Analyse statique de logiciel système par typage statique fort

Application au noyau Linux

Étienne Millon

24 juillet 2012

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 permettant d'effectuer toutes sortes de calculs et de communications. Le système d'exploitation permet de faire l'interface entre ces niveaux d'abstraction.

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

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

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

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

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

Par exemple, si un programme veut copier des données depuis un cédérom vers la mémoire principale, il devra interroger le bus SATA, interroger le lecteur sur la présence d'un disque dans

le lecteur, activer le moteur, calculer le numéro de trame des données sur le disque, demander la lecture, puis déclencher une copie de la mémoire.

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

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

2.2 Architecture Intel

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

Dans cette section nous décrivons le fonctionnement d'un processeur utilisant une architecture Intel 32 bits. Les exemples de code seront écrit 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].

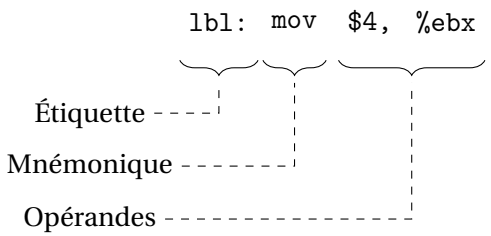
Assembleur

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

Les registres principaux sont nommés EAX, EBX, ECX, EDX, ESI et EDI. Ils peuvent être utilisés pour n'importe quel type d'opération, mais certains sont spécialisés : par exemple il est plus efficace d'utiliser EAX en accumulateur, ou ECX en compteur. ESP est le pointeur de pile (*stack pointer*), qui a un rôle particulier dans les instructions `push` et `pop`. EBP sert de point de repère dans les appels de fonction. EIP contient l'adresse de l'instruction courante.

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

un type d'opération à appliquer sur les opérandes. Elle peut aussi être précédée d'une étiquette, qui correspondra à l'adresse de cette instruction.



Ces opérandes peuvent être de plusieurs types :

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

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

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

Fonctions et conventions d'appel

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

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

- la valeur de retour d'une fonction est stockée dans EAX
- EAX, ECX et EDX peuvent être écrasés sans avoir à les sauvegarder

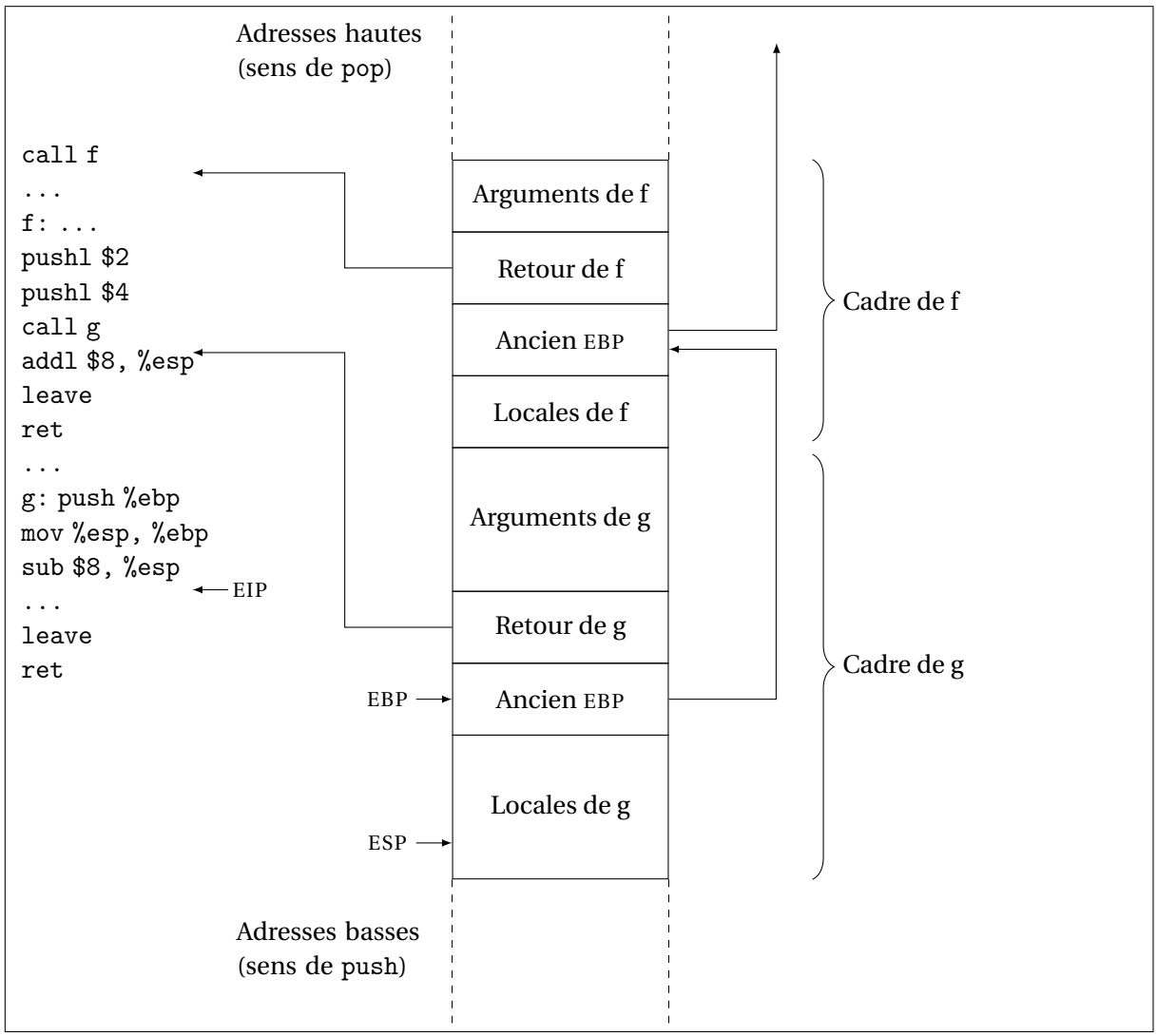


FIGURE 2.1 – Cadres de pile.

- 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.

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.

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.

Tâches, niveaux de privilèges

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

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

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

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

Mémoire virtuelle

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

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

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

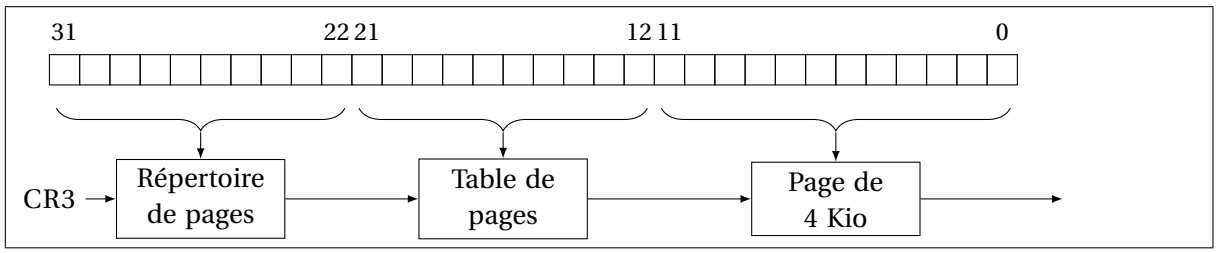


FIGURE 2.3 – Implantation de la mémoire virtuelle

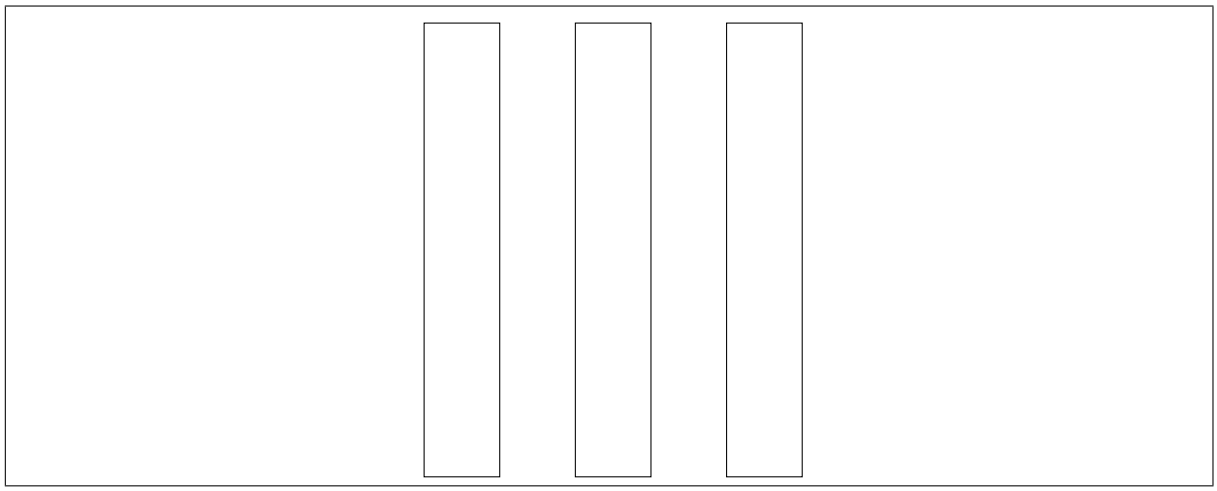


FIGURE 2.4 – Mécanisme de mémoire virtuelle.

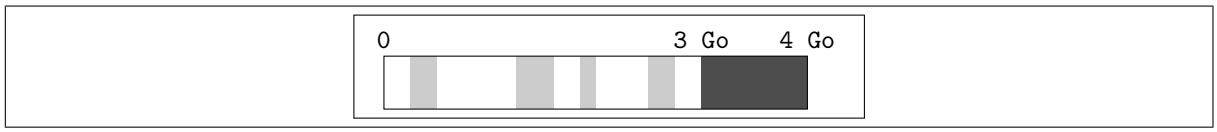


FIGURE 2.5 – L'espace d'adressage d'un processus. En gris clair, les zones accessibles à tous les niveaux de privilèges : code du programme, bibliothèques, tas, pile. En gris foncé, la mémoire du noyau, réservée au mode privilégié.

En ce qui concerne la mémoire, les différentes tâches ont une vision différente de la mémoire physique : c'est-à-dire que deux tâches lisant à une même adresse peuvent avoir un résultat différent. C'est le concept de mémoire virtuelle (fig 2.4).

2.3 Cas de Linux

Dans cette section, nous allons voir comment ses mécanismes sont implantés dans le noyau Linux. Une description plus détaillée pourra être trouvée dans [BP05], ou pour le cas de la mémoire virtuelle, [Gor04].

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

Une notion de tâche similaire à celle décrite dans la section 2.2 existe : elles s'exécutent l'une après l'autre, le changement s'effectuant sur interruptions.

Pour faire appel aux services du noyau, le code utilisateur peut faire appel à des appels systèmes, qui sont des fonctions exécutées par le noyau. Chaque tâche doit donc avoir deux piles : une pile "utilisateur" qui sert pour l'application elle-même, et une pile "noyau" qui sert aux appels système.

Grâce à la mémoire virtuelle, chaque processus possède sa propre vue de la mémoire dans son espace d'adressage (figure 2.5). Au moment de changer le processus en cours, l'ordonnanceur charge une valeur propre au nouveau processus dans CR3. Les adresses basses (inférieures à `PAGE_OFFSET` = 3 Gio = `0xc0000000`) sont réservées à l'utilisateur. On y trouvera par exemple :

- 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.

Appels système

Les programmes utilisateur s'exécutant en *ring* 3, ils ne peuvent pas contenir d'instructions privilégiées, et donc ne peuvent pas accéder directement au matériel (c'était le but !). Pour qu'ils puissent interagir avec le système (afficher une sortie, écrire sur le disque...), le mécanisme des appels système est nécessaire. Il s'agit d'une interface de haut niveau entre les *rings* 3 et 0. Du point de vue du programmeur, il s'agit d'un ensemble de fonctions C "magiques" qui font appel au système d'exploitation pour effectuer des opérations.

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

Dans la bibliothèque C

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

Dans la routine de traitement d'interruption

Étant donné la configuration du 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 anyway
        pushl %eax                            # save orig_eax
        CFI_ADJUST_CFA_OFFSET 4
        SAVE_ALL
        GET_THREAD_INFO(%ebp)
                                        # system call tracing in operation / emulation
```

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*.

```
        testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
        jnz syscall_trace_entry
        cmpl $(nr_syscalls), %eax
        jae syscall_badsys
syscall_call:
        call *sys_call_table(,%eax,4)
        movl %eax,PT_EAX(%esp)              # store the return value
        # ...
        INTERRUPT_RETURN
```

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

Dans l'implantation de l'appel système

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

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

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

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

Retour vers le ring 3

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

2.4 Sécurité des appels système

```
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

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

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

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

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

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

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

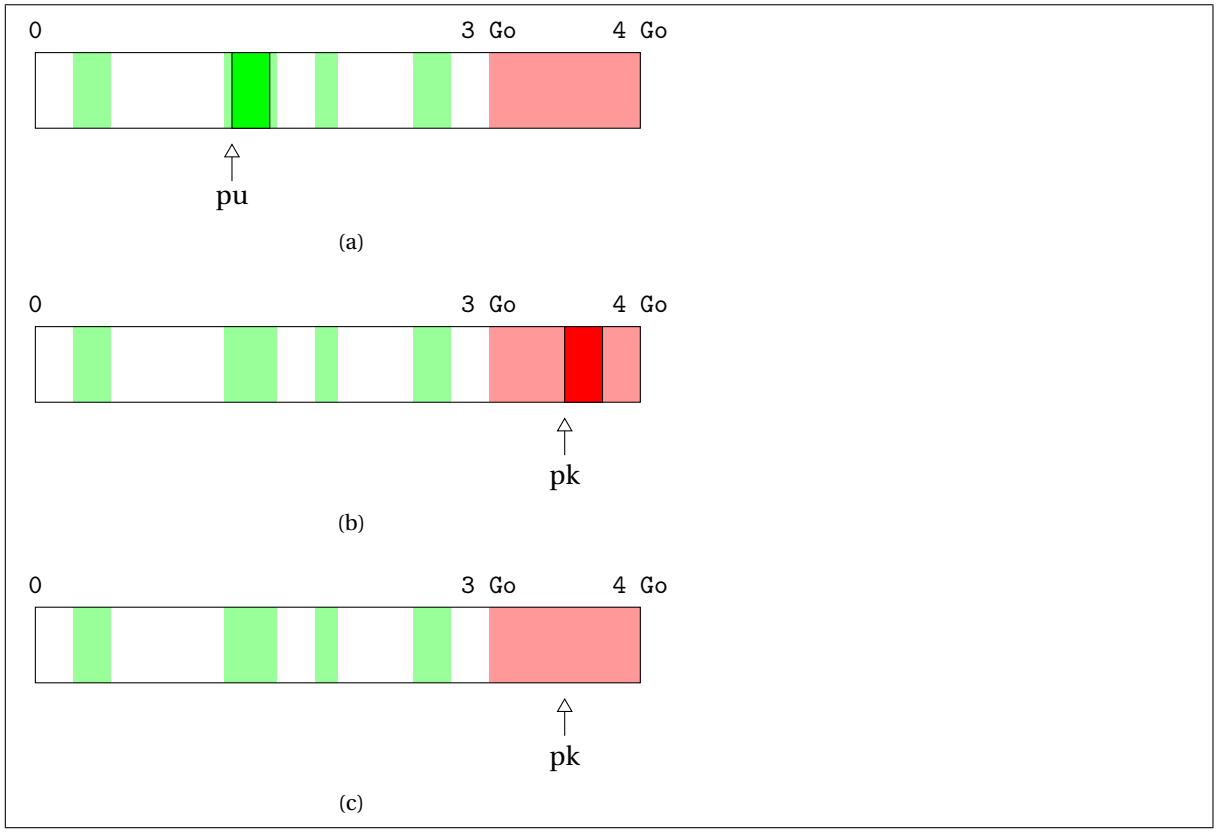
Mais la plupart des appels systèmes communiquent de l'information de manière indirecte, à travers un pointeur. Ainsi la fonction `gettimeofday` a pour prototype :

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

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

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

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


0
3 Go
4 Go
pu
(a)
0
3 Go
4 Go
pk
(b)
0
3 Go
4 Go
pk
(c)

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[LDG+10, CMP03], Haskell[PJ03, OGS08], Python[pyt] et Perl[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.

Dynamique, statique, mixte

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

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

FIGURE 3.1 – Session Python présentant le typage dynamique

Typage dynamique

La première est le typage *dynamique*. Pour différencier les différents types de données, on ajoute une étiquette à chaque valeur. Dans tout le programme, on ne manipulera que des valeurs étiquetées, c'est à dire des couples (donnée, nom de type). Si on veut réaliser l'opération $(\texttt{0x00000001}, Int) + (\texttt{0x0000f000}, Int)$, 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.

```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.

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

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

Benjamin Pierce résume cette approche dans cette définition très générale :

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

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

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

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,

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

FIGURE 3.3 – Transtypage en Java

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).

Fort, faible, sound

Si un système de types statique permet d'éliminer totalement la nécessité de réaliser des tests de typage, on dit qu'il est *fort*. Mais ce n'est que rarement le cas. En effet, il peut y avoir des constructions au sein du langage qui permettent de contourner le système de types, comme un opérateur de 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)
```

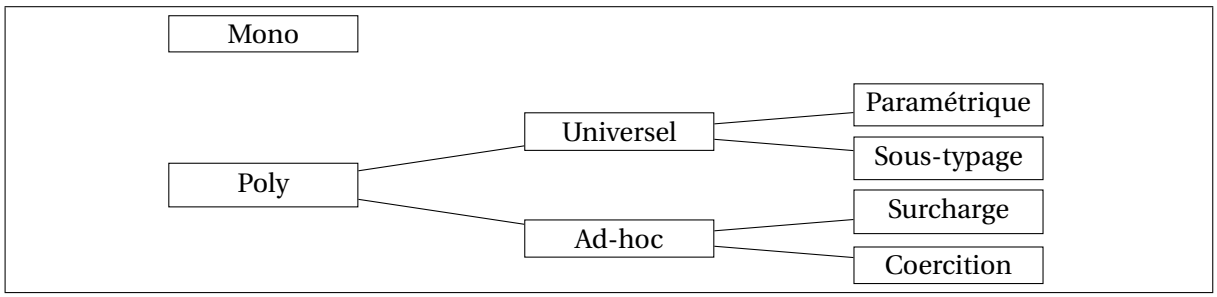


FIGURE 3.4 – Les différents types de polymorphisme.

Polymorphisme

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

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

Monomorphisme

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

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

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

Polymorphisme universel

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

Polymorphisme ad-hoc

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

Polymorphisme paramétrique

[Mil78]

[Ker81]

Historique + citer le papier de Milner sur le polymorphisme

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

FIGURE 3.5 – Fonction de concaténation de listes en OCaml.

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

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

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

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,héritage 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.

ntroduire 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).

Expressivité, garanties, types dépendants

3.3 Exemples

Faible dynamique : Perl

Faible statique : C

Fort dynamique : Python

Fort statique : OCaml

Fort statique à effets typés : Haskell

Theorem prover : Coq

CHAPITRE 4

ÉTAT DE L'ART

Analyse légère : Sparse [TTL]

Coccinelle [BDH+09] [PTS+11]

lire coccinelle09

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

Typage fort [Pie02]

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

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

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

Deuxième partie

Typage statique de langages impératifs

CHAPITRE 5

UN PREMIER SYSTÈME DE TYPES

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

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

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

La traduction depuis C sera explicitée dans le chapitre 8.

5.1 Syntaxe

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

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

⟨*programme*⟩ ::= (fonctions , globales , ⟨*bloc*⟩)

⟨*expr*⟩ ::= ⟨*lval*⟩
| unop ⟨*expr*⟩
| ⟨*expr*⟩ binop ⟨*expr*⟩
| cst
| & ⟨*lval*⟩
| & fonction

⟨*lval*⟩ ::= var
| ⟨*lval*⟩ . champ
| ⟨*lval*⟩ [⟨*expr*⟩]
| * ⟨*expr*⟩

⟨*bloc*⟩ ::= ⟨*instr*⟩ ; ⟨*bloc*⟩
| ϵ

⟨*instr*⟩ ::= ⟨*lval*⟩ ← ⟨*expr*⟩
| ⟨*lval*⟩ ← funexp (args)
| funexp (args)
| ↑ nom { ⟨*bloc*⟩ }
| if (⟨*expr*⟩) { ⟨*bloc*⟩ } else { ⟨*bloc*⟩ }
| { ⟨*bloc*⟩ } label:
| goto label
| forever { ⟨*bloc*⟩ }
| return ⟨*expr*⟩

Contrairement au langage C, le langage des expressions et des instructions est séparé. Par conséquence, l'évaluation des expressions peut se faire sans effet de bord.

expliquer pourquoi un while expr ne suffit pas

5.2 Sémantique (opérationnelle, à petits pas)

On définit une sémantique opérationnelle à petits pas, sous forme d'une relation de transition (notée $\rightarrow$) entre états de l'interpréteur.

Ces états sont constitués des composantes suivantes :

- un point de contrôle l dans le programme. Ils sont issus d'une première transformation en un graphe de flot de contrôle.
- un état mémoire σ qui associe à chaque variable une valeur.

Graphe de flot de contrôle

Dans la syntaxe ci-dessus, on peut classifier les instructions en deux familles : celles qui définissent le flot de contrôle (if, dowith, goto, forever) et celles qui définissent le flot de données. Une première transformation va transformer chaque fonction en son graphe de flot de contrôle, défini comme suit :

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

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

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

xxx

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

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

État mémoire

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

Ces valeurs peuvent être de plusieurs formes :

$v ::= n, f, \text{NIL}$	constantes
$\mid \&a$	pointeur vers l'adresse a
$\mid \&f$	pointeur vers la fonction f
$\mid \top$	valeur non initialisée

Pile d'appels

Left values

La mémoire est organisée en adresses, mais pourtant dans le programme cette notion n'est pas directement visible. Les accès sont réalisés à travers des "left values".

Jugements

Les jugements ont les formes suivantes :

- $\sigma \vdash lv \Rightarrow a$: la left-value lv correspond à l'adresse mémoire a.
- $\sigma \vdash e \Rightarrow v$: l'expression e s'évalue en v.
- $(l, \sigma) \rightarrow (l', \sigma')$: permet de définir la fonction de transition principale

Sémantique des left-values

$$\frac{(v,a) \in \sigma}{\sigma \vdash v \Rightarrow a}\ (\text{EVAL-LV-VAR}) \qquad \frac{\sigma \vdash e \Rightarrow \&a}{\sigma \vdash *e \Rightarrow a}\ (\text{EVAL-LV-DEREF}) \qquad \frac{\sigma \vdash lv \Rightarrow a}{\sigma \vdash lv.f \Rightarrow a+f}\ (\text{EVAL-LV-FIELD})$$

$$\frac{\sigma \vdash lv \Rightarrow a \qquad \sigma \vdash e \Rightarrow n}{\sigma \vdash lv[e] \Rightarrow a+n}\ (\text{EVAL-LV-ARRAY})$$

Sémantique des expressions

$$\frac{}{\sigma \vdash c \Rightarrow c}\ (\text{EVAL-CST}) \qquad \frac{\sigma \vdash lv \Rightarrow a \qquad (a,v) \in \sigma}{\sigma \vdash lv \Rightarrow v}\ (\text{EVAL-LV}) \qquad \frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash \text{op}\ e \Rightarrow \widehat{\text{op}}\ v}\ (\text{EVAL-UNOP})$$

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1\ \text{op}\ e_2 \Rightarrow v_1 \widehat{\text{op}}\ v_2}\ (\text{EVAL-BINOP}) \qquad \frac{\sigma \vdash lv \Rightarrow a}{\sigma \vdash \&lv \Rightarrow \&a}\ (\text{EVAL-ADDROF})$$

$$\frac{}{\sigma \vdash \&f \Rightarrow \&f}\ (\text{EVAL-ADDROFFUN})$$

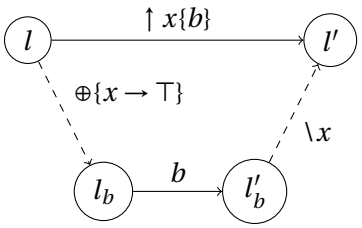
Sémantique des instructions

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

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

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

$$\frac{\begin{array}{cc}\langle l, \uparrow x\{b\}, l' \rangle & \langle l_b, b, l'_b \rangle \\ \sigma' = \sigma \oplus \{x \rightarrow \top\} & (l_b, \sigma', s) \rightarrow (l'_b, \sigma'')\end{array}}{(l,\sigma) \rightarrow (l', \sigma'' \backslash x)} \text{ (INSTR-DECL)}$$



call

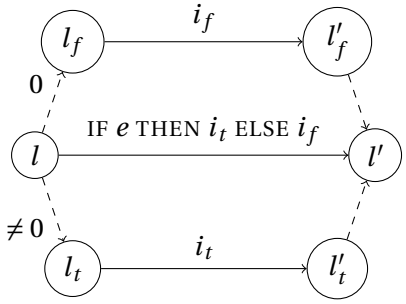
TODO pour :

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

lefinir $\sigma \vdash fe \Rightarrow f$

Sémantique des conditions

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



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

$$\frac{\begin{array}{c}\langle l, \text{IF } e \text{ THEN } i_t \text{ ELSE } i_f, l' \rangle \\ \sigma \vdash e \Rightarrow 0 \\ \langle l_i, i_f, l'_i \rangle \qquad (l_i, \sigma) \rightarrow (l'_i, \sigma')\end{array}}{(l,\sigma) \rightarrow (l', \sigma')} \text{ (IF-FALSE)}$$

5.3 Règles de typage

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

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

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

Les types des valeurs sont :

$$\begin{array}{lr}
\tau ::= \text{INT}, \text{FLOAT}, \text{VOID} & \text{constantes} \\
\quad | \ a & \text{variable} \\
\quad | \ (\tau_1, \ldots, \tau_n) \rightarrow \tau_r & \text{fonction} \\
\quad | \ [\tau] & \text{tableau} \\
\quad | \ \tau * & \text{pointeur} \\
\quad | \ \{f_1 : \tau_1, \ldots, f_n : \tau_n\} & \text{structure}
\end{array}$$

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.

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

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

Programme

Au niveau global, un programme P est bien typé (noté $\vdash P$) s'il existe un environnement Γ^0 permettant de typer ses composantes (les fonctions, les globales et le bloc d'initialisation).

$$\frac{\Gamma^0 = (\vec{\sigma}, \vec{\tau}) \qquad \Gamma^0 \vdash \vec{f} : \vec{\sigma} \qquad \Gamma^0 \vdash \vec{x} : \vec{t} \qquad \Gamma^0 \vdash b}{\vdash (\vec{f}, \vec{x}, b)} \ (\text{PROG})$$

Flot de contrôle

Les règles suivantes permettent de définir les jugements $\Gamma \vdash i$. En résumé, les instructions sont bien typées si leurs sous-instructions sont bien typées. La seule règle supplémentaire concerne la condition du if qui doit être typée en INT.

$$\frac{}{\Gamma \vdash \epsilon} \ (\text{PASS}) \qquad \frac{\Gamma \vdash s \qquad \Gamma \vdash b}{\Gamma \vdash s; b} \ (\text{SEQ}) \qquad \frac{\Gamma \vdash b}{\Gamma \vdash forever\{b\}} \ (\text{FOREVER})$$

$$\frac{\Gamma \vdash e : \text{INT} \qquad \Gamma \vdash i_t \qquad \Gamma \vdash i_f}{\Gamma \vdash \text{IF } e \text{ THEN } i_t \text{ ELSE } i_f} \ (\text{IF}) \qquad \frac{}{\Gamma \vdash gotol} \ (\text{GOTO}) \qquad \frac{\Gamma \vdash b}{\Gamma \vdash \{b\}l:} \ (\text{DOWITH})$$

Left values

On étend la relation de typage aux left values : chaque left value, vue comme une expression, peut être typée.

$$\frac{(v,\tau) \in \Gamma}{\Gamma \vdash v : \tau}\ (\textsc{Lv-Var}) \qquad \frac{\Gamma \vdash lv : \tau_s \qquad (f,\tau_f) \in \tau_s}{\Gamma \vdash lv.f : \tau_f}\ (\textsc{Lv-Field}) \qquad \frac{\Gamma \vdash e : \tau*}{\Gamma \vdash *e : \tau}\ (\textsc{Lv-Deref})$$

$$\frac{\Gamma \vdash lv : [\tau] \qquad \Gamma \vdash e : \textsc{Int}}{\Gamma \vdash lv[e] : \tau}\ (\textsc{Lv-Array})$$

Expressions

$$\frac{}{\Gamma \vdash n : \textsc{Int}}\ (\textsc{Const-Int}) \qquad \frac{}{\Gamma \vdash f : \textsc{Float}}\ (\textsc{Const-Float}) \qquad \frac{}{\Gamma \vdash \textsc{Nil} : \tau*}\ (\textsc{Const-Nil})$$

$$\frac{op \in \{+,-,\ddot{O},/,\&,|,\&\&,||,\lll,\ggg\} \qquad \Gamma \vdash e_1 : \textsc{Int} \qquad \Gamma \vdash e_2 : \textsc{Int}}{\Gamma \vdash e_1 op e_2 : \textsc{Int}}\ (\textsc{Op-Int})$$

$$\frac{op \in \{+.,-.,\ddot{O}.,/.\} \qquad \Gamma \vdash e_1 : \textsc{Float} \qquad \Gamma \vdash e_2 : \textsc{Float}}{\Gamma \vdash e_1 \text{op } e_2 : \textsc{Float}}\ (\textsc{Op-Float})$$

$$\frac{op \in \{=,\neq,\leq,\geq,<,>\} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\textsc{Int},\textsc{Float}\}}{\Gamma \vdash e_1 \text{op} e_2 : \textsc{Int}}\ (\textsc{Op-Cmp})$$

$$\frac{\tau \in \{\textsc{Int},\textsc{Float}\} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash -e : \tau}\ (\textsc{Unop-Minus}) \qquad \frac{op \in \{\text{¬},!\} \Gamma \vdash e : \textsc{Int}}{\Gamma \vdash ope : \textsc{Int}}\ (\textsc{Unop-Not})$$

$$\frac{\Gamma \vdash lv : \tau}{\Gamma \vdash \&lv : \tau*}\ (\textsc{AddrOf}) \qquad \frac{\Gamma \vdash f : \sigma \qquad \tau \leq \sigma}{\Gamma \vdash \&lv : \tau}\ (\textsc{AddrOfFun})$$

Fonctions

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

$$\frac{\Gamma' = \Gamma \oplus \{args(f) = \vec{\tau}\} \qquad \Gamma' \vdash body(f) \qquad \Gamma' \vdash !ret : \tau_r}{\Gamma \vdash f : Gen(\vec{\tau} \to \tau_r, \Gamma)}\ (\textsc{Fun})$$

Instructions

$$\frac{\Gamma \oplus \{x:\tau\} \vdash b}{\Gamma \vdash \uparrow x\{b\}}\ (\textsc{Decl}) \qquad \frac{\Gamma \vdash lv:\tau \qquad \Gamma \vdash e:\tau}{\Gamma \vdash lv \leftarrow e}\ (\textsc{Assign})$$

$$\frac{\Gamma \vdash lv:\tau_{ret} \qquad \Gamma \vdash fe:\sigma \qquad \Gamma \vdash \vec{e}:\vec{\tau} \qquad (\vec{\tau} \to \tau_r) \leq \sigma}{\Gamma \vdash lv \leftarrow fe(\vec{e})}\ (\textsc{Fcall})$$

5.4 Limitations

Programmes non typables

Incohérences

CHAPITRE 6

SYSTÈME AVEC LE QUALIFICATEUR "USER"

6.1 Éditions et ajouts

6.2 Passage sur le cas d'étude

6.3 Propriété d'isolation mémoire

Le déréférencement d'un pointeur dont la valeur est contrôlée par l'utilisateur ne peut se faire qu'à travers une fonction qui vérifie la sûreté de celui-ci.

CHAPITRE 7

SYSTÈME AVEC LE QUALIFICATEUR "SZ"

7.1 But

7.2 Annotation de `string.h`

Troisième partie

Expérimentation

CHAPITRE 8

IMPLANTATION

8.1 Langages intermédiaires

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

Lors de l'écriture d'un compilateur, on a besoin d'un langage intermédiaire qui fasse l'intermédiaire entre *frontend* et *backend*. Avec ce langage, on doit pouvoir (facilement) réaliser les types d'opérations suivantes :

- compiler ce langage vers un langage machine (interface avec le *backend*).
- exprimer des transformations intermédiaires sur cette représentation (analyses sémantiques, optimisations, etc).

À cause du premier point, un langage comme C est très séduisant, mais malheureusement il a de nombreux défauts qui font qu'il ne remplit pas le deuxième. Par exemple, certaines instructions sont ambiguës, et l'ensemble des opération n'est pas orthogonal (on peut faire la même chose de plusieurs manières).

Pour pallier ce problème, une première solution peut être d'ajouter des contraintes au C intermédiaire manipulé pour en obtenir un sous-langage. De nombreux sous-ensembles ont été définis pour aller dans ce sens.

Critères

Les critères à évaluer sont les suivants :

Forme (textuel / langage d'implem) Quelle est la représentation du langage ? Est-ce une bibliothèque (si oui, dans quel langage) a-t'il une forme textuelle (si oui, pour quels langages des *parsers* existent ils).

Maturité

Tools

Support Le langage a-t'il "fait ses preuves" ? Est-il susceptible de changer ?

Scope (analyse / compil) Pour quel type d'utilisation a-t'il été conçu ?

Expressivité Peut-on compiler "tout C" ?

Orthogonalité

Typage

Langages

LLVM LLVM[LA04] est un backend de compilateur développé par Apple.

Cmm (Haskell)

Cmm (OCaml)

CIL

Clight Utilisé dans Compcert (front-end)
[BDL06].

Cminor Utilisé dans Compcert (middle-end).

Cminusminus [PJNO97]
http ://www.cminusminus.org/

Newspeak [HL08]

8.2 Newspeak

8.3 Chaîne de compilation

La compilation vers C est faite en trois étapes (figure 8.1) : prétraitement du code source, compilation de C prétraité vers NEWSPEAK, puis compilation de NEWSPEAK vers ce langage.

Prétraitement

C2NEWSPEAK travaillant uniquement sur du code prétraité (dans directives de préprocesseur), la première étape consiste donc à faire passer le code par CPP.

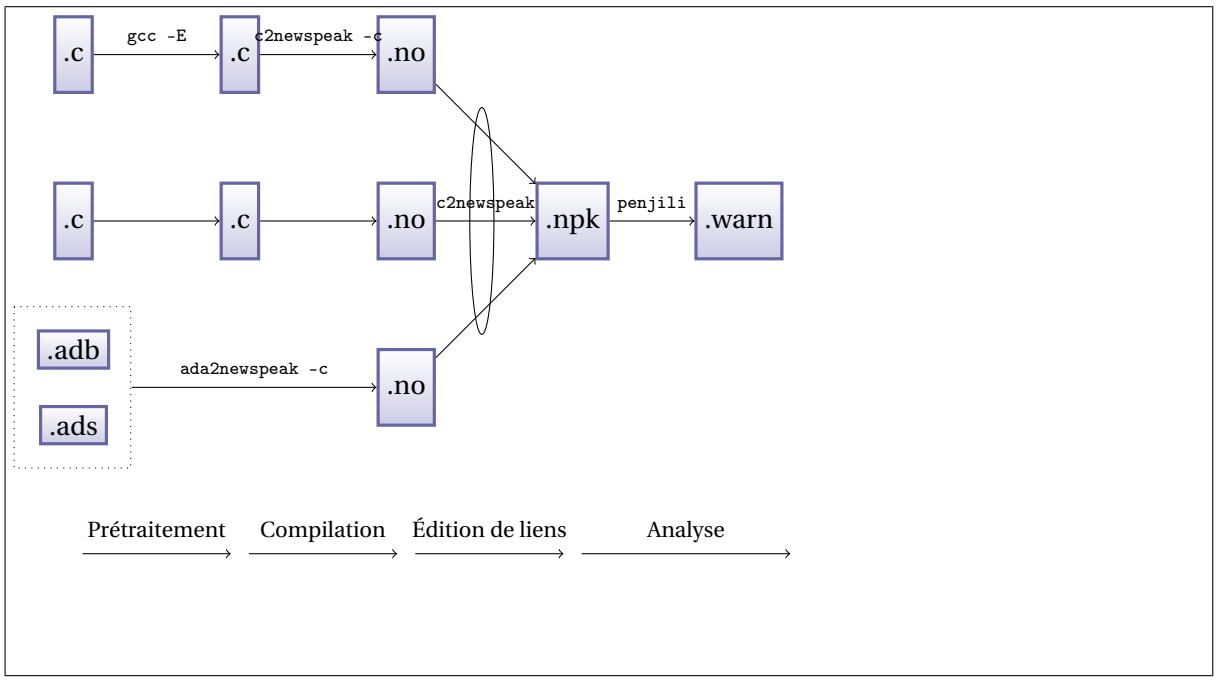


FIGURE 8.1 – Compilation depuis Newspeak

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 flôt 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].

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;
```

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

$\Gamma ::= \epsilon$	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 8.2 – Lambda calcul simplement typé avec entiers, flottants et couples

```
    int y[10];
    x.b = 1;
    y[1] = 1;
    return 0;
}
```

Implantation de l'algorithme de typage

Commençons par étudier le cas du lambda-calcul simplement typé (figure 8.2). Prenons l'exemple de la fonction suivante [1] :

1. On suppose que `plus` est une fonction de l'environnement global qui a pour type $\textsc{Int} \to \textsc{Int} \to \textsc{Int}$.

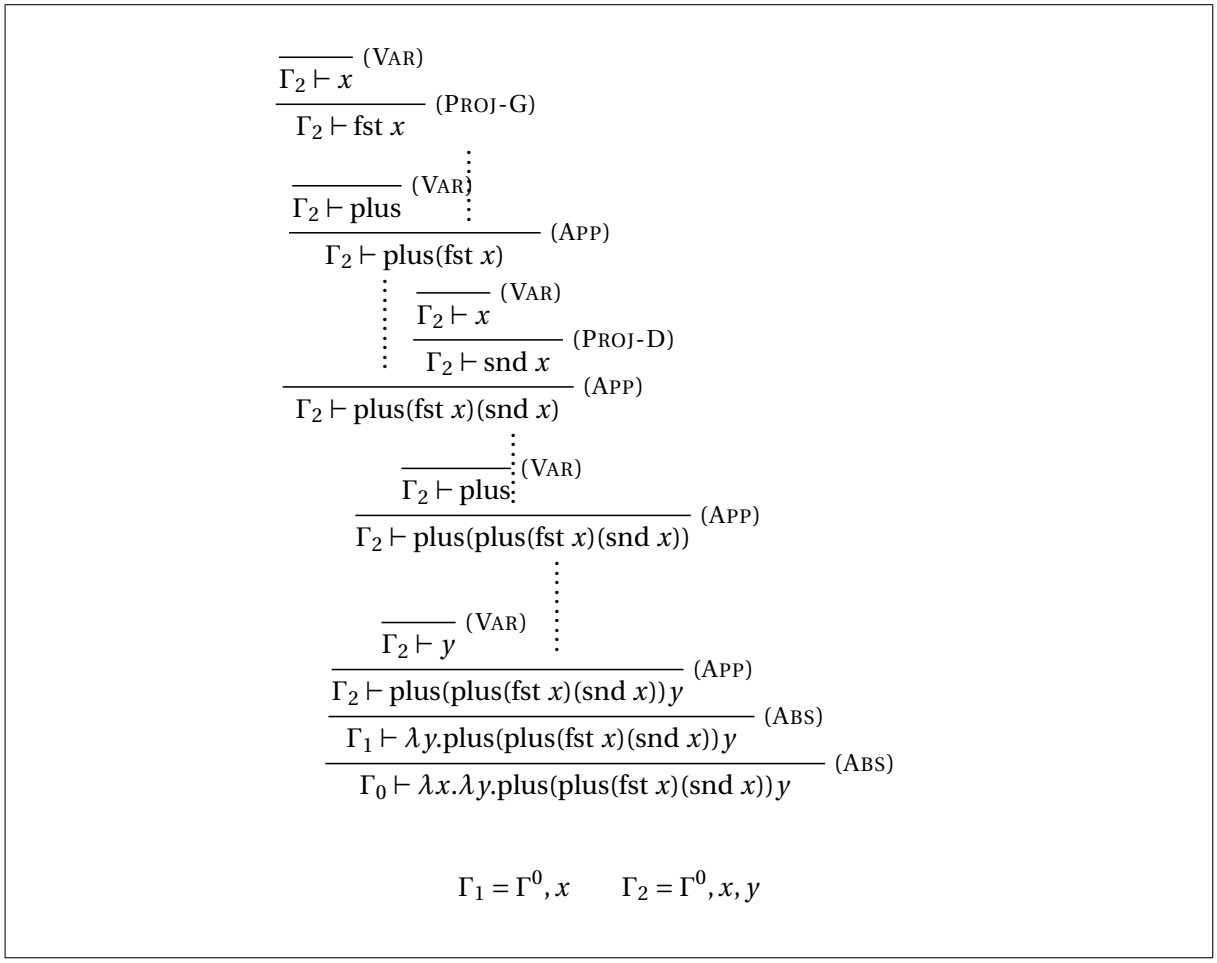


FIGURE 8.3 – Arbre d'inférence : règles à utiliser

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

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

Mais comment faire pour implanter cette analyse ? En fait le système de types de la figure 8.2 a une propriété particulièrement intéressante : chaque forme syntaxique (variable, abstraction, etc) est en conclusion d'exactement une règle de typage. Cela permet de toujours savoir quelle règle il faut appliquer (c'est à rapprocher du fait qu'on peut déduire un analyseur syntaxique d'une grammaire LL)

Partant du terme de conclusion (f), on peut donc en déduire un squelette d'arbre d'inférence (figure 8.3) [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.

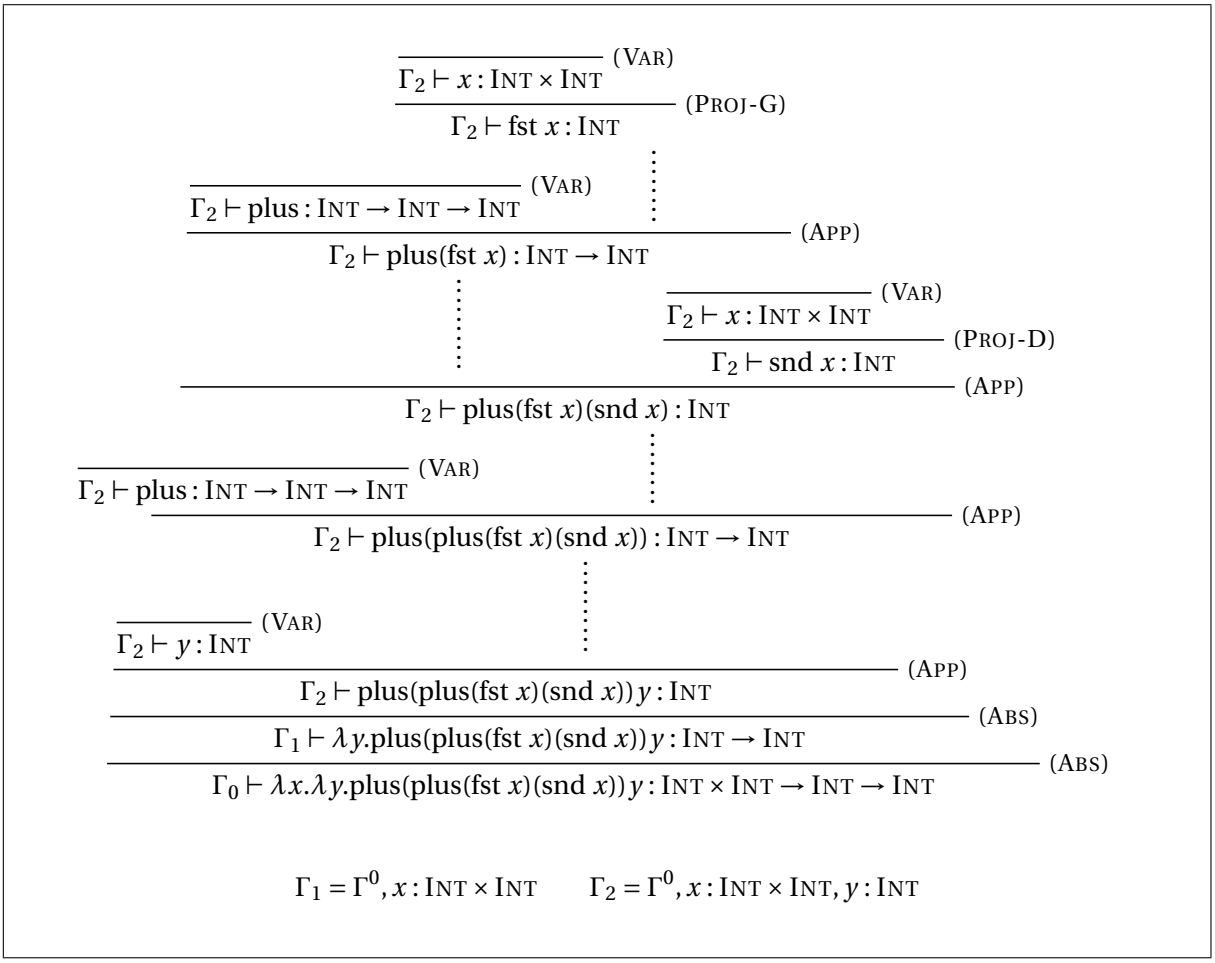


FIGURE 8.4 – Arbre d'inférence complet

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

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

Ici = 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.

ccurs check peut tre ?

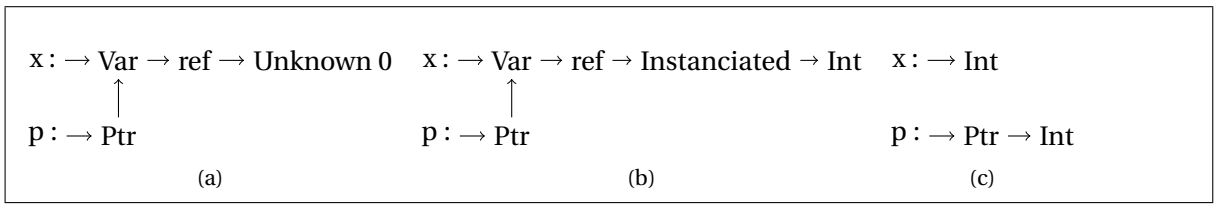


FIGURE 8.5 – Unification par partage

Une fois toutes les substitutions effectuées, on obtient un arbre de typage correct (figure 8.4, donc un programme totalement inféré.

L'implémentation de cet algorithme utilise le partage est les références (figure 8.5).

D'abord 8.5a, ensuite 8.5b, et enfin 8.5c.

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

and const_type =
  | Int_type
  | Float_type

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

Le programme C suivant :

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

est compilé ainsi en Tyspeak :

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

CHAPITRE 9

ÉTUDE DE CAS : UN DRIVER GRAPHIQUE

9.1 Description du problème

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

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

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

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

Si au contraire un tel pointeur est déréférencé sans vérification (avec `*` ou une fonction comme `memcpy`), le code s'exécutera correctement mais en rendant le système vulnérable, comme le montre la figure 9.1.

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

```
uint32_t *value_ptr;
uint32_t value;
struct drm_crtc *crtc;
int i, found;

info = data;
value_ptr = (uint32_t *)((unsigned long)info->value);
value = *value_ptr;
switch (info->request) {
case RADEON_INFO_DEVICE_ID:
        value = dev->pci_device;
        break;
case RADEON_INFO_NUM_GB_PIPES:
        value = rdev->num_gb_pipes;
        break;
case RADEON_INFO_NUM_Z_PIPES:
        value = rdev->num_z_pipes;
        break;
case RADEON_INFO_ACCEL_WORKING:
        /* xf86-video-ati 6.13.0 relies on this being false for evergreen */
        if ((rdev->family >= CHIP_CEDAR) && (rdev->family <= CHIP_HEMLOCK))
                value = false;
        else
                value = rdev->accel_working;
        break;
case RADEON_INFO_CRTC_FROM_ID:
        for (i = 0, found = 0; i < rdev->num_crtc; i++) {
                crtc = (struct drm_crtc *)minfo->crtcs[i];
                if (crtc && crtc->base.id == value) {
                        struct radeon_crtc *radeon_crtc = to_radeon_crtc(crtc);
                        value = radeon_crtc->crtc_id;
                        found = 1;
                        break;
                }
        }
        if (!found) {
                DRM_DEBUG_KMS("unknown crtc id %d\n", value);
                return -EINVAL;
        }
        break;
case RADEON_INFO_ACCEL_WORKING2:
        value = rdev->accel_working;
        break;
case RADEON_INFO_TILING_CONFIG:
        if (rdev->family >= CHIP_CEDAR)
                value = rdev->config.evergreen.tile_config;
        else if (rdev->family >= CHIP_RV770)
                value = rdev->config.rv770.tile_config;
```

```
                else if (rdev->family >= CHIP_R600)
                        value = rdev->config.r600.tile_config;
                else {
                        DRM_DEBUG_KMS("tiling config is r6xx+ only!\n");
                        return -EINVAL;
                }
        case RADEON_INFO_WANT_HYPERZ:
                mutex_lock(&dev->struct_mutex);
                if (rdev->hyperz_filp)
                        value = 0;
                else {
                        rdev->hyperz_filp = filp;
                        value = 1;
                }
                mutex_unlock(&dev->struct_mutex);
                break;
        default:
                DRM_DEBUG_KMS("Invalid request %d\n", info->request);
                return -EINVAL;
        }
        if (DRM_COPY_TO_USER(value_ptr, &value, sizeof(uint32_t))) {
                DRM_ERROR("copy_to_user\n");
                return -EFAULT;
        }
        return 0;
}

/* from drivers/gpu/drm/radeon/radeon_kms.c */
struct drm_ioctl_desc radeon_ioctls_kms[] = {
        /* KMS */
        DRM_IOCTL_DEF(DRM_RADEON_INFO, radeon_info_ioctl, DRM_AUTH|DRM_UNLOCKED)
};

/* from drivers/gpu/drm/radeon/radeon_drv.c */

static struct drm_driver kms_driver = {
        .driver_features =
            DRIVER_USE_AGP | DRIVER_USE_MTRR | DRIVER_PCI_DMA | DRIVER_SG |
            DRIVER_HAVE_IRQ | DRIVER_HAVE_DMA | DRIVER_IRQ_SHARED | DRIVER_GEM,
        .dev_priv_size = 0,
        .ioctls = radeon_ioctls_kms,
        .name = "radeon",
        .desc = "ATI Radeon",
        .date = "20080528",
        .major = 2,
        .minor = 6,
        .patchlevel = 0,
};
```

cf avant

FIGURE 9.1 – Bug freedesktop.org #29340. Le paramètre `data` provient de l'espace utilisateur via un appel système. Un appelant malveillant peut se servir de cette fonction pour lire la mémoire du noyau à travers le message d'erreur.

```
/* from drivers/gpu/drm/drm_dru.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);
}
```

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

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

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

9.2 Principes de l'analyse

Le problème est modélisé de la façon suivante : on associe à chaque variable `x` un type de données `t`, ce que l'on note `x:t`. En plus des types présents dans le langage C, on ajoute une distinction supplémentaire pour les pointeurs. D'une part, les pointeurs "noyau" (de type `t *`) sont créés en prenant l'adresse d'un objet présent dans le code source. D'autre part, les pointeurs "utilisateurs" (leur type est noté `t user*`) proviennent des interfaces avec l'espace utilisateur.

Il est sûr de déréférencer un pointeur noyau, mais pas un pointeur utilisateur. L'opérateur `*` prend donc un `t *` en entrée et produit un `t`.

Pour faire la vérification de type sur le code du programme, on a besoin de quelques règles. Tout d'abord, les types suivent le flot de données. C'est-à-dire que si on trouve dans le code `a = b`, `a` et `b` doivent avoir un type compatible. Ensuite, le qualificateur `user` est récursif : si on a un pointeur utilisateur sur une structure, tous les champs pointeurs de la structure sont également utilisateur. Enfin, le déréférencement s'applique aux pointeurs noyau seulement : si le code contient l'expression `*x`, alors il existe un type `t` tel que `x:t*` et `*x:t`.

Appliquons ces règles à l'exemple de la figure 9.1 : on suppose que l'interface avec l'espace utilisateur a été correctement annotée. Cela permet de déduire que `data:void user*`. En appliquant la première règle à la ligne 6, on en déduit que `info:struct drm_radeon_info user*` (comme en C, on peut toujours convertir de et vers un pointeur sur `void`).

Pour déduire le type de `value_ptr` dans la ligne 7, c'est la deuxième règle qu'il faut appliquer : le champ `value` de la structure est de type `uint32_t *` mais on y accède à travers un pointeur utilisateur, donc `value_ptr:uint32_t user*`.

À la ligne 8, on peut appliquer la troisième règle : à cause du déréférencement, on en déduit que `value_ptr:t *`, ce qui est une contradiction puisque d'après les lignes précédentes, `value_ptr:uint32_t user*`.

Si la ligne 3 était remplacée par l'appel à `copy_from_user`, il n'y aurait pas d'erreur de typage car cette fonction peut accepter les arguments `(uint32_t *, uint32_t user*, size_t)`.

9.3 Implantation

Une implantation est en cours. Le code source est d'abord prétraité par `gcc -E` puis converti en Newspeak [HL08], un langage destiné à l'analyse statique. Ce traducteur peut prendre en entrée tout le langage C, y compris de nombreuses extensions GNU utilisées dans le noyau. En particulier, l'exemple de la figure 9.1 peut être analysé.

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

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

Le prototype, disponible sur `http://penjili.org`, fera l'objet d'une démonstration.

9.4 Conclusion

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

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 10

CONCLUSION

10.1 Limitations

10.2 Perspectives

TABLE DES FIGURES

BIBLIOGRAPHIE

[BDH+09] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, January 2009. 23

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006. 42

[BP05] Daniel P. Bovet and Marco Cesati Ph. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, third edition edition, November 2005. 10

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77 : Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM. 23

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3) :103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.). 23

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005. 23

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up ? *Formal Methods in System Design*, 35(3) :229–264, 2009. 23

[CMP03] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly, 2003. 15

[CMP10] Dumitru Ceară, Laurent Mounier, and Marie-Laure Potet. Taint dependency sequences : A characterization of insecure execution paths based on input-sensitive cause sequences. In *ICST Workshops*, 2010. 23

[DRS00] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV : Towards a realistic tool for statically detecting all buffer overflows in C, 2000. 23

[EH94] Ana Erosa and Laurie J. Hendren. Taming control flow : A structured approach to eliminating goto statements. In *In Proceedings of 1994 IEEE International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994. 43

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Programming language design and implementation*, PLDI '99, pages 192–203, 1999. 23, 53

[FJKA06] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28 :1035–1087, November 2006. 23

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02 : Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, volume 37, pages 1–12, New York, NY, USA, May 2002. ACM Press. 23

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. 10

[Gra92] Philippe Granger. Improving the results of static analyses programs by local decreasing iteration. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 68–79, London, UK, UK, 1992. Springer-Verlag. 23

[Har88] Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4) :36–38, October 1988. 49

[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. 42, 43, 53

[Int] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 6, 11

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999. 15, 27, 41

[JW04] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004. 23

[KcS07] Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci.*, 174(7) :79–104, 2007. 23

[Ker81] Brian W. Kernighan. Why Pascal is not my favorite programming language. Technical report, AT&T Bell Laboratories, April 1981. 19

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988. 15, 41

[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. 42

[LDG+10] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, documentation and user's manual – release 3.12*. INRIA, August 2010. 15

[LZ06] Peng Li and Steve Zdancewic. Encoding information flow in Haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations (CSFW '06)*, Washington, DC, USA, 2006. IEEE Computer Society. 23

[Mau04] Laurent Mauborgne. ASTRÉE : Verification of absence of run-time error. In René Jacquart, editor, *Building the information Society (18th IFIP World Computer Congress)*, pages 384–392. The International Federation for Information Processing, Kluwer Academic Publishers, Aug 2004. 23

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, December 1978. 19

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. 15

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 1996. 6

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 17, 23

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003. 15

[PJNO97] Simon L. Peyton Jones, Thomas Nordin, and Dino Oliva. C- : A portable assembly language. In Chris Clack, Kevin Hammond, and Antony J. T. Davie, editors, *Implementation of Functional Languages*, volume 1467 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 1997. 42

[PTS+11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux : Ten years later. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, Newport Beach, CA, USA, March 2011. 23

[pyt] Python Programming Language – Official Website. 15

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010. 23

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01 : Proceedings of the 10th conference on USENIX Security Symposium*, page 16, Berkeley, CA, USA, 2001. USENIX Association. 23

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007. 5

[TTL] Linus Torvalds, Josh Triplett, and Christopher Li. Sparse - a semantic parser for C. `https://sparse.wiki.kernel.org/index.php/Main_Page`. 23, 53

[VB04] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded c programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 231–242, New York, NY, USA, 2004. ACM. 23

[Wal00] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2000. 15