Analyse statique de logiciel système par typage statique fort

Application au noyau Linux

Étienne Millon

12 novembre 2012

ABSTRACT

Abstract ici.

ACKNOWLEDGEMENTS

Remerciements

Cassedédi

TABLE DES MATIÈRES

CHAPITRE 1

INTRODUCTION

Première partie

Méthodes formelles pour la sécurité

CHAPITRE 2

SYSTÈMES D'EXPLOITATION

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

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

2.1 Rôle d'un système d'exploitation

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

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

Néanmoins c'est un peu restrictif, puisque cela ne permet qu'à une personne d'interagir avec le système. Dans la seconde moitié des années 60, sont apparus les premiers systèmes "à temps partagé", permettant à plusieurs utilisateurs de travailler en même temps.

Permettre l'exécution de plusieurs programmes en même temps est une idée révolutionnaire, mais elle n'est pas sans difficultés techniques : en effet les ressources de la machine doivent être aussi partagées entre les utilisateurs et les programmes. Par exemple, plusieurs programmes vont utiliser le processeur les uns à la suite des

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

Si deux programmes (ou plus) s'exécutent de manière concurrente sur le même matériel, il faut s'assurer que l'un ne puisse pas écrire dans la mémoire de l'autre, ou que les deux utilisent la carte réseau les uns à la suite des autres. Ce sont des rôles du système d'exploitation.

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

Par exemple, si un programme veut copier des données depuis un cédérom vers la mémoire principale, il devra interroger le bus SATA, interroger le lecteur sur la présence d'un disque dans le lecteur, activer le moteur, calculer le numéro de trame des données sur le disque, demander la lecture, puis déclencher une copie de la mémoire.

Si dans un autre cas il désire récupérer des données depuis une mémorette 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 :

À affiner ou supprimer

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

2.2 Architecture Intel

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

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

La référence pour la description de l'assembleur Intel est la documentation du constructeur [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.

La mémoire principale contient divers types des données :

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

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

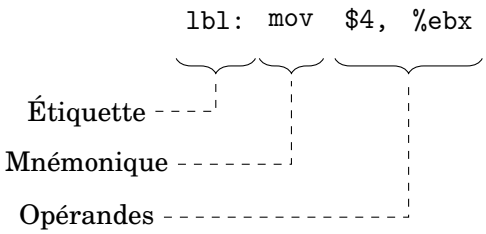
L'état du processeur est défini par la valeur de ses registres, qui sont des petites zones de mémoire interne (quelques bits chacun). Par exemple, la valeur du pointeur de pile est stockée dans ESP. Le registre EBP, couplé à ESP sert à adresser les variables locales et paramètres d'une fonction, comme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

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

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

Les calculs sont décrits sous forme d'une suite d'instructions. Chaque instruction est composée d'un mnémonique et d'une liste d'opérandes. Les mnémoniques (`mov`,

`call`, `sub`, etc) définissent un type d'opération à appliquer sur les opérandes. L'instruction peut aussi être précédée d'une étiquette, qui correspondra à son adresse.



Ces opérandes peuvent être de plusieurs types :

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

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

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

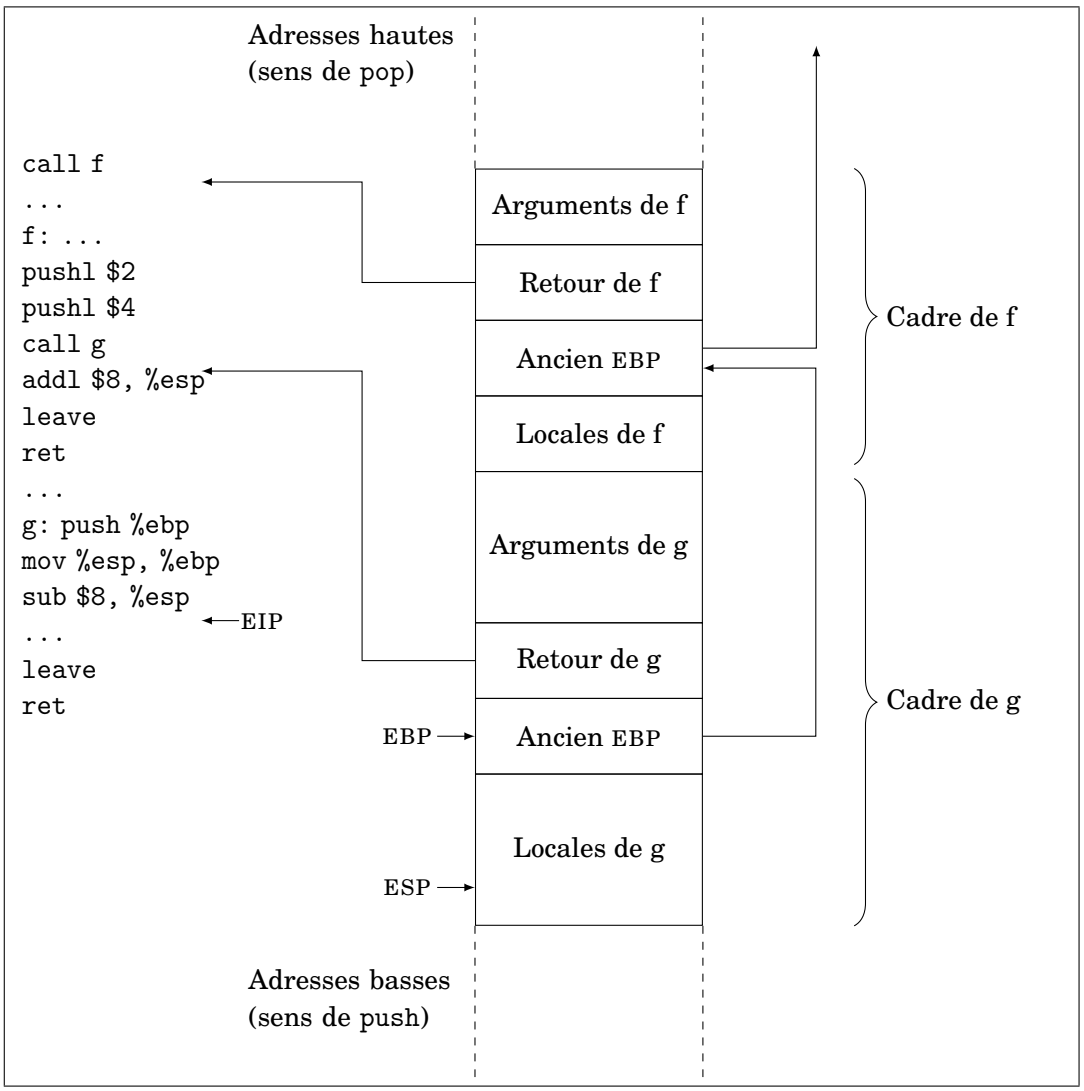


FIGURE 2.1 – Cadres de pile.

2.2.2 Fonctions et conventions d'appel

Mettre des vrais nombres plutôt que du symbolique

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

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

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

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

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

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

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

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

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

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

2.2.3 Tâches, niveaux de privilèges

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

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

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

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

Il y a 4 niveaux de privilèges (nommés aussi *rings* - figure 2.2) : le *ring* 0 est le plus privilégié, le *ring* 3 le moins privilégié. Dans l'exemple précédent, on pourrait isoler l'ordonnanceur de processus en le faisant s'exécuter en *ring* 0 alors que les autres tâches seraient en *ring* 3.

2.2.4 Mémoire virtuelle

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

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

Ce mécanisme est contrôlé par valeur du registre CR3 : les 10 premiers bits d'une adresse virtuelle sont un index dans le répertoire de pages qui commence à l'adresse contenue dans CR3. À cet index, se trouve l'adresse d'une table de pages. Les 10 bits suivants de l'adresse sont un index dans cette page, donnant l'adresse d'une page de 4 kibioctets (figure 2.3).

Faire cette figure

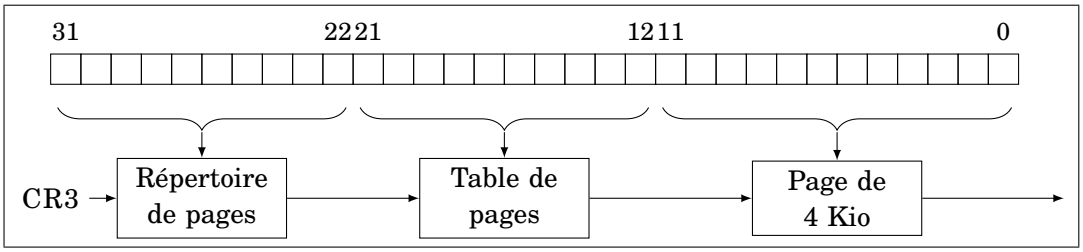


FIGURE 2.3 – Implantation de la mémoire virtuelle

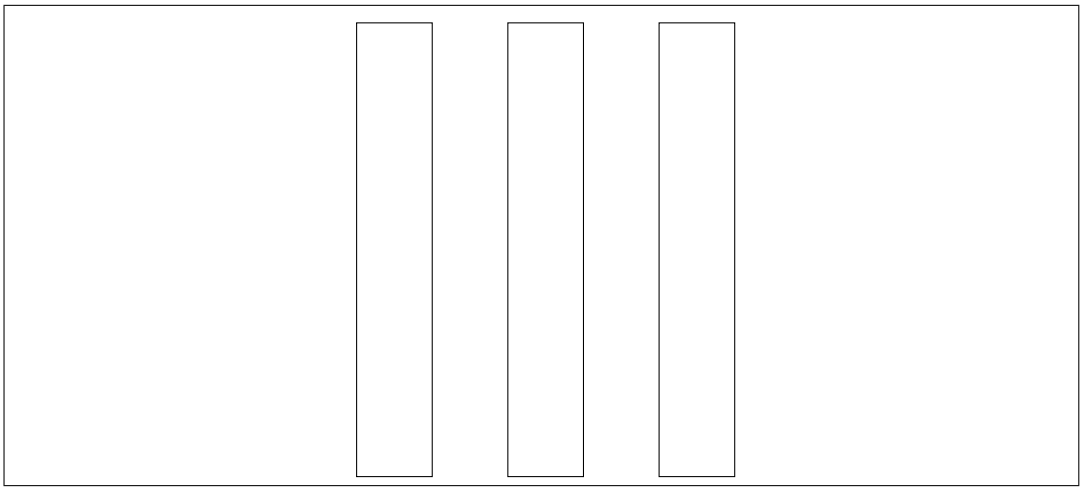


FIGURE 2.4 – Mécanisme de mémoire virtuelle.

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).

Redite qui n'apporte pas plus d'explication

2.3 Cas de Linux

Dans cette section, nous allons voir comment ces mécanismes sont implantés dans le noyau Linux. Une description plus détaillée pourra être trouvée dans [BC05], ou pour le cas de la mémoire virtuelle, [Gor04].

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

Une notion de tâche similaire à celle décrite dans la section 2.2.3 existe : elles s'exécutent l'une après l'autre, le changement s'effectuant sur interruptions.

Pour faire appel aux services du noyau, le code utilisateur doit faire appel à des appels systèmes, qui sont des fonctions exécutées par le noyau. Chaque tâche doit

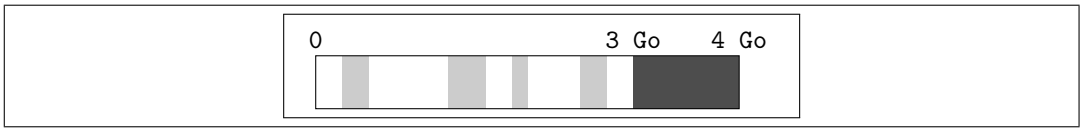


FIGURE 2.5 – L'espace d'adressage d'un processus. En gris clair, les zones accessibles à tous les niveaux de privilèges : code du programme, bibliothèques, tas, pile. En gris foncé, la mémoire du noyau, réservée au mode privilégié.

donc avoir deux piles : une pile "utilisateur" qui sert pour l'application elle-même, et une pile "noyau" qui sert aux appels système.

Grâce à la mémoire virtuelle, chaque processus possède sa propre vue de la mémoire dans son espace d'adressage (figure 2.5), et donc chacun un ensemble de tables de pages et une valeur de CR3associée. Au moment de changer le processus en cours, l'ordonnanceur charge donc le CR3du nouveau processus.

Les adresses basses (inférieures à `PAGE_OFFSET` = 3 Gio = `0xc0000000`) sont réservées à l'utilisateur. On y trouvera par exemple :

- le code du programme
- les données du programmes (variables globales)
- la pile utilisateur
- le tas (mémoire allouée par `malloc` et fonctions similaires)
- les bibliothèques partagées

Au dessus de `PAGE_OFFSET`, se trouve la mémoire réservée au noyau. Cette zone contient le code du noyau, les piles noyau des processus, etc.

2.3.1 Appels système

clarifier encore tout ça

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

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

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

---

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*.

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.

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);
```

```
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

Ici, pas de difficulté particulière : la communication entre le *ring* 0 et le *ring* 3 est faite uniquement à travers les registres, comme décrit dans la section 2.3.1.

Mais la plupart des appels systèmes communiquent de l'information de manière indirecte, à travers un pointeur. L'appellant alloue une zone mémoire dans son espace d'adressage et passe un pointeur à l'appel système. Ce mécanisme est utilisé par exemple par la fonction `gettimeofday` (figure 2.6).

Considérons une implémentation naïve de cet appel système qui écrirait directement à l'adresse pointée. La figure 2.7a présente ce qui se passe lorsque le pointeur fourni est dans l'espace d'adressage du processus : c'est le cas d'utilisation normal et l'écriture est donc possible.

Si l'utilisateur passe un pointeur dont la valeur est supérieure à `0xc0000000` (figure 2.7b), que se passe-t'il ? Comme le déréférencement est fait dans le code du noyau, il est également fait en *ring* 0, et va pouvoir être réalisé sans erreur : l'écriture se fait et potentiellement une structure importante du noyau est écrasée.

Un utilisateur malicieux peut donc utiliser cet appel système pour écrire à n'importe quelle adresse dans l'espace d'adressage du noyau. Ce problème vient du fait que l'appel système utilise les privilèges du noyau au lieu de celui qui contrôle la valeur des paramètres sensibles. Celà s'appelle le *Confused Deputy Problem*[Har88].

La bonne solution est de tester dynamiquement la valeur du pointeur : si la valeur du pointeur est supérieure à `0xc0000000`, il faut indiquer une erreur avant d'écrire (figure 2.7c. Sinon, cela ne veut pas dire que le déréférencement se fera sans erreur, mais au moins le noyau est protégé.

Dans le noyau, un ensemble de fonctions permet d'effectuer des copies sûres. La fonction `access_ok` réalise le test décrit précédemment. Les fonctions `copy_from_user` et `copy_to_user` réalisent une copie de la mémoire après avoir fait ce test. Ainsi, l'implantation correcte de l'appel système `gettimeofday` fait appel à celle-ci (figure 2.8).

Pour préserver la sécurité du noyau, il est donc nécessaire de vérifier la valeur

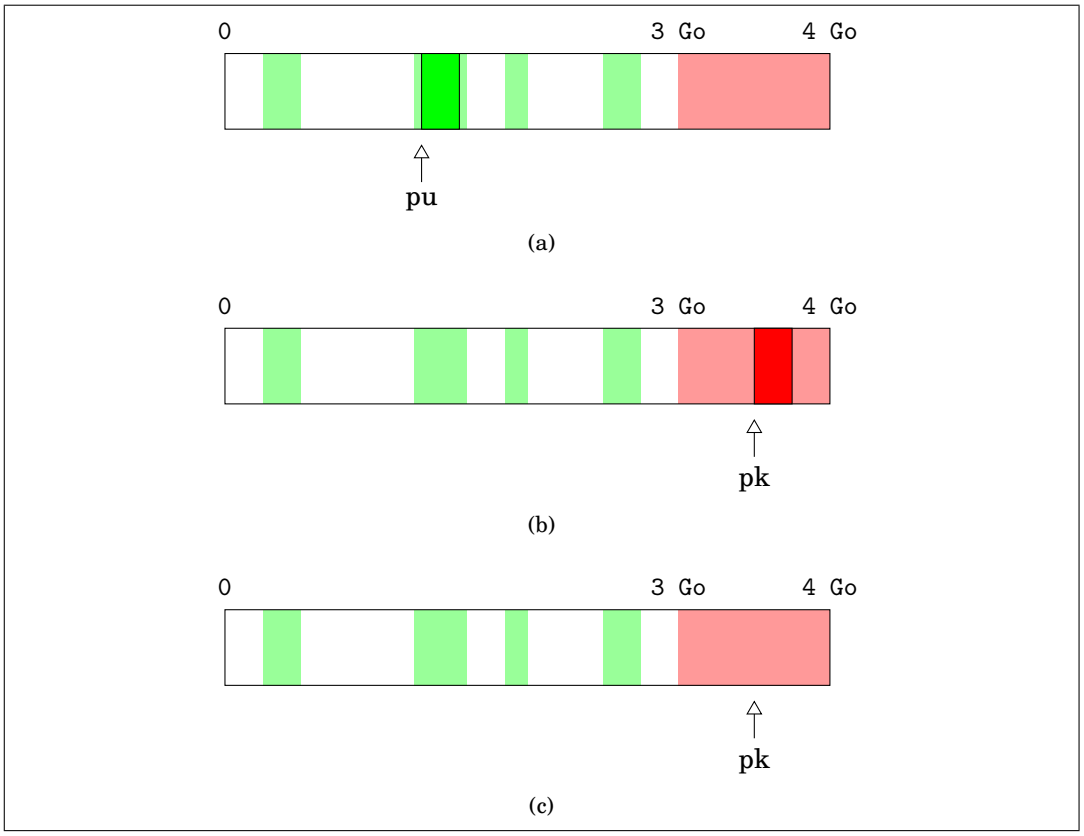


FIGURE 2.7 – Zones mémoire

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

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

FIGURE 2.8 – Implantation de l'appel système `gettimeofday`

CHAPITRE 3

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

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).

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 $(\texttt{0x00000001}, Int) + (\texttt{0x0000f000}, Int)$, 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 : $(\texttt{0x0000f001}, Int)$. Si au contraire on tente d'ajouter deux adresses $(\texttt{0x2e8d5a90}, Addr) + (\texttt{0x76a5e0ec}, Addr)$, 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 créé 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 toute les vérifications se font dans la sémantique dynamique (ie, à l'exécution). C'est à double tranchant : d'une part, cela permet d'être 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

```
>>> 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

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 3.1 (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.

```python
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.

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

FIGURE 3.3 – Transtypage en Java

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.

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 réa-

liser des tests de typage, on dit qu'il est *fort*. Mais ce n'est que rarement le cas. En effet, il peut y avoir des constructions au sein du langage qui permettent de contourner le système de types, comme un opérateur de transtypage3.3. 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)
```

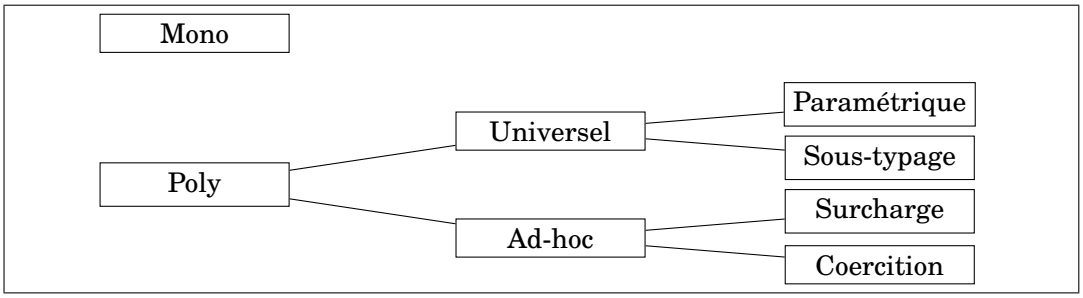


FIGURE 3.4 – Les différents types de polymorphisme.

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.

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

FIGURE 3.5 – Fonction de concaténation de listes en OCaml.

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]
[Ker81]

Historique + citer le papier de Milner sur le polymorphisme

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 :

$$\text{append} : \forall a.a\text{list} -> a\text{list} -> a\text{list}$$

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,hér multiple,late binding,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.

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

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

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

introduire l'inférence plus haut

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

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 coercition

Polymorphisme d'ordre supérieur

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

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

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

CHAPITRE 4

ÉTAT DE L'ART

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

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

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

lire coccinelle09

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

Ce système d'annotations sur les types a été formalisé sous le nom de *qualificateurs de types* : chaque type peut être décoré d'un ensemble de qualificateurs (à la manière de `const`), et des règles de typage permettent d'établir des propriétés sur le programme. Ces analyses ont été implantée dans l'outil CQual [FFA99, STFW01, FTA02, JW04, FJKA06]. .

lister les applications

L'interprétation abstraite est une technique d'analyse générique qui permet de simuler statiquement tous les comportements d'un programme Cousot [CC77, CC92]. Un exemple d'application est de calculer les bornes de variations des variables pour s'assurer qu'aucun débordement de tableau n'est possible. Cette technique est très puissante mais possède plusieurs inconvénients. D'une part, pour réaliser une analyse interprocédurale il faut partir d'un point en particulier du programme (comme

la fonction `main`). Cette hypothèse n'est pas facilement satisfaite dans un noyau de système d'exploitation, qui possède de nombreux points d'entrée. D'autre part, il est très difficile de faire passer à l'échelle un interpréteur abstrait [CCF+09, BBC+10].

L'approche par typage, plus légère, est séduisante. Pour les différents enjeux des systèmes de types statiques, on pourra se référer à [Pie02]. Il est possible d'encoder ce genre de propriétés dans un sytème de types, cf. [KcS07] et [LZ06].

Hoare

On peut aller plus loin que les simples types et utiliser un langage de contrats : chaque fonction est annotée par des pré- et post-conditions sur la mémoire[DRS00].

et Perl ?

Du côté de l'analyse dynamique, [SAB10].

Ce que nous voulons vérifier peut être vue comme une propriété de flot. Un *survey* des problèmes et techniques existantes peut être trouvé dans [SM03].

Interprétation Abstraite : widening [Gra92], CGS [VB04], Astrée : presentation[Mau04, CCF+05],

Divers : Taint sequences [CMP10],

Frama-C ?

CCurred ?

Deuxième partie

Typage statique de langages impératifs

CHAPITRE 5

SÉMANTIQUE D'UN LANGAGE IMPÉRATIF

On définit ici un langage impératif servant de base à nos analyses. Un premier système de types sera donné dans le chapitre 7.

5.1 Syntaxe

La figure 5.1 présente notre langage intermédiaire. Il contient la plupart des fonctionnalités présentes dans les langages impératifs comme C.

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

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

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

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

Les opérateurs sont donnés dans la figure 5.2.

5.2 Définitions préliminaires

On suppose avoir à notre disposition un ensemble infini dénombrable d'identificateurs $\textsc{Id}$ (par exemple des chaînes de caractères).

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

Constantes	$c ::= i$	Entier
	$\mid f$	Flottant
	$\mid$ NULL	Pointeur nul
Expressions	$e ::= c$	Constante
	$\mid lv$	Accès mémoire
	$\mid \diamond\ e$	Opération unaire
	$\mid e \diamond e$	Opération binaire
	$\mid \&lv$	Pointeur
	$\mid lv \leftarrow e$	Affectation
	$\mid \{f_1 : e_1; \ldots; f_n : e_n\}$	Structure
	$\mid \{e_1; \ldots; e_n\}$	Tableau
	$\mid f(e_1, \ldots, e_n)$	Appel de fonction
Left-values	$lv ::= x$	Variable
	$\mid *lv$	Déréférencement
	$\mid lv.f$	Accès à un champ
	$\mid lv[e]$	Accès à un élément
Instructions	$i ::=$ PASS	Instruction vide
	$\mid i; i$	Séquence
	$\mid e$	Expression
	$\mid$ IF$(e)\{i\}$ELSE$\{i\}$	Alternative
	$\mid$ WHILE$(e)\{i\}$	Boucle
	$\mid$ RETURN(e)	Retour de fonction
Fonctions	$f ::= (i,$	Corps
	$(x_1, \ldots, x_n),$	Arguments
	$((x'_1, e_1), \ldots, (x'_p, e_p)))$	Variables locales
Phrases	$p ::= f$	Déclaration de fonction
	$\mid$ struct $s\{x_1 : t_1; \ldots; x_n : t_n\}$	Déclaration de structure
	$\mid e$	Évaluation d'expression
	$\mid x = e$	Variable globale
Programme	$P ::= (p_1, \ldots, p_n)$	Phrases

FIGURE 5.1 – Syntaxe

Opérateurs	$\diamond ::= +, -, \times, /$	Arithmétique entière
	$\mid +., -., \times., /.$	Arithmétique flottante
	$\mid =, \neq, \leq, \geq, <, >$	Comparaisons
	$\mid \&, \mid, \hat{}, \sim$	Opérateurs bit à bit
	$\mid \&\&, \mid\mid, !$	Opérateurs logiques
	$\mid \ll, \gg$	Décalages

FIGURE 5.2 – Syntaxe des opeérateurs

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

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

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

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

5.3 Mémoire

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

$$
\begin{aligned}
\textsc{Val} &= \textsc{Int} \biguplus \textsc{Float} \biguplus \{\textsc{Null}\} \biguplus \Phi \\
\textsc{Int} &= \mathbb{Z}/2^{32}\mathbb{Z} - 2^{31} \\
\textsc{Float} &= \textsc{IeeeFloat}(32) \\
\Phi &= \text{fix}(\Phi') \\
\Phi'(X) &= X \\
&\quad \cup \textsc{Addr} \\
&\quad \cup \{*\varphi / \varphi \in \Phi\} \\
&\quad \cup \{\varphi.f / \varphi \in \Phi, f \in \textsc{Id}\} \\
&\quad \cup \{\varphi[n] / \varphi \in \Phi, n \in \textsc{Int}\} \\
\textsc{Addr} &= \textsc{Id} \biguplus (\mathbb{N} \times \textsc{Id})
\end{aligned}
$$

$\textsc{IeeeFloat}(n)$ correspond à l'ensemble des flottants IEEE 754 de n bits[oEE08]. Ici, $\textsc{Int}$ est choisi pour représenter les nombres entiers de -2^{31} à $2^{31}-1$, mais ce choix est arbitraire : de la même manière, on aurait pu choisir des nombres à 64 bits ou même de précision arbitraire.

L'ensemble des états mémoire est :

$$\textsc{Mem} = ((\textsc{Id} \times \textsc{Val})^*)^* \times (\textsc{Id} \times \textsc{Val})^*$$

Un état mémoire état mémoire $(s, g) \in \textsc{Mem}$ est composé :

- d'une part, d'une pile s de cadres, qui sont des listes d'association (nom de variable, valeur).
- d'autre part, une liste d'association qui représente les variables globales.

La structure de pile des locales permet de les organiser en niveaux indépendants : à chaque appel de fonction, un nouveau cadre de pile est créé, comprenant ses paramètres et ses variables locales.

Au contraire, pour les globales il n'y a pas de système d'empilement, puisque ces variables sont accessibles depuis tout point du programme.

Ces définitions sont résumées dans la figure 5.3

Définition 5.1 (Recherche de variable) *La recherche de variable permet d'associer à une variable x une adresse a.*

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

Valeurs	$v ::= c$	Constante
	$\mid \varphi$	Référence mémoire
	$\mid \{f_1 : v_1; \ldots; f_n : v_n\}$	Structure
	$\mid \{v_1; \ldots; v_n\}$	Tableau
Adresses	$a ::= (n, x)$	Variable locale
	$\mid x$	Variable globale
Chemins	$\varphi ::= a$	Adresse
	$\mid *\varphi$	Déréférencement
	$\mid \varphi.f$	Accès à un champ
	$\mid \varphi[n]$	Accès à un élément
Pile	$s ::= []$	Pile vide
	$\mid \{x_1; \ldots; x_n\} :: s$	Ajout d'un cadre
État mémoire	$m ::= (s,$	Pile
	$\{x_1; \ldots; x_n\},$	Globales
	$\{a_1 \mapsto v_1; \ldots; a_p \mapsto v_p\})$	Valeurs

FIGURE 5.3 – Interpréteur

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

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

Définition 5.2 (Manipulations de pile) *On définit l'empilement d'un cadre de pile* $f = ((x_1, v_1), \ldots, (x_n, v_n))$ *sur un état mémoire* $m = (s,g)$ *:*

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

On définit aussi l'extension du dernier cadre de pile :

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

De même on définit le dépilement :

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

5.4 Accesseurs

On définit quelques accesseurs. Un accesseur $[\cdot]$ permet d'accéder à une structure s et d'obtenir un élément e à partir d'un indice i (noté $e = s[i]$) ou de modifier le sous-élément à l'indice i par e (noté $s[i \leftarrow e]$).

Définition 5.3 (Accès à une liste d'associations) *Une liste d'association est une liste de paires (clef, valeur) avec l'invariant supplémentaire que les clefs sont uniques. Il est donc possible de trouver au plus une valeur associée à une clef donnée. L'écriture est également possible, en remplaçant un couple par un couple avec une valeur différente.*

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

$$\begin{aligned} l[x]_l &= v \text{ où } \{v\} = \{y/(x,y) \in l\} \\ l[x \leftarrow v]_l &= (x,v) :: \{(y,v) \in g(x)/y \neq x\}) \end{aligned}$$

Définition 5.4 (Accès par adresse) *Les états mémoire sont constitués des listes d'association (nom, valeur).*

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

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

Pour $m = (s, g)$*,*

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

Définition 5.5 (Accès par champ) *Les valeurs qui sont des structures possèdent des sous-valeurs, associées à des noms de champ.*

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

C'est une erreur d'accéder à un champ d'une valeur non structure ($4[f]$ *par exemple).*

$$\begin{aligned}\{f_1 : v_1; \ldots; f_n : v_n\}[f_i]_f &= v_i \\ \{f_1 : v_1; \ldots; f_n : v_n\}[f_p \leftarrow v]_f &= \{f_1 : v'_1; \ldots; f_n : v'_n\} \\ &\text{où } v'_p = v \\ &\quad \forall i \neq p, v'_i = v_i\end{aligned}$$

Définition 5.6 (Accès par indice) *On définit de même un accesseur* $[\cdot]_i$ *pour les accès par indice à des valeurs tableaux. Néanmoins le paramètre indice est toujours un entier et pas une expression arbitraire.*

$$\begin{aligned}\{v_1; \ldots; v_n\}[i]_i &= e_i \\ \{v_1; \ldots; v_n\}[i \leftarrow v]_i &= \{v'_1; \ldots; v'_n\} \\ &\text{où } v'_i = v \\ &\quad \forall j \neq i, v'_j = v_j\end{aligned}$$

Définition 5.7 (Accès par chemin) *L'accès par chemin* $[\cdot]_\varphi$ *permet de lire et de modifier la mémoire en profondeur.*

$$\begin{aligned}m[a]_\varphi &= m[a]_a \\ m[*\varphi]_\varphi &= m[\varphi']_\varphi \text{ où } \varphi' = m[\varphi]_\varphi \\ m[\varphi.f]_\varphi &= m[\varphi]_\varphi[f]_f \\ m[\varphi[i]]_\varphi &= m[\varphi]_\varphi[i]_i \\ m[a \leftarrow v]_\varphi &= m[a \leftarrow v]_a \\ m[*\varphi \leftarrow v]_\varphi &= m[\varphi' \leftarrow v]_\varphi \text{ où } \varphi' = m[\varphi]_\varphi \\ m[\varphi.f \leftarrow v]_\varphi &= m[\varphi \leftarrow (m[\varphi]_\varphi[f \leftarrow v]_f)]_\varphi \\ m[\varphi[i] \leftarrow v]_\varphi &= m[\varphi \leftarrow (m[\varphi]_\varphi[i \leftarrow v]_i)]_\varphi\end{aligned}$$

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

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

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

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

5.5 Expressions

Définition 5.8 (Évaluation d'une expression) *L'évaluation d'une expression e se fait sous un état mémoire particulier m et est susceptible de modifier celui-ci en le transformant en un nouveau m'. Le résultat est toujours une valeur v, c'est à dire que nous présentons pour les expressions une sémantique à grands pas. Cette évaluation est notée :*

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

Définition 5.9 (Évaluation d'une left-value) *L'évaluation d'une left-value lv produit un "chemin" φ dans une variable, qui est en fait équivalent à une left-value dont toutes les sous-expressions (d'indices) ont été évaluées.*

On note :

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

La sémantique présentée n'a pas d'erreur explicite. Si aucune règle ne peut s'appliquer, on considère que l'exécution est terminée.

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

Left-values

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

$$\frac{a = \text{Lookup}(x, m)}{\langle x, m\rangle \rightarrow \langle a, m\rangle}\ (\textsc{Phi-Var})$$

Les règles portant sur le déréférencement et l'accès à un champ de structure sont similaires : on commence par évaluer la left-value sur laquelle porte ce modificateur, et on place le même modificateur sur le chemin résultant.

$$\frac{\langle lv, m\rangle \to \langle \varphi, m'\rangle}{\langle *lv, m\rangle \to \langle *\varphi, m'\rangle}\ (\textsc{Phi-Deref}) \qquad \frac{\langle lv, m\rangle \to \langle \varphi, m'\rangle}{\langle lv.f, m\rangle \to \langle \varphi.f, m'\rangle}\ (\textsc{Phi-Struct})$$

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

$$\frac{\langle lv, m\rangle \to \langle \varphi, m'\rangle \qquad \langle e, m'\rangle \to \langle v, m''\rangle}{\langle lv[e], m\rangle \to \langle \varphi[v], m''\rangle}\ (\textsc{Phi-Array})$$

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

Expressions

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

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

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

$$\frac{\langle lv, m\rangle \to \langle \varphi, m'\rangle \qquad v = m'[\varphi]}{\langle lv, m\rangle \to \langle v, m'\rangle}\ (\textsc{Exp-Lv})$$

Pour évaluer une expression constituée d'un opérateur, on évalue une expression, puis l'autre (l'ordre d'évaluation, est encore imposé : de gauche à droite). À chaque opérateur $\diamond$, correspond un opérateur sémantique $\widehat{\diamond}$ qui agit sur les valeurs. Par exemple, l'opérateur $\widehat{+}$ est l'addition classique entre entiers. Afin d'interdire la division par zéro, celle ci et le modulo sont traités dans une règle à part.

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle}{\langle \diamond\, e, m\rangle \to \langle \widehat{\diamond}\, v, m'\rangle}\ (\textsc{Exp-UnOp})$$

$$\frac{\diamond \notin \{/, \%\} \qquad \langle e_1, m\rangle \to \langle v_1, m'\rangle \qquad \langle e_2, m'\rangle \to \langle v_2, m''\rangle}{\langle e_1 \diamond e_2, m\rangle \to \langle v_1 \,\widehat{\diamond}\, v_2, m''\rangle}\ (\textsc{Exp-BinOp})$$

$$\frac{\diamond \in \{/, \%\} \qquad \langle e_1, m\rangle \to \langle v_1, m'\rangle \qquad \langle e_2, m'\rangle \to \langle v_2, m''\rangle v_2 \neq \widehat{0}}{\langle e_1 \diamond e_2, m\rangle \to \langle v_1 \,\widehat{\diamond}\, v_2, m''\rangle}\ (\textsc{Exp-Div})$$

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

$$\frac{a = \text{Lookup}(x, m)}{\langle \&x, m\rangle \to \langle a, m\rangle}\ (\textsc{Exp-AddrOf})$$

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

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle \qquad \langle l\, v, m'\rangle \to \langle \varphi, m''\rangle \qquad m''' = m''[\varphi \leftarrow v]}{\langle l\, v \leftarrow e, m\rangle \to \langle v, m'''\rangle}\ (\textsc{Exp-Set})$$

Expressions composées

On commence par définir une opération d'évaluation de plusieurs expressions à la fois : on note

$$\left\langle m, \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix} \right\rangle \to \left\langle m', \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \right\rangle$$

si $\exists (m_1, \ldots, m_n), \forall i \in [1; n-1], \langle e_i, m_i\rangle \to \langle e_{i+1}, m_{i+1}\rangle$ avec $m = m_1$ et $m' = m_n$.

Notons que l'évaluation se fait encore de gauche à droite. On utilise la notation vecteur colonne pour signifier qu'il s'agit ici de métasyntaxe (il n'y a pas de tuples dans le langage).

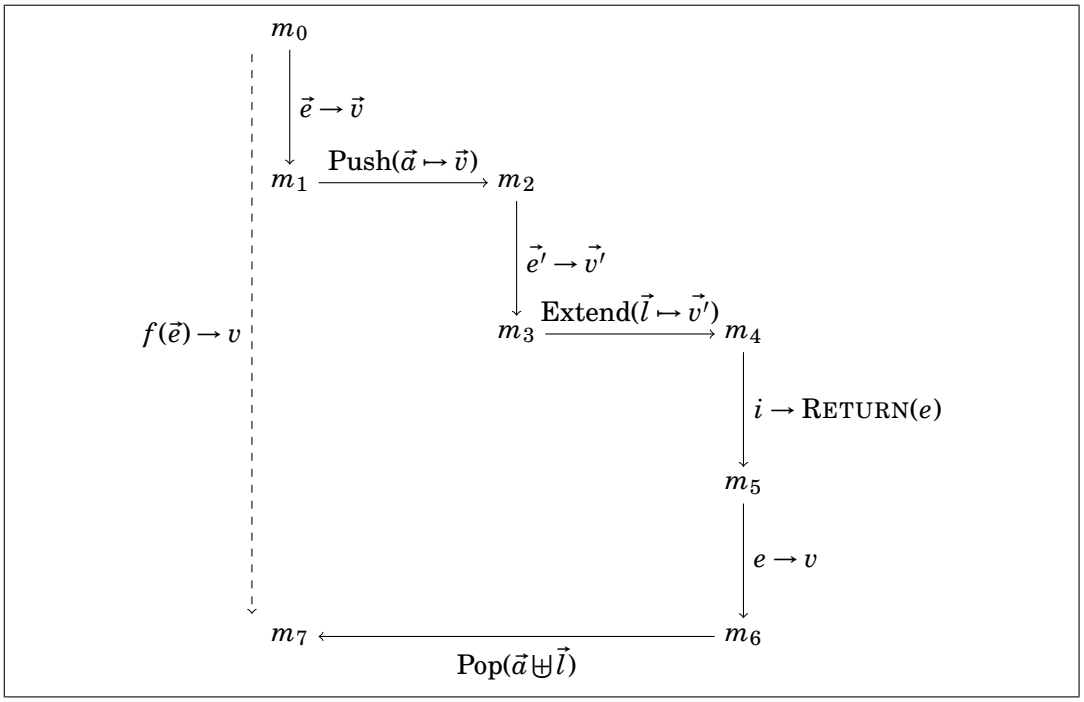


FIGURE 5.4 – L'appel d'une fonction

Cette évaluation chaînée est au coeur de la règle suivante qui permet d'évaluer les structures : à une structure (syntaxique) correspond une valeur structurelle dont les champs sont ceux de la première structure évalués :

$$\frac{\left\langle m, \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix} \right\rangle \to \left\langle m', \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \right\rangle}{\langle \{f_1 : e_1; \ldots; f_n : e_n\}, m \rangle \to \langle \{f_1 : v_1; \ldots; f_n : v_n\}, m' \rangle} \text{ (EXP-STRUCT)}$$

L'appel de fonction repose également sur cette évaluation multiple. Tout d'abord, les arguments sont évalués et placés dans un nouveau cadre de pile. Puis les expressions qui initialisent les variables locales sont elle aussi évaluées et ajoutées à ce même cadre de pile (opérateur Extend). Ensuite, le corps de la fonction est évalué jusqu'à se réduire en une instruction RETURN(v). Puis, le cadre précédemment utilisé est dépilé.

$$\frac{\begin{array}{l} f = (i,(a_1,\ldots,a_n),((l'_1,l'_1),\ldots,(l'_p,e'_p))) \qquad \left\langle m_0, \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix} \right\rangle \to \left\langle m_1, \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \right\rangle \\ m_2 = \text{Push}(m_1,((a_1,v_1),\ldots,(a_n,v_n))) \qquad \left\langle m_2, \begin{pmatrix} e'_1 \\ \vdots \\ e'_p \end{pmatrix} \right\rangle \to \left\langle m_3, \begin{pmatrix} v'_1 \\ \vdots \\ v'_p \end{pmatrix} \right\rangle \\ m_4 = \text{Extend}(m_3,((l_1,v_1),\ldots,(l_n,v_n))) \qquad \langle i, m_4\rangle \to \langle \textsc{Return}(e), m_5\rangle \\ \langle e, m_5\rangle \to \langle v, m_6\rangle \qquad m_7 = \text{Pop}(m_6) \qquad m_8 = \text{Cleanup}(m_7) \end{array}}{\langle f(e_1,\ldots,e_n), m_0\rangle \to \langle v, m_8\rangle} \text{ (Exp-Call)}$$

Cette évaluation est décrite dans la figure 5.4.

5.6 Instructions

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

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

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

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

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

Les seuls cas terminaux sont Pass et Return(e).

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

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

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle}{\langle e, m\rangle \to \langle \textsc{Pass}, m'\rangle} \text{ (Exp)}$$

Pour traiter l'alternative, on a besoin de 2 règles. Elles commencent de la même manière, en évaluant la condition. Si le résultat est 0 (et seulement dans ce cas), c'est la règle IF-FALSE qui est appliquée et l'instruction revient à évaluer la branche "else". Dans les autres cas, c'est la règle IF-TRUE qui s'applique et la branche "then" qui est prise.

$$\frac{\langle e, m\rangle \to \langle \widehat{0}, m'\rangle \qquad \langle i_f, m'\rangle \to \langle i', m''\rangle}{\langle \text{IF}(e)\{i_t\}\text{ELSE}\{i_f\}, m\rangle \to \langle i', m''\rangle}\ (\text{IF-FALSE})$$

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

Le traitement de la boucle est une simple règle de réécriture :

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

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

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

$\widehat{=}$

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

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

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

$\widehat{=}$

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

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

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

5.7 Phrases

$$m \vdash p \to m'$$

$$\frac{?}{m \vdash f \to m + f?}\ (\textsc{Ph-Fun}) \qquad \frac{}{m \vdash \text{struct } s\{\ldots\} \to m}\ (\textsc{Ph-Struct})$$

$$\frac{\langle e, m\rangle \to \langle v, m'\rangle}{m \vdash e \to m'}\ (\textsc{Ph-Exp}) \qquad \frac{\langle e, m\rangle \to \langle v, m'\rangle}{(s, g) \vdash x = e \to (s, (x, v) :: g)}\ (\textsc{Ph-Var})$$

5.8 Exécution

$$\frac{}{m \vdash [] \to^* m}\ (\textsc{Ph*-Nil}) \qquad \frac{m \vdash p \to m' \qquad m' \vdash ps \to^* m''}{m \vdash p :: ps \to^* m''}\ (\textsc{Ph*-Cons})$$

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

5.9 Exemple : l'algorithme d'Euclide

Version par divisions successives :

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

Soit :

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

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

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

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

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

donc

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

$$\begin{aligned}
s_0 &\rightarrow \langle \text{IF}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t; \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}\}; \text{RETURN}(a), m_0\rangle && (5.1)\\
&\rightarrow \langle t\leftarrow b; b\leftarrow a\%b; a\leftarrow t; \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}; \text{RETURN}(a), m_0\rangle && (5.2)\\
&\rightarrow \langle b\leftarrow a\%b; a\leftarrow t; \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}; \text{RETURN}(a), m_0\rangle && (5.3)\\
&\rightarrow \langle a\leftarrow t; \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}; \text{RETURN}(a), m_0''\rangle && (5.4)\\
&\rightarrow \langle \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}; \text{RETURN}(a), m_1\rangle && (5.5)\\
&\rightarrow \langle \text{IF}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t; \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}\}; \text{RETURN}(a), m_1\rangle && (5.6)\\
&\rightarrow \langle t\leftarrow b; b\leftarrow a\%b; a\leftarrow t; \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}; \text{RETURN}(a), m_1\rangle && (5.7)\\
&\rightarrow \langle \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}; \text{RETURN}(a), m_2\rangle && (5.8)\\
&\rightarrow \langle \text{IF}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t; \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}\}; \text{RETURN}(a), m_2\rangle && (5.9)\\
&\rightarrow \langle t\leftarrow b; b\leftarrow a\%b; a\leftarrow t; \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}; \text{RETURN}(a), m_2\rangle && (5.10)\\
&\rightarrow \langle \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}; \text{RETURN}(a), m_3\rangle && (5.11)\\
&\rightarrow \langle \text{IF}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t; \text{WHILE}(b\neq 0)\{t\leftarrow b; b\leftarrow a\%b; a\leftarrow t\}\}; \text{RETURN}(a), m_3\rangle && (5.12)\\
&\rightarrow \langle \text{PASS}; \text{RETURN}(a), m_3\rangle && (5.13)\\
&\rightarrow \langle \text{RETURN}(a), m_3\rangle && (5.14)
\end{aligned}$$

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

TODO

- phrases
- séparer les opérateurs unaires et binaires
- interdire d'avoir plusieurs variables qui ont le même nom dans un cadre
- dédupliquer la def de l'état mémoire
- coquille opeérateur
- figure "interpréteur" -> "état mémoire"
- accesseurs : utiliser des majs
- liste d'assos -> fonction
- syntaxe concrète
- procédures vs fonctions
- pointeurs sur fonction
- arithmétique de pointeurs
- top ?
- si on prend l'adresse d'une locale, et qu'on la redéférence, comment s'assurer qu'il n'y a pas de dangling pointer ? (ça peut être important pour la soundness de l'analyse)

  Options :

  - on fait juste une restriction (forget) au pop (solution actuelle).
  - on donne à chaque stackframe un numéro fresh. Cela revient à considérer la stack comme un heap.

CHAPITRE 6

TYPAGE

Dans ce chapitre, nous enrichissons le langage défini dans le chapitre 5 d'un système de types. Celui-ci permet de séparer les programmes bien formés, comme

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

des programmes mal formés comme

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

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

6.1 Définitions

Type	$t ::= \textsc{Int}$	
	$\mid \textsc{Float}$	
	$\mid \tau[]$	Tableau
	$\mid \tau*$	Pointeur
	$\mid \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$	Structure
	$\mid (\tau_1, \ldots, \tau_n) \to \tau_r$	Fonction

Environnement de typage

$$\Gamma ::= \varepsilon \mid (a, t), \Gamma'$$

6.2 Typage

Définition 6.1 (Typage d'une expression)

$$\Gamma \vdash e : t$$

Définition 6.2 (Typage d'une instruction)

$$\Gamma \vdash i$$

Définition 6.3 (Typage d'une phrase)

$$\Gamma, S \vdash p \to \Gamma', S'$$

6.3 Expressions

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

$$\frac{(x,t) \in \Gamma}{\Gamma \vdash x : t}\ (\text{LV-VAR}) \qquad \frac{\Gamma \vdash x : t*}{\Gamma \vdash *x : t}\ (\text{LV-DEREF}) \qquad \frac{}{\Gamma \vdash x.f : t}\ (\text{LV-FIELD})$$

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

$$\frac{\Gamma \vdash lv : t \qquad \Gamma \vdash e : t}{\Gamma \vdash lv \leftarrow e : t}\ (\text{SET}) \qquad \frac{\forall i, 1 \le i \le n \Rightarrow \Gamma \vdash e_i : t_i}{\Gamma \vdash \{f_1 : e_1; \ldots; f_n : e_n\} : \{f_1 : t_1; \ldots; f_n : t_n\}}\ (\text{STRUCT})$$

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

$$\frac{\forall i, 1 \le i \le n \Rightarrow \Gamma \vdash e_i : t_i \qquad \Gamma \vdash f : (t_1, \ldots, t_n) \to t}{\Gamma \vdash f(e_1, \ldots, e_n) : t}\ (\text{CALL})$$

Opérateurs

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

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

De même sur FLOAT.

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

Les opérateurs de comparaison peuvent s'appliquer à deux opérandes de types "comparables". On introduit donc un jugement COMPARABLE(τ) qui est vrai pour les types INT, FLOAT et pointeurs. Les comparaisons renvoient alors un INT.

$$\frac{\tau \in \{\text{INT}, \text{FLOAT}\}}{\text{COMPARABLE}(\tau)}\ (\text{CMP-NUM}) \qquad \frac{}{\text{COMPARABLE}(\tau*)}\ (\text{CMP-PTR})$$

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

Les opérateurs unaires de négations "−" et "−." s'appliquent respectivement aux INT et aux FLOAT.

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

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

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

6.4 Instructions

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

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

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

6.5 Fonctions

$$\frac{??}{\Gamma \vdash f : (t_1, \ldots, t_n) \to t}\ (\text{FUN})$$

6.6 Programme

$$\frac{\forall i, 1 \leq i \leq p \Rightarrow \Gamma e_i : t_i}{\Gamma \vdash ((f_1, \ldots, f_n), ((x_1, e_1), \ldots, (x_p, e_p)))} \text{ (PROG)}$$

TODO

- utiliser des phrases toplevel
- résoudre le problème du return

CHAPITRE 7

SÉMANTIQUE STATIQUE

Ici nous enrichissons le langage défini dans le chapitre 5 d'un système de types. Celui-ci permet d'obtenir plus de garanties que celui de C tel que décrit dans [ISO99].

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

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

7.1 Règles de typage

7.1.1 Types

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.

Les types des valeurs sont :

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

L'ensemble des types possibles (défini inductivement ci-dessus) sera noté TYP, et l'ensemble des variables de type par VARTYP.

7.1.2 Schémas de type

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

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

Un schéma de types correspond à un ensemble de types. Prenons l'exemple de la fonction identité : elle a pour schéma de types $\forall a.a \to a$, ce qui signifie que pour chaque type τ, on peut l'utiliser avec le type $\tau \to \tau$. Plus précisément, cela veut dire que puisque a est quantifiée, on peut le substituer par n'importe quel autre type.

Définition 7.1 (Substitution) *Une substitution est une fonction partielle de* VAR-TYP *dans* TYP. *Elle sera notée par exemple* $s = \{a \mapsto \text{INT}, b \mapsto (\text{FLOAT} \to \text{INT})\}$.

Le domaine de définition d'une substition est noté $Dom(s)$.

On définit aussi l'application d'une substitution sur un type quelconque : si s *est une substitution,* $\overline{s}$ *est son extension définie par :*

$$\begin{aligned}
\overline{s}(a) &= s(a) && \text{si } a \text{ est une variable}\\
\overline{s}(c) &= c && \text{si } c \text{ est une constante}\\
\overline{s}([\tau]) &= [\overline{s}(\tau)]\\
\overline{s}(\tau*) &= \overline{s}(\tau)*\\
\overline{s}(\tau_1,\ldots,\tau_n) \to \tau_r &= (\overline{s}(\tau_1),\ldots,\overline{s}(\tau_n)) \to \overline{s}(\tau_r)\\
\overline{s}(\{f_1:\tau_1,\ldots,f_n:\tau_n\}) &= \{f_1:\overline{s}(\tau_1),\ldots,f_n:\overline{s}(\tau_n)\}
\end{aligned}$$

Par souci de simplicité, on notera s *pour* $\overline{s}$.

Définition 7.2 (Instanciation) *Un schéma de types* $\sigma = \forall \vec{a}.\tau$ *peut être instancié en un type* μ *s'il existe une substition* s *telle que :*

- $Dom(s) \subseteq \vec{a}$
- $s(\tau) = \mu$

On note alors $\mu \leq \sigma$.

Définition 7.3 (Variables libres) *Les variables libres d'un type sont l'ensemble des variables de types qui apparaissent dans celui-ci :*

$$
\begin{aligned}
FreeVars(a) &= \{a\} && \text{si } a \text{ est une variable}\\
FreeVars(c) &= \emptyset && \text{si } c \text{ est une constante}\\
FreeVars([\tau]) &= FreeVars(\tau)\\
FreeVars(\tau*) &= FreeVars(\tau)\\
FreeVars((\tau_1,\ldots,\tau_n)\rightarrow\tau_r) &= \bigcup_{i=1}^{n} FreeVars(\tau_i)\cup FreeVars(\tau_r)\\
FreeVars(\{f_1:\tau_1,\ldots,f_n:\tau_n\}) &= \bigcup_{i=1}^{n} FreeVars(\tau_i)
\end{aligned}
$$

On étend cette définition aux schémas de types :

$$FreeVars(\forall\vec{a}.\tau) = FreeVars(\tau) - \vec{a}$$

ainsi qu'aux contextes de typage :

$$
\begin{aligned}
FreeVars(\varepsilon) &= \emptyset\\
FreeVars(\Gamma, x:\tau) &= FreeVars(\Gamma)\cup FreeVars(\tau)\\
FreeVars(\Gamma, f:\sigma) &= FreeVars(\Gamma)\cup FreeVars(\sigma)
\end{aligned}
$$

Définition 7.4 (Généralisation) *La généralisation consiste à construire un schéma de type à partir d'un type, en quantifiant sur les variables libres :*

$$Gen(\tau,\Gamma) = \forall\vec{a}.\tau \quad où \quad \vec{a} = FreeVars(\tau) - FreeVars(\Gamma)$$

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 instanciation de σ.

7.1.3 Environnements de typage

Chaque jugement de typage est effectué dans un environnement de typage Γ particulier, qui contient le contexte nécessaire : ici, le type des fonctions et variables du programme.

	$\Gamma ::= (\Gamma_{fun}, \Gamma_{var})$	Fonctions, variables
Environnements	$\Gamma_{fun} ::= \varepsilon$	Environnement vide
	$\mid \Gamma_{fun}, f : \sigma$	Ajout d'une fonction
	$\Gamma_{var} ::= \varepsilon$	Environnement vide
	$\mid \Gamma_{var}, v : \tau$	Ajout d'une variable

Lorsque ce n'est pas ambigü, si $\Gamma = (\Gamma_{fun}, \Gamma_{var})$ on notera les extensions d'environnement $\Gamma, f : \sigma$ pour $((\Gamma_{fun}, f : \sigma), \Gamma_{var})$ et $\Gamma, x : \tau$ pour $(\Gamma_{fun}, (\Gamma_{var}, x : \tau))$. De même, on notera $(f, \sigma) \in \Gamma$ si $(f, \sigma) \in \Gamma_{fun}$, ou $(x, \tau) \in \Gamma$ si $(x, \tau) \in \Gamma_{var}$.

7.1.4 Jugements de typage

Un des principes du typage est d'associer à chaque expression un type, qui décrit le genre des valeurs produites par l'évaluation de cette fonction.

Définition 7.5 (Jugement de typage) *Un jugement de typage est de la forme* $\Gamma \vdash e : \tau$ *et se lit "sous* Γ*, e est typable en* τ*".*

Les instructions et blocs, au contraire, n'ont pas de type.

Définition 7.6 (Bloc bien typé) *On note* $\Gamma \vdash i$ *pour "sous* Γ*, i est bien typé", c'est à dire que ces sous expressions sont typables 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*).*

Le cas des fonctions est particulier puisque celles-ci ont un schéma de types qui leur est associé.

Définition 7.7 (Fonction bien typée) *On note* $\Gamma \vdash f : \sigma$ *le fait qu'une fonction f est typable en un schéma* σ *dans* Γ*.*

Enfin, la notion de programme bien typé est intrinsèque : elle se fait indépendemment d'un environnement externe.

Définition 7.8 (Programme bien typé) *Un programme P est bien typé s'il existe un environnement* Γ *permettant de bien typer toutes les composantes (fonctions, globales et bloc d'initialisation) d'un programme. On notera alors* $\vdash P$*.*

7.1.5 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).

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

7.1.6 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.

$$\frac{}{\Gamma \vdash \varepsilon} \text{ (PASS)} \qquad \frac{\Gamma \vdash s \qquad \Gamma \vdash b}{\Gamma \vdash s;b} \text{ (SEQ)} \qquad \frac{\Gamma \vdash b}{\Gamma \vdash \text{WHILE}(1)\{b\}} \text{ (WHILE)}$$

$$\frac{}{\Gamma \vdash \text{GOTO } l} \text{ (GOTO)} \qquad \frac{\Gamma \vdash b}{\Gamma \vdash \text{DO}\{b\}\text{WITH } l :} \text{ (DOWITH)}$$

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

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

7.1.7 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.

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

Si une expression a un type pointeur, en la déréférençant 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 composite

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

7.1.8 Expressions

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

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

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.

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

La règle concernant les pointeurs sur fonction est similaire, à ceci près qu'une fonction a un schéma de type, qu'il faut instancier afin d'obtenir un type pointeur.

$$\frac{\Gamma \vdash f : \sigma \qquad \tau \leq \sigma}{\Gamma \vdash \&f : \tau}\ (\text{ADDROFFUN})$$

7.1.9 Fonctions

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

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

7.1.10 Instructions

Une affectation est bien typée si elle faite entre une left-value et une expression de même type.

$$\frac{\Gamma \vdash lv : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash lv \leftarrow e} \text{(ASSIGN)}$$

$$\frac{\Gamma \vdash lv : \tau_{ret} \qquad \Gamma \vdash fe : \sigma \qquad \Gamma \vdash \vec{e} : \vec{\tau} \qquad (\vec{\tau} \rightarrow \tau_r) \leq \sigma}{\Gamma \vdash lv \leftarrow fe(\vec{e})} \text{(FCALL)}$$

7.2 Limitations

7.2.1 Programmes non typables

7.2.2 Incohérences

CHAPITRE 8

ANALYSE DE PROVENANCE DES POINTEURS

Dans le chapitre 7, 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.

8.1 Éditions et ajouts

Qualificateurs	$q ::= \textsc{Kernel}$	Donnée noyau (sûre)
	$\mid \textsc{User}$	Donnée utilisateur (non sûre)

Ensuite,

Types	$\tau ::= \tau q *$	Pointeur qualifié
	$\mid \ldots$	Reste inchangé

voire :

Environnements $\quad \Gamma ::= \varepsilon \mid \Gamma, x : \tau\ q$

Règle de sûreté du déréférencement

$$\frac{\Gamma \vdash e : \tau\ \textsc{Kernel} *}{\Gamma \vdash *e : \tau}\ (\textsc{Lv-Deref-Kernel})$$

8.2 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.

CHAPITRE 9

ANALYSE DE TERMINAISON DES CHAÎNES C

Dans ce chapitre, nous présentons une autre extension au système de types du chapitre 7, similaire à celle du chapitre 8. 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.

9.1 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 protoype :

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

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

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

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

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

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

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

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

9.2 Approche

Cette propriété est un peu différente de la séparation entre espace utilisateur et espace noyau modélisée dans le chapitre 8 : 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].

9.3 Annotation de `string.h`

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

9.3.1 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);
```

9.3.2 Fonctions de concaténation

strcat

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

strncat

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

9.3.3 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);
```

9.3.4 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);
```

9.3.5 Fonctions diverses

memset

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

strerror

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

strlen

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

9.4 Typage des primitives

9.5 Extensions au système de types

9.6 Résultats

Troisième partie

Expérimentation

CHAPITRE 10

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.

10.1 Langages intermédiaires

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

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

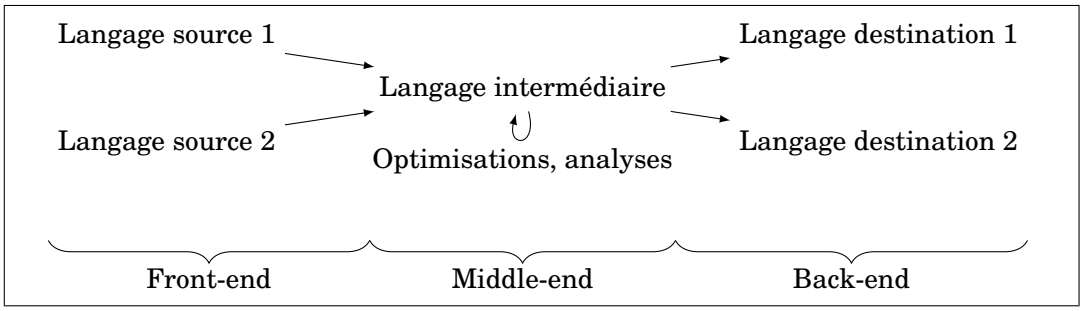


FIGURE 10.1 – Décomposition d'un compilateur : front-ends, middle-end, back-ends

fasse l'intermédiaire entre *front-end* et *back-end* (figure 10.1). Depuis ce langage on doit pouvoir exprimer des transformations intermédiaires sur cette représentation (analyses sémantiques, optimisations, etc), mais aussi compiler ce langage vers un langage machine.

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

Langages

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

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

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

Objective Caml [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 10.2.

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].

10.2 Newspeak

10.3 Chaîne de compilation

Mettre à jour la figure

La compilation vers C est faite en trois étapes (figure 10.2) : prétraitement du code source, compilation de C prétraité vers NEWSPEAK, puis compilation de NEWSPEAK vers ce langage.

10.3.1 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

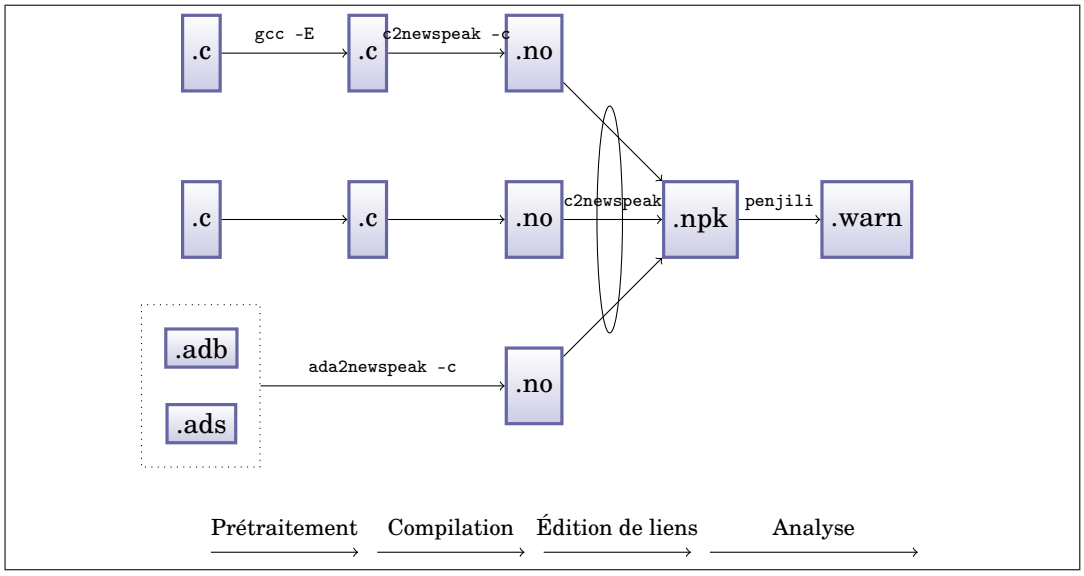


FIGURE 10.2 – Compilation depuis Newspeak

macros sont développées, les constantes remplacées par leurs valeurs, les commentaires supprimés, les fichiers d'en-tête inclus, etc.

10.3.2 Compilation (levée des ambigüités)

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

Au contraire, NEWSPEAK propose un nombre réduit de constructions. Rappelons que le but de ce langage est de faciliter l'analyse statique : des constructions orthogonales permettent donc d'éviter la duplication de règles sémantique, ou de code lors de l'implémentation d'un analyseur.

Par exemple, plutôt que de fournir une boucle *while*, une boucle *do/while* et une boucle for, NEWSPEAK fournit une unique boucle WHILE(1){}. La sortie de boucle est compilée vers un GOTO , 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].

10.3.3 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;
}
```

10.3.4 Implantation de l'algorithme de typage

Commençons par étudier le cas du lambda-calcul simplement typé (figure 10.3). Prenons l'exemple de la fonction suivante [1] :

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

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.

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

Termes

$t ::= x$	Variable
$\mid \lambda x.t$	Abstraction
$\mid t$	Application
$\mid n$	Entier
$\mid f$	Flottant
$\mid (t,t)$	Couple
$\mid \text{fst } t$	Projection gauche
$\mid \text{snd } t$	Projection droite

Types

$\tau ::= \textsc{Int}$	Entier
$\mid \textsc{Float}$	Flottant
$\mid \tau \to \tau$	Fonction
$\mid \tau \times \tau$	Produit

Contextes

$\Gamma ::= \varepsilon$	Contexte vide
$\mid \Gamma, x : \tau$	Extension

Règles

$$\frac{}{\Gamma \vdash n : \textsc{Int}}\ (\textsc{Int}) \qquad \frac{}{\Gamma \vdash f : \textsc{Float}}\ (\textsc{Float}) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1}{\Gamma \vdash f x : \tau_2}\ (\textsc{App}) \qquad \frac{\Gamma, x : \tau_1 \vdash y : \tau_2}{\Gamma \vdash \lambda x.y : \tau_1 \to \tau_2}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash x : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } x : \tau_1}\ (\textsc{Proj-g}) \qquad \frac{\Gamma \vdash x : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } x : \tau_2}\ (\textsc{Proj-d})$$

$$\frac{\Gamma \vdash x : \tau_1 \qquad \Gamma \vdash y : \tau_2}{\Gamma \vdash (x,y) : \tau_1 \times \tau_2}\ (\textsc{Tup})$$

FIGURE 10.3 – Lambda calcul simplement typé avec entiers, flottants et couples

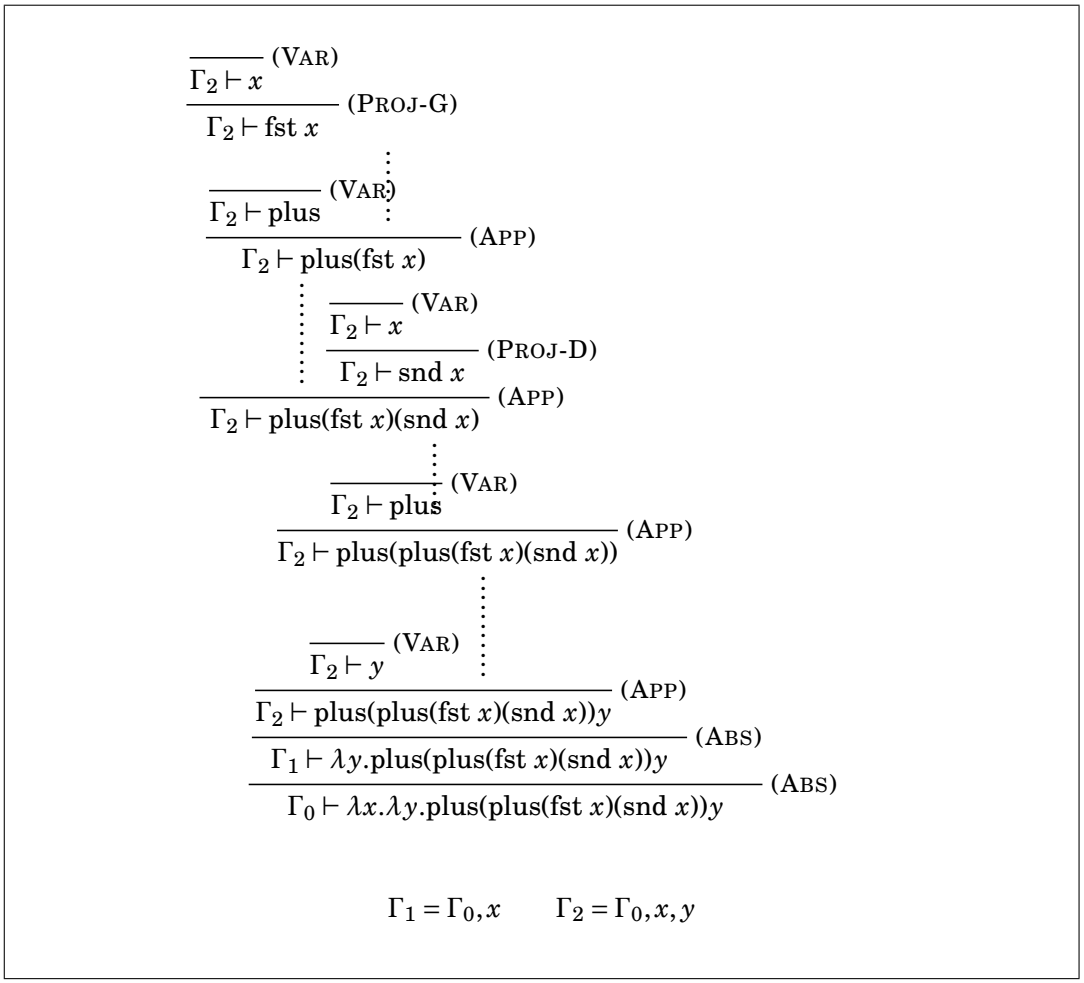


FIGURE 10.4 – Arbre d'inférence : règles à utiliser

Mais comment faire pour implanter cette analyse ? En fait le système de types de la figure 10.3 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 10.4) [2]

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

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

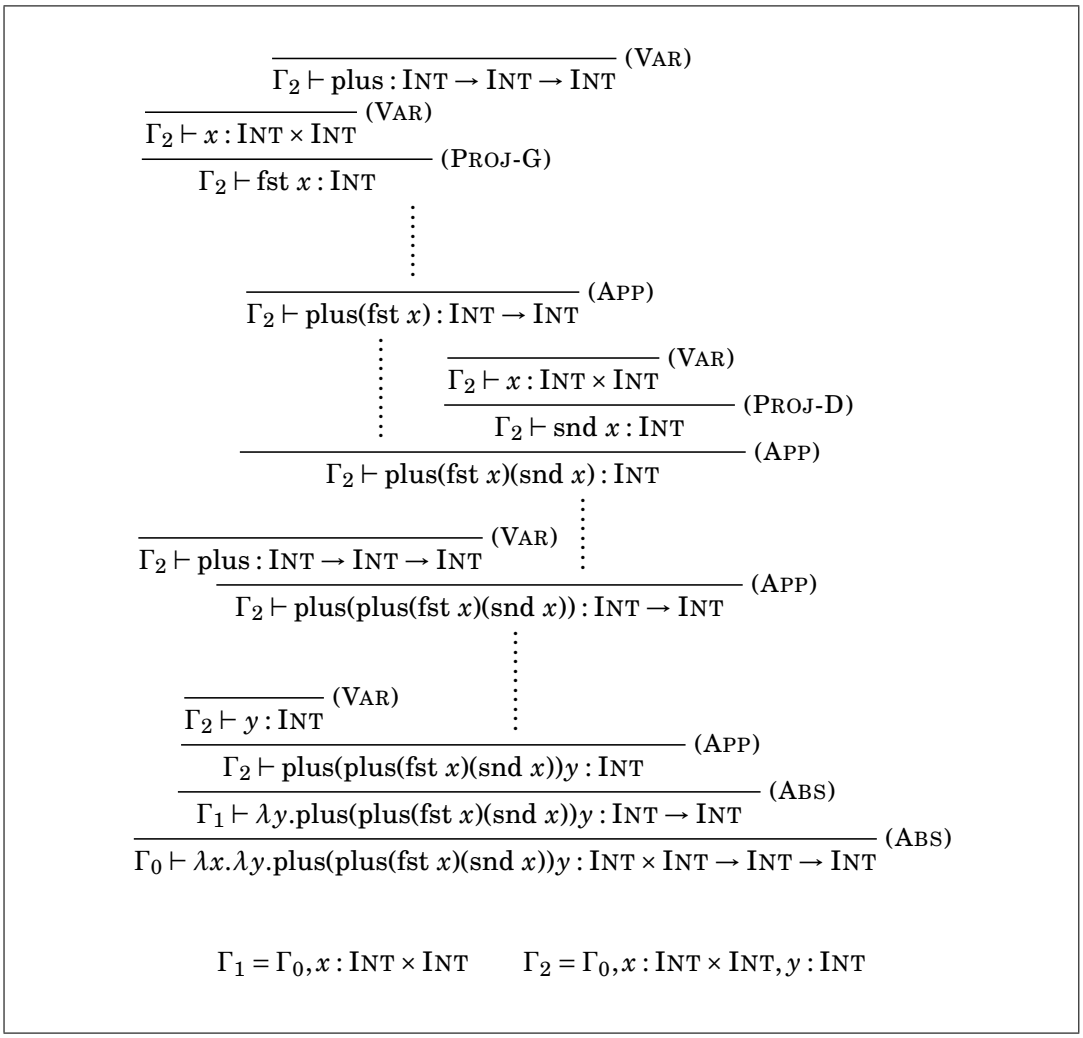


FIGURE 10.5 – Arbre d'inférence complet

$$\frac{\Gamma \vdash \ldots : \tau_3 \qquad \Gamma \vdash \ldots : \tau_1}{\Gamma \vdash \ldots : \tau_2} \text{(APP)}$$

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

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 -> type concret".

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

C'est faux

Une fois toutes les substitutions effectuées, on obtient un arbre de typage correct (figure 10.5, donc un programme totalement inféré.

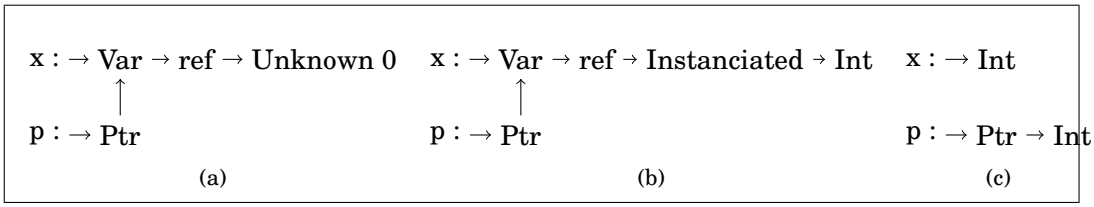


FIGURE 10.6 – Unification par partage

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

FIGURE 10.7 – Compilation d'un programme C - avant

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 10.6).

D'abord 10.6a, ensuite 10.6b, et enfin 10.6c.

Prenons l'exemple de la figure 10.7 et typons-le "à la main". On commence par oublier toutes les étiquettes de type présentes dans le programme. Celui-ci devient alors :

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

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

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

and const_type =
  | Int_type
  | Float_type

and ml_type =
```

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

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

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

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

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

```
module Env : sig
  type t

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

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

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

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

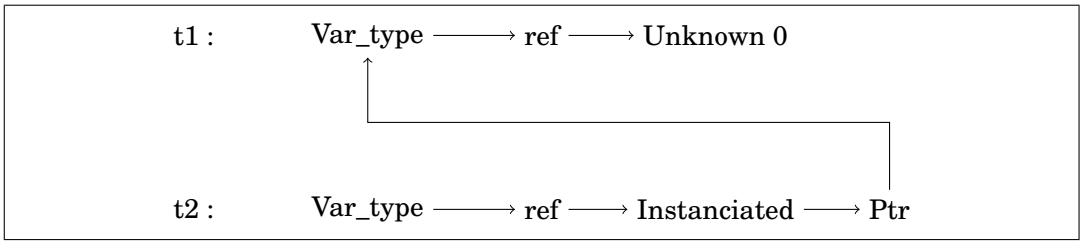


FIGURE 10.8 – Unification : partage

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

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

Ici il se passe plusieurs choses intéréssantes. D'une part nous faisont appel à une fonction externe typeof qui retourne le type d'une expression sous un environnement (dans une implantation complète il s'agirait d'un appel récursif). Dans ce cas, `typeof lhs1 env` est identique à `Env.get lhs1 env` et `typeof rhs1 env` à `Ptr_type t1`. L'autre aspect intéressant est la dernière ligne : la fonction `unify` va modifier en place les représentations des types afin de les rendre égales. L'implantation de `unify` sera décrite plus tard. Dans ce cas précis, elle va faire pointer la référence dans t2 vers t1 (figure 10.8).

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

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

Ici `typeof lhs2 env` est identique à `Ptr_type (Env.get "p" env)` et `typeof lhs2 env` à `Const_type Int_type`. Et dans cas, l'unification doit se faire entre t1 et `Const_type Int_type` : cela mute la référence derrière t1 (figure 10.9).

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

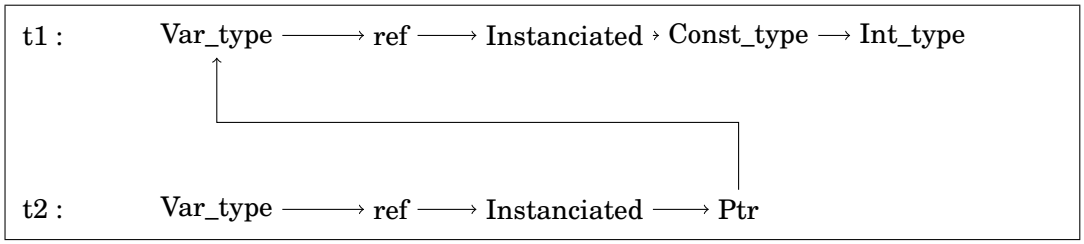


FIGURE 10.9 – Unification par mutation de références

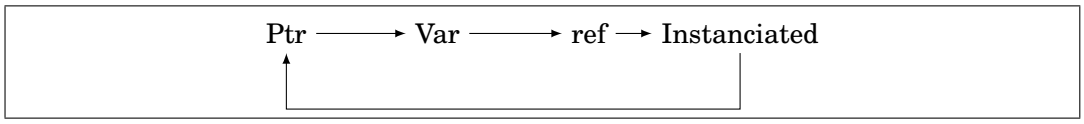


FIGURE 10.10 – Cycle dans le graphe de types

10.3.5 Algorithme d'unification

Voici une implantation de la fonction `unify`.

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

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

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

- si les deux types sont inconnus (de la forme `Var (ref (Instanciated t))`), on fait pointer l'une des deux références vers le premier type. Notons que cela crée un type de la forme `Var (ref (Instanciated (Var (ref (Unknown n)))))` qui sera réduit lors d'une prochaine étape d'unification.
- si un type est inconnu et pas l'autre, il faut de la même manière affecter la référence. Mais en faisant ça inconditionnellement, cela peut poser problème : par exemple en tentant d'unifier `a` avec `Ptr(a)` on pourrait créer un cycle dans le graphe (figure 10.10). Pour éviter cette situation, il suffit de s'assurer que le type inconnu n'est pas présent dans le type à affecter.
- si les deux types sont des types de base (comme INT ou FLOAT) égaux, on ne fait rien.
- si les deux types sont des constructeurs de type, il faut que les constructeurs soient égaux. On unifie en outre leurs arguments deux à deux.
- dans les autres cas, l'algorithme échoue.
- TODO sous typage pour les structures

TODO :

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

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

FIGURE 10.11 – Compilation d'un programme C - après

Le programme C (figure 10.7) est compilé ainsi en Tyspeak (figure 10.11).

CHAPITRE 11

ÉTUDE DE CAS : UN PILOTE DE CARTE GRAPHIQUE

11.1 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 implante un appel système, il faut donc empêcher qu'un pointeur passé en paramètre référence le noyau. Autrement dit, il est indispensable de véri-

cf annexe A

FIGURE 11.1 – 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.

fier 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 11.1.

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

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

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.

11.2 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 11.1 : 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`).

11.3 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 11.1 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 [8], fera l'objet d'une démonstration.

11.4 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.

CHAPITRE 12

CONCLUSION

12.1 Limitations

12.2 Perspectives

Code du module noyau

```
/* from drivers/gpu/drm/radeon/radeon_kms.c */
int radeon_info_ioctl(struct drm_device *dev, void *data, struct drm_file *filp)
{
        struct radeon_device *rdev = dev->dev_private;
        struct drm_radeon_info *info;
        struct radeon_mode_info *minfo = &rdev->mode_info;
        uint32_t *value_ptr;
        uint32_t value;
        struct drm_crtc *crtc;
        int i, found;

        info = data;
        value_ptr = (uint32_t *)((unsigned long)info->value);
        value = *value_ptr;
        switch (info->request) {
        case RADEON_INFO_DEVICE_ID:
                value = dev->pci_device;
                break;
        case RADEON_INFO_NUM_GB_PIPES:
                value = rdev->num_gb_pipes;
                break;
        case RADEON_INFO_NUM_Z_PIPES:
                value = rdev->num_z_pipes;
                break;
        case RADEON_INFO_ACCEL_WORKING:
                /* xf86-video-ati 6.13.0 relies on this being false for evergreen */
```

```
                if ((rdev->family >= CHIP_CEDAR) && (rdev->family <= CHIP_HEMLOCK)
                        value = false;
                else
                        value = rdev->accel_working;
                break;
        case RADEON_INFO_CRTC_FROM_ID:
                for (i = 0, found = 0; i < rdev->num_crtc; i++) {
                        crtc = (struct drm_crtc *)minfo->crtcs[i];
                        if (crtc && crtc->base.id == value) {
                                struct radeon_crtc *radeon_crtc = to_radeon_crtc(c
                                value = radeon_crtc->crtc_id;
                                found = 1;
                                break;
                        }
                }
                if (!found) {
                        DRM_DEBUG_KMS("unknown crtc id %d\n", value);
                        return -EINVAL;
                }
                break;
        case RADEON_INFO_ACCEL_WORKING2:
                value = rdev->accel_working;
                break;
        case RADEON_INFO_TILING_CONFIG:
                if (rdev->family >= CHIP_CEDAR)
                        value = rdev->config.evergreen.tile_config;
                else if (rdev->family >= CHIP_RV770)
                        value = rdev->config.rv770.tile_config;
                else if (rdev->family >= CHIP_R600)
                        value = rdev->config.r600.tile_config;
                else {
                        DRM_DEBUG_KMS("tiling config is r6xx+ only!\n");
                        return -EINVAL;
                }
        case RADEON_INFO_WANT_HYPERZ:
                mutex_lock(&dev->struct_mutex);
                if (rdev->hyperz_filp)
                        value = 0;
                else {
                        rdev->hyperz_filp = filp;
                        value = 1;
```

```
                }
                mutex_unlock(&dev->struct_mutex);
                break;
        default:
                DRM_DEBUG_KMS("Invalid request %d\n", info->request);
                return -EINVAL;
        }
        if (DRM_COPY_TO_USER(value_ptr, &value, sizeof(uint32_t))) {
                DRM_ERROR("copy_to_user\n");
                return -EFAULT;
        }
        return 0;
}

/* from 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);
}
```

Todo list

TABLE DES FIGURES

RÉFÉRENCES WEB

[1] The Objective Caml system, documentation and user's manual – release 3.12
http ://caml.inria.fr/pub/docs/manual-ocaml/

[2] Haskell Programming Language – Official Website
http ://www.haskell.org/

[3] Python Programming Language – Official Website
http ://www.python.org/

[4] Perl Programming Language – Official Website
http ://www.perl.org/

[5] Sparse - a Semantic Parser for C
https ://sparse.wiki.kernel.org/index.php/Main_Page

[6] CIL - C Intermediate Language
http ://kerneis.github.com/cil/

[7] The C - - language
http ://www.cminusminus.org/

[8] Penjili project
http ://www.penjili.org/

BIBLIOGRAPHIE

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor (extended version). Research report 6138, INRIA, 2007. 29 pages. 73

[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. 28

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, third edition edition, November 2005. 12

[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. 27

[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. 73

[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. 27

[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`.). 27

[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. 28

[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. 28

[CMP03] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly, 2003. 19

[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. 28

[DRS00] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV : Towards a realistic tool for statically detecting all buffer overflows in C, 2000. 28

[EH94] Ana Erosa and Laurie J. Hendren. Taming control flow : A structured approach to eliminating goto statements. In *In Proceedings of 1994 IEEE International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994. 74

[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. 27, 89

[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. 27

[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. 27

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. 12

[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. 28

[Har88] Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4) :36–38, October 1988. 16, 88

[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. 73, 74, 89

[Int] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 7, 13

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999. 19, 53, 63, 71

[JW04] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004. 27

[KcS07] Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci.*, 174(7) :79–104, 2007. 28

[Ker81] Brian W. Kernighan. Why Pascal is not my favorite programming language. Technical report, AT&T Bell Laboratories, April 1981. 25

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988. 19, 71

[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. 72

[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. 28

[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. 28

[Mer03] J. Merrill. GENERIC and GIMPLE : a new tree representation for entire functions. In *GCC developers summit 2003*, pages 171–180, 2003. 72

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, December 1978. 25

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil : Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag. 73

[oEE08] Institute of Electrical and Electronics Engineers. IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008. 34

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. 19

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 1996. 7, 64

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 21, 28

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003. 19

[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. 72

[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. 27

[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. 28

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21 :2003, 2003. 28

[Spe05] Brad Spengler. grsecurity 2.1.0 and kernel vulnerabilities. *Linux Weekly News*, 2005. 27

[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. 72

[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. 27

[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. 65

[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. 28

[Wal00] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2000. 19