leq, \geq, <, >\} \qquad \Gamma \vdash e_1 : \textsc{Int} \qquad \Gamma \vdash e_2 : \textsc{Int}}{\Gamma \vdash e_1 \boxplus e_2 : \textsc{Int}}\ (\textsc{Op-Int})$$

$$\frac{\boxplus \in \{+., -., \times., /., \leq ., \geq ., < ., > .\} \qquad \Gamma \vdash e_1 : \textsc{Float} \qquad \Gamma \vdash e_2 : \textsc{Float}}{\Gamma \vdash e_1 \boxplus e_2 : \textsc{Float}}\ (\textsc{Op-Float})$$

$$\frac{\boxplus \in \{=, \neq\} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \qquad \textsc{Eq}(t)}{\Gamma \vdash e_1 \boxplus e_2 : \textsc{Int}}\ (\textsc{Op-Eq})$$

$$\frac{\boxplus \in \{+_p, -_p\} \qquad \Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \textsc{Int}}{\Gamma \vdash e_1 \boxplus e_2 : t*}\ (\textsc{Ptr-Arith})$$

$\boxed{\textsc{Eq}(t)}$

$$\frac{t \in \{\textsc{Int}, \textsc{Float}\}}{\textsc{Eq}(t)}\ (\textsc{Eq-Num}) \qquad \frac{}{\textsc{Eq}(t*)}\ (\textsc{Eq-Ptr}) \qquad \frac{\textsc{Eq}(t)}{\textsc{Eq}(t[\,])}\ (\textsc{Eq-Array})$$

$$\frac{\forall i \in [1; n].\textsc{Eq}(t_i)}{\textsc{Eq}(\{l_1 : t_1; \ldots l_n : t_n\})}\ (\textsc{Eq-Struct})$$

C.3 Règles de typage des expressions et instructions

$\boxed{\Gamma \vdash e : t}$

$$\frac{\Gamma \vdash lv : t}{\Gamma \vdash \&lv : t*}\ (\textsc{Addr}) \qquad \frac{\forall i \in [1;n], \Gamma \vdash e_i : t_i}{\Gamma \vdash \{l_1 : e_1; \ldots; l_n : e_n\} : \{l_1 : t_1; \ldots; l_n : t_n\}}\ (\textsc{Struct})$$

$$\frac{\Gamma \vdash e : (t_1, \ldots, t_n) \to t \qquad \forall i \in [1;n], \Gamma \vdash e_i : t_i}{\Gamma \vdash e(e_1, \ldots, e_n) : t}\ (\textsc{Call}) \qquad \frac{\Gamma \vdash lv : t \qquad \Gamma \vdash e : t}{\Gamma \vdash lv \leftarrow e : t}\ (\textsc{Set})$$

$$\frac{\forall i \in [1;n], \Gamma \vdash e_i : t}{\Gamma \vdash [e_1; \ldots; e_n] : t[\,]}\ (\textsc{Array})$$

$$\frac{\Gamma = (\Gamma_G, \Gamma_L) \qquad \Gamma' = (\Gamma_G, [a_1 : t_1, \ldots, a_n : t_n, \underline{R} : t]) \qquad \Gamma' \vdash i}{\Gamma \vdash \text{fun}(a_1, \ldots, a_n)\{i\} : (t_1, \ldots, t_n) \to t}\ (\textsc{Fun})$$

$\boxed{\Gamma \vdash i}$

$$\frac{}{\Gamma \vdash \textsc{Pass}}\ (\textsc{Pass}) \qquad \frac{\Gamma \vdash i_1 \qquad \Gamma \vdash i_2}{\Gamma \vdash i_1; i_2}\ (\textsc{Seq}) \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash e}\ (\textsc{Exp})$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma, \text{local } x : t \vdash i}{\Gamma \vdash \textsc{Decl}\ x = e\ \textsc{in}\{i\}}\ (\textsc{Decl}) \qquad \frac{\Gamma \vdash e : \textsc{Int} \qquad \Gamma \vdash i_1 \qquad \Gamma \vdash i_2}{\Gamma \vdash \textsc{If}(e)\{i_1\}\textsc{Else}\{i_2\}}\ (\textsc{If})$$

$$\frac{\Gamma \vdash e : \textsc{Int} \qquad \Gamma \vdash i}{\Gamma \vdash \textsc{While}(e)\{i\}}\ (\textsc{While}) \qquad \frac{\underline{R} : t \in \Gamma \qquad \Gamma \vdash e : t}{\Gamma \vdash \textsc{Return}(e)}\ (\textsc{Return})$$

$\boxed{\Gamma \vdash p \to \Gamma'}$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e \to \Gamma}\ (\text{T-}\textsc{Exp}) \qquad \frac{\Gamma \vdash e : t \qquad \Gamma' = \Gamma, \text{global } x : t}{\Gamma \vdash x = e \to \Gamma'}\ (\text{T-}\textsc{Var})$$

$\boxed{\Gamma \vdash P}$

$$\frac{[\,] \vdash p_1 \to \Gamma_1 \qquad \Gamma_1 \vdash p_2 \to \Gamma_2 \qquad \ldots \qquad \Gamma_{n-1} \vdash p_n \to \Gamma_n}{\vdash p_1, \ldots, p_n}\ (\textsc{Prog})$$

C.4 Règles de typage des valeurs

$\boxed{m \vDash v : \tau}$

$$\frac{}{m \vDash n : \text{INT}}\ (\text{S-INT}) \qquad \frac{}{m \vDash d : \text{FLOAT}}\ (\text{S-FLOAT}) \qquad \frac{}{m \vDash () : \text{UNIT}}\ (\text{S-UNIT})$$

$$\frac{}{m \vDash \text{NULL} : \tau *}\ (\text{S-NULL}) \qquad \frac{m \vDash_\Phi \varphi : \tau}{m \vDash \widehat{\&}\,\varphi : \tau *}\ (\text{S-PTR}) \qquad \frac{\forall i \in [1;n].m \vDash v_i : \tau}{m \vDash [\widehat{v_1;\ldots;v_n}] : \tau[\,]}\ (\text{S-ARRAY})$$

$$\frac{\forall i \in [1;n].m \vDash v_i : \tau_i}{m \vDash \{l_1 : \widehat{v_1;\ldots;l_n} : v_n\} : \{l_1 : \tau_1;\ldots;l_n : \tau_n\}}\ (\text{S-STRUCT})$$

$$\frac{}{m \vDash \text{fun}(x_1,\ldots,x_n)\{i\} : \text{FUN}_n}\ (\text{S-FUN})$$

C.5 Règles de typage des extensions noyau

$$\frac{\Gamma \vdash lv : t}{\Gamma \vdash \diamondsuit\ lv : t\,@}\ (\text{ADDR-USER}) \qquad \frac{\Gamma \vdash e_d : t * \qquad \Gamma \vdash e_s : t\,@}{\Gamma \vdash \text{copy\_from\_user}(e_d, e_s) : \text{INT}}\ (\text{USET-GET})$$

$$\frac{\Gamma \vdash e_d : t\,@ \qquad \Gamma \vdash e_s : t *}{\Gamma \vdash \text{copy\_to\_user}(e_d, e_s) : \text{INT}}\ (\text{USER-PUT})$$

ANNEXE D

PREUVES

On présente ici les preuves de certains résultats établis dans le manuscrit : le caractère bien fondé de la composition de deux lentilles, et les théorèmes de sûreté du typage.

D.1 Composition de lentilles

Démonstration. On cherche à prouver que si $\mathscr{L}_1 \in \text{LENS}_{A,B}$ et $\mathscr{L}_2 \in \text{LENS}_{B,C}$, alors $\mathscr{L} = \mathscr{L}_1 \ggg \mathscr{L}_2 \in \text{LENS}_{A,C}$ ($\ggg$ est la composition de lentilles, définie page 39).

Il suffit pour cela d'établir les trois propriétés caractéristiques qui définissent les lentilles : PUTPUT, GETPUT et PUTGET. Cela est essentiellement calculatoire : on utilise la définition de $\ggg$ et les propriétés caractéristiques sur $\mathscr{L}_1$ et $\mathscr{L}_2$.

PutPut

$$
\begin{aligned}
& \text{put}_{\mathscr{L}}(a', \text{put}_{\mathscr{L}}(a, r)) \\
= \ & \text{put}_{\mathscr{L}}(a', \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r)), r)) \\
& \{ \text{définition de put}_{\mathscr{L}} \} \\
= \ & \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a', \text{get}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r)), r))), \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r)), r)) \\
& \{ \text{définition de put}_{\mathscr{L}} \} \\
= \ & \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a', \text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r))), \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r)), r)) \\
& \{ \text{GETPUT sur } \mathscr{L}_1 \} \\
= \ & \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a', \text{get}_{\mathscr{L}_1}(r)), \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r)), r)) \\
& \{ \text{PUTPUT sur } \mathscr{L}_2 \} \\
= \ & \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a', \text{get}_{\mathscr{L}_1}(r)), r) \\
& \{ \text{PUTPUT sur } \mathscr{L}_1 \} \\
= \ & \text{put}_{\mathscr{L}}(a', r) \\
& \{ \text{définition de} \ggg \}
\end{aligned}
$$

GetPut

$$\begin{aligned}
\text{put}_{\mathscr{L}}(\text{get}_{\mathscr{L}}(r), r) &= \text{put}_{\mathscr{L}}(\text{get}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_1}(r)), r) && \{ \text{ définition de get}_{\mathscr{L}} \ \} \\
&= \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_1}(r)), \text{get}_{\mathscr{L}_1}(r)), r) && \{ \text{ définition de put}_{\mathscr{L}} \ \} \\
&= \text{put}_{\mathscr{L}_1}(\text{get}_{\mathscr{L}_1}(r), r) && \{ \text{ GETPUT sur } \mathscr{L}_2 \ \} \\
&= r && \{ \text{GETPUT sur } \mathscr{L}_1 \}
\end{aligned}$$

PutGet

$$\begin{aligned}
\text{get}_{\mathscr{L}}(\text{put}_{\mathscr{L}}(a, r)) &= \text{get}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}}(a, r))) && \{ \text{définition de get}_{\mathscr{L}} \} \\
&= \text{get}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r)), r))) && \{ \text{définition de put}_{\mathscr{L}} \} \\
&= \text{get}_{\mathscr{L}_2}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r))) && \{ \text{PUTGET sur } \mathscr{L}_1 \} \\
&= a && \{ \text{PUTGET sur } \mathscr{L}_2 \}
\end{aligned}$$

□

D.2 Progrès

On rappelle l'énoncé du théorème 5.1.

Théorème D.1 (Progrès)**.** *Supposons que* $\Gamma \vdash i$*. Soit* m *un état mémoire tel que* $\Gamma \vDash m$*.*
Alors l'un des cas suivants est vrai :

- $i = \text{PASS}$
- $\exists v, i = \text{RETURN}(v)$
- $\exists (i', m'), \langle i, m \rangle \rightarrow \langle i', m' \rangle$
- $\exists \Omega \in \{\Omega_{div}, \Omega_{array}, \Omega_{ptr}\}, \langle i, m \rangle \rightarrow \Omega$

✧ ✧ ✧

Supposons que $\Gamma \vdash e : t$*. Soit* m *un état mémoire tel que* $\Gamma \vDash m$*. Alors l'un des cas suivant est vrai :*

- $\exists v \neq \Omega, e = v$
- $\exists (e', m'), \langle e, m \rangle \rightarrow \langle e', m' \rangle$
- $\exists \Omega \in \{\Omega_{div}, \Omega_{array}, \Omega_{ptr}\}, \langle e, m \rangle \rightarrow \Omega$

✧ ✧ ✧

Supposons que $\Gamma \vdash l\,v : t$*. Soit* m *un état mémoire tel que* $\Gamma \vDash m$*.*
Alors l'un des cas suivants est vrai :

- $\exists \varphi, l\,v = \varphi$
- $\exists (l\,v', m'), \langle l\,v, m \rangle \rightarrow \langle l\,v', m' \rangle$
- $\exists \Omega \in \{\Omega_{div}, \Omega_{array}, \Omega_{ptr}\}, \langle l\,v, m \rangle \rightarrow \Omega$

C'est-à-dire, soit :

- l'entité (instruction, expression ou valeur gauche) est complètement évaluée.
- un pas d'évaluation est possible.
- une erreur de division, tableau ou pointeur se produit.

Démonstration. On procède par induction sur la dérivation du jugement de typage. Puisque les jugements $\Gamma \vdash i$, $\Gamma \vdash e : t$ et $\Gamma \vdash l\,v : t$ sont interdépendants, on traite tous les cas par récursion mutuelle.

Le squelette de cette preuve est une analyse de cas selon la dernière règle utilisée. La plupart des cas ont la même forme : on utilise l'hypothèse de récurrence sur les sous-éléments syntaxiques (en appliquant éventuellement le lemme 5.1 d'inversion pour établir qu'ils sont bien typés). Dans le cas « valeur », on appelle une règle qui permet de transformer une opération syntaxique en opération sémantique (par exemple, on transforme le + unaire en un $\widehat{+}$ sémantique). Dans le cas « évaluation », on applique la règle CTX avec un contexte particulier qui permet de passer d'un jugement $\langle a, m\rangle \to \langle a', m'\rangle$ à un jugement $\langle b, m\rangle \to \langle b', m'\rangle$ (où a apparaît dans b). Enfin, dans le cas « erreur », on utilise EVAL-ERR avec ce même contexte C.

Ceci est valable pour la majorité des cas. Il faut faire attention en particulier aux opérations sémantiques qui peuvent produire des erreurs (comme la division, ou l'opérateur Lookup$(\cdot,\cdot)$).

Instructions

PASS : Ce cas est immédiat.

RETURN : Partant de $i = \text{RETURN}(e)$, on applique le lemme d'inversion. Il nous donne l'existence de t tel que $\Gamma \vdash e : t$. On applique alors l'hypothèse de récurrence à e.

- $e = v$. Alors $i = \text{RETURN}(v)$, ce qui nous permet de conclure.
- $\langle e, m\rangle \to \langle e', m'\rangle$. Alors en appliquant CTX avec $C = \text{RETURN}(\bullet)$, on conclut que $\langle \text{RETURN}(e), m\rangle \to \langle \text{RETURN}(e'), m'\rangle$.
- $\langle e, m\rangle \to \Omega$. On applique EVAL-ERR avec ce même C.

SEQ : Avec $i = i_1; i_2$, on applique l'hypothèse de récurrence à i_1.

- $i_1 = \text{PASS}$. On peut donc appliquer la règle SEQ et donc $\langle i, m\rangle \to \langle i_2, m\rangle$.
- $i_1 = \text{RETURN}(v)$. Alors on peut appliquer la règle RETURN : $\langle i, m\rangle \to \langle \text{RETURN}(v), m\rangle$.
- $\langle i_1, m\rangle \to \langle i_1', m'\rangle$. Soit $C = \bullet; i_2$. Par CTX il vient $\langle i, m\rangle \to \langle i_1'; i_2, m'\rangle$.
- $\langle i_1, m\rangle \to \Omega$. Avec ce même C dans EVAL-ERR on trouve $\langle i, m\rangle \to \Omega$.

EXP : Ici $i = e$. On peut appliquer l'hypothèse de récurrence à e qui est « plus petit » que i ($i ::= e$ introduit un constructeur implicite).

- $e = v$. Alors on peut appliquer EXP : $\langle e, m\rangle \to \langle \text{PASS}, m\rangle$.
- $\langle e, m\rangle \to \langle e', m'\rangle$. Alors $\langle i, m\rangle \to \langle e', m'\rangle$ (cela revient à appliquer CTX au constructeur implicite mentionné ci-dessus).
- $\langle e, m\rangle \to \Omega$. C'est-à-dire $\langle i, m\rangle \to \Omega$.

DECL : Ici $i = \text{DECL}\ x = e\ \text{IN}\{i'\}$. On commence par appliquer l'hypothèse de récurrence à e.

- $e = v$. On applique alors l'hypothèse de récurrence à i' sous $\Gamma' = \Gamma, \text{local}\ x : t$ et avec $m' = \text{Extend}(m, x \mapsto v)$.
  - $i' = \text{PASS}$. Dans ce cas la règle DECL-PASS s'applique.
  - $i' = \text{RETURN}(v)$. Idem avec DECL-RETURN.

  - $\langle i', m' \rangle \to \langle i'', m'' \rangle$. On peut alors appliquer la règle DECL-CTX.
  - $\langle i', m' \rangle \to \Omega$. On applique DECL-ERR.
- $\langle e, m \rangle \to \langle e', m' \rangle$. On pose $C = \text{DECL } x = \bullet \text{ IN}\{i'\}$ et on conclut avec la règle CTX.
- $\langle e, m \rangle \to \Omega$. Idem avec EVAL-ERR.

IF : Ici $i = \text{IF}(e)\{i_1\}\text{ELSE}\{i_2\}$. On applique l'hypothèse de récurrence à e.

- $e = v$.
  Si $v \neq 0$, on applique IF-TRUE. Dans le cas contraire, on applique IF-FALSE.
- $\langle e, m \rangle \to \langle e', m' \rangle$. On pose $C = \text{IF}(\bullet)\{i_1\}\text{ELSE}\{i_2\}$ et on conclut avec CTX.
- $\langle e, m \rangle \to \Omega$. Avec ce même C et EVAL-ERR.

WHILE : Ce cas est direct : on applique la règle d'évaluation WHILE.

Expressions

CST-INT : e est alors de la forme n, qui est une valeur.

CST-FLOAT : e est alors de la forme d, qui est une valeur.

CST-NULL : e est alors égale à NULL, qui est une valeur.

CST-UNIT : e est alors égale à `()`, qui est une valeur.

FUN : Ce cas est direct : la règle EXP-FUN s'applique.

OP-INT : Cela implique que $e = e_1 \boxplus e_2$. Par le lemme 5.1, on en déduit que $\Gamma \vdash e_1 : \text{INT}$ et $\Gamma \vdash e_2 : \text{INT}$.

Appliquons l'hypothèse de récurrence sur e_1. Trois cas peuvent se produire.

- $e_1 = v_1$. On a alors $\langle e_1, m \rangle \to \langle v_1, m' \rangle$ avec $m' = m$.
  On applique l'hypothèse de récurrence à e_2.
  - $e_2 = v_2$: alors $\langle e_2, m' \rangle \to \langle v_2, m'' \rangle$ avec $m'' = m$. On peut alors appliquer EXP-BINOP, sauf dans le cas d'une division par zéro ($\boxplus \in \{/; \%; /.\}$ et $v_2 = 0$) où alors $v_1 \mathbin{\widehat{\boxplus}} v_2 = \Omega_{div}$. Dans ce cas, on a alors par EXP-ERR $\langle e, m \rangle \to \Omega_{div}$. Notons que comme les opérandes sont bien typées, Ω_{typ} ne peut pas être levée.
  - $\exists (e_2', m''), \langle e_2, m' \rangle \to \langle e_2', m'' \rangle$.
    En appliquant CTX avec $C = v_1 \boxplus \bullet$, on en déduit $\langle v_1 \boxplus e_2, m' \rangle \to \langle v_1 \boxplus e_2', m'' \rangle$ soit $\langle e, m \rangle \to \langle v_1 \boxplus e_2', m'' \rangle$.
  - $\langle e_2, m' \rangle \to \Omega$. De EVAL-ERR avec $C = v_1 \boxplus \bullet$ vient alors $\langle e, m \rangle \to \Omega$.
- $\exists (e_1', m'), \langle e_1, m \rangle \to \langle e_1', m' \rangle$. En appliquant CTX avec $C = \bullet \boxplus e_2$, on obtient $\langle e_1 \boxplus e_2, m \rangle \to \langle e_1' \boxplus e_2, m' \rangle$, ou $\langle e, m \rangle \to \langle e_1' \boxplus e_2, m' \rangle$.
- $\langle e_1, m \rangle \to \Omega$. D'après EVAL-ERR avec $C = \bullet \boxplus e_2$, on a $\langle e, m \rangle \to \Omega$.

OP-FLOAT : Ce cas est similaire à OP-INT.

OP-EQ : Ce cas est similaire à OP-INT.

UNOP-PLUS-INT : Alors $e = + e_1$. En appliquant l'hypothèse d'induction sur e_1 :

- soit $e_1 = v_1$. Alors en appliquant EXP-UNOP, $\langle + v_1, m\rangle \to \langle \widehat{+} v_1, m\rangle$, c'est-à-dire en posant $v = \widehat{+} v_1$, $\langle e, m\rangle \to \langle v, m\rangle$.
- soit $\exists e_1', m', \langle e_1, m\rangle \to \langle e_1', m'\rangle$. Alors en appliquant CTX avec $C = + \bullet$, on obtient $\langle e, m\rangle \to \langle e_1', m'\rangle$.
- soit $\langle e_1, m\rangle \to \Omega$. De EVAL-ERR avec $C = + \bullet$ il vient$\langle e, m\rangle \to \Omega$.

UNOP-PLUS-FLOAT : Ce cas est similaire à UNOP-PLUS-INT.

UNOP-MINUS-INT : Ce cas est similaire à UNOP-PLUS-INT.

UNOP-MINUS-FLOAT : Ce cas est similaire à UNOP-PLUS-INT.

UNOP-NOT : Ce cas est similaire à UNOP-PLUS-INT.

ADDR : On applique l'hypothèse de récurrence à $l\,v$.

Les cas d'évaluation et d'erreur sont traités en appliquant respectivement CTX et EVAL-ERR avec $C = \&\bullet$. Dans le cas où $l\,v = \varphi$, on peut appliquer EXP-ADDR.

SET : On applique l'hypothèse de récurrence à $l\,v$.

- $l\,v = \varphi$. On applique l'hypothèse de récurrence à e.
  - $e = v$. Alors on peut appliquer EXP-SET.
  - $\langle e, m\rangle \to \langle e', m'\rangle$. On conclut avec $C = \varphi \leftarrow \bullet$.
  - $\langle e, m\rangle \to \Omega$. Idem.
- $\langle l\,v, m\rangle \to \langle l\,v', m'\rangle$. On conclut avec $C = \bullet \leftarrow e$.
- $\langle l\,v, m\rangle \to \Omega$. Idem.

ARRAY : On va appliquer l'hypothèse de récurrence à e_1, puis si $e_1 = v_1$, on l'applique à e_2, etc. Alors on se retrouve dans un des cas suivants :

- $\exists p \in [1; n], e_p', m : e_1 = v_1, \ldots, e_{p-1} = v_{p-1}, \langle e_p, m\rangle \to \langle e_p', m'\rangle$. Alors on peut appliquer CTX avec $C = [v_1, \ldots, v_{p-1}, \bullet, e_{p+1}, \ldots, e_n]$.
- $\exists p \in [1; n], \Omega : e_1 = v_1, \ldots, e_{p-1} = v_{p-1}, \langle e_p, m\rangle \to \Omega$. Dans ce cas EVAL-ERR est applicable avec ce même C.
- $e_1 = v_1, \ldots, e_n = v_n$. Alors on peut appliquer EXP-ARRAY en construisant un tableau.

STRUCT : Le schéma de preuve est similaire au cas ARRAY. En cas de pas d'évaluation ou d'erreur, on utilise le contexte $C = \{l_1 : v_1, \ldots, l_{p-1} : v_{p-1}, \bullet, l_{p+1} : e_{p+1}, \ldots, l_n : e_n\}$; et dans le cas où toutes les expressions sont évaluées, on applique EXP-STRUCT.

CALL : On commence par appliquer l'hypothèse de récurrence à e. Dans le cas d'un pas d'évaluation ou d'erreur, on applique respectivement CTX ou EVAL-ERR avec $C = \bullet(e_1, \ldots, e_n)$. Reste le cas où e est une valeur : d'après le lemme 5.2, e est de la forme $f = \text{fun}(\vec{a})\{i\}$.

Ensuite, appliquons le même schéma que pour ARRAY. En cas de pas d'évaluation ou d'erreur, on utilise CTX ou EVAL-ERR avec $C = f(v_1, \ldots, v_{p-1}, \bullet, e_{p+1}, \ldots, e_n)$. Le seul cas restant est celui où l'expression considérée a pour forme $f(v_1, \ldots, v_n)$ avec $f = \text{fun}(\vec{a})\{i\}$.

Soient $\Gamma' = (\Gamma_G, [a_1 : t_1, \ldots, a_n : t_n, \underline{R} : t])$ et $m_1 = \text{Push}(m_0, (a_1 \mapsto v_1, \ldots a_n \mapsto v_n))$ où $\Gamma = (\Gamma_G, \Gamma_L)$.

On applique alors l'hypothèse de récurrence à Γ', m_1 et i (le lemme d'inversion garantit que $\Gamma' \vdash i$).

- i = PASS. Ce cas est impossible puisqu'on prend l'hypothèse que les fonctions se terminent par une instruction RETURN$(\cdot)$ (page 56).
- $\langle i, m_1 \rangle \to \langle i', m_2 \rangle$.
  Alors on peut appliquer EXP-CALL-CTX.
- $\langle i, m \rangle \to \Omega$. On peut alors appliquer EXP-CALL-ERR.

Valeurs gauches

LV-VAR : Le but est d'appliquer PHI-VAR. La seule condition pour que cela soit possible est que Lookup(x, m) renvoie une adresse et non Ω_{var}.

Puisque $\Gamma \vdash x : t$, on peut appliquer le lemme 5.5 : x est soit une variable locale, soit une globale. Dans ces deux cas, Lookup(x, m) renvoie une adresse correcte.

LV-DEREF : Appliquons l'hypothèse de récurrence à e vue en tant qu'expression.

- $e = v$. Puisque $\Gamma \vdash v : t*$, on déduit du lemme 5.2 que $v = \text{NULL}$ ou $v = \widehat{\&}\,\varphi$.
  Dans le premier cas, puisque $\langle *\text{NULL}, m \rangle \to \Omega_{ptr}$, on a $\langle e, m \rangle \to \Omega_{ptr}$.
  Dans le second cas, PHI-DEREF s'applique.
- $\langle e, m \rangle \to \langle e', m' \rangle$. De CTX avec $C = *\bullet$, on obtient $\langle e, m \rangle \to \langle *e', m' \rangle$.
- $\langle e, m \rangle \to \Omega$. En appliquant EVAL-ERR avec $C = *\bullet$, on obtient $\langle e, m \rangle \to \Omega$.

LV-INDEX : De même, on applique l'hypothèse de récurrence à $l\,v$.

- $l\,v = v$.
  Comme $\Gamma \vdash v : t[]$, on déduit du lemme 5.2 que $v = [v_1; \ldots; v_p]$. Appliquons l'hypothèse de récurrence à e.
  - $e = v'$. Puisque $\Gamma \vdash e : \text{INT}$, on réapplique le lemme 5.2 et $v' = n$. D'après PHI-ARRAY, $\langle l\,v[e], m \rangle \to \langle [v_1; \ldots; v_p]\widehat{[n]}, m \rangle$. Deux cas sont à distinguer : si $n \in [0; p-1]$, la partie droite vaut v_{n+1} et donc $\langle l\,v[e], m \rangle \to \langle v_{n+1}, m \rangle$. Sinon elle vaut Ω_{array} et $\langle l\,v[e], m \rangle \to \Omega_{array}$ par EXP-ERR.
  - $\langle e, m \rangle \to \langle e', m' \rangle$. En appliquant CTX avec $C = v[\bullet]$, on en déduit
  - $\langle l\,v[e], m \rangle \to \langle l\,v[e'], m' \rangle$.
  - $\langle e, m \rangle \to \Omega$. Avec EVAL-ERR sous ce même contexte, $\langle l\,v[e], m \rangle \to \Omega$
- $\langle l\,v, m \rangle \to \langle e', m' \rangle$. On applique alors CTX avec $C = \bullet[e]$, et $\langle l\,v[e], m \rangle \to \langle e'[e], m' \rangle$.
- $\langle l\,v, m \rangle \to \Omega$. Toujours avec $C = \bullet[e]$, de EVAL-ERR il vient $\langle l\,v[e], m \rangle \to \Omega$.

LV-FIELD : On applique l'hypothèse de récurrence à $l\,v$.

- $l\,v = \varphi$ Alors PHI-STRUCT s'applique. Puisque $(l,t) \in S$, l'accès au champ l ne provoque pas d'erreur Ω_{field}. Donc $\langle e, m\rangle \to \langle \varphi[l], m\rangle$.
- $\langle l\,v, m\rangle \to \langle l\,v', m'\rangle$ En appliquant CTX avec $C = \bullet.l_S$, il vient $\langle l\,v, m\rangle \to \langle l\,v', m'\rangle$.
- $\langle l\,v, m\rangle \to \Omega$ En appliquant EVAL-ERR avec $C = \bullet.l_S$, on a $\langle l\,v, m\rangle \to \Omega$.

PTR-ARITH : Le schéma est similaire au cas OP-INT. Le seul cas intéressant arrive lorsque e_1 et e_2 sont des valeurs. D'après le lemme 5.2 :

- $e_1 = \text{NULL}$ ou $e_1 = \varphi$
- $e_2 = n$

D'après EXP-BINOP, $\langle e, m\rangle \to \langle e_1 \mathbin{\widehat{\boxplus}} n, m\rangle$.

On se réfère ensuite à la définition de $\widehat{\boxplus}$ (page 55) : si e_1 est de la forme $\varphi[m]$, alors $e_1 \mathbin{\widehat{\boxplus}} n = \varphi[m+n]$. Donc $\langle e, m\rangle \to \langle \varphi[m+n], m\rangle$.

Dans les autres cas ($e_1 = \text{NULL}$ ou $e_1 = \varphi$ avec φ pas de la forme $\varphi'[m]$), on a $e_1 \mathbin{\widehat{\boxplus}} n = \Omega_{ptr}$. Donc d'après EXP-ERR, $\langle e, m\rangle \to \Omega_{ptr}$.

□

D.3 Préservation

On rappelle l'énoncé du théorème 5.2.

Théorème D.2 (Préservation)**.** *Soient* Γ *un environnement de typage, et* m *un état mémoire tels que* $\Gamma \vDash m$.

Alors :

- *Si* $\Gamma \vdash l\,v : t$ *et* $\langle l\,v, m\rangle \to \langle \varphi, m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $m' \vDash_\Phi \varphi : \tau$ *où* $\tau \rhd t$.
- *Si* $\Gamma \vdash l\,v : t$ *et* $\langle l\,v, m\rangle \to \langle l\,v', m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $\Gamma \vdash l\,v' : t$.
- *Si* $\Gamma \vdash e : t$ *et* $\langle e, m\rangle \to \langle v, m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $m' \vDash v : \tau$ *où* $\tau \rhd t$.
- *Si* $\Gamma \vdash e : t$ *et* $\langle e, m\rangle \to \langle e', m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $\Gamma \vdash e' : t$.
- *Si* $\Gamma \vdash i$ *et* $\langle i, m\rangle \to \langle i', m'\rangle$, *alors* $\Gamma \vDash \text{Cleanup}(m')$ *et* $\Gamma \vdash i'$.

Autrement dit, si une construction est typable, alors un pas d'évaluation ne modifie pas son type et préserve le typage de la mémoire.

Démonstration. On procède par induction sur la dérivation de $\langle \cdot, m\rangle \to \langle \cdot, m'\rangle$. Plusieurs remarques sont à faire : d'abord, en ce qui concerne le typage de la mémoire, il suffit de montrer que $\Gamma \vDash m'$ car cela implique que $\Gamma \vDash \text{Cleanup}(m')$. Ensuite, la règle CTX est traitée à part, car elle peut être appliquée en contexte d'expression, d'instruction ou de valeur gauche. Enfin la règle TRANS ne pose pas de problème, il suffit d'appliquer l'hypothèse de récurrence à ses prémisses.

Cas $\Gamma \vdash l\,v : t$ et $\langle l\,v, m\rangle \to \langle \varphi, m'\rangle$

PHI-DEREF : On sait que $\Gamma \vdash *\,v : t$ où $v = \widehat{\&}\,\varphi$. Par inversion, $\Gamma \vdash v : t\,*$. Alors d'après le lemme 5.3, il existe τ' tel que $m \vDash v : \tau'$ et $\tau' \rhd t\,*$. Par inversion de la relation de typage sémantique, $\tau' = \tau\,*$ où $\tau \rhd t$. Alors par inversion de S-PTR, on obtient que $m \vDash_\Phi \varphi : \tau$.

PHI-VAR, PHI-STRUCT et PHI-ARRAY : Tout d'abord, la mémoire n'est pas modifiée donc il n'est pas nécessaire de montrer la compatibilité de m'. Ensuite, les prémisses de ces règles ont la forme φ , donc le lemme 5.6 s'applique avec la conclusion correcte.

Cas $\Gamma \vdash e : t$ et $\langle e, m\rangle \to \langle v, m'\rangle$

EXP-CST : Toutes les constantes sont des valeurs, donc le lemme 5.3 peut s'appliquer : $\tau = \text{Repr}(t)$ convient.

EXP-FUN : Idem : le lemme de réprésentabilité nous donne un candidat $\tau = \text{Repr}(t)$ qui convient.

EXP-LV : Puisque $\Gamma \vdash \varphi : t$ et $\Gamma \vDash m$, on a d'après le lemme 5.6 : $m \vDash v : \tau$ où $v = m[\varphi]$ avec $\tau \rhd t$.

EXP-UNOP : Il vient des définitions des différents opérateurs $\widehat{\boxminus}$ que $\Gamma \vdash \widehat{\boxminus}\ v : \tau$ avec $\tau \rhd t$.

EXP-BINOP : Idem avec les définitions des opérateurs $\widehat{\boxplus}$.

EXP-ADDR : On peut appliquer le lemme 5.3, qui nous donne un τ qui convient.

EXP-SET : Deux propriétés sont à prouver. D'une part, $\Gamma \vdash v : t$, et d'autre part, $\Gamma \vDash m'$ où $m' = m[\varphi \leftarrow v]$. Tout d'abord, le lemme d'inversion appliqué à $\Gamma \vdash \varphi \leftarrow v : t$ nous donne que $\Gamma \vdash \varphi : t$ et $\Gamma \vdash v : t$. Ensuite, comme $\Gamma \vdash \varphi : t$ et $\Gamma \vDash m$, on peut appliquer le lemme 5.3 : il existe τ tel que $m \vDash v : \tau$ et $\tau \rhd t$. On peut donc appliquer le lemme 5.6, qui nous permet de conclure que $\Gamma \vDash m'$.

EXP-STRUCT : Le lemme 5.3 s'applique à ce cas.

EXP-ARRAY : Idem, on conclut grâce au lemme de représentabilité.

EXP-CALL-RETURN : Par inversion, il vient que $\Gamma \vdash \text{fun}(a_1, \ldots, a_n)\{i\} : (t_1, \ldots, t_n) \to t'$ et $\forall i \in [1; n], \Gamma \vdash v_i : t_i$.

Posons $\Gamma' = (\Gamma_G, [a_1 : t_1, \ldots, a_n : t_n, \underline{R} : t])$ où $\Gamma = (\Gamma_G, \Gamma_L)$. Alors par inversions successives on obtient que $\Gamma' \vdash \text{RETURN}(v)$ et $\Gamma' \vdash v : t$.

Si on définit $m'' = \text{Push}(m, a_1 \mapsto v_1, \ldots, a_n \mapsto v_n)$, alors par M-PUSH on obtient que $\Gamma' \vDash m''$. Donc, par le lemme 5.3, il existe τ tel que $m'' \vDash v : \tau$ où $\tau \rhd t$.

Il reste à montrer que $m' \vDash v' : \tau$.

On distingue selon la forme de v. On applique un raisonnement similaire à celui de la preuve du lemme 5.6 : soit v est une référence au cadre nettoyé, et dans ce cas $v' = \text{NULL}$ et τ est un type pointeur, soit $v' = v$. Dans tous les cas on conclut car $m' \vDash v' : \tau$.

Cas $\Gamma \vdash i$ et $\langle i, m\rangle \to \langle i', m'\rangle$

SEQ : D'après le lemme d'inversion, $\Gamma \vdash i$.

EXP : D'après PASS, $\Gamma \vdash \text{PASS}$.

DECL-PASS : $\Gamma \vdash \text{PASS}$ est immédiat, et $\Gamma \vDash m'$ est établi par M-DECL suivie de M-DECLCLEAN. On a bien $x \notin \Gamma$ car les déclarations de variable ne peuvent pas masquer de variables visibles existantes (page 57).

DECL-RETURN : La compatibilité mémoire se démontre de la même manière que pour DECL-PASS. Il reste à montrer que $\Gamma \vdash \text{RETURN}(v'')$, ce qui fait de manière analogue au cas EXP-CALL-RETURN.

DECL-CTX : On part de $\Gamma \vdash \text{DECL } x = v \text{ IN}\{i\}$. Par inversion, il existe t tel que $\Gamma \vdash v : t$ et $\Gamma' \vdash i$ où $\Gamma' = \Gamma, \text{local } x : t$.

Comme $\Gamma \vDash m$, le lemme 5.3 s'applique : il existe τ tel que $m \vDash v : \tau$ où $\tau \rhd t$. De plus $x \notin \Gamma$ car il n'y a pas de masquage (page 57).

En appliquant M-DECL, on obtient donc que $\Gamma' \vDash m'$.

On applique alors l'hypothèse d'induction à $\langle i, m' \rangle \to \langle i', m'' \rangle$. Il vient que $\Gamma' \vdash i'$ et $\Gamma' \vDash m''$.

On a donc $\Gamma' \vdash \text{DECL } x = v' \text{ IN}\{i'\}$ par DECL et $\Gamma \vDash \text{Cleanup}(m''')$ par M-DECLCLEAN.

IF-FALSE : D'après le lemme d'inversion, $\Gamma \vdash i_f$.

IF-TRUE : D'après le lemme d'inversion, $\Gamma \vdash i_t$.

WHILE : D'après le lemme d'inversion, $\Gamma \vdash e : t$ et $\Gamma \vdash i$. Par SEQ, on a $\Gamma \vdash i; \text{WHILE}(e)\{i\}$. Enfin par IF il vient $\Gamma \vdash \text{IF}(e)\{i; \text{WHILE}(e)\{i\}\}$.

RETURN : Par le lemme d'inversion, $\Gamma \vdash \text{RETURN}(v)$.

EXP-CALL-CTX : On sait que $\Gamma \vdash \text{fun}(a_1, \ldots, a_n)\{i\}(v_1, \ldots, v_n)$ et $\Gamma \vDash m_0$. D'après le lemme d'inversion, il existe $t_1, \ldots, t_n$ tels que $\forall i \in [1; n], \Gamma \vdash v_i : t_i$, $\Gamma \vdash \text{fun}(a_1, \ldots, a_n)\{i\} : (t_1, \ldots, t_n) \to t$, donc qu'en posant $\Gamma' = (\Gamma_G, [a_1 : t_1, \ldots, a_n : t_n, \underline{R} : t])$ où $\Gamma = (\Gamma_G, \Gamma_L)$ on a $\Gamma' \vdash i$.

D'un autre côté, il existe par le lemme 5.3 des types τ_i tels que $\forall i \in [1; n], m_0 \vDash v_i : \tau_i$ avec $\tau_i \rhd t_i$. En appliquant M-PUSH, on a donc $\Gamma' \vDash m_1$.

On peut alors appliquer l'hypothèse d'induction à $\langle i, m_1 \rangle \to \langle i', m_2 \rangle$: la conclusion est que $\Gamma' \vdash i'$ et $\Gamma' \vDash m_2$. Comme $\Gamma' \vdash a_i : t_i$, on a $\forall i \in [1; n], \Gamma' \vdash v'_i : t_i$. Donc on a bien $\Gamma \vdash \text{fun}(a_1, \ldots, a_n)\{i'\}(v'_1, \ldots, v'_n) : t$.

D'autre part, en appliquant M-POP, on obtient que $\Gamma \vDash \text{Cleanup}(m_3)$.

À propos de la règle CTX

L'application de la règle CTX nécessite une explication particulière. En effet, ce cas repose sur un lemme d'inversion des constructions typées sous un contexte, qui est admis ici.

Par exemple, traitons le cas où le contexte C est tel que son « trou » soit une valeur gauche $l\,v$ et $C(\!|l\,v|\!)$ est une instruction (les autres cas sont similaires). La règle appliquée est alors de la forme :

$$\frac{\langle l\,v, m \rangle \to \langle l\,v', m' \rangle}{\langle C(\!|l\,v|\!), m \rangle \to \langle C(\!|l\,v'|\!), m' \rangle} \text{ (CTX)}$$

Si $\Gamma \vdash C(\!|l\,v|\!)$, on admet qu'il existe Γ' et t tels que :

- $\Gamma' \vdash l\,v : t$;

- Quelque soit $l\,v'$, si $\Gamma' \vdash l\,v' : t$, alors $\Gamma \vdash C(\!|l\,v'|\!)$.

Par exemple, pour $C = \bullet[2] = 1$, $\Gamma' = \Gamma$ et $t = \text{INT}[\,]$ conviennent. Pour $C = \text{DECL } x = 0 \text{ IN}\{\bullet = 3.0\}$, on prendra $\Gamma' = \Gamma, \text{local } x : \text{INT}$ et $t = \text{FLOAT}$. Le fait de passer « sous » une déclaration ajoute une variable locale à Γ, et ainsi l'ensemble des variables de Γ' contient celui de Γ.

Pour prouver la préservation dans ce cas, on commence par appliquer l'hypothèse de récurrence à la prémisse de CTX, c'est-à-dire $\langle l\,v, m\rangle \to \langle l\,v', m'\rangle$. Il vient que $\Gamma' \vdash l\,v' : t$ et $\Gamma' \vDash \text{Cleanup}(m')$.

D'après le précédent lemme d'inversion on en déduit que $\Gamma \vdash C(\!|l\,v'|\!)$. De plus Γ' contient plus de variables que Γ donc $\Gamma \vDash \text{Cleanup}(m')$.

□

D.4 Progrès pour les extensions noyau

(Théorème 6.1)

Démonstration. On procède de la même manière que pour le théorème 5.1 (prouvé en annexe D.2). En fait, puisque le schéma de preuve porte sur les règles de typage, il suffit de traiter les cas supplémentaires.

ADDR-USER : Alors $e = \diamondsuit\, l\,v$. On applique l'hypothèse de récurrence à $l\,v$.

- $l\,v = \varphi$. Alors on peut appliquer PHI-USER.
- $\langle l\,v, m\rangle \to \langle l\,v', m'\rangle$. On conclut en utilisant CTX avec $C = \diamondsuit\, \bullet$.
- $\langle l\,v, m\rangle \to \Omega$. On applique EVAL-ERR avec ce même C.

USER-GET : On applique l'hypothèse de récurrence à e_d.

- $e_d = v_d$. On applique l'hypothèse de récurrence à e_s.
  - $e_s = v_s$.
    D'après le lemme 5.2 adapté aux extensions noyau, v_s a pour forme φ_s.
    On distingue la forme de φ_s :
    - $\varphi_s = \widehat{\diamondsuit}\, \varphi$. Alors on applique USER-GET-OK. Le lemme 5.6 adapté aux extensions noyau assure que les prémisses sont correctes.
    - $\nexists\, \varphi, \varphi_s = \widehat{\diamondsuit}\, \varphi$. Alors on applique USER-GET-ERR.
  - $\langle e_s, m\rangle \to \langle e'_s, m'\rangle$. Posons $C = \text{copy\_from\_user}(v_d, \bullet)$. On conclut avec CTX.
  - $\langle e_s, m\rangle \to \Omega$. Idem avec EVAL-ERR.
- $\langle e_d, m\rangle \to \langle e'_d, m'\rangle$. On applique CTX avec $C = \text{copy\_from\_user}(\bullet, e_s)$.
- $\langle e_d, m\rangle \to \Omega$. On utilise EVAL-ERR avec ce même contexte.

USER-PUT : Ce cas est similaire au cas USER-GET, en appliquant les règles USER-PUT-OK et USER-PUT-ERR.

□

D.5 Préservation pour les extensions noyau

(Théorème 6.2)
De même, il suffit de prouver les cas correspondant aux nouvelles règles.

PHI-USER : On applique le lemme de représentation, qu'on a étend avec le cas Repr(t @) = Repr(Repr(t) @).

USER-GET-OK : Tout d'abord, d'après le lemme 6.1, t = INT, donc la préservation du type est établie car $m' \vDash 0 :$ INT La compatibilité mémoire est obtenue en appliquant M-WRITE.

USER-GET-ERR : La seule partie à prouver est la préservation, qui se fait de la même manière que dans le cas précédent.

USER-PUT-OK : Idem que dans le cas USER-PUT-OK.

USER-PUT-ERR : Idem que pour USER-GET-ERR.

LISTE DES FIGURES

Liste des définitions

Liste des théorèmes et lemmes

RÉFÉRENCES WEB

[☞1] The C - - language
http://www.cminusminus.org/

[☞2] OCaml – Home
http://ocaml.org/

[☞3] Penjili project
https://bitbucket.org/iwseclabs/c2newspeak

[☞4] Python Programming Language – Official Website
http://www.python.org/

[☞5] The Rust Programming Language
http://www.rust-lang.org/

[☞6] Sparse - a Semantic Parser for C
https://sparse.wiki.kernel.org/index.php/Main_Page

BIBLIOGRAPHIE

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs 2007, pages 5–21, 2007.

[ABD+07] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 43–48, New York, NY, USA, 2007. ACM.

[AH07] Xavier Allamigeon and Charles Hymans. Analyse statique par interprétation abstraite. In Eric Filiol, editor, *5ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'07)*, Rennes, France, June 2007.

[BA08] S. Bugrara and A. Aiken. Verifying the Safety of User Pointer Dereferences. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 325–338, 2008.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later : using static analysis to find bugs in the real world. *Commun. ACM*, 53(2) :66–75, February 2010.

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, third edition edition, November 2005.

[BDH+09] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, POPL, pages 114–126, Savannah, GA, USA, January 2009.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda — a functional language with dependent types. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system : an overview. In *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3) :103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *Proceedings of the 14th European Symposium on Programming*, ESOP 2005, pages 21–30, 2005.

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does Astrée scale up ? *Formal Methods in System Design*, 35(3) :229–264, 2009.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '78, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[CMP03] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly, 2003.

[DDMP10] Javier De Dios, Manuel Montenegro, and Ricardo Peña. Certified absence of dangling pointers in a language with explicit deallocation. In *Proceedings of the 8th international conference on Integrated formal methods*, IFM'10, pages 305–319, Berlin, Heidelberg, 2010. Springer-Verlag.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV : Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 155–167, New York, NY, USA, 2003. ACM.

[EH94] Ana Erosa and Laurie J. Hendren. Taming control flow : A structured approach to eliminating goto statements. In *In Proceedings of 1994 IEEE International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Programming language design and implementation*, PLDI '99, pages 192–203, 1999.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations : A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[FJKA06] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28 :1035–1087, November 2006.

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.

[GGTZ07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In Rocco Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 237–252. Springer Berlin Heidelberg, 2007.

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. *SIGPLAN Not.*, 37(5) :282–293, May 2002.

[Gon07] Georges Gonthier. The four colour theorem : Engineering of a formal proof. In *8th Asian Symposium on Computer Mathematics*, ASCM 2007, page 333, 2007.

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[Gra92] Philippe Granger. Improving the results of static analyses programs by local decreasing iteration. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 68–79, London, UK, UK, 1992. Springer-Verlag.

[Har88] Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4) :36–38, October 1988.

[HL08] Charles Hymans and Olivier Levillain. Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C. Technical Note 2008-IW-SE-00010-1, EADS IW/SE, 2008.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, October 1969.

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[JMG+02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone : A safe dialect of c. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[Jon10] M. Tim Jones. User space memory access from the Linux kernel. `http://www.ibm.com/developerworks/library/l-kernel-memory-access/`, 2010.

[JW04] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004.

[KcS07] Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci.*, 174(7) :79–104, 2007.

[Ker81] Brian W. Kernighan. Why Pascal is not my favorite programming language. Technical report, AT&T Bell Laboratories, April 1981.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988.

[LA04] Chris Lattner and Vikram Adve. LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lan96] Gérard Le Lann. The ariane 5 flight 501 failure - a case study in system engineering for computing systems, 1996.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml : A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3) :1–38, May 2006.

[LZ06] Peng Li and Steve Zdancewic. Encoding information flow in Haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations (CSFW '06)*, Washington, DC, USA, 2006. IEEE Computer Society.

[Mai90] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, POPL '90, pages 382–401, 1990.

[Mau04] Laurent Mauborgne. ASTRÉE : Verification of absence of run-time error. In René Jacquart, editor, *Building the information Society (18th IFIP World Computer Congress)*, pages 384–392. The International Federation for Information Processing, Kluwer Academic Publishers, Aug 2004.

[McA03] David A. McAllester. Joint RTA-TLCA invited talk : A logical algorithm for ML type inference. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, RTA, pages 436–451, 2003.

[Mer03] J. Merrill. GENERIC and GIMPLE : a new tree representation for entire functions. In *GCC developers summit 2003*, pages 171–180, 2003.

[MG07] Magnus O. Myreen and Michael J.C. Gordon. A Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, LNCS, pages 568–582. Springer-Verlag, 2007.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, December 1978.

[Min01a] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. of the Second Symposium on Programs as Data Objects (PADO II)*, volume 2053 of *Lecture Notes in Computer Science (LNCS)*, pages 155–172. Springer, May 2001. `http://www.di.ens.fr/~mine/publi/article-mine-padoII.pdf`.

[Min01b] A. Miné. The octagon abstract domain. In *Proc. of the Workshop on Analysis, Slicing, and Transformation (AST'01)*, pages 310–319. IEEE CS Press, Oct. 2001. `http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf`.

[NCH+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured : type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3) :477–526, May 2005.

[New00] Tim Newsham. Format string attacks. *Phrack*, 2000.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL : Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[oEE08] Institute of Electrical and Electronics Engineers. IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008.

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[Oiw09] Yutaka Oiwa. Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 259–269, New York, NY, USA, 2009. ACM.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PJNO97] Simon L. Peyton Jones, Thomas Nordin, and Dino Oliva. C- - : A portable assembly language. In Chris Clack, Kevin Hammond, and Antony J. T. Davie, editors, *Implementation of Functional Languages*, volume 1467 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 1997.

[PTS+11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux : Ten years later. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, Newport Beach, CA, USA, March 2011.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :pp. 358–366, 1953.

[RV98] Didier Rémy and Jérôme Vouillon. Objective ML : An effective object-oriented extension to ML. In *Theory And Practice of Object Systems*, pages 27–50, 1998.

[Sim03] Vincent Simonet. Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003.

[SLM11] Suman Saha, Julia Lawall, and Gilles Muller. An approach to improving the structure of error-handling code in the linux kernel. In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, LCTES '11, pages 41–50, New York, NY, USA, 2011. ACM.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21 :2003, 2003.

[Spe05] Brad Spengler. grsecurity 2.1.0 and kernel vulnerabilities. *Linux Weekly News*, 2005.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Selected papers from the 6th international joint conference on Theory and practice of software development*, TAPSOFT '95, pages 131–170, Amsterdam, The Netherlands, The Netherlands, 1996. Elsevier Science Publishers B. V.

[Sta11] Basile Starynkevitch. Melt - a translated domain specific language embedded in the gcc compiler. In Olivier Danvy and Chung chieh Shan, editors, *DSL*, volume 66 of *EPTCS*, pages 118–142, 2011.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01 : Proceedings of the 10th conference on USENIX Security Symposium*, page 16, Berkeley, CA, USA, 2001. USENIX Association.

[SY86] R E Strom and S Yemini. Typestate : A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1) :157–171, January 1986.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[The04] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2 :245–271, 1992.

[TT93] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical report, 1993.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.

[VB04] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded c programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 231–242, New York, NY, USA, 2004. ACM.

[vL11] Twan van Laarhoven. Lenses : viewing and updating data structures in Haskell. `http://www.twanvl.nl/files/lenses-talk-2011-05-17.pdf`, May 2011.

Résumé

Les noyaux de système d'exploitation manipulent des données fournies par les programmes utilisateur via les appels système. Si elles sont manipulées sans prendre une attention particulière, une faille de sécurité connue sous le nom de *Confused Deputy Problem* peut amener à des fuites de données confidentielles ou l'elévation de privilège d'un attaquant.

Le but de cette thèse est d'utiliser des techniques de typage statique afin de détecter les manipulations dangereuses de pointeurs contrôlés par l'espace utilisateur.

On commence par isoler un sous-langage sûr nommé SAFESPEAK du langage C dans lequel sont écrits la plupart des systèmes d'exploitation. Sa sémantique opérationnelle et un système de types sont décrits, et les propriétés classiques de sûreté du typage sont établies.

On ajoute ensuite à SAFESPEAK la notion de valeur provenant de l'espace utilisateur. La sûreté du typage est alors brisée, mais on peut la réétablir en donnant un type particulier aux pointeurs contrôlés par l'espace utilisateur, ce qui force leur déférencement à se faire de manière contrôlée. Cette technique permet de détecter un bug qui a frappé un pilote de carte graphique AMD dans le noyau Linux.

Ces travaux ouvrent des perspectives sur deux points. Il est possible d'appliquer cette analyse à d'autres interfaces du noyau Linux qui communiquent avec l'espace utilisateur. De plus, cette technique revient à séparer l'interface des types de leur interface. Ce concept de *types abstraits* étant absent de C, de nombreuses erreurs de programmation peuvent également être vérifiées à l'aide de cette méthode.