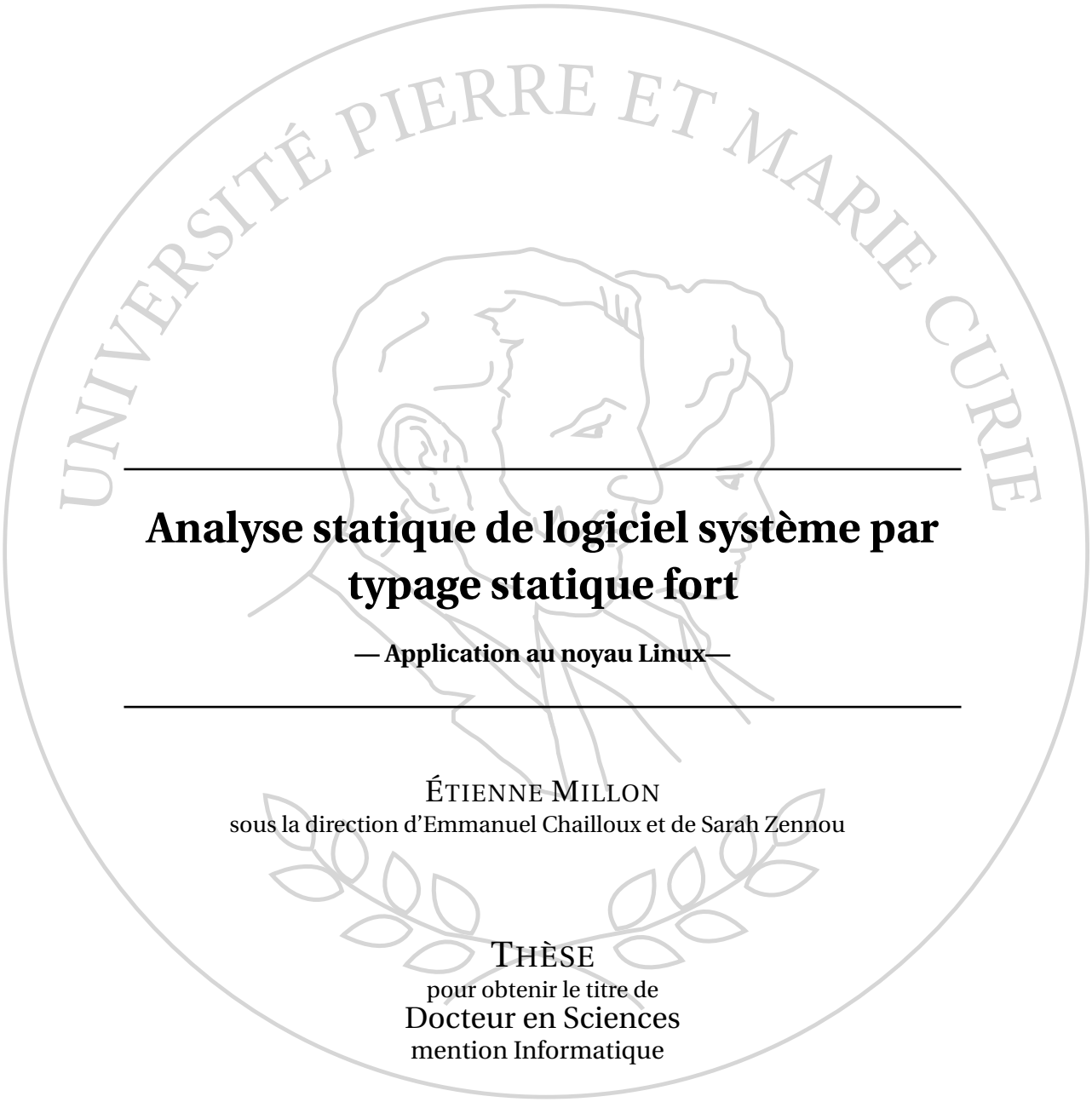
UNIVERSITÉ PIERRE ET MARIE CURIE
ÉCOLE DOCTORALE INFORMATIQUE, TÉLÉCOMMUNICATIONS ET ÉLECTRONIQUE



Analyse statique de logiciel système par typage statique fort

— 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 23 avril 2013

Français

English

Manipulating user-provided pointers in the kernel of an operating system can lead to security flaws if done in an incautious manner. We present an efficient system to detect and prevent this class of erroneous memory manipulation.

At the core of our approach is SAFESPEAK, an imperative language that we equip with a qualified type system, where two kinds of pointers are distinguished : *safe* pointers, whose value is statically proved to be controlled by the kernel, and *unsafe* ones, whose value comes from userspace through run-time system calls. Dereferencing unsafe pointers is forbidden in a static manner by the means of a strong type system.

A concrete case study is described based on a bug that affected a video driver in the Linux kernel. We also explain a technique to automatically translate GNU C code to our core language, which will enable us to analyze larger fractions of the kernel in order to find similar vulnerabilities.

Remerciements

Dédicace

TABLE DES MATIÈRES

CHAPITRE 1

INTRODUCTION

Communication, audiovisuel, transports, médecine : tout 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 2013, 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 basé 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 et l'isolation 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. Loin de là : on estime qu'avant relecture, 1000 lignes de code contiennent entre 5 et 100 erreurs de programmation en moyenne. 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.

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 des tests les plus exhaustifs 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 des erreurs, quitte à limiter la possibilité d'en écrire certains. C'est l'analyse statique de programmes.

Une des techniques d'analyse statique les plus répandues 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.

Pour garantir l'isolation d'un noyau de système d'exploitation, un des points cruciaux est de restreindre la manière dont sont traitées les informations provenant des programmes utilisateur. Le but de cet 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

On décrit ici la fonction générale d'un système d'exploitation. Ce sujet est détaillé en profondeur, y compris avec des cas d'étude, dans [Tan07].

Un ordinateur est constitué de nombreux composants matériels : microprocesseur, mémoire, et divers périphériques. Pourtant, 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 niveaux d'abstraction.

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 60, 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 deux programmes (ou plus) 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, ou 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 mémorette 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 mémorette USB, 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 le même but (copier de la mémoire depuis un périphérique amovible), ne sont pas effectuées 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 particulier est responsable de la gestion de la mémoire, des périphériques et des processus. Les détails d'implantation ne sont pas présentés à l'utilisateur ; à la place il manipule des abstractions, comme la notion de fichier.

1.2 Séparation entre noyau et espace utilisateur

Le noyau n'est pas le seul programme exécuté par le processeur : d'autres programmes peuvent être aussi être lancés. Ce sont ceux-ci qui sont d'ailleurs visibles pour l'utilisateur.

Mode du processeur	Privilège (code)	Privilège (données)	Accès possible
Noyau	Noyau	Noyau	☑
Noyau	Noyau	Utilisateur	☑
Noyau	Utilisateur	Noyau	☑
Noyau	Utilisateur	Utilisateur	☑
Utilisateur	Noyau	Noyau	□
Utilisateur	Noyau	Utilisateur	□
Utilisateur	Utilisateur	Noyau	□
Utilisateur	Utilisateur	Utilisateur	☑

FIGURE 1.1: Règles d'exécution de code et d'accès à la mémoire.

Puisque le noyau est garant du bon fonctionnement du système, 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].

Ainsi, pour une instruction manipulant des données en mémoire, les accès sont possibles sont décrits dans la figure 1.1. En cas d'erreur (signalée par un □), une erreur à l'exécution se produit.

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.

Les programmes utilisateur ne pouvant pas accéder à ces instructions de bas niveau, ils sont très limités dans ce qu'ils peuvent faire. Puisque l'interaction avec le matériel (comme l'écriture sur un disque dur) se fait uniquement via des instructions privilégiées, ils sont limités à l'utilisation du processeur et de la mémoire, permettant uniquement de réaliser des calculs.

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 par le mécanisme des *appels système*.

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 l'adresse sur le disque où commencer la lecture (*offset*), 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. On est en mode noyau, en train d'exécuter une instruction du noyau manipulant des données utilisateur.

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.

D'après la figure 1.1 aucune erreur ne se produit.

Mais même si ce cas ne produit pas d'erreur à l'exécution, il est tout de même erroné. 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 des 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].

La bonne manière d'implanter un appel système est donc d'interdire le déréférencement direct des pointeurs dont la valeur peut être controllée par l'utilisateur. Dans le cas du passage par adresse d'un argument, il aurait fallu vérifier à l'exécution que celle-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, qui permet de détecter ou d'empêcher de manipuler des données incompatibles entre elles.

Nous avons vu dans le chapitre 2 qu'au niveau du langage machine, les seules données qu'un ordinateur manipule sont des nombres. Selon les opérations effectuées, ils seront interprétés comme des entiers, des adresses mémoires, 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 : "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".

Il y a deux grandes familles de systèmes de types, selon quand se fait la vérification de types. On peut en effet l'effectuer au moment de l'exécution, ou au contraire prévenir les erreurs à l'exécution en la faisant au moment de la compilation (ou avant l'interprétation).

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 ces deux opérandes ont une étiquette "entier", et construit alors une valeur obtenue en faisant l'addition des deux valeurs, avec une étiquette "entier". Par exemple, le langage Python [3] utilise cette stratégie.

Typage statique : dans ce cas on fait les vérifications à la compilation. En quelque sorte, l'approche dynamique est pessimiste, puisqu'elle demande de traiter très souvent le cas où les types ne sont pas corrects. Intuitivement, dans le cas où toutes les fonctions se comportent bien, faire la vérification est inutile. Pour vérifier ceci, on donne à chaque fonction

```
Object o = new Integer(3);
Float f = (Float) o;

Exception in thread "main" java.lang.ClassCastException:
    java.lang.Integer cannot be cast to java.lang.Float
        at Cast.main(Cast.java:5)
```

FIGURE 1.2: Transtypage en Java

un contrat comme “si deux entiers sont passés, et que la fonction renvoie une valeur[2], 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.

Ainsi la fonction `"+"` est typée `(int, int)` → `int`. Les règles permettant de vérifier le typage sont par exemple les suivantes :

- une constante entière est toujours de type `int`.
- si f a pour type $(t_1, \ldots, t_n) \to t$, et que chaque e_i a pour type t_i ($i \in [1; n]$), alors $f(e_1, \ldots, e_n)$ a pour type t.
- si en considérant que chaque e_i a pour type t_i ($i \in [1; n]$), on arrive à typer le corps de f et que sa valeur de retour a alors pour type t, alors f a pour type $(t_1, \ldots, t_n) \to t$.

Typage fort ou faible Contrairement à la distinction claire entre typage statique ou dynamique, la séparation entre typage fort et faible est moins nette. À l'extrémité du spectre, les systèmes de types forts permettent d'éliminer totalement la nécessité de réaliser des tests de typage. 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 transtypage1.2. À l'exécution, une erreur de types est levée.

Polymorphisme Parfois, il est trop restrictif de donner un unique contrat à une fonction. Quel doit être le type d'une fonction ajoutant un élément à une liste, ou celui d'une fonction triant un tableau en place ?

En première approximation, on peut imaginer fournir une version du code par type de données à manipuler. C'est la solution retenue 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*.

La dernière possibilité est le polymorphisme universel, qui consiste à utiliser le même code quelque soit le type des arguments. Dans ce cas, on utilise une seule fonction pour retourner 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 schéma de types, instanciable en un type concret. En quelque sorte, on déplace la vérification du site d'appel au site de définition. Cette technique a été décrite en premier dans [Mil78].

2. La plupart des systèmes de types ne concernent que les termes qui convergent. Il est également possible que la fonction contienne une boucle infinie, ou provoque une erreur à cause d'une division par zéro. Dans ces cas, la fonction ne renvoie pas de valeur.

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

1.4 Langages d'implantation

Puisque notre but est d'analyser du code provenant de systèmes d'exploitation, il est nécessaire de s'intéresser aux langages de programmation dans lesquels ils sont écrits.

Assembleur Le noyau d'un système d'exploitation nécessite d'accéder au matériel, donc il est naturellement bas niveau. Pour accéder aux fonctionnalités spécifiques de chaque processeur, il est donc nécessaire d'en implanter une partie dans le langage d'assemblage natif de chaque architecture. Historiquement, les premiers système d'exploitations étaient entièrement écrits en assembleur. Cela est de plus en plus rare, sauf dans les cas où les ressources sont trop limitées pour exécuter du code compilé, comme dans les systèmes embarqués ou temps-réel.

C 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. Ceci dit, le système de types 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.

1.5 Langages d'analyse

Les langages décrits précédemments sont faits pour être facilement écrits par des programmeurs humains. En général ils sont ambigüs, peu précis ou ont des comportements implicites. Pour analyser du code source, il est plus pratique d'avoir une représentation intermédiaire plus simple afin d'avoir moins de traitements dupliqués. Dans de nombreux projets, des sous-ensembles de C ont été définis pour aller dans ce sens.

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'implémentation pouvant varier entre deux versions de GCC. Cette représentation étant réputée difficile à manipuler, le projet MELT[Sta11] permet de générer un greffon de compilateur à partir d'un dialecte de Lisp.

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

Objective Caml [1] utilise pour sa génération de code une représentation interne nommée Cmm, disponible dans les sources du compilateur sous le chemin `asmcomp/cmm.mli` (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.

C- - [PJNO97] [7], 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 utilise une représentation intermédiaire très similaire à C- -.

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. Puisque qu'ils sont développés en parallèle des compilateurs, le support des extensions est en général moins important dans ces langages.

CIL [NMRW02] [6] est une représentation en OCaml d'un programme C, développée depuis 2002. Grâce à un mécanisme de greffons, elle permet de prototyper rapidement des analyses statiques de programmes.

Compcert est un projet qui vise à produire un compilateur certifié pour C. C'est à dire que le fait que les transformations conservent la sémantique est prouvé. Il utilise de nombreux langages intermédiaires, dont CIL. Pour le front-end, le langage se nomme Clight[BDL06]. Les passes de middle-end, quant à elles, utilisent Cminor[AB07].

Newspeak [HL08] est un langage intermédiaire développé par EADS Innovation Works, et qui est spécialisé dans l'analyse de valeurs par interprétation abstraite. Il sera décrit plus en détails dans la section 7.1.

1.6 Plan de la thèse

Cette thèse comporte trois parties.

La partie I présente le contexte de ces travaux. Le fonctionnement général d'un système d'exploitation y est détaillé, et les problèmes de manipulation de pointeurs contrôlés par l'utilisateur y sont introduits. On fait ensuite un tour d'horizon des techniques existantes permettant de traiter ce problème. Cette partie se conclut par la proposition au cœur de cette thèse.

La partie II décrit notre solution : SAFESPEAK, un langage impératif. On y décrit sa syntaxe, sa sémantique ainsi qu'un système de types statiques. On l'étend ensuite pour capturer les problèmes d'adressage mémoire présents dans les systèmes d'exploitation en ajoutant des pointeurs contrôlés par l'utilisateur. Le système de types est également étendu. Pour chacune de ces variantes, on établit la propriété de sûreté de typage reliant la sémantique dynamique aux types statiques.

La partie III documente la démarche expérimentale associée à ces travaux. L'implantation du système de types est décrite, afin que la manière de transformer automatiquement du code C en SAFESPEAK. Un cas d'étude est déroulé, consistant d'un bug ayant touché le noyau

Linux. Il est démontré que le système de type capture précisément ce genre d'erreur de programmation. Enfin, les possibilités d'extension tant théoriques qu'expérimentales sont présentées.

Première partie

Méthodes formelles pour la sécurité

Le chapitre 2 décrit le contexte de ces travaux, notamment le fonctionnement général d'un système d'exploitation et la séparation du code en plusieurs niveaux de privilèges. Le mécanisme d'appels système est décrit, et on montre qu'une implantation naïve de la communication entre espaces utilisateur et noyau casse toute isolation. On présente la situation prise par le noyau Linux : séparer deux classes de pointeurs sensées être indépendante.

Le chapitre 3 consiste en un tour d'horizon des techniques existantes en analyses de programmes. Ces analyses se centrent sur, mais ne se limitent pas au problème de manipulation mémoire évoqué dans le chapitre 2.

CHAPITRE 2

SYSTÈMES D'EXPLOITATION

Le système d'exploitation est le programme qui permet à un système informatique d'exécuter d'autre programmes. Son rôle est donc capital et ses responsabilités multiples. Dans ce chapitre, nous allons voir quel est son rôle, et comment il peut être implanté. Pour ce faire, nous étudierons l'exemple d'une architecture Intel 32 bits, et d'un noyau Linux 2.6.

2.1 Architecture Intel

L'implantation d'un système d'exploitation est très proche du matériel sur lequel il s'exécute. Pour étudier une implantation en particulier, voyons ce que permet le matériel lui-même.

Dans cette section nous décrivons le fonctionnement d'un processeur utilisant une architecture Intel 32 bits. Les exemples de code seront écrits en syntaxe AT&T, celle que comprend l'assembleur GNU.

La référence pour la description de l'assembleur Intel est la documentation du constructeur [Int10] ; une bonne explication de l'agencement dans la pile peut aussi être trouvée dans [One96].

2.1.1 Assembleur

Pour faire des calculs, le processeur est composé de registres, qui sont des petites zones de mémoire interne, et peut accéder à la mémoire principale.

La mémoire principale contient divers types des données :

- le code des programmes à exécuter
- les données à disposition des programmes
- la pile d'appels

La pile d'appels est une zone de mémoire qui est notamment utilisée pour tenir une trace des calculs en cours. Par exemple, c'est ici que seront stockées les données propres à chaque fonction appelée : paramètres, adresse de retour et variables locales. La pile est manipulée par un pointeur de pile (*stack pointer*), qui est l'adresse du "haut de la pile". On peut la manipuler en empilant des données (les placer au niveau du pointeur de pile et déplacer celui si) ou dépilant des données (déplacer le pointeur de pile dans l'autre sens et retourner la valeur présente à cet endroit).

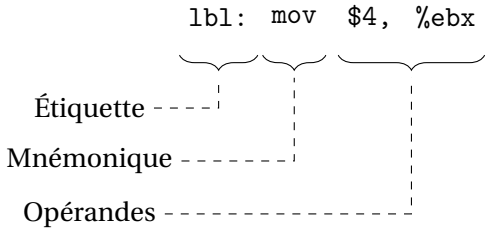
L'état du processeur est défini par la valeur de ses registres, qui sont des petites zones de mémoire interne (quelques bits chacun). Par exemple, la valeur du pointeur de pile est sto-

ckée dans ESP. Le registre EBP, couplé à ESP sert à adresser les variables locales et paramètres d'une fonction, comme ce sera expliqué dans la section 2.1.2.

L'adresse de l'instruction courante est accessible dans le registre EIP.

En plus de ces registres spéciaux, le processeur possède de nombreux registres génériques, qui peuvent être utilisés pour réaliser des calculs intermédiaires. Ils sont nommés EAX, EBX, ECX, EDX, ESI et EDI. Ils peuvent être utilisés pour n'importe quel type d'opération, mais certains sont spécialisés : par exemple il est plus efficace d'utiliser EAX en accumulateur, ou ECX en compteur.

Les calculs sont décrits sous forme d'une suite d'instructions. Chaque instruction est composée d'un mnémonique et d'une liste d'opérandes. Les mnémoniques (`mov`, `call`, `sub`, etc) définissent un type d'opération à appliquer sur les opérandes. L'instruction peut aussi être précédée d'une étiquette, qui correspondra à son adresse.



Ces opérandes peuvent être de plusieurs types :

- un nombre, noté $4
- le nom d'un registre, noté `%eax`
- une opérande mémoire, c'est à dire le contenu de la mémoire à une adresse effective. Cette adresse effective peut être exprimée de plusieurs manières :
  - directement : `addr`
  - indirectement : `(%ecx)`. L'adresse effective est le contenu du registre.
  - "base + déplacement" : `4(%ecx)`. L'adresse effective est le contenu du registre plus le déplacement (4 ici).

En pratique il y a des modes d'adressage plus complexes, et toutes les combinaisons ne sont pas possibles, mais ceux-ci suffiront à décrire les exemples suivants :

- `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`.
- `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`.

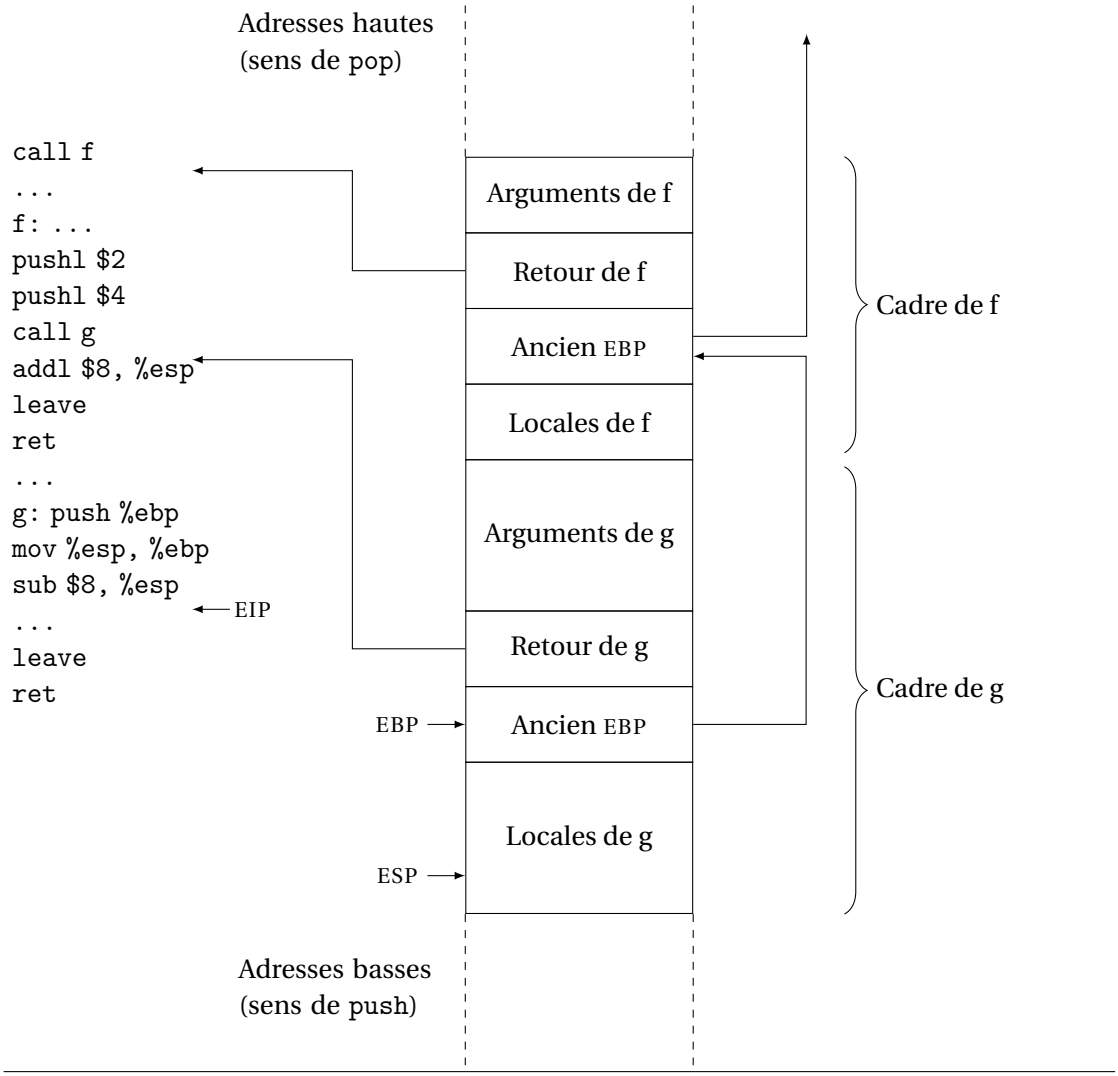


FIGURE 2.1: Cadres de pile.

2.1.2 Fonctions et conventions d'appel

Dans le langage d'assemblage, il n'y a pas de notion de fonction ; mais `call` et `ret` permettent de sauvegarder et de restaurer une adresse de retour, ce qui permet de faire un saut et revenir à l'adresse initiale. Ce système permet déjà de créer des procédures, c'est-à-dire des fonctions sans arguments ni valeur de retour.

Pour gérer ceux-ci, il faut que les deux morceaux (appelant et appelé) se mettent d'accord sur une convention d'appel commune. La convention utilisée sous GNU/Linux est appelée *cdecl* et possède les caractéristiques suivantes :

- la valeur de retour d'une fonction est stockée dans EAX
- EAX, ECX et EDX peuvent être écrasés sans avoir à les sauvegarder
- les arguments sont placés sur la pile (et enlevés) par l'appelant. Les paramètres sont empilés de droite à gauche.

Pour accéder à ses paramètres, une fonction peut donc utiliser un adressage relatif à ESP. Cela peut fonctionner, mais cela complique les choses si elle contient aussi des variables locales. En effet, les variables locales sont placées sur la pile, au dessus des (c'est à dire, empilées après) paramètres, augmentant le décalage.

Pour simplifier, la pile est organisée en cadres logiques : chaque cadre correspond à un niveau dans la pile d'appels de fonctions. Si f appelle g, qui appelle h, il y aura dans l'ordre sur la pile le cadre de f, celui de g puis celui de h.

Ces cadres sont chainés à l'aide du registre EBP : à tout moment, EBP contient l'adresse du cadre de l'appelant.

Prenons exemple sur la figure 2.1 : pour appeler `g(4,2)`, `f` empile les arguments de droite à gauche. L'instruction `call g` empile l'adresse de l'instruction suivante sur la pile puis saute au début de g.

Au début de la fonction, les trois instructions permettent de sauvegarder l'ancienne valeur de EBP, faire pointer EBP à une endroit fixe dans le cadre de pile, puis allouer 8 octets de mémoire pour les variables locales.

Dans le corps de la fonction g, on peut donc se référer aux variables locales par `-4(%ebp)`, `-8(%ebp)`, etc, et aux arguments par `8(%ebp)`, `12(%ebp)`, etc.

À la fin de la fonction, l'instruction `leave` est équivalente à `mov %ebp, %esp` puis `pop %ebp` et permet de défaire le cadre de pile, laissant l'adresse de retour en haut de pile. Le `ret` final la dépile et y saute.

2.1.3 Tâches, niveaux de privilèges

Sans mécanisme particulier, le processeur exécuterait uniquement une suite d'instruction 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 parallèle. 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.

Il y a 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.

2.1.4 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.

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 kibioctets (figure 2.2).

En ce qui concerne la mémoire, les différentes tâches ont une vision différente de la mémoire physique : c'est-à-dire que deux tâches lisant à une même adresse peuvent avoir un résultat différent. C'est le concept de mémoire virtuelle (fig 2.3).

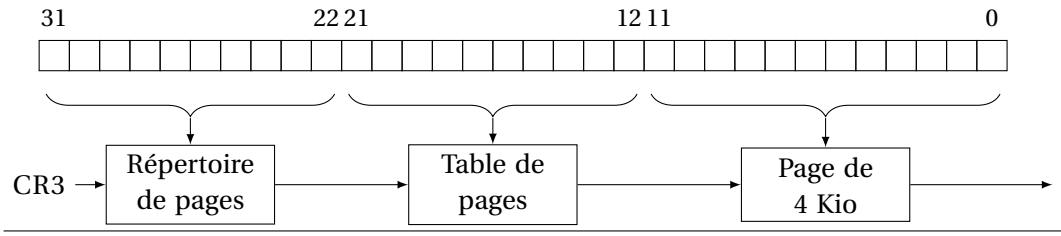


FIGURE 2.2: Implantation de la mémoire virtuelle

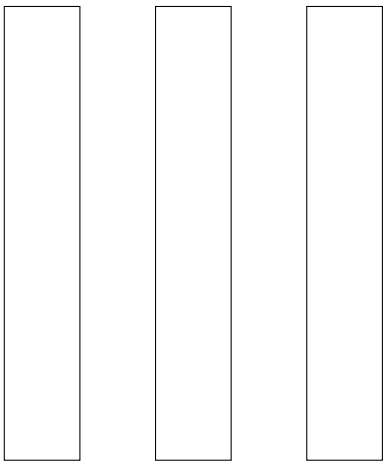


FIGURE 2.3: Mécanisme de mémoire virtuelle.

2.2 Sécurité des appels système

On a vu que les appels systèmes permettent aux programmes utilisateur d'accéder au 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'implémenter 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 systèmes : d'une part, ceux qui renvoient un simple nombre, 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 8.4.1.

Mais la plupart des appels systèmes 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 2.4).

Considérons une implémentation naïve de cet appel système qui écrirait directement à l'adresse pointée. La figure 2.5(a) présente ce qui se passe lorsque le pointeur fourni est dans l'espace d'adressage du processus : c'est le cas d'utilisation normal et l'écriture est donc possible.

```
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 2.4: Appel de gettimeofday

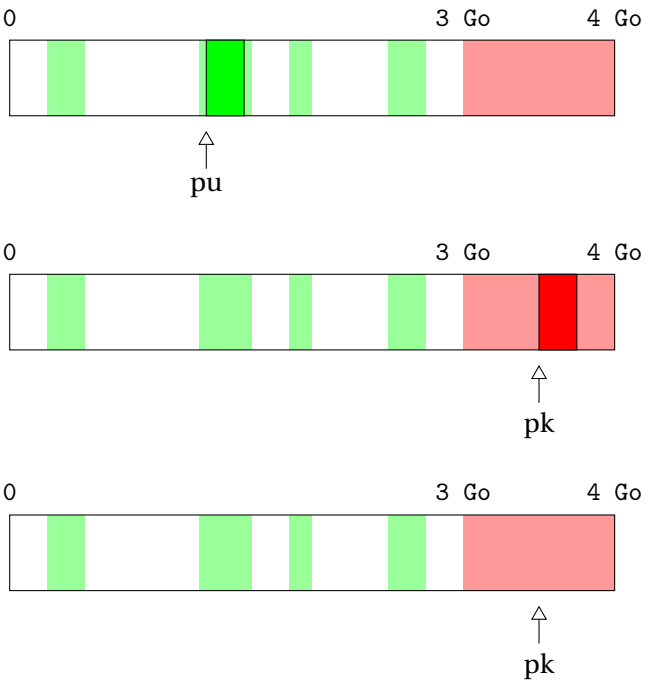


FIGURE 2.5: Zones mémoire

Si l'utilisateur passe un pointeur dont la valeur est supérieure à `0xc0000000` (figure 2.5(b)), 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 malicieux 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 système utilise les privilèges du noyau au lieu de celui qui contrôle la valeur des paramètres sensibles. Celà s'appelle le *Confused Deputy Problem*[Har88].

La bonne solution est de tester dynamiquement la valeur du pointeur : si la valeur du pointeur est supérieure à `0xc0000000`, il faut indiquer une erreur avant d'écrire (figure 2.5(c). Sinon, cela ne veut pas dire que le déréférencement se fera sans 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

```
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 2.6: Implantation de l'appel système `gettimeofday`

l'appel système `gettimeofday` fait appel à celle-ci (figure 2.6).

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

ÉTAT DE L'ART

Dans ce chapitre, nous présentons un tour d'horizon des techniques existantes permettant d'analyser des programmes. Un accent est mis sur la propriété de sûreté décrite dans le chapitre 2, mais on ne se limite pas à celle-ci.

L'analyse statique de programmes est un sujet de recherche actif depuis l'apparition de la science informatique.

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.

Dans la suite, nous considérerons essentiellement des techniques statiques, précisant le contraire lorsque c'est nécessaire.

Cohérence et complétude : le but d'une analyse statique est de catégoriser les programmes selon leurs caractéristiques à l'exécution. Or,

Théorème 3.1 (de Rice)**.** *Toute propriété non triviale sur le comportement dynamique des programmes est indécidable.[Ric53]*

Autrement dit, il n'est pas possible d'écrire un analyseur statique parfait, c'est à dire ne se trompant jamais. Toute technique statique va donc de se retrouver dans au moins un des cas suivants :

- un programme valide est rejeté : on parle de *faux positif.*
- un programme invalide n'est pas détecté : on parle de *faux négatif.*

En général on préfère s'assurer que les programmes acceptés possèdent la propriété recherchée, quitte à en rejeter certains.

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.

3.3 Interprétation abstraite

L'interprétation abstraite est une technique d'analyse générique qui permet de simuler statiquement tous les comportements d'un programme Cousot [CC77, CC92]. Un exemple d'application est de calculer les bornes de variations des variables pour s'assurer qu'aucun débordement de tableau n'est possible [AH07]. Cette technique est très puissante mais possède plusieurs inconvénients. D'une part, pour réaliser une analyse interprocédurale il faut partir d'un point en particulier du programme (comme la fonction `main`). Cette hypothèse n'est pas facilement satisfaite dans un noyau de système d'exploitation, qui possède de nombreux points d'entrée.

Les domaines les plus simples ne capturent aucune relation entre variables. Ce sont des domaines non relationnels. Par exemple le domaine des signes capture uniquement le signe des variables (figure 3.2), et le domaine des intervalles retient les bornes de variations extrémales des variables (figure 3.3).

Lorsque plusieurs variables sont analysées en même temps, utiliser de tels domaines consiste à considérer un produit cartésien d'ensembles abstraits (figure 3.4(a))

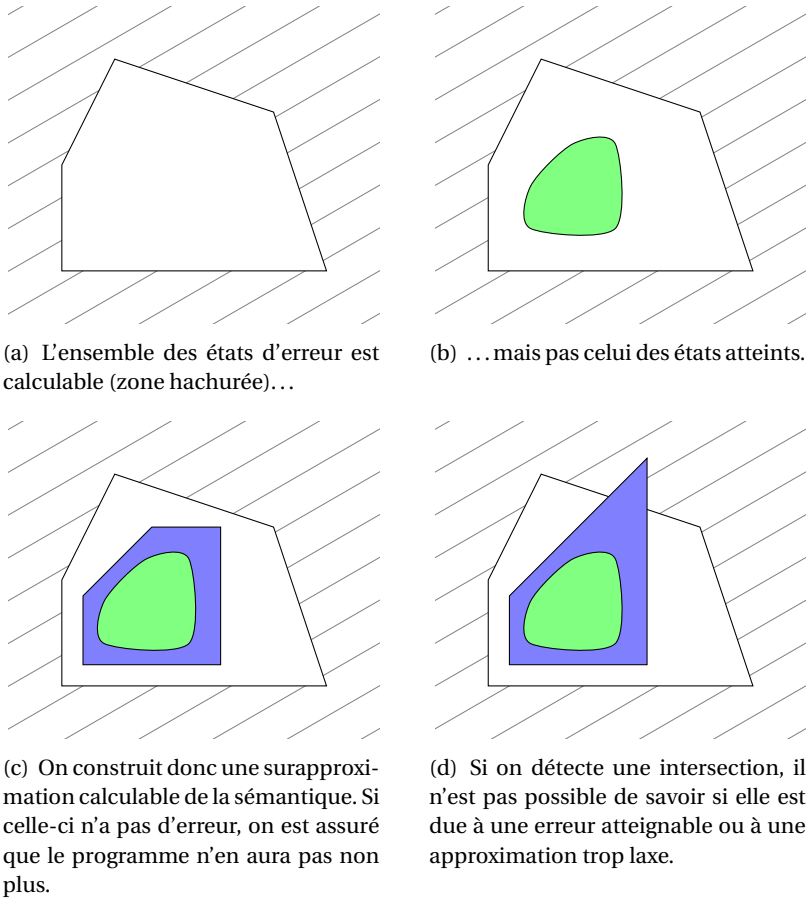
Cela revient à oublier les relations entre les variables. Des domaines abstraits plus précis permettent de retenir celles-ci. Pour ce faire, il faut modéliser l'ensemble des valeurs des variables comme un tout. Parmi les domaines relationnels courants on peut citer :

- Le domaine des polyèdres, historiquement l'un des premiers domaines relationnels. Il permet de retenir tous les invariants affines entre fonctions (figure 3.4(b)).
- Le domaine des zones permet de représenter des relations affines de forme $v_i - v_j \leq c$ (figure 3.4(c)).
- Le domaine des octogones est un compromis entre les polyèdres et les zones. Il permet de représenter les relations $\pm v_i \pm v_j \leq c$ (figure 3.4(d)).

En plus des domaines numériques, il est nécessaire d'employer des domaines spécialisés dans la modélisation de la mémoire. Cela est nécessaire pour pouvoir "suivre" 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 [SRH95]. Les raffinement 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



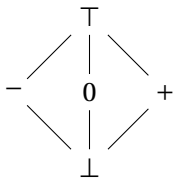
(a) L'ensemble des états d'erreur est calculable (zone hachurée)...

(b) ...mais pas celui des états atteints.

(c) On construit donc une surapproximation calculable de la sémantique. Si celle-ci n'a pas d'erreur, on est assuré que le programme n'en aura pas non plus.

(d) Si on détecte une intersection, il n'est pas possible de savoir si elle est due à une erreur atteignable ou à une approximation trop laxe.

FIGURE 3.1: Surapproximation en interprétation abstraite. Il n'est pas possible de déterminer si l'ensemble des états atteignables est inclus dans l'ensemble des états sûrs (figure 3.1(b)). En revanche, en construisant une surapproximation on peut parfois conclure (figures 3.1(c) et 3.1(d)).



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

FIGURE 3.2: Domaine des signes

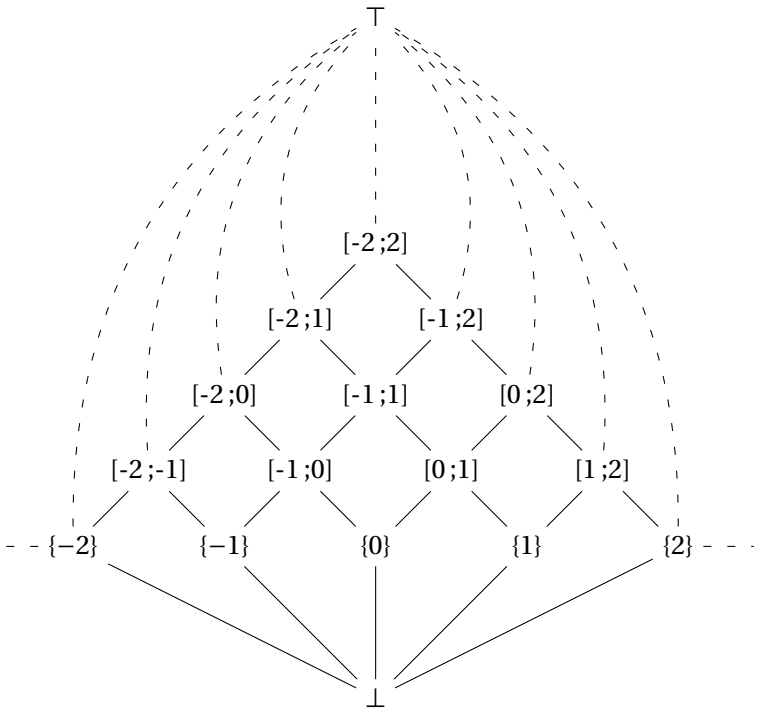
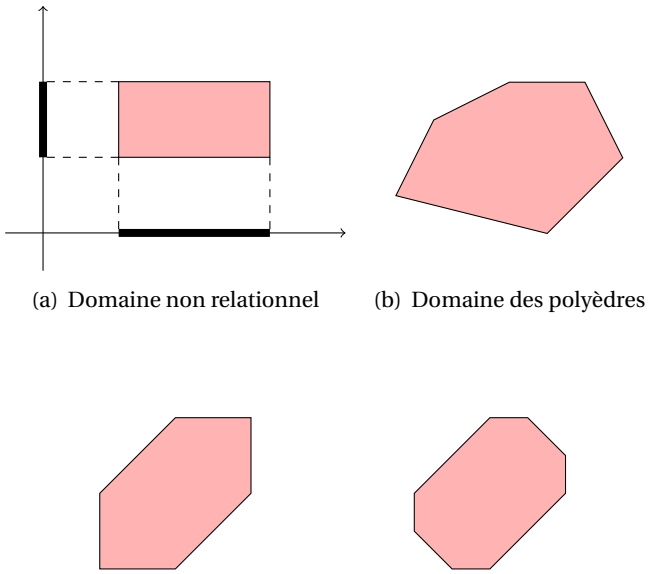


FIGURE 3.3: Domaine des intervalles



(a) Domaine non relationnel (b) Domaine des polyèdres

(c) Domaine des zones (d) Domaine des octaèdres

FIGURE 3.4: Quelques domaines abstraits

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]. Dans l'itération de point fixe, il est possible d'obtenir les résultats de manière plus efficace en choisissant un ordre particulier dans les calculs des sous-itérations[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.

Pour des retours d'expérience, on peut se référer aux descriptions d' Astrée [Mau04, CCF+05, CCF+09], CGS [VB04], ou Coverity [BBC+10],

Une interprétation abstraite est par construction sûre et incomplète, donc ce qui sépare une bon analyseur d'un mauvais est sa précision. Dans le cas du langage C, de nombreuses constructions rendent imprécises les analyses :

L'interprétation abstraite a été utilisée pour analyser le flot de données entre objets [LM12].

Elle a l'inconvénient de remonter de nombreuses fausses alarmes, puisqu'elle nécessite d'avoir une vue précise du programme. Elle est aussi plus adaptée aux programmes qui n'ont qu'un point d'entrée.

3.4 Typage

3.5 Analyse de code système

Les logiciels système demandant des garanties de sécurité et de fiabilité, de nombreuses analyses *ad-hoc* ciblent les noyaux de systèmes d'exploitation.

Ajouter un système de types forts à C est l'idée centrale de CCured [NCH+05]. Dans les cas où il n'est pas possible de conclure, des vérifications à l'exécution sont ajoutées. Cependant, cela nécessite une instrumentation dynamique qui est faite pour rester active, ce qui se paye en performances.

Saturn [ABD+07] est un système pour analyser du code système écrit en C. Il traite le problème des pointeurs utilisateur en utilisant une analyse de forme “pointe-sur” [BA08]. Comme l'interprétation abstraite, son but est d'être très précis, au détriment d'un temps de calcul important dans certains cas.

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 [1], 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 :

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

1. Comme dans la plupart des cas, la vérification de la terminaison d'un algorithme est réalisée de manière séparée.

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

$$\frac{\vdash_{\mathscr{L}} P \Rightarrow P' \qquad \{P\}\ i\ \{Q\} \qquad \vdash_{\mathscr{L}} Q' \Rightarrow Q}{\{P'\}\ i\ \{Q'\}}\ (\text{HOARE-CONSEQUENCE})$$

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.

La règle de conséquence permet de découpler 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 [DRS00], 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[LBR99] ou pour le langage C# dans Spec#[BLS05].

3.7 Proposition

Le but de cette thèse est de montrer que les techniques de typage statique permettent de vérifier que les pointeurs utilisateurs sont manipulés d'une manière qui préserve l'intégrité du noyau d'un système d'exploitation.

Deuxième partie

Un langage pour l'analyse de code système : SAFESPEAK

Dans cette partie, nous allons présenter un langage impératif modélisant le langage C. Le chapitre 4 décrit sa syntaxe, ainsi que sa sémantique. À ce point, de nombreux programmes sont acceptés mais qui provoquent 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 catégorisé 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 de typage (dynamique), lorsqu'on tente d'appliquer à une opération des valeurs incompatibles (additionner un entier et une fonction par exemple.
- 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.
- les erreurs mémoire, qui résultent d'un débordement de tableau, du déréférencement d'un pointeur invalide ou d'arithmétique de pointeur invalide.

En résumé, l'introduction des types simples enlève la possibilité de rencontrer des erreurs de typage dynamique, et l'ajout des qualificateurs interdit les erreurs de sécurité.

Langage	Types	Erreurs possibles		
		Typage	Sécurité	Mémoire
SAFESPEAK	sans	☑	N/A	☑
SAFESPEAK	simples	☐	N/A	☑
SAFESPEAK noyau	simples	☐	☑	☑
SAFESPEAK noyau	qualifiés	☐	☐	☑

CHAPITRE 4

SYNTAXE ET SÉMANTIQUE

On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out ?"... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

— Charles Babbage, *Passages from the Life of a Philosopher*

Dans ce chapitre nous présentons SAFESPEAK, un langage impératif inspiré de C. Sa syntaxe est tout d'abord décrite ; puis une sémantique opérationnelle est explicitée.

Ce langage servira de support au systèmes de types décrit dans le chapitre 5 et enrichi dans le chapitre 6.

La traduction depuis C sera explicitée dans le chapitre 7.

4.1 Notations

Ensembles inductifs

Dans ce chapitre (et les chapitres suivants), on définit de nombreux ensembles inductifs. Plutôt que d'écrire la construction explicite par point fixe, on emploie une notation en grammaire.

Étudions l'exemple des listes chaînées composées d'élements de $\mathbb{N}$.

Notons L cet ensemble ; si [] est la liste vide et $n :: l$ la liste formée d'une "tête" $n \in \mathbb{N}$ et d'une "queue" $l \in L$. Toute liste est donc d'une des formes suivantes :

- []
- $n_1 :: []$
- $n_1 :: n_2 :: []$
- etc.

On peut donc L de la manière inductive suivante :

$$\mathrm{L} = \mathrm{fix}(L')$$
$$L'(E) = \{[]\} \cup \{n :: l \,/\, n \in \mathbb{N}, l \in E\}$$

où

$$\begin{aligned} \text{fix}(f) &= \bigcup_{n=0}^{\infty} f^n(\emptyset) \\ f^0(x) &= x \\ \forall n > 0, f^n(x) &= f^{n-1}(f(x)) \end{aligned}$$

(L'itération n de l'union correspond aux listes comprenant au plus n éléments)

Plutôt que d'écrire cette définition précise mais chargée, on écrira à la place une définition en compréhension :

Listes	$l ::= []$	Liste vide
	$\mid \quad n :: l$	Construction de liste

Chaque ensemble est identifié de manière unique par les noms de variables métasyntaxiques : n pour les entiers et l pour les listes ici. Si plusieurs métavariables du même ensemble doivent apparaître, elles sont indicées. Par exemple, on peut définir des arbres binaires d'entiers de la manière suivante :

Arbres binaires	$a ::= \text{F}$	Feuille
	$\mid \quad \text{N}(a_1, n, a_2)$	Nœud

Cette notation a aussi l'avantage de s'étendre facilement aux définitions mutuellement récursives.

Inférence

La sémantique opérationnelle consiste en la définition d'une relation de transition $\cdot \to \cdot$ entre états de l'interpréteur [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'inférences, de la forme :

$$\frac{P_1 \quad \dots \quad P_n}{C} \text{ (NOM)}$$

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

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

$$\frac{}{A} \text{ (AX)}$$

Compte-tenu de la structure des règles, la preuve d'un jugement pourra donc être vue sous la forme d'un arbre :

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

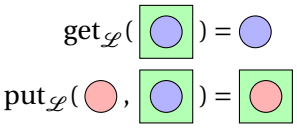


FIGURE 4.1: Fonctionnement d'une lentille

$$\dfrac{\dfrac{\dfrac{}{A_1}\,(\text{R3}) \quad \dfrac{}{A_2}\,(\text{R4})}{B_1}\,(\text{R2}) \quad \dfrac{\dfrac{}{A_3}\,(\text{R6})}{B_2}\,(\text{R5})}{C}\,(\text{R1})$$

Lentilles

La notion d'accesseur utilisée ici est directement inspirée des *lentilles* utilisées en programmation fonctionnelle, décrite dans [FGM+07] et [vL11].

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 sous-valeur de type A au sein d'une valeur de type 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 réecrit, 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 considère $L(E)$, 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 traité comme un cas d'erreur.*

$$\begin{aligned}\text{get}_T(t :: q) &= t\\ \text{put}_T(t', t :: q) &= t' :: q\\ \text{get}_Q(t :: q) &= q\\ \text{put}_Q(q', t :: q) &= t :: q'\end{aligned}$$

Alors $T = \langle \text{get}_T | \text{put}_T \rangle \in \text{LENS}_{L(E),E}$ *et* $Q = \langle \text{get}_Q | \text{put}_Q \rangle \in \text{LENS}_{L(E),L(E)}$.
On a par exemple :
$\text{get}_T(1 :: 6 :: 1 :: 8 :: []) = 1$
et :
$\text{put}_Q(7, 3 :: 6 :: 1 :: 5 :: []) = 7 :: 6 :: 1 :: 5 :: []$.

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 :

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

Un exemple est illustré dans la figure 4.2.

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 :*

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

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 :

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

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

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 :*

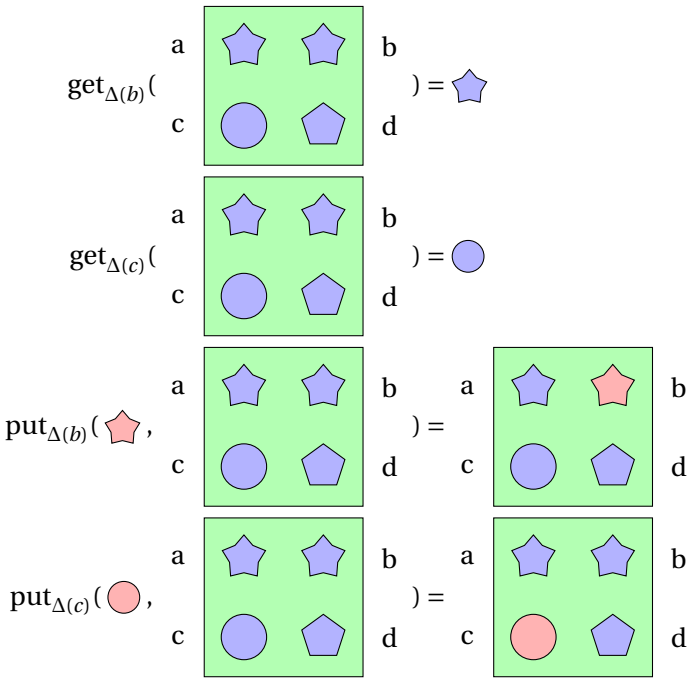


FIGURE 4.2: Fonctionnement d'une lentille indexée

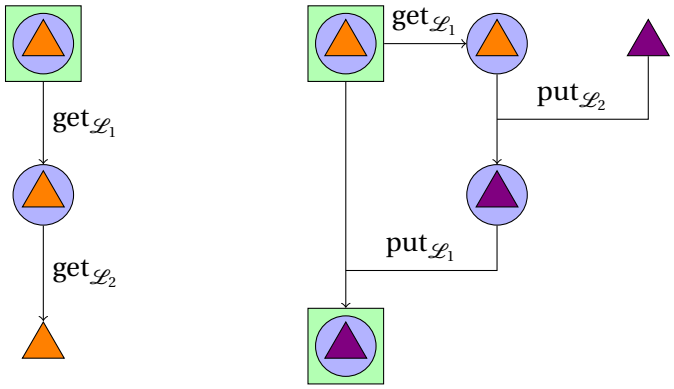


FIGURE 4.3: Composition de lentilles

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

On notera alors $\mathscr{L} = \mathscr{L}_1 \ggg \mathscr{L}_2$.

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

4.2 Fonctionnalités

Fonctions et procédures : Un des problèmes classiques dans les langages impératifs est de distinguer les fonctions (qui retournent une valeur) et les procédures (qui n'en retournent pas). La solution choisie par C est de marquer les procédures comme retournant un "faux" type `void`. Mais c'est uniquement syntaxique : il n'est pas possible de manipuler cette valeur de retour de type `void`.

L'autre possibilité, souvent prise dans les langages fonctionnels, est de ne pas faire de distinction entre ces deux cas et d'interdire les procédures. Les fonctions ne retournant pas de valeur "intéressante" renvoient alors une valeur d'un type à un seul élément appelé $()$ [2], et donc le type sera noté UNIT.

En C, puisqu'il n'y a pas de valeurs de type `void`, la notation `void *` a un sens particulier : elle désigne les pointeurs de type non défini, qui sont compatibles avec n'importe quel autre type de pointeur (c'est la seule forme — rudimentaire — de généricité qu'offre le langage). Ici, la valeur $()$ est une valeur comme les autres, et on peut constuire un pointeur de type UNIT$*$ qui n'aura pas de signification particulière : c'est seulement un pointeur vers une valeur de type UNIT.

Tableaux : ce sont des valeurs composées qui contiennent un certain nombre de valeurs d'un même type. Par exemple, 100 entiers. On accède à ces valeurs par un indice entier, qui dans le cas général n'est pas connu à la compilation. 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 [3].

Les tableaux sont notés $[e_1; \ldots; e_n]$, et le cas dégénéré ($n = 0$) est interdit.

Structures : comme les tableaux, ce sont des valeurs composées mais hétérogènes. Les différents éléments (appelés *champs*) sont désignés par noms l (pour *label*) et de manière statique (il n'y a pas de mécanisme pour faire référence à un nom dans le programme).

Les structures sont notées $\{l_1 : e_1; \ldots; l_n : e_n\}$ et comme dans le cas des tableaux, le cas dégénéré ($n = 0$) est interdit.

Dans le programme, le nom de champ l est complété de la définition complète de la structure S. Celle-ci n'est pas utilisée dans l'évaluation et sera donc décrite au chapitre 5. Bien sûr, écrire la totalité de la structure dans le code du programme serait fastidieux. Nous décrirons dans le chapitre 7 comment obtenir automatiquement ces annotations depuis un langage comme C qui utilise des noms de structures.

4.3 Principes

Nous voulons capturer l'essence de C. Les traits principaux sont les suivants :

Types de données : très simples. Entiers machine, flottants, pointeurs et types composés (structures et tableaux) composés de ceux-ci.

Variables : elles sont mutables, et on peut passer des données par valeur ou par pointeur.

---

2. Cette notation évoque un n-uplet à 0 composante.

3. Comme le fait remarquer Dijkstra, seule la numérotation à partir de 0 a du sens[Dij82].

Flôt de contrôle : il repose sur les construction “if” et “while”. Les autres types de boucle (“for” et “do/while”) peuvent être construits avec ces opérateurs.

Fonctions : le code est organisé en fonctions “simples”, c’est-à-dire qui ne sont pas des fermetures. Même si le corps d’une fonction peut être inclus dans le corps d’une autre, il n’est pas possible d’accéder aux variables de la portée entourante depuis la fonction intérieure.

4.4 Syntaxe

Les figures 4.4, 4.5 et 4.6 présentent notre langage intermédiaire. Il contient la plupart des fonctionnalités présentes dans les langages impératifs comme C.

Un programme est organisé en fonctions, qui contiennent des instructions, qui elles-mêmes manipulent des expressions.

Le flot de contrôle est simplifié par rapport à C : il ne contient que l’alternative ("if") et la boucle "while". Les autres formes de boucle ("do/while" et "for") peuvent être émulées par une boucle "while".

Les fonctionnalités manquantes, et comment les émuler, seront discutés dans le chapitre 9.

Pour l’alternative, on introduit également la forme courte $\textsc{If}(e)\{i\} = \textsc{If}(e)\{i\}\textsc{Else}\{\textsc{Pass}\}$.

Les opérateurs sont donnés dans la figure 4.6.

4.5 Définitions préliminaires

On suppose avoir à notre disposition un ensemble infini dénombrable d’identificateurs ID (par exemple des chaînes de caractères).

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 ensemble de définition). Pour $1 \leq i \leq |u|$, 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 :

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

La concaténation des listes u et v est la liste $u@v = w$ telle que :

$$\begin{aligned} |w| &= |u| + |v| \\ \forall i \in [1;|u|], w_i &= u_i \\ \forall j \in [1;|v|], w_{|u|+j} &= v_j \end{aligned}$$

4.6 Mémoire

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

La mémoire est constituée de variables, qui contiennent des valeurs. Ces variables sont organisées, d’une part en un ensemble de variables globales, et d’autre part en une pile de

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

FIGURE 4.4: Syntaxe – expressions

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 – instructions

Opérateurs binaires	$\boxplus ::= +, -, \times, /, \%$	Arithmétique entière
	$\mid \; +., -., \times., /.$	Arithmétique flottante
	$\mid \; +_p, -_p$	Arithmétique de pointeurs
	$\mid \; =, \neq, \leq, \geq, <, >$	Comparaisons
	$\mid \; \&, \mid, \hat{}$	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 – opérateurs

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.

La structure de pile des 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 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 [4].

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 left-value. Pour représenter quelle partie de la mémoire doit être accédée par cette left-value, on introduit la notion de chemin φ. Un chemin est en quelque sorte une left-value symbolique 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.

Les valeurs, quant à elles, peuvent avoir les formes suivantes (résumé 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 $\widehat{3}$.

  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 : de la même manière, on aurait pu choisir des nombres à 64 bits ou même de précision arbitraire. Les flottants sont les flottants IEEE 754 de 32 bits [oEE08].

- φ : une référence mémoire. Ce chemin correspond à un pointeur sur une left-value. Par exemple, l'expression $\&x$ s'évalue en $\varphi = (5, x)$ si x désigne lexicalement une variable dans le 5^{e} cadre de pile.

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

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
	$\mid *\varphi$	Déréférencement
	$\mid \varphi.l$	Accès à un champ
	$\mid \varphi[n]$	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

FIGURE 4.7: Valeurs

- $\{l_1 : \widehat{v_1; \ldots; l_n} : v_n\}$: une structure. Comme précédemment, on note $\widehat{\{\cdot\}}$ pour dénoter les valeurs.
- $[\widehat{v_1; \ldots; v_n}]$: un tableau. Pareillement, $\widehat{[\cdot]}$ permet de désigner les valeurs. Par exemple, si x vaut 2 et y vaut 3, l'expression $[x, y]$ s'évaluera en valeur $\widehat{[2,3]}$
- $\widehat{f}$: une fonction. Les valeurs fonctions comportent l'intégralité de la définition de la fonction (liste de paramètres, de variables locales et corps). Remarquons que contrairement à certains langages, l'environnement n'est pas capturé (il n'y a pas de clôture lexicale).
- Ω : une erreur. Par exemple le résultat de $5/0$ est Ω_{div}.

La figure E.1 résume comment ces valeurs sont organisées. Une pile est une liste de cadre 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).

Enfin, l'interprétation est définie comme une relation $\cdot \rightarrow \cdot$ 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 $\langle 3, ([], [x, 3])\rangle \rightarrow \langle \widehat{3}, ([], [x, 3])\rangle$ L'évaluation des expressions est détaillée dans la section 4.11.
- un couple $\langle i, m\rangle$ où i est une instruction et m un état mémoire. La réduction instructions est traitée dans la section 4.12.
- une erreur Ω. La propagation des erreurs est détaillée dans la section 4.13.

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'interpréteur	$\Xi ::= \langle e, m \rangle$	Expression, mémoire
	$\mid \langle i, m \rangle$	Instruction, mémoire
	$\mid \Omega$	Erreur

FIGURE 4.8: Composantes d'un état mémoire

4.7 Opérations sur les valeurs

Un certain nombre d'opérations est possibles 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 meme "forme" (même taille pour les tableaux, et même champs pour les structures) et que toutes leurs sous-valeurs sont égales deux à deux. Les références mémoire φ sont égales lorsque les chemins qu'ils décrivent sont syntaxiquement égaux[5].
- 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 opérarandes 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.
- 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.

5. Il est donc possible que deux pointeurs pointent sur la même adresse mais soient considérés différents. La raison pour ce choix est que la comparaison doit pouvoir se faire sans accéder à la mémoire.

- les opérateurs arithmétiques unaires +, −, +. et −. sont équivalents à l'opération binaire correspondante avec 0 ou 0. comme première opérande.
- ~ inverse tous les bits de son opérande. ! est une version logique, c'est à dire que !0 = 1 et si $n \neq 0$, $!n = 0$.

4.8 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.

$$\begin{aligned}\text{Lookup}((s,g),x) &= (|s|,x) \ si\ |s| > 0\ et \exists (x,v) \in s_1\\ \text{Lookup}((s,g),x) &= x\ si\ (x,v) \in g\end{aligned}$$

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* $c = ((x_1,v_1),\ldots,(x_n,v_n))$ *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 (figure 4.9(b)) :

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

De même on définit le dépilement (figure 4.9(c)) :

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

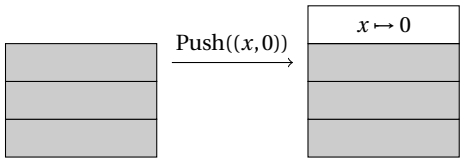
On définit aussi une opération de nettoyage de pile, qui sera utile pour les retours de fonction.

En effet, si une référence au dernier cadre est toujours présente après le retour, elle pourra se résoudre en un objet différent plus tard dans l'exécution du programme.

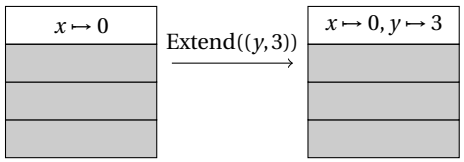
La fonction Cleanup est donnée par :

$$\begin{aligned}\text{Cleanup}(s,g) &= (s',g')\\ o\grave{u} g' &= \text{CleanupList}(|s|,g)\\ s' &= [\text{CleanupList}(|s|,s_1),\ldots,\text{CleanupList}(|s|,s_n)]\\ \text{CleanupList}(p,u) &= \{(x,v) \in u / v \text{ n'est pas une adresse}\}\\ &\quad \cup \{(x,\varphi) \in u / \text{Live}(p,\varphi)\}\\ \text{Live}(p,(n,x)) &= n < p\\ \text{Live}(p,(x)) &= \text{Vrai}\\ \text{Live}(p,*\varphi) &= \text{Live}(p,\varphi)\\ \text{Live}(p,\varphi.l) &= \text{Live}(p,\varphi)\\ \text{Live}(p,\varphi[n]) &= \text{Live}(p,\varphi)\end{aligned}$$

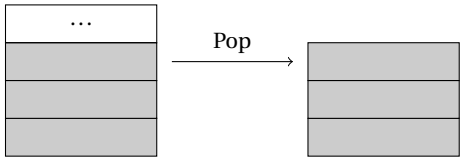
Sans cette règle, examinons le programme suivant :



(a) Empilement



(b) Extension de cadre



(c) Dépilement

FIGURE 4.9: Opérations de pile

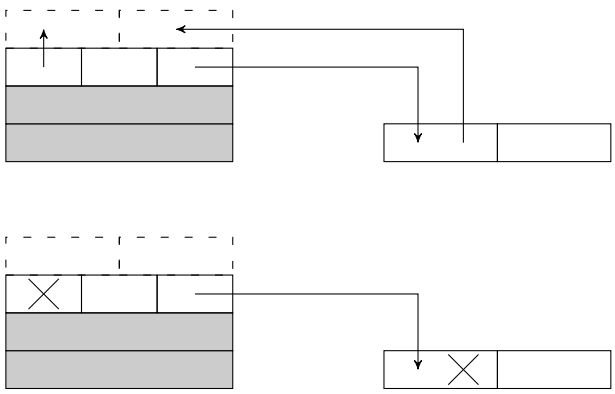


FIGURE 4.10: Nettoyage d'un cadre de pile

```
f ()                g (p)             h ()
  (x=0)               (x=0.0)           (p=f())
{                   {                 {
  return (&x);        *p = 1;           g(p);
}                   }                 }
```

L'exécution de $h()$ donne à p la valeur $(1, x)$. Donc en arrivant dans g, le déréférencement de p va modifier x.

4.9 Accesseurs

On définit ici quelques lentilles.

Définition 4.6 (Accès à une liste d'associations)**.** *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.

L'accesseur $[\cdot]_L$ est défini par :

$$l[x]_L = v \text{ où } \{v\} = \{y/(x,y) \in l\}$$
$$l[x \leftarrow v]_L = (x,v) :: \{(y,v) \in g(x) / y \neq x\})$$

Définition 4.7 (Accès par adresse)**.** *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.

Pour $m = (s, g)$,

$$\begin{aligned}
m[x]_A &= g[x]_L && \textit{Lecture d'une globale}\\
m[(n,x)]_A &= s_{|l|-n+1}[x]_L && \textit{Lecture d'une locale}\\
m[x \leftarrow v]_A &= (s, g[x \leftarrow v]_L) && \textit{Écriture d'une globale}\\
m[(n,x) \leftarrow v]_A &= (s', g) && \textit{Écriture d'une locale}\\
&\text{où } s'_{|l|-n+1} = s_{|l|-n+1}[x \leftarrow v]_L\\
&\forall i \neq |l| - n + 1, s'_i = s_i
\end{aligned}$$

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 $|l| - n + 1$ apparaît dans la définition précédente.

Définition 4.8 (Accès par champ)**.** *Les valeurs qui sont des structures possèdent des sous-valeurs, associées à des noms de champ.*

L'accesseur $[\cdot]_L$ 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.

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

Définition 4.9 (Accès par indice)**.** *On définit de même un accesseur $[\cdot]_I$ pour les accès par indice à des valeurs tableaux. Néanmoins le paramètre indice est toujours un entier et pas une expression arbitraire.*

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

Définition 4.10 (Accès par chemin)**.** *L'accès par chemin* Φ *permet de lire et de modifier la mémoire en profondeur.*

On peut accéder directement à une variable :

$$\Phi(a) = A(a)$$

Les accès à des sous-valeurs se font en composant les accesseurs (définition 4.3) :

$$
\begin{aligned}
\Phi(\varphi.l) &= \Phi(\varphi) \ggg L(l) \\
\Phi(\varphi[i]) &= \Phi(\varphi) \ggg I(i)
\end{aligned}
$$

Le déréférencement est défini comme suit :

$$
\begin{aligned}
m[*\varphi]_\Phi &= m[\varphi']_\Phi \text{ où } \varphi' = m[\varphi]_\Phi \\
m[*\varphi \leftarrow v]_\Phi &= m[\varphi' \leftarrow v]_\Phi \text{ où } \varphi' = m[\varphi]_\Phi
\end{aligned}
$$

Enfin, l'accès à la mémoire par le pointeur nul provoque une erreur :

$$
\begin{aligned}
m[\text{NULL}]_\Phi &= \Omega_{ptr} \\
m[\text{NULL} \leftarrow v]_\Phi &= \Omega_{ptr}
\end{aligned}
$$

Cette dernière définition mérite une explication. Dans le cas de la lecture, il suffit d'appliquer les bons accesseurs : $[\cdot]_L$ pour $\varphi.l$, etc.

En revanche, la modification est plus complexe. Les deux premiers cas ($\varphi = a$ et $\varphi = *\varphi'$) modifient directement une valeur complète (en modifiant une association), mais les deux suivants ($\varphi = \varphi'.l$ et $\varphi = \varphi'[i]$) ne font qu'altérer une sous-valeur existante. Il est donc nécessaire de procéder en 3 étapes :

- obtenir la valeur à modifier (soit $m[\varphi]_\varphi$)
- construitre une valeur altérée (en appliquant par exemple $[l \leftarrow v]_L$)
- affecter cette valeur au même chemin (le $m[\varphi \leftarrow \ldots]_\varphi$ externe)

Dans la suite, on notera uniquement $[\cdot]$ tous ces accesseurs lorsque ce n'est pas ambigü.

Contextes	$C ::= C_L$
	$\mid C \boxplus e$
	$\mid v \boxplus C$
	$\mid \boxminus C$
	$\mid C \leftarrow e$
	$\mid \varphi \leftarrow C$
	$\mid \{l_1 : v_1; \ldots; l_i : C; \ldots; l_n : e_n\}$
	$\mid [v_1; \ldots; C; \ldots; e_n]$
	$\mid C(e_1, \ldots, e_n)$
	$\mid f(v_1, \ldots, C, \ldots, e_n)$
Contextes (left-values)	$C_L ::= \bullet$
	$\mid *C_L$
	$\mid C_L.l_S$
	$\mid C_L[e]$
	$\mid \varphi[C]$
Contextes (instructions)	$C_I ::= C_I; i$
	$\mid \text{IF}(C)\{i_1\}\text{ELSE}\{i_2\}$
	$\mid \text{RETURN}(C)$
	$\mid \text{DECL } x = C \text{ IN}\{i\}$
	$\mid C$

FIGURE 4.11: Contextes d'exécution

4.10 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 4.11).
Le deuxième est résolu les règles d'inférence suivantes :

$$\frac{\langle e, m\rangle \rightarrow \langle e', m'\rangle}{\langle C(\!|e|\!), m\rangle \rightarrow \langle C(\!|e'|\!), m'\rangle}\ (\text{CTX}) \qquad \frac{\langle l\,v, m\rangle \rightarrow \langle l\,v', m'\rangle}{\langle C_L(\!|l\,v|\!)_L, m\rangle \rightarrow \langle C_L(\!|l\,v'|\!)_L, m'\rangle}\ (\text{CTX-LV})$$

$$\frac{\langle i, m\rangle \rightarrow \langle i', m'\rangle}{\langle C_I(\!|i|\!)_I, m\rangle \rightarrow \langle C_I(\!|i'|\!)_I, m'\rangle}\ (\text{CTX-INSTR})$$

Enfin, le troisième revient à définit l'opérateur de substitution $\cdot(\!|\cdot|\!)$ présent dans la règle précédente. Afin de pouvoir appliquer des substitution au niveau des left-values et des instructions, on définit aussi respectivement $\cdot(\!|\cdot|\!)_L$ et $\cdot(\!|\cdot|\!)_I$.

Dans la définition de l'ensemble des contextes, chaque cas hormis le cas de base fait apparaître exactement un "C". Chaque contexte est donc constitué d'exactement un "trou" $\bullet$ (une dérivation de C est toujours linéaire). L'opération de substitution consiste à remplacer ce trou, comme décrit dans la figure 4.12.

Par exemple, décomposons l'évaluation de $e_1 \boxplus e_2$ en $v = v_1 \mathbin{\widehat{\boxplus}} v_2$ depuis un état mémoire m :

1. on commence par évaluer, d'une manière ou d'une autre, 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 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, $\langle v_1 \boxplus v_2, m''\rangle \to^* \langle v_1 \mathbin{\widehat{\boxplus}} v_2, m''\rangle$
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 $v = v_1 \mathbin{\widehat{\boxplus}} v_2$.

4.11 Expressions

Définition 4.11 (Évaluation d'une expression)**.** *L'évaluation d'une expression e se fait sous un état mémoire particulier m et est susceptible de modifier celui-ci en le transformant en un nouveau m'. Le résultat est toujours une valeur v, c'est à dire que nous présentons pour les expressions une sémantique à grands pas. Cette évaluation est notée :*

$$\langle e, m\rangle \to \langle v, m'\rangle$$

Définition 4.12 (Évaluation d'une left-value)**.** *L'évaluation d'une left-value l v produit un "chemin" φ dans une variable, qui est en fait équivalent à une left-value dont toutes les sous-expressions (d'indices) ont été évaluées.*

On note :

$$\langle l\, v, m\rangle \to \langle \varphi, m'\rangle$$

Puisque des left-values peuvent apparaitre dans les expressions, et des expressions dans les left-values (en indice de tableau), leurs règles d'évaluation sont mutuellement récursives.

Left-values

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 left-value sur laquelle porte ce modificateur, et on place

$$
\begin{aligned}
\bullet(\!|e_0|\!) &= e_0 \\
(C \boxplus e)(\!|e_0|\!) &= C(\!|e_0|\!) \boxplus e \\
(v \boxplus C)(\!|e_0|\!) &= v \boxplus C(\!|e_0|\!) \\
(\boxminus C)(\!|e_0|\!) &= \boxminus C(\!|e_0|\!) \\
(*C)(\!|e_0|\!) &= * C(\!|e_0|\!) \\
(C.l_S)(\!|e_0|\!) &= C(\!|e_0|\!).l_S \\
(\varphi[C])(\!|e_0|\!) &= \varphi[C(\!|e_0|\!)] \\
(C[e])(\!|e_0|\!) &= C(\!|e_0|\!)[e] \\
(C \leftarrow e)(\!|e_0|\!) &= C(\!|e_0|\!) \leftarrow e \\
(\varphi \leftarrow C)(\!|e_0|\!) &= \varphi \leftarrow C(\!|e_0|\!) \\
\{l_1 : v_1; \ldots; l_i : C; \ldots; l_n : e_n\}(\!|e_0|\!) &= \{l_1 : v_1; \ldots; l_i : C(\!|e_0|\!); \ldots; l_n : e_n\} \\
[v_1; \ldots; C; \ldots; e_n](\!|e_0|\!) &= [v_1; \ldots; C(\!|e_0|\!); \ldots; e_n] \\
C(e_1, \ldots, e_n)(\!|e_0|\!) &= C(\!|e_0|\!)(e_1, \ldots, e_n) \\
f(v_1, \ldots, C, \ldots, e_n)(\!|e_0|\!) &= f(v_1, \ldots, C(\!|e_0|\!), \ldots, e_n)
\end{aligned}
$$

$$
\begin{aligned}
(C; i)(\!|e_0|\!) &= C(\!|e_0|\!); i \\
(\textsc{If}(C)\{i_1\}\textsc{Else}\{i_2\})(\!|e_0|\!) &= \textsc{If}(C(\!|e_0|\!))\{i_1\}\textsc{Else}\{i_2\} \\
(\textsc{Return}(C))(\!|e_0|\!) &= \textsc{Return}(C(\!|e_0|\!))
\end{aligned}
$$

$$
\begin{aligned}
C_L(\!|e_0|\!) &= C_L(\!|e_0|\!)_L \\
C \boxplus e(\!|e_0|\!) &= C(\!|e_0|\!) \boxplus e \\
v \boxplus C(\!|e_0|\!) &= v \boxplus C(\!|e_0|\!) \\
\boxminus C(\!|e_0|\!) &= \boxminus C(\!|e_0|\!) \\
C \leftarrow e(\!|e_0|\!) &= C(\!|e_0|\!) \leftarrow e \\
\varphi \leftarrow C(\!|e_0|\!) &= \varphi \leftarrow C(\!|e_0|\!) \\
\{l_1 : v_1; \ldots; l_i : C; \ldots; l_n : e_n\}(\!|e_0|\!) &= \{l_1 : v_1; \ldots; l_i : C(\!|e_0|\!); \ldots; l_n : e_n\} \\
[v_1; \ldots; C; \ldots; e_n](\!|e_0|\!) &= [v_1; \ldots; C(\!|e_0|\!); \ldots; e_n] \\
C(e_1, \ldots, e_n)(\!|e_0|\!) &= C(\!|e_0|\!)(e_1, \ldots, e_n) \\
f(v_1, \ldots, C, \ldots, e_n)(\!|e_0|\!) &= f(v_1, \ldots, C(\!|e_0|\!), \ldots, e_n)
\end{aligned}
$$

$$
\begin{aligned}
\bullet(\!|l_0|\!)_L &= \bullet \\
*C_L(\!|l_0|\!)_L &= * C_L(\!|l_0|\!)_L \\
C_L.l_S(\!|l_0|\!)_L &= C_L(\!|l_0|\!)_L.l_S \\
C_L[e](\!|l_0|\!)_L &= C_L(\!|l_0|\!)_L[e] \\
\varphi[C](\!|e_0|\!) &= \varphi[C(\!|e_0|\!)]
\end{aligned}
$$

FIGURE 4.12: Substitution dans les contextes d'évaluation

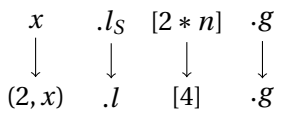


FIGURE 4.13: Évaluation des left-values.

le même modificateur sur le chemin résultant. Dans le cas des champs de structure, la définition de la structure S n'est pas prise en compte.

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

Enfin, pour évaluer un chemin dans un tableau, on commence par procéder comme précédemment, c'est-à-dire en évaluant la left-value 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 left-values 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.

Un exemple d'évaluation est donné dans la figure 4.13.

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) $\widehat{3}$.

$$\frac{}{\langle c, m\rangle \to \langle \widehat{c}, m\rangle}\ (\text{EXP-CST})$$

De même, une fonction est déjà évaluée :

$$\frac{}{\langle f, m\rangle \to \langle \widehat{f}, m\rangle}\ (\text{EXP-FUN})$$

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

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

Pour évaluer une expression constituée d'un opérateur, on évalue une 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 classique entre entiers. Comme précisé dans la section 4.7, la division par zéro via /, % ou /. provoque l'erreur Ω_{div}.

$$\frac{}{\langle \boxminus\, v, m\rangle \to \langle \widehat{\boxminus}\, v, m\rangle}\ (\text{EXP-UNOP}) \qquad \frac{}{\langle v_1 \boxplus v_2, m\rangle \to \langle v_1 \mathbin{\widehat{\boxplus}} v_2, m\rangle}\ (\text{EXP-BINOP})$$

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 $\varphi[n]$. On a alors :

$$\begin{aligned}\varphi[n] +_p m &= \varphi[n+m] \\ \varphi[n] -_p m &= \varphi[n-m]\end{aligned}$$

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 :

$$\begin{aligned}\varphi +_p m &= \Omega_{ptr} \text{ si } \nexists(\varphi', n), \varphi = \varphi'[n] \\ \varphi -_p m &= \Omega_{ptr} \text{ si } \nexists(\varphi', n), \varphi = \varphi'[n] \\ \textsc{Null} +_p m &= \Omega_{ptr} \\ \textsc{Null} -_p m &= \Omega_{ptr}\end{aligned}$$

Pour prendre l'adresse d'une variable, il suffit de résoudre celle-ci dans l'état mémoire courant.

$$\frac{}{\langle \& \varphi, m\rangle \rightarrow \langle \widehat{\&} \varphi, m\rangle} \text{ (Exp-AddrOf)}$$

L'affectation se déroule 3 étapes : d'abord, l'expression est évaluée en une valeur v. Ensuite, la left-value 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.

$$\frac{}{\langle \varphi \leftarrow v, m\rangle \rightarrow \langle v, m[\varphi \leftarrow v]_\Phi\rangle} \text{ (Exp-Set)}$$

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.

$$\frac{}{\langle \{l_1 : v_1;\ldots;l_n : v_n\}, m\rangle \rightarrow \langle \{l_1 : \widehat{v_1;\ldots;l_n} : v_n\}, m\rangle} \text{ (Exp-Struct)}$$

$$\frac{}{\langle [v_1,\ldots,v_n], m\rangle \rightarrow \langle \widehat{[v_1,\ldots,v_n]}, m\rangle} \text{ (Exp-Array)}$$

Les contextes utilisés dans l'appel de fonction sont similaires. Tout d'abord, les arguments sont évalués et placés dans un nouveau cadre de pile. Ensuite, le corps de la fonction est évalué jusqu'à se réduire en une instruction $\textsc{Return}(v)$. Enfin, le cadre précédemment utilisé est dépilé.

La dernière étape consiste à nettoyer la mémoire de références à l'ancien cadre de pile (on utilise la fonction Cleanup définie dans la section).

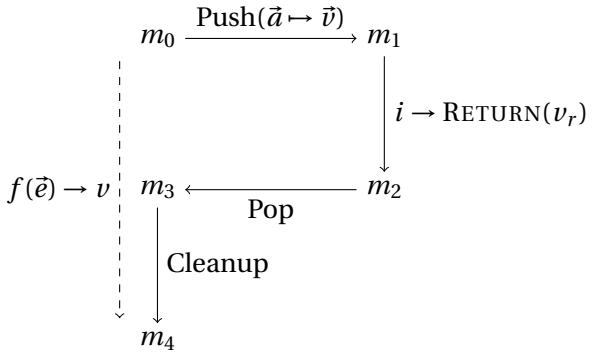


FIGURE 4.14: Appel d'une fonction. La taille de la pile croît de gauche à droite, et les réductions se font de haut en bas.

$$\frac{\begin{array}{c} f = \text{fun}(a_1, \ldots, a_n)\{i\} \qquad m_1 = \text{Push}(m_0, ((a_1 \mapsto v_1), \ldots, (a_n \mapsto v_n))) \\ \langle i, m_1 \rangle \to \langle \text{RETURN}(v), m_2 \rangle \qquad m_3 = \text{Pop}(m_2) \qquad m_4 = \text{Cleanup}(m_3) \end{array}}{\langle f(v_1, \ldots, v_n), m_0 \rangle \to \langle v, m_4 \rangle} \text{ (EXP-CALL)}$$

Cette évaluation est décrite dans la figure 4.14.

4.12 Instructions

Contrairement à l'évaluation des expressions, on choisit une sémantique de réécriture à petits pas. La sémantique fonctionne de la manière suivante : partant d'un état mémoire m, on veut exécuter une instruction i. Les règles d'évaluation suivantes permettent de réduire le problème en se ramenant à l'exécution d'une instruction i' "plus simple" en partant d'un état mémoire m'. Un tel pas est noté :

$$\langle i, m \rangle \to \langle i', m' \rangle$$

Par exemple, exécuter $x \leftarrow 3; y \leftarrow x$ revient à évaluer $y \leftarrow x$ depuis un état mémoire dans lequel on a déjà réalisé la première affectation. La seconde affectation se réalise de même et permet de réécrire l'instruction restante en PASS :

$$\begin{aligned} \langle (x \leftarrow 3; y \leftarrow x), m \rangle &\to \langle y \leftarrow x, m[x \mapsto \widehat{3}] \rangle \\ &\to \langle \text{PASS}, m[x \mapsto \widehat{3}][y \mapsto \widehat{3}] \rangle \end{aligned}$$

Il n'est pas possible de réduire plus loin l'instruction PASS. Dans un tel cas, l'évaluation est terminée.

Les seuls cas terminaux sont PASS et RETURN(e).

Les cas de la séquence et de l'affectation ont été utilisés dans l'exemple ci-dessus.

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

L'évaluation de DECL $x = v$ IN$\{i\}$ sous m se fait en trois parties :

- on crée un environnement mémoire m' en ajoutant à m l'association $x \mapsto v$.

- sous m', on évalue i jusqu'à atteindre une instruction PASS ou RETURN(v_r) sous un état m''. C'est cette instruction qui sera retournée.
- on enlève x de m'' et on enlève de cet état mémoire les pointeurs invalides.

$$\frac{m' = \text{Extend}(m, x, v) \qquad \langle i, m' \rangle \to \langle \text{PASS}, m'' \rangle \qquad m''' = \text{Cleanup}(m'' - x)}{\langle \text{DECL } x = v \text{ IN}\{i\}, m \rangle \to \langle \text{PASS}, m''' \rangle} \text{ (DECL)}$$

$$\frac{\begin{array}{c} m' = \text{Extend}(m, x, v) \\ \langle i, m' \rangle \to \langle \text{RETURN}(v_r), m'' \rangle \qquad m''' = \text{Cleanup}(m'' - x) \end{array}}{\langle \text{DECL } x = v \text{ IN}\{i\}, m \rangle \to \langle \text{RETURN}(v_r), m''' \rangle} \text{ (DECL-RETURN)}$$

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 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.

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

Pour traiter la boucle, on peut être tenté de procéder de la même manière :

$$\frac{v \neq 0}{\langle \text{WHILE}(v)\{i\}, m \rangle \to \langle \text{PASS}, m \rangle} \text{ (WHILE-FALSE-BAD)}$$

$$\frac{}{\langle \text{WHILE}(0)\{i\}, m \rangle \to \langle i; \text{WHILE}(e)\{i\}, m' \rangle} \text{ (WHILE-TRUE-BAD)}$$

Mais la seconde règle est impossible : puisque e a déjà été évaluée, il est impossible de la réintroduire non évaluée en partie droite.

À la place, on exprime la sémantique de la boucle comme une simple règle de réécriture :

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

Cette règle revient à dire qu'on peut dérouler une boucle. Pour la comprendre, on peut remarquer qu'une boucle "while" est en réalité équivalente une infinité de "if" imbriqués.

```
while(e) {
    i
}
```

$\widehat{=}$

```
if(e) {
    i;
    if(e) {
        i;
        if(e) {
            i;
            if(e) {
                i;
                if(e) {
                    i;
                    ...
                }
            }
        }
    }
}
```

Donc en remplaçant le second "if" par le "while", on obtient :

```
while(e) {
    i
}
```
$\widehat{=}$
```
if(e) {
    i;
    while(e) {
        i
    }
}
```

Enfin, si un "return" apparaît dans une séquence, on peut supprimer la suite :

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

4.13 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-structure est présente grâce aux contexte C dans la règle suivante :

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

4.14 Phrases

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

Il est donc logique que l'évaluation d'une phrase fasse passer d'un état mémoire à un autre :

$$m \Vdash p \to m'$$

L'évaluation d'une expression est uniquement faite pour ses effets de bord. Par exemple, après avoir défini les fonctions du programme, on pourra appeller main().

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle}{m \Vdash e \to m'} \text{ (T-EXP)}$$

La déclaration d'une variable globale (avec un initialiseur) consiste à évaluer cet initialiseur et à étendre l'état mémoire avec ce couple (variable, valeur).

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle}{(s, g) \Vdash x = e \to (s, (x \mapsto v) :: g)} \text{ (T-VAR)}$$

4.15 Exécution

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

On commence par étendre l'extension $\rightarrow^*$ au listes de la relation $\rightarrow$:

$$\frac{}{m \Vdash [\,] \rightarrow^* m}\ (\text{T*-NIL}) \qquad \frac{m \Vdash p \rightarrow m' \qquad m' \Vdash ps \rightarrow^* m''}{m \Vdash p :: ps \rightarrow^* m''}\ (\text{T*-CONS})$$

L'exécution d'un programme est définie par :

$$\frac{([\,],[\,]) \Vdash P \rightarrow^* m}{\Vdash P \rightarrow^* m}\ (\text{PROG})$$

4.16 Exemple : l'algorithme d'Euclide

Version par divisions successives :

```
function gcd(a, b)
  var t = 0;
  while b != 0
    t = b
    b = a mod b
    a = t
  return a
```

Soit :

$$f(a,b)(t=0)\{\text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \text{RETURN}(a)\}$$

$$\langle f(1071,462), m\rangle \rightarrow ?$$

$$\langle \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \text{RETURN}(a), m[a \mapsto 1071][b \mapsto 462][t \mapsto 0]\rangle \rightarrow ?$$

(on notera cet état $s_0 = \langle i_0, m_0\rangle$)

$$\langle a = 0, m_0\rangle \rightarrow \langle 0, m_0\rangle$$

donc

$$\langle \text{IF}(a = 0)\{\text{RETURN}(b)\}, m_0\rangle \rightarrow \langle \text{PASS}, m[a \mapsto 1071][b \mapsto 462]\rangle$$

$$
\begin{aligned}
s_0 &\to \langle \textsc{If}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t; \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}\}; \textsc{Return}(a), m_0\rangle && (4.1)\\
&\to \langle t \leftarrow b; b \leftarrow a\%b; a \leftarrow t; \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \textsc{Return}(a), m_0\rangle && (4.2)\\
&\to \langle b \leftarrow a\%b; a \leftarrow t; \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \textsc{Return}(a), m_0\rangle && (4.3)\\
&\to \langle a \leftarrow t; \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \textsc{Return}(a), m_0''\rangle && (4.4)\\
&\to \langle \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \textsc{Return}(a), m_1\rangle && (4.5)\\
&\to \langle \textsc{If}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t; \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}\}; \textsc{Return}(a), m_1\rangle && (4.6)\\
&\to \langle t \leftarrow b; b \leftarrow a\%b; a \leftarrow t; \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \textsc{Return}(a), m_1\rangle && (4.7)\\
&\to \langle \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \textsc{Return}(a), m_2\rangle && (4.8)\\
&\to \langle \textsc{If}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t; \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}\}; \textsc{Return}(a), m_2\rangle && (4.9)\\
&\to \langle t \leftarrow b; b \leftarrow a\%b; a \leftarrow t; \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \textsc{Return}(a), m_2\rangle && (4.10)\\
&\to \langle \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}; \textsc{Return}(a), m_3\rangle && (4.11)\\
&\to \langle \textsc{If}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t; \textsc{While}(b \neq 0)\{t \leftarrow b; b \leftarrow a\%b; a \leftarrow t\}\}; \textsc{Return}(a), m_3\rangle && (4.12)\\
&\to \langle \textsc{Pass}; \textsc{Return}(a), m_3\rangle && (4.13)\\
&\to \langle \textsc{Return}(a), m_3\rangle && (4.14)
\end{aligned}
$$

$$
\begin{aligned}
m_0' &= m_0[t \mapsto 462] = m[a \mapsto 1071][b \mapsto 462][t \mapsto 462]\\
m_0'' &= m_0'[b \mapsto 147] = m[a \mapsto 1071][b \mapsto 147][t \mapsto 462]\\
m_1 &= m_0''[a \mapsto 462] = m[a \mapsto 462][b \mapsto 147][t \mapsto 462]\\
m_2 &= m_1[t \mapsto 147][b \mapsto 21][a \mapsto 147] = m[a \mapsto 147][b \mapsto 21][t \mapsto 147]\\
m_3 &= m_2[t \mapsto 21][b \mapsto 0][a \mapsto 21] = m[a \mapsto 21][b \mapsto 0][t \mapsto 21]
\end{aligned}
$$

CHAPITRE 5

TYPAGE

Many C programmers believe that "strong typing" just means pounding extra hard on the keyboard.

Peter van den Linden

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).

Le but d'un tel système de types est de rejeter les programmes qui sont "évidemment faux", c'est à dire dont on peut prouver qu'il 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.

5.1 Principe

Le principe 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 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.

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 précis.

```
f()
(x=0)
{
  x = 1
  return x
}
```

(a) Programme bien formé

```
f()
(x=0)
{
  x = 1
  return (*x)
}
```

(b) Programme mal formé

FIGURE 5.1: Programmes bien et mal formés

Type	$t ::=$ INT	Entier
	| FLOAT	Flottant
	| UNIT	Unité
	| $t*$	Pointeur
	| $t[]$	Tableau
	| S	Structure
	| $(t_1, \ldots, t_n) \rightarrow t$	Fonction
Structure	$S ::= \{l_1 : t_1; \ldots; l_n : t_n\}$	Structure simple
Environnement de typage	$\Gamma ::= []$	Environnement vide
	| $(a, t) :: \Gamma'$	Extension

FIGURE 5.2: Types et environnements de typage

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 où 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.

5.2 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.

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

Plus précisément, un environnement Γ est une liste de couples (variable, type).

Par exemple, $(p, \text{INT}*) \in \Gamma$ permet de typer (sous Γ) l'expression p en INT$*$, $*p$ en INT et $p +_p 4$ en INT$*$.

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.

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 :

$\boxed{\Gamma \vdash ps \to^* \Gamma'}$

$$\frac{}{\Gamma \vdash [\,] \to^* \Gamma}\ (\text{T*-NIL}) \qquad \frac{\Gamma_1 \vdash p \to \Gamma_2 \qquad \Gamma_2 \vdash ps \to^* \Gamma_3}{\Gamma_1 \vdash p :: ps \to^* \Gamma_3}\ (\text{T*-CONS})$$

$\boxed{\Gamma \vdash P}$

$$\frac{[\,] \vdash P \to^* \Gamma}{\vdash P}\ (\text{PROG})$$

FIGURE 5.3: Typage d'une suite de phrases et d'un programme

$$\Gamma \vdash i$$

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

$$\Gamma \vdash p \to \Gamma'$$

si le typage de la phrase p transforme l'environnement Γ en Γ'.

On étend cette notation aux suites de phrases, ce qui définit le typage d'un programme, ce que l'on note $\vdash P$ (figure 5.3).

5.3 Expressions

Littéraux

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

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

Le pointeur nul, quant à lui, est compatible avec tous les types pointeur.

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

Enfin, le littéral unité a le type UNIT.

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

Left-values

Rappelons que l'environnement de typage Γ contient le type des variables accessibles du programme. Le cas où la left-value 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 left-value déréférencée. Si elle a un type pointeur, la valeur déréférencée est du type pointé.

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

Pour une left-value indexée (l'accès à tableau), on s'assure que l'indice soit entier, et que la left-value a un type tableau : le type de l'élement est encore une fois le type de base du type tableau (t pour $t[]$).

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

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 le champ.

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.

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

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 (figure 5.4). Les opérateurs = et $\neq$ renvoient alors un INT :

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

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

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

FIGURE 5.4: Jugements d'égalité sur les types

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

Les opérateurs unaires "+" et "−" appliquent aux INT, et leurs équivalents "+." et "−." aux FLOAT.

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

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

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 left-value 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 left-value (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.

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

5.4 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.

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

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 son initialiseur.

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

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

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

$\boxed{\Gamma \vdash p \to \Gamma'}$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e \to \Gamma}\ \text{(T-EXP)} \qquad \frac{\Gamma \vdash e : t \qquad \Gamma' = (x,t),\Gamma}{\Gamma \vdash x = e \to \Gamma'}\ \text{(T-VAR)}$$

FIGURE 5.5: Typage des phrases

5.5 Fonctions

Le typage des fonctions fait intervenir une variable virtuelle $\underline{R}$. Cela revient à typer l'instruction RETURN(e) comme $\underline{R} \leftarrow e$.

$$\frac{\Gamma \vdash \underline{R} \leftarrow e}{\Gamma \vdash \text{RETURN}(e)}\ \text{(RETURN)}$$

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 (s'il existe) 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.

$$\frac{\Gamma' = (\Gamma - \underline{R}), \vec{a} : \vec{t}, \underline{R} : t_r \qquad \Gamma' \vdash i}{\Gamma \vdash \text{fun}(\vec{a})\{i\} : \vec{t} \to t_r}\ \text{(FUN)}$$

Cette règle utilise les notations suivantes :

$$\vec{a} \overset{\text{def}}{=\!=} (a_1, \ldots, a_n) \text{ où } n = |a|$$
$$\vec{a} : \vec{t} \overset{\text{def}}{=\!=} a_1 : t_1, \ldots a_n : t_n \text{ où } n = |a|$$

5.6 Phrases

Le typage des phrases est détaillé dans la figure 5.5. 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 cette nouvelle valeur.

5.7 Sûreté du typage

5.7.1 But

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 cette section, 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 les 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.

5.7.2 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 (ou type sémantique) τ aux valeurs v. Cela est fait sous la forme d'un jugement $m \vDash v : \tau$.

Ces types sémantiques 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 statiquement. On le remplace donc par un cas plus simple où seul l'arité est conservée.

Type sémantique

$\tau ::=$	INT	Entier
$\mid$	FLOAT	Flottant
$\mid$	UNIT	Unité
$\mid$	$\tau *$	Pointeur
$\mid$	$\tau[\,]$	Tableau
$\mid$	$\{l_1 : \tau_1; \ldots; l_n : \tau_n\}$	Structure
$\mid$	$(\tau_1, \ldots, \tau_n) \to \tau$	Fonction

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.

La prochaine étape est de définir une relation de compatibilité entre les types sémantiques τ et statiques t. Nous noterons ceci sous la forme d'un jugement $\text{Comp}(\tau, t)$. Les règles sont décrites dans la figure 5.7, la règle importante étant COMP-FUN.

Grâce à ce jugement, on peut donner la définition suivante.

Définition 5.1 (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 les types sémantiques des variables visibles coïncident avec leurs types statiques.*

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

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

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

$$\frac{\forall i \in [1;n].m \vDash v_i : t_i}{m \vDash \{l_1 : \widehat{v_1;\ldots;l_n} : v_n\} : \{l_1 : t_1;\ldots;l_n : t_n\}}\ (\text{S-STRUCT}) \qquad \frac{}{m \vDash \text{fun}(x_1,\ldots,x_n)\{i\} : \text{FUN}_n}\ (\text{S-FUN})$$

FIGURE 5.6: Règles de typage sémantique

$\boxed{\text{Comp}(\tau, t)}$

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

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

$$\frac{}{\text{Comp}(\text{FUN}_n, (t_1,\ldots,t_n) \to t)}\ (\text{COMP-FUN})$$

FIGURE 5.7: Compatibilité entre types sémantiques et statiques

$$\Gamma \vDash m \overset{\text{def}}{=\!=} \forall (x \mapsto v) \in \text{Visible}(m).\exists \tau, t. \begin{cases} \Gamma \vdash x : t \\ m \vDash v : \tau \\ \text{Comp}(\tau, t) \end{cases}$$

Visible(m) *désigne l'ensemble des couples d'associations (variable, valeur) du cadre de pile le plus récents ainsi que des variables globales. Cela correspond aussi aux variables présentes dans l'environnement de typage.*

5.7.3 Progrès et préservation

On commence par énoncer quelques lemmes utiles dans la démonstration de ces théorèmes.

Les règles précédentes ont la particularité suivante : pour chaque forme syntaxique, il n'y a souvent qu'une règle qui peut s'appliquer. Cela permet de déduire quelle règle il faut appliquer pour vérifier (ou inférer) le type d'une expression.

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* $\Gamma \vdash *lv : t$, *alors* $\Gamma \vdash lv : t*$
  - *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* $\Gamma \vdash \text{fun}(a_1, \ldots, a_n)\{i\} : t$, *alors il existe* $(t_1, \ldots, t_n)$ *et* t' *tels que* $t' = (t_1, \ldots, t_n) \to t$.

Démonstration. Pour chaque jugement, on considère les règles qui peuvent amener à cette conclusion.

- Références mémoire :
  - $\Gamma \vdash x : t$
    La seule règle de cette forme est LV-VAR. Puisque sa prémisse est vraie, on en conclut que $x : t \in \Gamma$.
  - $\Gamma \vdash *\varphi : t$
    Idem avec LV-DEREF.
  - $\Gamma \vdash \varphi[] : t$
    Idem avec LV-INDEX.
  - $\Gamma \vdash \varphi.l : t$
    Idem avec LV-FIELD.
- Appel de fonction : pour en arriver à $\Gamma \vdash e(e_1, \ldots, e_n) : t$, seule la règle CALL s'applique, ce qui permet de conclure.

- Fonction : la seule règle possible pour conclure une dérivation de

$$\Gamma \vdash \text{fun}(a_1, \ldots, a_n)\{i\} : t$$

est FUN.

□

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* $\Gamma \vdash v :$ INT*, alors* $\exists n, v = d$.

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

Ces lemmes étant établis, on énonce maintenant le théorème de progrès.

Théorème 5.1 (Progrès)**.** *Supposons que* $\Gamma \vdash e : t$. *Soit* m *un état mémoire tel que* $\Gamma \vdash_{mem} m$. *Alors l'un des cas suivant est vrai :*

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

C'est à dire, soit :

- *e est complètement évaluée*
- *un pas d'évaluation préservant la compatibilité mémoire est possible*
- *une erreur de division, tableau ou pointeur se produit*

Théorème 5.2 (Progrès pour les left-values)**.** *Si* $\langle l\, v, m\rangle \to^* \langle v, m'\rangle$, v *est de la forme* φ *(référence mémoire) ou* Ω *(erreur).*

La preuve des théorèmes 5.1 et 5.2 se trouve en annexe D.2.

Lemme 5.3 (Permutation)**.** *L'ordre dans lequel les variables apparaissent dans un environnement n'influe pas sur la relation de typage.*

Pour toute permutation σ *de* $[1; n]$, *on note* $\sigma(x_1 : t_1, \ldots, x_n : t_n) = x_{\sigma(1)} : t_{\sigma(1)}, \ldots x_{\sigma(n)} : t_{\sigma(n)}$.

Alors : si $\Gamma \vdash e : t$ *et* $\Gamma' = \sigma(\Gamma)$, *alors* $\Gamma' \vdash e : t$.

Lemme 5.4 (Affaiblissement)**.** *De même que l'ordre n'influe pas le typage, on peut aussi ajouter des associations supplémentaires dans l'environnement sans modifier les typages dans cet environnement.*

Si $\Gamma \vdash e : t$ *et* $x \notin \text{dom}(\Gamma)$, *alors* $\Gamma, x : t' \vdash e : t$.

Lemme 5.5 (Substitution)**.** *Si dans une expression e il apparait une variable x de type t', le typage est préservé lorsqu'on remplace ses occurrences par une expression e' de même type.*

Si $\Gamma, x : t' \vdash e : t$ *et* $\Gamma \vdash e' : t'$*, alors* $\Gamma \vdash e[x/e'] : t$.

Ces lemmes permettent de prouver le théorème suivant :

Théorème 5.3 (Préservation)**.** *Si une expression est typable, alors un pas d'évaluation ne modifie pas son type :*

Si $\Gamma \vdash e : t$ *et* $\langle e, m\rangle \rightarrow \langle e', m'\rangle$*, alors* $\Gamma \vdash e' : t$.

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

Cela prouve 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.

CHAPITRE 6

QUALIFICATEURS DE TYPE

A friend of mine in a compiler writing class produced a compiler with one error message "you lied to me when you told me this was a program".

— Pete Fenelon

Dans le chapitre 5, nous avons vu comment ajouter un système de types forts statiques à un langage impératif (défini dans le chapitre 4).

Ici, nous étendons l'expressivité de SAFESPEAK avec un système d'annotations de "souillure" (*tainting* en anglais). Un cas d'erreur est ajouté, lorsqu'on tente d'accéder à une valeur souillée. Avec cet ajout, la propriété de progrès (théorème 5.1) n'est donc plus valable.

Afin de retrouver cette adéquation entre la sémantique et le systme de typage, ce dernier est étendu d'un système de *qualificateurs de type* qui décrivent l'origine des données. Ils permettent de restreindre certaines opérations sensibles à des expressions dont la valeur est sûre.

La propriété de progrès est alors retrouvée (théorème 6.1).

6.1 Extensions noyau pour SAFESPEAK

Jusqu'ici SAFESPEAK, tel qu'il a été présenté dans le chapitre 4 est un langage de programmation impératif généraliste. Aucune construction en particulier n'est prévue pour implanter un système d'exploitation.

On ajoute donc la notion de valeur provenant de l'espace utilisateur (cf. chapitre 2) en trois étapes (figure 6.1) :

- tout d'abord, on ajoute une expression d'annotation sur les variables indiquant que celles-ci sont contrôlés par un utilisateur non privilégié, ainsi que des opérateurs de copie sûre.
- ensuite, on étend l'ensemble des valeurs possibles pour les pointeurs à une valeur TAINTED(φ) signifiant que l'objet pointé se situe en espace utilisateur
- enfin, on définit une nouvelle erreur Ω_{taint} produite par le déréférencement d'un pointeur ayant une telle valeur.

Pour adapter l'évaluation, plusieurs cas sont à rajouter. D'une part, la présence de TAINT($\cdot$) dans une instruction consiste à ajouter un TAINTED($\cdot$) dans la valeur construite. Ceci ne peut être fait que dans le cas où la valeur est un chemin φ, c'est-à-dire que la construction TAINT($\cdot$) ne peut se faire que sur une expression de type pointeur.

Expressions	$e ::= \ldots$	
	$\mid$ TAINT(e)	Expression souillée
	$\mid$ $l\,v \Leftarrow_U e$	Lecture depuis l'espace utilisateur
	$\mid$ $e \Rightarrow_U e$	Écriture dans l'espace utilisateur
Contextes	$C ::= \ldots$	
	$\mid$ TAINT(C)	
Chemins	$\varphi ::= \ldots$	
	$\mid$ TAINTED(φ)	Valeur souillée
Erreurs	$\Omega ::= \ldots$	
	$\mid$ Ω_{taint}	Erreur de souillure

FIGURE 6.1: Ajouts liés aux pointeurs utilisateurs

$$\frac{}{\langle \text{TAINT}(\varphi), m\rangle \rightarrow \langle \text{TAINTED}(\varphi), m\rangle} \text{ (EXPR-TAINTED)}$$

D'autre part, une règle accède à la mémoire : EXP-LV ; pour rappel :

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

Puisque la définition des chemins φ a été changée, il est aussi nécessaire de redéfinir la lentille Φ utilisée ci-dessus (définition 4.10).

On rajoute donc le cas :

$$\Phi(\text{TAINTED}(\varphi)) = \Omega_{taint}$$

Pour accéder à ces valeurs, il faut utiliser les opérateurs $\cdot \Leftarrow_U \cdot$ et $\cdot \Rightarrow_U \cdot$.

6.2 Insuffisance des types simples

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

$$\frac{\Gamma \vdash e : t*}{\Gamma \vdash \text{TAINT}(e) : t*} \text{ (TAINT-IGNORE)}$$

Cette règle est compatible avec l'extension, sauf qu'elle introduit des termes qui sont bien typables mais dont l'évaluation provoque une erreur autre que Ω_{div}, Ω_{array} ou Ω_{ptr}, violant ainsi le théorème 5.1.

Par exemple, supposons que x soit une variable globale entière, et posons $e = *\text{TAINT}(\&x)$.

e est alors bien typée sous $\Gamma = x : \text{INT}$:

Qualificateurs	$q ::=$ KERNEL	Donnée noyau (sûre)
	| USER	Donnée utilisateur (non sûre)
Types	$t ::= \ldots$	
	$\mid \not{t}*$	Pointeur
	$\mid t\ q\ *$	Pointeur qualifié

FIGURE 6.2: Changements liés aux qualificateurs de types

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

Posons alors $m = ([[x \mapsto 0]], [])$ (on a bien $\Gamma \vdash_{mem} m$). L'évaluation de e sous m provoque une erreur, comme le montre la dérivation suivante.

$$\dfrac{\dfrac{\overline{m[*\text{TAINTED}(x)] = \Omega_{taint}}}{\langle *\text{TAINT}(\&x), m\rangle \rightarrow \langle \Omega_{taint}, m\rangle}\ (\text{EXP-LV}) \qquad \dfrac{}{\langle \Omega_{taint}, m\rangle \rightarrow \Omega_{taint}}\ (\text{EVAL-ERR})}{\langle *\text{TAINT}(\&x), m\rangle \rightarrow \Omega_{taint}}$$

6.3 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 définir un nouveau jugement de typage $\cdot \vdash_q \cdot : \cdot$, à partir d'un ensemble de règles. Pour la plupart, les règles seront identiques, donc sauf mention contraire, les règles portant sur $\cdot \vdash \cdot : \cdot$ s'appliqueront aussi sur $\cdot \vdash_q \cdot : \cdot$. Naturellement, la plupart des différences porteront sur le traitement des pointeurs.

La différence principale est qu'à chaque pointeur, on ajoute un *qualificateur* qui représente *qui* contrôle sa valeur (section 2.2). Les deux qualificateurs sont :

- KERNEL : il s'applique aux pointeurs contrôlés par le noyau. Par exemple, prendre l'adresse d'un objet de la pile noyau donne un pointeur noyau.
- USER : il s'applique aux pointeurs qui proviennent de l'espace utilisateur. Ces pointeurs proviennent toujours d'interfaces particulières, comme les appels système ou les paramètres de la fonction `ioctl`.

Cet ajout est précisé dans la figure 6.2.

Au niveau du système de types, la principale restriction est que seuls les pointeurs KERNEL peuvent être déréférencés de manière sûre :

$$\dfrac{\Gamma \vdash_q e : \tau\ \text{KERNEL}*}{\Gamma \vdash_q *e : \tau}\ (\text{LV-DEREF-KERNEL})$$

L'opérateur TAINT(·) transforme un pointeur selon la règle de souillure suivante :

$$\frac{\Gamma \vdash_q e : t\ q\ *}{\Gamma \vdash_q \text{TAINT}(x) : t\ \text{USER}\ *}\ (\text{TAINT})$$

Les opérateurs $\cdot \Leftarrow_U \cdot$ et $\cdot \Rightarrow_U \cdot$ sont typés de la manière suivante :

$$\frac{\Gamma \vdash lv : t \qquad \Gamma \vdash e : t\ \text{USER}\ *}{\Gamma \vdash lv \Leftarrow_U e : \text{INT}}\ (\text{GETU}) \qquad \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t\ \text{USER}\ *}{\Gamma \vdash e_1 \Rightarrow_U e_2 : \text{INT}}\ (\text{PUTU})$$

La définition du typage sémantique doit aussi être étendue au cas $\varphi = \text{TAINTED}(\varphi')$. Les références mémoires sont "nettoyées" pour accéder à la left-value encapsulée.

$$\frac{m \vDash \varphi' : t\ \text{KERNEL}\ *}{m \vDash \text{TAINTED}(\varphi') : t\ \text{USER}\ *}\ (\text{S-TAINTED})$$

6.3.1 Propriété d'isolation mémoire

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.

Théorème 6.1 (Progrès pour les types qualifiés)**.** *Supposons que* $\Gamma \vdash_q e : t$*. Soit* m *un état mémoire tel que* $\Gamma \vdash_{mem} m$*. Alors l'un des cas suivant est vrai :*

- $\exists v \neq \Omega, e = v$
- $\exists (e', m'), \Gamma \vdash_{mem} m' \wedge \langle e, m\rangle \rightarrow \langle e', m'\rangle$
- $\exists \Omega \in \{\Omega_{div}, \Omega_{array}, \Omega_{ptr}\}, \langle e, m\rangle \rightarrow \Omega$

Et nous donnons un équivalent du théorème 5.3.

Théorème 6.2 (Préservation pour les types qualifiés)**.** *Si une expression est typable et que son évaluation produit une valeur, alors cette valeur est du même type que l'expression.*
Si $\Gamma \vdash_q e : t$ *et* $e \rightarrow v$
alors $\Gamma \vdash_q v : t$*.*

Troisième partie

Expérimentation

On décrit ici la démarche expérimentale liée à l'implémentation des analyses décrites dans la partie II.

Le chapitre 7 décrit l'implémentation en elle-même : comment le code source C est compilé vers SAFESPEAK, et comment les types du programme sont vérifiés.

Ensuite, dans le chapitre 8, le cas d'un bogue de pilote graphique dans le noyau Linux est étudié. On montre que les analyses précedentes permettent de distinguer statiquement entre le cas incorrect et le cas corrigé.

Enfin, le chapitre 9 conclut : les limitations de cette approche sont présentées, ainsi qu'un résumé des contributions de cet ouvrage.

CHAPITRE 7

IMPLANTATION

Dans ce chapitre, nous décrivons la mise en œuvre des analyses statiques précédentes. Nous commençons par un tour d'horizon des représentations intermédiaires possibles, avant de décrire celle retenue : Newspeak. La chaîne de compilation est explicitée, partant de C pour aller au langage impératif décrit dans le chapitre 4. Enfin, nous donnons les détails d'un algorithme d'inférence de types à la Hindley-Milner, reposant sur l'unification et le partage de références.

7.1 Newspeak

Newspeak [HL08] est un langage conçu pour être à la fois :

- Précis : sa sémantique est définie formellement dans [HL08]
- Expressif : la plupart des primitives présentes dans les langages de bas niveau sont compilables en Newspeak.
- Simple : peu de primitives sont présentes.
- Minimal : aucun élément syntaxique ne peut être exprimé comme combinaison d'autres.
- Explicite : les constructions sont toutes indépendantes du contexte.
- Orienté analyse : les primitives sont décorées d'informations reflétant leur validité (tests de bornes, etc)
- Indépendant de l'architecture : toutes les caractéristiques comme la taille des types ou l'alignement des structures sont rendues explicites.

7.2 Chaîne de compilation

La compilation vers C est faite en trois étapes (figure 7.1) : prétraitement du code source, compilation de C prétraité vers NEWSPEAK, puis compilation de NEWSPEAK vers ce langage.

Trad

La première étape consiste à prétraiter les fichiers C source avec le logiciel `cpp`, comme pour une compilation normale. Cette étape interprète les directives comme `#include`, `#ifdef`, À cet étape, les commentaires sont aussi supprimés.

Une fois cette passe effectuée, nous avons des fichiers C prétraités, c'est à dire des unités de compilation autocontenues.

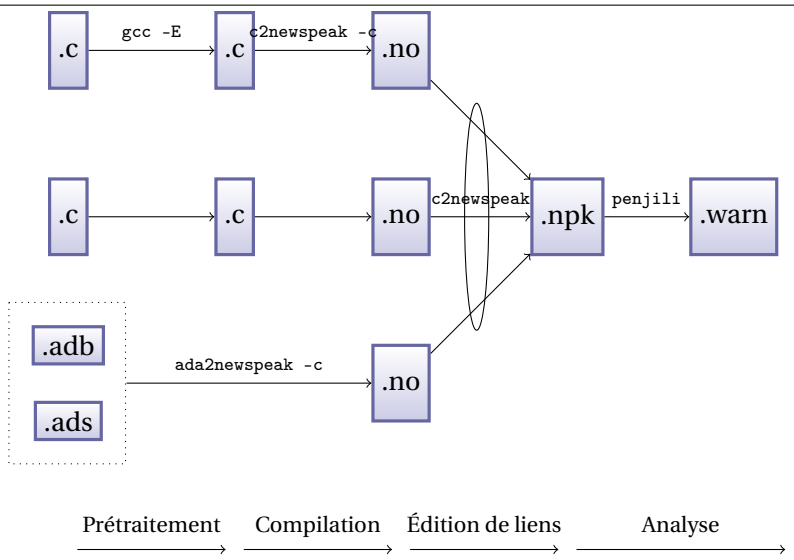


FIGURE 7.1: Compilation depuis Newspeak

Puisque la directive `#include` est textuelle, ces fichiers sont très grands et donnent lieu à beaucoup de duplication dans les passes suivantes.

À ce niveau, les fichiers sont passés à l'outil `c2newspeak` qui les traduit vers Newspeak. Dans cette étape, les types et les noms sont résolus, et le programme est annoté de manière à rendre les prochaines étapes indépendante du contexte. Par exemple, chaque déclaration de variable est adjointe d'une description complète du type.

Lors de cette étape, le flôt de contrôle est également simplifié. C en effet propose de nombreuses constructions ambigües ou redondantes.

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 sémantique, ou de code lors de l'implémentation 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).

La sémantique de Newspeak et la traduction de C vers Newspeak sont décrites dans [HL08].

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 un l'élément d'un tableau (tous deux sont traduits par un décalage numérique depuis le début de la zone mémoire). Pour supprimer cette ambiguïté, il faut s'interfacer dans les structures internes de `c2newspeak`, où les informations nécessaires sont encore présentes.

Ensuite, les différents fichiers sont liés ensemble. Cet 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 unique.

Enfin, 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 suffisament proches du lambda calcul simplement typé, on peut utiliser une variante de l'algorithme W de Damas et Milner [DM82]. 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.

7.2.1 Prétraitement

C2NEWSPEAK travaillant uniquement sur du code prétraité (dans directives de préprocesseur), la première étape consiste donc à faire passer le code par CPP : les macros sont développées, les constantes remplacées par leurs valeurs, les commentaires supprimés, les fichiers d'en-tête inclus, etc.

7.2.2 Compilation (levée des ambigüités)

Cette passe est réalisée par l'utilitaire C2NEWSPEAK. L'essentiel de la compilation consiste à mettre à plat les définition de types, et

7.2.3 Annotations

NEWSPEAK a de nombreux avantages, mais pour une analyse par typage il est trop bas niveau. Par exemple, dans le code suivant

```
struct s {
    int a;
    int b;
};

int main(void)
{
    struct s x;
    int y[10];
    x.b = 1;
    y[1] = 1;
    return 0;
}
```

(wip)

7.2.4 Implantation de l'algorithme de typage

On prend l'exemple d'un lambda-calcul simplement typé avec entiers, flottants et couples (figure 7.2).

Prenons l'exemple de la fonction suivante [1] :

$$f = \lambda x.\lambda y.\text{plus}(\text{plus}(\text{fst}\,x)(\text{snd}\,x))\,y$$

Soit en syntaxe ML :

```
let f x y =
  plus
    (plus
      (fst x)
```

1. On suppose que `plus` est une fonction de l'environnement global qui a pour type INT → INT → INT.

Expressions

$e ::= n$		Entier
	d	Flottant
	x	Variable
	$\lambda x.e$	Abstraction
	$e_1\ e_2$	Application
	(e_1, e_2)	Couple
	fst e	Projection gauche
	snd e	Projection droite

Types

$t ::= \text{INT}$		Entier
	FLOAT	Flottant
	$t \to t$	Fonction
	$t \times t$	Produit

Contextes

$\Gamma ::= \varepsilon$		Contexte vide
	$\Gamma, x : t$	Extension

$$\boxed{\Gamma \vdash e : t}$$

$$\frac{}{\Gamma \vdash n : \text{INT}}\ (\text{INT}) \qquad \frac{}{\Gamma \vdash d : \text{FLOAT}}\ (\text{FLOAT}) \qquad \frac{(x : t) \in \Gamma}{\Gamma \vdash x : t}\ (\text{VAR}) \qquad \frac{\Gamma, (x : t_1) \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2}\ (\text{ABS})$$

$$\frac{\Gamma \vdash e : t_1 \to t_2 \qquad \Gamma \vdash e' : t_1}{\Gamma \vdash e\ e' : t_2}\ (\text{APP}) \qquad \frac{\Gamma \vdash x : t_1 \times t_2}{\Gamma \vdash \text{fst}\ x : t_1}\ (\text{PROJ-G}) \qquad \frac{\Gamma \vdash x : t_1 \times t_2}{\Gamma \vdash \text{snd}\ x : t_2}\ (\text{PROJ-D})$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (\text{TUP})$$

FIGURE 7.2: Lambda calcul simplement typé avec entiers, flottants et couples

```
  (snd x)
)
y
```

Puisque `fst` et `snd` sont appliqués à `x`, ce doit être un tuple. En outre on additionne ces deux composantes ensemble, donc elles doivent être de type INT (et le résultat aussi). Par le même argument, `y` doit aussi être de type INT. En conclusion, `x` est de type INT × INT et `y` de type INT, donc f est de type INT × INT → INT → INT.

Pour implanter cette analyse, on peut remarquer qu'étant donné la forme d'un terme, on peut savoir quelle règle de typage a été utilisée en dernier. Il est ainsi possible de "remonter" l'arbre d'inférence afin de savoir quelles règles ont été employées (figure 7.3) [2]. Pour le moment, on ne connait pas les types.

2. Par souci de clarté, les prémisses des applications de (VAR) ne sont pas notées.

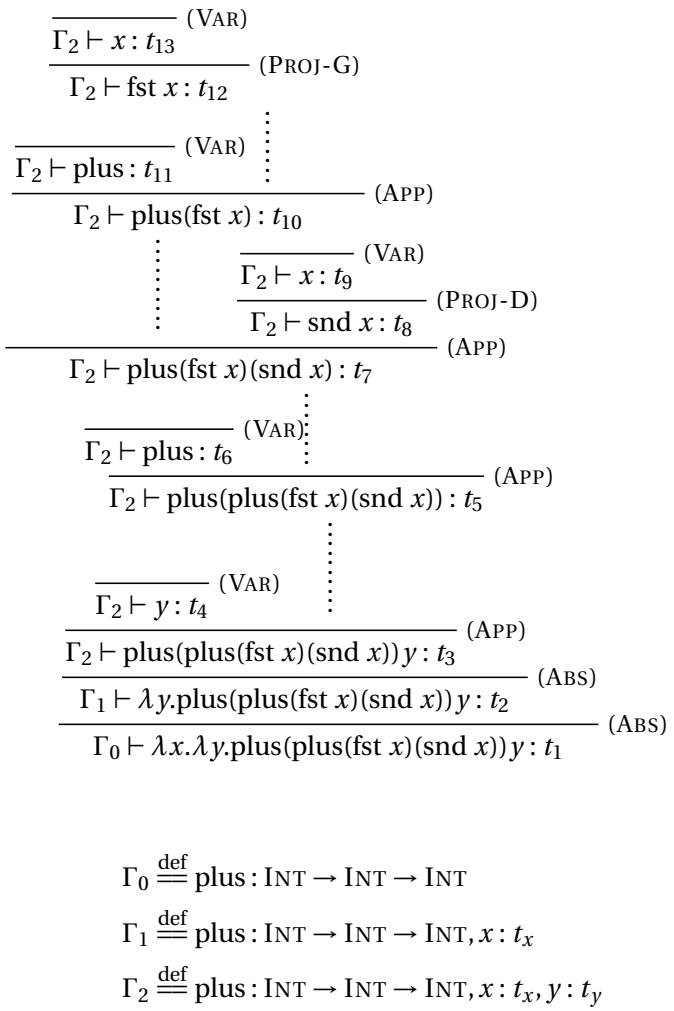


FIGURE 7.3: Arbre d'inférence : règles à utiliser

Une fois à cette étape, on peut donner un nom à chaque type inconnu : $t_1, t_2, \ldots$. L'utilisation qui en est faite permet de générer un ensemble de contraintes d'unification (figure 7.4).

Le symbole ? qui apparait dans les règles correspond à "n'importe quel type", dans le sens où l'unification avec ce symbole est toujours possible.

On en déduit donc un ensemble de contraintes d'égalité (figure 7.5(a)). La prochaine étape est de résoudre ce système. D'après (App1) et (App2), $t_6 = t_7 \to t_4 \to t_3$ d'où par (Var2), $t_7 \to t_4 \to t_3 = \text{INT} \to \text{INT} \to \text{INT}$. Le constructeur $\to$ étant injectif, on a donc $t_3 = t_4 = t_7 = \text{INT}$. En procédant de même avec (App3) et (App4), on obtient $t_8 = t_{12} = \text{INT}$. D'après (Proj-G1), (Var5), (Var3), et (Proj-D1) on a $t_{12} \times ? = t_{13} = t_x = t_9 = ? \times t_8$, donc $t_x = t_{12} \times t_8 = \text{INT} \times \text{INT}$. En remplaçant, on obtient tous les autres types (figures 7.6(b)) et donc un arbre de typage complet (figure 7.8).

Pour effectuer une résolution de manière automatique, il faut une technique d'unification. La manière la plus classique de procéder est due à Robinson [Rob65] et se déroule par substitution.

$\dfrac{(x, t_x) \in \Gamma}{\Gamma \vdash x : t_1}$ (VAR)	$t_1 = t_x$	$\dfrac{\Gamma, (x : t_x) \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1}$ (ABS)	$t_1 = t_x \to t_2$
$\dfrac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ e_2 : t_3}$ (APP)	$t_1 = t_2 \to t_3$	$\dfrac{\Gamma \vdash e : t_2}{\Gamma \vdash \text{fst}\ e : t_1}$ (PROJ-G)	$t_2 = t_1 \times ?$
$\dfrac{\Gamma \vdash e : t_2}{\Gamma \vdash \text{snd}\ e : t_1}$ (PROJ-D)	$t_2 = ? \times t_1$	$\dfrac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_3}$ (TUP)	$t_3 = t_1 \times t_2$

FIGURE 7.4: Contraintes crées par les applications de règles

		$t_1 = \text{INT} \times \text{INT} \to \text{INT}$
$t_1 = t_x \to t_2$	(Abs1)	$t_2 = \text{INT} \to \text{INT}$
$t_2 = t_y \to t_3$	(Abs2)	$t_3 = \text{INT}$
$t_5 = t_4 \to t_3$	(App1)	$t_4 = \text{INT}$
$t_4 = t_y$	(Var1)	$t_5 = \text{INT} \to \text{INT}$
$t_6 = t_7 \to t_5$	(App2)	$t_6 = \text{INT} \to \text{INT} \to \text{INT}$
$t_6 = \text{INT} \to \text{INT} \to \text{INT}$	(Var2)	$t_7 = \text{INT}$
$t_{10} = t_8 \to t_7$	(App3)	$t_8 = \text{INT}$
$t_9 = ? \times t_8$	(Proj-D1)	$t_9 = \text{INT} \times \text{INT}$
$t_9 = t_x$	(Var3)	$t_{10} = \text{INT} \to \text{INT}$
$t_{11} = t_{12} \to t_{10}$	(App4)	$t_{11} = \text{INT} \to \text{INT} \to \text{INT}$
$t_{11} = \text{INT} \to \text{INT} \to \text{INT}$	(Var4)	$t_{12} = \text{INT}$
$t_{13} = t_{12} \times ?$	(Proj-G1)	$t_{13} = \text{INT} \times \text{INT}$
$t_{13} = t_x$	(Var5)	$t_x = \text{INT} \times \text{INT}$
(a) Contraintes		$t_y = \text{INT}$
		(b) Solution

FIGURE 7.7: Contraintes d'égalité et solution obtenues à partir de la figure 7.3

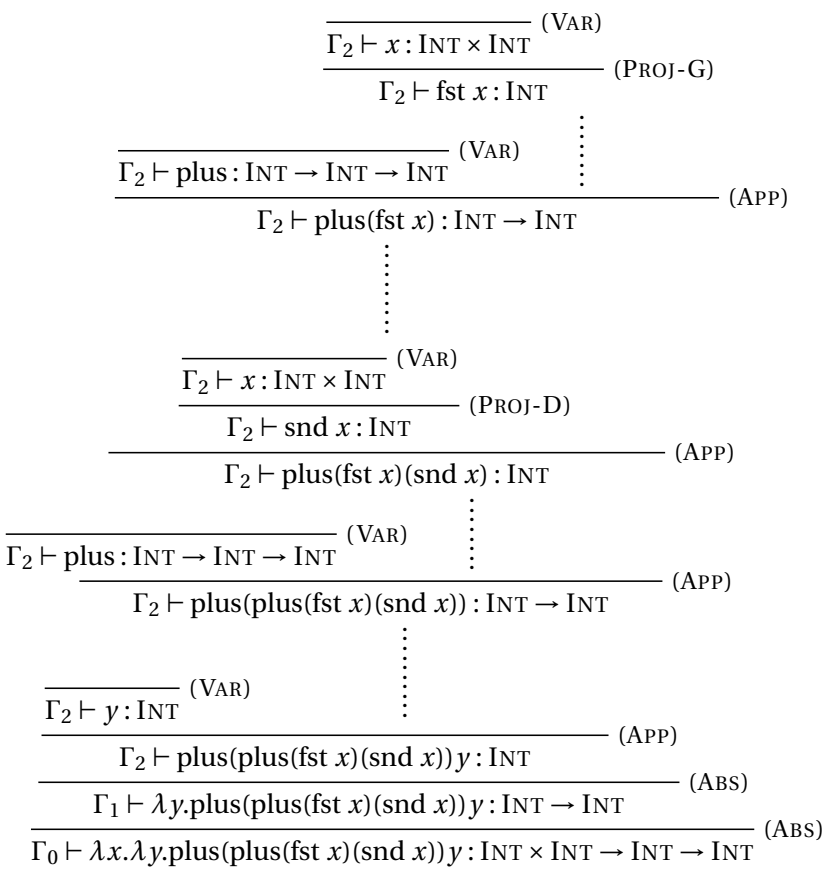


FIGURE 7.8: Arbre d'inférence complet

Pour résoudre ces contraintes, on commence par les simplifier : si $t_a \to t_b = t_c \to t_d$, alors $t_a = t_c$ et $t_b = t_d$. De même si $t_a \times t_b = t_c \times t_d$. Au contraire, si $t_a \to t_b = t_c \times t_d$, il est impossible d'unifier les types et il faut abandonner l'inférence de types. D'autre cas sont impossibles, par exemple $\text{INT} = t_1 \to t_2$ ou $\text{INT} = \text{FLOAT}$.

Une fois ces simplifications réalisées, les contraintes restantes sont d'une des formes suivantes :

- $t_i = t_i$. Il n'y a rien à faire, cette contrainte peut être supprimée.
- $t_i = t_j$ avec $i \neq j$: toutes les occurrences de t_j dans les autres contraintes peuvent être remplacées par t_i.
- $t_i = x$ (ou $x = t_i$) où x est un type concret : idem.

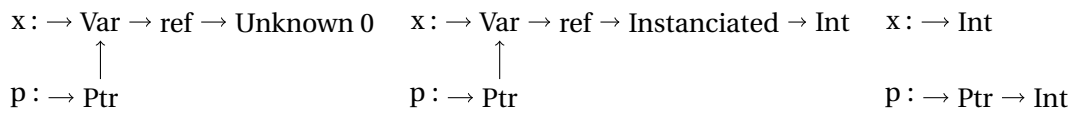


FIGURE 7.9: Unification par partage

```
int x;
int *p = &x;
x = 0;
```

FIGURE 7.10: Compilation d'un programme C – avant

Plutôt que de modifier toutes les occurrences d'un type t_i, on va affecter à t_i la valeur du nouveau type.

L'implémentation de cet algorithme utilise le partage et les références (figure 7.9).

D'abord 7.9(a), ensuite 7.9(b), et enfin 7.9(c).

Prenons l'exemple de la figure 7.10 et typons-le "à la main". On commence par oublier toutes les étiquettes de type présentes dans le programme. Celui-ci devient alors :

```
var x, p;
p = &x;
x = 0;
```

La premiere ligne introduit deux variables. On peut noter leurs types respectifs (inconnus pour le moment) t_1 et t_2. La première affectation `p = &x` implique que les deux côtés du signe "=" ont le même type. À gauche, le type est t_2, et à droite $Ptr(t_1)$. On applique le même raisonnement à la seconde affectation : à gauche, le type est t_1 et à droite Int. On en déduit que le type de x est Int et celui de p est Ptr(Int).

```
type var_type =
  | Unknown of int
  | Instanciated of ml_type

and const_type =
  | Int_type
  | Float_type

and ml_type =
  | Var_type of var_type ref
  | Const_type of const_type
  | Pair_type of ml_type * ml_type
  | Fun_type of ml_type * ml_type
```

Pour implanter cet algorithme, on représente les types de données du programmes à typer par une valeur de type `ml_type`. En plus des constantes de types comme int ou float, et des constructeurs de type comme pair et fun, le constructeur Var permet d'exprimer les variables de types (inconnues ou non).

Celles-ci sont numérotées par un int, on suppose avoir à disposition deux fonctions manipulant un compteur global d'inconnues.

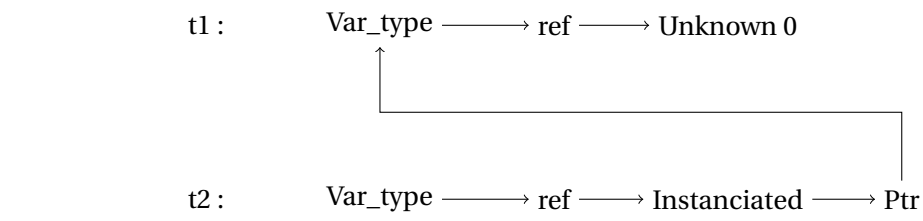


FIGURE 7.11: Unification : partage

```
module Counter : sig
  val reset_unknowns : unit -> unit
  val new_unknown : unit -> int
end
```

De plus, on a un module gérant les environnements de typage. Il pourra être implanté avec des listes d'association ou des tables de hachage, par exemple. Sa signature est :

```
module Env : sig
  type t

  (* Construction *)
  val empty : t
  val extend : ml_ident -> ml_type -> t -> t

  (* Interrogation *)
  val get : ml_ident -> t -> ml_type option
end
```

Reprenons l'exemple précédent. Partant d'un environnement vide (Env.empty), on commence par l'étendre de deux variables. Comme on n'a aucune information, il fait allouer des nouveaux noms d'inconnues (qui correspondent à t_1 et t_2) :

```
let t1 = Var_type (Unknown (new_unknown ())) in
let t2 = Var_type (Unknown (new_unknown ())) in
let env =
  Env.extend "p" t2
    (Env.extend "x" t1
      Env.empty
    ) in
```

La première instruction indique que les deux côtés de l'affectation doivent avoir le même type.

```
let lhs1 = Lv_var "p"
and rhs1 = AddrOf (Exp_var "x") in
let t_lhs1 = typeof lhs1 env
and t_rhs1 = typeof rhs1 env in
unify t_lhs1 t_rhs1;
```

Ici il se passe plusieurs choses intéréssantes. D'une part nous faisont appel à une fonction externe typeof qui retourne le type d'une expression sous un environnement (dans une

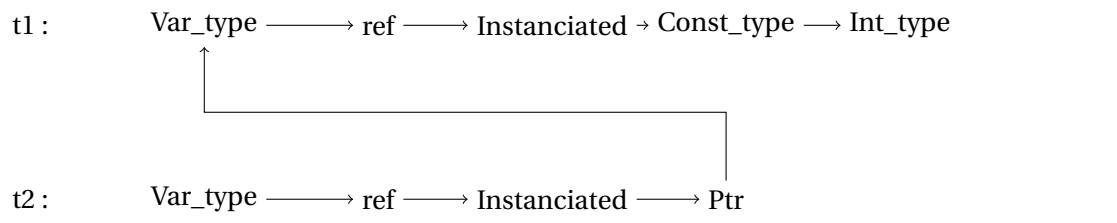


FIGURE 7.12: Unification par mutation de références

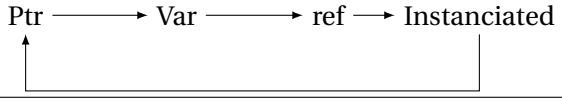


FIGURE 7.13: Cycle dans le graphe de types

implantation complète il s'agirait d'un appel récursif). Dans ce cas, `typeof lhs1 env` est identique à `Env.get lhs1 env` et `typeof rhs1 env` à `Ptr_type t1`. L'autre aspect intéressant est la dernière ligne : la fonction `unify` va modifier en place les représentations des types afin de les rendre égales. L'implantation de `unify` sera décrite plus tard. Dans ce cas précis, elle va faire pointer la référence dans t2 vers t1 (figure 7.11).

Enfin, la seconde affectation se déroule à peu près de la même manière.

```
let lhs2 = Lv_deref (Lv_var "p")
and rhs2 = Exp_int 0 in
let t_lhs2 = typeof lhs2 env
and t_rhs2 = typeof rhs2 env in
unify t_lhs2 t_rhs2;
```

Ici `typeof lhs2 env` est identique à `Ptr_type (Env.get "p" env)` et `typeof lhs2 env` à `Const_type Int_type`. Et dans cas, l'unification doit se faire entre t1 et `Const_type Int_type` : cela mute la référence derrière t1 (figure 7.12).

L'essence de l'algorithme d'inférence se situe donc dans 2 fonctions. D'une part, `unify` qui réalise l'unification des types grâce à au partage des références. D'autre part, la `typeof` qui encode les règles de typage elles-mêmes et les applique à l'aide de `unify`.

7.2.5 Algorithme d'unification

Voici une implantation de la fonction `unify`.

Celle-ci prend en entrée deux types t_1 et t_2. À l'issue de l'exécution de `unify`, ces deux types doivent pouvoir être considérés comme égaux. Si ce n'est pas possible, une erreur sera levée.

La première étape est de réduire ces deux types, c'est à dire à transformer les constructions `Var (ref (Instanciated t))` en `t`.

Ensuite, cela dépend des formes qu'ont les types réduits :

- si les deux types sont inconnus (de la forme `Var (ref (Instanciated t))`), on fait pointer l'une des deux références vers le premier type. Notons que cela crée un type de la forme `Var (ref (Instanciated (Var (ref (Unknown n)))))` qui sera réduit lors d'une prochaine étape d'unification.
- si un type est inconnu et pas l'autre, il faut de la même manière affecter la référence. Mais en faisant ça inconditionnellement, cela peut poser problème : par exemple en tentant d'unifier a avec `Ptr(a)` on pourrait créer un cycle dans le graphe (figure 7.13).

Pour éviter cette situation, il suffit de s'assurer que le type inconnu n'est pas présent dans le type à affecter.

- 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.
- TODO sous typage pour les structures

TODO :

- implem du polymorphisme
- implem du sous-typage
- généralisation depuis le toy language

```
Decl
  ( "x"
  , Newspeak.Scalar (Newspeak.Int (Newspeak.Signed, 32))
  , ()
  , [ Decl
        ( "p"
        , Newspeak.Scalar Newspeak.Ptr
        , ()
        , [ Set
              ( Local "p"
              , ( AddrOf (Local "x")
                , ()
                )
              , Newspeak.Scalar Newspeak.Ptr
              )
          ; Set
              ( Local "x"
              , ( Const (CInt Nat.zero)
                , ()
                )
              , Newspeak.Scalar (Newspeak.Int (Newspeak.Signed, 32))
              )
          ]
        )
    ]
  )
```

FIGURE 7.14: Compilation d'un programme C – après

Le programme C (figure 7.10) est compilé ainsi en Tyspeak (figure 7.14).

```
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 7.15: Implantation – fonction principale de `ptrtype`

7.3 Architecture de `ptrtype`

L'outil `ptrtype` lit un programme Newspeak (ou un fichier C), et réalise l'inférence de qualificateurs. En sortie, il affiche soit le programme complètement annoté, soit une erreur. Le cœur de l'outil est dans la

Si l'argument 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 (polymorphe en le type des étiquettes) est `'a Tyspeak.t`.

Le reste de l'outil est résumé dans la fonction `process_npk` (figure 7.15) :

- 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 $\preceq$ 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.
- Les annotations extérieures sont alors lues (variable `exttbl`), ce qui permet de créer un environnement initial.
- Les types de chaque fonction sont inférés, par le biais de la fonction suivante :

```
val infer : Newspeak.fid list
         -> Types.simple Env.t
```

```
          -> 'a Tyspeak.t
          -> Types.simple Tyspeak.t
```

- Ces types sont vérifiés. Cette étape est normalement redondante et est donc optionnelle (l'option `--no-check` permet de la désactiver).
- Le programme obtenu, de type `Types.simple Tyspeak.t`, est affiché sur le terminal.

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 : on peut créer de nouveaux types avec `new_unknown` et unifier deux types avec `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.

Les règles de typage sont implantées par `new_unknown` et `unify`. Par exemple, pour typer une déclaration (figure 7.16), 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.

De même, pour typer un appel de fonction, on infère le type de ses arguments et left-values 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 exemples, comme :

- addition de deux flottant (dans `infer_binop`) :

```
let infer_binop op (_, a) (_, b) =
  match op with
    (* [...] *)
    | N.PlusF _ ->
        unify a Float;
        unify b Float;
        Float
```

- adresse d'une left-value (dans `lval_type`) :

```
| T.AddrOf lv ->
    let lv' = infer_lv env lv in
    let ty = lval_type env lv in
    (T.AddrOf lv', Ptr (Kernel, ty))
```

- déréférencement d'une left-value (dans `lval_type`) :

```
| T.Deref(e, _sz) ->
    let (_, te) = infer_exp env e in
    let t = new_unknown () in
    unify (Ptr (Kernel, t)) te;
    t
```

```
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 7.16: Implantation – inférence des déclarations de variable et appels de fonction

7.5 Vérification de types

La passe de vérification de types, optionnelle et en théorie redondante, est présente pour s'assurer que les types inférés ne sont pas aberrants.

Là encore, on travaille par effet de bord : si tout ne se passe pas bien, on lève une erreur. La fonction principale a le type suivant :

```
val check : Types.simple Env.t -> Types.simple Tyspeak.t -> unit
```

7.6 Unification

Les types de données utilisés sont donnés dans la figure 7.17. Les types SAFESPEAK sont représentés soit par un constructeur de type "résolu" immutable comme `Int`, soit par une référence dans le cas d'une variable inconnue (placée alors derrière le constructeur `Var`).

```
type unknown = { id : int }

type 'a var_type =
  | Unknown of unknown
  | Instanciated of 'a

and qual =
  | Kernel
  | User
  | QVar of qual var_type ref

and simple =
  | Int
  | Float
  | Fun of simple list * simple list
  | Ptr of qual * simple
  | Array of simple
  | Struct of (int * simple) list ref
  | Var of simple var_type ref
```

FIGURE 7.17: Implantation – types

```
let rec shorten = function
  | Var ({contents = Instanciated (Var _ as t)} as vt) ->
      let t2 = shorten t in
      vt := Instanciated t;
      t2
  | Var {contents = Instanciated t} -> t
  | t -> t
```

FIGURE 7.18: Implantation – fonction de racourcissement des représentations de types

Ces références contiennent une valeur du type `simple var_type`[3], c'est à dire :

- soit un numéro d'inconnue (constructeur `Unknown`)[4].
- soit un type résolu (constructeur `Instanciated`) si cette inconnue a été unifiée avec un type concret.

Ce système peut créer des représentations de types arbitrairement longues, comme par exemple :

`Var (ref (Instanciated (Var (ref (Instanciated Int)))))`

Cela est dû au fait que `fun x -> Var (ref (Instanciated x))` est typée `simple -> simple` et peut donc être appliquée à loisir. Pour éviter cet effet d'allongement, on définit une fonction `shorten` qui supprime ces chaînes (figure 7.18).

La fonction d'unification, quant à elle, commence par raccourcir ses deux arguments puis faire une analyse de cas par filtrage.

Les cas principaux pour unifier deux types réduits `sta` et `stb` sont :

3. Ce type est polymorphe pour pouvoir être repris dans l'unification des qualificateurs (figure 7.21).

4. On place le numéro dans un enregistrement pour abstraire l'entier sous-jacent, en empêchant par exemple de faire de l'arithmétique sur celui-ci.

- `sta` et `stb` sont deux inconnues. Alors on modifie l'un pour pointer sur l'autre. Les références étant uniques et partagées, cela revient à substituer l'un par l'autre dans toutes les représentations de types.
- Un type est concret, l'autre une inconnue. Dans ce cas on modifie le second comme étant un `Instanciated` du premier. Il faut vérifier que le type inconnu remplacé n'apparaît pas dans le type concret, sinon on créé un cycle. Ce cas a lieu par exemple quand on cherche à unifier a avec a KERNEL $*$. Il faut alors signaler une erreur, ce que fait la fonction `occurs_check_failed`.
- Les deux types sont des types concrets. Alors ils sont de la forme respective $C(t_1,\ldots,t_n)$ et $D(u_1,\ldots,u_m)$ où C et D sont des constructeurs de type avec respectivement n et m arguments [5]. Si $C = D$ et $n = m$, alors on unifie récursivementchaque t_i avec u_i. Sinon on lève une erreur (fonction `type_clash`).

Cette analyse de cas est implantée dans la fonction `unify_now` (figure 7.19).
cas structure (figure 7.20).

Pour les pointeurs, il est nécessaire de définir des fonctions similaires sur les qualificateurs (figure 7.21).

La fonction appellée directement par le reste du code, appellée `unify`, peut retarder l'unification (figure 7.22). 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".

7.7 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
f : (Ptr (Int)) -> (Int)
Int (example.c:1#4)^f(Ptr (Int) x) {
  (.c:3#4)^!return =(int32)
    (coerce[-2147483648,2147483647]
      ( ( ([(x_Ptr (Int) : Ptr (Int))]32_Int
            : Int
          )
          + (1 : Int)
        ) : Int
      ) : Int
    );
}
```

L'opérateur `coerce[a,b]` est un artefact de Newspeak, destiné 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 un "plus" unaire typé (INT) → INT.

A contrario, lorsqu'il n'est pas possible d'inférer des types compatibles, l'analyseur s'arrête avec une erreur.

5. Le cas $n = 0$ ou $m = 0$ correspond aux types concrets comme INT ou FLOAT.

```
let rec unify_now ta tb =
  let sta = shorten ta in
  let stb = shorten tb in
  match (sta, stb) with
  | ((Var ({contents = Unknown na} as ra)),
     (Var  {contents = Unknown nb})) ->
       begin
         if na <> nb then
           ra := Instanciated stb
       end
  | ((Var ({contents = Unknown {id = n}} as r)), t)
    ->
      begin
        if occurs n t then
          occurs_check_failed sta stb
        else
          r := Instanciated t
      end
  | (_, (Var ({contents = Unknown _}))) -> unify_now stb sta

  | Int, Int
  | Float, Float -> ()

  | Ptr (qa, ta), Ptr (qb, tb) ->
      unify_quals qa qb;
      unify_now ta tb

  | Array a, Array b -> unify_now a b

  | Fun (args_a, rets_a), Fun (args_b, rets_b) ->
      List.iter2 unify_now args_a args_b;
      List.iter2 unify_now rets_a rets_b

  | Struct rfa, Struct rfb ->
      unify_structs rfa rfb

  | _ -> type_clash sta stb
```

FIGURE 7.19: Implantation – fonction d'unification

```
let unify_structs rfa rfb =
  let fa = !rfa in
  let fb = !rfb in

  let new_a = ref [] in
  let new_b = ref [] in

  let unify_fields = function
    | _, InBoth (ta, tb) -> unify_now ta tb
    | k, OnlyL f -> new_b := (k,f) :: !new_b
    | k, OnlyR f -> new_a := (k,f) :: !new_a
  in

  List.iter unify_fields (compare_lists fa fb);
  let by_offset (x, _) (y, _) =
    compare x y
  in
  rfa := List.sort by_offset (!new_a @ !rfa);
  rfb := List.sort by_offset (!new_b @ !rfb)
```

FIGURE 7.20: Implantation – structures

```
void f(int *p) {
    /*!npk userptr p */
    *p = 3;
}
```

Le commentaire `/*!npk userptr p */` est interpréter par l'analyseur et le fait unifier le type de p avec *t* USER *, c'est à dire qu'il force son qualificateur à être USER.

```
04-addrof.c:4#4 - Cannot unify qualifiers:
  Kernel
  User
```

On signale l'emplacement où la dernière unification a échoué.

```
let rec shorten_q = function
  | QVar ({contents = Instanciated (QVar _ as t)} as vt) ->
      let t2 = shorten_q t in
      vt := Instanciated t;
      t2
  | QVar {contents = Instanciated t} -> t
  | t -> t

let rec unify_quals a b =
  let sa = shorten_q a in
  let sb = shorten_q b in
  match (sa, sb) with
  | User, User -> ()
  | Kernel, Kernel -> ()
  | ((QVar ({contents = Unknown na} as ra)),
     (QVar  {contents = Unknown nb})) ->
       begin
         if na <> nb then
           ra := Instanciated sb
       end
  | ((QVar ({contents = Unknown {id = n}} as r)), q)
    ->
      begin
        if occurs_q n q then
          occurs_q_check_failed sa sb
        else
          r := Instanciated q
      end
  | (_, (QVar ({contents = Unknown _}))) -> unify_quals sb sa
  | _ -> Utils.error "Cannot unify qualifiers:\n  %s\n  %s\n"
                       (string_of_qual sa)
                       (string_of_qual sb)
```

FIGURE 7.21: Implantation – qualificateurs

```
let unify a b =
  if !Options.lazy_unification then
    Queue.add (Unify (a, b)) unify_queue
  else
    unify_now a b
```

FIGURE 7.22: Implantation – unification directe ou retardée

CHAPITRE 8

ÉTUDE DE CAS : UN PILOTE DE CARTE GRAPHIQUE

Dans ce chapitre, un exemple de mise en œuvre du système de type décrit dans le chapitre 6 et implanté dans le chapitre 7.

8.1 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 au fil des années était porté sur de nombreuses architectures et s'est enrichi de nombreux pilotes de périphériques. En 20XX, son code source comporte 8.8 millions de lignes de code (en grande majorité du C) dont 55% 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.

8.2 GNU C

Linux est écrit dans le langage C, mais pas sa version normalisée. Il utilise le dialecte GNU C qui est celui que supporte GCC.

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. De nouvelles fonctionnalités sont aussi présentes.

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 `i`.

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.

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.

8.3 Configuration

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.

8.4 Appels systèmes 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, [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 2.1.3 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 systèmes, 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 un ensemble de tables de pages et une valeur de CR3associée. Au moment de changer le processus en cours, l'ordonnanceur charge donc le CR3du 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 programmes (variables globales)
- la pile utilisateur
- le tas (mémoire allouée par `malloc` et fonctions similaires)
- 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.

8.4.1 Appels système

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 (c'était le but !). Pour qu'ils 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.

Voyons ce qui se passe derrière la magie apparente. Une explication plus détaillée est disponible dans la documentation fournie par Intel [Int10].

Dans la bibliothèque C

Il y a bien une fonction `getpid` présente dans la bibliothèque C du système. C'est la fonction qui est directement appelée par le programme. Cette fonction commence par placer le numéro de l'appel système (noté `__NR_getpid`, valant 20 ici) dans EAX, puis les arguments éventuels dans les registres (EBX, ECX, EDX, ESI puis EDI). Une interruption logicielle est ensuite déclenchée (`int 0x80`).

Dans la routine de traitement d'interruption

Étant donné la configuration du processeur [1], elle sera traitée en *ring* 0, à un point d'entrée prédéfini (`arch/x86/kernel/entry_32.S`, `ENTRY(system_call)`).

```
        # system call handler stub
ENTRY(system_call)
        RING0_INT_FRAME                         # can't unwind into user space anyway
        pushl %eax                            # save orig_eax
        CFI_ADJUST_CFA_OFFSET 4
        SAVE_ALL
        GET_THREAD_INFO(%ebp)
                                              # system call tracing in operation / emulation
        testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
        jnz syscall_trace_entry
        cmpl $(nr_syscalls), %eax
        jae syscall_badsys
syscall_call:
        call *sys_call_table(,%eax,4)
        movl %eax,PT_EAX(%esp)                 # store the return value
        # ...
        INTERRUPT_RETURN
```

L'exécution reprend donc en *ring* 0, avec dans ESP le pointeur de pile noyau du processus. Les valeurs des registres ont été préservées, la macro `SAVE_ALL` les place sur la pile. Ensuite, à l'étiquette `syscall_call`, le numéro d'appel système (toujours dans EAX) sert d'index dans le tableau de fonctions `sys_call_table`.

Dans l'implantation de l'appel système

Puisque les arguments sont en place sur la pile, comme dans le cas d'un appel de fonction "classique", la convention d'appel *cdecl* est respectée. La fonction implantant l'appel système, nommée `sys_getpid`, peut donc être écrite en C.

On trouve cette fonction dans `kernel/timer.c` :

1. Il est impropre de dire que le processeur est configuré — tout dépend uniquement de l'état de certains registres, ici la *Global Descriptor Table* et les *Interrupt Descriptor Tables*.

```
SYSCALL_DEFINE0(getpid)
{
        return task_tgid_vnr(current);
}
```

La macro `SYSCALL_DEFINE0` nomme la fonction `sys_getpid`, et définit entre autres des points d'entrée pour les fonctionnalités de débogage du noyau. À la fin de la fonction, la valeur de retour est placée dans EAX, conformément à la convention *cdecl*.

Retour vers le ring 3

Au retour de la fonction, la valeur de retour est placée à la place de EAX là où les registres ont été sauvegardés sur la pile noyau (`PT_EFLAGS(%esp)`). L'instruction `iret` (derrière la macro `INTERRUPT_RETURN`) permet de restaurer les registres et de repasser en mode utilisateur, juste après l'interruption. La fonction de la bibliothèque C peut alors retourner au programme appelant.

8.5 Bug

On décrit le cas d'un pilote video 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*[2]. Ils sont similaires à des appels système, mais spécifique à un périphérique particulier. Les fonctions 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 suivant est présent dans le pilote KMS pour les GPU AMD Radeon :

```
/* 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;
        [...]
}
```

2. Ce nom vient de la fonction `ioctl()` pour *Input/Output Control*.

```
--- 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 8.2: Patch résolvant le problème de pointeur utilisateur.

On peut voit 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 suivant, 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;
};
```

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 USER, `value`. Ceci est possible puisqu'avant la traduction, on efface les types présents dans le programme C. Ainsi, un pointeur ne peut pas être distingué d'un entier transtypé en un pointeur. Avec cette configuration, nous obtenons une erreur de type à la ligne 16.

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 8.2) (`DRM_COPY_FROM_USER` étant une simple macro pout `copy_from_user`). Dans ce cas, on n'obtient pas d'erreur de typage.

8.6 Détails

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).

Ensuite, il faut réécrire les fonctions de manipulation de pointeurs fournies par le noyau : `get_user`, `put_user`, `copy_from_user`, `copy_to_user`, etc. Leur implantation revient à réaliser un test puis à faire la copie. À leur place on utilise les primitives $\cdot \Leftarrow_U \cdot$ et $\cdot \Rightarrow_U \cdot$.

Enfin, on peut lancer l'inférence de type.

Quatrième partie

Conclusion

CHAPITRE 9

CONCLUSION

9.1 Limitations

Un seul type d'entier est supporté.

Transtypage Puisque l'approche retenue est basée sur le typage statique, il est impossible de capturer de nombreuses constructions qui sont possibles, où même idiomatiques en C : les unions, les conversions de types (explicites ou implicites) et le *type punning*. 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.

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 sont corrects de ce point de vue, on peut désactiver ces vérifications et le reste des propriétés est toujours valable.

Un autre endroit, plus problématique, où des tests dynamiques sont faits est lorsqu'on recherche en mémoire des pointeurs référençant un cadre d'appel qui n'est plus valide (à l'aide de l'opérateur Cleanup$(\cdot)$). Supprimer ce test rend l'analyse incorrecte, car il est alors possible de faire référence à une variable avec un type différent.

De même, 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 le nettoyage en posant Cleanup$(m) = m$. Cette garantie peut être obtenue avec une analyse statique préalable [DDMPn10].

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, à 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. Le typage est alors direct :

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \textsc{Alloc}(e) : t\ \textsc{Kernel}\ *}\ (\textsc{Malloc}) \qquad \frac{\Gamma \vdash e : t\ \textsc{Kernel}\ *}{\Gamma \vdash \textsc{Free}(e)}\ (\textsc{Free})$$

En ce qui concerne l'exécution, on peut ajouter une troisième composante aux états mémoire : $m = (s, g, h)$ où h associe à une étiquette numérique une valeur (comme les globales mais indexées par des entiers plutôt que par leur nom). La libération de mémoire est problématique parce qu'il faut faire confiance au programmeur pour ne pas accéder aux zones mémoires libérées. Encore une fois, on peut utiliser une analyse préalable comme [DDMPn10]. Il est sûr (mais pas possible en pratique) d'ignorer les commandes de libérations de mémoire.

Structures récursives Une autre limitation est que seules les structures simples sont supportées. Les structures récursives, contenant un pointeur vers un objet du même type structure, ne sont pas typables. La raison est syntaxique : l'accès à un champ est noté $lv.l_S$ afin d'avoir à disposition le type complet de la structure.

Par exemple, une liste chaînée d'entiers peut être décrite de la manière suivante en C :

```
struct int_list {
    int value;
    struct int_list *next;
};
```

La compilation est possible car la définition complète de `struct int_list` n'est pas nécessaire pour connaître la taille d'un pointeur vers un objet de ce type.

Pour compiler `l.next` en SAFESPEAK, il faudrait écrire le type en extension :

$$l.next_{value:\textsc{Int};next:\{value:\textsc{Int};next:...\}\ *}$$

On peut fermer cette récursion infinie en ajoutant directement un opérateur de point fixe qui rend accessible le type d'un pointeur vers le type structure en train d'être défini.

$$S ::= \ldots \quad | \quad \text{fix}(p \to S)$$

On écrit alors :

$$l.next_{\text{fix}(p \to \{value:\textsc{Int};next:p\})}$$

Les structures mutuellement récursives peuvent être également exprimée de cette manière.

```
struct a {
    int value;
    struct b *next;
};

struct b {
    float value;
    struct a *next;
};
```

Ces structures sont respectivement traduites en :

$$A = \text{fix}(p \rightarrow \{value : \text{INT}; next : \{value : \text{FLOAT}; next : p\}\ *\})$$
$$B = \text{fix}(p \rightarrow \{value : \text{FLOAT}; next : \{value : \text{INT}; next : p\}\ *\})$$

Ce schéma suffit à compiler toutes les types de structures possibles en C. En effet, les seuls cas posant problème arrivent lorsqu'on fait référence à la structure en cours de définition ; et il n'est alors possible que d'y accéder par pointeur.

9.1.1 Assembleur

Comme la plupart des outils d'analyse de code C, il est impossible de traiter l'assembleur en ligne qui peut se trouver entre deux instructions. Dans le cas de Linux, le code est fait pour être portable, et les parties dépendantes d'une certaine architecture (et donc le code assembleur) sont isolées explicitement. On peut alors si nécessaire ajouter une annotation de type sur une fonction dont l'implantation est faite en assembleur, mais au sein de cette implantation on ne peut bien sûr rien dire.

9.2 Travaux futurs

9.2.1 Transtypage

Les casts entre types entiers ne posent pas de problèmes. Ceux entre types pointeurs ne posent pas de problème non plus puisqu'au moment de la traduction en SAFESPEAK on efface les étiquettes de types sur les pointeurs.

Mais les casts entre entiers et pointeurs posent problème. Si on autorise les casts, la seule manière sensible de les typer est la suivante.

$$\frac{\Gamma \vdash e : t\ q\ *}{\Gamma \vdash (\text{INT})\ e : \text{INT}}\ (\text{PTRINT-BAD}) \qquad \frac{\Gamma \vdash e : \text{INT}}{\Gamma \vdash (\text{PTR})\ e : t\ q\ *}\ (\text{INTPTR-BAD})$$

Cela pose problème car il est alors possible de créer une fonction qui convertit n'importe quel type pointeur en n'importe quel autre type pointeur :

$$\vdash \text{fun}(p)\{\text{RETURN}((\text{PTR})\ (\text{INT})\ p)\} : (t_a\ q_a\ *) \rightarrow t_b\ q_b\ *$$

L'opération problématique est en fait l'opérateur (INT) de conversion d'un pointeur vers un entier, car l'entier résultant ne pose pas de problème de sûreté ni de sécurité. Pour lui donner une sémantique, on peut supposer que l'environnement d'exécution est paramétré

par une fonction de placement en mémoire $\text{addr}^\sharp : \Phi \to \text{INT}$ qui à un chemin associe un entier. On étend alors la sémantique par la règle suivante.

$$\frac{}{\langle (\text{INT})\ \widehat{\&\varphi}, m\rangle \to \langle \text{addr}^\sharp(\varphi), m\rangle}\ (\text{CASTINT})$$

Quant au transtypage dans le sens inverse, il est plus compliqué à traiter. À l'exécution il est en effet nécessaire d'avoir à disposition une fonction $\text{interp}_{(t\ q\ *)} : \text{INT} \to t\ q\ *$ qui permette d'interpréter un entier comme pointeur qualifié q vers un objet de type t de manière qu'on ajoute la règle suivante.

$$\frac{}{\langle (\text{PTR})\ \widehat{n}, m\rangle \to \langle \text{interp}_{(t\ q\ *)}(\widehat{n}), m\rangle}\ (\text{CASTPTR})$$

L'exécution de l'analogue de $\text{addr}^\sharp$ demanderait de créer un pointeur depuis un entier. La solution la plus sûre est donc d'interdire totalement ces conversions.

9.2.2 Analyse du noyau Linux

Ici nous avons présenté l'analyse expérimentale d'une fonction problématique d'une interface programmation particulière. Mais le principe de l'analyse est applicable à toutes les fonctions de traitement d'`ioctl`s des pilotes KMS, et même à toutes les fonctions faisant partie des différentes interfaces recevant un pointeur de l'espace utilisateur.

9.2.3 Autres types abstraits

Notre approche est basée sur le fait de rendre abstrait un type de C et d'y interdire certaines opérations : ici, on marque un certain type de pointeur comme "utilisateur" et on interdit l'opérateur $*$ dessus.

Le langage C n'ayant pas de types abstraits [1] , on ne peut pas séparer la réprésentation d'un type (par exemple : entier signé de 32 bits) des opérations qui y sont attachées.

Dans de nombreuses interfaces, on emploie des types entiers qui servent d'étiquettes. Par exemple, les descripteurs de fichiers renvoyés par la fonction `open()` et passés au fonctions `read()` et `write()` ont le type `int`. Le langage autorise donc par exemple de multiplier entre eux deux descripteurs de fichiers, ce qui ne correspond pas à une opération concrète.

On peut aussi distinguer plusieurs types abstraits entre eux. Par exemple, si un encodeur vidéo manipule des numéros de *frame* et des identifiants de codec tous les deux entiers, on peut interdire d'utiliser avec un identifiant de codec une fonction prenant en paramètre un numéro de *frame*.

9.3 Conclusion

1. La seule abstraction possible est lorsqu'on manipule une structure par pointeur. Il n'est alors pas nécessaire de connaître sa définition totale. L'idiome est alors de placer uniquement une déclaration en avance (`struct s;`) dans l'en-tête (`.h`) et de renseigner la définition complète dans l'implantation (`.c`).

Annexes

ANNEXE A

MODULE RADEON KMS

```
/* 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

RÈGLES D'ÉVALUATION

$$\frac{\langle e, m\rangle \to \langle e', m'\rangle}{\langle C(\!|e|\!), m\rangle \to \langle C(\!|e'|\!), m'\rangle}\ (\textsc{Ctx}) \qquad \frac{\langle l\,v, m\rangle \to \langle l\,v', m'\rangle}{\langle C_L(\!|l\,v|\!)_L, m\rangle \to \langle C_L(\!|l\,v'|\!)_L, m'\rangle}\ (\textsc{Ctx-Lv})$$

$$\frac{\langle i, m\rangle \to \langle i', m'\rangle}{\langle C_I(\!|i|\!)_I, m\rangle \to \langle C_I(\!|i'|\!)_I, m'\rangle}\ (\textsc{Ctx-Instr})$$

Contextes

$$\begin{aligned} C ::= {} & C_L \\ \mid {} & C \boxplus e \\ \mid {} & v \boxplus C \\ \mid {} & \boxminus C \\ \mid {} & C \leftarrow e \\ \mid {} & \varphi \leftarrow C \\ \mid {} & \{l_1 : v_1; \ldots; l_i : C; \ldots; l_n : e_n\} \\ \mid {} & [v_1; \ldots; C; \ldots; e_n] \\ \mid {} & C(e_1, \ldots, e_n) \\ \mid {} & f(v_1, \ldots, C, \ldots, e_n) \end{aligned}$$

Contextes (left-values)

$$\begin{aligned} C_L ::= {} & \bullet \\ \mid {} & *\,C_L \\ \mid {} & C_L.l_S \\ \mid {} & C_L[e] \\ \mid {} & \varphi[C] \end{aligned}$$

Contextes (instructions)

$$\begin{aligned} C_I ::= {} & C_I; i \\ \mid {} & \textsc{If}(C)\{i_1\}\textsc{Else}\{i_2\} \\ \mid {} & \textsc{Return}(C) \\ \mid {} & \textsc{Decl}\ x = C\ \textsc{in}\{i\} \\ \mid {} & C \end{aligned}$$

FIGURE B.1: Règles d'évaluation – contextes

$\boxed{\Xi \to \Omega}$

$$\frac{}{\langle \Omega, m\rangle \to \Omega}\ (\textsc{Exp-Err}) \qquad \frac{\langle e, m\rangle \to \Omega}{\langle C(\!|e|\!), m\rangle \to \Omega}\ (\textsc{Eval-Err})$$

FIGURE B.2: Règles d'évaluation – erreurs

$\boxed{\langle l\,v, m\rangle \to \langle \varphi, m\rangle}$

$$\frac{a = \text{Lookup}(x, m)}{\langle x, m\rangle \to \langle a, m\rangle}\ (\textsc{Phi-Var}) \qquad \frac{}{\langle *\varphi, m\rangle \to \langle \widehat{*}\varphi, m\rangle}\ (\textsc{Phi-Deref})$$

$$\frac{}{\langle l\,v.l_S, m\rangle \to \langle l\,\widehat{v.l}, m\rangle}\ (\textsc{Phi-Struct}) \qquad \frac{}{\langle \varphi[n], m\rangle \to \langle \widehat{\varphi[n]}, m\rangle}\ (\textsc{Phi-Array})$$

$\boxed{\langle e, m\rangle \to \langle v, m\rangle}$

$$\frac{}{\langle c, m\rangle \to \langle \widehat{c}, m\rangle}\ (\textsc{Exp-Cst}) \qquad \frac{}{\langle f, m\rangle \to \langle \widehat{f}, m\rangle}\ (\textsc{Exp-Fun}) \qquad \frac{}{\langle \varphi, m\rangle \to \langle m[\varphi]_\Phi, m\rangle}\ (\textsc{Exp-Lv})$$

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

$$\frac{}{\langle \& \varphi, m\rangle \to \langle \widehat{\&}\, \varphi, m\rangle}\ (\textsc{Exp-AddrOf}) \qquad \frac{}{\langle \varphi \leftarrow v, m\rangle \to \langle v, m[\varphi \leftarrow v]_\Phi\rangle}\ (\textsc{Exp-Set})$$

$$\frac{}{\langle \{l_1 : v_1; \ldots; l_n : v_n\}, m\rangle \to \langle \{l_1 : \widehat{v_1; \ldots; l_n} : v_n\}, m\rangle}\ (\textsc{Exp-Struct})$$

$$\frac{}{\langle [v_1, \ldots, v_n], m\rangle \to \langle [\widehat{v_1, \ldots, v_n}], m\rangle}\ (\textsc{Exp-Array})$$

$$\frac{\begin{array}{c} f = \text{fun}(a_1, \ldots, a_n)\{i\} \qquad m_1 = \text{Push}(m_0, ((a_1 \mapsto v_1), \ldots, (a_n \mapsto v_n))) \\ \langle i, m_1\rangle \to \langle \textsc{Return}(v), m_2\rangle \qquad m_3 = \text{Pop}(m_2) \qquad m_4 = \text{Cleanup}(m_3) \end{array}}{\langle f(v_1, \ldots, v_n), m_0\rangle \to \langle v, m_4\rangle}\ (\textsc{Exp-Call})$$

FIGURE B.3: Règles d'évaluation – left-values et expressions

$$\boxed{\langle i, m\rangle \to \langle i', m\rangle}$$

$$\frac{\langle i, m\rangle \to \langle \textsc{Pass}, m'\rangle}{\langle (i; i'), m\rangle \to \langle i', m'\rangle}\ (\textsc{Seq}) \qquad \frac{}{\langle (\textsc{Pass}; i), m\rangle \to \langle i, m\rangle}\ (\textsc{Pass}) \qquad \frac{}{\langle v, m\rangle \to \langle \textsc{Pass}, m\rangle}\ (\textsc{Exp})$$

$$\frac{m' = \text{Extend}(m, x, v) \qquad \langle i, m'\rangle \to \langle \textsc{Pass}, m''\rangle \qquad m''' = \text{Cleanup}(m'' - x)}{\langle \textsc{Decl}\ x = v\ \textsc{in}\{i\}, m\rangle \to \langle \textsc{Pass}, m'''\rangle}\ (\textsc{Decl})$$

$$\frac{\begin{array}{c} m' = \text{Extend}(m, x, v) \\ \langle i, m'\rangle \to \langle \textsc{Return}(v_r), m''\rangle \qquad m''' = \text{Cleanup}(m'' - x) \end{array}}{\langle \textsc{Decl}\ x = v\ \textsc{in}\{i\}, m\rangle \to \langle \textsc{Return}(v_r), m'''\rangle}\ (\textsc{Decl-Return})$$

$$\frac{}{\langle \textsc{If}(0)\{i_t\}\textsc{Else}\{i_f\}, m\rangle \to \langle i_f, m\rangle}\ (\textsc{If-False}) \qquad \frac{v \neq 0}{\langle \textsc{If}(v)\{i_t\}\textsc{Else}\{i_f\}, m\rangle \to \langle i_t, m\rangle}\ (\textsc{If-True})$$

$$\frac{}{\langle \textsc{While}(e)\{i\}, m\rangle \to \langle \textsc{If}(e)\{i; \textsc{While}(e)\{i\}\}, m\rangle}\ (\textsc{While})$$

$$\frac{}{\langle \textsc{Return}(v); i, m\rangle \to \langle \textsc{Return}(e), m\rangle}\ (\textsc{Return})$$

$$\boxed{m \Vdash p \to m'}$$

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle}{m \Vdash e \to m'}\ (\text{T-}\textsc{Exp}) \qquad \frac{\langle e, m\rangle \to \langle v, m'\rangle}{(s, g) \Vdash x = e \to (s, (x \mapsto v) :: g)}\ (\text{T-}\textsc{Var})$$

$$\boxed{m \Vdash p \to^* m'}$$

$$\frac{}{m \Vdash [\,] \to^* m}\ (\text{T}^*\text{-}\textsc{Nil}) \qquad \frac{m \Vdash p \to m' \qquad m' \Vdash ps \to^* m''}{m \Vdash p :: ps \to^* m''}\ (\text{T}^*\text{-}\textsc{Cons})$$

$$\boxed{\Vdash P \to^* m}$$

$$\frac{([\,], [\,]) \Vdash P \to^* m}{\Vdash P \to^* m}\ (\textsc{Prog})$$

FIGURE B.4: Règles d'évaluation – instructions et phrases

ANNEXE C

RÈGLES DE TYPAGE

$\boxed{\Gamma \vdash c : t}$

$$\frac{}{\Gamma \vdash i : \text{INT}}\ (\text{CST-INT}) \qquad \frac{}{\Gamma \vdash d : \text{FLOAT}}\ (\text{CST-FLOAT}) \qquad \frac{}{\Gamma \vdash \text{NULL} : t*}\ (\text{CST-NULL})$$

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

$\boxed{\Gamma \vdash lv : t}$

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

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

FIGURE C.1: Règles de typage – constantes et variables

$\boxed{\Gamma \vdash \boxminus\, e : t}$

$$\frac{\Gamma \vdash e : \textsc{Int}}{\Gamma \vdash +e : \textsc{Int}}\ (\textsc{Unop-Plus-Int}) \qquad \frac{\Gamma \vdash e : \textsc{Float}}{\Gamma \vdash +.e : \textsc{Float}}\ (\textsc{Unop-Plus-Float})$$

$$\frac{\Gamma \vdash e : \textsc{Int}}{\Gamma \vdash -e : \textsc{Int}}\ (\textsc{Unop-Minus-Int}) \qquad \frac{\Gamma \vdash e : \textsc{Float}}{\Gamma \vdash -.e : \textsc{Float}}\ (\textsc{Unop-Minus-Float})$$

$$\frac{\boxminus \in \{\sim, !\} \qquad \Gamma \vdash e : \textsc{Int}}{\Gamma \vdash \boxminus\, e : \textsc{Int}}\ (\textsc{Unop-Not})$$

$\boxed{\Gamma \vdash e_1 \boxplus e_2 : t}$

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

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

$$\frac{\boxplus \in \{=, \neq\} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \qquad \textsc{Eq}(t)}{\Gamma \vdash e_1 \boxplus e_2 : \textsc{Int}}\ (\textsc{Op-Eq})$$

$$\frac{\boxplus \in \{+_p, -_p\} \qquad \Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \textsc{Int}}{\Gamma \vdash e_1 \boxplus e_2 : t*}\ (\textsc{Ptr-Arith})$$

$\boxed{\textsc{Eq}(t)}$

$$\frac{t \in \{\textsc{Int}, \textsc{Float}\}}{\textsc{Eq}(t)}\ (\textsc{Eq-Num}) \qquad \frac{}{\textsc{Eq}(t*)}\ (\textsc{Eq-Ptr}) \qquad \frac{\textsc{Eq}(t)}{\textsc{Eq}(t[\,])}\ (\textsc{Eq-Array})$$

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

FIGURE C.2: Règles de typage – opérateurs

$\boxed{\Gamma \vdash e : t}$

$$\frac{\Gamma \vdash lv : t}{\Gamma \vdash \&lv : t*}\ (\textsc{Addr}) \qquad \frac{\forall i \in [1;n], \Gamma \vdash e_i : t_i}{\Gamma \vdash \{l_1 : e_1; \ldots; l_n : e_n\} : \{l_1 : t_1; \ldots; l_n : t_n\}}\ (\textsc{Struct})$$

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

$$\frac{\forall i \in [1;n], \Gamma \vdash e_i : t}{\Gamma \vdash [e_1; \ldots; e_n] : t[\,]}\ (\textsc{Array}) \qquad \frac{\Gamma' = (\Gamma - \underline{R}), \vec{a} : \vec{t}, \underline{R} : t_r \qquad \Gamma' \vdash i}{\Gamma \vdash \text{fun}(\vec{a})\{i\} : \vec{t} \to t_r}\ (\textsc{Fun})$$

$\boxed{\Gamma \vdash i}$

$$\frac{}{\Gamma \vdash \textsc{Pass}}\ (\textsc{Pass}) \qquad \frac{\Gamma \vdash i_1 \qquad \Gamma \vdash i_2}{\Gamma \vdash i_1; i_2}\ (\textsc{Seq}) \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash e}\ (\textsc{Exp})$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma, x : t \vdash i}{\Gamma \vdash \textsc{Decl}\ x = e\ \textsc{in}\{i\}}\ (\textsc{Decl}) \qquad \frac{\Gamma \vdash e : \textsc{Int} \qquad \Gamma \vdash i_1 \qquad \Gamma \vdash i_2}{\Gamma \vdash \textsc{If}(e)\{i_1\}\textsc{Else}\{i_2\}}\ (\textsc{If})$$

$$\frac{\Gamma \vdash e : \textsc{Int} \qquad \Gamma \vdash i}{\Gamma \vdash \textsc{While}(e)\{i\}}\ (\textsc{While}) \qquad \frac{\Gamma \vdash \underline{R} \leftarrow e}{\Gamma \vdash \textsc{Return}(e)}\ (\textsc{Return})$$

$\boxed{\Gamma \vdash p \to \Gamma'}$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e \to \Gamma}\ (\text{T-}\textsc{Exp}) \qquad \frac{\Gamma \vdash e : t \qquad \Gamma' = (x, t), \Gamma}{\Gamma \vdash x = e \to \Gamma'}\ (\text{T-}\textsc{Var})$$

FIGURE C.3: Règles de typage

ANNEXE D

PREUVES

D.1 Composition de lentilles

Démonstration. Pour prouver que $\mathscr{L}_1 \ggg \mathscr{L}_2 \in \text{LENS}_{A,C}$, il suffit de prouver les trois propriétés caractéristiques.

GetPut

$$
\begin{aligned}
& \text{put}_{\mathscr{L}}(\text{get}_{\mathscr{L}}(r), r) \\
=& \{\text{définition de get}_{\mathscr{L}}\} \\
& \text{put}_{\mathscr{L}}(\text{get}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_1}(r)), r) \\
=& \{\text{définition de put}_{\mathscr{L}}\} \\
& \text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_1}(r)), \text{get}_{\mathscr{L}_1}(r)), r) \\
=& \{\text{GETPUT sur } \mathscr{L}_2\} \\
& \text{put}_{\mathscr{L}_1}(\text{get}_{\mathscr{L}_1}(r), r) \\
=& \{\text{GETPUT sur } \mathscr{L}_1\} \\
& r
\end{aligned}
$$

PutGet

$$
\begin{aligned}
& \text{get}_{\mathscr{L}}(\text{put}_{\mathscr{L}}(a, r)) \\
=& \{\text{définition de get}_{\mathscr{L}}\} \\
& \text{get}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}}(a, r))) \\
=& \{\text{définition de put}_{\mathscr{L}}\} \\
& \text{get}_{\mathscr{L}_2}(\text{get}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_1}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r)), r))) \\
=& \{\text{PUTGET sur } \mathscr{L}_1\} \\
& \text{get}_{\mathscr{L}_2}(\text{put}_{\mathscr{L}_2}(a, \text{get}_{\mathscr{L}_1}(r))) \\
=& \{\text{PUTGET sur } \mathscr{L}_2\} \\
& a
\end{aligned}
$$

PutPut

$$
\begin{aligned}
&\mathrm{put}_{\mathscr{L}}(a', \mathrm{put}_{\mathscr{L}}(a, r)) \\
=&\{\text{définition de } \mathrm{put}_{\mathscr{L}}\} \\
&\mathrm{put}_{\mathscr{L}}(a', \mathrm{put}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_2}(a, \mathrm{get}_{\mathscr{L}_1}(r)), r)) \\
=&\{\text{définition de } \mathrm{put}_{\mathscr{L}}\} \\
&\mathrm{put}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_2}(a', \mathrm{get}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_2}(a, \mathrm{get}_{\mathscr{L}_1}(r)), r))), \mathrm{put}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_2}(a, \mathrm{get}_{\mathscr{L}_1}(r)), r)) \\
=&\{\text{GETPUT sur } \mathscr{L}_1\} \\
&\mathrm{put}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_2}(a', \mathrm{put}_{\mathscr{L}_2}(a, \mathrm{get}_{\mathscr{L}_1}(r))), \mathrm{put}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_2}(a, \mathrm{get}_{\mathscr{L}_1}(r)), r)) \\
=&\{\text{PUTPUT sur } \mathscr{L}_2\} \\
&\mathrm{put}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_2}(a', \mathrm{get}_{\mathscr{L}_1}(r)), \mathrm{put}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_2}(a, \mathrm{get}_{\mathscr{L}_1}(r)), r)) \\
=&\{\text{PUTPUT sur } \mathscr{L}_1\} \\
&\mathrm{put}_{\mathscr{L}_1}(\mathrm{put}_{\mathscr{L}_2}(a', \mathrm{get}_{\mathscr{L}_1}(r)), r) \\
=&\{\text{définition de } \ggg\} \\
&\mathrm{put}_{\mathscr{L}}(a', r)
\end{aligned}
$$

□

D.2 Progrès

On prouve de manière récursive mutuelle les théorèmes 5.1 et 5.2

Démonstration. On procède par induction sur la dérivation de $\Gamma \vdash e : t$, et plus précisément sur la dernière règle utilisée.

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.

LV-VAR : Puisque $(x, t) \in \Gamma$ et $\Gamma \vdash_{mem} m$, il existe $(x \mapsto v) \in m$. La règle d'évaluation PHI-VAR s'applique donc.

LV-DEREF : Appliquons l'hypothèse de récurrence à $l\,v$ (vue en tant qu'expression).

- $l\,v = v$. Puisque $\Gamma \vdash v : t*$, on déduit du lemme 5.2 que $v = \varphi$ ou $v = \mathrm{NULL}$.
  Dans le premier cas, la règle PHI-DEREF s'applique : $\langle e, m\rangle \to \langle \hat{*}\varphi, m\rangle$. Dans le second, puisque $\langle *\mathrm{NULL}, m\rangle \to \Omega_{ptr}$, on a $\langle e, m\rangle \to \Omega_{ptr}$.
- $\langle l\,v, m\rangle \to \langle e', m'\rangle$. De CTX avec $C = *\bullet$, on obtient $\langle e, m\rangle \to \langle *e', m'\rangle$.
- $\langle l\,v, m\rangle \to \Omega$. 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 $\Gamma \vdash e :$ 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 du théorème 5.2 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 $\langle l\,v, m\rangle \to \Omega$.

OP-INT : Cela implique que $e = e_1 \boxplus e_2$. Par le lemme 5.1, on en déduit que $\Gamma \vdash e_1 :$ INT et $\Gamma \vdash e_2 :$ 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 $\langle e, m\rangle \to \Omega_{div}$.
  - $\exists (e_2', m''), \langle e_2, m'\rangle \to \langle e_2', m''\rangle$.
    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.

OP-COMPARABLE : 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, $\langle + v_1, m\rangle \rightarrow \langle \widehat{+} v_1, m\rangle$, c'est à dire en posant $v = \widehat{+} v_1$, $\langle e, m\rangle \rightarrow \langle v, m\rangle$.
- soit $\exists e_1', m', \langle e_1, m\rangle \rightarrow \langle e_1', m'\rangle$. Alors en appliquant CTX avec $C = + \bullet$, on obtient $\langle e, m\rangle \rightarrow \langle e_1', m'\rangle$.
- soit $\langle e_1, m\rangle \rightarrow \Omega$. De EVAL-ERR avec $C = + \bullet$ il vient$\langle e, m\rangle \rightarrow \Omega$.

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.

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, $\langle e, m\rangle \rightarrow \langle e_1 \mathbin{\widehat{\boxplus}} n, m\rangle$.

On se réfère ensuite à la définition de $\widehat{\boxplus}$ (page 50) : si e_1 est de la forme $\varphi[m]$, alors $e_1 \mathbin{\widehat{\boxplus}} n = \varphi[m+n]$. Donc $\langle e, m\rangle \rightarrow \langle \varphi[m+n], m\rangle$.

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, $\langle e, m\rangle \rightarrow \Omega_{ptr}$.

ADDR :

$$\left(\frac{\Gamma \vdash l\,v : t}{\Gamma \vdash \&l\,v : t*}\ (\text{ADDR}) \right)$$

On applique l'hypothèse de récurrence du théorème 5.2 à $l\,v$.

Les cas d'évaluation et d'erreur sont traités en appliquant respectivement CTX et EVAL-ERR avec $C = \&\bullet$. On détaille le cas où $l\,v = \varphi$.

$$\left(\frac{}{\langle \&\,\varphi, m\rangle \rightarrow \langle \widehat{\&}\,\varphi, m\rangle}\ (\text{EXP-ADDROF}) \right)$$

SET :

$$\left(\frac{\Gamma \vdash l\,v : t \qquad \Gamma \vdash e : t}{\Gamma \vdash l\,v \leftarrow e : t}\ (\text{SET}) \right)$$

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 :

- $\exists p \in [1;n], e'_p, m : e_1 = v_1, \ldots, e_{p-1} = v_{p-1}, \langle e_p, m\rangle \rightarrow \langle e'_p, m'\rangle$. Alors on peut appliquer CTX avec $C = [v_1, \ldots, v_{p-1}, \bullet, e_{p+1}, \ldots, e_n]$.
- $\exists p \in [1;n], \Omega : e_1 = v_1, \ldots, e_{p-1} = v_{p-1}, \langle e_p, m\rangle \rightarrow \Omega$. 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 $f = \text{fun}(\vec{a})\{i\}$: EXP-CALL s'applique alors.

FUN : Ce cas est direct : la règle EXP-FUN s'applique.

□

D.3 Préservation

.
(théorème 5.3)

Démonstration. On procède par induction sur la forme de l'expression e.

Constante c On détaille par exemple le cas d'une constante entière.
D'une part, $\Gamma \vdash e : t$ donc d'après le lemme 5.2, $t = \text{INT}$.
D'autre part, la seule règle d'évaluation possible est EXP-CST qui évalue en une constante entière.

Accès mémoire $l\,v$

Opération unaire $\boxminus\, e$

Opération binaire $e \boxplus e$

Pointeur $\&l\,v$

Affectation $l\,v \leftarrow e$

Structure $\{l_1 : e_1; ; l_n : e_n\}$

Tableau $[e_1; \ldots; e_n]$

Fonction f

Appel de fonction $e(e_1, \ldots, e_n)$

□

TODO

- titres dans les chapeaux buggés

ANNEXE E

TODO

E.1 État de l'art

- IA :
  - produit réduit
  - polyspace ?
  - APRON ?
  - Frama-C ?
- difficultés : récursion
- hoare : quand est-ce que le compile time suffit ? et le runtime nécessaire ?
- types : citer H98[PJ03], perl [Wal00], DAOC[CMP03] & RWH[OGS08] ?
- proof assistants
  - Dependent types
  - proof : theorem :: type : term
  - Coq
  - Agda, termination checker
  - proof irrelevance
  - Theorems for Free[Wad89]
- Analyse de flot : Ce que nous voulons vérifier peut être vue comme une propriété de flot. Un tour d'horizon des problèmes et techniques existantes peut être trouvé dans [SM03].
- Divers : Taint sequences [CMP10],
- Régions [TJ92] [TT94] [TT93]

L'analyse de taintage consiste à ajouter des étiquettes au données décrivant leur provenance, de manière à ce que chaque accès dangereux soit vérifié à l'exécution. Cela se marie bien avec les langages typés dynamiquement, comme le montre l'exemple célèbre du mode "souillé" (*tainted*) de Perl [Wal00]. Il est aussi possible, mais plus délicat, d'appliquer cette technique à des langages compilés [CLO07, SAB10].

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].

- Les pointeurs sur fonction rendent floue la limite qui est habituellement présente entre instructions et données. En leur présence il est impossible de faire une analyse de flot de contrôle indépendante du flot de données. Pour pouvoir les traiter, il faut que le domaine abstrait en question soit assez précis pour qu'un pointeur abstrait se concrétise en un ensemble réduit de fonctions. Dans le cas où le domaine ne peut pas borner l'ensemble des fonctions possibles et renvoie $\top$, l'analyse ne peut pas continuer.
- L'allocation dynamique de données, présente dans le langage C par le biais des fonctions `malloc` et `free`, modifie le modèle mémoire nécessaire. Sans celle-ci, l'ensemble des zones mémoire possibles peut être décrit statiquement : ce sont les noms de variable. Ce qu'introduit `malloc` au langage, c'est une zone mémoire qui n'a pas de nom, et sur laquelle on n'a qu'un pointeur.
- Le transtypage (*casts*) entre entiers et pointeurs est particulièrement délicat à traiter. Dans les modèles abstraits, les pointeurs sur données ou sur fonctions n'ont pas de représentation numérique, seulement une représentation symbolique. Même dans l'exécution concrète, la représentation numérique d'un pointeur est lié à de nombreux choix faits par l'environnement d'exécution (comme la randomisation de l'espace d'adressage) qui ne peuvent pas facilement être modélisés.
- Les nombres flottants (types `float`, `double` et `long double`) ont une sémantique particulière, et il n'est pas correct d'approcher leur sémantique par une sémantique dans $\mathbb{R}$. Afin d'être correct, il faut établir des domaines spécifiques au flottant, comme [CMC08]. Un tour d'horizon des difficultés liées aux flottants est effectué dans [Mon08].

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. Celle-ci sont ignorées par le compilateur, mais un outil d'analyse statique ad-hoc nommé Sparse [5] [TTL] utilisé pour détecter les cas les plus simples d'erreurs. Il demande aussi au programmeur d'ajouter beaucoup d'annotations dans le programme. quantifier

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. 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 chaîne de format [STFW01] 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]. Puisqu'elle consiste à ajouter un qualificateur à chaque étage de type, cette approche est plus générique mais plus complexe que la nôtre.

E.2 Évaluateur

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 très bas niveau.

Les types de données de C sont établis pour représenter les mots mémoire manipulables par les processeurs : essentiellement des entiers et flottants de plusieurs tailles. Les types composés correespondent 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.

Le système de types rudimentaire de C 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 d'établir un langage plus simple mais qui permettra de raisonner sur une certaine classe de programmes C.

- substitutions dans les ctx : éditer C_l et C_i (en fait, séparer selon le truc substitué (2è arg), pas le premier)
- cas d'erreur si on accède à un index d'une struct ou vice versa ?
- Exp-AddrOf sur toutes les lv
- C : := &C
- changer les para de présentation des règles
- widehats sur les constantes ?
- liste d'assos → fonction
- définir les opérations d'ajout/remplacement sur les états mémoire
- interdire d'avoir plusieurs variables qui ont le même nom dans un cadre
- return implicite en fin de fct
- clarifier quand il faut un $\langle \cdot, \cdot \rangle \to \langle \cdot, \cdot \rangle$ et quand il faut un $\langle \cdot, \cdot \rangle \to^* \langle \cdot, \cdot \rangle$
- "et" et "ou" lazy
- dans les lentilles, L désigne à la fois celle des champs et celle des listes d'asso

Limitations :

- (ou feature) variables non initialisées
- tableaux de taille dynamique ?
- structures récursives et corécursives
- figures gramdef : singulier ou pluriel ?

Extrait PLAS

We present SAFESPEAK, a memory-safe imperative programming language. Let us go through its major features.

Scalar data : there are four types of scalars : integers, floats, pointers and a special "unit" value. Integers are 32 bit wide signed integers, akin to the `int32_t` data type in C, but this size is arbitrary. Floats are 32 bit wide floating point numbers with their usual semantics, Pointers are used to reference any modifiable piece of data, as embodied by the concept of *left-values* described below. Finally, the special unit value, written () is the one that is returned by procedures that do not return a useful value. It is similar to the `unit` type of some functional languages. A particular TAINT(e) construct allows us to construct expressions annotated as coming from userspace, emulating a system call, and two operators $\cdot \Leftarrow_U \cdot$ and $\cdot \Rightarrow_U \cdot$ provide safe copy from and to userspace.

$e ::=$	**Expressions**
$\mid c$	Constant
$\mid lv$	Left-value
$\mid \boxminus\, e$	Unary operation
$\mid e \boxplus e$	Binary operation
$\mid \& lv$	Pointer
$\mid lv \leftarrow e$	Assignment
$\mid \{l_1 : e_1; \ldots; l_n : e_n\}$	Structure
$\mid [e_1; \ldots; e_n]$	Array
$\mid \text{fun}(x_1, \ldots, x_n)\{i\}$	Function
$\mid e(e_1, \ldots, e_n)$	Function call
$\mid \text{TAINT}(e)$	Tainted value
$\mid lv \Leftarrow_U e$	Load from userspace
$\mid e \Rightarrow_U e$	Store to userspace

$c ::=$	**Constants**
$\mid n$	Integer
$\mid d$	Float
$\mid \text{NULL}$	Null pointer
$\mid ()$	Unit value

Composite data : there are two kinds of composite data : arrays and structures. Arrays are homogeneous pieces of memory with a fixed, statically known size. Elements within an array can be accessed with an integer index. In the case of an array with n elements, the first one is accessed with the index 0, and the last one with $n-1$. Trying to access to an element with any other index results in a run-time error. Arrays are written $[e_1; \ldots; e_n]$ with $n > 0$. Unlike in C, there is no particular relation between arrays and pointers : arrays are plain values that can be passed around.

The other kind of composite data is structures, that are an heterogeneous collection of variables of a fixed size. Elements are accessed with a *label* l_S, that is always statically known (labels are not first-class elements of the language). The subscript S index is placed here by our translator to guide type inference, and is ignored by the evaluator. Structures are written $\{l_1 : e_1; \ldots; l_n : e_n\}$ with $n > 0$.

Memory is organised in variables, that hold scalar or composite data. It is split between a stack of local variables, and a set of global variables. Local variables are structured in a list of stack frames that are used to hold information for each function call. Global variables are on the other hand an unstructured set accessible anywhere in the program. We suppose that inside every function, all locals have a distinct name ; same goes for globals. Local variables from two different function can have the same name.

Left-values : Inside expressions, it is possible to directly access visible variables, that is the ones that are part of the most recent stack frame, and global ones. It is also possible to manipulate other parts of the memory by the means of a pointer. Pointers hold enough informa-

tion to address in a non ambiguous manner any part of the memory.

$lv ::=$	**Left-values**
$\mid x$	Variable
$\mid *\, lv$	Dereference
$\mid lv.l_S$	Field access
$\mid lv[e]$	Indexed access

Operators are used to combine expressions together. We support C's operators on scalar data. To help the type system, we distinguish between integer operations and floating point operations by writing the latter with an extra dot : for example $3+2=5$ and $3.14 +_{.} 2.72 = 5.86$. Equality (and difference) tests can be done on scalar data as well as on composite data. In that case, it is done recursively on their members. Pointers are equal if they describe the same path in memory. Pointer arithmetic is possible by the means of operators $+_p$ and $-_p$, that move a pointer inside an array (if it used on another type of pointer, it results in a run-rime error Ω_{ptr}).

Functions are first-class expressions that can be assigned to variables that can then be called. However, unlike in some other programming languages, they are not lexical closures, meaning that no capture of variables from enclosing scopes happen at the site of a function declaration.

This is a compromise between C where functions can only be global, and functional languages where it is necessary to build closures containing (pointers to) captured variables at each function call. This also eliminates the need for special-cased "function pointers".

Each function call creates a stack frame to hold room for the function's parameters and local variables. Upon return, the most recent frame is destroyed. A specificity of our system is that at this moment, we also clear references that are about to become invalid. More precisely, we turn to NULL every pointer that refer to the latest stack frame. Using this technique, we can avoid so-called *dangling* pointers.

Note that this is a dynamic step and requires a full memory scan, which is very inefficient. This is done only for soundness reasons ; as discussed in Section **??**, if a preliminary analysis proves that this is unnecessary we can skip this step.

Instructions inside functions includes the empty instruction, sequence, evaluation of an expression (including assignment), or variable declation. Control flow is restricted to alternative (IF) and loops (WHILE). The RETURN construct also exits early from the current function.

$i ::=$	**Instructions**
$\mid$ PASS	Empty instruction
$\mid i_1; i_2$	Sequence
$\mid e$	Expression
$\mid$ DECL $x = e$ IN$\{i\}$	Declaration
$\mid$ IF$(e)\{i\}$ELSE$\{i\}$	Alternative
$\mid$ WHILE$(e)\{i\}$	Loop
$\mid$ RETURN(e)	Function return

A program is a sequence of top-level sentences, of the form of global declarations or expressions. A declaration creates a global variable that is accessible everywhere after in the

$v ::=$		**Values**
	$\widehat{c}$	Constant
	$\widehat{f}$	Function
	$\widehat{\&}\,\varphi$	Reference
	$\widehat{\{l_i : v_i; \ldots\}}$	Structure
	$\widehat{[v_i; \ldots]}$	Array
	Ω	Error
$\varphi ::=$		**Paths**
	a	Address
	$*\varphi$	Dereference
	$\varphi.l$	Field
	$\varphi[n]$	Index
	TAINTED(φ)	Tainted value

$s ::=$		**Stack**
	$[\,]$	Empty stack
	$\{x_i \mapsto v_i; \ldots\} :: s$	Extra frame
$m ::=$		**Memory**
	$(s,$	Stack,
	$\{x_1 \mapsto v_1; \ldots\})$	globals

$a ::=$		**Addresses**
	(n, x)	Local variable
	(x)	Global variable

$\Omega ::=$		**Errors**
	Ω_{array}	Buffer overflow
	Ω_{ptr}	Invalid dereference
	Ω_{div}	Division by zero
	Ω_{field}	Bad field
	Ω_{taint}	Isolation error
$\Xi ::=$		**Interpreter**
	$\langle e, m \rangle$	Expression, memory
	$\langle i, m \rangle$	Instruction, memory
	Ω	Error

FIGURE E.1: Memory and values

whole program. On the other hand, a plain expression is only evaluated for its side-effects [1] .

$s ::=$		**Toplevel sentences**
	$x = e$	Global variable
	e	Expression evaluation
$P ::= (s_1, \ldots, s_n)$		**Program**

Evaluation

Operational semantics described thereafter are a binary relation between states Ξ of an interpreter described in Figure E.1.

Every syntactic constant c has a corresponding value $\widehat{c}$. The same holds for functions, because as they do not capture their lexical environment, there is no need to build a closure.

Composite values also have their matching semantic parts : an array of values is a value, and a structure of values is a value. The same $\widehat{\cdot}$ notation is used to remove the ambiguity.

Pointers are reduced to a memory path φ that is an evaluated left-value : type information is removed from the field names, and array indices are evaluated to integers. Also, names are

1. There is no implicit entry point such as a main() function, so it is necessary to manually insert a call it.

made explicit by adjoining the name of local variables with the index of the stack frame they belong to. This is necessary in the cases where two frames hold a variable with the same name ; for example in the presence of recursive functions.

The semantics described here are given in the style of Plotkin's Structural Operational Semantics[Plo04] : expressions are gradually reduced to constants, and instructions are simplified until a terminal instruction of the form PASS or RETURN(v).

Reading memory A variable is evaluated in two steps ; first, its address is computed.

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

Lookup : MEM × ID → ADDR is an operator mapping variable names to addresses. That is, it performs name resolution. It returns (n, x) where n is the size of the stack if a local variable of this name exists ; or then (x) if a global of this name exists.

Then, an address is a path φ, that is used to access the corresponding value stored in m.

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

We omit here a formal definition of $m[\cdot]_\Phi$. In a nutshell, it walks the memory to return the value ultimately described by φ. It can raise Ω_{array} if a bad array access occurs or Ω_{ptr} if NULL is dereferenced.

Writing memory The $m[\cdot]_\Phi$ notation is extended to perform updates. We note $m[\varphi \leftarrow v]_\Phi$ the memory state obtained by replacing the value at φ by v. The same kind of errors can occur.

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

Variable declaration extends the current stack frame with a new binding $x \mapsto v$.

$$\frac{m' = \text{Extend}(m, x, v) \qquad \langle i, m'\rangle \to \langle \text{PASS}, m''\rangle \qquad m''' = \text{Cleanup}(m'' - x)}{\langle \text{DECL } x = v \text{ IN}\{i\}, m\rangle \to \langle \text{PASS}, m'''\rangle} \text{ (DECL)}$$

It relies on the following operator :

$$\begin{aligned} &\text{Extend} : \text{MEM} \times \text{ID} \times \text{VAL} \to \text{MEM} \\ &\quad \text{Extend}(f :: fs, g), x) = (((x \mapsto v) :: f) :: fs), g) \end{aligned}$$

Function call uses a complex rule, because it handles the calling convention (pushing and popping parameters on the stack). We suppose that all functions exit through a RETURN(·) instruction.

$$\frac{\begin{array}{c} f = \text{fun}(a_1, \ldots, a_n)\{i\} \qquad m_1 = \text{Push}(m_0, ((a_1 \mapsto v_1), \ldots, (a_n \mapsto v_n))) \\ \langle i, m_1\rangle \to \langle \text{RETURN}(v), m_2\rangle \qquad m_3 = \text{Pop}(m_2) \qquad m_4 = \text{Cleanup}(m_3) \end{array}}{\langle f(v_1, \ldots, v_n), m_0\rangle \to \langle v, m_4\rangle} \text{ (EXP-CALL)}$$

Push : MEM × FRAME → MEM adds a new stack frame, and Pop : MEM → MEM removes it.

Cleanup : MEM → MEM removes references to dangling pointers. It walk all the memory and replaces references to the current stack frame [2] with NULL. For example :

$$\begin{aligned}&\text{Cleanup}([[x \mapsto 0]; [p \to \{t : \&(2, x); u : 5\}]], [\,])\\ &\quad ([[x \mapsto 0]; [p \to \{t : \text{NULL}; u : 5\}]], [\,])\end{aligned}$$

System calls are emulated through the TAINT(·) operator. For example, the bridge between the functions `read()` and `sys_read()` is the following. Integer parameters don't need to be converted, as they do not have qualifiers.

```
read = fun (fd, p, n) {
  Decl up = Taint(p) in {
    sys_read (fd, up, n)
  }
}
```

Its evaluation is :

To allow a safe copy between userspace and the kernel, the Linux kernel provides the macros `get_user()` or `put_user()`, which check and copy scalar data, or `copy_from_user()` and `copy_to_user()` which act similarly to `memcpy()`. All of these constructs return the error `-EFAULT` when the user pointer is within the bounds of the kernel's memory. To model them, we provide the $\cdot \Leftarrow_U \cdot$ and $\cdot \Rightarrow_U \cdot$ operators that can copy a value at a time.

Their semantics is not detailed, but it consists in checking that the user pointer is of the form TAINTED(φ). If it is the case, then a copy from φ (in the case of $\Leftarrow_U$) or to φ (in the case of $\Rightarrow_U$) is performed and the expressions reduces to 1. Else, no copy is done and it reduces to 0.

Contexts To simplify the presentation of evaluation rules, we use reduction contexts, as described by Felleisen *et al.* [WF94, FH92]. The idea is that if it is possible to reduce an expression e_1 to an expression e_2, then it is possible to reduce a bigger expression where e_1 appears by substituting it by e_2.

To make this work we have to proceed in three steps : split an expression according to a context, apply a reduction under a context, and merge the reduced expression with a context.

Splitting across a context corresponds to defining a grammar of contexts. By defining these (Figure E.2), we make an explicit choice of evaluating some constructs in a left-to-right order. For example, the contexts for binary operations are $C \boxplus e$ and $v \boxplus C$, so it is impossible to start reducing the right part before the left one has been fully evaluated.

A similar remark can be done for n-ary constructs (structure and array literals, as well as function calls) : because there are always values on the left hand of C, arguments are evaluated left-to-right.

The second step is expressed in the following rule. Any transition between interpreter states can be lifted in a bigger context.

$$\frac{\langle e, m\rangle \to \langle e', m'\rangle}{\langle C(\!|e|\!), m\rangle \to \langle C(\!|e'|\!), m'\rangle}\ (\text{CTX})$$

The $C(\!|\cdot|\!)$ operator corresponds to the last operation : it pastes an expression in place of the unique "hole" • that appears in each context [3] .

2. That is, every address of the form (n, x) where n is the current size of the stack.

3. In the definition of C, every alternative produces exactly one nonterminal C or one terminal •. So, every derivation tree is linear, and • appears exactly once in every evaluation context.

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

FIGURE E.2: Reduction contexts

For example, let us unroll the evaluation of $3 + x$ to the value $\widehat{5}$ starting from a memory state $m = ([\,], \{x \mapsto \widehat{2}\})$. It is also depicted in Figure E.3.

(a) Let us first remark that $\langle 3, m\rangle \rightarrow \langle \widehat{3}, m\rangle$ (because of EXP-CST)

(b) By applying CTX from (a) with $C = \bullet + x$, we obtain $\langle 3 + x, m\rangle \rightarrow \langle \widehat{3} + x, m\rangle$. Note that with such an evaluation context, the right hand operand does not have to be evaluated.

(c) x, as a variable name, is a left-value. To evaluate it, the first step is to turn it into a path $\varphi = \text{Lookup}(x, m)$ (because of rule PHI-VAR). Because there are no local variables and a global variable named x, $\varphi = (x)$. Note that in this step, no memory lookup was involved (only the names of variables were used). So $\langle x, m\rangle \rightarrow \langle (x), m\rangle$.

(d) In order to evaluate this path into a value, we have to perform a memory lookup. The global variable (x) maps to 2 in m, so according to EXP-LV, $\langle (x), m\rangle \rightarrow \langle \widehat{2}, m\rangle$.

(e) Because of closure of $\rightarrow$, we get from (c) and (d) that $\langle x, m\rangle \rightarrow \langle \widehat{2}, m\rangle$.

(f) By applying CTX with $C = \widehat{3} + \bullet$ to (e), $\langle \widehat{3} + x, m\rangle \rightarrow \langle \widehat{3} + \widehat{2}, m\rangle$ holds. The fact that the left hand operand is a value is important.

(g) Because of EXP-BINOP, $\langle \widehat{3} + \widehat{2}, m\rangle \rightarrow \langle \widehat{3} \hat{+} \widehat{2}, m\rangle$, ie $\langle \widehat{3} + \widehat{2}, m\rangle \rightarrow \langle \widehat{5}, m\rangle$.

(h) By closure of $\rightarrow$, we get from (b), (f) and (g) that $\langle 3 + x, m\rangle \rightarrow \langle \widehat{5}, m\rangle$.

Evaluation rules are given in annex in Figures **??** and **??**. Most are simple, but a few deserve to be explained in detail.

Error propagation is done in two cases. First, if an expression produces an error (seen as a value), then the same error (seen as a state Ξ) propagates to the interpreter. Second, if a sub-expression causes an error, then a bigger expression causes the same error.

$$\frac{}{\langle \Omega, m\rangle \rightarrow \Omega}\ (\textsc{Exp-Err}) \qquad \frac{\langle e, m\rangle \rightarrow \Omega}{\langle C(\!|e|\!), m\rangle \rightarrow \Omega}\ (\textsc{Eval-Err})$$

$$\text{(b)}\ \frac{\text{(a)}\ \dfrac{}{\langle 3, m\rangle \to \langle \widehat{3}, m\rangle}}{\langle 3+x, m\rangle \to \langle \widehat{3}+x, m\rangle} \qquad \text{(f)}\ \frac{\text{(e)}\ \dfrac{\text{(c)}\ \dfrac{}{\langle x, m\rangle \to \langle (x), m\rangle} \quad \text{(d)}\ \dfrac{}{\langle (x), m\rangle \to \langle \widehat{2}, m\rangle}}{\langle x, m\rangle \to \langle \widehat{2}, m\rangle}}{\langle \widehat{3}+x, m\rangle \to \langle \widehat{3}+\widehat{2}, m\rangle}$$

$$\text{(h)}\ \frac{(b) \qquad (f) \qquad \text{(g)}\ \dfrac{}{\langle \widehat{3}+\widehat{2}, m\rangle \to \langle \widehat{3}\widehat{+}\widehat{2}, m\rangle}}{\langle 3+x, m\rangle \to \langle \widehat{5}, m\rangle}$$

FIGURE E.3: Evaluation of $3+x$ into $\widehat{5}$

Programs are evaluated one sentence after another. Declarations add a global variable.

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle}{m \Vdash e \to m'}\ \text{(T-EXP)} \qquad \frac{\langle e, m\rangle \to \langle v, m'\rangle}{(s, g) \Vdash x = e \to (s, (x \mapsto v) :: g)}\ \text{(T-VAR)}$$

E.3 Typage

- ordre des sections
- versions $\Gamma \vdash i$ des propriétés
- preuve de progres : état mémoire : doublet/triplet
- définir les opérations d'ajout/remplacement sur les contextes de typage
- 2 pointeurs peuvent etre égaux sans comparer les valeurs pointées
- extension de contextes : : : ou , ?

Fonctions

- page 50 règle CALL une remarque disant que cette règle doit être utilisée avec une autre qui va typer le corps de la fonction (mettre la ref) parce que sinon ça surprend
- 5.5 le fait de choisir une unique variable R t'oblige à ajouter une opération de suppression du R de la fonction appelante factice. Pourquoi ne pas générer des variables fraîches à partir du nom de la fonction + un identifiant unique ?
- lemme 5.1 cas fonction, à quoi ça sert d'introduire la notation t' alors qu'il n'apparaît pas dans une règle ?

Rq

(passe Sarah 17/01)

- 5.3 left-values la règle LV-VAR suppose que x n'apparaît qu'une fois dans Γ ou alors toujours accompagné du même type
- p53 le terme de dérivation (première phrase de la preuve) n'a jamais été expliqué ?
- lemme 5.1 constantes. Il faut expliquer à quel ensemble n et f appartiennent
- lemme 5.2 même remarque que 5.1 concernant n et f (plus confusion f float et f fonction)
- lemme 5.3 ça ne marche que si une variable n'apparaît qu'une fois dans le contexte ou avec toujours le même type (cf rem ci-dessous sur 5.3)

- il manque les preuves de 5.3 et 5.4
- lemme 5.4 $dom(\Gamma)$ a été défini quelque part ?
- théormèe 5.2 rappeler où a été défini l'évaluation d'une expression et dans quel cas elle produit des valeurs

E.4 Qualificateurs

- appliquer taint sur des sous-valeurs ?
- vTaint doit se propager aux accès de champ
- étendre l'état mémoire aux variables utilisateur

(passe Sarah)

- 6.1.1 français dans "qui représente qui contrôle sa valeur"
- 6.1.1 mettre une ref sur la description du noyau linux
- la traduction de taintage par teintage est incorrecte (et des fois du garde la même orthographe que celle anglaise avec le 'a') La traduction de tainted c'est plutôt sali, pollué. Tu peux aussi garder le terme anglais et le mettre en italique
- j'ai arrêté de lire à partir de 6.2 parce que le texte n'est pas vraiment clair

Sarah 13/02

- expliquer pourquoi on a besoin d'étendre l'expressivité de C_ML (rappeler le contexte des pointeurs usr et kernel et dire que ce sont les ptrs maitrises par le usr qui sont souillés)
- typo intro systme
- 6.1 phrase "tout d'abord..." pas très français non ?
- 5.1 étape "ensuite" à quoi correspond le phi ?
- figure 6.1 ça serait plus propre de tout remettre et surligner/colorer les nouveautés (à expliquer dans le texte)
- 6.2 pourquoi ne pas nommer cette extension de C_ML
- 3ème parag 6.3 choisir entre "qui représente" et "qui contrôle"
- idem figure 6.2 ça serait plus clair de remettre tout
- je me demande si ça ne serait pas plus clair de présenter d'abord l'extension des types ptr à KERNEL et USR et ensuite d'expliquer que ce qu'on cherche c'est de voir quelles expressions/valeurs sont TAINTED (donc à mettre après)

Extrait PLAS

The main distinctiveness is that instead of having a plain pointer type $t*$, our one is decorated with a type qualifier q. This annotation expresses who controls the value of the pointer. If the kernel controls the value of the pointer, then it cannot be abused. On the other hand, one has to be careful with user-controlled pointers, because the caller can abuse the kernel and access reserved memory. The only safe case when dereferencing such a pointer is if its value is outside the kernel's memory.

In order to avoid dangerous cases, we have to dynamically check that the destination of every user-controlled pointer is in userspace. Kernel pointers (that is to say, kernel-controlled

pointers) can be dereferenced without further check, but user pointers have to be manipulated with a restricted interface that will check whether their destination is in userspace.

As mentioned before, this is done using the following constructs :

$$\frac{\Gamma \vdash lv : t \qquad \Gamma \vdash e : t\ \textsc{User}\ *}{\Gamma \vdash lv \Leftarrow_U e : \textsc{Int}}\ (\textsc{GetU}) \qquad \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t\ \textsc{User}\ *}{\Gamma \vdash e_1 \Rightarrow_U e_2 : \textsc{Int}}\ (\textsc{PutU})$$

To add qualifiers to a type system, the rules of interest are those that manipulate pointers : dereferencing, pointer arithmetic and referencing (taking the address of a left-value).

Dereferencing the easiest one ; our goal is to authorize dereferencing only KERNEL pointers :

$$\frac{\Gamma \vdash lv : t*}{\Gamma \vdash *lv : t}\ (\textsc{Lv-Deref})$$

Pointer arithmetic can be done inside a USER or KERNEL memory zone. There is no concern of jumping from userspace to kernelspace, because pointer arithmetic is checked at runtime : if these operators overflow or are applied to a bad pointer (such as a pointer to an integer field), Ω_{ptr} is raised.

$$\frac{\boxplus \in \{+_p, -_p\} \qquad \Gamma \vdash e_1 : t* \qquad \Gamma \vdash e_2 : \textsc{Int}}{\Gamma \vdash e_1 \boxplus e_2 : t*}\ (\textsc{Ptr-Arith})$$

The reference case is trickier because a type qualifier has to be synthesized. Because it is created on the kernel stack, it has a KERNEL qualifier in all cases :

$$\frac{\Gamma \vdash lv : t}{\Gamma \vdash \&lv : t*}\ (\textsc{Addr})$$

The TAINT$(\cdot)$ operator turns a USER pointer into a KERNEL pointer. It is an important rule, because it is the only source of USER pointers in the type system. The return value of a function is emulated with a virtual left-value $\underline{R}$.

$$\frac{\Gamma \vdash \underline{R} \leftarrow e}{\Gamma \vdash \textsc{Return}(e)}\ (\textsc{Return}) \qquad \frac{\Gamma' = (\Gamma - \underline{R}), \vec{a} : \vec{t}, \underline{R} : t_r \qquad \Gamma' \vdash i}{\Gamma \vdash \text{fun}(\vec{a})\{i\} : \vec{t} \rightarrow t_r}\ (\textsc{Fun})$$

E.5 Implem

The language described above, as well as a type inference algorithm, have been implemented in OCaml as part of the Newspeak framework of program analysis[HL08]. It is released under the GNU Lesser General Public License, and is available on `http://penjili.org` (directory `src/ptrtype` in the distribution). Our implementation consists of the following steps.

but to analyze larger parts of the kernel, it may be necessary to define a “maximal” configuration file (which is impossible because of incompatibilities between some options).

E.6 Étude de cas

Le paramètre `data` provient de l’espace utilisateur via un appel système. Un appelant malveillant peut se servir de cette fonction pour lire la mémoire du noyau à travers le message d’erreur.

Le problème est modélisé de la façon suivante : on associe à chaque variable `x` un type de données `t`, ce que l’on note `x:t`. En plus des types présents dans le langage C, on ajoute une

distinction supplémentaire pour les pointeurs. D'une part, les pointeurs "noyau" (de type `t *`) sont créés en prenant l'adresse d'un objet présent dans le code source. D'autre part, les pointeurs "utilisateurs" (leur type est noté `t user*`) proviennent des interfaces avec l'espace utilisateur.

Il est sûr de déréférencer un pointeur noyau, mais pas un pointeur utilisateur. L'opérateur `*` prend donc un `t *` en entrée et produit un `t`.

Pour faire la vérification de type sur le code du programme, on a besoin de quelques règles. Tout d'abord, les types suivent le flot de données. C'est-à-dire que si on trouve dans le code `a = b`, `a` et `b` doivent avoir un type compatible. Ensuite, le qualificateur `user` est récursif : si on a un pointeur utilisateur sur une structure, tous les champs pointeurs de la structure sont également utilisateur. Enfin, le déréférencement s'applique aux pointeurs noyau seulement : si le code contient l'expression `*x`, alors il existe un type `t` tel que `x:t*` et `*x:t`.

Appliquons ces règles à l'exemple de la figure **??** : on suppose que l'interface avec l'espace utilisateur a été correctement annotée. Cela permet de déduire que `data:void user*`. En appliquant la première règle à la ligne 6, on en déduit que `info:struct drm_radeon_info user*` (comme en C, on peut toujours convertir de et vers un pointeur sur `void`).

Pour déduire le type de `value_ptr` dans la ligne 7, c'est la deuxième règle qu'il faut appliquer : le champ `value` de la structure est de type `uint32_t *` mais on y accède à travers un pointeur utilisateur, donc `value_ptr:uint32_t user*`.

À la ligne 8, on peut appliquer la troisième règle : à cause du déréférencement, on en déduit que `value_ptr:t *`, ce qui est une contradiction puisque d'après les lignes précédentes, `value_ptr:uint32_t user*`.

Si la ligne 3 était remplacée par l'appel à `copy_from_user`, il n'y aurait pas d'erreur de typage car cette fonction peut accepter les arguments `(uint32_t *, uint32_t user*, size_t)`.

Le principe de cette technique (associer des types aux valeurs puis restreindre les opérations sur certains types) peut être repris. Par exemple, si on définit un type "numéro de bloc" comme étant un nouvel alias de `int`, on peut considérer que multiplier deux telles valeurs est une erreur.

E.7 Conclusion

E.7.1 Future work

We showed that type theory can be a useful tool for verifying the absence of certain runtime properties. While adding static labels to variables seems to be a crude approximation of reality, in some cases it has enough power to capture real-world problems.

In this particular example, we work around C's lack of abstract types in order to disallow dereference for a certain class of pointers, distinguished by syntactic rules.

We defined an imperative language with an explicit stack, and described operational semantics for it modelling that of the C programming language. It includes a memory model that expresses the separation between user and kernel spaces present in most operating systems. We added a type system that is sound with respect to a property of isolation between this two memory spaces.

Finally, we demonstrate an implementation on this analysis on a bug that affected the Linux kernel.

A first step towards making this analysis more practical is to demonstrate its scalability by running it on larger fragments of the kernel.

There are also several places where we can improve significantly the expressivity of our type system. For example, our current type system is only monomorphic ; but it would make sense to generalize free qualifier variables in the type of global functions.

TABLE DES FIGURES

Liste des définitions

Liste des théorèmes et propriétés

RÉFÉRENCES WEB

[1] The Objective Caml system, documentation and user's manual – release 3.12
http ://caml.inria.fr/pub/docs/manual-ocaml/

[2] Haskell Programming Language – Official Website
http ://www.haskell.org/

[3] Python Programming Language – Official Website
http ://www.python.org/

[4] Perl Programming Language – Official Website
http ://www.perl.org/

[5] Sparse - a Semantic Parser for C
https ://sparse.wiki.kernel.org/index.php/Main_Page

[6] CIL - C Intermediate Language
http ://kerneis.github.com/cil/

[7] The C - - language
http ://www.cminusminus.org/

[8] Penjili project
http ://www.penjili.org/

BIBLIOGRAPHIE

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor (extended version). Research report 6138, INRIA, 2007. 29 pages. 7

[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. 25

[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. 22

[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. 25

[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. 25

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, third edition edition, November 2005. 100

[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*, pages 114–126, Savannah, GA, USA, January 2009. 22

[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. 7

[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. 26

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77 : Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM. 22

[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`.). 22

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005. 25

[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. 25

[CLO07] James Clause, Wanchun Li, and Alessandro Orso. Dytan : a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM. 131

[CMC08] Liqian Chen, Antoine Miné, and Patrick Cousot. A sound floating-point polyhedra abstract domain. In G. Ramalingam, editor, *Asian Symposium on Programming Languages and Systems (APLAS'08)*, volume 5356 of *LNCS*, pages 3–18, Bangalore, India, December 2008. Springer. 132

[CMP03] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly, 2003. 131

[CMP10] Dumitru Ceară, Laurent Mounier, and Marie-Laure Potet. Taint dependency sequences : A characterization of insecure execution paths based on input-sensitive cause sequences. In *ICST Workshops*, 2010. 131

[DDMPn10] 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. 107, 108

[Dij82] Edsger W. Dijkstra. Why numbering should start at zero. circulated privately, August 1982. 36

[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. 78

[DRS00] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV : Towards a realistic tool for statically detecting all buffer overflows in C, 2000. 26

[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. 78

[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. 132

[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. 33

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2) :235–271, September 1992. 138

[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. 132

[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. 25

[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. 132

[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. 25

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. 100, 108

[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. 25

[Har88] Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4) :36–38, October 1988. 4, 18

[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. 7, 77, 78, 142

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, October 1969. 25

[Int10] Intel, Santa Clara, CA, USA. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2010. 13, 101

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999. 6

[JW04] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004. 132

[KcS07] Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci.*, 174(7) :79–104, 2007. 131

[Ker81] Brian W. Kernighan. Why Pascal is not my favorite programming language. Technical report, AT&T Bell Laboratories, April 1981. 5

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988. 6, 132

[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. 6

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml : A notation for detailed design, 1999. 26

[LM12] Shuying Liang and Matthew Might. Hash-flow taint analysis of higher-order programs. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS '12, pages 8 :1–8 :12, New York, NY, USA, 2012. ACM. 25

[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. 131

[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. 25

[Mer03] J. Merrill. GENERIC and GIMPLE : a new tree representation for entire functions. In *GCC developers summit 2003*, pages 171–180, 2003. 6

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, December 1978. 5

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3) :12, May 2008. 132

[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. 25

[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. 7

[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. 39

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. 131

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 1996. 13

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 6

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003. 131

[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. 7

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61 :17–139, 2004. 137

[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. 22

[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. 21

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, January 1965. 81

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010. 131

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21 :2003, 2003. 131

[Spe05] Brad Spengler. grsecurity 2.1.0 and kernel vulnerabilities. *Linux Weekly News*, 2005. 22

[SRH95] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation, 1995. 22

[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. 6

[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. 132

[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. 132

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007. 2

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2 :245–271, 1992. 131

[TT93] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical report, 1993. 131

[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. 131

[TTL] Linus Torvalds, Josh Triplett, and Christopher Li. Sparse - a semantic parser for C. `https://sparse.wiki.kernel.org`. 132

[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. 25

[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. 33

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM. 131

[Wal00] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2000. 131

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1) :38–94, November 1994. 138