



---

# **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

Soutenue le xx-yy-2013 devant un jury composé de

aaa

aaa

aaa

aaa

aaa



Abstract

ABSTRACT

i



## Dédicace

# TABLE DES MATIÈRES

<b>Table des matières</b>	<b>iv</b>
<b>I Méthodes formelles pour la sécurité</b>	<b>1</b>
<b>1 Introduction</b>	<b>5</b>
<b>2 Systèmes d'exploitation</b>	<b>7</b>
2.1 Rôle d'un système d'exploitation . . . . .	7
2.2 Architecture Intel . . . . .	9
2.2.1 Assembleur . . . . .	9
2.2.2 Fonctions et conventions d'appel . . . . .	12
2.2.3 Tâches, niveaux de privilèges . . . . .	12
2.2.4 Mémoire virtuelle . . . . .	13
2.3 Cas de Linux . . . . .	14
2.3.1 Appels système . . . . .	15
2.4 Sécurité des appels système . . . . .	17
<b>3 État de l'art</b>	<b>21</b>
3.1 Taxonomie . . . . .	21
3.2 Méthodes syntaxiques . . . . .	22
3.3 Interprétation abstraite . . . . .	22
3.4 Typage . . . . .	26
3.5 Logique de Hoare . . . . .	30
<b>II Typage statique de langages impératifs</b>	<b>33</b>
<b>4 Un langage impératif : SAFESPEAK</b>	<b>37</b>
4.1 Notations . . . . .	37
4.2 But et comparaison à C . . . . .	42
4.3 Principes . . . . .	44
4.4 Syntaxe . . . . .	44
4.5 Définitions préliminaires . . . . .	45

## TABLE DES MATIÈRES

v

4.6	Mémoire . . . . .	47
4.7	Opérations sur les valeurs . . . . .	48
4.8	Opérations sur les états mémoire . . . . .	50
4.9	Accesseurs . . . . .	52
4.10	Contextes d'évaluation . . . . .	55
4.11	Expressions . . . . .	57
4.12	Instructions . . . . .	62
4.13	Erreurs . . . . .	64
4.14	Phrases . . . . .	65
4.15	Exécution . . . . .	65
4.16	Exemple : l'algorithme d'Euclide . . . . .	65
<b>5</b>	<b>Typage</b>	<b>69</b>
5.1	Principe . . . . .	69
5.2	Environnements et notations . . . . .	70
5.3	Expressions . . . . .	71
5.4	Instructions . . . . .	75
5.5	Fonctions . . . . .	75
5.6	Phrases . . . . .	76
5.7	Sûreté du typage . . . . .	76
<b>6</b>	<b>Qualificateurs de type</b>	<b>83</b>
6.1	Extensions noyau pour SAFESPEAK . . . . .	83
6.2	Insuffisance des types simples . . . . .	85
6.3	Extensions du système de types . . . . .	85
6.3.1	Propriété d'isolation mémoire . . . . .	87
6.4	Analyse de terminaison des chaînes C . . . . .	88
6.4.1	But . . . . .	88
6.4.2	Approche . . . . .	89
6.4.3	Annotation de <code>string.h</code> . . . . .	90
6.4.4	Typage des primitives . . . . .	92
6.4.5	Extensions au système de types . . . . .	92
6.4.6	Résultats . . . . .	92
<b>III</b>	<b>Expérimentation</b>	<b>93</b>
<b>7</b>	<b>Implantation</b>	<b>97</b>
7.1	Langages intermédiaires . . . . .	97
7.2	Newspeak . . . . .	99

7.3	Chaîne de compilation . . . . .	99
7.3.1	Prétraitement . . . . .	99
7.3.2	Compilation (levée des ambiguïtés) . . . . .	99
7.3.3	Annotations . . . . .	100
7.3.4	Implantation de l'algorithme de typage . . . . .	101
7.3.5	Algorithme d'unification . . . . .	109
<b>8</b>	<b>Étude de cas : un pilote de carte graphique</b>	<b>113</b>
8.1	Description du problème . . . . .	113
8.2	Principes de l'analyse . . . . .	114
8.3	Implantation . . . . .	115
8.4	Conclusion . . . . .	115
<b>9</b>	<b>Conclusion</b>	<b>117</b>
9.1	Limitations . . . . .	117
9.2	Perspectives . . . . .	117
<b>A</b>	<b>Code du module noyau</b>	<b>121</b>
<b>B</b>	<b>Règles d'évaluation</b>	<b>125</b>
<b>C</b>	<b>Règles de typage</b>	<b>129</b>
<b>D</b>	<b>Preuves</b>	<b>133</b>
D.1	Composition de lentilles . . . . .	133
D.2	Progrès . . . . .	134
D.3	Preuve du théorème 5.3 . . . . .	138
<b>E</b>	<b>PLAS (TODO)</b>	<b>141</b>
E.1	Introduction . . . . .	141
E.2	Related Work . . . . .	142
E.3	Core Language . . . . .	143
E.4	Evaluation . . . . .	146
E.5	Typing . . . . .	152
E.6	Type Safety . . . . .	154
E.7	Implementation . . . . .	155
E.8	Experimentation . . . . .	157
E.9	Limitations . . . . .	159
E.10	Conclusion . . . . .	160
	<b>Table des figures</b>	<b>161</b>



<i>TABLE DES MATIÈRES</i>	vii
<b>Liste des définitions</b>	<b>163</b>
<b>Liste des théorèmes et propriétés</b>	<b>163</b>
<b>Références web</b>	<b>165</b>
<b>Bibliographie</b>	<b>167</b>



## **Première partie**

# **Méthodes formelles pour la sécurité**



Intro partie I ici.

Le chapitre 1.

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



# 1

## INTRODUCTION





## SYSTÈMES D'EXPLOITATION

Le système d'exploitation est le programme qui permet à un système informatique d'exécuter d'autres programmes. Son rôle est donc capital et ses responsabilités multiples. Dans ce chapitre, nous allons voir 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.

Pour une description plus détaillée des rôles d'un système d'exploitation ainsi que plusieurs cas d'étude détaillés, on pourra se référer à [Tan07].

### 2.1 Rôle d'un système d'exploitation

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 (partage *temporel*) ; et chaque programme aura à sa disposition une partie de la mémoire principale, ou du disque dur (partage *spatial*).

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 utilisent la carte réseau les uns à la suite des autres. Ce sont des rôles du système d'exploitation.

Cela passe donc par une limitation des possibilités du programme : plutôt que de permettre n'importe quel type d'instruction, il communique avec le système d'exploitation. Celui-ci centralise donc les appels au matériel, ce qui permet d'abstraire certaines opérations.

Par exemple, si un programme veut copier des données depuis un cédérom vers la mémoire principale, il devra 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.

Si dans un autre cas il désire récupérer des données depuis une mémoire USB, il devrait 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 en pratique de la même manière. C'est pourquoi le système d'exploitation fournit les notions de fichier, lecteur, etc : le programmeur n'a plus qu'à utiliser des commandes de haut niveau ("monter un lecteur", "ouvrir un fichier", "lire dans un fichier") et selon le type de lecteur, le système d'exploitation effectuera les actions appropriées.

En résumé, un système d'exploitation est l'intermédiaire entre le logiciel et le matériel, et en particulier assure les rôles suivants :

- Gestion des processus : un système d'exploitation peut permettre d'exécuter plusieurs programmes à la fois. Il faut alors orchestrer ces différents processus et les séparer en terme de temps et de ressources partagées.
- Gestion de la mémoire : chaque processus, en plus du noyau, doit disposer d'un espace mémoire différent. C'est-à-dire qu'un processus ne doit pas pouvoir interférer avec un autre.
- Gestion des fichiers : les processus peuvent accéder à une hiérarchie de fichiers, indépendamment de la manière d'y accéder.
- Gestion des périphériques : le noyau étant le seul code ayant des privilèges, c'est lui qui doit communiquer avec les périphériques matériels.
- Abstractions : le noyau fournit aux programmes une interface unifiée, permettant de stocker des informations de la même manière sur un disque dur ou une clef USB (alors que l'accès se déroulera de manière très différente en pratique).

## 2.2 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].

### 2.2.1 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).

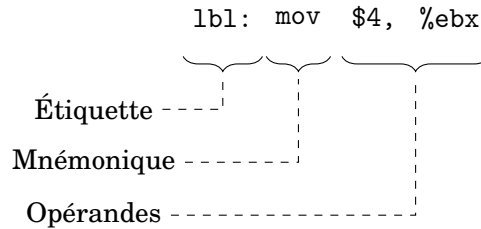
L'état du processeur est défini par la valeur de ses registres, qui sont des petites zones de mémoire interne (quelques bits chacun). Par exemple, la valeur du pointeur de pile est stockée dans ESP. Le registre EBP, couplé à ESP sert à adresser les variables locales et paramètres d'une fonction, comme ce sera expliqué dans la section 2.2.2.

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.

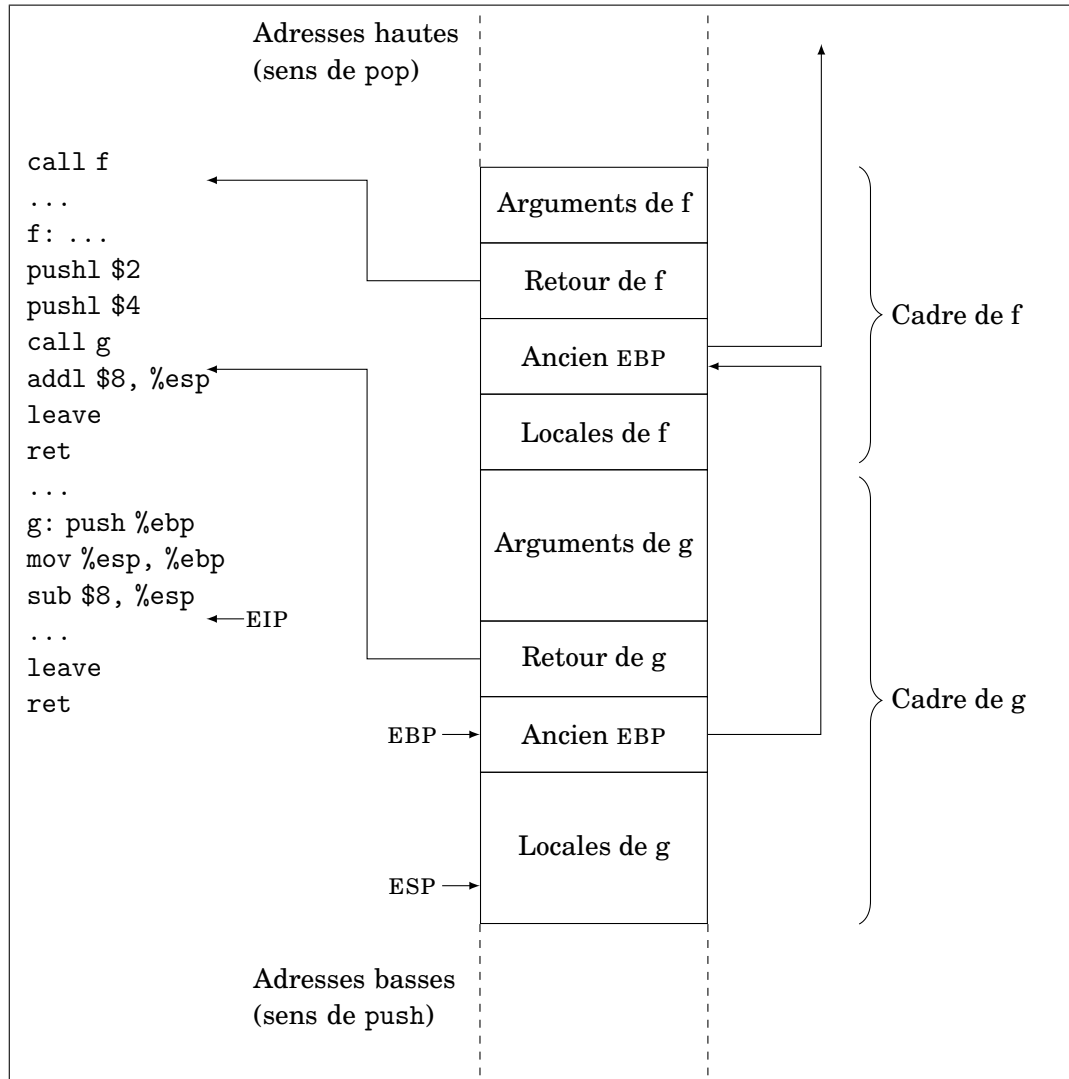


Ces opérandes peuvent être de plusieurs types :

- 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.

### 2.2.2 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 EDI 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 chaînés 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`.

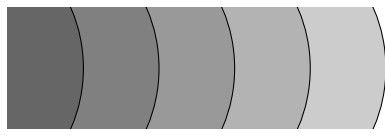
Au début de la fonction, les trois instructions permettent de sauvegarder l'ancienne valeur de EBP, faire pointer EBP à une endroit fixe dans le cadre de pile, puis allouer 8 octets de mémoire pour les variables locales.

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.

### 2.2.3 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.



**FIGURE 2.2:** Les différents *rings*. Seul le *ring* 0 a accès au hardware via des instructions privilégiées. Pour accéder aux fonctionnalités du noyau, les programmes utilisateur doivent passer par des appels système.

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* - figure 2.2) : 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.

#### 2.2.4 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 fuir ; 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 2.3).

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 2.4).

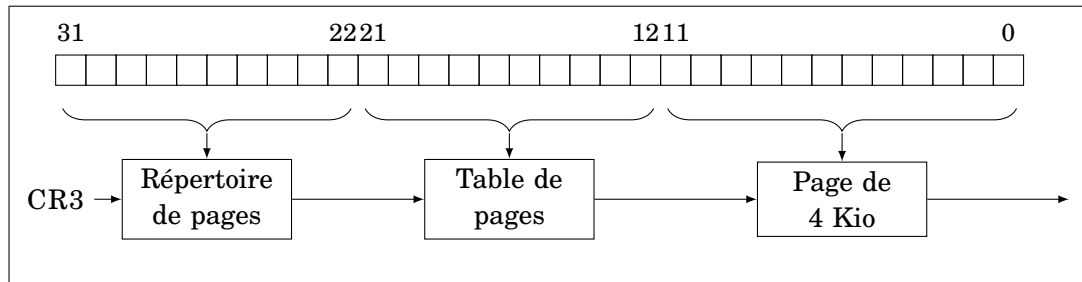


FIGURE 2.3: Implantation de la mémoire virtuelle

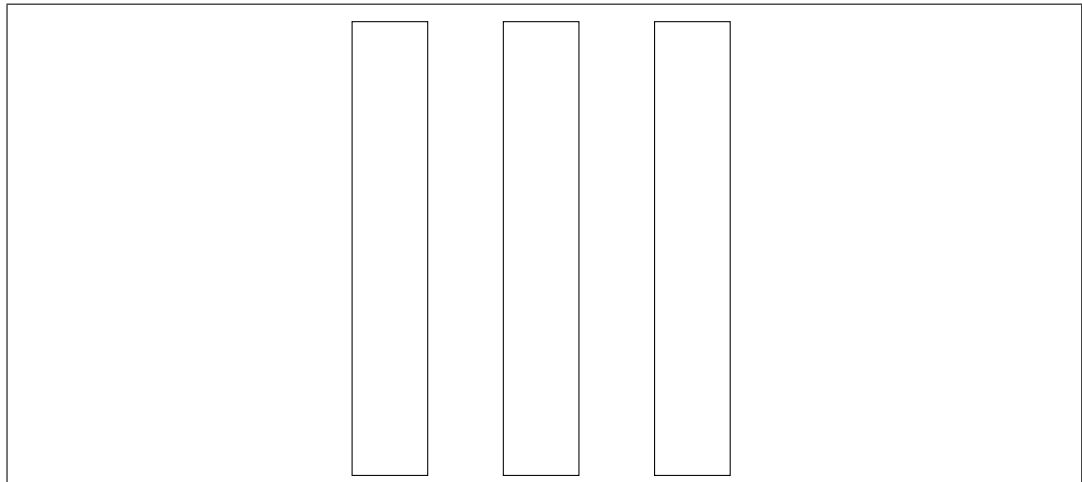


FIGURE 2.4: Mécanisme de mémoire virtuelle.

## 2.3 Cas de 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].

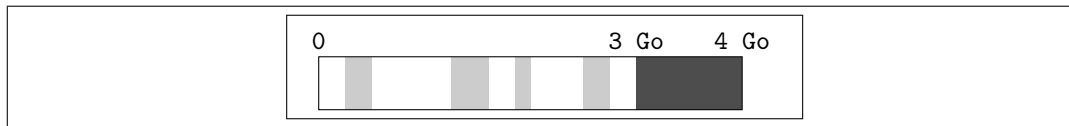
Deux rings sont utilisés : en *ring* 0, le code noyau et en *ring* 3, le code utilisateur.

Une notion de tâche similaire à celle décrite dans la section 2.2.3 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 2.5), et donc chacun un ensemble de tables de pages et une valeur de CR3 associée. Au moment de changer le processus en cours, l'ordonnanceur charge donc le CR3 du nouveau processus.





**FIGURE 2.5:** L'espace d'adressage d'un processus. En gris clair, les zones accessibles à tous les niveaux de privilèges : code du programme, bibliothèques, tas, pile. En gris foncé, la mémoire du noyau, réservée au mode privilégié.

Les 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.

### 2.3.1 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<sup>1</sup>, 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)
    RINGO_INT_FRAME                # can't unwind into user space anyw
    pushl %eax                    # save orig_eax
    CFI_ADJUST_CFA_OFFSET 4
    SAVE_ALL
    GET_THREAD_INFO(%ebp)

                                # system call tracing in operation / emula
    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 :

```

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

```

---

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*.

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.

## 2.4 Sécurité des appels système

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

Comme pour toutes les interfaces, on peut être plus ou moins fin. D'un côté, une interface pas assez fine serait trop restrictive et ne permettrait pas d'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 2.3.1.

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 2.6).

Considérons une implémentation naïve de cet appel système qui écrirait directement à l'adresse pointée. La figure 2.7(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.

Si l'utilisateur passe un pointeur dont la valeur est supérieure à `0xc0000000` (figure 2.7(b)), que se passe-t-il ? Comme le déréférencement est fait dans le code du

```

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 2.6: Appel de gettimeofday

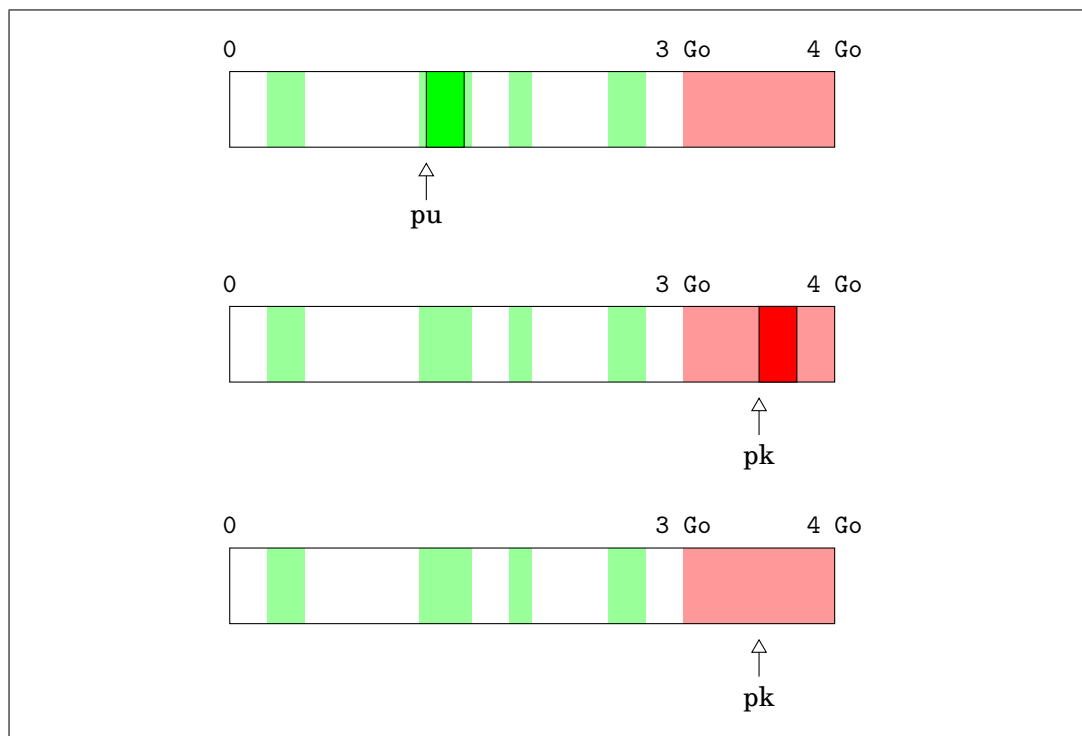


FIGURE 2.7: Zones mémoire

```
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 2.8:** Implantation de l'appel système gettimeofday

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. Cela 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 2.7(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 l'appel système `gettimeofday` fait appel à celle-ci (figure 2.8).

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.



## 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<sup>+</sup>09, PTS<sup>+</sup>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. 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. On peut en citer quelques uns.

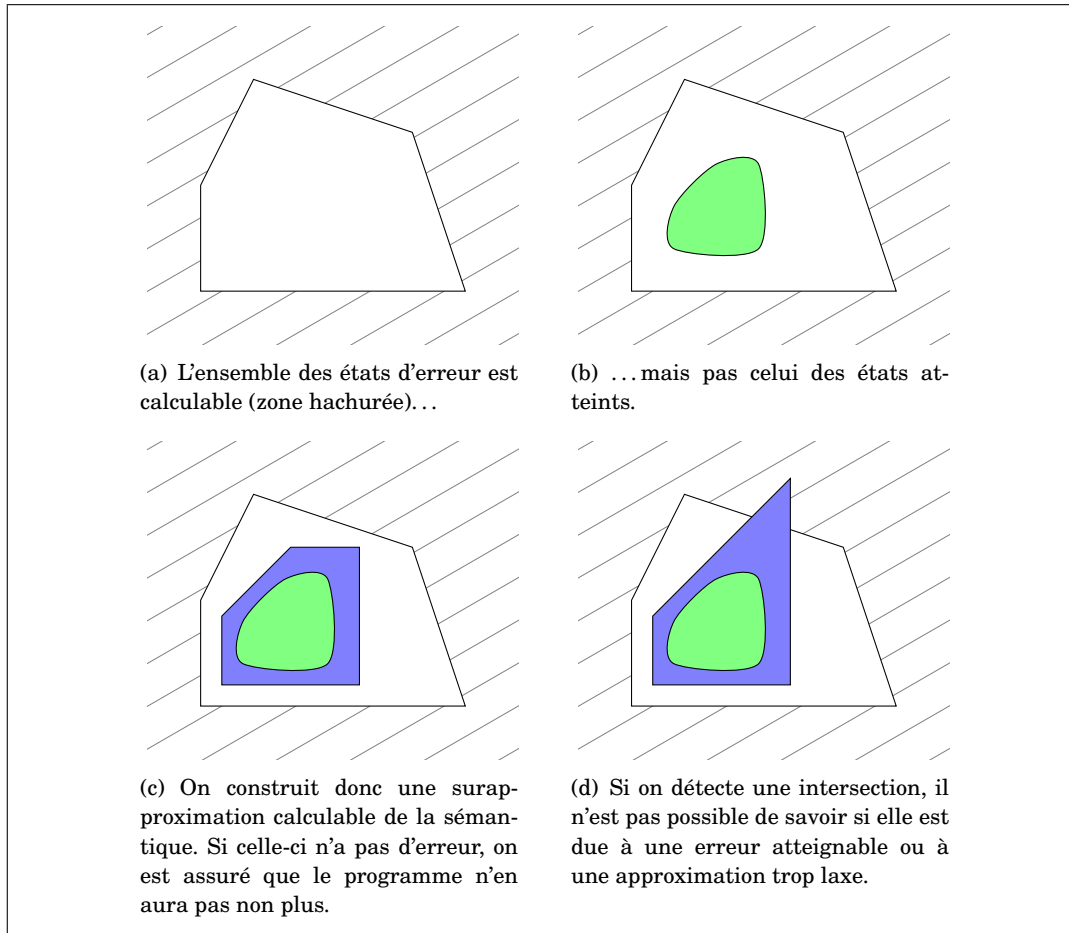
**Le domaine des signes** capture uniquement le signe des variables (figure 3.2).

**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





**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)).



**FIGURE 3.2:** Domaine des signes

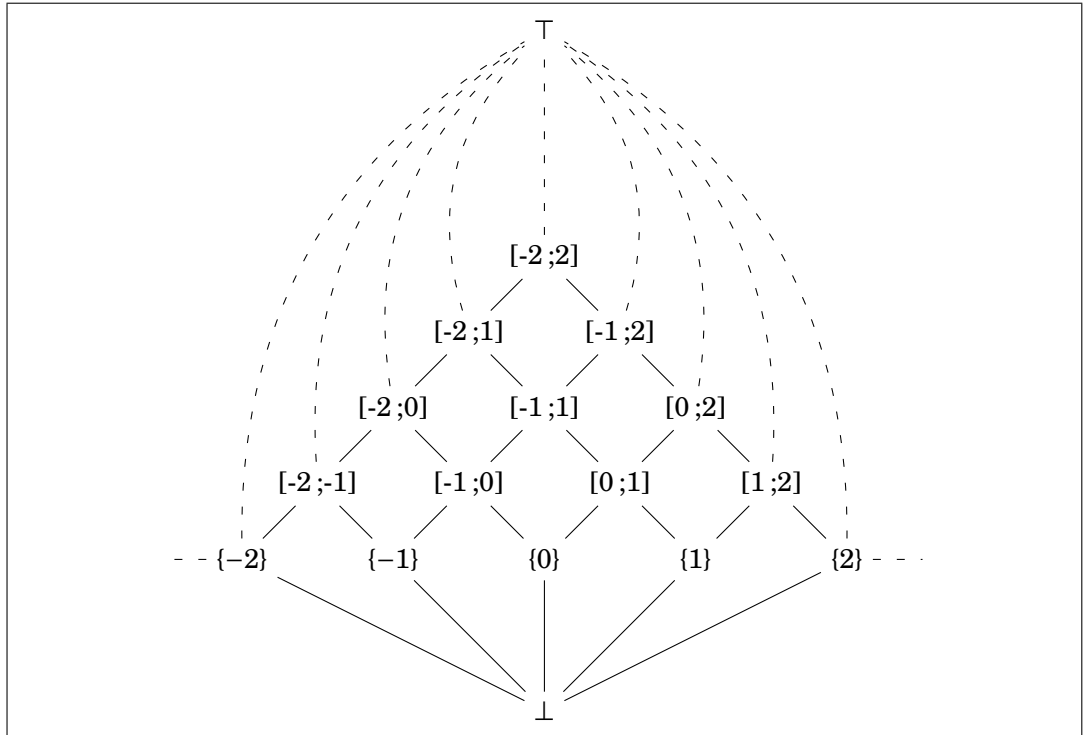


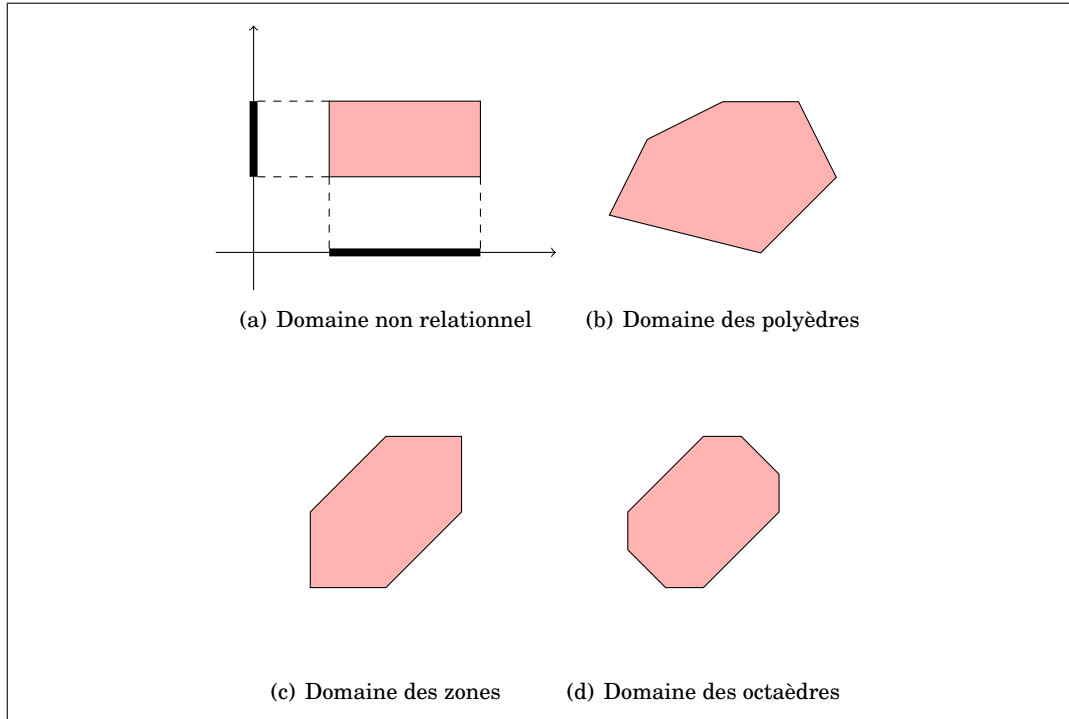
FIGURE 3.3: Domaine des intervalles

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

**FIGURE 3.4:** Quelques domaines abstraits

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*)  $\nabla$  est employé. L'idée est qu'une fois qu'on a calculé quelques termes d'une suite croissante, on peut réaliser une projection de celle-ci. Par exemple, dans le domaine des intervalles,  $[0;2] \nabla [0;3] = [0; +\infty[$ . On atteint alors un point fixe mais qui est plus grand que celui qu'on aurait obtenu sans cette accélération : on perd en précision. Pour en gagner, on peut redescendre sur le treillis des points fixe avec une suite d'itérations décroissantes [Gra92]. 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'ingénierie 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<sup>+</sup>05, CCF<sup>+</sup>09], CGS [VB04], Coverity [BBC<sup>+</sup>10], ou de Penjili.

Une interprétation abstraite est par construction sûre et incomplète, donc ce qui

sépare un bon analyseur d'un mauvais est sa précision. Dans le cas du langage C, de nombreuses constructions rendent imprécises les analyses :

- 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].
- 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.

### 3.4 Typage

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, ajouter deux adresses, ou déréférencer le résultat d'une division sont des comportements qu'on voudrait pouvoir empêcher.

En un mot, le but du typage est de classer 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". Symboliquement :

```
fonction addition(x, y) {
  if ( (etiquette_type(x) != type_entier)
      || (etiquette_type(y) != type_entier)
    ) {
    erreur ("L'addition est seulement définie pour deux entiers")
  } else {
    vx = valeur(x)
    vy = valeur(y)
    v = addition_machine(vx, vy)
    return Valeur(valeur=v, type=type_entier)
  }
}
```

Le langage Python [3] utilise cette stratégie, qui est illustrée dans la session interactive de la figure 3.5.

**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 un contrat comme "si deux entiers sont passés, la fonction renverra un entier sans provoquer d'erreur". 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  $(\text{int} * \text{int}) \rightarrow \text{int}$ . Les règles permettant de vérifier le typage sont par exemple les suivantes :

- une constante entière est toujours de type int.

```

>>> a = 3
>>> c = 4.5
>>> type(a)
<type 'int'>
>>> a+a
6
>>> type(a+a)
<type 'int'>
>>> a+c
7.5
>>> type(a+c)
<type 'float'>
>>> def d(x):
...     return 2*x
...
>>> type(d)
<type 'function'>
>>> a+d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'function'

```

FIGURE 3.5: Session Python présentant le typage dynamique

- si  $f$  a pour type  $(t_1, \dots, t_n) \rightarrow t$ , que  $e_1$  a pour type  $t_1, \dots$ , que  $e_n$  a pour type  $t_n$ , alors  $f(e_1, \dots, e_n)$  a pour type  $t_n$
- si en considérant que  $e_1$  a pour type  $t_n, \dots$ , et que  $e_n$  a pour type  $t_n$ , on arrive à typer le corps de  $f$  et que sa valeur de retour a alors pour type  $t$ , alors  $f$  a pour type  $(t_1, \dots, t_n) \rightarrow t$

Cet ensemble de règles, une fois formalisé et implanté, est en général assez efficace pour vérifier la correction d'un programme.

**Typage fort ou faible :** Si un système de types statique permet d'éliminer totalement la nécessité de réaliser des tests de typage, on dit qu'il est *fort*. Mais ce n'est que rarement le cas. En effet, il peut y avoir des constructions au sein du langage qui permettent de contourner le système de types, comme un opérateur de transtypage3.6. À l'exécution, une erreur de types est levée :

```

Exception in thread "main" java.lang.ClassCastException:
  java.lang.Integer cannot be cast to java.lang.Float

```

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

**FIGURE 3.6:** Transtypage en Java

```
let rec append lx ly =
  match lx with
  | [] -> ly
  | x::xs -> x::append xs ly
```

**FIGURE 3.7:** Fonction de concaténation de listes en OCaml.

```
at Cast.main(Cast.java:5)
```

**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 ?

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].

Une amélioration peut être de générer à la compilation une fonction qui aurait le même code mais des annotations de typage différentes : on obtient alors plusieurs copies de la fonction qui sont chacune spécialisées pour un type en particulier. C'est la solution retenue par le langage C++.

En fait, si à l'exécution toutes les données sont représentées de manière uniforme, il n'est pas nécessaire de créer plusieurs copies de la fonction.

Par exemple, la fonction de la figure 3.7 n'opère que sur la structure du type liste (en utilisant ses constructeurs [] et (: :)) ainsi que le filtrage) : les éléments de lx et ly ne sont pas manipulés à part pour les transférer dans le résultat.

Elle peut donc être typée avec n'importe quelle type  $a$   $list \rightarrow a\ list \rightarrow a\ list$ . Pour introduire cette généralité, on modifie le système de types pour transformer :

$$\forall a. \text{append} : a\ list \rightarrow a\ list \rightarrow a\ list$$

en :

$$\text{append} : \forall a. a\ list \rightarrow a\ list \rightarrow a\ list$$

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

**Expressivité :** les systèmes de types les plus simples expriment des contrats essentiellement liés à la sûreté d'exécution, pour ne pas utiliser des valeurs de types incompatibles entre eux. Mais il est possible d'étendre le langage avec des annotations plus riches : par exemple en vérifiant statiquement que des listes ne sont pas vides [KcS07], ou dans le domaine de la sécurité, d'empêcher des fuites d'information [LZ06].

**Qualificateurs de types :** Dans le cas particulier des vulnérabilités liées à une mauvaise utilisation de la mémoire, les développeurs du noyau Linux ont ajouté un système d'annotations au code source. Un pointeur peut être décoré d'une annotation `__kernel` ou `__user` selon s'il est sûr ou pas. Celle-ci sont ignorées par le compilateur, mais un outil d'analyse statique ad-hoc nommé Sparse [♣<sup>5</sup>] peut être utilisé pour détecter les cas les plus simples d'erreurs.

Ce système d'annotations sur les types a été formalisé sous le nom de *qualificateurs de types* : 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 implantées dans l'outil CQual [FFA99, STFW01, FTA02, JW04, FJKA06]. .

### 3.5 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  $\mathcal{L}$ .

Chaque instruction (ou *commande*)  $c$  est annotée d'une pré-condition  $P$  et d'une post-condition  $Q$ , ce que l'on note  $\{P\} c \{Q\}$ . Cela signifie que si  $P$  est vérifiée et que l'exécution de  $c$  se termine<sup>1</sup>, alors  $Q$  sera vérifiée.

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

$$\frac{\{P\} c_1 \{Q\} \quad \{Q\} c_2 \{R\}}{\{P\} c_1; c_2 \{R\}} \text{ (HOARE-SEQ)}$$

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

---

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



$$\frac{\vdash_{\mathcal{L}} P \Rightarrow P' \quad \{P\} c \{Q\} \quad \vdash_{\mathcal{L}} Q' \Rightarrow Q}{\{P'\} c \{Q'\}} \text{ (HOARE-CONSEQUENCE)}$$

Il est alors possible d'annoter le programme avec ses invariants formalisés de manière explicite dans  $\mathcal{L}$ . 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 :  $\{T\} c \{T\}$  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].

---

**TODO**

- IA :
  - citer pour penjili[XACH07]
  - difficultés :
    - récursion
  - produit réduit
  - CIL, autres front end
  - polyspace ?
  - APRON ?
- 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 dynamique [SAB10] et Perl ?.
- 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],
- Frama-C ?
- CCurred ?

## **Deuxième partie**

# **Typage statique de langages impératifs**



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	☐	☐	☑



## UN LANGAGE IMPÉRATIF : SAFESPEAK

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’éléments 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 :

$$L = \text{fix}(L')$$

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

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

|  $n :: l$

Construction de liste

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

### Arbres binaires

bi-

$a ::= F$

Feuille

|  $N(a_1, n, a_2)$

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  $\cdot \rightarrow \cdot$  entre états de l'interpréteur<sup>1</sup>.

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 :

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

---

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



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 prémisses, 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 :

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

## Lentilles

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

**Définition 4.1** (Lentille). *Étant donnés deux ensembles  $R$  et  $A$ , une lentille  $\mathcal{L} \in \text{LENS}_{R,A}$  (ou accessor) 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 constituée des opérations suivantes :*

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

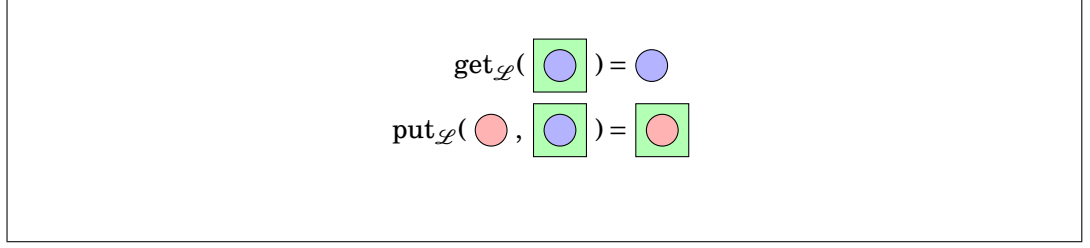
telles que pour tous  $a \in A, a' \in A, r \in R$  :

$$\begin{aligned} \text{put}_{\mathcal{L}}(\text{get}_{\mathcal{L}}(r), r) &= r & \text{(GetPut)} \\ \text{get}_{\mathcal{L}}(\text{put}_{\mathcal{L}}(a, r)) &= a & \text{(PutGet)} \\ \text{put}_{\mathcal{L}}(a', \text{put}_{\mathcal{L}}(a, r)) &= \text{put}_{\mathcal{L}}(a', r) & \text{(PutPut)} \end{aligned}$$

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

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

Une illustration se trouve dans la figure 4.1.

**FIGURE 4.1:** Fonctionnement d'une lentille

**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  $\Delta$  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.

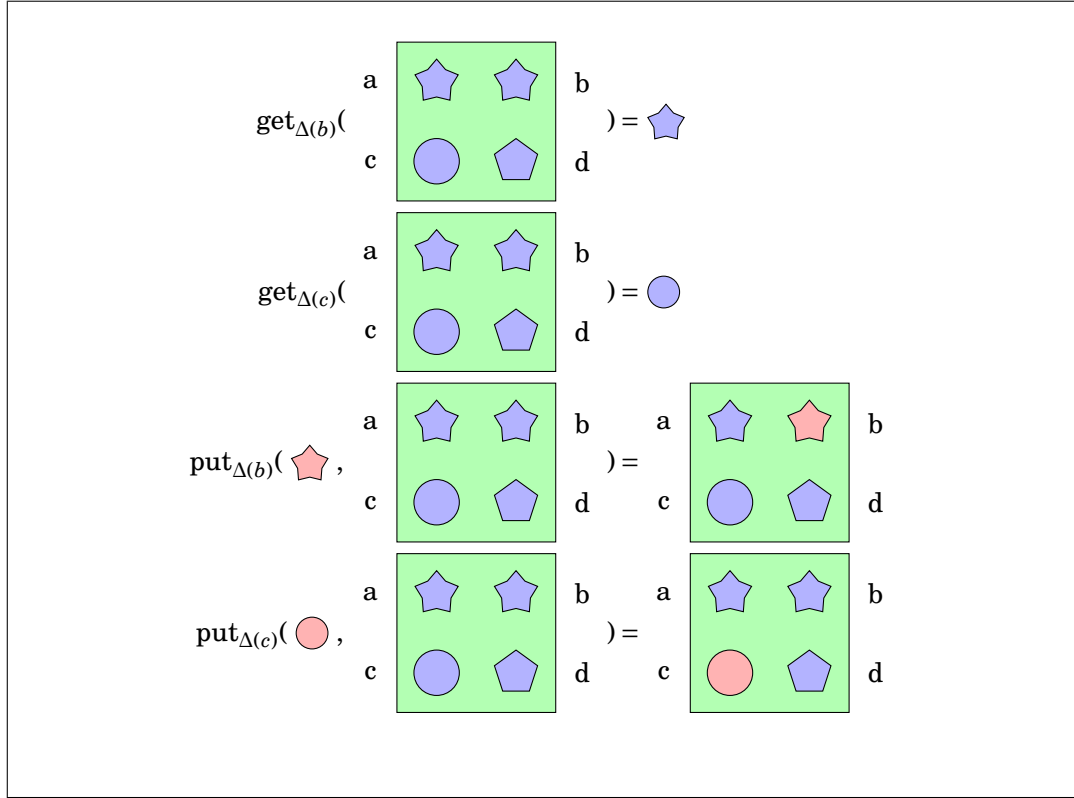


FIGURE 4.2: Fonctionnement d'une lentille indexée

**Exemple 4.2** (Lentille "n<sup>e</sup> élément d'un tuple"). Soient  $n \in \mathbb{N}$ , et  $n$  ensembles  $E_1, \dots, E_n$ . Pour tout  $i \in [1; n]$ , on définit :

$$g_i((x_1, \dots, x_n)) = x_i$$

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

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

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

$$= 1$$

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

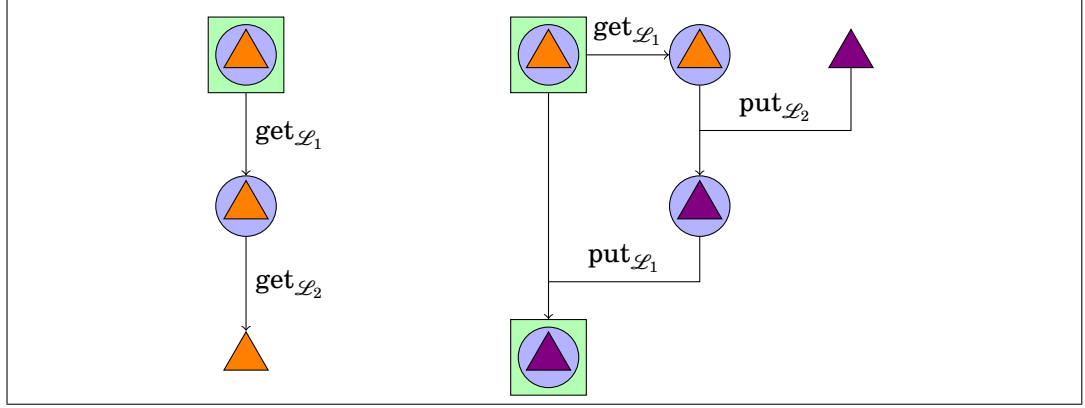


FIGURE 4.3: Composition de lentilles

$$=(9,2,1,5,3)$$

**Définition 4.3** (Composition de lentilles). Soient  $\mathcal{L}_1 \in \text{LENS}_{A,B}$  et  $\mathcal{L}_2 \in \text{LENS}_{B,C}$ .

La composition de  $\mathcal{L}_1$  et  $\mathcal{L}_2$  est la lentille  $\mathcal{L} \in \text{LENS}_{A,C}$  définie de la manière suivante :

$$\begin{aligned} \text{get}_{\mathcal{L}}(r) &= \text{get}_{\mathcal{L}_2}(\text{get}_{\mathcal{L}_1} r) \\ \text{put}_{\mathcal{L}}(a, r) &= \text{put}_{\mathcal{L}_1}(\text{put}_{\mathcal{L}_2}(a, \text{get}_{\mathcal{L}_1} r), r) \end{aligned}$$

On notera alors  $\mathcal{L} = \mathcal{L}_1 \gg \mathcal{L}_2$ .

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 But et comparaison à C

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 correspondent à des zones de mémoire contigües, homogènes (dans le cas des tableaux) ou hétérogènes (dans le cas des structures).

Une des spécificités de C est qu’il expose au programmeur la notion de pointeur, c’est à dire des variables qui représentent directement une adresse en mémoire. Les

pointeurs peuvent être typés (on garde une indication sur le type de l'objet stocké à cette adresse) ou non typés.

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.

**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 construire 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 ( $\Omega_{array}$ ) 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<sup>3</sup>.

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

---

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].

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.

**Flôt de contrôle :** il repose sur les constructions "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  $\text{If}(e)\{i\} = \text{If}(e)\{i\}\text{ELSE}\{\text{PASS}\}$ .

Les opérateurs sont donnés dans la figure 4.6.

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

FIGURE 4.4: Syntaxe - expressions

## 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 :

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

<b>Instructions</b>	$i ::= \text{PASS}$	Instruction vide
	$i; i$	Séquence
	$e$	Expression
	$\text{DECL } x = e \text{ IN } \{i\}$	Déclaration de variable
	$\text{IF}(e)\{i\}\text{ELSE}\{i\}$	Alternative
	$\text{WHILE}(e)\{i\}$	Boucle
	$\text{RETURN}(e)$	Retour de fonction
<b>Fonctions</b>	$f ::= \text{fun}(x_1, \dots, x_n)\{i\}$	Arguments, corps
<b>Phrases</b>	$p ::= x = e$	Variable globale
	$e$	Évaluation d'expression
<b>Programme</b>	$P ::= (p_1, \dots, p_n)$	Phrases

FIGURE 4.5: Syntaxe - instructions

<b>Opérateurs binaires</b>	$\boxplus ::= +, -, \times, /, \%$	Arithmétique entière
	$+, -, \times, /.$	Arithmétique flottante
	$+_p, -_p$	Arithmétique de pointeurs
	$=, \neq, \leq, \geq, <, >$	Comparaisons
	$\&,  , ^$	Opérateurs bit à bit
	$\&\&,   $	Opérateurs logiques
	$\ll, \gg$	Décalages
<b>Opérateurs unaires</b>	$\boxminus ::= +, -$	Arithmétique entière
	$+, -, .$	Arithmétique flottante
	$\sim$	Négation bit à bit
	$!$	Négation logique

FIGURE 4.6: Syntaxe - opérateurs



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 contextes d'appel<sup>4</sup>. 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 ambiguë une variable, on note simplement  $x$  la variable globale nommée  $x$ , et  $(n, x)$  la variable locale nommée  $x$  dans le  $n^{\text{e}}$  cadre de pile<sup>5</sup>.

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  $\varphi$ . 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<sup>e</sup> cadre de pile.

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

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

---

4. qu'on appellera donc aussi cadres de pile pour *stack frames* en anglais

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

Les valeurs entières sont les entiers signés sur 32 bits, c'est à dire entre  $-2^{31}$  à  $2^{31} - 1$ . Mais ce choix est arbitraire : 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].

- $\varphi$  : 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<sup>e</sup> cadre de pile.
- $\{l_1 : v_1; \dots; l_n : v_n\}$  : une structure. Comme précédemment, on note  $\{\cdot\}$  pour dénoter les valeurs.
- $[v_1; \dots; v_n]$  : un tableau. Pareillement,  $[\cdot]$  permet de désigner les valeurs. Par exemple, si  $x$  vaut 2 et  $y$  vaut 3, l'expression  $[x, y]$  s'évaluera en valeur  $[2, 3]$
- $\hat{f}$  : 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).
- $\Omega$  : une erreur. Par exemple le résultat de  $5/0$  est  $\Omega_{div}$ .

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  $\Xi$  ; 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  $\langle 3, ([, [x, 3]) \rangle \rightarrow \langle \hat{3}, ([, [x, 3]) \rangle$ . 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  $\Omega$ . La propagation des erreurs est détaillée dans la section 4.13.

## 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  $\Omega_{div}$  est levée.
- les versions "pointées"  $+. , -. , \times. , /. ,$  sur les flottants

<b>Valeurs</b>	$v ::= \hat{c}$	Constante
	$\hat{\&}\varphi$	Référence mémoire
	$\{l_1 : \widehat{v_1}; \dots; l_n : \widehat{v_n}\}$	Structure
	$[\widehat{v_1}; \dots; \widehat{v_n}]$	Tableau
	$\hat{f}$	Fonction
	$\Omega$	Erreur
<b>Chemins</b>	$\varphi ::= a$	Adresse
	$*\varphi$	Déréférencement
	$\varphi.l$	Accès à un champ
	$\varphi[n]$	Accès à un élément
<b>Adresses</b>	$a ::= (n, x)$	Variable locale
	$(x)$	Variable globale
<b>Erreur</b>	$\Omega ::= \Omega_{array}$	Débordement de tableau
	$\Omega_{ptr}$	Erreur de pointeur
	$\Omega_{div}$	Division par zéro
	$\Omega_{field}$	Erreur de champ

FIGURE 4.7: Valeurs

<b>Pile</b>	$s ::= []$	Pile vide
	$\{x_1 \mapsto v_1; \dots; x_n \mapsto v_n\} :: s$	Ajout d'un cadre
<b>État mémoire</b>	$m ::= (s, \{x_1 \mapsto v_1; \dots; x_n \mapsto v_n\})$	Pile, globales
<b>État d'inter- préteur</b>	$\Xi ::= \langle e, m \rangle$	Expression, mémoire
	$\langle i, m \rangle$	Instruction, mémoire
	$\Omega$	Erreur

FIGURE 4.8: Composantes d'un état mémoire

- les opérations d'arithmétique de pointeur  $+_p$  et  $-_p$  qui à un chemin mémoire et un entier associent un chemin mémoire.
- les opérations d'égalité  $=$  et  $\neq$ . L'égalité entre entiers ou entre flottants est immédiate. Deux valeurs composées (tableaux ou structures) sont égales si elles ont la même "forme" (même taille pour les tableaux, et même champs pour les structures) et que toutes leurs sous-valeurs sont égales deux à deux. Les références mémoire  $\varphi$  sont égales lorsque les chemins qu'ils décrivent sont syntaxiquement égaux<sup>6</sup>.
- les opérations de comparaison  $\leq, \geq, <, >$  sont définies avec leur sémantique habituelle sur les entiers et les flottants. Sur les références mémoires, elles sont définies dans le cas où les deux opérandes sont de la forme  $\varphi[\cdot]$  par :  $\varphi[n] \boxplus \varphi[m] \stackrel{\text{def}}{=} n \boxplus m$ . Dans les autres cas, l'erreur  $\Omega_{ptr}$  est renvoyée.
- les opérateurs bit à bit sont définis sur les entiers.  $\&$ ,  $|$  et  $\wedge$  représentent respectivement la conjonction, la disjonction et la disjonction exclusive (XOR).
- des versions logiques de la conjonction ( $\&\&$ ) et de la disjonction ( $||$ ) sont également présentes. Leur sémantique est donnée par le tableau suivant :

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

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

## 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.*

---

6. 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.

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

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  $c = ((x_1, v_1), \dots, (x_n, v_n))$  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 :

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

Sans cette règle, examinons le programme suivant :

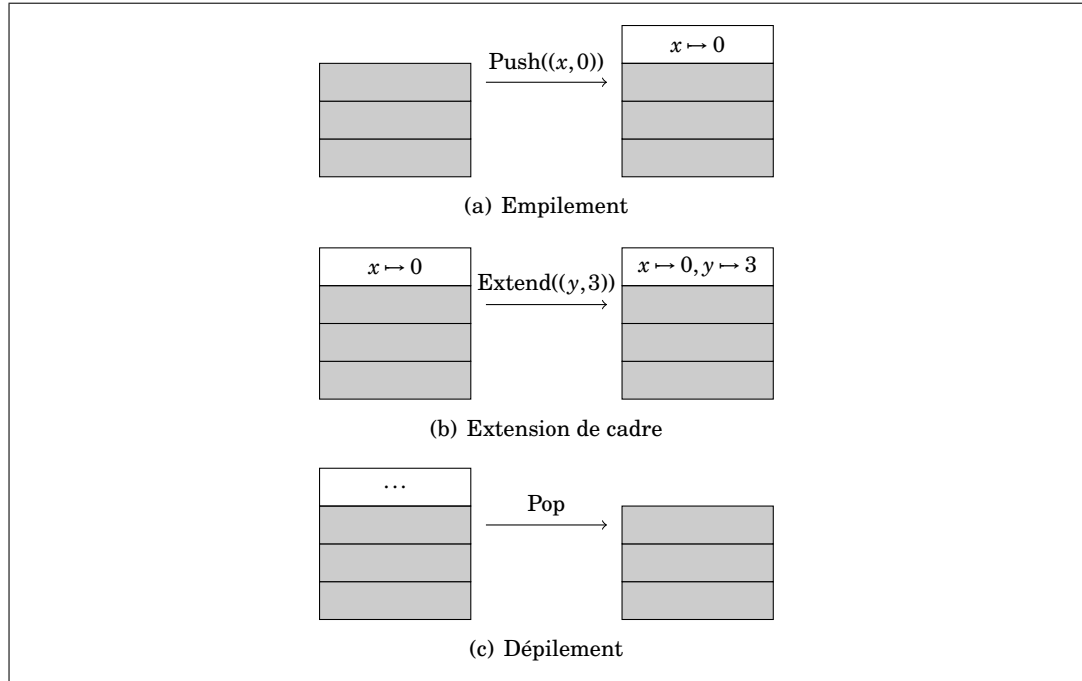


FIGURE 4.9: Opérations de pile

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 :

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

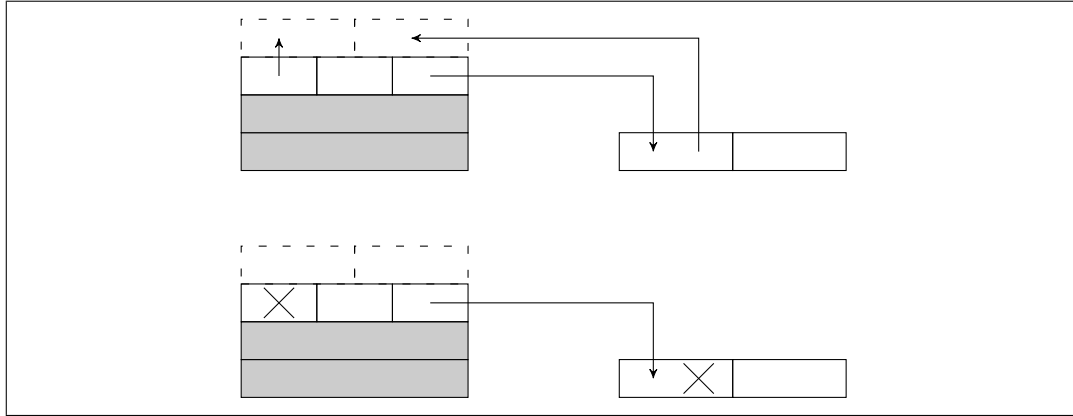


FIGURE 4.10: Nettoyage d'un cadre de pile

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

**Définition 4.7** (Accès par adresse). *Les états mémoire sont constitués des listes d'association (nom, valeur).*

*L'accessneur 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  $m = (s, g)$ ,*

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

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'accessneur  $[\cdot]_L$  permet de lire et de modifier un champ de ces valeurs.*

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

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

**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.

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

**Définition 4.10** (Accès par chemin). L'accès par chemin  $\Phi$  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) :

$$\Phi(\varphi.l) = \Phi(\varphi) \gg L(l)$$



$$\Phi(\varphi[i]) = \Phi(\varphi) \ggg I(i)$$

*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  $m[\varphi]_{\varphi}$ )
- construire une valeur altérée (en appliquant par exemple  $[l \leftarrow v]_L$ )
- affecter cette valeur au même chemin (le  $m[\varphi \leftarrow \dots]_{\varphi}$  externe)

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

## 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

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

FIGURE 4.11: Contextes d'exécution

- 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 :

$$\begin{array}{c}
 \frac{\langle e, m \rangle \rightarrow \langle e', m' \rangle}{\langle C \langle e \rangle, m \rangle \rightarrow \langle C \langle e' \rangle, m' \rangle} \text{ (CTX)} \qquad \frac{\langle lv, m \rangle \rightarrow \langle lv', m' \rangle}{\langle C_L \langle lv \rangle_L, m \rangle \rightarrow \langle C_L \langle lv' \rangle_L, m' \rangle} \text{ (CTX-LV)} \\
 \\
 \frac{\langle i, m \rangle \rightarrow \langle i', m' \rangle}{\langle C_I \langle i \rangle_I, m \rangle \rightarrow \langle C_I \langle i' \rangle_I, m' \rangle} \text{ (CTX-INSTR)}
 \end{array}$$

Enfin, le troisième revient à définir 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'exactlyement 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  $v = v_1 \hat{\boxplus} v_2$  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  $\langle e_1, m \rangle \rightarrow \langle v_1, m' \rangle$ .
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  $\langle e_1 \boxplus e_2, m \rangle \rightarrow^* \langle v_1 \boxplus e_2, m' \rangle$
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 :  $\langle e_2, m' \rangle \rightarrow \langle v_2, m'' \rangle$ .
4. Appliquons la règle CTX à 3. avec  $C = v_1 \boxplus \bullet$ . On obtient  $\langle v_1 \boxplus e_2, m' \rangle \rightarrow^* \langle v_1 \boxplus v_2, m' \rangle$ .
5. En combinant les résultats de 2. et 4. on en déduit que  $\langle e_1 \boxplus e_2, m \rangle \rightarrow^* \langle v_1 \boxplus v_2, m'' \rangle$ .
6. D'après la règle EXP-BINOP,  $\langle v_1 \boxplus v_2, m'' \rangle \rightarrow^* \langle v_1 \hat{\boxplus} v_2, m'' \rangle$
7. D'après 5. et 6., on a par combinaison  $\langle e_1 \boxplus e_2, m \rangle \rightarrow^* \langle v, m'' \rangle$  en posant  $v = v_1 \hat{\boxplus} v_2$ .

## 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 :*

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

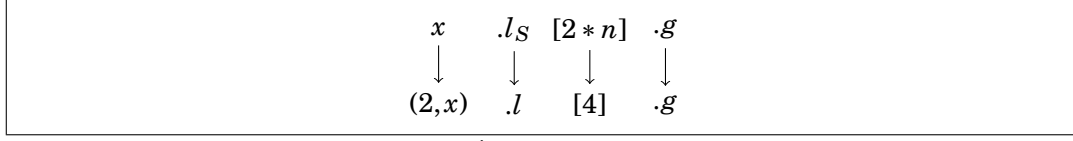
**Définition 4.12** (Évaluation d'une left-value). *L'évaluation d'une left-value  $lv$  produit un "chemin"  $\varphi$  dans une variable, qui est en fait équivalent à une left-value dont toutes les sous-expressions (d'indices) ont été évaluées.*

On note :

$$\langle lv, m \rangle \rightarrow \langle \varphi, m' \rangle$$

$$\begin{aligned}
& \bullet \langle e_0 \rangle = e_0 \\
& (C \boxplus e) \langle e_0 \rangle = C \langle e_0 \rangle \boxplus e \\
& (v \boxplus C) \langle e_0 \rangle = v \boxplus C \langle e_0 \rangle \\
& (\boxplus C) \langle e_0 \rangle = \boxplus C \langle e_0 \rangle \\
& (*C) \langle e_0 \rangle = *C \langle e_0 \rangle \\
& (C.l_S) \langle e_0 \rangle = C \langle e_0 \rangle . l_S \\
& (\varphi[C]) \langle e_0 \rangle = \varphi[C \langle e_0 \rangle] \\
& (C[e]) \langle e_0 \rangle = C \langle e_0 \rangle [e] \\
& (C \leftarrow e) \langle e_0 \rangle = C \langle e_0 \rangle \leftarrow e \\
& (\varphi \leftarrow C) \langle e_0 \rangle = \varphi \leftarrow C \langle e_0 \rangle \\
& \{l_1 : v_1; \dots; l_i : C; \dots; l_n : e_n\} \langle e_0 \rangle = \{l_1 : v_1; \dots; l_i : C \langle e_0 \rangle; \dots; l_n : e_n\} \\
& [v_1; \dots; C; \dots; e_n] \langle e_0 \rangle = [v_1; \dots; C \langle e_0 \rangle; \dots; e_n] \\
& C(e_1, \dots, e_n) \langle e_0 \rangle = C \langle e_0 \rangle (e_1, \dots, e_n) \\
& f(v_1, \dots, C, \dots, e_n) \langle e_0 \rangle = f(v_1, \dots, C \langle e_0 \rangle, \dots, e_n) \\
\\
& (C; i) \langle e_0 \rangle = C \langle e_0 \rangle; i \\
& (\text{IF}(C)\{i_1\}\text{ELSE}\{i_2\}) \langle e_0 \rangle = \text{IF}(C \langle e_0 \rangle)\{i_1\}\text{ELSE}\{i_2\} \\
& (\text{RETURN}(C)) \langle e_0 \rangle = \text{RETURN}(C \langle e_0 \rangle) \\
\\
& C_L \langle e_0 \rangle = C_L \langle e_0 \rangle_L \\
& C \boxplus e \langle e_0 \rangle = C \langle e_0 \rangle \boxplus e \\
& v \boxplus C \langle e_0 \rangle = v \boxplus C \langle e_0 \rangle \\
& \boxplus C \langle e_0 \rangle = \boxplus C \langle e_0 \rangle \\
& C \leftarrow e \langle e_0 \rangle = C \langle e_0 \rangle \leftarrow e \\
& \varphi \leftarrow C \langle e_0 \rangle = \varphi \leftarrow C \langle e_0 \rangle \\
& \{l_1 : v_1; \dots; l_i : C; \dots; l_n : e_n\} \langle e_0 \rangle = \{l_1 : v_1; \dots; l_i : C \langle e_0 \rangle; \dots; l_n : e_n\} \\
& [v_1; \dots; C; \dots; e_n] \langle e_0 \rangle = [v_1; \dots; C \langle e_0 \rangle; \dots; e_n] \\
& C(e_1, \dots, e_n) \langle e_0 \rangle = C \langle e_0 \rangle (e_1, \dots, e_n) \\
& f(v_1, \dots, C, \dots, e_n) \langle e_0 \rangle = f(v_1, \dots, C \langle e_0 \rangle, \dots, e_n) \\
\\
& \bullet \langle l_0 \rangle_L = \bullet \\
& *C_L \langle l_0 \rangle_L = *C_L \langle l_0 \rangle_L \\
& C_L.l_S \langle l_0 \rangle_L = C_L \langle l_0 \rangle_L . l_S \\
& C_L[e] \langle l_0 \rangle_L = C_L \langle l_0 \rangle_L [e] \\
& \varphi[C] \langle e_0 \rangle = \varphi[C \langle e_0 \rangle]
\end{aligned}$$

FIGURE 4.12: Substitution dans les contextes d'évaluation

**FIGURE 4.13:** Évaluation des left-values.

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?

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

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 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.

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

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.

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

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  $\hat{c}$ . Par exemple, au chiffre (symbole) 3, on associe le nombre (entier)  $\hat{3}$ .

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

De même, une fonction n'est pas évaluée plus :

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

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

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

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  $\hat{\boxplus}$  qui agit sur les valeurs. Par exemple, l'opérateur  $\hat{+}$  est l'addition classique entre entiers. Comme précisé dans la section 4.7, la division par zéro via  $/$ ,  $\%$  ou  $/$  provoque l'erreur  $\Omega_{div}$ .

$$\frac{}{\langle \boxplus v, m \rangle \rightarrow \langle \hat{\boxplus} v, m \rangle} \text{ (EXP-UNOP)} \qquad \frac{}{\langle v_1 \boxplus v_2, m \rangle \rightarrow \langle v_1 \hat{\boxplus} v_2, m \rangle} \text{ (EXP-BINOP)}$$

Il est nécessaire de dire un mot sur les opérations  $\hat{+}_p$  et  $\hat{-}_p$  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 :

$$\varphi +_p m = \Omega_{ptr} \text{ si } \nexists (\varphi', n), \varphi = \varphi'[n]$$

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

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

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

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  $\varphi$ . Enfin, un nouvel état mémoire est construit, où la valeur accessible par  $\varphi$  est remplacée par  $v$ . Comme dans le langage C, l'expression d'affectation produit une valeur, qui est celle qui a été affectée.

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

### Expressions composées

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

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

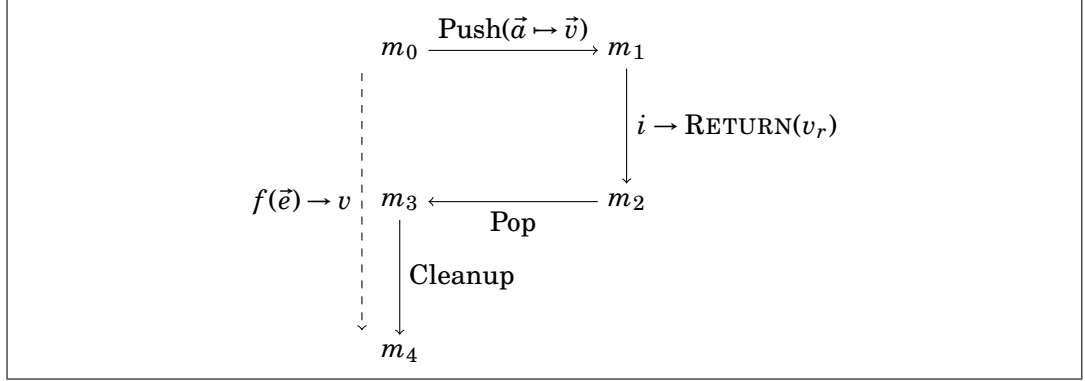
$$\frac{}{\langle [v_1, \dots, v_n], m \rangle \rightarrow \langle [\widehat{v_1}, \dots, \widehat{v_n}], m \rangle} \text{ (EXP-ARRAY)}$$

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  $\text{RETURN}(v)$ . 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 ).

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

Cette évaluation est décrite dans la figure 4.14.



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

## 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 :

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

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 :

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

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.

$$\begin{array}{c} \frac{\langle i, m \rangle \rightarrow \langle \text{PASS}, m' \rangle}{\langle (i; i'), m \rangle \rightarrow \langle i', m' \rangle} \text{ (SEQ)} \qquad \frac{}{\langle (\text{PASS}; i), m \rangle \rightarrow \langle i, m \rangle} \text{ (PASS)} \\[10pt] \frac{}{\langle v, m \rangle \rightarrow \langle \text{PASS}, m \rangle} \text{ (EXP)} \end{array}$$



L'évaluation de  $\text{DECL } x = v \text{ IN}\{i\}$  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  $\text{PASS}$  ou  $\text{RETURN}(v_r)$  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.

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

$$\frac{m' = \text{Extend}(m, x, v) \quad \langle i, m' \rangle \rightarrow \langle \text{RETURN}(v_r), m'' \rangle \quad m''' = \text{Cleanup}(m'' - x)}{\langle \text{DECL } x = v \text{ IN}\{i\}, m \rangle \rightarrow \langle \text{RETURN}(v_r), m''' \rangle} \text{ (DEC-RETURN)}$$

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  $\text{IF-FALSE}$  qui est appliquée et l'instruction revient à évaluer la branche "else". Dans les autres cas, c'est la règle  $\text{IF-TRUE}$  qui s'applique et la branche "then" qui est prise.

$$\frac{}{\langle \text{IF}(0)\{i_t\}\text{ELSE}\{i_f\}, m \rangle \rightarrow \langle i_f, m \rangle} \text{ (IF-FALSE)}$$

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

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

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

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

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 :

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

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.

$$\begin{array}{lcl}
 \text{while}(e) \{ & & \text{if}(e) \{ \\
 \quad i & \cong & \quad i; \\
 \} & & \quad \text{if}(e) \{ \\
 & & \quad i; \\
 & & \quad \text{if}(e) \{ \\
 & & \quad \quad i; \\
 & & \quad \quad \text{if}(e) \{ \\
 & & \quad \quad \quad i; \\
 & & \quad \quad \quad \dots \\
 & & \quad \quad \} \\
 & & \quad \} \\
 & & \} \\
 & & \}
 \end{array}$$

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

$$\begin{array}{lcl}
 \text{while}(e) \{ & & \text{if}(e) \{ \\
 \quad i & \cong & \quad i; \\
 \} & & \quad \text{while}(e) \{ \\
 & & \quad \quad i \\
 & & \quad \} \\
 & & \}
 \end{array}$$

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

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

### 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  $\Omega$ , 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 :

$$\frac{}{\langle \Omega, m \rangle \rightarrow \Omega} \text{ (EXP-ERR)} \qquad \frac{\langle e, m \rangle \rightarrow \Omega}{\langle C[e], m \rangle \rightarrow \Omega} \text{ (EVAL-ERR)}$$

## 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 :

$$m \vdash p \rightarrow m'$$

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

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

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).

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

## 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  $\rightarrow$  :

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

L'exécution d'un programme est alors :

$$([], []) \vdash P \rightarrow^* m$$

## 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

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

$$s_0 \rightarrow \langle \text{IF}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t; \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_0 \rangle \quad (4.1)$$

$$\rightarrow \langle t \leftarrow b; b \leftarrow a \% b; a \leftarrow t; \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_0 \rangle \quad (4.2)$$

$$\rightarrow \langle b \leftarrow a \% b; a \leftarrow t; \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_0 \rangle \quad (4.3)$$

$$\rightarrow \langle a \leftarrow t; \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_0'' \rangle \quad (4.4)$$

$$\rightarrow \langle \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_1 \rangle \quad (4.5)$$

$$\rightarrow \langle \text{IF}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t; \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_1 \rangle \quad (4.6)$$

$$\rightarrow \langle t \leftarrow b; b \leftarrow a \% b; a \leftarrow t; \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_1 \rangle \quad (4.7)$$

$$\rightarrow \langle \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_2 \rangle \quad (4.8)$$

$$\rightarrow \langle \text{IF}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t; \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_2 \rangle \quad (4.9)$$

$$\rightarrow \langle t \leftarrow b; b \leftarrow a \% b; a \leftarrow t; \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_2 \rangle \quad (4.10)$$

$$\rightarrow \langle \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_3 \rangle \quad (4.11)$$

$$\rightarrow \langle \text{IF}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t; \text{WHILE}(b \neq 0)\{t \leftarrow b; b \leftarrow a \% b; a \leftarrow t\}; \text{RETURN}(a)\}, m_3 \rangle \quad (4.12)$$

$$\rightarrow \langle \text{PASS}; \text{RETURN}(a)\}, m_3 \rangle \quad (4.13)$$

$$\rightarrow \langle \text{RETURN}(a)\}, m_3 \rangle \quad (4.14)$$

$$\begin{aligned}
m'_0 &= m_0[t \mapsto 462] = m[a \mapsto 1071][b \mapsto 462][t \mapsto 462] \\
m''_0 &= m'_0[b \mapsto 147] = m[a \mapsto 1071][b \mapsto 147][t \mapsto 462] \\
m_1 &= m''_0[a \mapsto 462] = m[a \mapsto 462][b \mapsto 147][t \mapsto 462] \\
m_2 &= m_1[t \mapsto 147][b \mapsto 21][a \mapsto 147] = m[a \mapsto 147][b \mapsto 21][t \mapsto 147] \\
m_3 &= m_2[t \mapsto 21][b \mapsto 0][a \mapsto 21] = m[a \mapsto 21][b \mapsto 0][t \mapsto 21]
\end{aligned}$$


---

## TODO

- 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  $\rightarrow$  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  $\langle \cdot, \cdot \rangle \rightarrow \langle \cdot, \cdot \rangle$  et quand il faut un  $\langle \cdot, \cdot \rangle \rightarrow^* \langle \cdot, \cdot \rangle$
- "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



## TYPAGE

Dans ce chapitre, nous enrichissons le langage défini dans le chapitre 4 d'un système de types. Celui-ci permet de séparer les programmes bien formés, comme celui de la figure 5.1(a) des programmes mal formés comme celui de la figure 5.1(b).

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

<pre>f () (x=0) {   x = 1   return x }</pre>	<pre>f () (x=0) {   x = 1   return (*x) }</pre>
(a) Programme bien formé	(b) Programme mal formé

**FIGURE 5.1:** Programmes bien et mal formés

<b>Type</b>	$t ::= \text{INT}$	Entier
	$\text{FLOAT}$	Flottant
	$t[]$	Tableau
	$t^*$	Pointeur
	$S$	Structure
	$(t_1, \dots, t_n) \rightarrow t_r$	Fonction
<b>Structure</b>	$S ::= \{l_1 : t_1; \dots; l_n : t_n\}$	Structure simple
<b>Environnement</b>	$\Gamma ::= []$	Environnement vide
<b>de typage</b>	$(a, t) :: \Gamma'$	Extension

FIGURE 5.2: Types et environnements de typage

tout le programme, toutes les instances de cette variable<sup>1</sup> 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  $\text{RETURN}(x)$  permet de conclure que le type de la fonction est  $() \rightarrow \text{INT}$ .

Dans la seconde fonction, au contraire, l'opérateur  $*$  est appliqué à  $x$  (le début de l'analyse est identique et permet de conclure que  $x$  porte des valeurs entières). Or cet opérateur prend un argument d'un type pointeur de la forme  $t^*$  et renvoie alors une valeur de type  $t$ . Ceci est valable pour tout  $t$  ( $\text{INT}$ ,  $\text{FLOAT}$  où même  $t^*$  : le déréférencement d'un pointeur sur pointeur donne un pointeur), mais le type de  $x$ ,  $\text{INT}$ , 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 E.4.

Pour maintenir les contextes de typage, un environnement  $\Gamma$  associe un type à un ensemble de variables.

Plus précisément, un environnement  $\Gamma$  est une liste de couples (variable, type).

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

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

---

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.



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  $\Gamma$ .

$$\Gamma \vdash e : t$$

**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  $\Gamma$  l'instruction  $i$  est bien typée :

$$\Gamma \vdash i$$

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

On note

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

si le typage de la phrase  $p$  transforme l'environnement  $\Gamma$  en  $\Gamma'$ .

On étend cette notation aux suites de phrases :

$$\begin{cases} \Gamma \vdash [] \rightarrow^* \Gamma \\ \Gamma \vdash p :: ps \rightarrow^* \Gamma' \text{ si } \exists \Gamma'', \begin{cases} \Gamma \vdash p \rightarrow \Gamma'' \\ \Gamma'' \vdash ps \rightarrow^* \Gamma' \end{cases} \end{cases}$$

**Typage d'un programme :** Un programme est bien typé si on peut typer sa suite de phrases en partant d'un environnement vide. On note

$$\vdash P \stackrel{\text{def}}{=} \exists \Gamma_f, [] \vdash P \rightarrow^* \Gamma_f$$

## 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  $\Gamma$  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'élément est encore une fois le type de base du type tableau ( $t$  pour  $t[]$ ).

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

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 :

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

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

## Opérateurs

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

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

De même sur FLOAT.

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

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. Les opérateurs  $=$  et  $\neq$  renvoient alors un INT.

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

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

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

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

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

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

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

L'arithmétique de pointeurs préserve le type des pointeurs.

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

### 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.

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

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

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

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

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

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.

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

## 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.

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

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.

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

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

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

## 5.5 Fonctions

Le typage des fonctions fait intervenir une variable virtuelle  $\underline{R}$ . Cela revient à typer l'instruction  $\text{RETURN}(e)$  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  $\Gamma'$  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  $\Gamma'$ , alors la fonction est typable en  $(t_1, \dots, t_n) \rightarrow t$  sous  $\Gamma$ .

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

Cette règle utilise les notations suivantes :

$$\begin{aligned} \vec{a} &\stackrel{\text{def}}{=} (a_1, \dots, a_n) \text{ où } n = |\alpha| \\ \vec{a} : \vec{t} &\stackrel{\text{def}}{=} a_1 : t_1, \dots, a_n : t_n \text{ où } n = |\alpha| \end{aligned}$$

## 5.6 Phrases

L'évaluation 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.

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

La déclaration d'une variable globale commence de la même manière, mais on enrichit l'environnement de cette nouvelle valeur.

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

## 5.7 Sûreté du typage

Un programme bien typé possède des propriétés de sûreté.

**Progrès :** l'évaluation d'un terme bien typé ne reste pas bloquée ; il y a toujours une règle qui s'applique.

**Préservation :** l'évaluation d'un terme bien typé produit un terme bien typé.

Puisqu'il s'agit de propriétés reliant la syntaxe à la sémantique, deux types d'environnement sont utilisés en même temps. D'une part, les environnements de typage  $\Gamma$  pour la syntaxe, et d'autre part les environnements mémoire  $m$  pour la sémantique. Il est nécessaire de définir une relation de compatibilité entre ces deux mondes, pour exprimer par exemple qu'un état mémoire contient dans la variable  $x$  un entier.

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  $\Gamma \vdash \Box e : t$ , alors on est dans un des cas suivants :
    - $\Box \in \{+, -, \sim, !\}$ ,  $t = \text{INT}$ ,  $\Gamma \vdash e : \text{INT}$
    - $\Box \in \{+., -.\}$ ,  $t = \text{FLOAT}$ ,  $\Gamma \vdash e : \text{FLOAT}$
  - si  $\Gamma \vdash e_1 \Box e_2 : t$ , un des cas suivants se présente :
    - $\Box \in \{+, -, \times, /, \&, |, ^, \&\&, ||, \ll, \gg, \leq, \geq, <, >\}$ ,  $\Gamma \vdash e_1 : \text{INT}$ ,  $\Gamma \vdash e_2 : \text{INT}$ ,  $t = \text{INT}$
    - $\Box \in \{+., -., \times., /., \leq., \geq., <., >.\}$ ,  $\Gamma \vdash e_1 : \text{FLOAT}$ ,  $\Gamma \vdash e_2 : \text{FLOAT}$ ,  $t = \text{FLOAT}$
    - $\Box \in \{=, \neq\}$ ,  $\Gamma \vdash e_1 : t'$ ,  $\Gamma \vdash e_2 : t'$ ,  $\text{EQ}(t')$ ,  $t = \text{INT}$
    - $\Box \in \{\leq, \geq, <, >\}$ ,  $t = \text{INT}$ ,  $\Gamma \vdash e_1 : t'$ ,  $\Gamma \vdash e_2 : t'$ ,  $t' \in \{\text{INT}, \text{FLOAT}\}$
    - $\Box \in \{+_p, -_p\}$ ,  $\exists t', t = t' *$ ,  $\Gamma \vdash e_1 : t' *$ ,  $\Gamma \vdash e_2 : \text{INT}$

- *Appel de fonction* : si  $\Gamma \vdash e(e_1, \dots, e_n) : t$ , il existe  $(t_1, \dots, t_n)$  tels que

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

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

*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  $x : t \in \Gamma$ .
  - $\Gamma \vdash * \varphi : t$   
De même, seule la règle LV-DEREF convient. On en conclut que  $\Gamma \vdash \varphi : t*$ .
  - $\Gamma \vdash \varphi[] : t$   
Idem avec LV-INDEX.
  - $\Gamma \vdash \varphi.l : t$   
 $\Gamma \vdash \varphi : \{l : t; \dots\}$
- *Appel de fonction* : pour en arriver à  $\Gamma \vdash e(e_1, \dots, e_n) : t$ , seule la règle CALL s'applique, ce qui permet de conclure.
- *Fonction* : la seule règle possible pour conclure une dérivation de

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

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.*

- si  $\Gamma \vdash v : \text{INT}$ ,  $v = n$ .
- si  $\Gamma \vdash v : \text{FLOAT}$ ,  $v = d$ .



- si  $\Gamma \vdash v : \text{UNIT}$ ,  $v = ()$ .
- si  $\Gamma \vdash v : t*$ ,  $v = \varphi$  ou  $v = \text{NULL}$ .
- si  $\Gamma \vdash v : (t_1, \dots, t_n) \rightarrow t$ ,  $v = f$ .
- si  $\Gamma \vdash v : t*$ ,  $v = \varphi$ .
- si  $\Gamma \vdash v : t[]$ ,  $v = [v_1; \dots; v_n]$ .
- si  $\Gamma \vdash v : (t_1, \dots, t_n) \rightarrow t$ ,  $v = \text{fun}(a_1, \dots, a_n)\{i\}$ .

Ces lemmes étant établis, on énonce maintenant le théorème de progrès accompagné d'une notion de compatibilité entre environnements de typage  $\Gamma$  et états mémoire  $m$ .

**Définition 5.1** (Compatibilité mémoire). *Soient  $\Gamma$  un environnement de typage et  $m = x_1 \mapsto v_1, \dots, x_n \mapsto v_n$  un état mémoire. On dit que  $m$  est compatible avec  $\Gamma$  si*

$$\exists(t_1, \dots, t_n), \forall i \in [1; n], \begin{cases} \Gamma \vdash x_i : t_i \\ \Gamma \vdash v_i : t_i \end{cases}$$

On note alors  $\Gamma \vdash_{\text{mem}} m$ .

**Définition 5.2** (Typage de la mémoire).

$$\Gamma \models m \stackrel{\text{def}}{=} \forall x \in \text{Visible}(m). \Gamma, m \models x$$

$$\Gamma, m \models x \stackrel{\text{def}}{=} \exists t. \Gamma, m \models x : t$$

La définition de  $\Gamma, m \models x : t$  dépend de  $v = m[x]_{\Phi}$ . On pose  $\Gamma, m \models x : t \stackrel{\text{def}}{=} P(v)$  avec :

- $P(c) \stackrel{\text{def}}{=} \Gamma \vdash c : t$
- $P(f) \stackrel{\text{def}}{=} \Gamma \vdash f : t$
- $P(\varphi) \stackrel{\text{def}}{=} \exists t'. t = t' * \wedge \Gamma, m \models y : t'$  où  $y = m[\varphi]_{\Phi}$
- $P([v_1; \dots; v_n]) \stackrel{\text{def}}{=} \exists t'. t = t'[] \wedge \forall i \in [1; n]. \Gamma \vdash v_i : t'$
- $P(\{l_1 : v_1; \dots; l_n : v_n\}) \stackrel{\text{def}}{=} \exists(t_1, \dots, t_n). t = \{l_1 : t_1; \dots; l_n : t_n\} \wedge \forall i \in [1; n]. \Gamma \vdash v_i : t_i$

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

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

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  $\langle lv, m \rangle \rightarrow^* \langle v, m' \rangle$ ,  $v$  est de la forme  $\varphi$  (référence mémoire) ou  $\Omega$  (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  $\sigma$  de  $[1; n]$ , on note  $\sigma(x_1 : t_1, \dots, x_n : t_n) = x_{\sigma(1)} : t_{\sigma(1)}, \dots, x_{\sigma(n)} : t_{\sigma(n)}$ .*

*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.

## TODO

- 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  $\Gamma$  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 ?
- théorème 5.2 rappeler où a été défini l'évaluation d'une expression et dans quel cas elle produit des valeurs



## QUALIFICATEURS DE TYPE

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 système 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 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  $\text{TAINTED}(\varphi)$  signifiant que l'objet pointé se situe en espace utilisateur

<b>Expressions</b>	$e ::= \dots$	
	$\text{TAINT}(e)$	Expression souillée
	$lv \Leftarrow_U e$	Lecture depuis l'espace utilisateur
	$e \Rightarrow_U e$	Écriture dans l'espace utilisateur
<b>Contextes</b>	$C ::= \dots$	
	$\text{TAINT}(C)$	
<b>Chemins</b>	$\varphi ::= \dots$	
	$\text{TAINTED}(\varphi)$	Valeur souillée
<b>Erreurs</b>	$\Omega ::= \dots$	
	$\Omega_{\text{taint}}$	Erreur de souillure

FIGURE 6.1: Ajouts liés aux pointeurs utilisateurs

- enfin, on définit une nouvelle erreur  $\Omega_{\text{taint}}$  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  $\text{TAINT}(\cdot)$  dans une instruction consiste à ajouter un  $\text{TAINTED}(\cdot)$  dans la valeur construite. Ceci ne peut être fait que dans le cas où la valeur est un chemin  $\varphi$ , c'est à dire que la construction  $\text{TAINT}(\cdot)$  ne peut se faire que sur une expression de type pointeur.

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

D'autre part, une règle accède à la mémoire :  $\text{EXP-LV}$  ; pour rappel :

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

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

On rajoute donc le cas :

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

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  $\Omega_{div}$ ,  $\Omega_{array}$  ou  $\Omega_{ptr}$ , 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  $\Gamma = x : \text{INT}$  :

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

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.

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

## 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

<b>Qualificateurs</b>	$q ::= \text{KERNEL}$	Donnée noyau (sûre)
	$\text{USER}$	Donnée utilisateur (non sûre)
<b>Types</b>	$t ::= \dots$	
	$t^*$	Pointeur
	$tq^*$	Pointeur qualifié

FIGURE 6.2: Changements liés aux qualificateurs de types

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 2.4). 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 :

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

L'opérateur **TAINT**( $\cdot$ ) transforme un pointeur selon la règle de souillure suivante :

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

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

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



### 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  $\Gamma \vdash_q v : t$ .*

---

## 6.4 Analyse de terminaison des chaînes C

Dans ce chapitre, nous présentons une autre extension au système de types du chapitre 5, similaire à celle de la section précédente. Il s'agit cette fois-ci de détecter les pointeurs sur caractères (`char *`) qui sont terminés par un caractère NUL et donc une chaîne C correcte. La bibliothèque C propose quantité de fonctions manipulant ces chaînes et appeler une fonction comme `strcpy` sur un pointeur quelconque est un problème de sécurité que nous cherchons à détecter.

### 6.4.1 But

Le langage C ne fournit pas directement de type "chaîne de caractère". C'est au programmeur de les gérer via des pointeurs sur caractère (`char *`).

En théorie le programmeur est libre de choisir une représentation : des chaînes préfixées par la longueur, une structure contenant la taille et un pointeur vers les données, ou encore une chaîne avec un terminateur comme 0.

Néanmoins c'est ce dernier style qui est le plus idiomatique : par exemple, les littéraux de chaîne ("comme ceci") ajoutent un octet nul à la fin. De plus, le standard décrit dans la bibliothèque d'exécution de nombreuses fonctions destinées à les manipuler — c'est le fichier `<string.h>` ([ISO99] section 7.21).

Ainsi la fonction `strcpy` a pour prototype :

```
char *strcpy(char *dest, const char *src);
```

Elle réalise la copie de la chaîne pointée par `src` à l'endroit pointé par `dest`. Pour détecter la fin de la chaîne, cette fonction parcourt la mémoire jusqu'à trouver un caractère nul. Une implémentation naïve pourrait être :

```
char *strcpy(char *dest, const char *src)
{
    int i;
    for(i=0;src[i]!=0;i++) {
        dest[i] = src[i];
    }
    return dest;
}
```

La copie n'est arrêtée que lorsqu'un 0 est lu. Autrement dit, si quelqu'un contrôle la valeur pointée par `src`, il pourra écraser autant de données qu'il le désire. On est dans le cas d'école du débordement de tampon sur la pile tel que décrit dans [One96]. Considérons la fonction suivante :

```
void f(char *src)
{
    char buf[100];
    strcpy(buf, src);
}
```

Si le pointeur `src` pointe sur une chaîne de longueur supérieure à 100 (ou une zone mémoire qui n'est pas une chaîne et ne contient pas de 0), les valeurs placées sur la pile juste avant `buf` (à une adresse supérieure) seront écrasées. Avec les conventions d'appel habituelles, il s'agit de l'adresse de retour de la fonction. Un attaquant pourra donc détourner le flot d'exécution du programme.

Pour éviter ces cas de fonctions vulnérables, on peut introduire une distinction entre les pointeurs `char *` classiques (représentant l'adresse d'un caractère par exemple) et les pointeurs sur une chaîne terminée par un caractère nul.

Dans certaines bases de code (la plus célèbre étant celle de Microsoft), une convention syntaxique est utilisée : les pointeurs vers des chaînes terminées par 0 ont un nom qui commence par `sz`, comme `"szTitle"`. C'est pourquoi nous appellerons ce qualificateur de type `sz`.

### 6.4.2 Approche

Cette propriété est un peu différente de la séparation entre espace utilisateur et espace noyau modélisée précédemment : autant un pointeur reste contrôlé par l'utilisateur (ou sûr) toute sa vie, autant le fait d'être terminé par un octet nul dépend de l'ensemble de l'état mémoire. Il y a deux problèmes principaux à considérer.

D'une part, l'*aliasing* rend l'analyse difficile : si `p` et `q` pointent tous les deux vers une même zone mémoire, le fait de modifier l'un peut modifier l'autre. D'autre part, ce n'est pas parce qu'une fonction maintient l'invariant de terminaison, qu'elle le maintient à chaque instruction.

On peut résoudre en partie le problème d'*aliasing* en étant très conservateur, c'est à dire en sous-approximant l'ensemble des chaînes du programme (on traitera une chaîne légitime comme une chaîne non terminée, interdisant par excès de zèle les fonctions comme `strcpy`).

Le second problème est plus délicat puisqu'il casse l'hypothèse habituelle que chaque variable conserve le même type au long de sa vie. Plusieurs techniques sont possibles pour contourner ce problème : la première est d'être encore une fois conservateur et d'interdire ces constructions (on ne pourrait alors analyser que les programmes ne manipulant les chaînes qu'à travers les fonctions de la bibliothèque standard). Une autre est d'insérer des annotations permettant de s'affranchir localement du système de types. Enfin, il est possible d'utiliser un système de types où les va-

riables ont en plus d'un type, un automate d'états possible dépendant de la position dans le programme : c'est le concept de *typestates*[SY86].

### 6.4.3 Annotation de `string.h`

Une première étape est d'annoter l'ensemble des fonctions manipulant les chaînes de caractères.

#### Fonctions de copie

##### **memcpy**

```
void *memcpy(void *dest, const void *src, size_t n);
```

##### **memmove**

```
void *memmove(void *dest, const void *src, size_t n);
```

##### **strcpy**

```
char *strcpy(char *dest, const char *src);
```

##### **strncpy**

```
char *strncpy(char *dest, const char *src, size_t n);
```

#### Fonctions de concaténation

##### **strcat**

```
char *strcat(char *dest, const char *src);
```

##### **strncat**

```
char *strncat(char *dest, const char *src, size_t n);
```

#### Fonctions de comparaison

##### **memcmp**

```
int memcmp(const void *s1, const void *s2, size_t n);
```

##### **strcmp**

```
int strcmp(const char *s1, const char *s2);
```

**strncmp**

```
int strncmp(const char *s1, const char *s2, size_t n);
```

**strcoll**

```
int strcoll(const char *s1, const char *s2);
```

**strxfrm**

```
size_t strxfrm(char *dest, const char *src, size_t n);
```

**Fonctions de recherche****memchr**

```
void *memchr(const void *s, int c, size_t n);
```

**strchr**

```
char *strchr(const char *s, int c);
```

**strcspn**

```
size_t strcspn(const char *s, const char *reject);
```

**strpbrk**

```
char *strpbrk(const char *s, const char *accept);
```

**strrchr**

```
char *strrchr(const char *s, int c);
```

**strspn**

```
size_t strspn(const char *s, const char *accept);
```

**strstr**

```
char *strstr(const char *haystack, const char *needle);
```

**strtok**

```
char *strtok(char *str, const char *delim);
```

**Fonctions diverses****memset**

```
void *memset(void *s, int c, size_t n);
```

**strerror**

```
char *strerror(int errnum);
```

**strlen**

```
size_t strlen(const char *s);
```

**6.4.4 Typage des primitives****6.4.5 Extensions au système de types****6.4.6 Résultats****TODO**

- 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

## **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 précédentes 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.



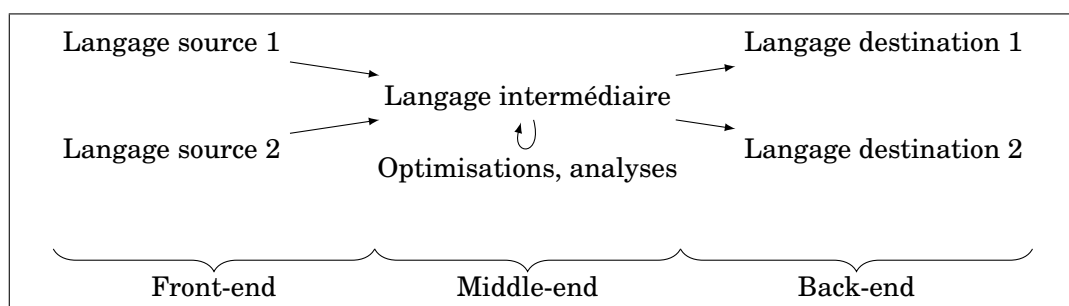
## IMPLANTATION

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 Langages intermédiaires

Le langage C [KR88, ISO99] a été conçu pour être une sorte d’assembleur portable, permettant décrire du code indépendamment de l’architecture sur laquelle il sera compilé. Historiquement, c’est il a permis de créer Unix, et ainsi de nombreux logiciels bas niveau sont écrits en C. En particulier, il existe des compilateurs de C vers les différents langages machine pour à peu près toutes les architectures.

Lors de l’écriture d’un compilateur, on a besoin d’un langage intermédiaire qui



**FIGURE 7.1:** Décomposition d’un compilateur : front-ends, middle-end, back-ends

fasse l'intermédiaire entre *front-end* et *back-end* (figure 7.1). Depuis ce langage on doit pouvoir exprimer des transformations intermédiaires sur cette représentation (analyses sémantiques, optimisations, etc), mais aussi compiler ce langage vers un langage machine.

L'idée de prendre C comme langage intermédiaire est très séduisante, mais malheureusement sa sémantique est trop complexe et trop peu spécifiée. Il est donc judicieux d'utiliser un langage plus simple à cet effet. Dans de nombreux projets, des sous-ensembles de C ont été définis pour aller dans ce sens.

## Langages

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 une passe de compilation 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** [Cam1] 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] [Cam7], 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.

**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 7.2.

**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].

## 7.2 Newspeak

(wip)

## 7.3 Chaîne de compilation

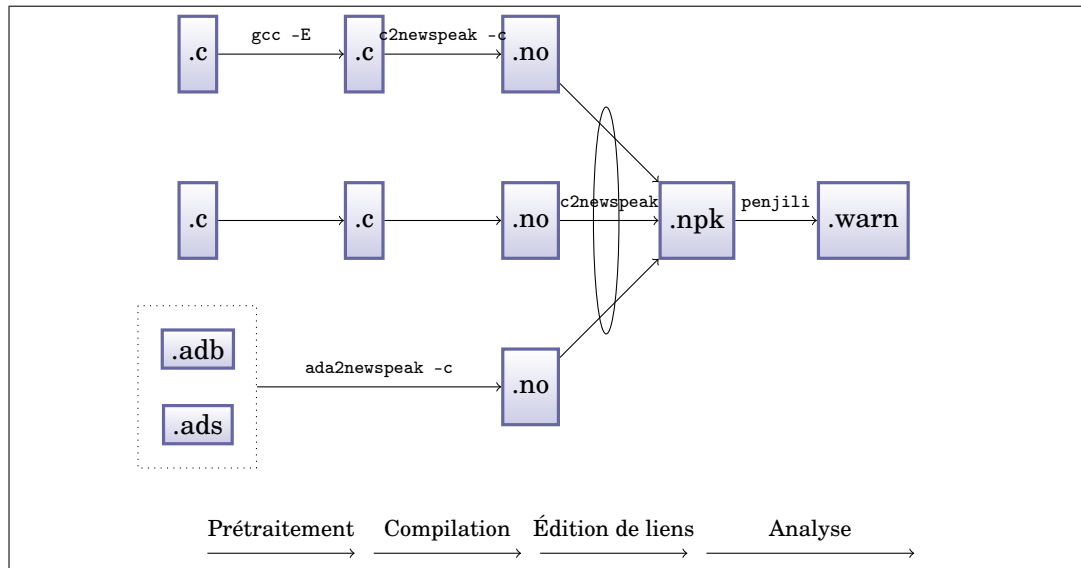
La compilation vers C est faite en trois étapes (figure 7.2) : prétraitement du code source, compilation de C prétraité vers NEWSPEAK, puis compilation de NEWSPEAK vers ce langage.

### 7.3.1 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.

### 7.3.2 Compilation (levée des ambiguïtés)

Cette passe est réalisée par l'utilitaire C2NEWSPEAK. L'essentiel de la compilation consiste à mettre à plat les définition de types, et à simplifier le flot de contrôle. C en effet propose de nombreuses constructions ambiguës ou redondantes.

**FIGURE 7.2:** Compilation depuis Newspeak

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`, 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]. En ce qui concerne l'élimination des sauts vers l'arrière, on peut se référer à [EH94].

### 7.3.3 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)

### 7.3.4 Implantation de l'algorithme de typage

Commençons par étudier le cas du lambda-calcul simplement typé (figure 7.3). Prenons l'exemple de la fonction suivante<sup>1</sup> :

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

On voit que 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`.

Mais comment faire pour implanter cette analyse ? En fait le système de types de la figure 7.3 a une propriété particulièrement intéressante : chaque forme syntaxique (variable, abstraction, etc) est en conclusion exactement d'une règle de typage. Cela permet de toujours savoir quelle règle il faut appliquer.

Partant du terme de conclusion ( $f$ ), on peut donc en déduire un squelette d'arbre d'inférence (figure 7.4)<sup>2</sup>

Une fois à cette étape, on peut donner un nom à chaque type inconnu :  $\tau_1, \tau_2, \dots$ . L'utilisation qui en est faite permet de générer un ensemble de contraintes d'unification. Par exemple, pour chaque application de la règle (APP) :

$$\frac{\Gamma \vdash \dots : \tau_3 \quad \Gamma \vdash \dots : \tau_1}{\Gamma \vdash \dots : \tau_2} \text{ (APP)}$$

on doit déduire que  $\tau_3 = \tau_1 \rightarrow \tau_2$ .

- 
1. On suppose que `plus` est une fonction de l'environnement global qui a pour type `INT → INT → INT`.
  2. Par souci de clarté, les prémisses des applications de (VAR) ne sont pas notées.

<b>Termes</b>	$t ::= x$	Variable
	$\lambda x. t$	Abstraction
	$t$	Application
	$n$	Entier
	$f$	Flottant
	$(t, t)$	Couple
	$\text{fst } t$	Projection gauche
	$\text{snd } t$	Projection droite
<b>Types</b>	$\tau ::= \text{INT}$	Entier
	$\text{FLOAT}$	Flottant
	$\tau \rightarrow \tau$	Fonction
	$\tau \times \tau$	Produit
<b>Contextes</b>	$\Gamma ::= \varepsilon$	Contexte vide
	$\Gamma, x : \tau$	Extension
$\frac{}{\Gamma \vdash n : \text{INT}} (\text{INT}) \quad \frac{}{\Gamma \vdash f : \text{FLOAT}} (\text{FLOAT}) \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} (\text{VAR})$		
$\frac{\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash x : \tau_1}{\Gamma \vdash fx : \tau_2} (\text{APP}) \quad \frac{\Gamma, x : \tau_1 \vdash y : \tau_2}{\Gamma \vdash \lambda x. y : \tau_1 \rightarrow \tau_2} (\text{ABS})$		
$\frac{\Gamma \vdash x : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } x : \tau_1} (\text{PROJ-G}) \quad \frac{\Gamma \vdash x : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } x : \tau_2} (\text{PROJ-D}) \quad \frac{\Gamma \vdash x : \tau_1 \quad \Gamma \vdash y : \tau_2}{\Gamma \vdash (x, y) : \tau_1 \times \tau_2} (\text{TUP})$		

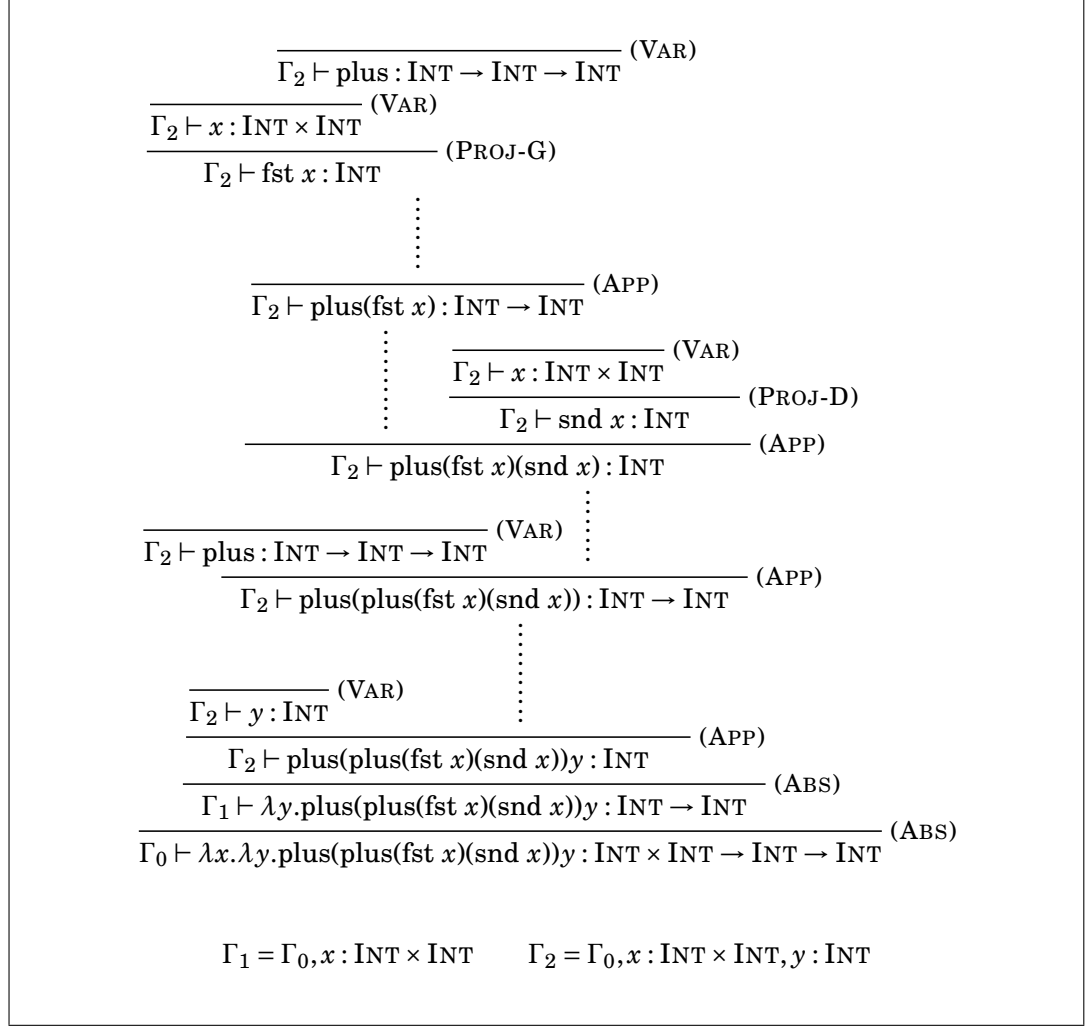
**FIGURE 7.3:** Lambda calcul simplement typé avec entiers, flottants et couples



$$\begin{array}{c}
\frac{}{\Gamma_2 \vdash x} \text{(VAR)} \\
\hline
\Gamma_2 \vdash \text{fst } x \quad \text{(PROJ-G)} \\
\vdots \\
\frac{}{\Gamma_2 \vdash \text{plus}} \text{(VAR)} \\
\hline
\Gamma_2 \vdash \text{plus}(\text{fst } x) \quad \text{(APP)} \\
\vdots \\
\frac{}{\Gamma_2 \vdash x} \text{(VAR)} \\
\hline
\Gamma_2 \vdash \text{snd } x \quad \text{(PROJ-D)} \\
\hline
\Gamma_2 \vdash \text{plus}(\text{fst } x)(\text{snd } x) \quad \text{(APP)} \\
\vdots \\
\frac{}{\Gamma_2 \vdash \text{plus}} \text{(VAR)} \\
\hline
\Gamma_2 \vdash \text{plus}(\text{plus}(\text{fst } x)(\text{snd } x)) \quad \text{(APP)} \\
\vdots \\
\frac{}{\Gamma_2 \vdash y} \text{(VAR)} \\
\hline
\Gamma_2 \vdash \text{plus}(\text{plus}(\text{fst } x)(\text{snd } x))y \quad \text{(APP)} \\
\hline
\Gamma_1 \vdash \lambda y. \text{plus}(\text{plus}(\text{fst } x)(\text{snd } x))y \quad \text{(ABS)} \\
\hline
\Gamma_0 \vdash \lambda x. \lambda y. \text{plus}(\text{plus}(\text{fst } x)(\text{snd } x))y \quad \text{(ABS)}
\end{array}$$

$\Gamma_1 = \Gamma_0, x \quad \Gamma_2 = \Gamma_0, x, y$

**FIGURE 7.4:** Arbre d'inférence : règles à utiliser

**FIGURE 7.5:** Arbre d'inférence complet

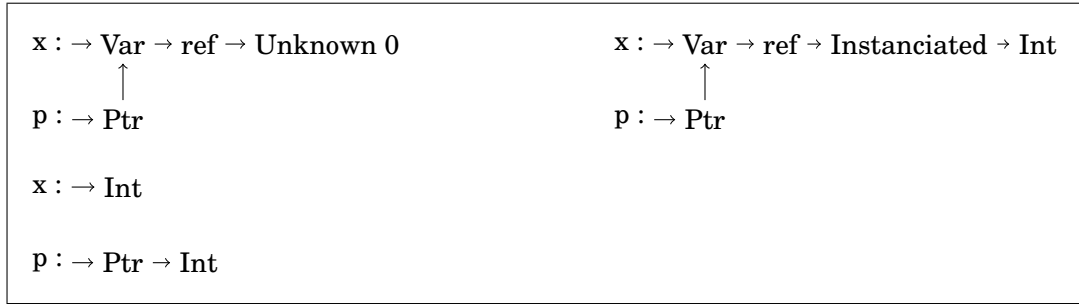
Ce signe = est à prendre comme une contrainte d'égalité : partant d'un ensemble de contraintes de la forme "type avec inconnue = type avec inconnue", on veut obtenir une substitution "inconnue  $\rightarrow$  type concret".

Pour résoudre ces contraintes, on commence par les simplifier : si  $\tau_a \rightarrow \tau_b = \tau_c \rightarrow \tau_d$ , alors  $\tau_a = \tau_c$  et  $\tau_b = \tau_d$ . De même si  $\tau_a \times \tau_b = \tau_c \times \tau_d$ . Au contraire, si  $\tau_a \rightarrow \tau_b = \tau_c \times \tau_d$ , il est impossible d'unifier les types et il faut abandonner l'inférence de types. D'autres cas sont impossibles, par exemple  $\text{INT} = \tau_1 \rightarrow \tau_2$  ou  $\text{INT} = \text{FLOAT}$ .

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

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

Une fois toutes les substitutions effectuées, on obtient un arbre de typage correct (figure 7.5, donc un programme totalement inféré).

**FIGURE 7.6:** Unification par partage

```

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

```

**FIGURE 7.7:** Compilation d'un programme C - avant

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

L'implémentation de cet algorithme utilise le partage et les références (figure 7.6).

D'abord 7.6(a), ensuite 7.6(b), et enfin 7.6(c).

Prenons l'exemple de la figure 7.7 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 première 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  $\text{Ptr}(t_1)$ . On applique le même raisonnement à la seconde affectation : à gauche, le type est  $t_1$  et à droite  $\text{Int}$ . On en déduit que le type de  $x$  est  $\text{Int}$  et celui de  $p$  est  $\text{Ptr}(\text{Int})$ .

```

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.

```

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

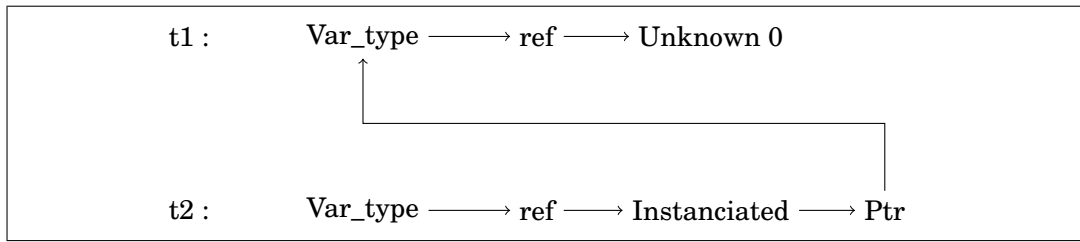
```

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

```

**FIGURE 7.8:** Unification : partage

```

(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;
  
```

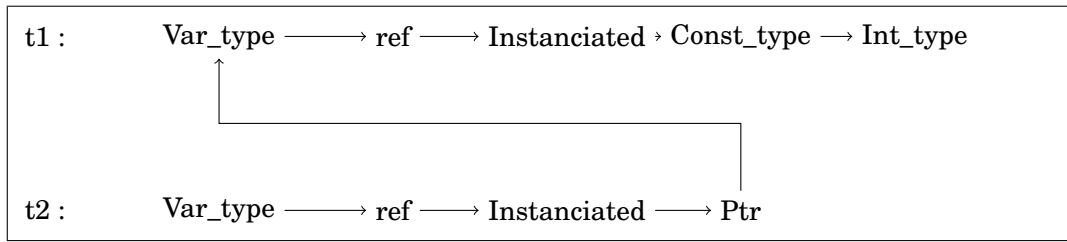
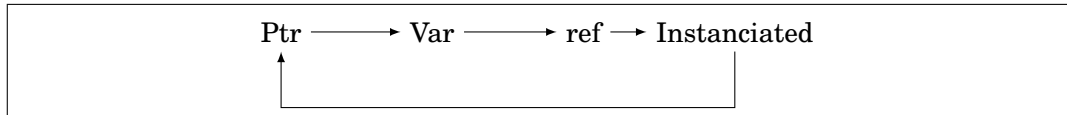
Ici il se passe plusieurs choses intéressantes. D'une part nous faisons appel à une fonction externe `typeof` qui retourne le type d'une expression sous un environnement (dans une implantation complète il s'agirait d'un appel récursif). Dans ce cas, `typeof lhs1 env` est identique à `Env.get lhs1 env` 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 7.8).

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 rhs2 env` à `Const_type Int_type`. Et dans ce cas, l'unification doit se faire entre `t1` et `Const_type Int_type` : cela mute la référence derrière `t1` (figure 7.9).

**FIGURE 7.9:** Unification par mutation de références**FIGURE 7.10:** Cycle dans le graphe de types

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`.

### 7.3.5 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 (Instanciaded t))` en `t`.

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

- si les deux types sont inconnus (de la forme `Var (ref (Instanciaded t))`), on fait pointer l'une des deux références vers le premier type. Notons que cela crée un type de la forme `Var (ref (Instanciaded (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 7.10). 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 7.11:** Compilation d'un programme C - après

Le programme C (figure 7.7) est compilé ainsi en Tyspeak (figure 7.11).



## ÉTUDE DE CAS : UN PILOTE DE CARTE GRAPHIQUE

### 8.1 Description du problème

(wip)

Un système d'exploitation moderne comme GNU/Linux est séparé en deux niveaux de privilèges : le noyau, qui gère directement le matériel, et les applications de l'utilisateur, qui communiquent avec le noyau par l'interface restreinte des *appels système*.

Pour assurer l'isolation, ces deux parties n'ont pas accès aux mêmes zones mémoire (cf. figure 2.5).

Si le code utilisateur tente d'accéder à la mémoire du noyau, une erreur sera déclenchée. En revanche, si cette écriture est faite au sein de l'implantation d'un appel système, il n'y aura pas d'erreur puisque le noyau a accès à toute la mémoire : l'isolation aura donc été brisée.

Pour celui qui implante un appel système, il faut donc empêcher qu'un pointeur passé en paramètre référence le noyau. Autrement dit, il est indispensable de vérifier dynamiquement que la zone dans laquelle pointe le paramètre est accessible par l'appelant[Har88].

Si au contraire un tel pointeur est déréférencé sans vérification (avec \* ou une fonction comme memcpy), le code s'exécutera correctement mais en rendant le système vulnérable, comme le montre la figure 8.1.

Pour éviter cela, le noyau fournit un ensemble de fonctions qui permettent de vérifier dynamiquement la valeur d'un pointeur avant de le déréférencer. Par exemple, dans la figure précédente, la ligne 8 aurait dû être remplacée par :

```
copy_from_user(&value, value_ptr, sizeof(value));
```

cf annexe A
-------------

**FIGURE 8.1:** Bug freedesktop.org #29340. 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.

L'analyse présentée ici permet de vérifier automatiquement et statiquement que les pointeurs qui proviennent de l'espace utilisateur ne sont déréférencés qu'à travers une de ces fonctions sûres.

## 8.2 Principes de l'analyse

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 8.1 : 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`).

### 8.3 Implantation

Une implantation est en cours. Le code source est d'abord prétraité par `gcc -E` puis converti en Newspeak [HL08], un langage destiné à l'analyse statique. Ce traducteur peut prendre en entrée tout le langage C, y compris de nombreuses extensions GNU utilisées dans le noyau. En particulier, l'exemple de la figure 8.1 peut être analysé.

À partir de cette représentation du programme et d'un ensemble d'annotations globales, on propage les types dans les sous-expressions jusqu'aux feuilles.

Si aucune contradiction n'est trouvée, c'est que le code respecte la propriété d'isolation. Sinon, cela peut signifier que le code n'est pas correct, ou bien que le système de types n'est pas assez expressif pour le code en question.

Le prototype, disponible sur [♣<sup>8</sup>], fera l'objet d'une démonstration.

### 8.4 Conclusion

Nous avons montré que le problème de la manipulation de pointeurs non sûrs peut être traité avec une technique de typage. Elle est proche des analyses menées dans CQual [FFA99] ou Sparse [♣<sup>5</sup>].

Plusieurs limitations sont inhérentes à cette approche : notamment, la présence d'unions ou de *casts* entre entiers et pointeurs fait échouer l'analyse.

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.



# CHAPITRE 9

## CONCLUSION

### 9.1 Limitations

### 9.2 Perspectives





# **Annexes**





## CODE DU MODULE NOYAU

```
/* 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(c);
            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);
```

```
}
```



## **RÈGLES D'ÉVALUATION**

$\frac{\langle e, m \rangle \rightarrow \langle e', m' \rangle}{\langle C \langle e \rangle, m \rangle \rightarrow \langle C \langle e' \rangle, m' \rangle} \text{ (CTX)}$	
$\frac{\langle lv, m \rangle \rightarrow \langle lv', m' \rangle}{\langle C_L \langle lv \rangle_L, m \rangle \rightarrow \langle C_L \langle lv' \rangle_L, m' \rangle} \text{ (CTX-LV)}$	
$\frac{\langle i, m \rangle \rightarrow \langle i', m' \rangle}{\langle C_I \langle i \rangle_I, m \rangle \rightarrow \langle C_I \langle i' \rangle_I, m' \rangle} \text{ (CTX-INSTR)}$	
<b>Contextes</b>	$C ::= C_L$ $  C \boxplus e$ $  v \boxplus C$ $  \boxplus C$ $  C \leftarrow e$ $  \varphi \leftarrow C$ $  \{l_1 : v_1; \dots; l_i : C; \dots; l_n : e_n\}$ $  [v_1; \dots; C; \dots; e_n]$ $  C(e_1, \dots, e_n)$ $  f(v_1, \dots, C, \dots, e_n)$
<b>Contextes (left-values)</b>	$C_L ::= \bullet$ $  * C_L$ $  C_L.l_S$ $  C_L[e]$ $  \varphi[C]$
<b>Contextes (instructions)</b>	$C_I ::= C_I; i$ $  \text{IF}(C)\{i_1\}\text{ELSE}\{i_2\}$ $  \text{RETURN}(C)$ $  \text{DECL } x = C \text{ IN}\{i\}$ $  C$

FIGURE B.1: Règles d'évaluation - contextes



$$\frac{}{\langle \Omega, m \rangle \rightarrow \Omega} \text{ (EXP-ERR)} \qquad \frac{\langle e, m \rangle \rightarrow \Omega}{\langle C[e], m \rangle \rightarrow \Omega} \text{ (EVAL-ERR)}$$

**FIGURE B.2:** Règles d'évaluation - erreurs

$$\begin{array}{ll} \frac{a = \text{Lookup}(x, m)}{\langle x, m \rangle \rightarrow \langle a, m \rangle} \text{ (PHI-VAR)} & \frac{}{\langle * \varphi, m \rangle \rightarrow \langle \widehat{* \varphi}, m \rangle} \text{ (PHI-DEREF)} \\ \frac{}{\langle lv.l_S, m \rangle \rightarrow \langle lv.\widehat{l}, m \rangle} \text{ (PHI-STRUCT)} & \frac{}{\langle \varphi[n], m \rangle \rightarrow \langle \widehat{\varphi[n]}, m \rangle} \text{ (PHI-ARRAY)} \\ \frac{}{\langle c, m \rangle \rightarrow \langle \widehat{c}, m \rangle} \text{ (EXP-CST)} & \frac{}{\langle f, m \rangle \rightarrow \langle \widehat{f}, m \rangle} \text{ (EXP-FUN)} \\ & \frac{}{\langle \varphi, m \rangle \rightarrow \langle m[\varphi]_\Phi, m \rangle} \text{ (EXP-LV)} \end{array}$$

**FIGURE B.3:** Règles d'évaluation - constantes et left-values

$$\begin{array}{ll} \frac{}{\langle \boxminus v, m \rangle \rightarrow \langle \widehat{\boxminus v}, m \rangle} \text{ (EXP-UNOP)} & \frac{}{\langle v_1 \boxplus v_2, m \rangle \rightarrow \langle v_1 \widehat{\boxplus} v_2, m \rangle} \text{ (EXP-BINOP)} \\ \frac{}{\langle \& \varphi, m \rangle \rightarrow \langle \widehat{\& \varphi}, m \rangle} \text{ (EXP-ADDR OF)} & \frac{}{\langle \varphi \leftarrow v, m \rangle \rightarrow \langle v, m[\varphi \leftarrow v]_\Phi \rangle} \text{ (EXP-SET)} \end{array}$$

**FIGURE B.4:** Règles d'évaluation - opérations

$$\begin{array}{l} \frac{}{\langle \{l_1 : v_1; \dots; l_n : v_n\}, m \rangle \rightarrow \langle \widehat{\{l_1 : v_1; \dots; l_n : v_n\}}, m \rangle} \text{ (EXP-STRUCT)} \\ \frac{}{\langle [v_1, \dots, v_n], m \rangle \rightarrow \langle \widehat{[v_1, \dots, v_n]}, m \rangle} \text{ (EXP-ARRAY)} \\ \frac{\begin{array}{l} f = \text{fun}(a_1, \dots, a_n)\{i\} \quad m_1 = \text{Push}(m_0, ((a_1 \mapsto v_1), \dots, (a_n \mapsto v_n))) \\ \langle i, m_1 \rangle \rightarrow \langle \text{RETURN}(v), m_2 \rangle \quad m_3 = \text{Pop}(m_2) \quad m_4 = \text{Cleanup}(m_3) \end{array}}{\langle f(v_1, \dots, v_n), m_0 \rangle \rightarrow \langle v, m_4 \rangle} \text{ (EXP-CALL)} \end{array}$$

**FIGURE B.5:** Règles d'évaluation - expressions composées

$$\begin{array}{c}
\frac{\langle i, m \rangle \rightarrow \langle \text{PASS}, m' \rangle}{\langle (i; i'), m \rangle \rightarrow \langle i', m' \rangle} \text{ (SEQ)} \qquad \frac{}{\langle (\text{PASS}; i), m \rangle \rightarrow \langle i, m \rangle} \text{ (PASS)} \\
\\
\frac{}{\langle v, m \rangle \rightarrow \langle \text{PASS}, m \rangle} \text{ (EXP)} \\
\\
\frac{m' = \text{Extend}(m, x, v) \quad \langle i, m' \rangle \rightarrow \langle \text{PASS}, m'' \rangle \quad m''' = \text{Cleanup}(m'' - x)}{\langle \text{DECL } x = v \text{ IN } \{i\}, m \rangle \rightarrow \langle \text{PASS}, m''' \rangle} \text{ (DECL)} \\
\\
\frac{m' = \text{Extend}(m, x, v) \quad \langle i, m' \rangle \rightarrow \langle \text{RETURN}(v_r), m'' \rangle \quad m''' = \text{Cleanup}(m'' - x)}{\langle \text{DECL } x = v \text{ IN } \{i\}, m \rangle \rightarrow \langle \text{RETURN}(v_r), m''' \rangle} \text{ (DEC-RETURN)} \\
\\
\frac{}{\langle \text{IF}(0)\{i_t\}\text{ELSE}\{i_f\}, m \rangle \rightarrow \langle i_f, m \rangle} \text{ (IF-FALSE)} \\
\\
\frac{v \neq 0}{\langle \text{IF}(v)\{i_t\}\text{ELSE}\{i_f\}, m \rangle \rightarrow \langle i_t, m \rangle} \text{ (IF-TRUE)} \\
\\
\frac{}{\langle \text{WHILE}(e)\{i\}, m \rangle \rightarrow \langle \text{IF}(e)\{i; \text{WHILE}(e)\{i\}\}, m \rangle} \text{ (WHILE)} \\
\\
\frac{}{\langle \text{RETURN}(v); i, m \rangle \rightarrow \langle \text{RETURN}(e), m \rangle} \text{ (RETURN)} \qquad \frac{\langle e, m \rangle \rightarrow \langle v, m' \rangle}{m \vdash e \rightarrow m'} \text{ (T-EXP)} \\
\\
\frac{\langle e, m \rangle \rightarrow \langle v, m' \rangle}{(s, g) \vdash x = e \rightarrow (s, (x \mapsto v) :: g)} \text{ (T-VAR)} \qquad \frac{}{m \vdash [] \rightarrow^* m} \text{ (T*-NIL)} \\
\\
\frac{m \vdash p \rightarrow m' \quad m' \vdash ps \rightarrow^* m''}{m \vdash p :: ps \rightarrow^* m''} \text{ (T*-CONS)}
\end{array}$$

FIGURE B.6: Règles d'évaluation

## RÈGLES DE TYPAGE

$\frac{}{\Gamma \vdash i : \text{INT}} \text{ (CST-INT)}$	$\frac{}{\Gamma \vdash d : \text{FLOAT}} \text{ (CST-FLOAT)}$	$\frac{}{\Gamma \vdash \text{NULL} : t*} \text{ (CST-NULL)}$
$\frac{}{\Gamma \vdash () : \text{UNIT}} \text{ (CST-UNIT)}$	$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \text{ (LV-VAR)}$	$\frac{\Gamma \vdash lv : t*}{\Gamma \vdash *lv : t} \text{ (LV-DEREF)}$
$\frac{\Gamma \vdash e : \text{INT} \quad \Gamma \vdash lv : t[]}{\Gamma \vdash lv[e] : t} \text{ (LV-INDEX)}$		$\frac{(l,t) \in S \quad \Gamma \vdash lv : S}{\Gamma \vdash lv.l_S : t} \text{ (LV-FIELD)}$

**FIGURE C.1:** Règles de typage - constantes et variables

$\boxplus \in \{+, -, \times, /, \&,  , ^, \&\&,   , \ll, \gg, \leq, \geq, <, >\}$			$\frac{\Gamma \vdash e_1 : \text{INT} \quad \Gamma \vdash e_2 : \text{INT}}{\Gamma \vdash e_1 \boxplus e_2 : \text{INT}} \text{ (OP-INT)}$
$\boxplus \in \{+, -, \times, /, \leq, \geq, <, >\}$			$\frac{\Gamma \vdash e_1 : \text{FLOAT} \quad \Gamma \vdash e_2 : \text{FLOAT}}{\Gamma \vdash e_1 \boxplus e_2 : \text{FLOAT}} \text{ (OP-FLOAT)}$
$\frac{t \in \{\text{INT}, \text{FLOAT}\}}{\text{EQ}(t)} \text{ (EQ-NUM)}$	$\frac{}{\text{EQ}(t*)} \text{ (EQ-PTR)}$	$\frac{\text{EQ}(t)}{\text{EQ}(t[\ ])} \text{ (EQ-ARRAY)}$	
$\boxplus \in \{=, \neq\} \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t$			$\frac{\text{EQ}(t)}{\Gamma \vdash e_1 \boxplus e_2 : \text{INT}} \text{ (OP-EQ)}$
$\frac{\Gamma \vdash e : \text{INT}}{\Gamma \vdash +e : \text{INT}} \text{ (UNOP-PLUS-INT)}$		$\frac{\Gamma \vdash e : \text{FLOAT}}{\Gamma \vdash +.e : \text{FLOAT}} \text{ (UNOP-PLUS-FLOAT)}$	
$\frac{\Gamma \vdash e : \text{INT}}{\Gamma \vdash -e : \text{INT}} \text{ (UNOP-MINUS-INT)}$		$\frac{\Gamma \vdash e : \text{FLOAT}}{\Gamma \vdash -.e : \text{FLOAT}} \text{ (UNOP-MINUS-FLOAT)}$	
$\boxminus \in \{\sim, !\} \quad \Gamma \vdash e : \text{INT}$			$\frac{}{\Gamma \vdash \boxminus e : \text{INT}} \text{ (UNOP-NOT)}$
$\boxplus \in \{+_p, -_p\} \quad \Gamma \vdash e_1 : t* \quad \Gamma \vdash e_2 : \text{INT}$			$\frac{}{\Gamma \vdash e_1 \boxplus e_2 : t*} \text{ (PTR-ARITH)}$

FIGURE C.2: Règles de typage - opérateurs

$\frac{\Gamma \vdash lv : t}{\Gamma \vdash \&lv : t*}$ (ADDR)	$\frac{\Gamma \vdash lv : t \quad \Gamma \vdash e : t}{\Gamma \vdash lv \leftarrow e : t}$ (SET)	$\frac{\forall i \in [1;n], \Gamma \vdash e_i : t}{\Gamma \vdash [e_1; \dots; e_n] : t[ ]}$ (ARRAY)
$\frac{\forall i \in [1;n], \Gamma \vdash e_i : t_i}{\Gamma \vdash \{l_1 : e_1; \dots; l_n : e_n\} : \{l_1 : t_1; \dots; l_n : t_n\}}$ (STRUCT)		
$\frac{\Gamma \vdash e : (t_1, \dots, t_n) \rightarrow t \quad \forall i \in [1;n], \Gamma \vdash e_i : t_i}{\Gamma \vdash e(e_1, \dots, e_n) : t}$ (CALL)	$\frac{}{\Gamma \vdash \text{PASS}}$ (PASS)	
$\frac{\Gamma \vdash i_1 \quad \Gamma \vdash i_2}{\Gamma \vdash i_1; i_2}$ (SEQ)	$\frac{\Gamma \vdash e : t}{\Gamma \vdash e}$ (EXP)	$\frac{\Gamma \vdash e : t \quad \Gamma, x : t \vdash i}{\Gamma \vdash \text{DECL } x = e \text{ IN } \{i\}}$ (DECL)
$\frac{\Gamma \vdash e : \text{INT} \quad \Gamma \vdash i_1 \quad \Gamma \vdash i_2}{\Gamma \vdash \text{IF}(e)\{i_1\}\text{ELSE}\{i_2\}}$ (IF)	$\frac{\Gamma \vdash e : \text{INT} \quad \Gamma \vdash i}{\Gamma \vdash \text{WHILE}(e)\{i\}}$ (WHILE)	
$\frac{\Gamma \vdash \underline{R} \leftarrow e}{\Gamma \vdash \text{RETURN}(e)}$ (RETURN)	$\frac{\Gamma' = (\Gamma - \underline{R}), \vec{a} : \vec{t}, \underline{R} : t_r \quad \Gamma' \vdash i}{\Gamma \vdash \text{fun}(\vec{a})\{i\} : \vec{t} \rightarrow t_r}$ (FUN)	
$\frac{\Gamma \vdash e : t}{\Gamma \vdash e \rightarrow \Gamma}$ (T-EXP)	$\frac{\Gamma \vdash e : t \quad \Gamma' = (x, t), \Gamma}{\Gamma \vdash x = e \rightarrow \Gamma'}$ (T-VAR)	

**FIGURE C.3:** Règles de typage





## PREUVES

### D.1 Composition de lentilles

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

#### GetPut

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

#### PutGet

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

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

### PutPut

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

□

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

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

- $lv = v$ . Puisque  $\Gamma \vdash v : t*$ , on déduit du lemme 5.2 que  $v = \varphi$  ou  $v = \text{NULL}$ .  
Dans le premier cas, la règle PHI-DEREF s'applique :  $\langle e, m \rangle \rightarrow \langle \widehat{*}\varphi, m \rangle$ . Dans le second, puisque  $\langle * \text{NULL}, m \rangle \rightarrow \Omega_{ptr}$ , on a  $\langle e, m \rangle \rightarrow \Omega_{ptr}$ .
- $\langle lv, m \rangle \rightarrow \langle e', m' \rangle$ . De CTX avec  $C = * \bullet$ , on obtient  $\langle e, m \rangle \rightarrow \langle *e', m' \rangle$ .
- $\langle lv, m \rangle \rightarrow \Omega$ . En appliquant EVAL-ERR avec  $C = * \bullet$ , on obtient  $\langle e, m \rangle \rightarrow \Omega$ .

**LV-INDEX :** De même, on applique l'hypothèse de récurrence à  $lv$ .

- $lv = v$ .  
Comme  $\Gamma \vdash v : t[]$ , on déduit du lemme 5.2 que  $v = [v_1; \dots; v_p]$ . Appliquons l'hypothèse de récurrence à  $e$ .
  - $e = v'$ . Puisque  $\Gamma \vdash e : \text{INT}$ , on réapplique le lemme 5.2 et  $v' = n$ . D'après PHI-ARRAY,  $\langle lv[e], m \rangle \rightarrow \langle [v_1; \dots; v_p][\widehat{n}], m \rangle$ . Deux cas sont à distinguer : si  $n \in [0; p-1]$ , la partie droite vaut  $v_{n+1}$  et donc  $\langle lv[e], m \rangle \rightarrow \langle v_{n+1}, m \rangle$ . Sinon elle vaut  $\Omega_{array}$  et  $\langle lv[e], m \rangle \rightarrow \Omega_{array}$  par EXP-ERR.
  - $\langle e, m \rangle \rightarrow \langle e', m' \rangle$ . En appliquant CTX avec  $C = v[\bullet]$ , on en déduit
  - $\langle lv[e], m \rangle \rightarrow \langle lv[e'], m' \rangle$ .
  - $\langle e, m \rangle \rightarrow \Omega$ . Avec EVAL-ERR sous ce même contexte,  $\langle lv[e], m \rangle \rightarrow \Omega$
- $\langle lv, m \rangle \rightarrow \langle e', m' \rangle$ . On applique alors CTX avec  $C = \bullet[e]$ , et  $\langle lv[e], m \rangle \rightarrow \langle e'[e], m' \rangle$ .
- $\langle lv, m \rangle \rightarrow \Omega$ . Toujours avec  $C = \bullet[e]$ , de EVAL-ERR il vient  $\langle lv[e], m \rangle \rightarrow \Omega$ .

**LV-FIELD :** On applique l'hypothèse de récurrence du théorème 5.2 à  $lv$ .

- $lv = \varphi$  Alors PHI-STRUCT s'applique. Puisque  $(l, t) \in S$ , l'accès au champ  $l$  ne provoque pas d'erreur  $\Omega_{field}$ . Donc  $\langle e, m \rangle \rightarrow \langle \varphi[l], m \rangle$ .
- $\langle lv, m \rangle \rightarrow \langle lv', m' \rangle$  En appliquant CTX avec  $C = \bullet.l_S$ , il vient  $\langle lv, m \rangle \rightarrow \langle lv', m' \rangle$ .
- $\langle lv, m \rangle \rightarrow \Omega$  En appliquant EVAL-ERR avec  $C = \bullet.l_S$ , on a  $\langle lv, m \rangle \rightarrow \Omega$ .

**OP-INT :** Cela implique que  $e = e_1 \boxplus e_2$ . Par le lemme 5.1, on en déduit que  $\Gamma \vdash e_1 : \text{INT}$  et  $\Gamma \vdash e_2 : \text{INT}$ .

Appliquons l'hypothèse de récurrence sur  $e_1$ . Trois cas peuvent se produire.

- $e_1 = v_1$ . On a alors  $\langle e_1, m \rangle \rightarrow \langle v_1, m' \rangle$  avec  $m' = m$ .

On applique l'hypothèse de récurrence à  $e_2$ .

- $e_2 = v_2$  : alors  $\langle e_2, m' \rangle \rightarrow \langle v_2, m'' \rangle$  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  $v_1 \boxplus v_2 = \Omega_{div}$ . Dans ce cas, on a alors par EXP-ERR  $\langle e, m \rangle \rightarrow \Omega_{div}$ .
- $\exists (e'_2, m''), \langle e_2, m' \rangle \rightarrow \langle e'_2, m'' \rangle$ .  
En appliquant CTX avec  $C = v_1 \boxplus \bullet$ , on en déduit  $\langle v_1 \boxplus e_2, m' \rangle \rightarrow \langle v_1 \boxplus e'_2, m'' \rangle$  soit  $\langle e, m \rangle \rightarrow \langle v_1 \boxplus e'_2, m'' \rangle$ .
- $\langle e_2, m' \rangle \rightarrow \Omega$ . De EVAL-ERR avec  $C = v_1 \boxplus \bullet$  vient alors  $\langle e, m \rangle \rightarrow \Omega$ .
- $\exists (e'_1, m'), \langle e_1, m \rangle \rightarrow \langle e'_1, m' \rangle$ . En appliquant CTX avec  $C = \bullet \boxplus e_2$ , on obtient  $\langle e_1 \boxplus e_2, m \rangle \rightarrow \langle e'_1 \boxplus e_2, m' \rangle$ , ou  $\langle e, m \rangle \rightarrow \langle e'_1 \boxplus e_2, m' \rangle$ .
- $\langle e_1, m \rangle \rightarrow \Omega$ . D'après EVAL-ERR avec  $C = \bullet \boxplus e_2$ , on a  $\langle e, m \rangle \rightarrow \Omega$ .

**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,  $\langle + v_1, m \rangle \rightarrow \langle \hat{+} v_1, m \rangle$ , c'est à dire en posant  $v = \hat{+} v_1$ ,  $\langle e, m \rangle \rightarrow \langle v, m \rangle$ .
- 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,  $\langle e, m \rangle \rightarrow \langle e_1 \hat{\boxplus} n, m \rangle$ .

On se réfère ensuite à la définition de  $\hat{\boxplus}$  (page 60) : si  $e_1$  est de la forme  $\varphi[m]$ , alors  $e_1 \hat{\boxplus} n = \varphi[m + n]$ . 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  $\varphi$  pas de la forme  $\varphi'[m]$ ), on a  $e_1 \hat{\boxplus} n = \Omega_{ptr}$ . Donc d'après EXP-ERR,  $\langle e, m \rangle \rightarrow \Omega_{ptr}$ .

**ADDR :**

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

On applique l'hypothèse de récurrence du théorème 5.2 à  $lv$ .

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

$$\left( \overline{\langle \&\varphi, m \rangle \rightarrow \langle \hat{\&\varphi}, m \rangle} (\text{EXP-ADDR OF}) \right)$$

**SET :**

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

**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 :

- $\exists p \in [1; n], e'_p, m : e_1 = v_1, \dots, e_{p-1} = v_{p-1}, \langle e_p, m \rangle \rightarrow \langle e'_p, m' \rangle$ . Alors on peut appliquer CTX avec  $C = [v_1, \dots, v_{p-1}, \bullet, e_{p+1}, \dots, e_n]$ .
- $\exists p \in [1; n], \Omega : e_1 = v_1, \dots, e_{p-1} = v_{p-1}, \langle e_p, m \rangle \rightarrow \Omega$ . Dans ce cas EVAL-ERR est applicable avec ce même  $C$ .
- $e_1 = v_1, \dots, e_n = v_n$ . 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  $C = \{l_1 : v_1, \dots, l_{p-1} : v_{p-1}, \bullet, l_{p+1} : e_{p+1}, \dots, l_n : e_n\}$ ; 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  $C = \bullet(e_1, \dots, e_n)$ . 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  $C = f(v_1, \dots, v_{p-1}, \bullet, e_{p+1}, \dots, e_n)$ . Le seul cas restant est celui où l'expression considérée a pour forme  $f(v_1, \dots, v_n)$  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 Preuve du 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**  $lv$

**Opération unaire**  $e \boxminus e$

**Opération binaire**  $e \boxplus e$

**Pointeur**  $\&lv$

**Affectation**  $lv \leftarrow e$

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

**Tableau**  $[e_1; \dots; e_n]$

**Fonction**  $f$

**Appel de fonction**  $e(e_1, \dots, e_n)$

□

## TODO

- titres dans les chapeaux buggés





## PLAS (TODO)

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.

### E.1 Introduction

Modern operating systems such as GNU/Linux are comprised of several independent programs : the kernel, which handles the machine's hardware as well as trusted core features ; and user applications.

These programs run with different privilege levels : low-level operations can only be executed by the kernel. They also have different views on the memory. If a user program had access to the kernel's internal data structures, it could use it to obtain information it is normally denied, or gain extra privileges (for example getting access to the root account). This is why the kernel stores its code and data in a place in memory that is not accessible to unprivileged programs. On an Intel 32-bit system, it corresponds to addresses higher than 0xc0000000, or 3 GiB.

Because user-mode programs cannot run low-level instructions, they cannot do anything interesting on their own. They have to use the kernel’s facilities to perform input/output or manipulate the operating system’s abstractions (such as creating a new process). To this end, they can communicate with the kernel via the narrow interface of *system calls*.

For example, every time a user process calls the `read()` function, an architecture-specific mechanism invokes the kernel-mode function `sys_read()`. The user passes a file descriptor, a pointer and an amount of bytes to fill and the kernel will actually perform input/output and copy the resulting bytes on the user’s buffer.

But if the user supplies a pointer whose value is in the kernel’s reserved area, this zone will be overwritten, potentially leading to a security breach. In a sense, the kernel has been abused because it accessed the memory with its own privileges instead of the originator’s. This is known as the *confused deputy problem* [Har88].

When implementing a system call, it is thus necessary to forbid direct dereference of a pointer whose value can be controlled by the user. In case of a system call with a pointer argument, one has to dynamically verify that the address provided by the user lies within the process’s memory space.

The object of this work is to detect the places where it is necessary to insert dynamic checks. To this end, we augment the type system of a subset of the C programming language with *type qualifiers* : instead of having a single pointer type, we add syntactic rules to distinguish between safe pointers that can be dereferenced with the `*`, `->` and `[]` operators ; and unsafe pointers that have to go through a special API provided by the kernel, consisting most notably of the two functions `copy_from_user` and `copy_to_user`.

In Section E.2, we present the different techniques that have been used to tackle this issue. Then we present our solution, consisting of an imperative language (Sections E.3 and E.4) that is equipped by a static type system (Section E.5), whose safety is discussed (Section E.6). In Section E.7, we present our implementation, and in Section E.8 how we used it on a bug affecting a video driver in the Linux kernel. Section E.9 describes the limitations of our current approach. Finally, Section E.10 concludes and discusses future work.

## E.2 Related Work

Dynamic taint checking consists in adding taint labels to data, so that every sensitive access is checked at run-time. This goes well with dynamic languages, with the famous example of Perl’s “tainted mode” [Wal00]. It is also possible, but harder, to use this technique for compiled languages [CLO07].

Adding a strong type system to C is the core idea of CCured [NCH<sup>+</sup>05]. It copes with the loose memory model of C by adding dynamic checks when static guarantees



cannot be obtained. However, it requires dynamic instrumentation and is meant to stay activated, incurring a non-negligible run-time cost.

The CQual system[FJKA06] provides a generic type qualifier framework on top of existing type systems. It has been used to infer const annotations [FFA99], tainting analysis for format strings [STFW01] and flow-sensitive properties such as locking invariants [FTA02]. It has even been applied to the class of user/kernel vulnerabilities discussed here [JW04]. Because it adds a qualifier at every level, it is more generic but also more complex than our method.

Abstract interpretation is a generic framework to analyze dynamic properties of programs. As such, it can be used to follow data flow between objects [LM12]. It has the disadvantage of detecting a lot of false positive errors because it requires a relatively precise view of the program. It is also best used for programs that have a single entry point, unlike kernel modules.

Saturn [ABD<sup>+</sup>07] is a system for analyzing system code written in C. It tackles the problem of user pointers using points-to analysis [BA08]. Like abstract interpretation, it aims at being precise at the expense of time in some corner cases.

For the development of the Linux kernel, the Sparse [TTL] custom tool is used to check the origin of pointers. However, it requires the programmer to manually annotate the program, and is not sound.

## E.3 Core Language

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.

$e ::=$	<b>Expressions</b>
$c$	Constant
$lv$	Left-value
$\Box e$	Unary operation

$e \boxplus e$	Binary operation
$\& lv$	Pointer
$lv \leftarrow e$	Assignment
$\{l_1 : e_1; \dots; l_n : e_n\}$	Structure
$[e_1; \dots; e_n]$	Array
$\text{fun}(x_1, \dots, x_n)\{i\}$	Function
$e(e_1, \dots, e_n)$	Function call
$\text{TAINT}(e)$	Tainted value
$lv \Leftarrow_U e$	Load from userspace
$e \Rightarrow_U e$	Store to userspace
$c ::=$	<b>Constants</b>
$n$	Integer
$d$	Float
$\text{NULL}$	Null pointer
$()$	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  $[e_1; \dots; e_n]$  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  $\{l_1 : e_1; \dots; l_n : e_n\}$  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 information to address in a non ambiguous manner any part of the memory.

$lv ::=$	<b>Left-values</b>
$x$	Variable
$*lv$	Dereference
$lv.l_S$	Field access
$lv[e]$	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-time error  $\Omega_{ptr}$ ).

**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 E.9, 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
PASS	Empty instruction
$i_1; i_2$	Sequence
$e$	Expression
DECL $x = e$ IN $\{i\}$	Declaration
IF( $e$ ){ $i$ }ELSE{ $i$ }	Alternative
WHILE( $e$ ){ $i$ }	Loop
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 whole program. On the other hand, a plain expression is only evaluated for its side-effects<sup>1</sup>.

$s ::=$	Toplevel sentences
$x = e$	Global variable
$e$	Expression evaluation
$P ::= (s_1, \dots, s_n)$	<b>Program</b>

## E.4 Evaluation

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

Every syntactic constant  $c$  has a corresponding value  $\hat{c}$ . 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  $\hat{\phantom{x}}$  notation is used to remove the ambiguity.

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

---

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

$v ::=$	<b>Values</b>		
$\hat{c}$	Constant		
$\hat{f}$	Function	$s ::=$	<b>Stack</b>
$\hat{\&} \varphi$	Reference	$[]$	Empty stack
$\widehat{\{l_i : v_i; \dots\}}$	Structure	$\{x_i \mapsto v_i; \dots\} :: s$	Extra frame
$\widehat{[v_i; \dots]}$	Array	$m ::=$	<b>Memory</b>
$\Omega$	Error	( $s,$	Stack,
		$\{x_1 \mapsto v_1; \dots\})$	globals
$\varphi ::=$	<b>Paths</b>		
$a$	Address	$a ::=$	<b>Addresses</b>
$* \varphi$	Dereference	$(n, x)$	Local variable
$\varphi.l$	Field	$(x)$	Global variable
$\varphi[n]$	Index		
$\text{TAINTED}(\varphi)$	Tainted value		
	$\Omega ::=$	<b>Errors</b>	
	$\Omega_{array}$	Buffer overflow	
	$\Omega_{ptr}$	Invalid dereference	
	$\Omega_{div}$	Division by zero	
	$\Omega_{field}$	Bad field	
	$\Omega_{taint}$	Isolation error	
	$\Xi ::=$	<b>Interpreter</b>	
	$\langle e, m \rangle$	Expression, memory	
	$\langle i, m \rangle$	Instruction, memory	
	$\Omega$	Error	

FIGURE E.1: Memory and values

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.

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

Lookup : MEM  $\times$  ID  $\rightarrow$  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  $\varphi$ , that is used to access the corresponding value stored in  $m$ .

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

We omit here a formal definition of  $m[\cdot]_\Phi$ . In a nutshell, it walks the memory to return the value ultimately described by  $\varphi$ . It can raise  $\Omega_{array}$  if a bad array access occurs or  $\Omega_{ptr}$  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  $\varphi$  by  $v$ . The same kind of errors can occur.

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

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

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

It relies on the following operator :

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

**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( $\cdot$ ) instruction.

$$\frac{f = \text{fun}(a_1, \dots, a_n)\{i\} \quad m_1 = \text{Push}(m_0, ((a_1 \mapsto v_1), \dots, (a_n \mapsto v_n))) \quad \langle i, m_1 \rangle \rightarrow \langle \text{RETURN}(v), m_2 \rangle \quad m_3 = \text{Pop}(m_2) \quad m_4 = \text{Cleanup}(m_3)}{\langle f(v_1, \dots, v_n), m_0 \rangle \rightarrow \langle v, m_4 \rangle} \text{ (EXP-CALL)}$$

Push :  $\text{MEM} \times \text{FRAME} \rightarrow \text{MEM}$  adds a new stack frame, and Pop :  $\text{MEM} \rightarrow \text{MEM}$  removes it.

Cleanup :  $\text{MEM} \rightarrow \text{MEM}$  removes references to dangling pointers. It walk all the memory and replaces references to the current stack frame<sup>2</sup> with NULL. For example :

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

**System calls** are emulated through the Taint( $\cdot$ ) 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( $\varphi$ ). If it is the case, then a copy from  $\varphi$  (in the case of  $\leftarrow_U$ ) or to  $\varphi$  (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

---

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

$$\begin{aligned}
C ::= & \bullet \\
& | C \boxplus e \quad | v \boxplus C \quad | \boxminus C \\
& | C \leftarrow e \quad | \varphi \leftarrow C \\
& | \{l_1 : v_1; \dots; l_i : C; \dots; l_n : e_n\} \\
& | [v_1; \dots; C; \dots; e_n] \\
& | C(e_1, \dots, e_n) \quad | f(v_1, \dots, C, \dots, e_n) \\
& | \text{TAINT}(C) \\
& | C \Leftarrow_U e \quad | \varphi \Leftarrow_U C \\
& | C \Rightarrow_U e \quad | v \Rightarrow_U C \\
& | \& C \\
& | *C \quad | C.l_S \quad | C[e] \quad | \varphi[C] \\
& | C; i \\
& | \text{IF}(C)\{i_1\}\text{ELSE}\{i_2\} \\
& | \text{RETURN}(C) \\
& | \text{DECL } x = C \text{ IN}\{i\}
\end{aligned}$$
**FIGURE E.2:** Reduction contexts

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.

$$\frac{\langle e, m \rangle \rightarrow \langle e', m' \rangle}{\langle C[e], m \rangle \rightarrow \langle C[e'], m' \rangle} \text{ (CTX)}$$

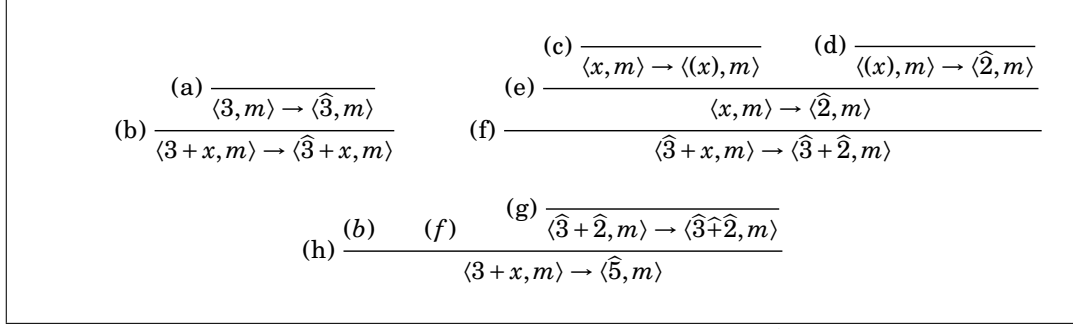
The  $C[\cdot]$  operator corresponds to the last operation : it pastes an expression in place of the unique "hole"  $\bullet$  that appears in each context<sup>3</sup>.

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)

3. In the definition of  $C$ , every alternative produces exactly one nonterminal  $C$  or one terminal  $\bullet$ . So, every derivation tree is linear, and  $\bullet$  appears exactly once in every evaluation context.



**FIGURE E.3:** Evaluation of  $3 + x$  into  $\widehat{5}$ 

- (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,  $\langle \widehat{3} + \widehat{2}, m \rangle \rightarrow \langle \widehat{3} + \widehat{2}, m \rangle$ , 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  $\Xi$ ) propagates to the interpreter. Second, if a sub-expression causes an error, then a bigger expression causes the same error.

$$\frac{}{\langle \Omega, m \rangle \rightarrow \Omega} \text{ (EXP-ERR)} \qquad \frac{\langle e, m \rangle \rightarrow \Omega}{\langle C[e], m \rangle \rightarrow \Omega} \text{ (EVAL-ERR)}$$

**Programs** are evaluated one sentence after another. Declarations add a global variable.

$t ::=$	<b>Types</b>
INT, FLOAT, UNIT	Ground types
$t[]$	Array
$t \ q^*$	Qualified pointer
$S$	Structure
$(t_1, \dots, t_n) \rightarrow t$	Function
$q ::=$	<b>Qualifiers</b>
KERNEL	Kernel data (safe)
USER	User data (unsafe)
$S ::=$	<b>Structure</b>
$\{l_1 : t_1; \dots; l_n : t_n\}$	Simple structure
$\Gamma ::=$	<b>Typing environment</b>
$[]$	Empty environment
$(a, t) :: \Gamma'$	Extension

**FIGURE E.4:** Types and typing environments

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

## E.5 Typing

We add a static type system to SAFESPEAK. The goal is to devise a tractable syntactic method to reject a class of buggy programs.

Our types (Figure E.4) are all ground types; no polymorphism is present. 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 poin-

ters 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 :

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

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 :

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

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),  $\Omega_{ptr}$  is raised.

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

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 :

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

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  $\underline{R}$ .

$$\frac{\Gamma \vdash \underline{R} \leftarrow e}{\Gamma \vdash \text{RETURN}(e)} (\text{RETURN}) \quad \frac{\Gamma' = (\Gamma - \underline{R}), \vec{a} : \vec{t}, \underline{R} : t_r \quad \Gamma' \vdash i}{\Gamma \vdash \text{fun}(\vec{a})\{i\} : \vec{t} \rightarrow t_r} (\text{FUN})$$

Toplevel sentences are typed one after another. Because they can extend an environment, the general form of toplevel typing is  $\Gamma \vdash p \rightarrow \Gamma'$ .

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

The whole set of typing rules can be found in annex in Figures ?? and ??.

## E.6 Type Safety

This type system is sound with respect to the semantics defined in Section E.4 ; that is, we have the desired property that *well-typed programs cannot go wrong*.

Let us start by defining a semantic typing relation between typing environments, memory states, variable names and types.

Variables in memory have store types  $\tau$ , depending only on a memory state  $m$ . They are very similar to the previous types  $t$ , but the arrow function type is replaced by  $\text{FUN}_n$ , denoting a  $n$ -ary function. This is because at runtime, no type information for functions is left.

We define a relation  $m \models x : \tau$  to express that  $x$  is semantically typed  $\tau$  under  $m$  (Figure E.5). Constants are easy to type as there is enough information to recognize them ; functions have a simple dynamic type, only retaining the arity. For memory references, anything that can be dereferenced to a  $\tau$  is a  $\tau \text{ KERNEL } *$  ; and user memory references are “untainted” to access the underlying storage.

$\frac{}{m \models n : \text{INT}} \text{ (S-INT)}$	$\frac{}{m \models d : \text{FLOAT}} \text{ (S-FLOAT)}$	$\frac{}{m \models () : \text{UNIT}} \text{ (S-UNIT)}$
$\frac{m \models m[\varphi]_{\Phi} : \tau}{\hat{\&} \varphi : \tau \text{ KERNEL } *} \text{ (S-REF)}$	$\frac{}{m \models \text{NULL} : t \ q \ *} \text{ (S-NULL)}$	
$\frac{m \models \varphi' : t \text{ KERNEL } *}{m \models \text{TAINTED}(\varphi') : t \text{ USER } *} \text{ (S-TAINTED)}$	$\frac{}{m \models \text{fun}(x_1, \dots, x_n)\{i\} : \text{FUN}_n} \text{ (S-FUN)}$	
$\frac{\forall i \in [1; n]. m \models v_i : t}{m \models [v_1; \dots; v_n] : t[\ ]} \text{ (S-ARRAY)}$	$\frac{\forall i \in [1; n]. m \models v_i : t_i \quad S = \{l_1 : t_1; \dots; l_n : t_n\}}{m \models \{l_1 : v_1; \dots; l_n : v_n\} : S} \text{ (S-STRUCT)}$	

FIGURE E.5: Semantic typing rules

We also need a  $\text{Comp}(\cdot, \cdot)$  relation to express compatibility between types of expressions and semantic types (of values) : ground types are compatible with themselves, type constructors are compatible if their components are compatible, and we add a rule for functions :

$$\text{Comp}(\text{FUN}_n, (t_1, \dots, t_n) \rightarrow t)$$

Then a memory state  $m$  is well-typed under a typing environment  $\Gamma$ , which we will write  $\Gamma \models m$ , if the semantic types of visible variable coincide with their static

types<sup>4</sup> :

$$\forall x \in \text{dom}(\Gamma). \exists \tau, t. \begin{cases} \text{Comp}(\tau, t) \\ m \models x : \tau \\ \Gamma \vdash x : t \end{cases}$$

**Théorème E.1** (Progress). *Let us suppose that  $\Gamma \vdash e : t$ .*

*Then let  $m$  be a memory state such that  $\Gamma \models m$ .*

*Then one of the following cases holds :*

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

*That is, either :*

- *$e$  is fully evaluated*
- *an evaluation step is possible, that preserves memory compatibility*
- *a division, array or pointer error occurs*

**Théorème E.2** (Preservation). *If an expression is well-typed, then an evaluation step does not modify its type :*

*If  $\Gamma \vdash e : t$ ,  $\Gamma \models m$  and  $\langle e, m \rangle \rightarrow \langle e', m' \rangle$ , then  $\Gamma \vdash e' : t$ .*

This proves that no term stays stuck, and that that typing is well-behaved we respect to the semantics. It means that the type is really a contract between expressions and functions : if their evaluation converge, then a value of the inferred type will be produced.

## E.7 Implementation

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.

The first one consists in preprocessing the C files that compose the kernel. We have to manually define macros that are usually defined by the kernel's build system. The configuration system for Linux uses `cpp` macros too (named `CONFIG_*`), so we have make a choice of configuration file. We just used the default one, but to analyze

---

4. `dom(Γ)` returns the set of variables present in  $\Gamma$ , which also corresponds to visible variables.

larger parts of the kernel, it may be necessary to define a “maximal” configuration file (which is impossible because of incompatibilities between some options).

Once we have a set of preprocessed C files<sup>5</sup>, we run it through the `c2newspeak` utility that compiles it to the Newspeak. In that step, types and names are resolved, and the program is decorated with enough annotation to make the next steps context-free.

Because Linux is written in the GNU C dialect, it required to adapt `c2newspeak`. Two kinds of extensions were encountered : the main particularity is the `__attribute__` notation that decorates declarations. In the vast majority of cases, they can be ignored as they are either compiler hints (e.g. `used` disables the “variable is not used” warning), optimizations (e.g. `hot` groups variable together so they are put in cache together) or too low-level to be taken in account (e.g. `aligned(n)` specifies that a variable needs to be aligned on `n` bits). Besides attributes, GNU C allows advanced uses of `typeof`, as well as statement-expressions (expressions of the form `({stmt})`), and first-class labels (`&&lbl` and `goto *p` constructs) that had to be properly supported.

Newspeak is designed for static analysis by abstract interpretation and has quite a low-level view on programs. For example, it does not distinguish between field access and indexed access : as in an assembly language, it represents both of them with a numeric offset (static or dynamic) from the start of a memory zone. To remove such an ambiguity, we have to hook into `c2newspeak`’s internal data structures, where this information is still available, and with it we can emit `SAFESPEAK` code.

Then, different files are linked together. This step consists in ensuring that the assumptions made by the different files are consistent. static variables, not visible outside the file they are defined in, are renamed so that they have a unique name.

Finally, the implementation of an inference algorithm for the above type system is quite straightforward. As it is similar enough to simply typed lambda calculus, we can use a variant of Damas and Milner’s Algorithm W[DM82]. As usual, instead of explicit substitutions we rely on sharing of references to implement unification. As our type system is monomorphic, we raise an error whenever free type variables occur.

At the end of this step, we have either a fully annotated program, or a type error.

Let us run the analysis on a small example :

```
int f(int *x) { return (*x + 1); }
```

Running our analyzer outputs a fully annotated program :

```
% ptrtype example.c
```

---

5. As `cpp`’s `#include` statement is textual, headers get included in every file, leading to a lot of duplicate work for the next passes.

```

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
};
}

```

The `coerce[a,b]` operator is a Newspeak artifact to detect integer overflows when doing a value analysis by abstract interpretation. For the purposes of this paper, it can be seen as the identity function typed  $(\text{INT}) \rightarrow \text{INT}$ .

*A contrario*, when it is not possible to type check, the analyzer bails out with an error.

```

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

```

The `/*!nkp userptr p */` comment is interpreted by the analyzer and causes it to unify the type of `p` with `t USER *`, effectively forcing its qualifier.

```

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

```

We just signal the location where a unification failed, but it is possible to keep track of all the unification sites to create a trace, easier to debug.

## E.8 Experimentation

We now describe the case of a linux kernel video driver that involved a user/kernel pointer bug. It is listed on the <http://freedesktop.org> bug tracker as #29340.

To switch graphic modes, GPU drivers can support Kernel Mode Setting (KMS). To configure a device, the user has to communicate with the kernel driver with a mechanism known as *ioctl*s Named after the function `ioctl()` for Input/Output Control,

they are a sort of device-specific system call mechanism. As such, these class of functions manipulate user-supplied pointers and need to be written with care.

The following code is part of the KMS driver for AMD Radeon devices :

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

We can see that the argument data is converted to a struct `drm_radeon_info` \*. A pointer `value_ptr` is extracted from its value field, and finally this pointer is dereferenced.

However, the argument data is a pointer to a (kernel-allocated) structure of the following type, whose fields come from a userland `ioctl` invocation.

```
/* from include/drm/radeon_drm.h */
struct drm_radeon_info {
    uint32_t    request;
    uint32_t    pad;
    uint64_t    value;
};
```

To show that this is buggy, we annotated the function `radeon_info_ioctl` so that its second parameter is a kernel pointer to a structure containing a `USER` field value. This is possible because we erase the types present in the C program before



translation. So, an pointer cannot be distinguished from a integer casted to a pointer. Using this, we obtain a type error at line 16.

The good way to proceed was recorded as commit number d8ab3557 by replacing line 16 by :

```
if (DRM_COPY_FROM_USER(&value, value_ptr,
                      sizeof(value)))
    return -EFAULT;
```

(DRM\_COPY\_FROM\_USER being a simple wrapper macro for `copy_from_user`). In that case, we obtain no type error.

## E.9 Limitations

Our system has several limitations.

First, as we used static typing, it is impossible to capture a lot of constructs that are possible or even idiomatic in C : unions, type casts, type punning. The first two ones are equivalent constructs. We can replace each cast from a type  $t_1$  to a type  $t_2$  by the call of a function `cast $t_1, t_2$`  but this means adding a flaw in the type system : it would be typed  $(t_1) \rightarrow t_2$ , in other words the dreaded type  $\alpha \rightarrow \beta$  of `Obj.magic` in OCaml or `unsafeCoerce` in Haskell.

Type punning consists in modifying the bit pattern of some data to manipulate it in an efficient way. For example, it is common to define a set of macros to access low-level parts of IEEE754 floats, for exemple their exponent. Usually, a combination of unions and bit masks is used. In such cases, strong typing is of course impossible. To handle this, we could wrap the manipulation in a macro and add an explicit type information such as `float_exponent : (FLOAT)  $\rightarrow$  INT`.

The operational semantics uses runtime support for several cases. Contrary to C, we dynamically check for buffer overflows and NULL dereference, but this is not critical to our approach : if we suppose that the programs we analyze are correct with respect to these properties, we could provide a lower-level implementation without those checks.

A more concerning place where we add a runtime behaviour is when the memory is searched for dangling pointers at every function return (with the `Cleanup(·)` operator). Removing this check makes the analysis unsound, because it is then possible to make reference to the current stack frame with a former one. However, contrary to type punning, there are few cases where such a behaviour is desired. If we can have a static guarantee that the address of a local variable will not be accessible after the function has returned, we can remove the `Cleanup(·)` step. This can be done with a dedicated static analysis[DDMPn10].

Dynamic memory allocation, embodied in userspace by the `malloc()` and `free()` functions, has not been implemented. Similarly to `get_{from,to}_user()`, it manipulates data as memory zones, with a pointer and a size in bytes. We would have to define a higher level operator taking an expression and returning the address of a freshly allocated cell in a heap. Then typing is straightforward :

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \text{ALLOC}(e) : t \text{ KERNEL} *} \qquad \frac{\Gamma \vdash e : t \text{ KERNEL} *}{\Gamma \vdash \text{FREE}(e)}$$

To define evaluation, we can expand the memory states to include an explicit heap of values, mapping unique labels to values. Deallocation is problematic because we rely on the programmer not to access deallocated memory zones. Once again, it is possible to dedicate an analysis like [DDMPn10] to enforce this. It is also sound, albeit impractical, to ignore deallocation constructs.

Another limitation is that our system does not allow recursive structures, containing a pointer to a field of the same type. For simple (not mutually) recursive structures, it is possible to extend the grammar of type annotation to allow the following, where  $p$  binds a pointer type to the structure.

$$S ::= \dots \mid \text{fix}(p \rightarrow S)$$

## E.10 Conclusion

We showed that type theory can be a useful tool for verifying the absence of certain run-time 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

2.1	Cadres de pile . . . . .	11
2.2	Les différents <i>rings</i> . . . . .	13
2.3	Implantation de la mémoire virtuelle . . . . .	14
2.4	Mécanisme de mémoire virtuelle. . . . .	14
2.5	Espace d'adressage d'un processus . . . . .	15
2.6	Appel de <code>gettimeofday</code> . . . . .	18
2.7	Zones mémoire . . . . .	18
2.8	Implantation de l'appel système <code>gettimeofday</code> . . . . .	19
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)). . . . .	23
3.2	Domaine des signes . . . . .	23
3.3	Domaine des intervalles . . . . .	24
3.4	Quelques domaines abstraits . . . . .	25
3.5	Session Python présentant le typage dynamique . . . . .	28
3.6	Transtypage en Java . . . . .	29
3.7	Fonction de concaténation de listes en OCaml. . . . .	29
4.1	Fonctionnement d'une lentille . . . . .	40
4.2	Fonctionnement d'une lentille indexée . . . . .	41
4.3	Composition de lentilles . . . . .	42
4.4	Syntaxe - expressions . . . . .	45
4.5	Syntaxe - instructions . . . . .	46
4.6	Syntaxe - opérateurs . . . . .	46
4.7	Valeurs . . . . .	49
4.8	Composantes d'un état mémoire . . . . .	49
4.9	Opérations de pile . . . . .	52
4.10	Nettoyage d'un cadre de pile . . . . .	53
4.11	Contextes d'exécution . . . . .	56
4.12	Substitution dans les contextes d'évaluation . . . . .	58
4.13	Évaluation des left-values. . . . .	59
4.14	L'appel d'une fonction. La taille de la pile croît de gauche à droite, et les réductions se font de haut en bas. . . . .	62

5.1	Programmes bien et mal formés . . . . .	69
5.2	Types et environnements de typage . . . . .	70
6.1	Ajouts liés aux pointeurs utilisateurs . . . . .	84
6.2	Changements liés aux qualificateurs de types . . . . .	86
7.1	Décomposition d'un compilateur : front-ends, middle-end, back-ends . . .	97
7.2	Compilation depuis Newspeak . . . . .	100
7.3	Lambda calcul simplement typé avec entiers, flottants et couples . . . . .	102
7.4	Arbre d'inférence : règles à utiliser . . . . .	103
7.5	Arbre d'inférence complet . . . . .	104
7.6	Unification par partage . . . . .	106
7.7	Compilation d'un programme C - avant . . . . .	106
7.8	Unification : partage . . . . .	108
7.9	Unification par mutation de références . . . . .	109
7.10	Cycle dans le graphe de types . . . . .	109
7.11	Compilation d'un programme C - après . . . . .	111
8.1	Bug freedesktop.org #29340 . . . . .	114
B.1	Règles d'évaluation - contextes . . . . .	126
B.2	Règles d'évaluation - erreurs . . . . .	127
B.3	Règles d'évaluation - constantes et left-values . . . . .	127
B.4	Règles d'évaluation - opérations . . . . .	127
B.5	Règles d'évaluation - expressions composées . . . . .	127
B.6	Règles d'évaluation . . . . .	128
C.1	Règles de typage - constantes et variables . . . . .	129
C.2	Règles de typage - opérateurs . . . . .	130
C.3	Règles de typage . . . . .	131
E.1	Memory and values . . . . .	147
E.2	Reduction contexts . . . . .	150
E.3	Evaluation of $3 + x$ into $\hat{5}$ . . . . .	151
E.4	Types and typing environments . . . . .	152
E.5	Semantic typing rules . . . . .	154

## LISTE DES DÉFINITIONS

4.1	Définition (Lentille)	39
4.2	Définition (Lentille indexée)	40
4.3	Définition (Composition de lentilles)	42
4.4	Définition (Recherche de variable)	50
4.5	Définition (Manipulations de pile)	51
4.6	Définition (Accès à une liste d'associations)	52
4.7	Définition (Accès par adresse)	53
4.8	Définition (Accès par champ)	53
4.9	Définition (Accès par indice)	54
4.10	Définition (Accès par chemin)	54
4.11	Définition (Évaluation d'une expression)	57
4.12	Définition (Évaluation d'une left-value)	57
5.1	Définition (Compatibilité mémoire)	79
5.2	Définition (Typage de la mémoire)	79

## LISTE DES THÉORÈMES ET PROPRIÉTÉS

3.1	Théorème (de Rice)	21
5.1	Lemme (Inversion)	77
5.2	Lemme (Formes canoniques)	78
5.1	Théorème (Progrès)	79
5.2	Théorème (Progrès pour les left-values)	80
5.3	Lemme (Permutation)	80
5.4	Lemme (Affaiblissement)	80
5.5	Lemme (Substitution)	80
5.3	Théorème (Préservation)	80
6.1	Théorème (Progrès pour les types qualifiés)	87
6.2	Théorème (Préservation pour les types qualifiés)	87

E.1	Théorème (Progress) . . . . .	155
E.2	Théorème (Preservation) . . . . .	155

## RÉFÉRENCES WEB

- [🌐<sup>1</sup>] The Objective Caml system, documentation and user's manual – release 3.12  
<http://caml.inria.fr/pub/docs/manual-ocaml/>
- [🌐<sup>2</sup>] Haskell Programming Language – Official Website  
<http://www.haskell.org/>
- [🌐<sup>3</sup>] Python Programming Language – Official Website  
<http://www.python.org/>
- [🌐<sup>4</sup>] Perl Programming Language – Official Website  
<http://www.perl.org/>
- [🌐<sup>5</sup>] Sparse - a Semantic Parser for C  
[https://sparse.wiki.kernel.org/index.php/Main\\_Page](https://sparse.wiki.kernel.org/index.php/Main_Page)
- [🌐<sup>6</sup>] CIL - C Intermediate Language  
<http://kerneis.github.com/cil/>
- [🌐<sup>7</sup>] The C - - language  
<http://www.cminusminus.org/>
- [🌐<sup>8</sup>] 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. 99
- [ABD<sup>+</sup>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. 143
- [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. 143
- [BBC<sup>+</sup>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. 14
- [BDH<sup>+</sup>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. 99
- [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. 31

- [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, 25
- [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<sup>+</sup>05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREE analyzer. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005. 25
- [CCF<sup>+</sup>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. 142
- [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. 26
- [CMP03] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly, 2003. 32
- [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. 32
- [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. 159, 160
- [Dij82] Edsger W. Dijkstra. Why numbering should start at zero. circulated privately, August 1982. 43

- [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. 156
- [DRS00] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV : Towards a realistic tool for statically detecting all buffer overflows in C, 2000. 31
- [EH94] Ana Erosa and Laurie J. Hendren. Taming control flow : A structured approach to eliminating goto statements. In *Proceedings of 1994 IEEE International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994. 100
- [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. 30, 115, 143
- [FGM<sup>+</sup>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. 39
- [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. 149
- [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. 30, 143
- [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. 30
- [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. 30, 143
- [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. 14
- [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. 19, 113, 142
- [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. 99, 100, 115, 155
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, October 1969. 30
- [Int10] Intel, Santa Clara, CA, USA. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, June 2010. 9, 15
- [ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999. 88, 97
- [JW04] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004. 30, 143
- [KcS07] Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci.*, 174(7) :79–104, 2007. 30
- [Ker81] Brian W. Kernighan. Why Pascal is not my favorite programming language. Technical report, AT&T Bell Laboratories, April 1981. 29
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988. 42, 97
- [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. 98
- [LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml : A notation for detailed design, 1999. 31
- [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. 143
- [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. 30
- [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. 98
- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, December 1978. 29
- [Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3) :12, May 2008. 26
- [NCH<sup>+</sup>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. 142
- [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. 99
- [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. 48
- [OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. 32
- [One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 1996. 9, 88
- [Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 29

- [PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003. 32
- [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. 98
- [Pl04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61 :17–139, 2004. 148
- [PTS<sup>+</sup>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
- [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. 32
- [SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21 :2003, 2003. 32
- [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. 25
- [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. 98
- [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. 30, 143

- [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. 90
- [Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007. 7
- [TTL] Linus Torvalds, Josh Triplett, and Christopher Li. Sparse - a semantic parser for C. <https://sparse.wiki.kernel.org>. 143
- [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. 39
- [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. 32
- [Wal00] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2000. 32, 142
- [WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1) :38–94, November 1994. 149
- [XACH07] 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. 32