frama C

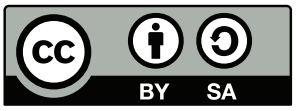
Software Analyzers

PDG — Documentation technique

Documentation du greffon PDG

Calcul de dépendances dans un programme C

Anne Pacalet et Patrick Baudin



CEA LIST, Laboratoire de Sûreté et Sécurité des Logiciels, Saclay, F-91191

Versions

Seules les versions Vx.0 sont des documents achevés ; les autres sont à considérer comme des brouillons.

V3.0 - 2 novembre 2020 :
- Document sous licence Creative Commons “Attribution-ShareAlike 4.0 International” https://creativecommons.org/licenses/by/4.0/.

V2.0 - 3 décembre 2008 :
- réorganisation,
- mise à jour,
- partie sur l’utilisation.

V1.4 - 30 mai 2008 :
- ajout d’une partie sur le marquage.

V1.3 - 22 novembre 2007 :
- mise à jour de la partie sur l’interprocédural.

V1.2 - 07 novembre 2007 :
- mise à jour de la partie sur les dépendances de contrôle.

V1.1 - 20 septembre 2007 :
- réorganisation complète,
- développement de la partie sur les dépendances de contrôle,
- discussion au sujet des boucles infinies.

V1.0 - 26 juin 2007 :
- relecture et petites corrections.

V0.2 - 24 mai 2007 :
- révision générale,

V0.1 - 23 mars 2007 :
- création du module de calcul de PDG,
- extraction de la documentation du PDG du rapport sur le *slicing*.

Table des matières

Chapitre 1

Introduction

1.1 Objectif

L'objectif initial a été de réaliser un module de *slicing* dans le cadre d'un outil généraliste d'analyse de programme (voir [Pacalet(2007)]). Or, la plupart des réductions à effectuer se basent sur l'analyse des dépendances entre les données du programme. En effet, si l'utilisateur demande la sélection d'une instruction `return x;`, il va falloir retrouver ce qui permet de calculer cette valeur de `x` dans les instructions qui précèdent.

Il s'agit donc de calculer le **graphe de dépendances** d'une fonction (appelé **PDG** : *Program Dependence Graph* dans la littérature) c'est-à-dire de représenter finement les liens de dépendances entre les différentes instructions qui la composent. Le résultat ce calcul est un graphe dans lequel les sommets représentent les instructions, éventuellement décomposées en plusieurs noeuds représentant le calcul d'informations élémentaires.

Dans le cadre de l'outil de *slicing*, l'intérêt de ce calcul préalable est de pouvoir travailler en plusieurs passes lors de l'application des requêtes de réduction sans avoir à refaire ce calcul qui peut être lourd (alias, dépendances partielles, etc). En effet, même si l'on souhaite à terme calculer des réductions qui utilisent davantage la sémantique du programme, les réductions à l'aide des dépendances peuvent simplifier le problème.

Par la suite, il a été jugé intéressant de considérer ce calcul comme un module à part entière car il peut avoir d'autres utilités comme étudier la propagation du flot d'information pour des analyses de sécurité, par exemple.

1.2 Spécifications du PDG

Le PDG que l'on souhaite calculer comporte plusieurs types de dépendances :

1.2.1 Dépendances sur la valeur des données

Les **dépendances sur la valeur** d'une donnée sont les plus intuitives.

Exemple 1

```
x = a + b;
```

Ici, `x` dépend de `a` et `b` car la valeur de `x` après cette instruction dépend des valeurs de `a` et `b` avant.

La question se pose néanmoins de définir la granularité à laquelle on s'intéresse aux données. L'utilisation d'autres analyses et structures de données de Frama-C conduit à choisir la même précision (pour plus de détail, voir par exemple l'analyse de valeurs de Frama-C).

1.2.2 Dépendances de calcul d'adresse

Pour les affectations, la valeur qui est écrite en mémoire dépend de la partie droite, mais le choix de l'adresse à laquelle on l'écrit peut également dépendre de variables qui apparaissent dans la partie gauche.

Exemple 2

```
*p = x;
```

Ici, la valeur de la donnée modifiée dépend de `x`, mais le choix de la case dans laquelle on la range dépend de `p`. En effet, si p pointe sur a ou b, c'est soit `a` soit `b` qui va être modifié.

On parle alors de **dépendances sur l'adresse**.

1.2.3 Dépendances de contrôle

Lorsqu'une donnée peut-être modifiée par plusieurs chemins d'exécution, elle dépend des conditions qui permettent de choisir le chemin.

Exemple 3

```
if (c)
  x = a;
L :
```

La valeur de `x` en `L` dépend de `a`, mais aussi de `c`.

Il s'agit d'une **dépendance de contrôle**.

1.2.4 Dépendances sur les déclarations

Lorsque des variables sont utilisées dans une instruction, celle-ci dépend de leurs déclarations, car si on veut la compiler, il faut que les variables existent.

Les déclarations des variables lues (partie droite d'une affectation) sont considérées comme participant au calcul de la valeur. Les déclarations des variables utilisées pour déterminer la case affectée (partie gauche d'une affectation) sont considérées comme des dépendances sur l'adresse.

Exemple 4

```
/* 1 */ int x;
/* 2 */ int y;
...
/* i */ x = 3;
/* j */ y = 4;
...
/* n */ x = y;
```

L'instruction (n) a une dépendance d'adresse sur la déclaration de x (1), et des dépendances de donnée sur l'affectation de y (j) et sa déclaration (2). Dans ce cas, on aurait pu se passer de de cette dernière dépendance car (j) dépend déjà de (2), mais ce n'est pas forcement le cas en présence d'alias.

1.2.5 Résumé

On voit donc qu'on distingue trois types de dépendances :
- les calculs de valeurs,
- les calculs d'adresses,
- le contrôle.

1.3 État de l'art

Voyons tout d'abord ce que dit la littérature sur ce sujet afin de voir les solutions qui peuvent répondre à notre besoin, et les points à modifier.

1.3.1 Origine

Les graphes de dépendances ont principalement été étudiés dans le cadre du *slicing*, mais ils sont aussi utilisés dans les travaux sur la compilation.

1.3.2 Graphes de dépendances

[Ottenstein and Ottenstein(1984)], puis [Ferrante et al.(1987)Ferrante, Ottenstein, and Warren] introduisent la notion de PDG (*Program Dependence Graph*). Un tel graphe est utilisé pour représenter les différentes dépendances entre les instructions d'un programme. Ils l'exploitent pour calculer les instructions qui influencent la valeur des variables en un point.

Cette représentation, initialement intraprocédurale, a été étendue à l'analyse interprocédurale dans [Horwitz et al.(1988)Horwitz, Reps, and Binkley] où elle porte le nom de SDG (*System Dependance Graph*). Elle est maintenant, à quelques variantes près, quasiment universellement utilisée.

1.3.3 Exploitation du graphe

Lorsque l'on utilise le graphe de dépendance pour faire du *slicing*, le calcul se résume à un problème d'accessibilité à un noeud car comme le dit Susan Horwitz :

Definition (slicing selon [Horwitz et al.(1988)Horwitz, Reps, and Binkley])

A slice of a program with respect to program point p and variable x consists of a set of statements of the program that might affect the value of x at p.

c'est-à-dire qu'il faut garder toutes les instructions correspondant à des noeuds du graphe pour lesquels il existe un chemin vers le noeud représentant le calcul de `x` en `p`. Mais le problème est que ce noeud n'existe que si `x` est défini par l'instruction située en `p`.

Nous verrons que cette limitation peut être levée si l'on garde (ou recalcule) les structures de données utilisées lors de la construction du graphe.

Le traitement des appels de fonction est souvent compliqué par le fait qu'il est également ramené à un problème d'accessibilité dans un graphe qui, cette fois, représente toute l'application. Or, dans un tel graphe, si on ne prend pas de précautions supplémentaires, on parcourt des chemins impossibles qui entrent dans une fonction par un appel, et sortent par un autre site d'appel. Ces chemins existent en effet dans le graphe, mais pas dans la réalité.

Comme nous nous proposons de traiter ce problème de manière modulaire, nous devrions échapper à une partie de ce problème. Mais nous verrons par la suite que l'utilisation d'une analyse d'alias globale produit néanmoins quelques effets de bord indésirables.

1.3.4 Programmes non structurés

Les premiers algorithmes utilisés fonctionnent correctement sur des programmes structurés, mais produisent des résultats erronés en présence de `goto`.

Le problème vient du fait que ces instructions ne modifient pas de données : il n'y a donc pas de dépendance de donnée ; et il n'y a pas de dépendance de contrôle non plus car dans le CFG, une seule branche sort du noeud pour aller vers le point de branchement.

Plusieurs personnes ([Choi and Ferrante(1994)], [Agrawal(1994)], [Harman and Danicic(1998)], [Kumar and Horwitz(2002)] entre autres) se sont donc intéressées aux sauts (`goto`) qui brisaient la structure du programme. Ce point, qui nous intéresse tout particulièrement, est présenté en détail en §3.

1.3.5 Pointeurs et données structurées

De nombreux articles s'intéressent aux traitements des données structurées, et plus encore des pointeurs. Dans le cadre de cette étude, nous n'avons pas exploré ces recherches étant donné que nous nous appuyons déjà sur une analyse d'alias précise.

1.4 Plan

Les trois chapitres suivants exposent comment est calculé notre graphe de dépendances.

Puis, nous verrons au chapitre 5 comment exploiter les informations calculées et au chapitre 6 comment associer des informations aux éléments et les propager dans le graphe.

Chapitre 2

Dépendances liées aux données

Le calcul de dépendances de données et d'adresse consiste principalement à retrouver les éléments de flot correspondant aux données utilisées dans les expressions. Mais comme les données peuvent être incluses les unes dans les autres, il ne suffit pas de retrouver les éléments de flot qui calculent exactement ces données, mais aussi ceux qui ont une intersection possible.

Par exemple, dans la séquence :

```
d = d0; x = d.a;
```

il faut être capable de voir que `x` dépend de `d0`.

Autre exemple : dans la séquence :

```
d0.a = a; d0.b = b; d = d0;
```

il faut voir que `d` dépend éventuellement de la valeur initiale de `d0` (si `d0` contient d'autres champs que `.a` et `.b`), mais aussi de `a`, et de `b`.

2.1 Recherche arrière

Le premier calcul mis en œuvre procédait par recherche en arrière des éléments de la table ayant une intersection avec les données présentes en partie droite de l'instruction. Mais cette recherche était compliquée en cas de dépendances partielles comme dans le second exemple ci-dessus. Cette solution a donc été abandonnée.

2.2 Propagation avant d'un état

La méthode finalement choisie pour calculer ces dépendances consiste à propager en avant, par une analyse du type flot de données, un **état des données** qui contient pour chaque donnée, une liste de liens vers les éléments du graphe qui ont permis de déterminer sa valeur en ce point.

Cet état doit avoir les propriétés suivantes :

- on veut pouvoir associer un nouveau noeud à une donnée en précisant s'il faut faire l'union avec l'ensemble précédemment stocké. Par exemple ;
  - pour l'instruction `x = 3;` on construit un nouvel élément dans le PDG, et on mémorise dans l'état que `x` est maintenant associé à cet élément. L'ancienne association est perdue.

  - pour l'instruction `*p = 3;` si `p` peut pointer sur `x`, il faut mémoriser que `x` peut être défini par l'élément du PDG correspondant, mais comme ce n'est pas sûr, il faut faire l'union avec ce qui était précédemment stocké pour `x`.
- lorsque l'on demande l'ensemble associé à une donnée, le résultat doit contenir au moins ce qu'on a stocké (il peut contenir plus d'élément en cas de perte de précision),
- la consultation ne doit pas modifier l'état,
- il faut savoir faire l'union de deux états.

La structure de donnée du module `Lmap` de FRAMA-C correspond à ces critères, et peut donc être utilisée pour ce calcul.

2.3 Propagation arrière d'un état

Une autre solution aurait pu être de propager en arrière un état contenant les utilisations de variables, et mettre à jour le graphe en rencontrant la définition. Le coût de ce calcul semble être le même que le précèdent, mais la propagation avant nous permet d'avoir, à chaque point de programme, un état donnant la correspondance entre une donnée et les éléments du graphe, même si cette donnée n'est pas utilisée à ce point. Cette information nous permet par la suite de définir des critères de *slicing* moins restrictifs.

2.4 Traitement de l'affectation

Le principe de l'algorithme du traitement d'une affectation `lval = exp;` est donc le suivant :

- recherche des données $\{d_v\}$ utilisées dans `exp` à l'aide des résultats de l'analyse d'alias préalable,
- calcul de *dpdv*, c'est-à-dire l'union des ensembles associées à ces données d_v dans l'état,
- recherche des données $\{d_a\}$ utilisées pour calculer l'adresse de *lval*,
- calcul de *dpda* c'est-à-dire l'union des ensembles associées à ces données d_a dans l'état,
- recherche de l'élément e correspondant à cette instruction dans le graphe, et création de cet élément s'il n'existe pas,
- ajout des dépendances *dpdv* et *dpda* à e,
- recherche des données $\{d_x\}$ potentiellement modifiées par cette affectation,
- calcul du nouvel état (après l'instruction) en ajoutant dans l'ancien état un lien entre les $\{d_x\}$ et e.

2.5 Déclarations

Les déclarations de variable doivent être traitées séparément des valeurs, car on peut parfois dépendre de l'adresse d'une variable sans dépendre de ce qu'elle contient.

C'est par exemple le cas lorsque la variable apparaît à gauche d'une affectation (`x = 3;`) ou encore quand on n'utilise que son adresse (`p = &x;`).

On garde donc une table qui permet de retrouver les éléments du graphe de dépendances qui correspondent aux déclarations.

2.6 Calcul de conditions

Les noeuds du graphe de dépendances représentant les calculs de condition des `if` ou `switch` ont des dépendances de donnée sur les données utilisées.

2.7 Dépendances de donnée dans les boucles

Pour les boucles *explicites*, il suffirait d'effectuer deux tours de la boucle pour obtenir toutes les dépendances de donnée car le premier capture les dépendances avec les données provenant d'avant la boucle, ou interne à un tour, et le second capture les dépendances entre un tour et le suivant.

Mais comme les boucles peuvent être introduites par la présence de sauts quelconques, le plus simple est dans un premier temps d'itérer jusqu'à obtenir un point fixe sur l'état des données. Il faut noter que les noeuds ne sont créés que lors de la première itération, les suivantes n'ajoutant que de nouvelles dépendances entre ces noeuds. Le nombre de noeuds étant fini (de l'ordre de grandeur du nombre d'instruction de la fonction), l'atteignabilité du point fixe est garantie.

2.8 Appels de fonction

Le traitement des appels de fonction est présenté dans le chapitre 4

Chapitre 3

Dépendances de contrôle

3.1 Introduction

Intuitivement, un noeud $\mathfrak{n}$ du PDG a une dépendance de contrôle sur un noeud $\mathfrak{c}$ si le fait d'exécuter $\mathfrak{n}$ dépend du résultat de l'exécution de $\mathfrak{c}$. Typiquement, $\mathfrak{c}$ est un noeud qui a plusieurs successeurs, comme un IF par exemple, et en fonction de la branche qui est choisie, $\mathfrak{n}$ est exécuté ou non.

Nous allons voir qu'il existe de nombreuses façons de calculer ces dépendances de contrôle, mais que nous avons du les adapter car elle ne correspondent pas exactement à ce que l'on souhaitait faire. Le principal problème est que nous nous proposons d'analyser correctement toute fonction, même en présence de sauts quelconques, voire de boucles infinies ; ce qui, comme nous allons le voir, pose des problèmes particuliers au niveau des dépendances de contrôle.

3.2 Etat de l'art

Commençons tout d'abord par rappeler quelques définitions et rapporter les résultats que l'on trouve dans la littérature.

3.2.1 CFG

Le **graphe de flot de contrôle** est un graphe orienté qui définit l'ordre d'exécution des instructions. Un noeud $\mathfrak{a}$ est connecté à un noeud $\mathfrak{b}$ si l'instruction $\mathfrak{b}$ peut suivre immédiatement l'instruction $\mathfrak{a}$ dans une trace l'exécution. On dit que $\mathfrak{b}$ est un **successeur** de $\mathfrak{a}$. On représente l'ensemble des successeurs d'un noeud $\mathfrak{a}$ par $Succ(\mathfrak{a})$. On dit aussi que $\mathfrak{a}$ est un **prédécesseur** de $\mathfrak{b}$.

Un noeud est considéré comme une entrée dans le CFG s'il n'a pas de prédécesseur. Il est généralement considéré qu'il y a un unique noeud d'entrée, et que tous les noeuds du CFG sont atteignables depuis ce point d'entrée. Cette hypothèse semble raisonnable car on s'intéresse au CFG d'une fonction qui a bien un seul point d'entrée, et les instructions non atteignables depuis le point d'entrée sont du code mort que l'on peut donc ignorer dans les analyses.

Un noeud est considéré comme une sortie du CFG s'il n'a pas de successeur. Son unicité et son accessibilité sont discutées plus loin.

3.2.2 Postdominateurs

La plupart des algorithmes de calcul des dépendances de contrôle se basent sur un CFG dans lequel sont ajoutés deux noeuds spéciaux START et STOP (on notera $\mathfrak{E}$ ce dernier), et sur la notion de **postdominateur** dont une définition est la suivante :

Definition (postdominateur)

Une instruction $\mathfrak{a}$ est **postdominée** par une instruction $\mathfrak{b}$ ($\mathfrak{b}$ est un **postdominateur** de $\mathfrak{a}$) si tous les chemins qui vont de $\mathfrak{a}$ au noeud $\mathfrak{E}$ contiennent $\mathfrak{b}$.

En d'autres termes, si on passe par l'instruction $\mathfrak{a}$, on passe forcement par tous ses postdominateurs avant de sortir. Ou encore, toutes les traces partant de $\mathfrak{a}$ et allant à $\mathfrak{E}$ passent par $\mathfrak{b}$.

Certains auteurs définissent également le **premier postdominateur** (appelé aussi **postdominateur immédiat**) de la façon suivante :

Definition (premier postdominateur)

$\mathfrak{b}$ est le premier postdominateur de $\mathfrak{a}$ si et seulement si :
- $\mathfrak{b}$ postdomine $\mathfrak{a}$,
- et $\mathfrak{b}$ est postdominé par tous les autres postdominateurs de $\mathfrak{a}$.

$\mathfrak{b}$ est donc unique.

Cela permet de construire un arbre (appelé PDT pour *Post-Dominator Tree*) représentant cette relation dans lequel les noeuds sont les mêmes que ceux du CFG et le père de chaque noeud est son premier postdominateur.

L'ensemble des postdominateurs d'un noeud $\mathfrak{a}$ est donné par :

$$Pd(\mathfrak{a}) = \{a\} \bigcup \bigcap_{s \in Succ(\mathfrak{a})} Pd(\mathfrak{s})$$

qui traduit le fait que $\mathfrak{b}$ postdomine $\mathfrak{a}$ si et seulement si $\mathfrak{b} = \mathfrak{a}$ ou $\mathfrak{b}$ postdomine tous les successeurs de $\mathfrak{a}$. La méthode de calcul consiste à initialiser tous les ensembles à $\top$, et à itérer jusqu'à stabilisation. La fonction étant décroissante, la convergence est assurée.

La notion classique de postdominateurs suppose que le CFG ait un point unique de sortie $\mathfrak{E}$, et que celui-ci soit atteignable depuis tous les autres points du graphe. Si ce n'est pas le cas, à la fin de ce calcul, pour les $\mathfrak{a}$ n'ayant pas de chemin vers $\mathfrak{E}$, on a : $Pd(\mathfrak{a}) = \top$.

3.2.3 Dépendances de contrôle

Intuitivement, on dit qu'une instruction $\mathfrak{a}$ a une **dépendance de contrôle** sur une instruction $\mathfrak{c}$ si, en fonction du choix que l'on fait en $\mathfrak{c}$, on passe ou non en $\mathfrak{a}$. Cela suppose donc qu'il

y ait un choix à faire en $\mathfrak{c}$, c'est-à-dire que le noeud correspondant dans le CFG ait plusieurs successeurs.

Les dépendances de contrôle sont définies par [Ferrante et al.(1987)Ferrante, Ottenstein, and Warren] de la façon suivante :

Definition (dépendances de contrôle selon [Ferrante et al.(1987)Ferrante, Ottenstein, and Warren])

Pour deux noeuds $\mathfrak{a}$ et $\mathfrak{b}$ du CFG, $\mathfrak{b}$ dépend de $\mathfrak{a}$ ssi :
- il existe un chemin P de $\mathfrak{a}$ a $\mathfrak{b}$ tel que tout noeud Z de P, différent de $\mathfrak{a}$ et de $\mathfrak{b}$, est postdominé par $\mathfrak{b}$,
- et $\mathfrak{a}$ n'est pas postdominé par $\mathfrak{b}$.

Ce qui signifie que :
- plusieurs chemins partent de $\mathfrak{a}$,
- qu'il existe un chemin qui passe par $\mathfrak{b}$,
- et qu'il existe aussi un chemin qui ne passe pas par $\mathfrak{b}$ (sinon, $\mathfrak{a}$ serait postdominé par $\mathfrak{b}$).

Ce qui conduit à une autre définition, équivalente à la précédente :

Definition (dépendance de contrôle)

Une instruction $\mathfrak{b}$ a une **dépendance de contrôle** vis à vis de $\mathfrak{a}$ si :
- $\mathfrak{b}$ postdomine certains successeurs de $\mathfrak{a}$,
- $\mathfrak{b}$ ne postdomine pas tous les successeurs de $\mathfrak{a}$.

Pour calculer le CDG, l'algorithme de référence est le suivant :

Algorithme (calcul du CDG selon [Ferrante et al.(1987)Ferrante, Ottenstein, and Warren])

- soit ACFG le CFG (+ START et STOP) dans lequel sont ajoutés :
  - un noeud ENTRY,
  - une arrête (ENTRY,START),
  - une arrête (ENTRY, STOP),
- soit S l'ensemble des arrêtes ($\mathfrak{a}$,$\mathfrak{b}$) de ACFG telles que $\mathfrak{b}$ ne postdomine pas $\mathfrak{a}$,
  (c'est-à-dire les arrêtes partant des noeuds $\mathfrak{a}$ qui ont plusieurs successeurs)
- soit $\mathfrak{l}$ le plus petit ancêtre commun à $\mathfrak{a}$ et $\mathfrak{b}$ dans PDT
  (on peut montrer que soit $\mathfrak{l}$=$\mathfrak{a}$, soit $\mathfrak{l}$ est le père de $\mathfrak{a}$ dans PDT)
  - si $\mathfrak{l}$ est le père de $\mathfrak{a}$ dans PDT, tous les noeuds du PDT sur le chemin entre $\mathfrak{l}$ et $\mathfrak{b}$ ($\mathfrak{b}$ compris, mais pas $\mathfrak{l}$) dépendent de $\mathfrak{a}$,
  - si $\mathfrak{l} = \mathfrak{a}$, tous les noeuds du PDT sur le chemin entre $\mathfrak{a}$ et $\mathfrak{b}$ ($\mathfrak{a}$ et $\mathfrak{b}$ compris) dépendent de $\mathfrak{a}$.

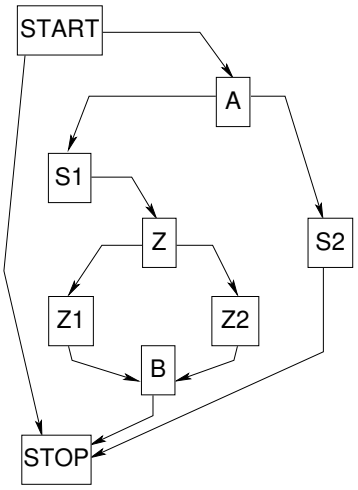
En fait, il est plus simple de dire que tous les noeuds du PDT sur le chemin entre $\mathfrak{b}$ et le père de $\mathfrak{a}$ ($\mathfrak{b}$ compris, mais pas le père de $\mathfrak{a}$) dépendent de $\mathfrak{a}$.

La relation de dépendance étant transitive, on peut choisir de calculer uniquement les dépen-

dances directes ou d'inclure les dépendances indirectes.

Exemple 5

Dans le CFG ci-contre, $\mathfrak{b}$ a bien une dépendance de contrôle sur $\mathfrak{a}$. On remarque que, par contre, Z1 ou Z2 ne dépendent pas directement de $\mathfrak{a}$, car ils ne postdominent pas S2. En revanche, il y a néanmoins une dépendance indirecte, comme on pouvait s'y attendre, car Z dépend de $\mathfrak{a}$, et Z1 et Z2 dépendent de Z.



En fait, $\mathfrak{a}$ dépend directement de c si $\mathfrak{a}$ est forcement atteint si on passe par l'un des successeurs de c, mais il y a des chemins qui partent de c et qui ne passe pas pas $\mathfrak{a}$.

3.2.4 Cas particuliers

Ce qui a été présenté ci-dessus s'applique bien à des programmes bien structurés, mais nécessite des adaptations si on s'intéresse :

— aux instructions qui modifie le flot de contrôle telles que les sauts,
— aux CFG qui contiennent des noeuds pour lesquels il n'y a pas de chemin vers le noeud de sortie.

Nous allons étudier plus précisément ces deux points ci-dessous.

3.3 Nos définitions

3.3.1 Chemins

Dans ce qui suit, on utilise beaucoup la notion de **chemin** : un chemin est une liste de noeuds du CFG telle que si deux noeuds $\mathfrak{a}$ et $\mathfrak{b}$ se suivent dans la liste, $\mathfrak{b}$ est un successeur de $\mathfrak{a}$. Nous appelons **trace** un chemin qui se termine par un noeud n'ayant pas de successeur, ou un chemin qui est infini.

On définit quelques notations pour désigner les chemins :
— $[a, b]$ un chemin allant de $\mathfrak{a}$ à $\mathfrak{b}$
— $[a, b[$ une trace partant de $\mathfrak{a}$ et passant par $\mathfrak{b}$
— $[a, -[$ une trace partant de $\mathfrak{a}$,
— $[a, b, c]$ un chemin allant de $\mathfrak{a}$ à $\mathfrak{c}$ en passant par $\mathfrak{b}$,

— $[a; s, b]$ un chemin allant de $\mathfrak{a}$ à $\mathfrak{b}$ en passant par $\mathfrak{s} \in Succ(\mathfrak{a})$,
— $[a, \neg b[$ une trace partant de $\mathfrak{a}$ et ne passant pas par $\mathfrak{b}$,
et des ensembles de chemins :
— $[\![a, -[$ toutes les traces partant de $\mathfrak{a}$,
— $[\![a, b[$ toutes les traces partant de $\mathfrak{a}$ et passant par $\mathfrak{b}$,
— ...
On dit qu'un noeud appartient à un chemin, et on écrit - un peu abusivement - $\mathfrak{x} \in [a, b]$ si $\mathfrak{x}$ apparaît au moins une fois dans la liste qui décrit le chemin.

3.3.2 Postdominateurs

Avec les notations ci-dessus, on peut définir $Pd(\mathfrak{a})$ l'ensemble des postdominateurs de $\mathfrak{a}$, de la façon suivante :

$$\mathfrak{b} \in Pd(\mathfrak{a}) \Leftrightarrow def \forall t \in [\![a, \mathfrak{E}], \mathfrak{b} \in t$$

On remarque que si on applique la définition ci-dessus à un CFG qui contient des noeuds tels qu'il n'y a pas de chemin vers la sortie, on a :

$$\forall \mathfrak{a}, [\![a, \mathfrak{E}] = \emptyset \Rightarrow \forall \mathfrak{b}, \mathfrak{b} \in Pd(\mathfrak{a})$$

c'est-à-dire que l'on considère que de tels noeuds sont postdominés par tous les autres, ce qui n'est pas très intéressant en terme de trace d'exécution !

Dans ce qui suit, on note $P(\mathfrak{a})$ l'ensemble des noeuds qui sont forcement atteints quand on passe par $\mathfrak{a}$, et nous laisserons volontairement cette notion un peu floue pour l'instant. Sa définition sera précisée en §3.5.3.

3.3.3 Dépendances de contrôle

On définit $D(\mathfrak{c}, \mathfrak{s})$ comme l'ensemble des noeuds qui sont forcement atteints si on passe par $\mathfrak{s}$, mais pas forcement si on passe par $\mathfrak{c}$. Plus formellement :

$$D(\mathfrak{c}, \mathfrak{s}) = P(\mathfrak{s}) - P(\mathfrak{c})$$

Par exemple, dans une séquence simple, si $\mathfrak{c}$ représente un `if` et $\mathfrak{s}$ la première instruction de l'une des branche, $D(\mathfrak{c}, \mathfrak{s})$ donne les instructions de cette branche qui dépendent de la condition.

On définit alors $DpdC(\mathfrak{a})$, l'ensemble des dépendances de contrôle de $\mathfrak{a}$, par :

$$\mathfrak{c} \in DpdC(\mathfrak{a}) \Leftrightarrow def \mathfrak{a} \in \bigcup_{\mathfrak{s} \in Succ(\mathfrak{c})} D(\mathfrak{c}, \mathfrak{s})$$

Il est sans doute plus naturel de définir $CoDpdC(\mathfrak{c})$, l'ensemble des co-dépendances de contrôle de $\mathfrak{c}$, comme l'ensemble des noeuds ayant une dépendance de contrôle sur $\mathfrak{c}$, c'est-à-dire :

$$\mathfrak{a} \in CoDpdC(\mathfrak{c}) \Leftrightarrow def \mathfrak{c} \in DpdC(\mathfrak{a})$$

On a alors :

$$CoDpdC(\mathfrak{c}) = \bigcup_{\mathfrak{s} \in Succ(\mathfrak{c})} D(\mathfrak{c}, \mathfrak{s})$$

On remarque que cette définition ne suppose pas que $\mathfrak{c}$ ait plusieurs successeurs, mais si $\mathfrak{c}$ n'a qu'un successeur $\mathfrak{s}$:

$$Succ(\mathfrak{c}) = \{\mathfrak{s}\} \Rightarrow CoDpdC(\mathfrak{c}) = \{\mathfrak{x} | (\exists t \in [\![c, -[, \mathfrak{x} \notin t) \wedge (\forall t \in [\![s, -[, \mathfrak{x} \in t)\}$$

or, comme $\mathfrak{c}$ n'a qu'un successeur :

$$\forall t \in [\![c, -[, t = [c; s, -[$$

donc :

$$\forall t_s \in [\![s, -[, \mathfrak{x} \in t_s \Rightarrow \forall t_c \in [\![c, -[, \mathfrak{x} \in t_c$$

$$\forall t_s \in [\![s, -[, \mathfrak{x} \in t_s \Rightarrow \nexists t_c \in [\![c, -[, \mathfrak{x} \notin t_c$$

et donc, finalement :

$$Succ(\mathfrak{c}) = \{\mathfrak{s}\} \Rightarrow CoDpdC(\mathfrak{c}) = \{\}$$

Donc, un noeud n'ayant qu'un seul successeur ne peut pas être une dépendance de contrôle. **Attention** : ceci n'est pas vrai pour les saut inconditionnels, car ceux-ci donne lieu à un traitement spécial décrit en §3.4.

3.4 Les sauts inconditionnels

3.4.1 Présentation du problème

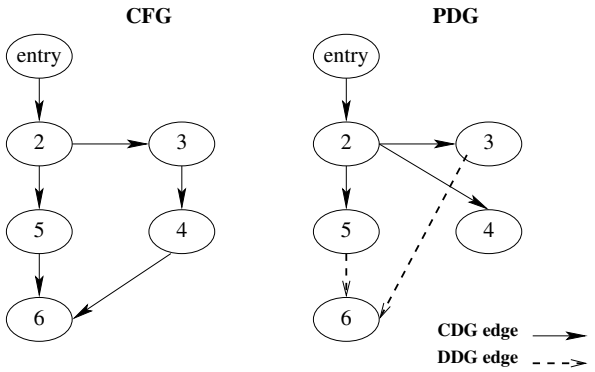
Comme on l'a vu, la définitions précédente des dépendances de contrôle conduit à ne construire des dépendances que sur les noeuds ayant plusieurs successeurs, c'est-à-dire ceux qui présente une forme de choix dans le CFG. Or certaines instructions n'ayant qu'un successeur peuvent aussi, par leur présence, modifier le flot de contrôle. En C, c'est le cas par exemple des `goto` explicites, mais aussi des `break`, `continue` ou `return`. Dans CIL, c'est aussi le cas des boucles puisqu'elles sont toutes transformées en `while(1)`.

Lorsque l'on souhaite utiliser le CDG pour calculer une réduction, on aimerait qu'il contienne également les liens nécessaires sur ces instructions afin de déterminer si elles peuvent être supprimées, ou si elles doivent être présentes dans le programme réduit.

L'exemple simple suivant, tiré de [Choi and Ferrante(1994)], met en évidence ce problème :

Exemple 6

```
1 : <entry>
2 : if (Q) goto 5;
3 : x = 4;
4 : goto 6;
5 : x = 5;
6 : y = x;
7 : <exit>
```



Réduction (fausse)
par rapport à <y,7> :

```
1 : <entry>
2 : if (Q) goto 5;
3 : x = 4;

5 : x = 5;
6 : y = x;
7 : <exit>
```

On voit que dans le graphe de dépendances (PDG), personne ne dépend de 4, et la réduction de ce programme par rapport au noeud 6 donne le résultat erroné ci-contre qui donne `y=5` même lorsque Q est faux.

3.4.2 Etat de l'art

La plupart des solutions proposées pour résoudre ce problème utilisent la notion de **successeur lexical** (sous différents noms).

Definition (successeur lexical immédiat selon [Agrawal(1994)])

A statement, S', is said to be **the immediate lexical successor** of a statement, S, in a program, if deleting S from the program will cause the control to pass to S' whenever it reaches the corresponding location in the new program.
If is a compound statement, such as an If or a While statement, delleting means delleting it allong with the statements that constitute its body.

[Choi and Ferrante(1994)] présente une méthode qui ajoute un pseudo-lien dans le CFG entre les `goto` et leur successeur lexical immédiat, et qui se sert de ce CFG modifié pour calculer les dépendances de contrôle selon la méthode classique. En fait, c'est un peu comme s'il remplaçait les `goto L;` par `if (1) goto L;` pour mettre en évidence le chemin qui apparaît dans le CFG si on supprime l'instruction.

[Agrawal(1994)] donne un algorithme qui permet de traiter les `goto` après un *slicing* "normal".

Il s'agit, pour chaque `goto` (G) non visible, de déterminer si son premier postdominateur présent dans la réduction est différent du premier successeur lexical. Si c'est le cas, il faut rentre (G) visible ainsi que toutes ses dépendances. Ceci donne les mêmes résultats que l'algorithme précédent, mais il permet de ne modifier ni le CFG, ni le PDT. Par contre, le calcul doit être fait pour chaque réduction.

[Harman and Danicic(1998)] propose un algorithme qui donne des résultats plus précis que les deux précédents dans certains cas.

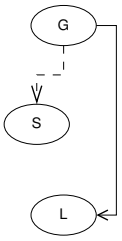
Enfin, [Kumar and Horwitz(2002)] présente un algorithme qui prend en compte le problème spécifique des `switch` et donne également de meilleurs résultats sur certains exemples.

3.4.3 Discussion

Dans un premier temps, l'algorithme de [Choi and Ferrante(1994)] semble simple à implémenter, mais il nous oblige à calculer un nouveau CFG et le PDT correspondant, ce que nous ne souhaitons pas faire car ces informations sont utilisées par différents modules de l'outil. Par ailleurs, l'algorithme de [Agrawal(1994)] est à appliquer à chaque nouvelle réduction, ce qui n'est pas très intéressant dans un environnement interactif, d'autant plus que l'on peut vouloir utiliser les dépendances de contrôle pour autre chose que le *slicing*.

Voyons donc plus précisément ce que l'on veut obtenir :

Le problème



- soit un programme P,
- soit $\mathfrak{g}$ un noeud du CFG de P correspondant à un `goto`,
- soit $\mathfrak{l}$ le noeud correspondant au successeur de $\mathfrak{g}$ (label du `goto`),
- soit P', le programme P dans lequel on remplace le `goto` par un ';' (NOP),
- soit $\mathfrak{g}$' le noeud correspondant dans le CFG de P',
- soit $\mathfrak{s}$ le successeur lexical de $\mathfrak{g}$.

On veut qu'un noeud $\mathfrak{a}$ ait une dépendance de contrôle sur $\mathfrak{g}$ si et seulement si $\mathfrak{a}$ postdomine soit $\mathfrak{g}$, soit $\mathfrak{s}$, mais pas les deux.

Examinons les différents cas :

1. si $\mathfrak{s}$=$\mathfrak{l}$, cela signifie que le `goto` ne sert à rien, personne ne dépend donc de $\mathfrak{g}$,
2. si $\mathfrak{l}$ postdomine $\mathfrak{s}$, $\mathfrak{s}$ dépend de $\mathfrak{g}$, ainsi que tous les postdominateurs de $\mathfrak{s}$ qui ne postdomine pas $\mathfrak{l}$,
3. si $\mathfrak{l}$ ne postdomine pas $\mathfrak{s}$, tous les $\mathfrak{a}$ qui postdominent $\mathfrak{s}$, mais pas $\mathfrak{l}$ ou l'inverse, dépendent de $\mathfrak{g}$.

Donc, de manière générale, on peut calculer :

$$CoDpdC(\mathfrak{g}) = (P(\mathfrak{s}) \cup P(\mathfrak{l})) - (P(\mathfrak{s}) \cap P(\mathfrak{l}))$$

ce qui nous donne bien les noeuds atteints par l'une ou l'autre des branches, mais pas par les deux.

On peut d'ailleurs montrer qu'on obtient la même chose que si on calcule $CoDpdC(\mathfrak{g}^*)$ sur CFG* où CFG* est le CFG dans lequel le noeud $\mathfrak{g}$ est remplacé par un noeud $\mathfrak{g}^*$ correspondant à une instruction `if (true) goto L;`

3.5 Boucle infinie et exit

Comme on l'a vu, la plupart des définitions de dépendance de contrôle se basent sur le CFG, la notion de postdominateurs, et utilisent l'hypothèse que le CFG a un unique noeud de sortie $\mathfrak{E}$ atteignable depuis tous les autres points du graphe. Or, comme l'explique très bien [Ranganath et al.(2004)Ranganath, Amtoft, Banerjee, Dwyer, and Hatcliff], cette hypothèse ne tient plus dans les programmes contenant des boucles infinies ou des `exit` (ou même des exceptions, mais pour le langage C, nous n'avons pas le problème). Cet article présente également avec beaucoup de détails différents types de dépendances et des algorithmes pour les calculer, mais il s'avère probablement trop complexe pour ce que l'on souhaite faire.

Le fait qu'une fonction n'atteigne pas forcement un point de sortie pose deux problèmes différents :

— la préservation de la non-terminaison,
— le calcul des dépendances de contrôle pour les instructions n'ayant pas de chemin vers la sortie.

3.5.1 Préservation de la non-terminaison

La question qui se pose est de savoir s'il faut ajouter des dépendances de contrôle sur les instructions qui, potentiellement, ne terminent pas.

Exemple 7

```
while (f(x) > 0) x++;
L: y = 3;
```

Si l'on s'intéresse au calcul de `y` en `L`, on peut se demander si cette instruction a une dépendance de contrôle sur la boucle, car si dans certains cas, la boucle ne termine pas, `L` n'est pas atteint le même nombre de fois dans le programme source P, et dans le programme P' dans lequel la boucle est remplacée par un NOP.

Pour prendre en compte ce type de problème et permettre d'effectuer par la suite des analyses sensibles à la non-terminaison (*non-terminaison sensitive*), il faut ajouter des dépendances à toutes les instructions qui suivent une construction qui peut ne pas terminer comme un appel de fonction, ou une boucle dont on ne sait pas déterminer la terminaison. Dans l'absolu, il faudrait aussi s'intéresser aux autres instructions qui ont une terminaison anormale (`Segmentation Fault` par exemple) mais il semble raisonnable de considérer que ces instructions ne sont jamais présentes exprès, et que leur absence doit être vérifiée par ailleurs.

3.5.2 Postdominateurs généralisés

Définition

En cas de boucle infinie, le CFG contient des noeuds pour lesquels il n'existe pas de chemin vers la sortie. Or, pour un tel noeud $\mathfrak{a}$, les postdominateurs tels que définis plus haut me peuvent pas être utilisé pour déterminer les dépendances de contrôle. En effet, on ne peut pas parler des instructions qui vont être forcement exécutées entre $\mathfrak{a}$ et $\mathfrak{E}$ car il n'existe pas de telles traces.

La plupart des travaux existants travaillent dans ce cas sur un CFG augmenté dans lequel le noeud $\mathfrak{E}$ est ajouté, ainsi que des arrêtes pour le rendre accessible. Outre le fait que certaines analyses n'aient pas besoin de cette hypothèse, et qu'une telle modification vienne "polluer" le CFG, il semble difficile (impossible ?) de ne pas ajouter de dépendances parasites dans le cas des boucles infinies.

Pourtant, même si $\mathfrak{a}$ est dans une boucle infinie, on a intuitivement une notion de postdominateurs. On aimerait bien dire qu'une instruction $\mathfrak{b}$ postdomine une instruction $\mathfrak{a}$ si et seulement si $\mathfrak{b}$ appartient à tous les chemins **partant de** $\mathfrak{a}$, c'est-à-dire :

Definition (postdominateurs généralisés)

$$\mathfrak{b} \in Pd^{\infty}(\mathfrak{a}) \Leftrightarrow def \forall t \in [\![a, -[, \mathfrak{b} \in t$$

On ne considère donc plus uniquement les chemins qui atteignent la sortie, mais toutes les traces d'exécution possible.

Méthode de calcul

Intuitivement, si on souhaite connaître tous les noeuds qui ont b comme postdominateurs en considérant tous les chemins, on peut calculer les ensembles suivants :

$$E_b(a) = \begin{cases} b & \text{si } b = a \\ \bigcap_{s \in Succ(\mathfrak{a})} E_b(s) & \text{sinon.} \end{cases}$$

en partant d'ensembles initialement vides. Ce calcul termine car il est croissant.

A la fin du calcul, on a :

$$\forall b, E_a(b) = \{a\} \vee E_a(b) = \bot$$

et $E_a(b) = \{a\}$ veut bien dire que a est dans tous les chemins entre b et a.

On peut faire ce calcul pour tous les points du CFG, et faire l'union de tous les résultats obtenus :

$$Pd^{\infty}(()a) = \bigcup_{b \in CFG} E_b(a)$$

Bien sûr, ce n'est pas la manière la plus efficace de faire le calcul, mais on voit trivialement que c'est le résultat que l'on souhaite obtenir.

Il suffit maintenant de trouver un calcul équivalent, moins coûteux, mais qui termine néanmoins...

En fait, il suffit de faire directement l'union des résultats au fur et à mesure du calcul, mais comment prouver la terminaison ?

Discussion

On remarque que cette définition ne donne pas les mêmes postdominateurs que précédemment dès qu'il y a une boucle dans le CFG, même qui la sortie $\mathfrak{E}$ est accessible depuis tous les noeuds, car il existe alors un chemin possible qui consiste à boucler indéfiniment. Par conséquent, les instructions situées après la boucle ne postdomine pas les instructions de la boucle puisqu'il existe un chemin pour lequel on y passe pas. Cela traduit le fait que l'exécution de ces instructions dépend du fait que la boucle termine...

Si on utilise cette définition des postdominateurs pour calculer des dépendances de contrôle, on va donc ajouter des liens entre la boucle et tout ce qui suit. Ces nouvelles dépendances traduisent la possible non-terminaison de la boucle, et peuvent donc servir dans le cadre d'une analyse préservant la non-terminaison comme on l'a vu précédemment. Mais on ne souhaite pas nécessairement ajouter toutes ces dépendances, car en pratique, on peut montrer par ailleurs que la plupart des boucles d'un programme terminent.

3.5.3 Postdominateurs augmentés

On peut essayer de faire un mélange des postdominateurs classiques et des postdominateurs généralisés en supposant que toutes les boucles ayant une sortie se terminent (attention : on considère pour l'instant qu'il y a un chemin possible entre cette sortie et $\mathfrak{E}$), mais en considérant néanmoins les traces infinies dans les cas où une boucle n'a pas de sortie.

Definition (postdominateurs augmentés)

$$Pd^{+}(\mathfrak{a}) = \begin{cases} \{b | \forall t \in [\![a, -[, b \in t\} & \text{si } [\![a, \mathfrak{E}] = \{\} \\ \{b | \forall t \in [\![a, \mathfrak{E}[, b \in t\} & \text{sinon.} \end{cases}$$

Méthode de calcul

Pour faire ce calcul, il faut être capable de distinguer les chemins qui mènent à $\mathfrak{E}$ des autres. On note $ToRet(\mathfrak{a})$ la propriété qui dit que $\mathfrak{a}$ a un chemin vers $\mathfrak{E}$.

Après un calcul classique des postdominateurs (en partant de $\mathfrak{E}$), les postdominateurs des noeuds tels que $ToRet(\mathfrak{a})$ sont établis. Reste à calculer l'information pour les autres noeuds, c'est-à-dire ceux qui ont les postdominateurs à $\top$.

Pour ceux-là, on fait un calcul similaire :

$$Pd^+(\mathfrak{x}) = \{\mathfrak{x}\} \bigcup \bigcap_{s \in Succ(\mathfrak{x})}^{a} Pd^+(\mathfrak{s})$$

en redéfinissant simplement l'intersection utilisée de la façon suivante :

$$Pd^+(\mathfrak{a}) \cap^a Pd^+(\mathfrak{b}) = \begin{cases} Pd^+(\mathfrak{a}) \cap Pd^+(\mathfrak{b}) & \text{si } ToRet(\mathfrak{a}) = ToRet(\mathfrak{b}) \\ Pd^+(\mathfrak{a}) & \text{si } ToRet(\mathfrak{a}) \wedge \neg ToRet(\mathfrak{b}) \\ Pd^+(\mathfrak{b}) & \text{si } \neg ToRet(\mathfrak{a}) \wedge ToRet(\mathfrak{b}) \end{cases}$$

Discussion

On remarque qu'avec cette définition, il existe des noeuds $\mathfrak{a}$ tels que :

$$Succ(\mathfrak{a}) = \{\mathfrak{s}1, \mathfrak{s}2\} \wedge Pd^+(\mathfrak{a}) \neq Pd^+(\mathfrak{s}1) \cap Pd^+(\mathfrak{s}2)$$

dans le cas où on atteint la sortie à partir de l'un des successeurs, mais pas de l'autre.

Par exemple, si on considère un noeud $\mathfrak{c}$ correspondant à un IF dont l'une des branche est une boucle infinie et que l'autre permet d'atteindre la sortie, la boucle infinie dépendra de $\mathfrak{c}$, alors que l'accès à la sortie n'en dépendra pas.

3.6 En résumé

Pour calculer les dépendances de contrôle, on commence donc par calculer les postdominateurs augmentés définis en §3.5.3. Puis, pour chaque saut, on calcule ses co-dépendances de contrôle de la façon suivante :

— pour un IF, on applique la définition donné en §3.3.3, c'est-à-dire :

$$CoDpdC(\mathfrak{c}) = \bigcup_{\mathfrak{s} \in Succ(\mathfrak{c})} Pd^+(\mathfrak{s}) - Pd^+(\mathfrak{c})$$

— pour un saut inconditionnel (`goto`, `break`, etc), comme mentionné en §3.4.3, on calcule :

$$CoDpdC(\mathfrak{g}) = (Pd^+(\mathfrak{s}) \cup Pd^+(\mathfrak{l})) - (Pd^+(\mathfrak{s}) \cap Pd^+(\mathfrak{l})) \text{ où } \begin{cases} \mathfrak{s} = Succ_L(\mathfrak{g}) \\ \mathfrak{l} = Succ(\mathfrak{g}) \end{cases}$$

Si on considère $\mathfrak{s}$ et $\mathfrak{l}$ comme deux pseudo-successeurs de $\mathfrak{g}$, les pseudo-postdominateurs de $\mathfrak{g}$ sont donnés par $(Pd^+(\mathfrak{s}) \cap Pd^+(\mathfrak{l}))$, et on retrouve bien la formule précédente.

— pour les boucles, comme dans CIL, elles sont toutes infinies, on traite la séquence `while(1) S;` comme si on avait `L : S; goto L;`

Chapitre 4

Dépendances interprocédurales

On a vu qu'un PDG est associé à une fonction. La question se pose donc de savoir comment calculer des dépendances interprocédurales, c'est-à-dire comment mettre en relation les appels de fonctions et les dépendances des fonctions appelées.

Nous allons tout d'abord voir qu'un appel de fonction est représenté par plusieurs éléments dans le PDG (§4.1). Puis, nous allons voir que pour mettre en relation des appels et les fonctions appelées, ils faut ajouter d'autres éléments à chaque fonction (§4.2).

4.1 Appels de fonction

L'instruction contenant l'appel est représentée par plusieurs éléments dans le graphe de dépendances afin de pouvoir plus précisément mettre en relation les appels aux fonctions appelées.

Les éléments créés sont les suivants :

- un élément pour chaque paramètre de la fonction appelée ; les dépendances sont crées par une simulation de l'affectation des arguments d'appel dans les paramètres formels,
- un élément représentant le contrôle du point d'entrée de la fonction appelée (un peu comme si l'appel était dans un bloc et que ce noeud représentait ce bloc),
- un élément pour chaque sortie, dépendant des entrées correspondantes.

Pour ne pas avoir à calculer les flots de données de toutes les fonctions de l'application, il a été décidé d'utiliser les dépendances (*from*) calculées indépendamment par Frama-C. La liste des entrées et des sorties, ainsi que les dépendances entre les unes et les autres sont extraites des spécifications des fonctions appelées, et non de leur PDG [1]. Ceci est vrai également pour les fonctions dont le code est absent de l'application étudiée car cela permet d'être cohérent avec les autres analyses. Cela permettra en particulier, d'utiliser d'éventuelles propriétés fournies par la suite par l'utilisateur.

1. c'est peut-être un problème si on fait de la coupure de branche, car les dépendances peuvent être réduites par une telle spécialisation.

Exemple 8

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

/*@ assigns \result {a},
            G.a {G, a} */
int g (int a);

int f (int x, int y) {
  G.b = x;
  x = g (x+y);
  return x + G.b;
}
```

Ici, pour représenter l'appel à `g` dans `f` dans le PDG, on va avoir :
- un élément représentant le point d'entrée dans `g`,
- un élément e_1 pour représenter `a = x+y`, c'est-à-dire l'affectation de l'argument de l'appel dans le paramètre formel de `g`,
- un élément e_2 pour calculer la valeur de retour de `g`, qui dépend de la valeur de e_1 (utilisation de la spécification de g),
- et enfin, un élément e_3 qui représente la seconde sortie de `g` : `G.a` qui dépend du paramètre `a` et donc de e_1 et de des éléments $\{e_G\}$ correspondant à la valeur de G avant l'appel (selon la spécification de g).

On note que, contrairement à ce qui était fait dans la version précédente, on ne crée par d'élément pour l'entrée implicite G de g dans f. Cela permet d'améliorer la précision des dépendances lorsque l'ajout d'un tel noeud conduisait au regroupement de plusieurs données.

Ainsi, dans l'exemple précédent, on ne crée pas d'élément pour représenter la valeur de G avant l'appel, même si l'élément e_3 en dépend, et on conserve donc l'information que $G.b$ ne dépend que de l'affectation précédent l'appel.

4.2 Entrées/sorties d'une fonction

Pour relier un appel de fonction au PDG de la fonction appelée, il faut ajouter des éléments représentant ses entrées/sorties, c'est-à-dire :

- un élément correspondant au point de contrôle d'entrée dans la fonction,
- deux éléments pour chaque paramètre (cf. §4.2.2),
- un élément pour les entrées implicites (cf. §4.2.1),
- un élément pour la sortie de la fonction si celle-ci retourne quelque chose.

On note que, contrairement à ce qui était fait dans la version précédente, on ne crée par d'élément pour les sorties implicites de la fonction. Cela permet d'améliorer la précision des dépendances lorsque l'ajout d'un tel noeud conduisait au regroupement de plusieurs données.

C'est par exemple le cas lorsqu'une fonction calcule $G.a$, puis $G.a.x$ car un élément de sortie regrouperait les deux alors que si par la suite on s'intéresse juste à $G.a.x$ à la sortie de la fonction, le fait de ne pas avoir créé cet élément permet de retrouver l'information plus précise.

4.2.1 Entrées implicites

Au cours du calcul du PDG, on mémorise l'utilisation des données qui ne sont pas préalablement définies. Cela permet par la suite que créer des éléments pour ces entrées dites implicites. On ne crée pas d'élément pour les variables locales non initialisées, mais un message d'avertissement est émis. Il est possible que ce soit une fausse alerte dans le cas où l'initialisation

est faite dans une branche dont la condition est forcement vrai à chaque exécution où l'on passe par la suite par l'utilisation.

Diverses stratégies de regroupement de ces entrées peuvent être utilisées. A ce jour, l'outil construit tous les éléments lui permettant d'avoir une meilleure précision. C'est-à-dire que deux éléments peuvent représenter les données qui ont une intersection.

4.2.2 Déclaration des paramètres formels

En plus de l'élément représentant la valeur des paramètres, on crée un second élément qui représente sa déclaration, le premier dépendant du second.

Cette représentation peut permettre d'avoir une meilleure précision dans le cas où certains calcul ne dépendent pas de la valeur du paramètre, mais uniquement de sa déclaration.

Exemple 9

```
int g (int a) {
  G = 2 * a;
  a = calcul_a ();
  return a;
}
int f (void) {
  int x = calcul_x ();
  return g (x);
}
```

On voit que la valeur de retour de `g` ne nécessite pas la valeur initiale de `a`, mais seulement sa déclaration. La valeur de retour de `f` ne dépend donc pas de l'appel à `calcul_x`.

Ce point n'est pas encore implémenté dans la version actuelle, car dans des cas plus complexe, il est délicat de savoir ce qu'il faut garder dans la fonction appelante. Le plus simple serait sans doute de transformer le paramètre formel en une variable locale, mais le filtrage permet à l'heure actuelle de garder ou d'effacer des éléments existants, mais pas d'effectuer des transformations de code...

4.3 Fonctions à nombre d'arguments variable

Pour l'instant, on ne calcule pas le PDG des fonctions à nombre d'arguments variable, c'est-à-dire que pour le reste de l'application, tout se passe comme si on n'avait pas le code source de ces fonctions.

En revanche, les appels à de telles fonctions sont gérées de manière semblable à ce qui est fait pour les autres appels, c'est-à-dire :

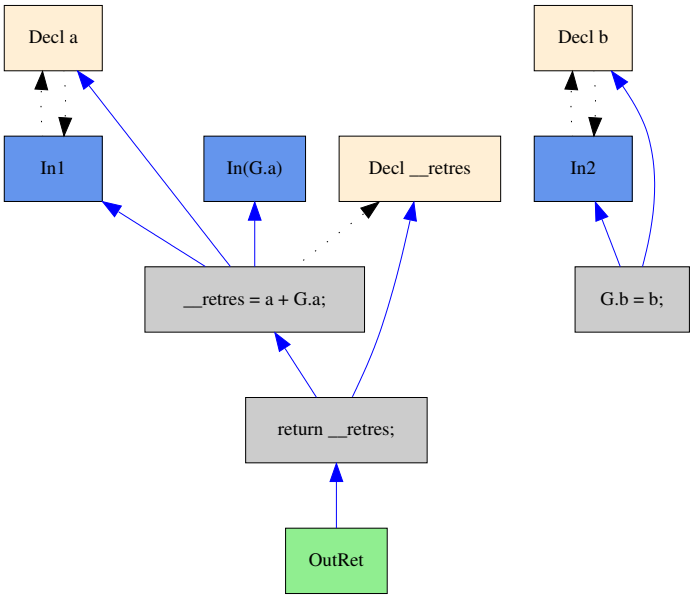
- création d'un noeud d'entrée pour chaque argument d'appel, (il y en a donc éventuellement plus que que paramètres formels dans le déclaration de la fonction appelée)
- utilisation des informations *from* pour créer les éventuelles entrées implicites, les sorties, et les liens de dépendance.

4.4 Exemple

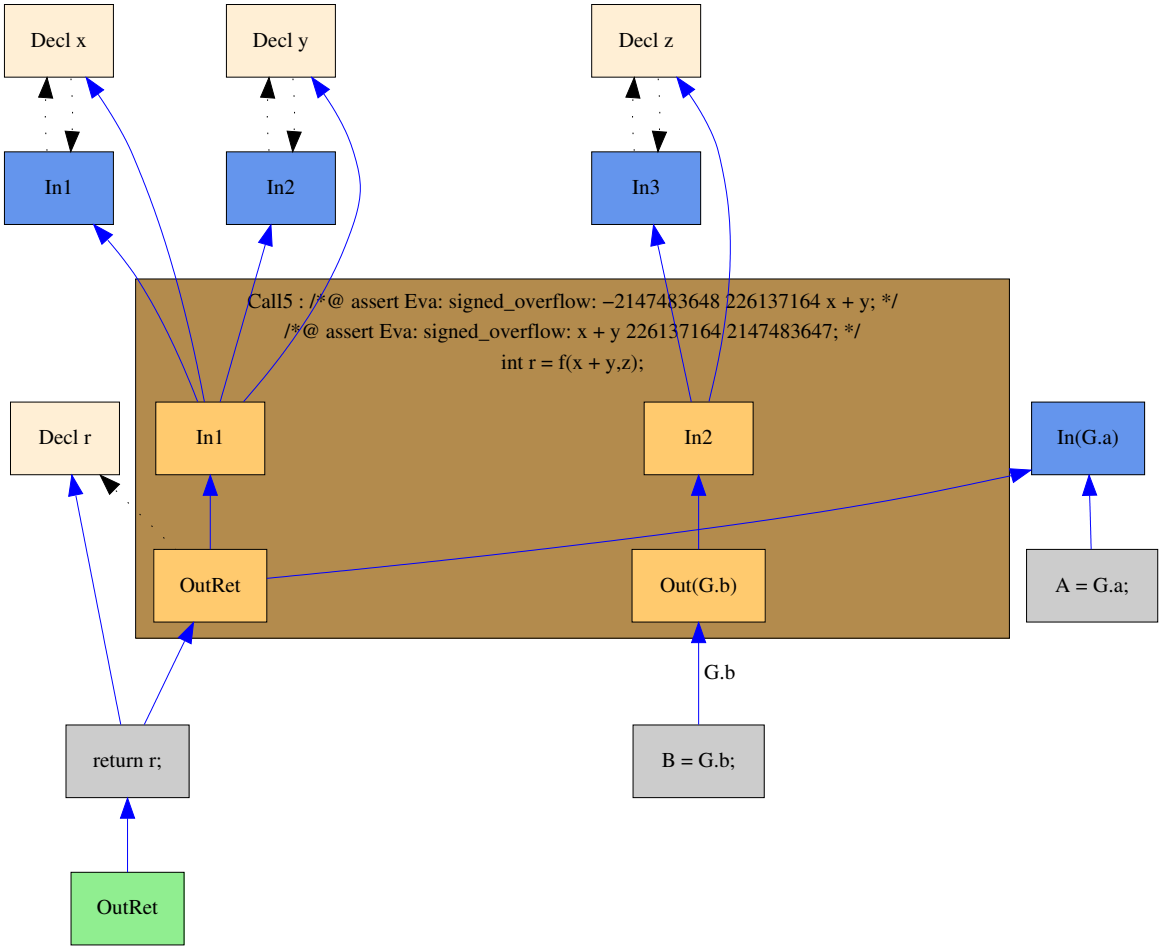
Exemple 10

```
struct {int a; int b; } G;
int A, B;

int f (int a, int b) {
    G.b = b;
    return a + G.a;
}

int g (int x, int y, int z) {
  int r = f (x+y, z);
  A = G.a;
  B = G.b;
  return r;
}
```

Graphe de la fonction `f` :



Graphe de la fonction g :



Les graphes sont ceux qui sont effectivement produits par l'outil.

Chapitre 5

Retrouver l'information

Jusqu'à présent, on a vu comment calculer le graphe de dépendances. Dans ce chapitre, nous présentons les fonctions fournies pour trouver des éléments du PDG suivant un certain nombre de critères. L'annexe A rappelle certains objectifs initiaux d'une utilisation de ces fonctions pour effectuer une analyse d'impact.

Par ailleurs, outre la manipulation d'ensemble d'éléments du PDG, un système de gestion et de propagation de marques est également proposé. Il est présenté au chapitre 6.

5.1 A partir de leur clé

Au cours du calcul, chaque élément du PDG est associé à une clé qui correspond à l'élément de programme qu'il représente (cf. `pdgIndex.mli`). On peut par la suite retrouver un élément à partir de cette clé comme par exemple l'élément correspondant à une instruction simple, à un paramètre d'entrée, à une déclaration locale, etc.

5.2 A partir d'une zone mémoire

Grâce à l'état qui est propagé lors de la construction des dépendances de données (cf. §2.2) on peut retrouver les éléments qui participent au calcul d'une zone mémoire à un point de programme.

Cette fonction doit également vérifier si les éléments trouvés définissent complètement la donnée recherchée ou non, et si ce n'est pas le cas, indiquer la zone susceptible de ne pas être complètement définie au point considéré.

5.3 A partir de propriétés

D'autres fonctionnalités de Frama-C permettent d'interpréter une propriété du programme et d'en extraire les zones mémoire nécessaires à son évaluation. Comme on sait par ailleurs trouver les éléments qui correspondent a une zone mémoire en un point de programme (voir ci-dessus), on peut ainsi trouver par exemple les éléments dont dépendent une assertion ou tout autre propriété.

5.4 En exploitant les dépendances

Après avoir retrouver des éléments dans le graphe, on veut en général exploiter leurs dépendances. On peut facilement retrouver les dépendances avant ou arrière, avec ou sans filtrage sur leur type (donnée, contrôle,...), récursivement ou non.

Chapitre 6

Marquage générique

6.1 Objectif

On souhaite pouvoir associer une information à certains éléments d'un PDG, propager cette information dans les dépendances de ces éléments, et calculer ce qu'il faut propager dans les autres PDG de l'application pour avoir un traitement interprocédural cohérent.

Par exemple, l'information peut être un simple booléen qui indique si l'élément correspondant est visible ou non. Pour qu'un élément soit visible, il faut que toutes ses dépendances le soit aussi. On voit dans cet exemple qu'on s'intéresse dans un premier temps à une propagation arrière.

Comme un PDG est associé à une fonction, la propagation s'arrête aux entrées de la fonction. Par ailleurs, les appels de fonction sont traités localement, sans descendre dans la fonction appelée. Pour gérer la propagation interprocédurale, les fonctions élémentaires de marquage doivent fournir l'information à propager aussi bien dans les fonctions appelées que dans les appelants.

6.2 Marquage générique

Le marquage pouvant être utilisé pour différents besoins, les marques peuvent être définies par l'utilisateur. Il lui suffit de définir comment en faire l'union.

On utilise deux types d'union :
- l'une permet de faire une simple union de deux marques,
- l'autre permet d'ajouter une marque à un élément de programme : elle calcule donc une nouvelle marque à partir de l'ancienne, et de la marque à ajouter et fournit aussi la marque à propager dans les dépendances.

6.3 Propagation arrière

6.3.1 Marquage intraprocédural

La propagation arrière s'effectue très simplement en suivant les dépendances, en ajoutant la marque propagée à la marque existante, et en s'arrêtant quand la marque à propager rendue

par la fonction de marquage est $m_\perp$.

En cours de propagation, lorsque l'on traite un élément correspondant à la sortie d'un appel de fonction, ou à une entrée de la fonction courante, la marque à propager ainsi que l'identifiant de l'élément sont stockés. A l'issu du marquage, on a donc :

- un ensemble de couple (entrée de la fonction, marque à propager),
- une liste des appels de fonction, avec pour chacun un ensemble de couples (sortie de l'appel, marque à propager).

C'est cette information qui permettra de faire la propagation interprocédurale.

6.3.2 Propagation interprocédurale

Méthode automatique

Le module de marquage permet de faire automatiquement la propagation interprocédurale si on le souhaite. Il est utilisé de cette façon dans la détection de code mort par exemple.

L'utilisateur peut, s'il le souhaite, donner des fonctions de transformation des marques :

- une qui est utilisée pour transformer les marques des entrées d'une fonction avant la propagation dans les appelants,
- une autre qui est utilisée pour transformer les marques des sorties d'un appel de fonction avant propagation dans la fonction appelée.

Dans les cas les plus simples, ces fonctions ne font rien, et rendent simplement la marque qui leur est passée.

Méthode manuelle

Dans certains cas, il est souhaitable de gérer cette propagation à la main. C'est par exemple le cas dans le *slicing* où une fonction source est associée à plusieurs marquages différents.

Des fonctions aident néanmoins à retrouver les éléments à marquer dans les autres fonctions à partir du résultat fourni par le marquage intraprocédural. Ces fonctions traduisent les informations :

- (entrée de la fonction, marque à propager) en un ensemble d'éléments à marquer dans une - ou toutes les - fonction(s) appelante(s),
- (appel, {(sortie de l'appel, marque à propager)}) en un ensemble d'éléments à marquer dans la fonction appelée.

Les éléments rendu sont les nouveaux points de départ de marquages intraprocéduraux. L'utilisateur peut, comme lors de la propagation automatique, donner des fonctions de transformation des marques aux traductions.

Données non définies

Lorsque les points de départ marquage initial sont des éléments du PDG, toutes les propagations s'effectueront a priori sur des éléments existant. Mais ce n'est pas forcement le cas lorsqu'il s'agit de marquer les données qui n'interviennent pas dans les calculs. Par exemple :

Exemple 11

```
int X1, X2;

void f (int x) { /*@ assert X1 >= 0; */ return x + 1; }

void f2 (void)  { X2 = f (2); }

void main (void) { X1 = 0; X2 = 0; f2 (); }
```

Ici, si on s'intéresse à l'assertion de `f`, il faut sélectionner `X1` à l'entrée de `f`, propager cette information à travers `f2` pour enfin marquer l'affectation à `X1` dans le `main` alors que cette donnée n'apparaît pas dans les PDG de `f` et `f2`.

On voit donc que dans les informations fournies et calculées pour gérer l'interprocédural, il peut y avoir des éléments ne correspondant pas à des noeuds de PDG.

Terminaison

La terminaison du processus dépend de la définition des marques. C'est donc à l'utilisateur de s'assurer que celle-ci permet la stabilisation du marquage.

6.4 Propagation avant

La propagation avant n'est pas effectuée automatiquement à l'heure actuelle.

La propagation intraprocédurale ne pose pas de problèmes particuliers, car elle est très semblable au calcul en arrière : il suffit simplement de parcourir les liens de dépendance dans l'autre sens.

La partie interprocédurale semble un peu plus délicate, car il n'y a aucun point de repère dans le PDG pour identifier les endroits à partir desquels il faut propager (hormis le noeud correspondant au `return` pour la propagation vers les appelants et les arguments explicites pour la propagation dans les appels de fonction).

Des fonctions supplémentaires sont donc fournies pour :

- trouver les noeuds d'entrées d'une fonction qui doivent être sélectionnés à partir des noeuds sélectionnés dans l'appelant,
- trouver les noeuds de sortie d'un appel de fonction qui doivent être sélectionnés à partir des noeuds sélectionnés dans la fonction appelée.

En cas d'utilisation intensive de ces fonctionnalités, il serait sans doute intéressant de mémoriser les liens entre les différents PDG (cela est également vrai pour la propagation interprocédurale arrière...).

Chapitre 7

Conclusion

La version actuelle de ce greffon semble fonctionner. Elle est utilisée par les greffons Security, Sparecode et Slicing. Ces résultats peuvent également être visualisés graphiquement en utilisant la fonction d'exportation au format `.dot`.

D'autres information relatives au développement peuvent être trouvées dans la documentation du code dont un point d'entrée est :

`doc/code/pdg/index.html`

7.1 Limitations

Les fonctions ayant un nombre d'arguments variable ne sont pas traitées (mais les appels à de telles fonctions sont gérés).

Par ailleurs, les calculs se basant sur l'analyse de valeur, et sur le calcul des dépendances fonctionnelles (From) il hérite des limitations de ces modules.

Annexe A

Aide à l'analyse d'impact

Le document [Baudin(2004)] spécifie un outil d'aide à l'analyse d'impact après avoir analysé les besoins dans ce domaine. L'exploitation du graphe de dépendances est présenté comme étant un composant important de cet outil. Nous reprenons ici les éléments de spécification décrits dans le paragraphe 5.3 de ce document, auxquels nous ajoutons quelques nouveaux critères.

A.1 Définition d'ensembles

Il s'agit de définir des sous-ensembles d'instructions répondant à différents critères.

Donnons tout d'abord quelques notations :

- S : un ensemble d'instructions,
- L : un point de programme,
- V : une zone mémoire (le V se réfère à **V**ariable, mais il peut s'agir d'une donnée quelconque comme un champ de structure ou un élément de tableau),

Les ensembles sont tous relatifs à un point de programme L, et peuvent se classer en deux catégories :

- ceux qui contiennent des instructions situées **avant** L : ils portent un indice 0,
- et ceux qui contiennent des instructions situées **après** L : ils portent un indice 1.

$R_0(S, L)$: sous-ensemble d'instructions de S depuis lesquelles L est accessible.

▲ mais L ne postdomine pas forcement toutes les instructions de cet ensemble.

$R_1(S, L)$: sous-ensemble d'instructions de S qui sont (éventuellement) accessibles depuis L.

$R_{L0}(S, L)$: accessibilité à un point du code.

- il s'agit du sous-ensemble des instructions de $R_0(S, L)$ qui conditionnent le passage en L, c'est-à-dire les instructions de branchement et les dépendances de ces conditions de branchement.

$R_{V0}(S, L, V)$: contenu d'une zone mémoire en un point du code.

- il s'agit du critère traditionnel de *slicing*, c'est-à-dire que cet ensemble contient les instructions de S qui ont une influence sur la valeur de V en L.

$R_{V1}(S, L, V)$: utilisation d'une zone mémoire.

— il s'agit du sous-ensemble des instructions de $R_1(S, L)$ influencées par la valeur qu'à la zone mémoire V a en L.

$R_{DV0}(S, L, V)$: définition de la valeur d'une variable à un point de programme.

— il s'agit de l'ensemble des instructions qui **définissent** tout ou partie de la valeur de V en L. Ces instructions sont donc nécessairement des affectations ou des appels de fonction.
On note que : $R_{DV0}(S, L, V) \subseteq R_{V0}(S, L, V)$

▲ Il serait probablement utile d'avoir un élément spécial qui indique qu'il existe un ou plusieurs chemin menant à L pour le(s)quel(s) V n'est pas entièrement défini.

$R_{DV1}(S, L, V)$: modification de la valeur d'une variable.

— il s'agit de trouver les instructions I_i accessibles depuis L qui modifient la valeur de V, et telles qu'il existe un chemin entre L et I_i le long duquel V n'est pas modifiée. Cela signifie que ces instructions sont les premières à modifier la valeur de V sur les chemins partant de L.

— Il n'est pas sûr que cet ensemble soit très utile. L'ensemble $R_{P1}(S, L, V)$ ci-dessous en sans doute plus intéressant.

$R_{P0}(S, L, V)$: portée arrière de la valeur d'une variable à un point de programme.

— il s'agit d'un sous-ensemble de $R_{L0}(S, L)$ contenant les instructions I_i pour lesquelles la valeur de V n'a été modifiée sur aucun chemin entre le point qui précède I_i et L. C'est-à-dire que la valeur de V en L est la même qu'avant chacune de ces instructions, et qu'elle n'est pas modifiée entre les deux.

$R_{P1}(S, L, V)$: portée de la valeur d'une variable à un point de programme .

— il s'agit d'un sous-ensemble de $R_1(S, L)$ contenant les instructions I_i pour lesquelles la valeur de V n'a été modifiée sur aucun chemin entre L et le point de programme qui précède I_i.

$R_{PV0}(S, L, V)$: utilisation arrière d'une zone mémoire dans sa portée.

— il s'agit du sous-ensemble des instructions $R_{P0}(S, L, V)$ qui sont influencées, directement ou non, par V.

$R_{PV1}(S, L, V)$: utilisation d'une zone mémoire dans sa portée.

— il s'agit du sous-ensemble des instructions influencées, directement ou non, par la valeur de V en L qui sont dans la portée de cette valeur.
On a donc : $R_{PV1}(S, L, V) = R_{V1}(S, L, V) \cap R_{P1}(S, L, V)$

Exemple 12

On montre ici différents ensembles relatifs au point L et à la variable v :

S	R_0	R_{L0}	R_{V0}	R_{DV0}	R_{P0}	R_{VP0}	R_1	R_{V1}	R_{DV1}	R_{P1}	R_{VP1}
`v = a;`	X	X	X	X							
`x = b;`	X		X								
`z = c;`	X										
`if (c1>v) {`	X	X	X								
`z++;`											
`}`											
`else {`											
`if (c2)`	X	X	X								
`v += x;`	X	X	X	X							
`if (c3) {`	X	X	X		X						
`z += v;`	X	X			X	X					
L: `y = v;`							X	X		X	X
`z++;`							X			X	
`v++;`							X	X		X	X
`}`											
`z += 2*y;`							X	X			
`}`											
`z += x;`							X	X			
`v = 0;`									X		

La construction des ensembles ci-dessus exploite le CFG (le flot de contrôle) et le PDG (les dépendances).

La construction des ensembles ci-dessous exploite en plus la sémantique du programme car elle utilise une contrainte $C(...v_i...)$ portant sur les valeurs des données en L :

$R_{C0}(S, L, C(...v_i...))$: accessibilité contrainte à un point.

— il s'agit de déterminer l'ensemble des instructions qui, si elles sont présentes, rendent fausse la contrainte C au point L. En fait, il s'agit de déplacer C pour essayer de couper des branches. La condition de branchement sera alors éventuellement remplacée par un `assert`.

☺ Dans la séquence `int x = 0; x=x-1; L:` pour laquelle on a une contrainte $x \geq 0$, il ne s'agit pas de supprimer l'instruction `x=x-1;` mais bien de dire que cette branche est impossible...

$R_{C1}(S, L, C(...v_i...))$: accessibilité contrainte depuis un point.

— il s'agit de déterminer le sous-ensemble des instructions de $R_1(S, L)$ qui ne peuvent pas être atteintes si l'état en L est tel que C est satisfaite.

On notera que certaines instructions n'appartenant ni à R_{L0}, ni à R_1 peuvent également ne pas être atteintes du fait de la contrainte.

A.2 Exploitation du graphe pour le calcul d'ensembles

La plupart des ensembles définis en A.1 peuvent être calculés simplement en exploitant le flot de contrôle et les fonction du graphe de dépendances présentées en $5.

Bibliographie

[Agrawal(1994)] H. Agrawal. On slicing programs with jump statements. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, 1994. URL http://citeseer.ist.psu.edu/agrawal94slicing.html.

[Baudin(2004)] P. Baudin. Analyse d'impact : spécification de l'outil. Technical Report DTSI/SOL/04-187, CEA, juin 2004.

[Choi and Ferrante(1994)] J.-D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.*, 16(4) :1097–1113, 1994. ISSN 0164-0925. URL http://doi.acm.org/10.1145/183432.183438.

[Ferrante et al.(1987)Ferrante, Ottenstein, and Warren] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3) :319–349, 1987. ISSN 0164-0925. URL http://doi.acm.org/10.1145/24039.24041.

[Harman and Danicic(1998)] M. Harman and S. Danicic. A new algorithm for slicing unstructured programs, 1998. URL http://citeseer.ist.psu.edu/harman98new.html.

[Horwitz et al.(1988)Horwitz, Reps, and Binkley] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23-7, pages 35–46, Atlanta, GA, June 1988. URL http://citeseer.nj.nec.com/horwitz90interprocedural.html.

[Kumar and Horwitz(2002)] S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *Fundamental Approaches to Software Engineering*, pages 96–112, 2002. URL http://citeseer.ist.psu.edu/kumar02better.html.

[Ottenstein and Ottenstein(1984)] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1 : Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-131-8. URL http://doi.acm.org/10.1145/800020.808263.

[Pacalet(2007)] A. Pacalet. Conception d'un outil de slicing. Technical report, INRIA, juin 2007. (Note : *document de travail en cours d'évolution*).

[Ranganath et al.(2004)Ranganath, Amtoft, Banerjee, Dwyer, and Hatcliff] V. Ranganath, T. Amtoft, A. Banerjee, M. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. Technical Report 8, SAnToS Lab., Kansas State University, 2004. URL citeseer.ist.psu.edu/ranganath05new.html. (Note : *Une version courte de ce document a été publiée à ESOP'2005*).