

Analyse statique de logiciel système par typage statique fort

Application au noyau Linux

Étienne Millon

7 août 2012

Abstract ici.

ABSTRACT

iii

Cassedéti

TABLE DES MATIÈRES

Table des matières	vi
1 Introduction	1
I Méthodes formelles pour la sécurité	3
2 Systèmes d'exploitation	5
2.1 Rôle d'un système d'exploitation	5
2.2 Architecture Intel	6
2.2.1 Assembleur	6
2.2.2 Fonctions et conventions d'appel	7
2.2.3 Tâches, niveaux de privilèges	9
2.2.4 Mémoire virtuelle	9
2.3 Cas de Linux	10
2.3.1 Appels système	11
2.4 Sécurité des appels système	13
3 Typage	15
3.1 Présentation et but	15
3.2 Taxonomie	15
3.2.1 Dynamique, statique, mixte	15
3.2.2 Fort, faible, sound	18
3.2.3 Polymorphisme	19
3.2.4 Expressivité, garanties, types dépendants	21
3.3 Exemples	21
3.3.1 Faible dynamique : Perl	21
3.3.2 Faible statique : C	21
3.3.3 Fort dynamique : Python	21
3.3.4 Fort statique : OCaml	21
3.3.5 Fort statique à effets typés : Haskell	21
3.3.6 Theorem prover : Coq	21
4 État de l'art	23
II Typage statique de langages impératifs	25
5 Un premier système de types	27
5.1 Syntaxe	27

5.2	Sémantique	28
5.2.1	Graphe de flot de contrôle	28
5.2.2	État mémoire	29
5.2.3	Left values	29
5.2.4	Jugements	30
5.2.5	Sémantique des left-values	30
5.2.6	Sémantique des expressions	30
5.2.7	Sémantique des instructions	31
5.2.8	Sémantique des conditions	31
5.3	Règles de typage	32
5.3.1	Programme	32
5.3.2	Flot de contrôle	32
5.3.3	Left values	33
5.3.4	Expressions	33
5.3.5	Fonctions	34
5.3.6	Instructions	35
5.4	Limitations	35
5.4.1	Programmes non typables	35
5.4.2	Incohérences	35
6	Système avec le qualificateur "user"	37
6.1	Éditions et ajouts	37
6.2	Passage sur le cas d'étude	37
6.3	Propriété d'isolation mémoire	37
7	Système avec le qualificateur "sz"	39
7.1	But	39
7.2	Approche	40
7.3	Annotation de <code>string.h</code>	40
7.3.1	Fonctions de copie	41
7.3.2	Fonctions de concaténation	41
7.3.3	Fonctions de comparaison	41
7.3.4	Fonctions de recherche	42
7.3.5	Fonctions diverses	42
7.4	Typage des primitives	43
7.5	Extensions au système de types	43
7.6	Résultats	43
III	Expérimentation	45
8	Implantation	47
8.1	Langages intermédiaires	47
8.2	Newspeak	49
8.3	Chaîne de compilation	49
8.3.1	Prétraitement	49

8.3.2	Compilation (levée des ambiguïtés)	50
8.3.3	Annotations	50
8.3.4	Implantation de l'algorithme de typage	50
9	Étude de cas : un driver graphique	57
9.1	Description du problème	57
9.2	Principes de l'analyse	58
9.3	Implantation	58
9.4	Conclusion	59
10	Conclusion	61
10.1	Limitations	61
10.2	Perspectives	61
A	Code du noyau	63
	Table des figures	67
	Références web	69
	Bibliographie	71

CHAPITRE



INTRODUCTION

Première partie

Méthodes formelles pour la sécurité

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 permettant 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 un utilisateur d'utiliser 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 CPU 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 voulait 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 à s'exécuter en mode privilégié, 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 elle va s'exécuter. 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.

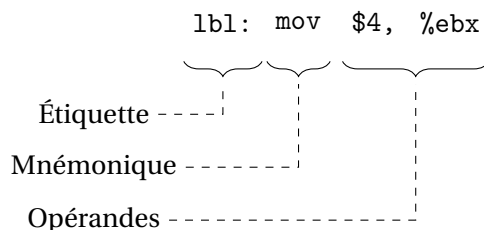
La référence pour la description de l'assembleur Intel est la documentation du constructeur [Int] ; une bonne explication de l'agencement dans la pile peut aussi être trouvée dans [One96].

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.

Les registres principaux 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. ESP est le pointeur de pile (*stack pointer*), qui a un rôle particulier dans les instructions push et pop. EBP sert de point de repère dans les appels de fonction. EIP contient l'adresse de l'instruction courante.

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. Elle peut aussi être précédée d'une étiquette, qui correspondra à l'adresse de cette instruction.



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 décrémente le pointeur de pile `ESP` de 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`.

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 :

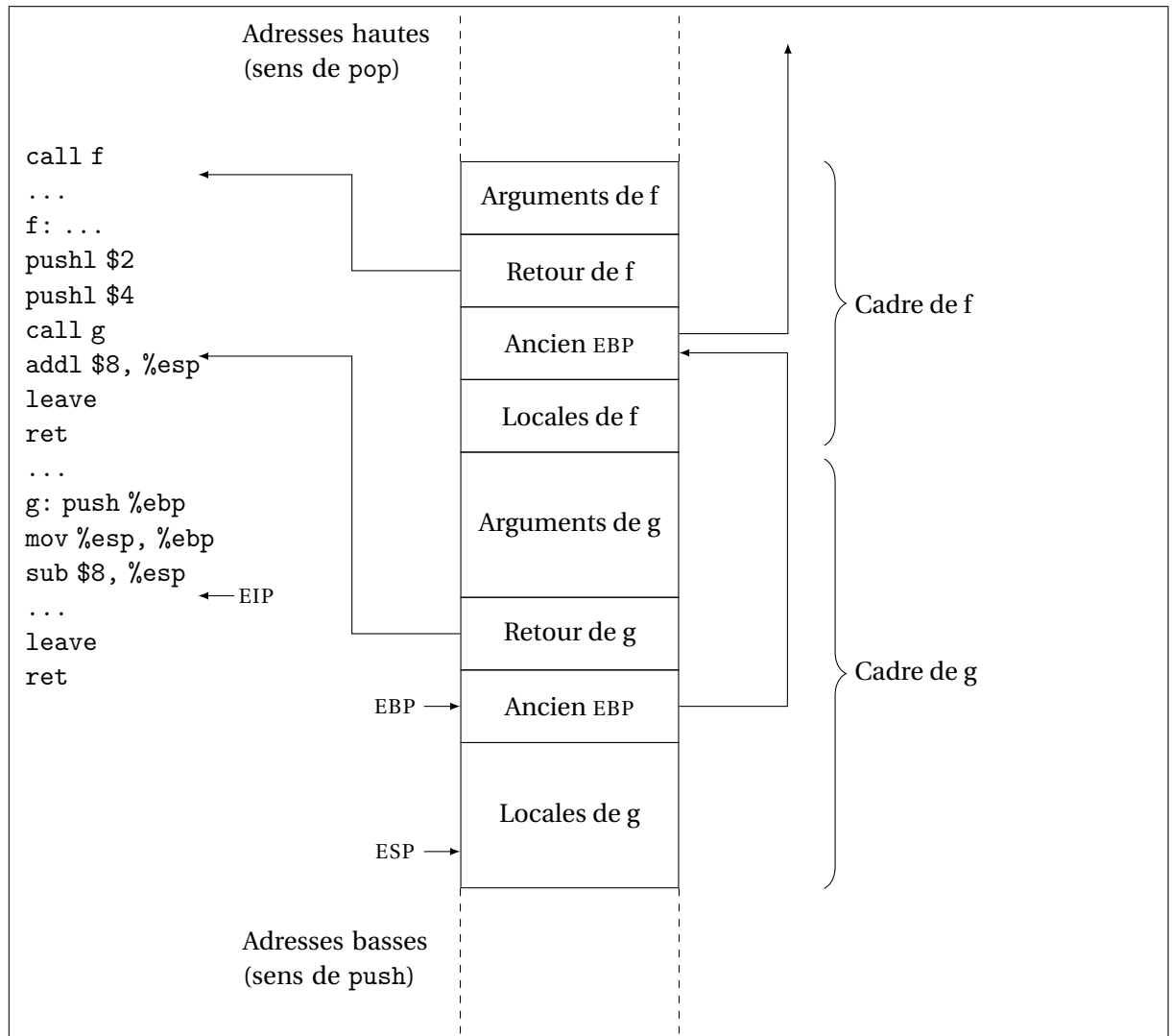


FIGURE 2.1 – Cadres de pile.

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

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

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

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



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.

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.

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. Si à cet endroit le code sauvegarde les registres et restaure un autre ensemble de registres, on peut 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 fuiter ; et s'il peut y écrire, il peut en détourner l'exécution.

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

Ce mécanisme est contrôlé par valeur du registre `CR3` : les 10 premiers bits d'une adresse sont un index dans le répertoire de pages qui commence à l'adresse contenue dans `CR3`. À cet index, se

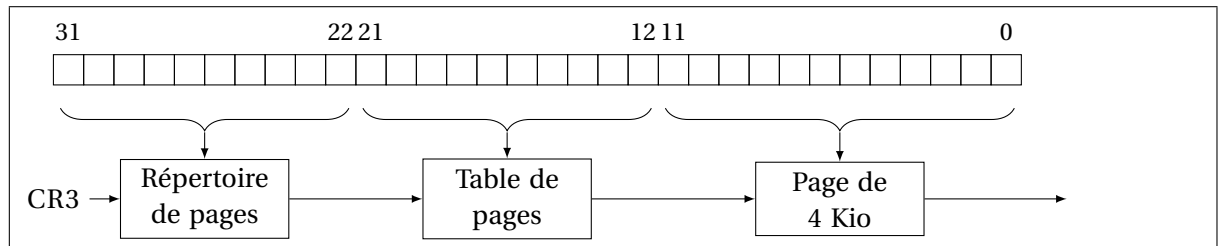


FIGURE 2.3 – Implantation de la mémoire virtuelle

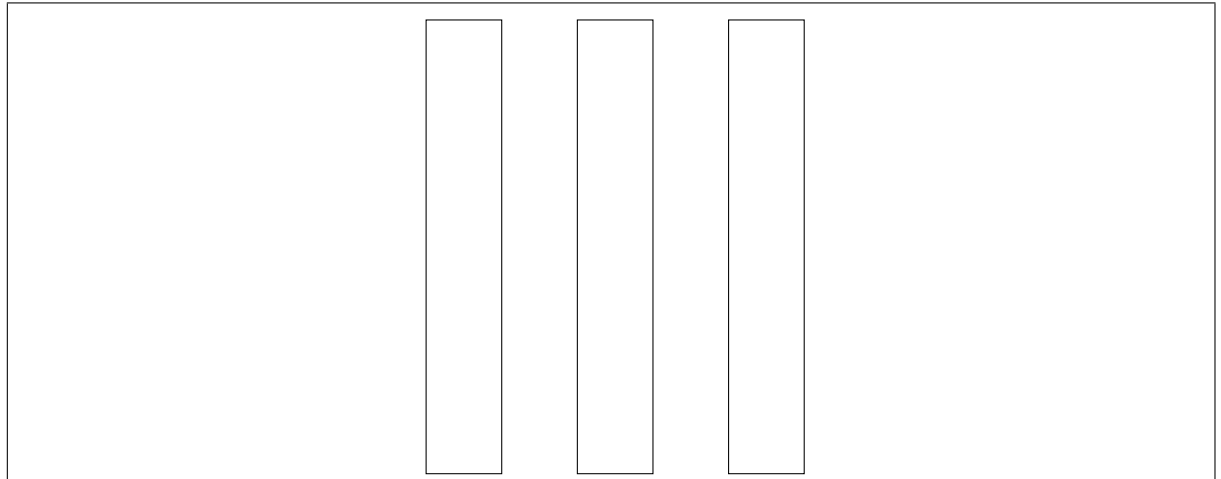


FIGURE 2.4 – Mécanisme de mémoire virtuelle.

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).

2.3 Cas de Linux

Dans cette section, nous allons voir comment ses mécanismes sont implantés dans le noyau Linux. Une description plus détaillée pourra être trouvée dans [BP05], 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 peut 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). Au moment de changer le processus en cours, l'ordonnanceur charge une valeur propre au nouveau processus dans CR3. Les adresses basses (inférieures à `PAGE_OFFSET = 3 Gio = 0xc0000000`) sont réservées à l'utilisateur. On y trouvera par exemple :

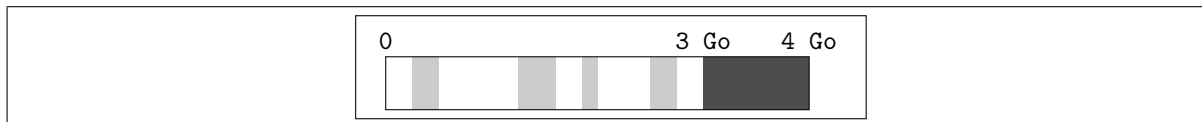


FIGURE 2.5 – L'espace d'adressage d'un processus. En gris clair, les zones accessibles à tous les niveaux de privilèges : code du programme, bibliothèques, tas, pile. En gris foncé, la mémoire du noyau, réservée au mode privilégié.

- 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 [Int].

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 processeur¹, 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 anyway
    pushl %eax                                     # save orig_eax
```

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*.

```

CFI_ADJUST_CFA_OFFSET 4
SAVE_ALL
GET_THREAD_INFO(%ebp)
                                # system call tracing in operation / emulation
testl $_TIF_WORK_SYSCALL_ENTRY, TI_flags(%ebp)
jnz syscall_trace_entry
cmpl $(nr_syscalls), %eax
jae syscall_badsys
syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,PT_EAX(%esp)      # store the return value
    # ...
    INTERRUPT_RETURN

```

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

Dans l'implantation de l'appel système

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

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

```

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

```

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

Retour vers le ring 3

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

```
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

2.4 Sécurité des appels système



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

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

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

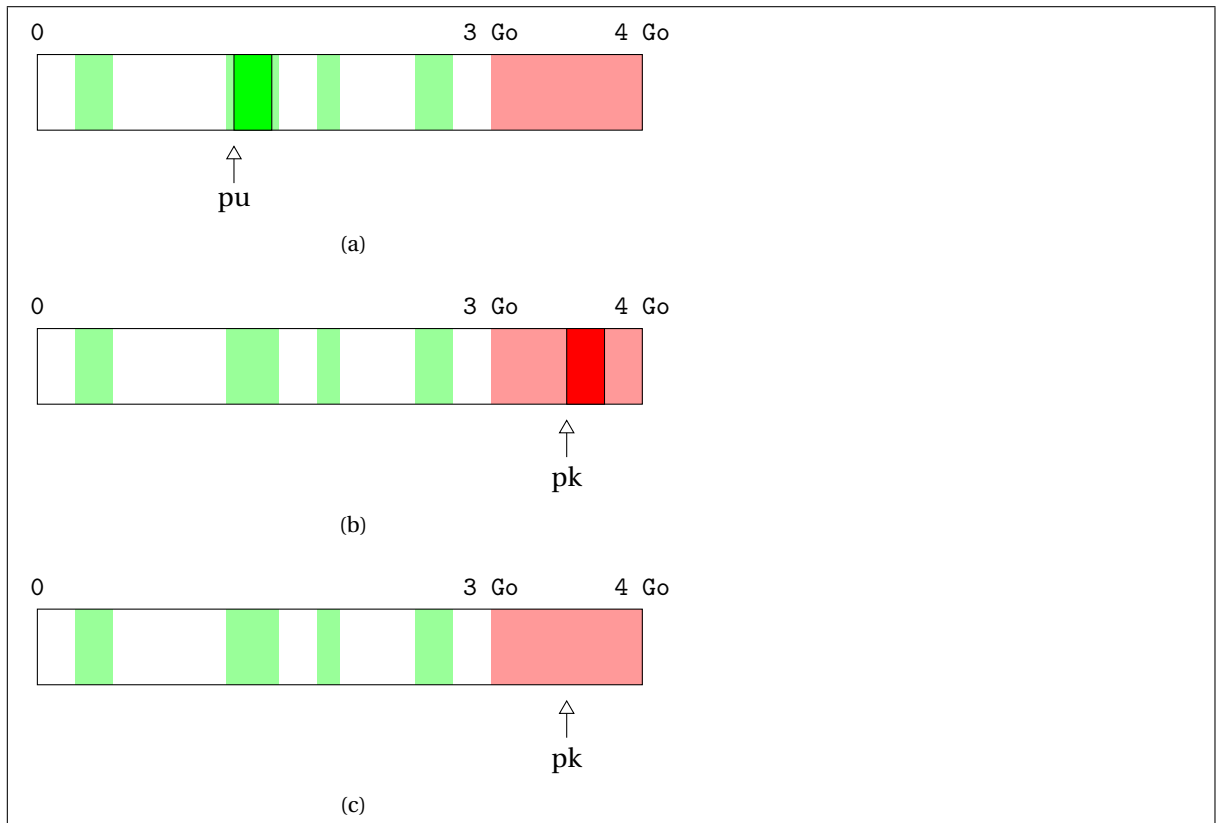
Il y a deux grands types d'appels systèmes : d'une part, ceux qui renvoient un simple nombre, comme getpid qui renvoie le numéro du processus appelant.

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

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

Mais la plupart des appels systèmes communiquent de l'information de manière indirecte, à travers un pointeur. Ainsi la fonction gettimeofday a pour prototype :

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```







Dans le cas d'un appel système qui ne fait que renvoyer un nombre, il n'y a pas de difficulté. En revanche, certains appels système remplissent une structure avec leurs résultats. C'est alors la responsabilité de l'appelant (le programmeur utilisateur) d'allouer une zone mémoire correcte pour contenir le résultat (figure 2.6).

Puisque c'est le noyau qui implémente l'appel système, le déréférencement se déroule en *ring 0*. Il se pose alors le problème suivant : le noyau va pouvoir écrire à toute adresse fournie par l'utilisateur.

Notons que dans ce cas, c'est le noyau qui remplit la structure : le déréférencement se fait en *ring 0*.

TYPAGE

Dans ce chapitre, nous explorons la notion de type dans les langages de programmation. Tout d'abord pourquoi elle existe et en quoi elle aide à rendre les programmes plus sûrs. Il y a autant de système de types que de langages de programmation, donc nous présenterons ensuite une taxonomie de ces systèmes en les regroupant par caractéristiques communes. Cette classification sera appuyée par des exemples de code C[ISO99, KR88], OCaml[¹] [CMP03], Haskell[²] [PJ03, OGS08], Python[³] et Perl[⁴] [Wal00].

3.1 Présentation et but

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".

3.2 Taxonomie

La définition d'un langage de programmation introduit la plupart du temps celle d'un système de types. Il y a donc de nombreux systèmes de types différents, dont nous pouvons donner une classification sommaire.

3.2.1 Dynamique, statique, mixte

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).

```
>>> 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 3.1 – Session Python présentant le typage dynamique

Typage dynamique

La première est le typage *dynamique*. Pour différencier les différents types de données, on ajoute une étiquette à chaque valeur. Dans tout le programme, on ne manipulera que des valeurs étiquetées, c'est à dire des couples (donnée, nom de type). Si on veut réaliser l'opération $(0x00000001, Int) + (0x0000f000, Int)$, on vérifie tout d'abord qu'on peut réaliser l'opération $+$ entre deux *Int*. Ensuite on réalise l'opération elle-même, qu'on étiquette avec le type du résultat : $(0x0000f001, Int)$. Si au contraire on tente d'ajouter deux adresses $(0x2e8d5a90, Addr) + (0x76a5e0ec, Addr)$, la vérification échoue et l'opération s'arrête avec une erreur.

La figure 3.1 est une session interactive Python qui illustre le typage dynamique. Chaque variable, en plus de sa valeur, possède une étiquette qui permet d'identifier le type de celle-ci. On peut accéder au type d'une valeur x avec la construction `type(x)`.

Au moment de réaliser une opération comme $+$, l'interpréteur Python vérifie les types des opérandes : s'ils sont compatibles, il crée une valeur de résultat, et sinon il lève une exception.

Comme l'implémentation elle-même des fonction a accès aux informations de type, elle peut faire des traitements particuliers. Par exemple, pour l'addition de a (de type entier) et de c (de type flottant), la fonction d'addition va d'abord convertir a en flottant, puis réaliser l'addition dans le domaine des flottants.

Typage statique

Le typage dynamique est simple à comprendre puisque toutes les vérifications se font dans la sémantique dynamique (ie, à l'exécution). C'est à double tranchant : d'une part, cela permet d'être


```
def f(b):  
    x = None  
    r = None  
    if b:  
        x = 1  
    else:  
        x = lambda y: y + 1  
    b = not b  
    if b:  
        r = x (1)  
    else:  
        r = x + 1  
    return r
```

FIGURE 3.2 – Fonction Python non typable statiquement.

plus flexible, mais d'autre part, cela permet à des programmes incorrects d'être exécutés.

On peut lire le code source d'un programme et essayer de "deviner" quels seront les types des différentes expressions. Dans certains cas, cela n'est pas possible (fig 3.2) ; mais lorsqu'on peut conclure cela élimine la nécessité de faire les tests de type dynamiques car on a réalisé le typage *statiquement*.

Bien sûr, deviner n'est pas suffisant : il faut formaliser cette analyse. Dans le cas dynamique, ce sont les fonctions elles-mêmes qui réalisent les tests de type et qui appliquent des règles comme "si les arguments ont pour type `int` alors la valeur de retour a pour type `int`" : la fonction qui réalise ce test sur les valeurs. Dans le cas statique, c'est le compilateur (ou l'interpréteur) qui réalise ce test sur les expressions non évaluées. En appliquant de proche en proche un ensemble de règles (liées uniquement au langage de programmation), on finit par associer à chaque sous-expression du programme un type.

Benjamin Pierce résume cette approche dans cette définition très générale :

Définition 1 (Système de types) *Un système de types est une méthode syntaxique tractable qui vise à prouver l'absence de certains comportements de programmes en classifiant leurs phrases selon le genre de valeurs qu'elles produisent. [Pie02]*

A première vue, cela semble moins puissant que le typage dynamique : en effet, il existe des programmes qui s'exécuteront sans erreur de type mais sur lesquels le typage statique ne peut s'appliquer. Dans la figure 3.2, on peut voir par une simple analyse de cas que si on fournit un booléen à `f`, elle retourne un entier. Mais selon la valeur de `b`, la variable `x` contiendra une valeur de type entier ou fonction.

Même si cet exemple est construit artificiellement, il illustre le problème suivant : les types statiques demandent un certain effort et au programmeur et au compilateur. Mais Dans le cas où le typage statique est possible, les garanties sont importantes : les valeurs portées par une variable auront toujours le même type. Par voie de conséquence, la vérification dynamique de types réussira toujours, et on peut la supprimer. Il est également possible de supprimer toutes les étiquettes de typage : on parle de *type erasure*. Une conséquence heureuse de cette suppression est que l'exécution de ce programme se fera de manière plus rapide.

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

FIGURE 3.3 – Transtypage en Java

Connaître les types à la compilation permet aussi de réaliser plus d'optimisations. Par exemple, en Python, considérons l'expression $y = x - x$. Sans information sur le type de x , aucune simplification n'est possible : l'implémentation de la différence sur ce type est une fonction quelconque, sans propriétés particulières *a priori*. Si au contraire, on sait que x est un entier, on peut en déduire que $y = 0$, sans réaliser la soustraction (si c'était la seule utilisation de x , le calcul de x aurait alors pu être éliminé).

3.2.2 Fort, faible, sound

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 transtypage 3.3. 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  
    at Cast.main(Cast.java:5)
```

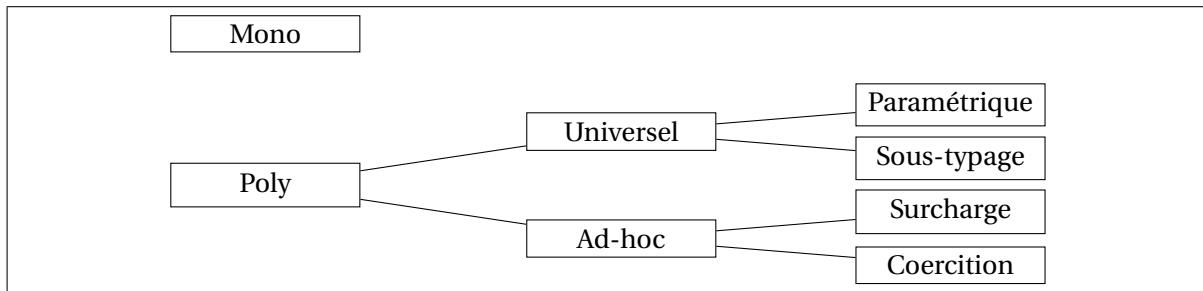


FIGURE 3.4 – Les différents types de polymorphisme.



3.2.3 Polymorphisme

Dans le cas du typage statique, restreindre une opération à un seul type de données peut être assez restrictif.

Par exemple, quel doit être le type d'une fonction qui trie un tableau ?

Monomorphisme

Une première solution peut être de forcer des types concrets, c'est à dire qu'une même fonction ne pourra s'appliquer qu'à un seul type de données.

Il est confortable pour le programmeur de n'avoir à écrire un algorithme qu'une seule fois, indépendamment du type des éléments considérés.

Il existe deux grandes classes de systèmes de types introduisant du polymorphisme.

Polymorphisme universel

Le polymorphisme est dit universel si toute fonction générique peut s'appliquer à n'importe quel type.

Polymorphisme ad-hoc

Le polymorphisme est *ad-hoc* si les fonctions génériques ne peuvent s'appliquer qu'à un ensemble de types prédéfini.

Polymorphisme paramétrique

[Mil78]

Historique + citer le papier de Milner sur le polymorphisme

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

FIGURE 3.5 – Fonction de concaténation de listes en OCaml.

```
show :: Show a => a -> String
read :: Read a => String -> a

showRead :: String -> String
showRead x = show (read x)
```

FIGURE 3.6 – Cas d'ambiguïté avec de la surcharge ad-hoc.

[Ker81]

La fonction de la figure 3.5 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.

Moralement, cette fonction est donc indépendante du type de données contenu dans la liste : elle pourra agir sur des listes de n'importe quel type d'élément.

Plutôt qu'un type, on peut lui donner le *schéma de types* suivant :

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

C'est à dire que append peut être utilisé avec n'importe quel type concret a en substituant les variables quantifiées (on parle d' *instanciation*).

Polymorphisme par sous-typage

héritage, sous-
typage, classe, méthode,
multiple, late bin-
ding, Liskov

Certains langages définissent la notion de sous-typage. C'est une relation d'ordre partiel sur les types, qui modélise la relation "est un". Chaque sous-classe peut redéfinir le comportement de chaque méthode de ses superclasses.

Polymorphisme par surcharge

Considérons l'opération d'addition : +. On peut considérer que certains types l'implémentent, et pas d'autres : ajouter deux flottants ou deux entiers a du sens, mais pas ajouter deux pointeurs.

On dira que + est *surchargé*. À chaque site d'appel, il faudra *résoudre la surcharge* pour déterminer quelle fonction appeler.

introduire l'infé-
rence plus haut

Cela rend l'inférence de types impossible dans le cas général, puisque certaines constructions sont ambiguës.

Dans le code Haskell de la figure 3.6, show peut s'appliquer à toutes les valeurs de types "affichables" et renvoie une représentation textuelle. read réalise le contraire avec les types "lisibles".

Lorsqu'on compose ces deux fonctions, le type de la valeur intermédiaire est capital puisqu'il détermine les instances de show et read à utiliser.

Polymorphisme par coercion**Polymorphisme d'ordre supérieur**

`g f = (f true, f 2)`

$$g : (\forall a. a \rightarrow a) \rightarrow (bool * int)$$

Pas inférable (annotations nécessaires).

3.2.4 Expressivité, garanties, types dépendants**3.3 Exemples****3.3.1 Faible dynamique : Perl****3.3.2 Faible statique : C****3.3.3 Fort dynamique : Python****3.3.4 Fort statique : OCaml****3.3.5 Fort statique à effets typés : Haskell****3.3.6 Theorem prover : Coq**

ÉTAT DE L'ART

Analyse légère : Sparse [5]

Coccinelle [BDH⁺09] [PTS⁺11]

lire coccinelle09

Interprétation Abstraite : Cousot [CC77, CC92], widening [Gra92], CGS [VB04], Astrée : presentation[Mau04, CCF⁺05], scaling [CCF⁺09]

Typage fort [Pie02]

Politiques de sécurité par les types : [KcS07, LZ06]

Qualificateurs de type : CQual [FFA99, STFW01, FTA02, JW04, FJKA06]

Divers : Contracts [DRS00], Taint sequences [CMP10], Dynamic taint analysis [SAB10], perl

Frama-C ?

CCurred ?

REx : Coverity [BBC⁺10]

survey : Language-based Information-Flow Security [SM03]

Deuxième partie

Typage statique de langages impératifs

UN PREMIER SYSTÈME DE TYPES

Dans ce chapitre nous présentons un langage impératif permettant de modéliser C. Nous donnons sa syntaxe, une sémantique opérationnelle ainsi qu'un système de types permettant d'obtenir plus de garanties que le système de type original de C tel que décrit dans [ISO99].

Il permet le polymorphisme sur les types pointeurs, permettant par exemple de typer :

$$\vdash \text{memcpy} : \forall a.(a^*, a^*, \text{size_t}) \rightarrow \text{VOID}$$

La traduction depuis C sera explicitée dans le chapitre 8.

5.1 Syntaxe

La grammaire suivante définit un langage impératif. On suppose qu'on peut compiler un programme écrit en C vers ce langage.

Un programme est un triplet $P = (\vec{f}, \vec{x}, b)$ constitué d'un ensemble de fonctions, d'un ensemble de variables et d'un bloc d'instructions. Ce bloc sera exécuté au lancement du programme ; il peut par exemple contenir le code d'initialisation des variables globales et l'appel à la fonction principale.

Programmes	$P ::= (\vec{f}, \vec{v}, b)$	Fonctions, globales, initialiseur
Expressions	$e ::= lv$	Left-value
	$op\ e$	Opération unaire
	$e\ op\ e$	Opération binaire
	cst	Constante
	$\&lv$	Pointeur sur donnée
	$\&f$	Pointeur sur fonction
Left-values	$lv ::= var$	Variable
	$lv.champ$	Accès à un champ
	$lv[e]$	Accès à un tableau
	$*e$	Déréférencement

Blocs	$b ::= i; b$ ϵ	Séquence Bloc vide
Instructions	$i ::= lv \leftarrow e$	Affectation
	$lv \leftarrow funexp(args)$	Appel de fonction
	$funexp(args)$	Appel de procédure
	$DECL\ nom\{b\}$	Déclaration
	$IF(e)\{b\}ELSE\{b\}$	Alternative
	$DO\{b\}WITH\ label :$	Nommage de bloc
	$GOTO\ label$	Saut en avant
	$WHILE(1)\{b\}$	Boucle infinie
	$RETURN\ e$	Retour de fonction

fonctions

expliquer pourquoi
un while expr ne
suffit pas

La différence principale avec C est que les expressions sont sans effets de bords, donc que leur évaluation peut être faite sans modifier l'environnement.

5.2 Sémantique

Dans cette section, nous définissons une sémantique pour ce langage impératif; elle pourra servir à l'implantation d'un interpréteur et à raisonner de manière formelle sur les programme. Mathématiquement, cela consiste en la définition d'une relation de transition \rightarrow entre états de l'interpréteur.

Un état est constitué d'une part d'un point de contrôle dans le programme (section 5.2.1, et d'autre part de l'état σ de la mémoire (section 5.2.2).

5.2.1 Graphe de flot de contrôle

Dans la syntaxe ci-dessus, on peut classer les instructions en deux familles : celles qui définissent le flot de contrôle ($IF(\cdot)\{ \cdot \} ELSE\{ \cdot \}$, $DO\{ \cdot \}WITH\ ;$, $GOTO\ ;$, $WHILE(1)\{ \cdot \}$) et celles qui définissent le flot de données. Une première transformation va transformer chaque fonction en son graphe de flot de contrôle, défini comme suit :

- les nœuds sont des points de contrôle, qui représentent par exemple l'adresse mémoire de l'instruction qui vient d'être exécutée.
- les arêtes sont soit des instructions "de données" (affectation, appel de fonction, déclaration), soit des conditions (ie une expression).

```

int gcd(int a, int b)
{
    if (a == 0) {
        return b;
    }
    while (b != 0) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
}

```

xxx

```

int32 gcd(int32 a, int32 b) {
    if (a == 0) {
        !return = b_int32;
        goto lbl0;
    }
    while (1) {
        if (b == 0) {
            goto lbl1;
        }
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    lbl1:
    lbl0:
    ;
}

```

Intuitivement, on peut "passer" d'un état à un autre soit en passant par une arête "condition" qui s'évalue à une valeur "vrai", soit en appliquant les effets de bord d'une arête "instruction".

Dans la suite, on suppose qu'on a à notre disposition un ensemble de jugements : $\langle l, instr, l' \rangle$ qui signifie qu'on peut passer du point l au point l' en effectuant l'instruction $instr$.

5.2.2 État mémoire

La mémoire interne de l'interpréteur est une correspondance entre l'ensemble des adresses (infini dénombrable) et l'ensemble des valeurs. Un état mémoire σ est une fonction partielle de *Addr* vers *Val*.

Ces valeurs peuvent être de plusieurs formes :

Valeurs	$v ::= n$	Entier
	f	Flottant
	NIL	Pointeur nul
	$\&a$	Pointeur sur l'adresse a
	$\&f$	Pointeur sur la fonction f
	\top	Valeur non initialisée

Pile d'appels

5.2.3 Left values

La mémoire est organisée en adresses, mais pourtant dans le programme cette notion n'est pas directement visible. Les accès sont réalisés à travers des "left values". Dans le langage C, elles correspondent aux constructions qui peuvent se retrouver à gauche du signe "=" dans une affectation.

Une left value est donc soit une variable (locale ou globale), soit l'accès à un indice d'une left value, soit l'accès à un champ d'une left value, soit le déréférencement d'une left value.

5.2.4 Jugements

La sémantique concrète que nous définissons ici est constituée de jugements logiques. Le jugement principal est une relation de transition \rightarrow entre états de l'interpréteur : il sera donc noté $(l, \sigma) \rightarrow (l', \sigma')$.

Pour aider à définir ces pas, il est nécessaire d'évaluer les expressions en fonction d'un certain état de la mémoire. On notera que dans l'état mémoire σ , l'expression e produit la valeur v en écrivant $\sigma \vdash e \Rightarrow v$.

Enfin, les expressions font intervenir des left-values. Puisque les états mémoires sont définis en termes d'adresses, il faut faire la correspondance à l'aide d'un jugement $\sigma \vdash lv \Rightarrow a$.

5.2.5 Sémantique des left-values

addr / Value

Confusion : que mettre dans σ : addr \rightarrow value ou lv \rightarrow addr ? Il faut les 2.

$$\begin{array}{c} \frac{(v, a) \in \sigma}{\sigma \vdash v \Rightarrow a} \text{ (EVAL-LV-VAR)} \quad \frac{\sigma \vdash e \Rightarrow \&a}{\sigma \vdash *e \Rightarrow a} \text{ (EVAL-LV-DEREF)} \quad \frac{\sigma \vdash lv \Rightarrow a}{\sigma \vdash lv.f \Rightarrow a + f} \text{ (EVAL-LV-FIELD)} \\ \\ \frac{\sigma \vdash lv \Rightarrow a \quad \sigma \vdash e \Rightarrow n}{\sigma \vdash lv[e] \Rightarrow a + n} \text{ (EVAL-LV-ARRAY)} \end{array}$$

5.2.6 Sémantique des expressions

Les expressions peuvent être évaluées en valeurs. Dans le cas où l'expression est une constante, c'est aussi le résultat.

$$\frac{}{\sigma \vdash c \Rightarrow c} \text{ (EVAL-CST)}$$

Si l'expression est une left-value, on établit à quelle adresse elle correspond et on récupère dans l'état mémoire à quelle valeur celle-ci correspond.

$$\frac{\sigma \vdash lv \Rightarrow a \quad (a, v) \in \sigma}{\sigma \vdash lv \Rightarrow v} \text{ (EVAL-LV)}$$

En ce qui concerne les opérations (unaires ou binaires), on commence par évaluer les opérandes. Le résultat est l'opération "concrète" sur les valeurs, notée $\widehat{\text{op}}$. Par exemple, pour la construction syntaxique $+$, on utilise l'addition sur les valeurs $\widehat{+}$ (c'est-à-dire l'addition usuelle).

$$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash \text{op } e \Rightarrow \widehat{\text{op}} v} \text{ (EVAL-UNOP)} \quad \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 \text{ op } e_2 \Rightarrow v_1 \widehat{\text{op}} v_2} \text{ (EVAL-BINOP)}$$

Enfin, les adresses sont aussi des valeurs. Le cas des pointeurs sur fonction est direct puisque toutes les fonctions sont globales ; pour le cas des pointeurs sur données on commence par déterminer l'adresse de l'objet pointé depuis l'état mémoire.

$$\frac{}{\sigma \vdash \&f \Rightarrow \&f} \text{ (EVAL-ADDRofFUN)} \quad \frac{\sigma \vdash lv \Rightarrow a}{\sigma \vdash \&lv \Rightarrow \&a} \text{ (EVAL-ADDRof)}$$

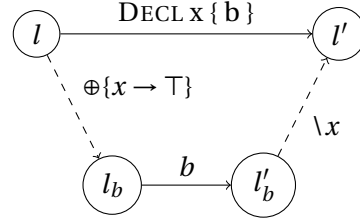
5.2.7 Sémantique des instructions

La règle la plus simple concerne l'affectation : on peut affecter une expressions à une left value si elles ont le même type.

$$\frac{\langle l, lv \leftarrow e, l' \rangle \quad \sigma \vdash lv \Rightarrow a \quad \sigma \vdash e \Rightarrow v}{(l, \sigma) \rightarrow (l', \sigma[a \mapsto v])} \text{ (INSTR-ASSIGN)}$$

Déclarer une variable, c'est rendre accessible dans un bloc une variable non initialisée, qui n'est plus accessible par la suite : Si on suppose qu'on peut traverser le bloc interne b sous un σ enrichi d'une nouvelle variable x , on peut donc traverser l'instruction $\text{DECL } x \{ b \}$.

$$\frac{\langle l, \text{DECL } x \{ b \}, l' \rangle \quad \langle l_b, b, l'_b \rangle \quad \sigma' = \sigma \oplus \{x \mapsto \top\} \quad (l_b, \sigma', s) \rightarrow (l'_b, \sigma'')}{(l, \sigma) \rightarrow (l', \sigma'' \setminus x)} \text{ (INSTR-DECL)}$$



fcall

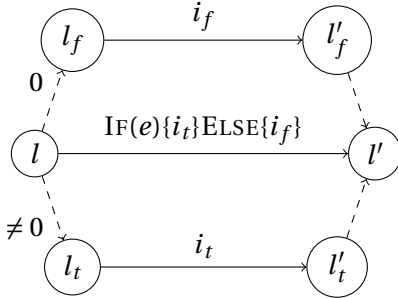
TODO pour :

$$\frac{\langle l, lv \leftarrow fe(\vec{e}), l' \rangle \quad \sigma \vdash fe \Rightarrow f \quad \sigma \vdash \vec{e} \Rightarrow \vec{v} \quad \sigma' = \sigma \oplus \{args(f) = \vec{v}\} \oplus \{!ret \mapsto \top\} \quad (Entry(f), \sigma') \rightarrow (Exit(f), \sigma'') \quad \sigma'' \vdash !ret \Rightarrow v_{ret} \quad \sigma'' \vdash lv \Rightarrow a}{(l, \sigma) \rightarrow (l', \sigma'' \setminus (args(f) \cup \{!ret\}) \oplus \{a \mapsto v_{ret}\}, ?)} \text{ (INSTR-FCALL)}$$

definir $\sigma \vdash fe \Rightarrow f$

5.2.8 Sémantique des conditions

On utilise un encodage similaire à la déclaration. Tout d'abord, on évalue la condition dans un contexte σ . Si elle s'évalue en un entier non nul, et qu'une transition à travers le bloc i_t est possible, alors on peut faire passer à travers le "IF".



$$\frac{\langle l, \text{IF}(e)\{i_t\}\text{ELSE}\{i_f\}, l' \rangle \quad \sigma \vdash e \Rightarrow n \quad n \neq 0 \quad \langle l_i, i_t, l'_i \rangle \quad (l_i, \sigma) \rightarrow (l'_i, \sigma')}{(l, \sigma) \rightarrow (l', \sigma')} \text{ (IF-TRUE)}$$

$$\frac{\langle l, \text{IF}(e)\{i_t\}\text{ELSE}\{i_f\}, l' \rangle \quad \sigma \vdash e \Rightarrow 0 \quad \langle l_i, i_f, l'_i \rangle \quad (l_i, \sigma) \rightarrow (l'_i, \sigma')}{(l, \sigma) \rightarrow (l', \sigma')} \text{ (IF-FALSE)}$$

5.3 Règles de typage

Dans cette section, on définit la notion de programme bien typé. L'analyse par typage permet de vérifier qu'à chaque expression on peut associer un type, et ce de manière cohérente entre plusieurs utilisations d'une variable.

Un jugement de typage est de la forme $\Gamma \vdash e : \tau$ et se lit "sous Γ , e est typable en τ ". Un environnement de typage Γ contient le contexte nécessaire à l'analyse, c'est à dire le type des fonctions et variables du programme.

Les instructions et blocs, au contraire, n'ont pas de type. On note $\Gamma \vdash i$ pour "sous Γ , i est bien typé", c'est à dire que ces sous expressions sont typables et que les types sont en accord avec le flot de données (par exemple, pour que l'instruction $lv \leftarrow e$ soit bien typée sous Γ , il faut que les types de lv et de e puissent avoir le même type sous Γ).

Les types des valeurs sont :

Types	$\tau ::= \text{INT, FLOAT, VOID}$	Constante
	a	Variable
	$(\tau_1, \dots, \tau_n) \rightarrow \tau_r$	Fonction
	$[\tau]$	Tableau
	$\tau *$	Pointeur
	$\{f_1 : \tau_1, \dots, f_n : \tau_n\}$	Structure

Et les environnements de typage :

Environnements	$\Gamma ::= \epsilon$	Environnement vide
	$\Gamma, v : \tau$	Ajout d'une variable

On va associer à chaque variable globale un type. Mais faire de même pourrait être trop restrictif. En effet, une fonction comme `memcpy` peut être utilisée pour copier des tableaux d'entiers, mais aussi de flottants. On va donc associer un schéma de types à chaque fonction.

Schémas

$$\sigma ::= \forall \vec{a}. \tau$$

En associant un schéma de type σ à une fonction f , on indique que la fonction pourra être utilisée avec tout type τ qui est une instantiation de σ .

Gen, inst

5.3.1 Programme

Au niveau global, un programme P est bien typé (noté $\vdash P$) s'il existe un environnement $\Gamma = (\vec{\sigma}, \vec{\tau})$ permettant de typer ses composantes (les fonctions, les globales et le bloc d'initialisation).

$$\frac{(\vec{\sigma}, \vec{\tau}) \vdash \vec{f} : \vec{\sigma} \quad (\vec{\sigma}, \vec{\tau}) \vdash \vec{x} : \vec{\tau} \quad (\vec{\sigma}, \vec{\tau}) \vdash b}{\vdash (\vec{f}, \vec{x}, b)} \text{ (PROG)}$$

5.3.2 Flot de contrôle

Les règles suivantes permettent de définir les jugements $\Gamma \vdash i$. De manière générale, les instructions sont bien typées si leurs sous-instructions sont bien typées.

$$\begin{array}{c}
\frac{}{\Gamma \vdash \epsilon} \text{ (PASS)} \quad \frac{\Gamma \vdash s \quad \Gamma \vdash b}{\Gamma \vdash s; b} \text{ (SEQ)} \quad \frac{\Gamma \vdash b}{\Gamma \vdash \text{forever}\{b\}} \text{ (FOREVER)} \quad \frac{}{\Gamma \vdash \text{GOTO } l} \text{ (GOTO)} \\
\\
\frac{\Gamma \vdash b}{\Gamma \vdash \text{DO}\{b\}\text{WITH } l:} \text{ (DOWITH)}
\end{array}$$

Dans le cas de la conditionnelle, il est en plus nécessaire de vérifier que la condition est un entier.

$$\frac{\Gamma \vdash e : \text{INT} \quad \Gamma \vdash i_t \quad \Gamma \vdash i_f}{\Gamma \vdash \text{IF}(e)\{i_t\}\text{ELSE}\{i_f\}} \text{ (IF)}$$

5.3.3 Left values

On associe à chaque left-value un type, qui est aussi le type des valeurs que celle-ci peut contenir. Le cas des variables est direct : si un couple (variable, type) est dans l'environnement de typage, la variable possède ce type.

$$\frac{(\nu, \tau) \in \Gamma}{\Gamma \vdash \nu : \tau} \text{ (LV-VAR)}$$

Si une expression a un type pointeur, en la déréréférençant on obtient une valeur du type pointé.

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

accès à un élément
d'un type compo-
site

$$\frac{\Gamma \vdash lv : \tau_s \quad (f, \tau_f) \in \tau_s}{\Gamma \vdash lv.f : \tau_f} \text{ (LV-FIELD)} \quad \frac{\Gamma \vdash lv : [\tau] \quad \Gamma \vdash e : \text{INT}}{\Gamma \vdash lv[e] : \tau} \text{ (LV-ARRAY)}$$

5.3.4 Expressions

Les constantes ont leurs types habituels. Notons que le pointeur nul (NIL) a un type polymorphe.

$$\frac{}{\Gamma \vdash n : \text{INT}} \text{ (CONST-INT)} \quad \frac{}{\Gamma \vdash f : \text{FLOAT}} \text{ (CONST-FLOAT)} \quad \frac{}{\Gamma \vdash \text{NIL} : \tau^*} \text{ (CONST-NIL)}$$

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

$$\frac{\text{op} \in \{+, -, \times, /, \&, |, \&\&, ||, \ll, \gg\} \quad \Gamma \vdash e_1 : \text{INT} \quad \Gamma \vdash e_2 : \text{INT}}{\Gamma \vdash e_1 \text{ op } e_2 : \text{INT}} \text{ (OP-INT)}$$

De même sur FLOAT.

$$\frac{\text{op} \in \{+, -, \times, /\} \quad \Gamma \vdash e_1 : \text{FLOAT} \quad \Gamma \vdash e_2 : \text{FLOAT}}{\Gamma \vdash e_1 \text{op} e_2 : \text{FLOAT}} \text{ (OP-FLOAT)}$$

Les opérateurs de comparaison peuvent s'appliquer à deux opérandes de types INT ou FLOAT et renvoient un INT.

$$\frac{\text{op} \in \{=, \neq, \leq, \geq, <, >\} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\text{INT}, \text{FLOAT}\}}{\Gamma \vdash e_1 \text{op} e_2 : \text{INT}} \text{ (OP-CMP)}$$

L'opérateur unaire – peut être appliqué à un INT ou à un FLOAT.

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

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

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

On peut prendre l'adresse d'une left-value, et obtenir un pointeur vers celle-ci. Le type du résultat est un type pointeur vers le type de base.

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

La règle concernant les pointeurs sur fonction est similaire, à ceci près que les fonctions ont des schémas de type. Un pointeur sur fonction peut donc avoir n'importe quelle instanciation de ce schéma.

$$\frac{\Gamma \vdash f : \sigma \quad \tau \leq \sigma}{\Gamma \vdash \&lv : \tau} \text{ (ADDR OF FUN)}$$

5.3.5 Fonctions

Pour typer une fonction, on commence par ajouter ses paramètres dans l'environnement de typage, et on type la définition de la fonction. Le type résultant est généralisé.

$$\frac{\Gamma' = \Gamma \oplus \{args(f) = \vec{\tau}\} \quad \Gamma' \vdash body(f) \quad \Gamma' \vdash !ret : \tau_r}{\Gamma \vdash f : Gen(\vec{\tau} \rightarrow \tau_r, \Gamma)} \text{ (FUN)}$$

5.3.6 Instructions

$$\begin{array}{c}
\frac{\Gamma \oplus \{x:\tau\} \vdash b}{\Gamma \vdash \text{DECL } x\{b\}} \text{ (DECL)} \qquad \frac{\Gamma \vdash lv:\tau \quad \Gamma \vdash e:\tau}{\Gamma \vdash lv \leftarrow e} \text{ (ASSIGN)} \\
\\
\frac{\Gamma \vdash lv:\tau_{ret} \quad \Gamma \vdash fe:\sigma \quad \Gamma \vdash \vec{e}:\vec{\tau} \quad (\vec{\tau} \rightarrow \tau_r) \leq \sigma}{\Gamma \vdash lv \leftarrow fe(\vec{e})} \text{ (FCALL)}
\end{array}$$

5.4 Limitations**5.4.1 Programmes non typables****5.4.2 Incohérences**

SYSTÈME AVEC LE QUALIFICATEUR "USER"

Dans le chapitre 5, nous avons vu comment ajouter un système de types forts statiques à un langage impératif. Ici, nous étendons ce système afin de lui ajouter des *qualificateurs de type* qui décrivent l'origine des données. Ils permettent de restreindre certaines opérations sensibles à des expressions dont la valeur est sûre.

6.1 Éditions et ajouts

6.2 Passage sur le cas d'étude

6.3 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.

SYSTÈME AVEC LE QUALIFICATEUR "SZ"

Dans ce chapitre, nous présentons une autre extension au système de types du chapitre 5, similaire à celle du chapitre 6. 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.

7.1 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 "szUser". C'est pourquoi nous appellerons ce qualificateur de type `sz`.

7.2 Approche

Cette propriété est un peu différente de la séparation entre espace utilisateur et espace noyau modélisée dans le chapitre 6 : 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 variables 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].

7.3 Annotation de `string.h`

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

7.3.1 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);
```

7.3.2 Fonctions de concaténation

strcat

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

strncat

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

7.3.3 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);
```

7.3.4 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);
```

7.3.5 Fonctions diverses

memset

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

strerror

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

strlen

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

7.4 Typage des primitives

7.5 Extensions au système de types

7.6 Résultats

Troisième partie

Expérimentation

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 5. 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.

8.1 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 fasse l’intermédiaire entre *front-end* et *back-end* (figure 8.1). 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.

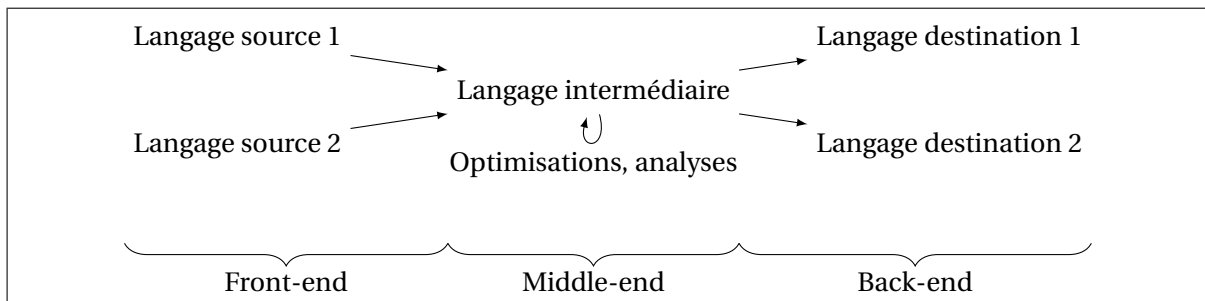


FIGURE 8.1 – Décomposition d’un compilateur : front-ends, middle-end, back-ends

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 [♣¹] utilise pour sa génération de code une représentation interne nommée Cmm, disponible dans les sources du compilateur sous le chemin `asmcomp/cmm.mli` (il s'agit donc d'une structure de données OCaml). Ce langage a l'avantage d'être très restreint, mais malheureusement il n'existe pas directement de traducteur permettant de compiler C vers Cmm.

C- [PJNO97] [♣⁷], dont le nom est inspiré du précédent, est un projet qui visait à unifier les langages intermédiaires utilisés par les compilateurs. L'idée est que si un front-end peut émettre du C- (sous forme de texte), il est possible d'obtenir du code machine efficace. Le compilateur Haskell GHC utilise une représentation intermédiaire très similaire à C-.

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

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

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 8.2.

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].

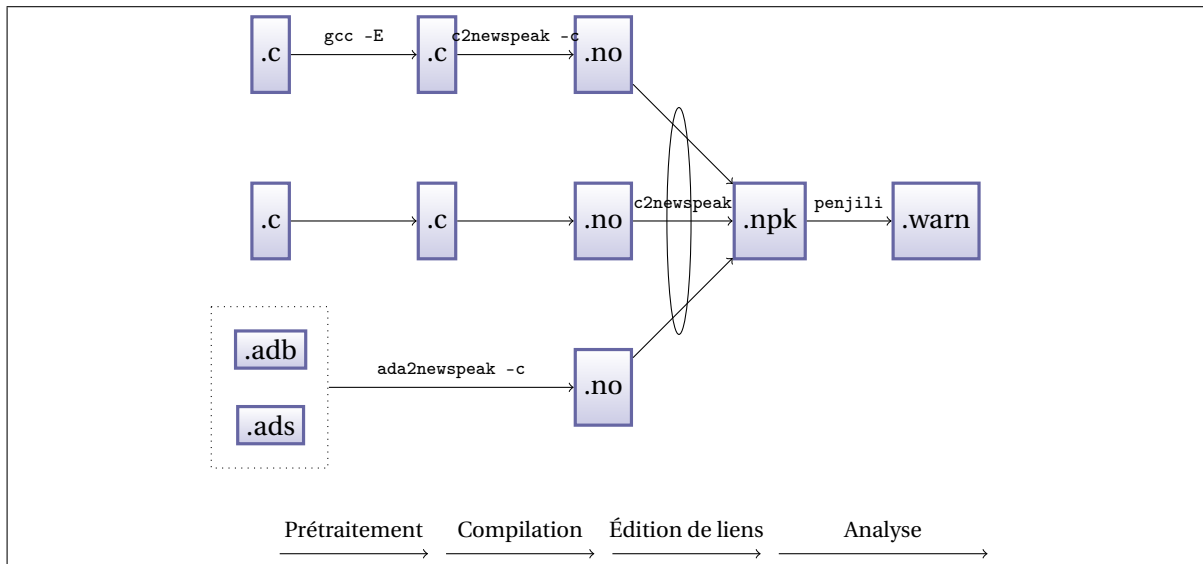


FIGURE 8.2 – Compilation depuis Newspeak

8.2 Newspeak



8.3 Chaîne de compilation

La compilation vers C est faite en trois étapes (figure 8.2) : prétraitement du code source, compilation de C prétraité vers NEWSPEAK, puis compilation de NEWSPEAK vers ce langage.

Mettre à jour la figure

8.3.1 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.

8.3.2 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.

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].

8.3.3 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;
}
```



8.3.4 Implantation de l'algorithme de typage

Commençons par étudier le cas du lambda-calcul simplement typé (figure 8.3).

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

FIGURE 8.3 – Lambda calcul simplement typé avec entiers, flottants et couples

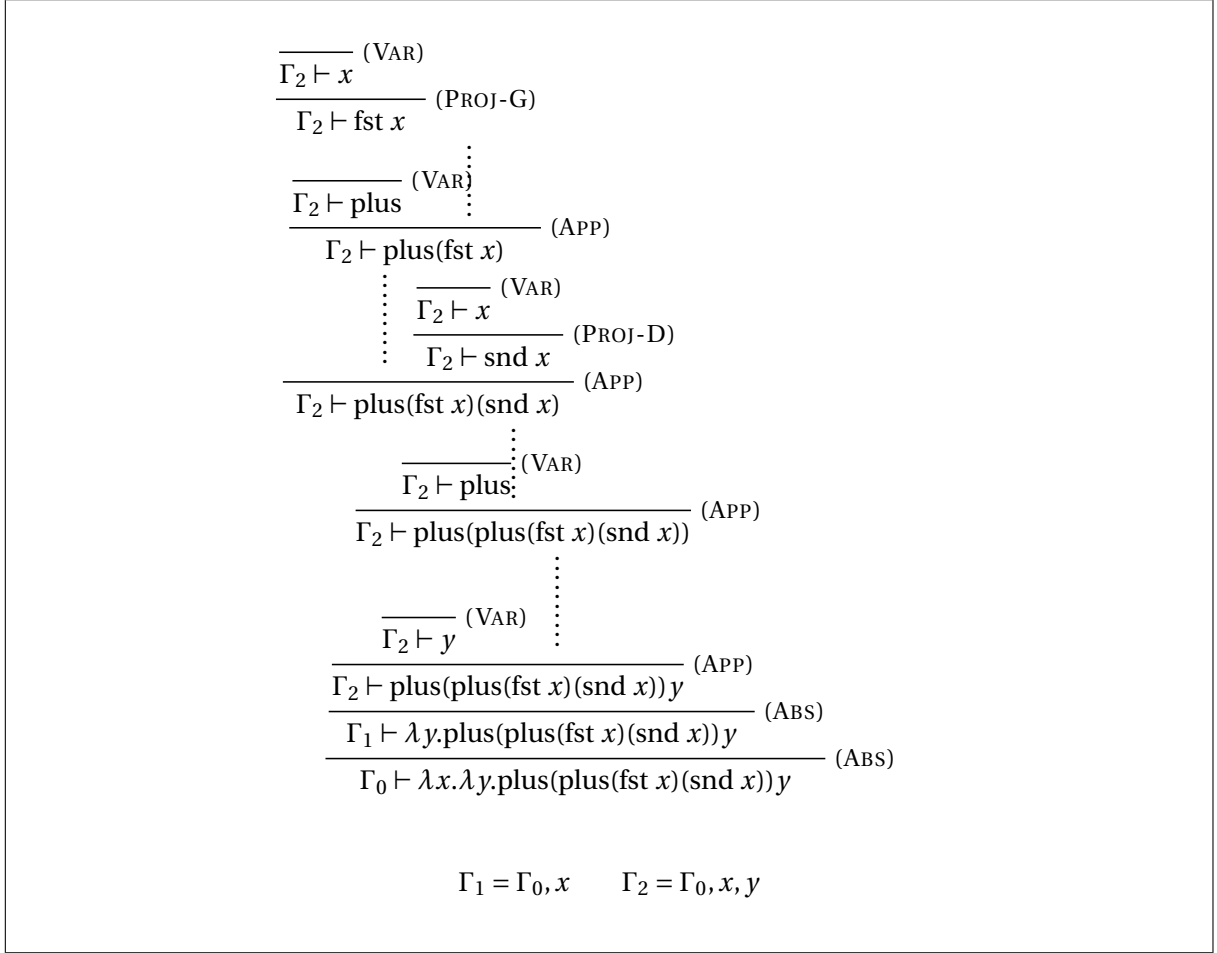


FIGURE 8.4 – Arbre d'inférence : règles à utiliser

Prenons l'exemple de la fonction suivante¹ :

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

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 8.3 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 8.4)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.

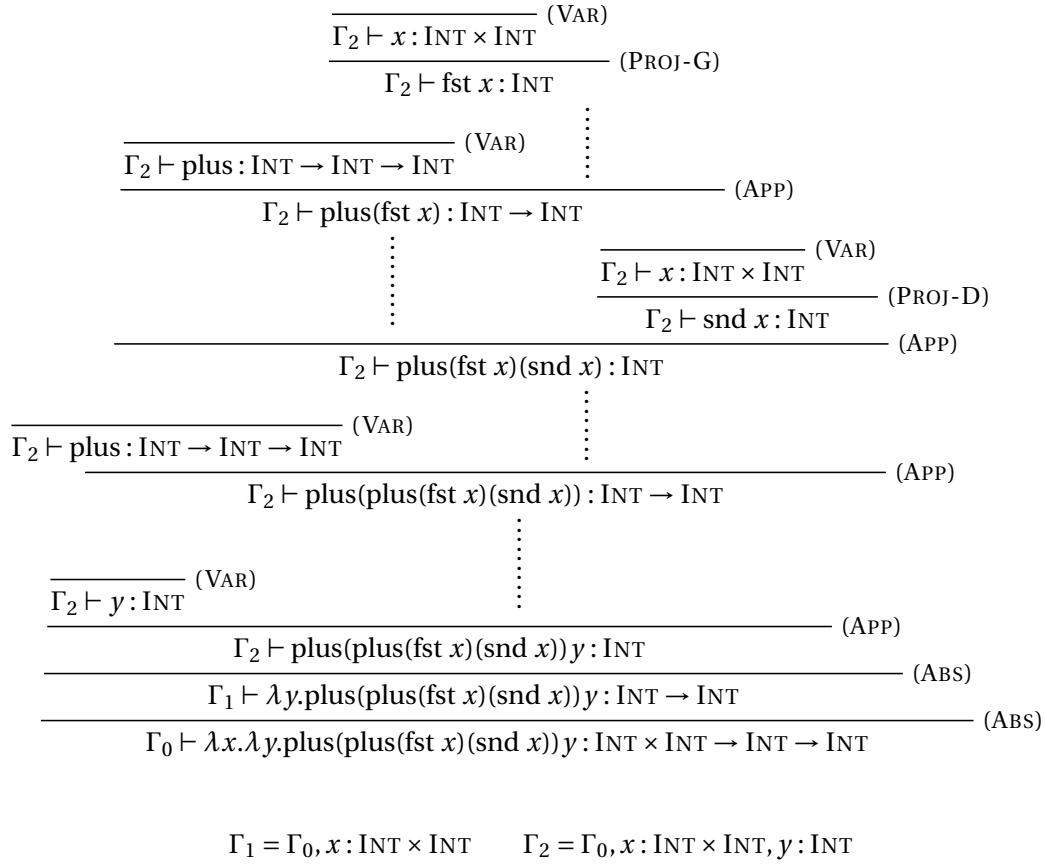


FIGURE 8.5 – Arbre d'inférence complet

Une fois à cette étape, on peut donner un nom à chaque type inconnu : τ_1, τ_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 $\tau_3 = \tau_1 \rightarrow \tau_2$.

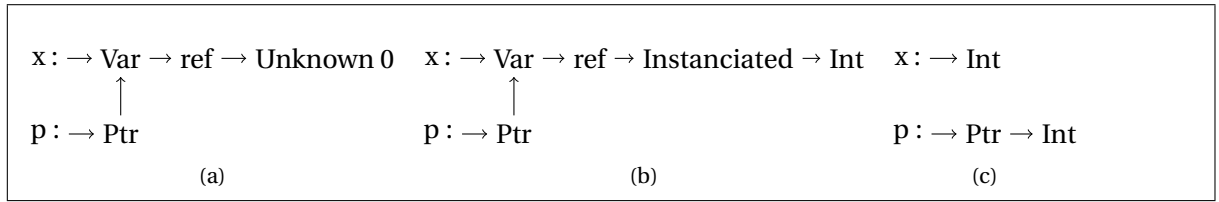


FIGURE 8.6 – Unification par partage

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 τ_j dans les autres contraintes peuvent être remplacées par τ_i .
- $\tau_i = x$ (ou $x = \tau_i$) où x est un type concret : idem.

occurs check peut être ?

C'est faux

Une fois toutes les substitutions effectuées, on obtient un arbre de typage correct (figure 8.5), donc un programme totalement inféré.

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

L'implémentation de cet algorithme utilise le partage et les références (figure 8.6).

D'abord 8.6a, ensuite 8.6b, et enfin 8.6c.

```

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

```



Le programme C suivant :

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

est compilé ainsi en Tyspeak :

```
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))
        )
    ]
  )
]
)
```


ÉTUDE DE CAS : UN DRIVER GRAPHIQUE

9.1 Description du problème



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 plante 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 9.1.

cf annexe A

FIGURE 9.1 – 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.

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));
```

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.

9.2 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 9.1 : 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`).

9.3 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 9.1 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 <http://penjili.org>, fera l'objet d'une démonstration.

9.4 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 [5].

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.

CONCLUSION

10.1 Limitations

10.2 Perspectives



CODE DU 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(crtc);
        value = radeon_crtc->crtc_id;
        found = 1;
        break;
    }
}
if (!found) {
    DRM_DEBUG_KMS("unknown crtc id %d\n", value);
    return -EINVAL;
}
break;
case RADEON_INFO_ACCEL_WORKING2:
    value = rdev->accel_working;
    break;
case RADEON_INFO_TILING_CONFIG:
    if (rdev->family >= CHIP_CEDAR)
        value = rdev->config.evergreen.tile_config;
    else if (rdev->family >= CHIP_RV770)
        value = rdev->config.rv770.tile_config;
    else if (rdev->family >= CHIP_R600)
        value = rdev->config.r600.tile_config;
    else {
        DRM_DEBUG_KMS("tiling config is r6xx+ only!\n");
        return -EINVAL;
    }
case RADEON_INFO_WANT_HYPERZ:
    mutex_lock(&dev->struct_mutex);
    if (rdev->hyperz_filp)
        value = 0;
    else {
        rdev->hyperz_filp = filp;
        value = 1;
    }
    mutex_unlock(&dev->struct_mutex);
    break;
default:
    DRM_DEBUG_KMS("Invalid request %d\n", info->request);
    return -EINVAL;
}
if (DRM_COPY_TO_USER(value_ptr, &value, sizeof(uint32_t))) {
    DRM_ERROR("copy_to_user\n");
    return -EFAULT;
}
return 0;

```



```

}

/* from 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);
}

```


TABLE DES FIGURES

2.1	Cadres de pile	8
2.2	Les différents <i>rings</i>	9
2.3	Implantation de la mémoire virtuelle	10
2.4	Mécanisme de mémoire virtuelle.	10
2.5	Espace d'adressage d'un processus	11
2.6	Appel de <code>gettimeofday</code>	13
3.1	Session Python présentant le typage dynamique	16
3.2	Fonction Python non typable statiquement.	17
3.3	Transtypage en Java	18
3.4	Les différents types de polymorphisme.	19
3.5	Fonction de concaténation de listes en OCaml.	20
3.6	Cas d'ambigüité avec de la surcharge ad-hoc.	20
8.1	Décomposition d'un compilateur : front-ends, middle-end, back-ends	47
8.2	Compilation depuis Newspeak	49
8.3	Lambda calcul simplement typé avec entiers, flottants et couples	51
8.4	Arbre d'inférence : règles à utiliser	52
8.5	Arbre d'inférence complet	53
8.6	Unification par partage	54
9.1	Bug freedesktop.org #29340	57

RÉFÉRENCES WEB

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

BIBLIOGRAPHIE

- [AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor (extended version). Research report 6138, INRIA, 2007. 29 pages. 48
- [BBC⁺10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later : using static analysis to find bugs in the real world. *Commun. ACM*, 53(2) :66–75, February 2010. 23
- [BDH⁺09] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, January 2009. 23
- [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. 48
- [BP05] Daniel P. Bovet and Marco Cesati Ph. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, third edition edition, November 2005. 10
- [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. 23
- [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.>). 23
- [CCF⁺05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005. 23
- [CCF⁺09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3) :229–264, 2009. 23
- [CMP03] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly, 2003. 15
- [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. 23
- [DRS00] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV : Towards a realistic tool for statically detecting all buffer overflows in C, 2000. 23

- [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. 50
- [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. 23, 59
- [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. 23
- [FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02 : Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, volume 37, pages 1–12, New York, NY, USA, May 2002. ACM Press. 23
- [Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. 10
- [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. 23
- [Har88] Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4) :36–38, October 1988. 57
- [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. 48, 50, 58
- [Int] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 6, 11
- [ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999. 15, 27, 39, 47
- [JW04] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004. 23
- [KcS07] Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci.*, 174(7) :79–104, 2007. 23
- [Ker81] Brian W. Kernighan. Why Pascal is not my favorite programming language. Technical report, AT&T Bell Laboratories, April 1981. 20
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988. 15, 47
- [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. 48
- [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. 23

- [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. 23
- [Mer03] J. Merrill. GENERIC and GIMPLE : a new tree representation for entire functions. In *GCC developers summit 2003*, pages 171–180, 2003. 48
- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, December 1978. 19
- [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. 48
- [OGS08] Bryan O’Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O’Reilly Media, Inc., 1st edition, 2008. 15
- [One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 1996. 6, 40
- [Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 17, 23
- [PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003. 15
- [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. 48
- [PTS⁺11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux : Ten years later. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, Newport Beach, CA, USA, March 2011. 23
- [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. 23
- [SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21 :2003, 2003. 23
- [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. 48
- [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. 23
- [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. 40

- [Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007. 5
- [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. 23
- [Wal00] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2000. 15