frama C

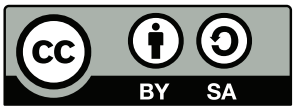
Software Analyzers

Slicing — Documentation technique

Documentation du greffon Slicing

Conception d'un outil de Slicing

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.

V4.0 - 2 novembre 2020 :
- Document sous licence Creative Commons “Attribution-ShareAlike 4.0 International” https://creativecommons.org/licenses/by/4.0/.

V3.0 - 3 décembre 2008 :
- corrections mineures,
- ajouts sur le *slicing* des déclarations globales,
- ajouts de la gestion des annotations.

V2.0 - 26 juin 2007 :
- extraction de la documentation du PDG,
- mise à jour vis à vis de l'état de l'outil,
- fusion de la documentation sur l'interprocédural,
- réorganisation de la partie sur l'interprocédural,
- mise en annexe des projets non implémentés.

V1.0 - 27 juin 2006 :
- ajout de la présentation du document,
- ajout du chapitre "Utilisation" (manuel),
- développement du calcul de certains ensembles sur les graphes de dépendances,
- mise à jour du chapitre sur l'interprocédural,
- ajout d'une conclusion résumant l'état d'avancement des travaux.

V0.2 - 27 avril 2006 :
- réorganisation de la définition des ensembles d'instructions,
- corrections diverses.

V0.1 - 22 février 2006 :
- bibliographie sur le *slicing*,
- analyse du document [Baudin(2004)],
- ce qu'on prévoit de faire,
- premiers éléments sur la construction de graphe de dépendances,
- intégration et tri des fichiers de l'ancien document (FLD),
- algorithme de calcul des dépendances de contrôle.

Table des matières

Chapitre 1

Introduction

L'objectif de ces travaux est de concevoir et développer un outil de *slicing* se basant sur le noyau FRAMA-C d'analyse statique de programme C en s'inspirant du module équivalent déjà développé sur la base de l'outil CAVEAT qui est décrit dans [Pacalet and Baudin(2005)].

1.1 État de l'art

Voyons tout d'abord ce que dit la littérature sur ce sujet afin de se positionner par rapport aux différentes méthodes.

On s'appuie ici sur la présentation des travaux sur le calcul de PDG effectué dans un autre document.

1.1.1 Origine

La technique consistant à réduire un logiciel - appelée *slicing* en anglais - a été beaucoup étudiée depuis son introduction par la thèse de [Weiser(1979)]. L'article [Weiser(1981)] rappelle que sa définition du *slicing* est la suivante :

Definition (slicing selon Weiser)

Le programme d'origine et le programme transformé doivent être identique si on les regarde à travers la fenêtre défini par le critère de *slicing* défini par un point de programme et une liste de variable. C'est à dire que la séquence de valeurs observées pour ces variables à ce point doivent être les mêmes.
De plus, le programme transformé doit être obtenu par suppression de certaines instructions du programme d'origine.

Cette définition a été remise en cause par la suite au motif que cette définition n'est pas suffisante pour traduire la compréhension intuitive que l'on a du *slicing*. On trouve l'exemple suivant dans [Kumar and Horwitz(2002)], mais cette critique revient a plusieurs reprises.

Exemple 1

P	P1	P2
a = 1 ;		a = 1 ;
b = 3 ;		b = 3 ;
c = a*b ;		
a = 2 ;	a = 2 ;	
b = 2 ;	b = 2 ;	
d = a+b ;	d = a+b ;	d = a+b ;

P1 est probablement ce qu'on attend du *slicing* de P si l'on s'intéresse à la variable d après la dernière instruction, mais P2 est aussi correct du point de vue de Weiser.

Il semble peu probable de construire un algorithme qui construit P2, mais on voit néanmoins que la définition est insuffisante pour spécifier complètement ce que doit être le résultat d'un *slicing*.

[Choi and Ferrante(1994)] formalise la définition de la façon suivante :

Definition (Correct-Executable-Slice)

Let P_{slice} be an executable slice of a program P_{org}, S be the set of statements in P_{slice} and $Proj_S(P_{org}, \sigma_0)$ be the projection of $Trace(P_{org}, \sigma_0)$ that contains only those statement traces produced by statements in S. P_{slice} is a correct executable slice of P_{org} if $Proj_S(P_{org}, \sigma_0)$ is identical to $Trace(P_{org}, \sigma_0)$.

Même si cette définition possède les mêmes inconvénients que celle d'origine, elle lui permet de faire des preuves de correction des ses algorithmes relativement à cette formalisation.

1.1.2 Graphes de dépendances

Comme on l'a vu dans [Pacalet(2007)] les graphes de dépendances sont à la base des calculs. [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 un *slicing non-executable* qui s'intéresse uniquement aux instructions qui influencent la valeur des variables, mais cette représentation sera également utilisée pour calculer un *slicing* exécutable.

Cette représentation, initialement intraprocédurale, a été étendue à l'interprocédurale dans [Horwitz et al.(1988)Horwitz, Reps, and Binkley] où elle porte le nom de SDG (*System Dependance Graph*). Dans ce papier, Susan Horwitz s'intéresse à un *slicing* ayant la définition suivante :

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.

Mais elle précise par ailleurs que x doit être défini ou utilisé en p. En effet, lorsque le graphe est construit, le *slicing* se résume à un problème d'accessibilité à un noeud, or un noeud représente un point de programme et ne contient que les relations concernant les variables qui y apparaissent.

Nous avons vu dans [Pacalet(2007)] que l'on peut lever cette limitation en gardant une correspondance entre les données à un point de programme et les noeud du graphe.

1.1.3 Slices vs. Chops

On a vu que le *slicing* s'intéresse à ce qui se passe avant d'arriver à un certain point.

La notion de *chopping* est similaire sauf qu'elle s'intéresse aux instructions qui vont être influencées par une certaine donnée à un certain point.

1.1.4 Critères de *slicing*

Nous n'avons pour l'instant parlé que du *slicing* dit **statique** qui s'intéresse à toutes les exécutions d'un programme, par opposition au *slicing* **dynamique** qui considère la projection du programme sur l'exécution d'un jeu d'entrée particulier, ou **quasi-statique** qui fixe uniquement une partie des entrées.

Une version plus générale qui inclus à la fois l'un et l'autre est appelé *slicing* **conditionnel** ([Fox et al.(2004)Fox, Danicic, Harman, and Hierons]). Il s'agit de spécifier le cas d'étude sous forme de des propriétés sur le jeu d'entrées (préconditions).

Par la suite, la notion de *slicing* **conditionnel arrière** a été introduit dans plusieurs buts différents :

- la première technique décrite dans [Comuzzi and Hart(1996)] et appelée p-slice, s'intéresse aux instructions qui participent à l'établissement de la postcondition,
- alors que la deuxième ([Fox et al.(2001)Fox, Harman, Hierons, and Danicic]) a un objectif un peu différent car vue comme une aide à la preuve permettant d'éliminer du programme tous les cas dans lesquels la propriété est vérifiée (automatiquement), pour ne garder que ceux pour lesquels le résultat n'est pas certain afin que l'utilisateur puisse se concentrer sur la vérification de cette partie.
  Dans [Harman et al.(2001a)Harman, Hierons, Fox, Danicic, and Howroyd], ces auteurs continuent les travaux précédents en combinant une propagation avant et une propagation arrière. Quelques exemples et la présentation d'un outil mettant en œuvre le travail de cette équipe est présenté dans [Daoudi et al.(2002)Daoudi, Ouarbya, Howroyd, Danicic, Marman, Fox, and Ward].
  Ils définissent leur objectif de la façon suivante :

Definition (pre-post conditioned slicing)

Statements are removed if they cannot lead to satisfaction of the negation of the post condition, when executed in an initial state which satisfies the pre-condition.

— une troisième approche est présentée très brièvement dans ([Chung et al.(2001)Chung, Lee, Yoon, and Kwon] : il s'agit également de spécifier le critère de slicing par une pré-post et de combiner une propagation avant et arrière de propriétés pour ne garder que ce qui participe à l'établissement de la post dans le contexte de la précondition.

Ces nouveaux types de *slicing* ne se basent plus uniquement sur des relations de dépendance, mais nécessitent différentes techniques complémentaires comme l'exécution symbolique ou encore le calcul de WP.

Dans le cadre de notre étude, il semble intéressant d'explorer cette voie étant donné que nous disposons (ou disposerons) de tels outils d'analyse dans FRAMA-C.

1.1.5 Transformations

Dans nos précédents travaux, nous avons exploré certaines transformations de programme comme la spécialisation de fonction (et donc la transformation des appels), et également la propagation de constantes.

Divers papiers, dont [Harman et al.(2001b)Harman, Hu, Munro, and Zhang], présentent un *slicing* qu'ils appellent **amorphe** dans lequel il s'agit de combiner des techniques de *slicing* et de transformation de programme.

[Ward(2002)] (pour une raison qui m'échappe) préfère parler de *slicing* **sémantique**

1.1.6 Outils

On peut citer deux outils qui mettent en œuvre cette technique sur les programmes C-ANSI :

— *unravel*, présenté dans [Lyle and Wallace(1997)], est disponible sur le web (cf. [[Unravel()]]) ;
— et le *Wisconsin Program-Slicing Tool* ([Reps(1993)]), est commercialisé par *Gramma-Tech* sous le nom *CodeSurfer* (cf. site web [[CodeSurfeur()]]).

Ces outils permettent tous les deux :

— de sélectionner les parties de code utiles au calcul d'une variable à un point de contrôle,
— de visualiser les instructions sélectionnées sur le code source de l'utilisateur,
— et de faire des opérations d'union et l'intersection entre plusieurs réductions.

1.1.7 Pour en savoir plus

Un état de l'art en 1995 est présenté dans [Tip(1995)].
Plus récemment, [Xu et al.(2005)Xu, Qian, Zhang, Wu, and Chen] font un tour très complet des évolutions jusqu'en 2005.

Par ailleurs, un site internet - dont l'adresse est donnée dans la bibliographie sous la référence [SlicingWebLinks()] - a tenu à jour une liste des projets et une bibliographie très impressionnante sur le *slicing* jusqu'en 2003, mais ce site ne semble plus maintenu.

1.2 Ce que l'on veux faire

Le document [Baudin(2004)] spécifie un outil d'aide à l'analyse d'impact après avoir analysé les besoins dans ce domaine. Le module de *slicing* est présenté comme étant un composant important de cet outil.

Résumons ici les caractéristiques principales de l'outil qui ont servies de spécification initiale :

- l'outil sert à construire un programme compilable à partir d'un programme source, les deux programmes ayant le même comportement vis à vis des critères fournis par l'utilisateur ;
- la construction de ce programme résultat est un processus interactif.
- **commandes** de l'outil
  - elles correspondent à des fonctions internes (en OCaml) et doivent permettre d'accéder aux différentes fonctionnalités.
    Elles pourront être utilisées directement, combinées dans des *scripts* pour effectuer des opérations plus évoluées ou encore servir à construire un protocole de communication avec une interface graphique ;
- commandes **d'interrogation**
  - elles servent à questionner le système. Elles ne doivent pas modifier l'état interne de l'outil. Même si elles effectuent certains calculs pour répondre, et qu'elles stockent ces résultats pour éviter un recalcul ultérieur, ceci doit être transparent pour la suite des traitements ;
- commandes dites **d'action**
  - elles doivent permettent de construire le résultat du *slicing*. Ces commandes doivent pouvoir s'enchaîner et se combiner ;
- l'outil maintient une **liste d'attente** des actions à effectuer. Ceci est utile car certains traitements peuvent être décomposés en plusieurs actions que l'utilisateur peut ensuite choisir d'appliquer ou non. Cela permet d'avoir un contrôle plus fin des calculs ;
- le programme résultant ne peut être généré que lorsque la liste d'attente est vide ;
- les traitements doivent être modulaires ;
- chaque fonction du programme source peut :
  - ne pas apparaître dans le résultat,
  - être simplement copiée dans le résultat,
  - correspondre à une ou plusieurs fonctions spécialisées du résultat.
- dans la représentation interne du résultat, les instructions sont associées à un marquage qui indique si elles doivent être cachées, ou sinon, la raison de leur présence, c'est-à-dire par exemple si elles participent au contrôle ou à la valeur d'une donnée sélectionnée, etc.

— identification d'instructions non significatives (ou superflues)
  — il s'agit d'être capable de distinguer ce que l'on doit ajouter lorsque l'on ne fait pas de spécialisation. Prenons par exemple le cas d'un appel $L : f(a, b)$; où seul a est nécessaire au calcul qui nous intéresse, mais que l'on ne souhaite pas spécialiser en $f(a)$;. Il faudra alors ajouter ce qui permet de calculer b en L aux instructions sélectionnées, mais en préservant l'information qu'elles sont *superflues* ;
— spécialisation du code
  — il s'agit d'être capable d'effectuer des transformations syntaxiques pour réduire le code au maximum. Par exemple, si après une instruction `y = x++;`, on n'a besoin que de `x`, il faut être capable de décomposer cette instruction afin de ne garder que `x++;`
  Ce type de transformation étant déjà faite en amont du module de *slicing*, le problème ne se pose plus pour ce type d'instruction, mais il existe encore pour la spécialisation des appels de fonction. En effet, si une fonction f prend, par exemple, deux arguments dont un seul sert à calculer ce qui nous intéresse, on veut être capable de construire une fonction f_1 qui ne prend qu'un seul argument, et transformer l'appel à f en un appel à f_1.
  Par ailleurs, on peut également envisager de transformer des instructions lorsque l'on connaît la valeur constante d'une variable.

1.3 Plan des documents

Ce document présente uniquement les principes de ce qui a été développé. Pour plus de détail sur l'implémentation ou les commandes disponibles, le lecteur est invité à consulter la documentation du code car elle sera toujours plus détaillée et à jour que ce rapport. Comme la boîte à outil de *slicing* est un greffon de Frama-C, son interface est définie dans le module `Db.Slicing`. Par ailleurs, l'introduction de la documentation interne du module donne une idée de l'architecture, et donne des points d'entrée.

Les principes de calcul des graphes de dépendances sur lequel s'appuie le *slicing* est présenté dans [Pacalet(2007)].

Le chapitre 2 présente la façon dont sont marquées les instructions d'une fonction pour effectuer une réduction intraprocédurale.

Puis, le chapitre 3 expose comment sont organisés les calculs pour faire une propagation des réductions à l'ensemble de l'application, et comment on se propose de spécialiser les fonctions.

Enfin, le chapitre 4 présente comment utiliser le module développé et propose donc une vision davantage dédiée à un utilisateur potentiel.

Ce rapport étant un document de travail, il va évoluer au fur et à mesure des réalisations :
— la version 1.0 du 27 juin 2006 présentait les développements en cours concernant les graphe de dépendance et le marquage d'une fonction. La présentation des travaux à faire pour la gestion d'un *slicing* interprocédural était moins détaillée car plus prospective. Suivait un chapitre contenant un manuel d'utilisation. Enfin, la conclusion présentait l'état courant de l'outil, et les perspectives d'évolution.

— dans la version 2.0 du 26 juin 2007, la partie sur le graphe de dépendance a été extraite car elle fait l'objet d'un document séparé. Le *slicing* interprocédural est cette fois présenté en détail. Les parties issues du document de spécification dont les idées n'ont finalement pas été retenues ont été déplacées en annexe D pour mémoire. La partie sur l'utilisation de l'outil a pour l'instant été conservée, même si elle n'est pas très spécifique à l'utilisation du greffon de *slicing*.

Fin 2008, ce document peut être considéré comme stable, car même si le développement évolue, il ne s'agit plus de modifier les fonctionnalité de bases, mais plutôt d'ajuster des points de détail pour améliorer la précision par exemple.

Chapitre 2

Réduction d'une fonction

2.1 Fonction spécialisée

On appelle **fonction spécialisée** la réduction d'une fonction source obtenue suite à l'application d'une ou plusieurs actions. Ce chapitre expose comment est représentée cette réduction, et comment elle est calculée.

La réduction d'une fonction doit distinguer les instructions visibles de celles qui ne le sont pas. Une information booléenne, attachée à chaque instruction, devrait donc suffire à indiquer si celle-ci appartient ou non à la fonction spécialisée, mais la mise en place d'un outil de navigation incite à enrichir les informations calculées. On décide donc d'avoir une annotation plus précise du flot de données, que l'on appelle **marquage**, pour préciser la raison de la présence ou de l'absence d'un certain élément. Cela peut permettre de visualiser l'impact des instructions d'une fonction sur un point de programme (contrôle, données utilisées, déclaration nécessaires, etc.)

Le document qui présente le calcul de PDG expose les éléments qui composent le graphe de dépendance. La plupart correspondent à une instruction. Quelques exceptions néanmoins :

- les éléments représentant les entrées/sorties d'une fonction ne sont pas associés à une instruction,
- le label d'une instruction est représenté par un élément en plus de ceux qui représentent l'instruction,
- un appel de fonction est représenté par plusieurs éléments.

Nous ne nous considérons pas, dans un premier temps, la spécialisation des fonctions appelées qui sera étudiée dans le chapitre 3. L'appel de fonction est donc simplement vu pour l'instant comme une instruction représentée par plusieurs éléments.

Par contre, nous distinguons la visibilité d'un label de celle de l'instruction associée. Dans la suite, le label sera souvent considéré comme une instruction à part entière.

On peut donc dire que la fonction spécialisée contient un **marquage** qui fait correspondre une **marque** aux instructions et labels d'une fonction.

2.2 Marquage

On calcule le marquage d'une fonction en réponse à une requête. Celle-ci se traduit en général en terme d'éléments du PDG.

Initialement, le marquage peut contenir le résultat de précédents calculs ou être nouvellement créé, c'est-à-dire vide.

Le calcul élémentaire est très simple : il consiste à parcourir le PDG, à calculer la marque pour chaque élément, et à l'associer à l'instruction ou au label correspondant en la combinant avec l'éventuelle valeur précédente.

Une idée de l'algorithme appliqué est présenté en A.2.

2.2.1 Passage à un point de programme

Une première requête consiste à marquer les éléments du flot qui servent à déterminer ce qui permet de passer à un point de programme.

La marque propagée s'appelle **marque de contrôle**, et se note mC.

Pour faire ce calcul, on propage la marque dans les dépendances de contrôle du point choisi, puis récursivement dans toutes les dépendances des points ainsi sélectionnés. Cela correspond à marquer mC tous les éléments de l'ensemble R_{L0}.

Exemple 2

mC	`x = y+1;`
	`a = 0;`
	`b = 10;`
	`...`	

mC	`if (x > 0) {`
	`...`
	`L: a += b;`
	`...`
	`}`
		Il s'agit ici de sélectionner ce qui contrôle le passage au point `L`. Le test `x>0` est donc marqué mC ainsi que le calcul de `x` dont dépend ce test.

2.2.2 Valeur d'une donnée

On peut également demander à sélectionner ce qui permet de calculer une donnée à un point de programme. On peut par exemple demander à ne garder que ce qui permet de calculer l'une des sorties d'une fonction.

La première étape consiste à trouver l'élément, ou les éléments, correspondant dans le graphe. Des éléments particuliers représentent les sorties de la fonction. Pour des calculs internes, on peut utiliser l'identification d'une affectation pour parler de la donnée affectée,

ou alors il faut disposer de l'état utilisé lors de la construction du graphe. On peut alors retrouver les éléments qui ont participé au calcul en un point de n'importe quelle donnée, du moins lorsque le point considéré se trouve dans sa portée.

La marque associée au calcul d'une donnée s'appelle **marque de valeur** et se note mV. Elle sert, comme son nom l'indique, à annoter les éléments qui participent au calcul de la valeur de la donnée demandée.

Les données qui permettent de calculer l'adresse de la case modifiée (partie gauche de l'affectation) sont annotés par une **marque d'adresse** (mA). Il s'agit par exemple du calcul de l'indice d'un élément de tableau ou d'un pointeur.

Les éléments qui permettent d'atteindre le point de programme sont marqués mC.

Exemple 3

mCA	`x = y+1;`
mV	`b = 10;`
	`...`
mC	`if (x > 0) {`
	`...`
	`L: t[x] = b;`
	`...`
	`}`

Il s'agit ici de sélectionner ce qui participe au calcul de l'instruction située au point `L`. Le test `x>0` est donc marqué mC ainsi que le calcul de `x` dont dépend ce test. La valeur de `b` participe au calcul de la partie droite et est donc marqué mV. La partie gauche dépend de `x` qui doit donc avoir la marque mA. On voit donc que `x` cumule deux informations et est donc marqué mCA.

2.2.3 Éléments superflus

On a vu que le marquage est relatif aux instructions, et que certaines instructions (appels de fonctions) sont représentées par plusieurs éléments du PDG. Si on ne souhaite pas décomposer les instructions (ie. spécialiser les appels de fonctions), il peut arriver qu'une même instruction corresponde à des éléments visibles et d'autres invisibles. Ces derniers, et leurs dépendances, sont alors marqués **superflus** (marque mS). Certaines instructions deviennent alors visibles alors qu'elles ne sont pas strictement nécessaires au calcul demandé.

Exemple 4

```
int G;
int f (int x, int y) {
  G = x + 1;
  return y + 2;
}
int g (void) {
  int a = 1;
  int b = a+1;
  return f (b, a);
}
```

Si l'utilisateur demande à sélectionner ce qui permet de calculer la valeur de retour de `g`, on n'aurait en fait besoin que de la valeur de `a`, mais comme on ne spécialise pas `f`, il faut aussi marquer l'instruction qui calcule `b` comme superflue.

2.2.4 Marques

En résumé, les marques possibles sont les suivantes :

mV :	marque de valeur
mC :	marque de contrôle
mA :	marque d'adresse
mS :	marque pour un élément superflu

Les marques mV, mC, mA peuvent se superposer lorsqu'un élément participe au calcul pour plusieurs raisons. On notera par exemple mVA la marque associée à un élément qui participe à la valeur et au calcul d'adresse. Par la suite, on appelle marque $mCAV$ toute marque de cette famille.

La marque mS est l'élément neutre de la combinaison.

Chapitre 3

Réduction interprocédurale

3.1 Objectif

Le filtrage élémentaire vu précédemment n'est qu'un préalable à la réduction d'une application car lorsqu'une fonction a été réduite, il faut également traiter ses appels pour rester cohérent, et on peut aussi vouloir réduire les fonctions appelées.

3.1.1 Spécification du problème

Pour étendre le marquage intraprocédural déjà effectué à un calcul interprocédural, une première approche serait de propager le marquage à travers les appels de fonctions, mais cela conduit soit à un trop grand nombre de fonctions spécialisées si l'on distingue tous les appels, soit à trop de perte de précision si l'on en fait l'union. Ce problème est exposé plus en détail en §3.3.3. L'objectif est donc d'obtenir un compromis acceptable et paramétrable.

Dès lors que l'on décide de pouvoir avoir plusieurs spécialisations d'une même fonction source dans l'application finale, le mécanisme permettant d'obtenir un résultat cohérent devient relativement complexe. Par exemple, lorsqu'on s'intéresse au calcul de certaines données d dans une fonction g, on a vu au chapitre précédent comment calculer le marquage des éléments du PDG pour pouvoir réduire g. Si g appelle h, et que toutes les sorties de h ne sont pas nécessaires au calcul de d dans g, on peut aussi vouloir demander la spécialisation de h. Par ailleurs, si g est appelée, il est possible de remplacer un ou plusieurs appels par un appel à g_1. Ce traitement doit être appliqué récursivement puisque dès que l'on modifie le marquage d'une fonction, on peut avoir besoin également de modifier les fonctions appelées et appelantes.

On décide donc de décomposer un traitement complet en différentes actions élémentaires parfaitement définies. Elles pourront ensuite être combinées automatiquement en fonction d'options de plus haut niveau.

3.1.2 Organisation des résultats

Le processus complet consiste à appliquer successivement un certain nombre de requêtes exprimées par l'utilisateur afin de construire une nouvelle application. Nous appellerons **projet courant** l'état de cette nouvelle application à un instant donné. Il est composé :

— d'un ensemble de résultats (initialement vide) sur les fonctions tels qu'ils ont été présentés en 2, c'est-à-dire de fonctions spécialisées ;
— et d'une liste d'actions qui sont encore à appliquer pour obtenir une application cohérente.

Lorsque l'utilisateur exprime des requêtes, elles sont traduites en **actions** qui sont rangées dans la liste des tâches en attente. Les actions s'appliquent en séquence, c'est-à-dire qu'on ne peut pas appliquer une action si la précédente n'est pas terminée. L'application d'une action peut éventuellement générer de nouvelles actions.

Une action peut dans certains cas être supprimée ou modifiée : ce point est abordé en §3.6.1.

A la fin de l'analyse, il n'y a plus d'action à appliquer, et l'application résultante peut être générée.

3.1.3 Appel de fonction

Lors du marquage intraprocédural, on obtient les marques strictement nécessaires au calcul demandé. On peut en particulier en extraire les signatures des appels de fonction. Pour chacun d'entre eux, dès lors qu'il y a au moins un élément visible, il va falloir décider quelle est la fonction appelée. On a vu par exemple qu'on peut choisir de ne pas spécialiser les appels de fonction : les entrées invisibles des appels devront alors être marqué comme superflus.

3.1.4 Propagation aux fonctions appelées

Pour se laisser la possibilité de faire évoluer l'outil, on décide d'offrir plusieurs possibilités :
— une spécialisation par appel : pour chaque appel, on détermine les sorties utiles, et on génère la fonction appelée correspondante (si elle n'existe pas déjà),
— pas de spécialisation de fonction : dès qu'on a besoin d'une des sorties d'une fonction, on laisse l'appel, et on a donc besoin de sélectionner tous les arguments d'appel,
— une seule spécialisation par fonction : regroupement de toutes les spécialisations nécessaires à l'application en une seule fonction,
— regroupement de fonctions spécialisées *a posteriori*, à la demande de l'utilisateur.

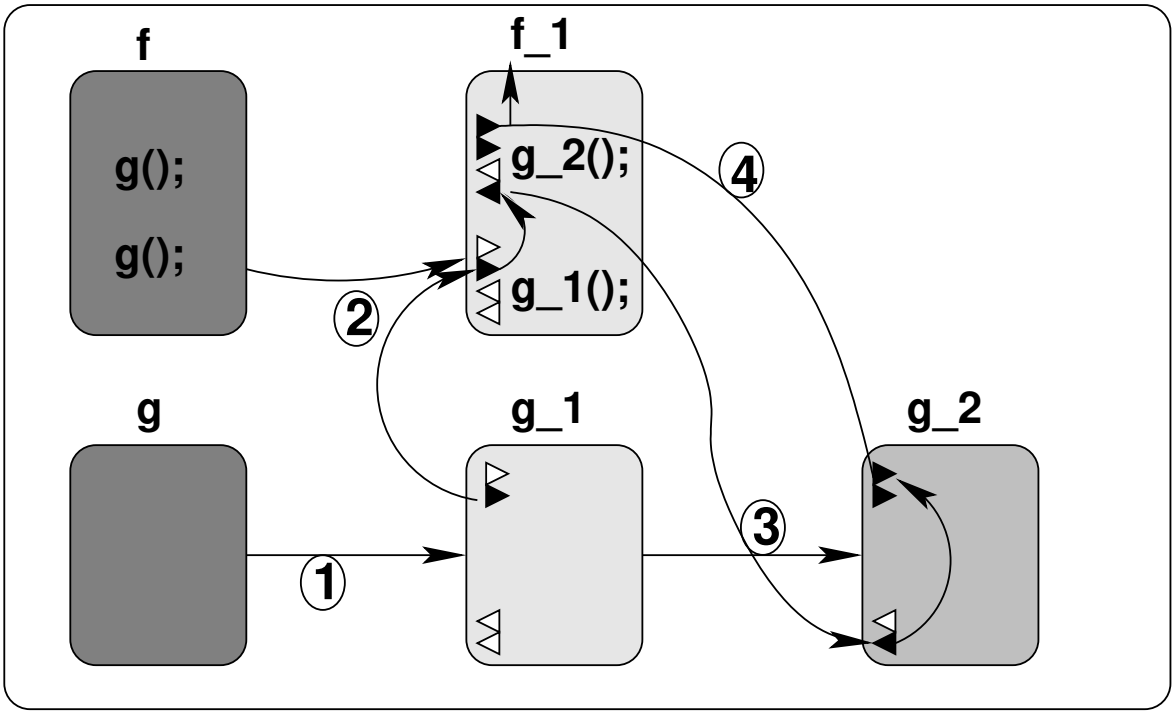
3.1.5 Propagation aux fonctions appelantes

Lorsqu'une fonction spécialisée a été calculée, on peut vouloir l'appeler à la place de la fonction initiale, et ainsi propager la réduction. Il faut pour cela pouvoir créer une requête qui désigne le point d'appel, et la fonction spécialisée à utiliser. Cette requête revient à s'intéresser au calcul des entrées visibles de la spécialisation au dessus du point d'appel.

L'application d'une telle requête peut à son tour générer une demande de spécialisation, éventuellement de la même fonction comme dans l'exemple ci-dessous.

Exemple 5

La figure suivante montre un exemple de différentes étapes de la transformation des appels à g dans f par des appels à la fonction filtrée g_1, et comment on peut envisager de propager l'information.



1. calcul de la fonction g_1 : seule la seconde entrée est visible, les sorties ne sont pas visibles ;
2. remplacement du second appel à g dans f par un appel à g_1, et sélection des éléments qui permettent de calculer l'entrée visible. On voit que ce calcul nécessite de calculer la seconde sortie du premier appel à g_1. Il faut donc calculer une seconde fonction g_2.
3. calcul de la fonction g_2 : elle doit avoir les mêmes éléments que g_1 et calculer en plus sa seconde sortie. On voit ce cela nécessite la visibilité de la première entrée.
4. sélection de ce qui permet de calculer la première entrée de g_2 dans f.

3.1.6 Sélection des appels d'une fonction

En plus des critères de *slicing* déjà vus qui consiste à sélectionner une donnée à un point de programme, l'aspect interprocédural conduit à définir un nouveau critère qui permet de ne garder que ce qui permet d'appeler une fonction donnée, et de calculer les contextes d'appels. La fonction peut être :

- une fonction externe dont on n'a pas le code,
- une fonction présente dans le code source,
- une fonction issue d'un précédent filtrage,

et on peut choisir de sélectionner :

- uniquement ce qui permet d'atteindre les points de programme des appels à cette fonction,
- ou bien également ce qui permet de calculer les entrées.

Ce filtrage peut par exemple être utilisé pour réduire une application afin de permettre une analyse des fuites mémoire. Il suffit pour cela de sélectionner les fonctions d'allocation (`alloc`, `malloc`, etc.) et la fonction de libération (`free`).

3.2 Principe et notations

On rappelle ici les aspects du calcul intraprocédural, et diverses notations, qui vont être utilisées par la suite. On s'appuie en grande partie sur la représentation des entrées/sortie et des appels de fonction dans le PDG présenté dans [Pacalet(2007)].

3.2.1 Marques élémentaires

On considère qu'on a un ensemble de marques élémentaires, non détaillées ici, muni :
- d'un ordre partiel ($\leq$),
- d'une opération d'union ($+$),
- d'un plus petit élément ($\bot_m$) utilisé pour marquer les éléments invisibles,
- et d'un plus grand élément ($\top_m$) correspondant au marquage des élément de la fonction source.

On considère de plus un élément appelé *Spare* tel que :
- $\forall m.m \neq \bot_m \Rightarrow m = Spare \vee Spare \leq m$

On a donc :

$$\forall m.m \neq \bot_m \Rightarrow m + Spare = m$$

Cette marque est donc la plus petite qui rende un élément visible. Elle est utilisée pour marquer les élément dit *superflus*, c'est-à-dire ceux qui ne seraient pas nécessaire si la réduction était plus précise.

3.2.2 Marquage

On a vu au chapitre 2 que l'on dispose d'une procédure permettant de marquer une instruction et de la propager selon les dépendances (PDG).

Au cours de ce calcul, les relations de dépendances entre les entrées et les sorties des fonctions appelées sont données par la spécification [1].

Dans le résultat obtenu, lorsqu'un élément de fonction $e1$ est utilisé par un élément de fonction $e2$, la marque $m1$ associée à $e1$ est supérieure ou égale à la marque $m2$ associée à $e2$ car $m1$ est l'union des marques de tous les éléments qui dépendent de $e1$.

3.2.3 Fonctions

Chaque fonction source peut être spécialisée une ou plusieurs fois. Au niveau *intraprocédural*, cela signifie qu'on lui attribue un certain marquage.

Dans la suite, on note :

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.

— $f, g, h, ...$ une fonction quelconque (source ou spécialisée),
— une fonction source est munie d'un indice 0 : $f_0, g_0, h_0, ...$
— les fonctions spécialisées ont un autre indice : $f_i, g_j, h_k, ...$ (sauf mention contraire, quand on précise l'indice, on suppose donc qu'il est différent de 0).

3.2.4 Entrées/sorties

Le graphe de dépendance contient des éléments particuliers qui représentent les entrées et les sorties de la fonction. On note $InOut(f)$ l'ensemble de ces éléments pour la fonction f.

On ne précise pas de quelle fonction f il s'agit, car on considère le même ensemble d'entrées/-sorties pour toutes les spécialisations :

$$\forall i. InOut(f_0) = InOut(f_i)$$

3.2.5 Signature

On appelle **signature**, et on note *sig*, une fonction qui associe une marque aux éléments d'un ensemble d'entrées/sorties.

On définit $\top_s$ qui représente une signature dont tous les éléments ont une marque $\top_m$:

$$sig = \top_s \equiv \forall e \in dom(sig). sig(e) = \top_m$$

et de même, $\bot_s$ qui représente une signature dont tous les éléments ont une marque $\bot_m$:

$$sig = \bot_s \equiv \forall e \in dom(sig). sig(e) = \bot_m$$

On définit $sig_f(f)$ la **signature d'une fonction** f telle que :
— $dom(sig_f(f)) = InOut(f)$
— $sig_f(f_0) = \top_s$
— $sig_f(f_i)(e)$ est la marque attribuée à e dans f_i.

On notera parfois $inSig_f(f)$ et $outSig_f(f)$ les fonctions dont le domaine est réduit aux entrées ou aux sorties.

3.2.6 Appel de fonction

Comme nous parlons ici de la partie interprocédurale du traitement, on s'intéresse principalement aux appels de fonction. On considère que l'on sait identifier de manière unique un appel de fonction c, (par exemple par l'identification de la fonction appelante et un élément du PDG, ou un numéro d'ordre). Dans la suite, pour préciser le nom de la fonction appelée, on notera c_g un appel à une fonction g.

On note $call(f)$ l'ensemble des appels de fonction de f. On remarque que l'on ne précise pas la spécialisation, car :

$$\forall i. call(f_i) = call(f_0)$$

Pour chaque appel de fonction c_g de f_i, le graphe de dépendance contient des éléments qui représentent les entrées/sorties de la fonction appelée.

On définit la **signature d'un appel** de fonction c dans la fonction f_i, et on note $sig_c(c, f_i)$, la fonction qui donne les marques de ces éléments dans f_i.

3.3 Calcul interprocédural

3.3.1 Gestion des fonctions

Au niveau *intraprocédural*, une fonction spécialisée est caractérisée par un certain marquage de ses éléments. En interprocédural, on s'intéresse à la propagation des marques aux appels de fonction. On se propose donc d'ajouter aux fonctions spécialisées une fonction $\mathcal{C}all$, décrite en §3.3.2, qui associe, à chaque appel de fonction, l'identification de la fonction à appeler. L'objectif est de remplacer certains appels par des appels à des fonctions spécialisées.

La réduction d'une application consiste à construire un **projet** qui contient une liste de fonction, initialisée à la liste des fonctions source, et complétée au cours de l'étude par les fonctions spécialisées calculées.

L'objectif est d'obtenir un projet cohérent, tel que défini en §3.3.5.

3.3.2 Appels de fonction

On appelle $\mathcal{C}all(f_i)$ la fonction qui fait correspondre chaque appel de f_i à une fonction appelée. $\mathcal{C}all(f_i)(c)$ donne donc la fonction appelée par f_i pour l'appel c.

On note $\mathcal{C}all(f_i)(c) = \bot_c$ quand l'appel est invisible.

Par ailleurs, $\mathcal{C}all(f_i)(c) = \top_c$ signifie que l'appel n'est pas encore attribué, c'est-à-dire que l'on n'a pas encore choisi la fonction à appeler.

Si la fonction appelée est déterminée par l'accès à un pointeur de fonction, il n'est pris en considération dans $\mathcal{C}all$ que si on connaît statiquement la fonction appelée à partir de l'analyse de valeur. Dans les autres cas, on laissera l'appel tel qu'il est dans la fonction source.

Les fonctions source appellent forcement des fonctions source :

$$\forall c \in call(f), \exists g. \mathcal{C}all(f_0)(c) = g_0$$

Quand l'appel est attribué, il correspond forcement à une spécialisation de la fonction initialement appelée :

$$\forall c \in call(f), \mathcal{C}all(f_0)(c) = g_0 \Rightarrow \forall f_i.(\mathcal{C}all(f_i)(c) = \bot_c \vee \mathcal{C}all(f_i)(c) = \top_c \vee \exists j. \mathcal{C}all(f_i)(c) = g_j)$$

Par ailleurs, si $\mathcal{C}all(f_i)(c) = g_j$ on veut que la signature de la fonction appelée $sig_f(g_j)$ soit *compatible* à la signature de l'appel $sig_c(c, f_i)$. Voyons maintenant ce que cela veut dire...

3.3.3 Problème de compatibilité des signatures

La première idée qui vient lorsque l'on souhaite propager le marquage aux fonctions appelées est d'appliquer les mêmes règles de propagation dans les dépendances que pour le calcul intraprocédural. Mais lorsque l'on souhaite utiliser une même fonction spécialisée dans différents contextes, cela conduit à avoir une trop grande perte de précision. On veut par exemple pouvoir marquer comme *Spare* les entrées qui ne sont pas réellement utilisées pour un certain appel, même si elles portent d'autres marques par ailleurs.

Par exemple, dans le cas suivant :

```
int X, Y;

void g (int x, int y) {
  X = x; Y = y;
  }
```

```
int f_a (int x_a,
         int y_a) {
  g (x_a, y_a);
  return X;
  }
```

```
int f_b (int x_b,
         int y_b) {
  g (x_b, y_b);
  return Y;
  }
```

si l'utilisateur demande le marquage des sorties 0 de `f_a` et `f_b`, et la construction d'une seule spécialisation pour `g`, on aimerait que `x_a` et `y_b` soient marquées *Spare*

Pour obtenir un tel comportement, on décide d'étendre le marquage comme suit.

3.3.4 Couple de marques

Pour garder un maximum de précision, il faut distinguer les sélections réellement effectuées par l'utilisateur des marques introduites par une approximation. Pour cela, on marque chaque élément d'une spécialisation non plus par une seule marque, mais par un couple $< m_1, m_2 >$ où :

- la marque m_1 correspond à la propagation d'une sélection par l'utilisateur dans la fonction d'origine et dans ses appelants, MAIS on ne la propage pas directement dans les fonctions appelées car cela conduit à avoir une trop grande perte de précision. Il faut noter que m_1 ne peut en principe pas être *Spare*, sauf si l'utilisateur le demande explicitement.
- la marque m_2 correspond à la propagation d'une marque m_1 d'un appelant. Donc, lorsque la sortie d'un appel a une marque m_1, la sortie correspondante de la fonction appelée est marquée m_2. Par ailleurs, si une fonction a une entrée marquée $m =< m_1, m_2 >$ avec $m_2 \neq \bot_m$, ses appelants doivent propager $m' =< m_1, Spare >$ sur l'entrée correspondante. Attention, comme on le verra plus loin, cela ne veut pas dire que les entrées des appelants ont la marque m', car il faut combiner cette marque avec les marques de l'appelante.

Du point de vue de l'utilisateur, la marque associée à un élément est l'union de m_1 et de m_2.

Dans l'exemple précédent, les entrées x et y de g seront marquées en m_2, et cela conduira bien à marquer *Spare* les entrées inutiles dans f_a et f_b.

3.3.5 Cohérence du projet

Avant de voir comment calculer le résultat, voyons les propriétés que doit avoir un marquage final pour être cohérent. Les éléments qui nous intéressent sont les signatures des fonctions et des appels.

Nous ne parlons ci-dessous que des marques propagées, sachant qu'en plus, chacune peut contenir aussi une marque mise manuellement par l'utilisateur.

- les marques des sorties d'un appel (c_g, f_i) ne dépendent que du contexte d'appel, c'est-à-dire des marques de leurs dépendances dans f_i. Néanmoins, il semble qu'une marque

m_2 ne puisse venir que de marques m_2 sur les sorties de f_i, sauf la marque *Spare* qui peut venir d'entrées d'appels de fonction.

— les marques des sorties d'une fonction spécialisée sont déterminées par les marques des sorties des appels à cette fonction. On ne peut avoir $m_1 \neq \perp_m$ que si l'utilisateur a placé explicitement une marques sur la sortie. La marque m_2 est l'union des marques m_1 et m_2 des sorties des appels.

— les marques des entrées d'une fonction sont uniquement déterminées par la propagation des marques des autres éléments de la fonction. Une marque m_1 provient nécessairement d'une sélection utilisateur dans la fonction considérée ou une de ses fonctions appelées.

— les marques des entrées d'un appel (c_g, f_i) sont une combinaison des marques des entrées de g et des sorties de (c_g, f_i) Si l'on remplace la fonction appelée, il faut recalculer ces marques d'entrées.

En résumé, considérons une entrée e et une sortie s d'une fonction spécialisée g_i, on note $m(e) = sig_f(g_i)(e)$, $m(s) = sig_f(g_i)(s)$.

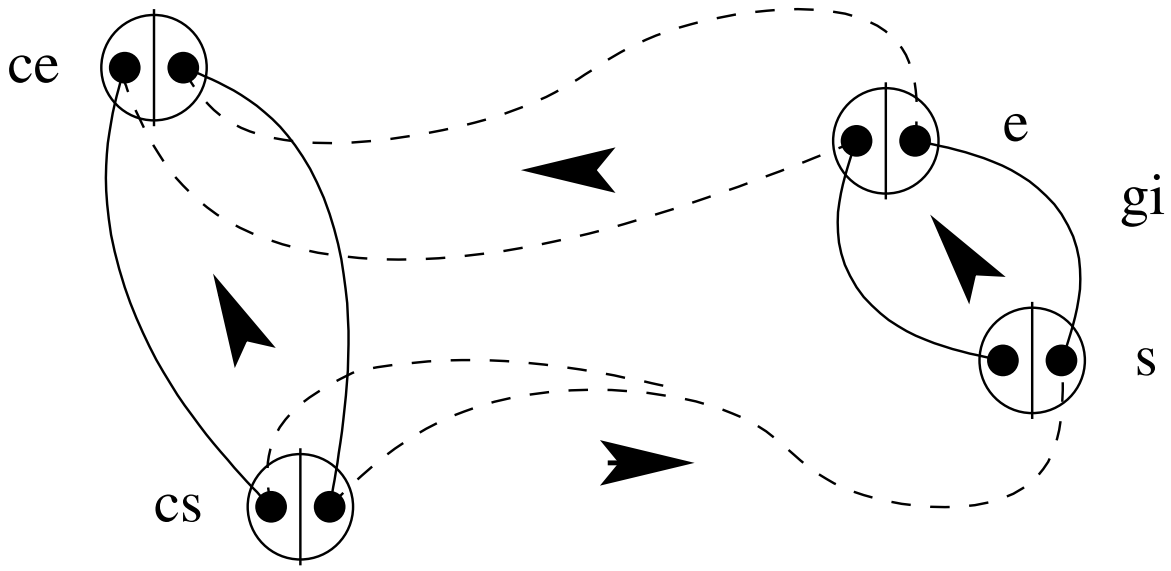


Figure 3.1 – *Propagation des marques dans les appels de fonction*

Soit c un appel quelconque à cette fonction, on note $m(ce) = sig_c(c)(e)$ et $m(cs) = sig_c(c)(s)$. Pour avoir un projet cohérent, on doit avoir les propriétés suivantes :

— $m_2(s) \geq m_1(cs) + m_2(cs)$
— $m_1(e) \leq m_1(ce)$
— $m_2(e) \neq \perp_m \Rightarrow m_1(ce) \geq Spare$

Par ailleurs, si la sortie s dépend de l'entrée e :

— $m_1(cs) \leq m_2(s) \leq m_2(e) \leq m_2(ce)$
— $m_1(cs) \leq m_1(ce)$
— $m_1(s) \leq m_1(e) \leq m_1(ce)$

On a vu que le calcul du résultat final se décompose en actions élémentaires. Entre deux actions, on impose que le projet soit **partiellement cohérent**, c'est-à-dire que pour tous $\mathcal{C}all(f_i)(c)$:

- $\mathcal{C}all(f_i)(c) = \bot_c \iff sig_c(c, f_i) = \bot_s$
- $\mathcal{C}all(f_i)(c) = \top_c$ et il doit y avoir une action en attente pour attribuer l'appel,
- $\mathcal{C}all(f_i)(c) = g_j$ et :
  - g_j doit exister et avoir une signature compatible à l'appel, ou une action de modification en attente (car dans ce cas, on ne sait pas forcement évaluer la compatibilité des signatures).
  - ou g_j n'existe pas encore, mais fait l'objet d'une action de construction en attente.

Lorsque la liste des actions est vide, le projet doit être **cohérent** c'est-à-dire que tous les appels de fonction doivent être affectés à une fonction ayant une signature compatible avec celle de l'appel.

3.4 Actions pour le calcul interprocédural

La difficulté du traitement est d'enchaîner correctement les actions pour obtenir une application cohérente. De plus, pour avoir une boite à outils suffisamment flexible, les actions doivent correspondre à des traitements suffisamment élémentaires, et parfaitement spécifiés (surtout leur comportement vis à vis des différentes options).

On décide donc que toute action doit être décomposée et traduite en une séquence d'**actions élémentaires** définies ci-dessous. Elles doivent être simples, et le moins paramétrables possible, la flexibilité venant plutôt de la traduction d'une action de haut niveau en actions élémentaires. Ces dernières ne sont donc pas toutes accessibles par l'utilisateur, mais peuvent être vues comme des étapes de calcul qui seront combinées par la suite en fonction des besoins.

L'application d'une action élémentaire :

- peut éventuellement générer de nouvelles actions,
- mais ne peut créer ou modifier qu'une seule fonction (ou aucune).

Cette partie présente donc tout d'abord les actions élémentaires, et le détail de l'application de chacune d'entre elles. Puis, on verra que la génération de ces actions dépend du mode de fonctionnement choisi. Les modes proposés sont exposés en §3.4.4.

Les actions sont résumées sous forme de fiches signalétiques, en annexe C.

3.4.1 Création une spécialisation : *NewSlice*

On appelle $f_i = NewSlice(f_0)$ l'action qui permet de construire une spécialisation f_i de f_0 . Initialement, toutes les marques sont mises à $\bot_m$ et la fonction $\mathcal{C}all(f_i)(c) = \top_c$. Les modifications peuvent ensuite être faite à l'aide des actions présentées ci-dessous. Cela permet d'avoir un traitement cohérent.

3.4.2 Modification d'une spécialisation

Les actions suivantes permettent de modifier la marquage une spécialisation Elles ne créent pas de nouvelles fonctions.

Attention : pour l'instant, les actions **ajoutent** des marques. Veut-on pouvoir réduire une spécialisation ?

Dans tous les cas, lorsque le marquage d'une fonction est modifié, il faut ensuite gérer ses appels de fonction. Toutes les actions de modification ci-dessous génèrent donc une requête *ExamineCalls* qui permet d'effectuer cette tâche. Son application est présentée en §3.4.2.

Ajout d'une marque utilisateur : *AddUserMark*

L'action $AddUserMark(f_i, (e, m)$ permet d'ajouter et de propager une marque sur n'importe quel élément de f_i. Il s'agit d'une marque utilisateur, elle est donc ajoutée en m_1.

Propagation d'une marque de sortie : *AddOutputMarks*

L'action $AddOutputMarks(f_i, outSig_c)$ est générée lorsque f_i est appelée, mais qu'elle ne calcule pas assez de choses. Pour chaque sortie s, si $outSig_c(s) =< m_1, m_2 >$, il faut ajouter $< \bot_m, m_1 + m_2 >$ à $outSig_f(f_i)(s)$ et propager la nouvelle marque dans f_i.

Marquage des entrées d'un appel : *ModifCallInputs*

L'action $ModifCallInputs(c_g, f_i)$ est générée lorsque les marques d'entrée d'un appel $g_j = \mathcal{C}all(f_i)(c_g)$ dans f_i sont insuffisantes pour la fonction appelée. Pour chaque entrée e de l'appel, on commence par recalculer sa marque en fonction des marques de ses dépendances. Cela permet d'obtenir un résultat correct même si on avait besoin de plus d'entrée avant d'appeler g_j. Puis, si $inSig_f(g_j)(e) =< m_1, m_2 >$, il faut ajouter $< m_1, Spare >$ à $inSig_c(c_g, f_i)$ si $m_2 \neq \bot_m$ et $< m_1, \bot_m >$ sinon, et propager la nouvelle marque ainsi obtenue dans f_i.

Gestion des appels : *ExamineCalls*

L'action $ExamineCalls(f_i)$ est automatiquement appliquée après toute modification du marquage de la fonction f_i car il faut examiner $\mathcal{C}all(f_i)$ afin d'en vérifier la cohérence, c'est-à-dire pour voir si les fonctions appelées conviennent toujours.

Pour chaque appel, on regarde donc :

- si $outSig_c(c, f_i) = \bot_s$ (on n'utilise pas les sorties de l'appel de fonction), alors $\mathcal{C}all(f_i)(c) = \bot_c$,
- sinon (certaines sorties sont marquées) :
  - si $\mathcal{C}all(f_i)(c) = \bot_c$: rien n'était appelé précédemment, il faire faire comme si $\mathcal{C}all(f_i)(c) = \top_c$ (voir ci-dessous),
  - si $\mathcal{C}all(f_i)(c) = \top_c$: aucune fonction n'a encore été attribuée à cet appel. Une action *ChooseCall*est chargée d'affecter une fonction à cet appel (cf. §3.4.3).
  - si $\mathcal{C}all(f_i)(c) = g_j$: il faut comparer $outSig_c(c, f_i)$ à $outsigf(g_j)$ comme nous allons le voir.

Pour que la fonction appelée g_j convienne, il faut qu'elle calcule au moins toutes les sorties nécessaires à cet appel. C'est-à-dire que si une sortie de l'appel est marquée $< m_1, m_2 >$, et que la sortie correspondante de g_j est marquée $< m'_1, m'_2 >$, on doit avoir :

$$m'_2 \geq m_1 + m_2$$

Si ce n'est pas le cas, une action *MissingOutputs* est générée (cf. §3.4.3). On pourrait générer directement $AddOutputMarks(g_j, outSig_c(c, f_i)$ mais ça ne permettrait pas de choisir une autre fonction g plutôt que d'étendre g_j.

Par ailleurs, *ExamineCalls* s'occupe également de la propagation des marques aux appelants, c'est-à-dire que si f_i est appelée par h_k et que les marques de ses entrées ont été modifiées, il faut les propager aux entrées des appels. Pour cela, on génère une action *MissingInputs*(cf. §3.4.3). On pourrait directement générer *ModifCallInputs*(c_f, h_k), mais cela ne permettrait pas de modifier l'appel de fonction pour choisir d'appeler une autre spécialisation de f dans h_k.

Les étapes de l'attribution, puis de la modification des appels de fonction peuvent sembler complexes au premier abord, d'autant plus que leur enchaînement dépend d'un mode de fonctionnement qui sera explicité en §3.4.4 et éventuellement des interventions ponctuelles de l'utilisateur. La figure 3.4.2 tente de résumer ce processus.

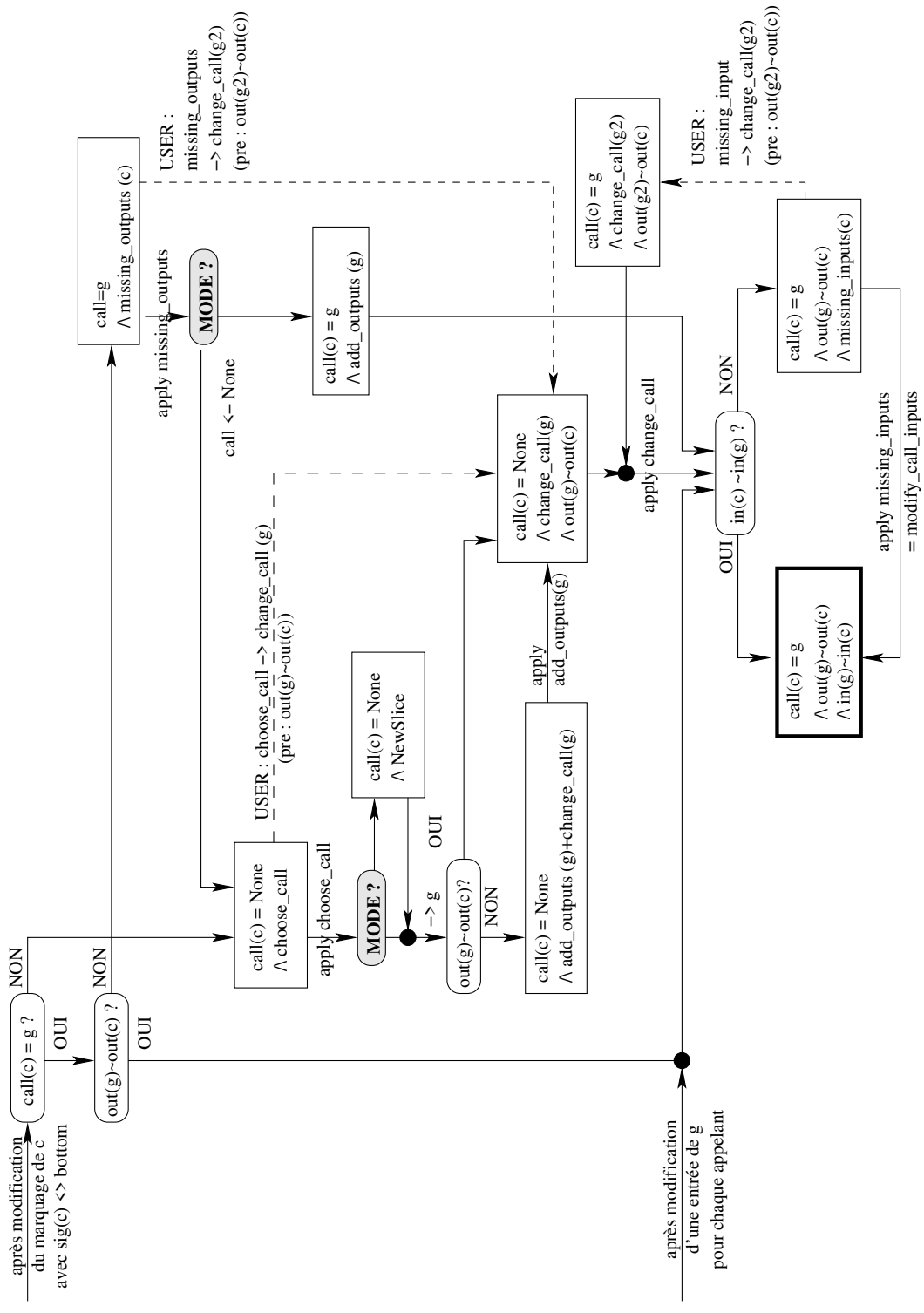


Fig. 3.1 : Gestion d'un appel c à une fonction g
suite à la modification des marques de la signature de c dans la fonction appelante ou à celle des entrées de g.

3.4.3 Affecter ou modifier un appel de fonction

On a vu que lors de la création ou la modification du marquage d'une fonction f_i, des actions *ChooseCall*(c, f_i), *MissingOutputs*(c, f_i) ou *MissingInputs*(c, f_i) pouvaient être générées lorsqu'un appel c doit être attribué ou modifié. Dans tous les cas, on se pose la question de la fonction à appeler car soit on ne l'a pas encore choisie, soit elle ne convient plus.

Attribution d'un appel : *ChooseCall*

Une action *ChooseCall*(c, f_i) peut-être générée (selon le mode choisi) lorsque $\mathcal{C}all(f_i)(c)$ doit être modifié car le marquage de f_i a changé : il s'agit de trouver une fonction g_k pour remplacer l'ancienne valeur de $\mathcal{C}all(f_i)(c)$ qui pouvait être $\top_c$, $\bot_c$ ou encore g_j, mais cette dernière ne convient plus.

Lors de l'application de cette action, outre le mode de fonctionnement, le critère principal de choix est le marquage des sorties dans l'appel ($outSig_c(c, f_i)$) qui est confronté au marquage des spécialisations de g déjà calculées, en appliquant éventuellement un traitement particulier à g_j, la fonction initialement appelée.

En fonction du mode :

- soit on trouve une g_k qui convient,
- soit on trouve une g_k que l'on souhaite utiliser, mais qui ne calcule pas assez de choses : il faut donc créer *AddOutputMarks*$(g_k, sig_c(c, f_i))$,
- soit on ne trouve pas de spécialisation existante et on lance alors la création d'une nouvelle spécialisation soit en partant de g_0, soit en partant de g_j (à voir...)

Dans tous les cas, après avoir choisi g_k, on applique une action *ChangeCall*(cf. §3.4.3).

Action *MissingInputs*

On rappelle qu'une action *MissingInputs*(c, f_i) est générée lorsque la fonction g_j attribuée à $\mathcal{C}all(f_i)(c) = g_j$ a été modifiée, et qu'elle nécessite le marquage d'entrées qui ne le sont pas $sig_c(c, f_i)$.

En fonction du mode de fonctionnement, une telle action peut conduire à :

- la modification du marquage de f_i, c'est-à-dire l'application de *ModifCallInputs*(c, f_i),
- le choix d'une autre fonction, c'est-à-dire l'application de *ChooseCall*(c, f_i) (dont le résultat dépend du mode courant),
- une transformation manuelle par l'utilisateur de cette action par un *ChangeCall*(c, f_i, g_k) où g_k doit nécessairement calculer suffisamment de sorties.

Action *MissingOutputs*

L'action *MissingOutputs*(c, f_i) est générée lorsque le marquage de f_i a été modifié et que la fonction g_j attribuée à $\mathcal{C}all(f_i)(c) = g_j$ ne calcule pas suffisamment de sorties pour le nouveau marquage.

En fonction du mode de fonctionnement, une telle action peut conduire à :

- la modification du marquage de g_j, c'est-à-dire l'application de *AddOutputMarks*$(g_j, sig_c(c, f_i))$,

— le choix d'une autre fonction, c'est-à-dire l'application de $\mathit{ChooseCall}(c, f_i)$ (dont le résultat dépend du mode courant),
— une transformation manuelle par l'utilisateur de cette action par un $\mathit{ChangeCall}(c, f_i, g_k)$ où g_k doit nécessairement calculer suffisamment de sorties.

Changement d'un appel : *ChangeCall*

L'action $\mathit{ChangeCall}(c, f_i, g_j)$ permet de transformer f_i pour avoir $\mathcal{C}all(f_i)(c) = g_j$. Elle ne peut être appliqué que si g_j calcule bien toutes les sorties nécessaire à l'appel c dans f_i. Il faut donc appliquer *AddOutputMarks* à g_j avant d'appliquer *ChangeCall* s'il en manque.

L'application de $\mathit{ChangeCall}(c, f_i, g_j)$ consiste à modifier $\mathcal{C}all(f_i)(c)$, mais aussi à appliquer $\mathit{ModifCallInputs}(c, f_i)$ dans le cas où la signature de g_j nécessite des entrées qui ne sont pas visibles dans $sig_c(c, f_i)$.

3.4.4 Modes de fonctionnement

La génération de ces actions, ainsi que l'effet de leur application sur le projet, dépend du choix d'un mode de fonctionnement qui décide de la précision des spécialisations. Ce mode peut offrir de nombreuses possibilités, mais on se limite dans un premier temps à quatre comportements :

1. pas de spécialisation des appels de fonction (**DontSliceCalls**) (cf. §3.4.4) ;
2. pas de spécialisation, mais propagation du marquage aux fonctions appelées (**PropagateMarksOnly**)(cf. §3.4.4) ;
3. le moins de spécialisations possibles (**MinimizeNbSlice**) (cf. §3.4.4) : l'outil ne crée pas plus d'une spécialisation par fonction ;
4. spécialisations les plus précises possible (**PreciseSlices**) (cf. §3.4.4) : on regroupe néanmoins les fonctions ayant la même visibilité, même si le marquage n'est pas le même.

Pour plus de flexibilité, du point de vue de la boîte à outil décrite ici, on suppose que le mode peut être changé à tout moment.

Réduction avec *DontSliceCalls*

Il s'agit du choix qui permet d'utiliser le marquage intraprocédural sans propagation aux fonctions appelées. Il peut être utilisé si l'on s'intéresse aux liens de dépendance dans une fonction, sans s'intéresser au reste de l'application. Tous les appels visibles sont donc attribués aux fonctions source :

$$\forall c. \exists o. \mathit{outSig}_c(c, f_i)(o) \neq \bot_m \Rightarrow \mathcal{C}all(f_i)(c) = \mathcal{C}all(f_0)(c)$$

Dans ce cas, il n'est pas utile de générer d'action *ChooseCall*, et les *MissingInputs* (marquage à *Spare* des entrées non utilisées) sont directement appliquées. Par ailleurs, il n'y a pas non plus d'actions *MissingOutputs* puisque les fonctions source calculent toutes les sorties.

Si un autre mode a été utilisé, il est possible néanmoins d'être amené à appliquer les actions suivantes :

— $\mathit{ChooseCall}(c, f_i)$ est traduite en $\mathit{ChangeCall}(c, f_i, g_0)$,
— $\mathit{MissingInputs}(c, f_i)$ en $\mathit{ModifCallInputs}(c, f_i)$
— $\mathit{MissingOutputs}(c, f_i)$ en *AddOutputMarks* ? ou $\mathit{ChangeCall}(c, f_i, g_0)$?

Réduction avec *PropagateMarksOnly*

Ce mode est à utiliser si l'on veut voir la propagation des dépendances dans l'ensemble de l'application sans calculer de spécialisations. Il est donc très semblable au mode précédent sauf que l'on calcule un marquage également pour les fonctions appelées. Néanmoins, il ne s'agit pas d'une réduction à proprement parlé puisque tout reste visible.

Réduction avec *MinimizeNbSlice*

Lorsque l'on applique une action *ChooseCall* sur un appel à une fonction g et qu'il n'y a pas encore de spécialisation, on en crée une nouvelle.

S'il y a une spécialisation g_j (et une seule), elle est choisie pour être appelée, indépendamment de son marquage.

Si plusieurs g_j existent, c'est *la plus proche*[2] qui doit être choisie.

S'il manque des marques aux sorties de la fonction choisie, il faut les lui ajouter. Pour cela, il n'y a pas de génération de *MissingOutputs* puisqu'on ne souhaite avoir qu'une seule spécialisation : elle est directement remplacée par *AddOutputMarks*.

Enfin, l'action *ChangeCall* permet d'appeler la fonction choisie. C'est cette dernière action qui s'occupe ensuite de propager le marquage des entrées dans f_i en appliquant automatiquement le *ModifCallInputs*(c, f_i).

Réduction avec *PreciseSlices*

Dans ce mode, lors de l'application de *ChooseCall*(c, f_i), le choix de la fonction à appeler s'effectue de la façon suivante :

- on sélectionne les spécialisations qui ont la même visibilité que $outSig_c(c, f_i)$ (même si elles n'ont pas exactement le même marquage) ;
- s'il n'y en a pas, on en crée une nouvelle ;
- s'il n'y en a qu'une, elle est choisie ;
- s'il y en a plusieurs, il faut en choisir une. Dans le cas, on choisit pour l'instant la première spécialisation trouvée. Il est clair que ce choix peut être amélioré si on définit des critères de choix.
  Il faut remarquer que cette situation ne peut se produire que si l'utilisateur est intervenu.

Remarque : Si $\mathcal{C}all(f_i)(c)$ est déjà attribué avant l'application de *ChooseCall*, on ignore son ancienne valeur.

Dans ce mode, l'application des actions *MissingOutputs*(c, f_i) et *MissingInputs*(c, f_i) est équivalente à l'application de *ChooseCall*(c, f_i).

Il faut noter que ce mode peut être également être choisi si l'on veut piloter tous les choix manuellement car les actions sont toutes générées, et l'outil ne fait un choix que lorsqu'elles sont appliquées. Elles peuvent donc être préalablement transformées en *ChangeCall* par l'utilisateur.

2. Les critères de choix restent à définir.

3.4.5 Autres actions

Duplication d'une spécialisation : *CopySlice*

L'action $f_j = \mathit{CopySlice}(f_i)$ permet simplement de dupliquer f_i et donc de créer une nouvelle fonction spécialisée f_j. Le marquage, ainsi que la fonction $\mathcal{C}all(f_i)$ sont simplement copiés.

Il faut noter que, initialement, f_j n'est pas appelée.

Combinaison de spécialisations : *Combine*

On peut combiner deux spécialisations (c'est-à-dire en faire l'union) à l'aide de $f_3 = \mathit{Combine}(f_1, f_2)$.

La table de marquage se calcule simplement en combinant les marques, sans qu'il y ait besoin de propager. Il faut ensuite résoudre les appels de fonction, c'est-à-dire calculer $\mathcal{C}all(f_3)$. Ce qui est fait pour un appel donné c dépend de sa signature dans le résultat f_3 par rapport à ce qu'on a dans f_1 et f_2. On sait que, par construction $sig_c(c, f_3) = sig_c(c, f_1) \cup sig_c(c, f_2)$ et qu'elle est donc supérieure ou égale.

- si $sig_c(c, f_1) = sig_c(c, f_2)$:
  - si $\mathcal{C}all(f_1)(c) = \mathcal{C}all(f_2)(c) : \mathcal{C}all(f_3)(c) = \mathcal{C}all(f_1)(c)$
  - si $\mathcal{C}all(f_1)(c) = \top_c : \mathcal{C}all(f_3)(c) = \mathcal{C}all(f_2)(c)$
  - si $\mathcal{C}all(f_2)(c) = \top_c : \mathcal{C}all(f_3)(c) = \mathcal{C}all(f_1)(c)$
  - sinon, cela veut dire que les deux appels sont attribués à des fonctions différentes : on choisit celle qui a la plus petite signature (ie. la plus précise).
- sinon, si $sig_c(c, f_3) = sig_c(c, f_1) : \mathcal{C}all(f_3)(c) = \mathcal{C}all(f_1)(c)$
- sinon, si $sig_c(c, f_3) = sig_c(c, f_2) : \mathcal{C}all(f_3)(c) = \mathcal{C}all(f_2)(c)$
- sinon, $\mathcal{C}all(f_3)(c) = g_i$, et on crée une action $g_i = \mathit{Combine}(\mathcal{C}all(f_1)(c), \mathcal{C}all(f_2)(c))$.

Il faut noter que la création de f_3 ne modifie ni f_1, ni f_2. Si l'objectif est de remplacer f_1 et f_2 par f_3, cela peut être fait par une combinaison d'actions élémentaires (cf. §3.6.4).

Suppression d'une spécialisation : *DeleteSlice*

Une fonction spécialisée peut être supprimée à l'aide de l'action $\mathit{DeleteSlice}(f_i)$, mais cette action ne peut être appliquée que que si f_i n'est pas appelée, sauf si elle n'est appelée que par elle même. Cela signifie que ses appels doivent être supprimés au préalable à l'aide d'actions *ChangeCall* (ou de *DeleteSlice* sur les appelant...).

3.5 Sélection persistante

Après une première utilisation des fonctions présentées ci-dessus, il est apparu qu'un besoin utilisateur pouvait également être de marquer une instruction (ou une donnée à un point de programme) pour toutes les spécialisations d'une fonction. Ce type de sélection a été nommée **persistante**. Cela signifie que chaque fonction a, en parallèle de ses différents marquages, un marquage minimum qui est utilisé à la création de toute nouvelle spécialisation. Lorsque l'utilisateur ajoute une sélection persistante, elle est ajoutée à la fois dans ce marquage minimum, et dans toutes les spécialisations existantes. L'ajout d'une sélection persistante dans une fonction peut également rendre tous ses appels visibles (si l'option est positionnée) car l'utilisateur souhaite que le programme réduit passe aussi souvent au point marqué que

le programme source : cela suppose donc que tous les chemins qui mènent à la fonction soit visibles.

3.6 Gérer des actions élémentaires

Les actions élémentaires présentées ci-dessus doivent être combinées pour répondre à des requêtes de plus haut niveau. L'utilisateur exprime ses requêtes en ajoutant des actions dans la liste du projet, mais celles-ci doivent ensuite être décomposées pour être appliquées, et l'application d'actions élémentaires en génère d'autres.

La granularité à laquelle l'utilisateur peut intervenir reste à définir en fonction de ce que l'on souhaite faire de l'outil.

Nous allons tout d'abord examiner comment peut être gérée la liste d'action (§3.6.1), puis nous verrons un certain nombre d'exemple de combinaison d'actions élémentaires pour répondre à des requêtes de plus haut niveau.

3.6.1 Modification ou suppression d'actions

Les actions de l'utilisateur peuvent être supprimées par ses soins avant d'être appliquée. En revanche, les actions générées telles que *ChooseCall*, *MissingInputs* ou *MissingOutputs* consistent à rendre un appel de fonction cohérent, elles ne peuvent donc pas être supprimées. Par contre, l'utilisateur peut les transformer en *ChangeCall* pour préciser la fonction qu'il souhaite voir appeler. Dans ce cas, cette nouvelle action ne pourra à son tour pas être supprimée.

3.6.2 Ordre des actions

A priori, le marquage étant parfaitement défini, il ne dépend pas de l'ordre des calculs. Néanmoins, le résultat obtenu peut dépendre de l'ordre d'application des actions, car si on applique une action qui construit une nouvelle fonction g_j, puis une autre pour construire f_i qui appelle une fonction g, g_j va pouvoir être choisie pour l'appel, alors que si la première action n'avait pas encore été appliquée, c'est peut-être une autre spécialisation qui aurait été appelée, ou une nouvelle action qui aurait été générée. De plus, certaines actions ont une précondition qui n'est éventuellement satisfaite que si les actions générées sont appliquées dans l'ordre. On ne prévoit donc pas dans un premier temps de pouvoir modifier l'ordre des actions.

3.6.3 Propagation aux appelants

Les actions élémentaires ne propagent le marquage des entrées d'une certaine fonction f_i qu'aux appels (c, h_k) tels que $\mathcal{C}all(h_k)(c) = f_i$. Si on souhaite construire de nouvelles fonctions spécialisées pour toutes les fonctions h qui appellent f afin de propager le marquage, il faut enchaîner différentes actions élémentaires.

Voyons par exemple comment piloter l'ajout d'une marque utilisateur m sur un élément e de f avec propagation dans les appelants :

— $f_i = NewSlice(f_0)$

— $AddUserMark(f_i, (e, m))$
— $\forall h_0.\exists c_f \in call(h_0) \Rightarrow h_k = NewSlice(h_0) \wedge \forall c_f.ChangeCall(c_f, h_k, f_i)$

3.6.4 Fusion de deux spécialisation

Si deux spécialisations f_1 et f_2 ont été calculées, l'utilisateur peut vouloir les fusionner. Pour cela, il faut appliquer les actions suivantes :
— $f_3 = Combine(f_1, f_2)$
— $\forall c, h_k.\mathcal{C}all(h_k)(c) = f_1 \Rightarrow ChangeCall(c, h_k, f_3)$
— $\forall c, h_k.\mathcal{C}all(h_k)(c) = f_2 \Rightarrow ChangeCall(c, h_k, f_3)$
— $DeleteSlice(f_1), DeleteSlice(f_2)$

3.7 Production du résultat

On rappelle que l'objectif est de générer des fichiers source compilables. Or, après avoir effectué tout le travail de filtrage présenté précédemment, on obtient un projet contenant, pour chaque fonction, zéro, une, ou plusieurs fonctions filtrées, avec pour chaque fonction filtrée, une table de marquage permettant de déterminer les instruction visibles, et les appels de fonction. Voyons ce qu'il manque pour obtenir le résultat final.

3.7.1 Déclarations globales

Pour produire des fichiers de résultats, il faut avant tout être capable d'y mettre ce qui est en dehors des fonctions, à savoir les déclarations globales.

Il aurait été possible de calculer des liens de dépendance vers les déclarations globales, mais on a considéré que la suppression de déclarations inutiles était une fonctionnalité intéressante en soi, et cela est donc fait sous forme d'une passe supplémentaire sur le résultat.

3.7.2 Fonctions

Pour une fonction qui a été analysée, on a calculé la représentation de la fonction source, ainsi qu'une ou plusieurs spécialisations. Les fonctions non analysées ne figureront pas dans le résultat car elles ne participent pas au calcul demandé.

Fonction spécialisée

Pour générer le code correspondant à une fonction spécialisée, il faut lui donner un nom qui n'entre pas en conflit avec celui de des fonctions source ou des autres spécialisation. Il faut également avoir sa signature pour savoir si elle possède une valeur de retour, et connaître la liste de ses arguments visibles.

Garder ou non la fonction source

Une fonction source n'a besoin d'être présente dans l'application générée que si elle est appelée. Le traitement d'un appel de fonction qui ne conduit pas à appeler une spécialisation doit donc mémoriser un lien avec la fonction source.

Variables locales statiques

Lorsqu'une fonction est présente en plusieurs exemplaires dans le résultat, il faut penser à sortir les éventuelles variables statiques communes à plusieurs spécialisations afin qu'elles soient partagées par les différentes fonctions.

Modification des appels

Pour générer un appel à une fonction spécialisée, il faut avoir les mêmes informations que celles qui sont utilisées pour la définition de la fonction, à savoir :

— son nom,
— la liste des paramètres formels nécessaires au calcul des sorties sélectionnées,
— le type de retour pour savoir si la fonction produit un résultat.

Génération des nouveaux prototypes

Lorsque l'on produit des fonctions spécialisées, il faut aussi générer les prototypes associés. Comme elles ont forcément la même portée que la fonction source, il suffit de générer ces nouveaux prototypes à l'endroit où l'on rencontre le prototype initial.

Gestion des blocs

Même si le graphe de dépendance contient des éléments représentant les blocs, et donc que le *slicing* associe des marques à ces éléments, il est délicat de supprimer un bloc, même si est sensé être invisible. En effet, cela peut parfois changer la sémantique, et dans une construction `if (c) { S1; } else { S2; }`, on peut même obtenir un résultat syntaxiquement incorrect si on n'y prend pas garde.

Comme il peut être utile par ailleurs d'avoir un outil de nettoyage de code qui supprime proprement les blocs vide, on décide de ne pas les traiter au niveau du *slicing*. Tous les blocs sont donc considérés comme visibles indépendamment de leur marque, et le nettoyage est effectué lors d'une passe supplémentaire sur le code généré.

3.7.3 Annotations

Les annotations peuvent être utilisées pour construire des requêtes de *slicing*, mais au delà de cette possibilité, il faut déterminer si les annotations présentes dans le code source doivent être mises dans le résultat. On regarde pour cela si les données qui permettent d'évaluer la propriété sont préservées dans le programme réduit : si c'est le cas, l'annotation va être gardée, dans le cas contraire, elle va être slicée.

Chapitre 4

Utilisation

Dans ce chapitre essaye de présenter quelques bases qui permettent de tester la boîte à outil de *slicing* à partir de l'interpréteur ou d'un fichier de commande. Pour l'utilisation de l'interface graphique, se reporter à la documentation correspondante. Il est également possible d'exécuter certaines actions élémentaires en ligne de commande. Voir les options proposées à l'aide de :

```
bin/toplevel.top --help
```

4.1 Utilisation interactive

Lorsqu'on lance l'outil pour faire du slicing, il faut lui fournir le ou les fichiers à analyser, et au minimum l'option `-deps` qui permet d'effectuer l'analyse de dépendances (utiliser `-help` pour voir les autres options) :

```
bin/toplevel.top -deps fichier.c
```

On se retrouve alors avec un prompt `#` qui signale que l'on est sous un interpréteur OCaml.

Pour quitter l'interpréteur, il faut utiliser la commande :

```
#quit;;
```

(le `#` n'est pas le prompt, il fait parti de la commande). On peut aussi utiliser `Ctrl-D`.

Astuce

Comme tout interpréteur OCaml, il n'a pas de capacité d'éditer la ligne de commande (pour corriger une faute de frappe par exemple), ni d'historique (pour rappeler une commande précédente). Il est donc fortement conseillé, même si ce n'est pas indispensable, d'installer `ledit`, que l'on peut obtenir à l'adresse : `ftp://ftp.inria.fr/INRIA/Projects/cristal/Daniel.de_Rauglaudre/Tools/`. Il suffit ensuite de faire précéder le nom de la commande par `ledit`, tout simplement. Faire `man ledit` pour avoir plus d'information sur les possibilités de cet outil.

Sous l'interpréteur, on peut utiliser n'importe quelle instruction OCAML, et toutes les commandes spécifiques à l'outil.

Pour ceux qui ne sont pas familier avec OCAML, pour lancer une commande, il faut taper son nom suivie des arguments, ou de parenthèses vide s'il n'y a pas d'arguments, et terminer par deux points-virgules.

L'interpréteur affiche le type de retour de la commande (`unit` correspond au type vide), suivi éventuellement de la valeur retournée. Dans la plupart des cas, la première information n'intéresse pas l'utilisateur, mais cet affichage ne peut pas être désactivé. Il peut parfois être utile pour retrouver la signature des commandes< par exemple :

```
module S = Db.Slicing;;
```

affiche le type du nouveau module S, et permet donc de voir la liste des commandes du module `Db.Slicing`.

L'organisation de FRAMA-C regroupe toutes les commandes dans le module `Db`. Le sous-module `Db.Slicing` contient les commandes de *slicing*. Le nom d'une commande doit normalement être préfixé par le nom du module a qui elle appartient. Pour pouvoir utiliser les noms sans préfixe, il faut ouvrir le module à l'aide de :

```
open Db;;
```

On peut aussi choisir de renommer le module pour avoir un préfixe plus court :

```
module S = Db.Slicing;;
```

Des commandes provenant d'autres modules sont également utiles dans le cadre de l'utilisation du *slicing*. `Kui`, en particulier, fournit des fonctions qui permettent de naviguer dans les informations générées par les autres analyseurs.

4.2 Fichier de commandes

Les commandes peuvent être tapées directement, mais on peut également charger des fichiers contenant les commandes grâce à la commande `#use "cmds.ml";;`. Le contenu du fichier `cmds.ml` est alors interprété comme si on l'avait tapé, et on reprend la main sous l'interpréteur à l'issue de son exécution.

Si l'on veut simplement lancer un fichier de commandes, on peut utiliser la redirection Unix. L'interpréteur se terminera alors à la rencontre du EOF (fin de fichier).

Astuce

En utilisant `ledit` (voir l'astuce page précédente) on obtient également la possibilité de mémoriser l'historique, et d'avoir donc une journalisation des commandes lancées (très pratique comme base pour écrire un script, ou pour rejouer une séquence). Pour cela, il suffit de taper :

`ledit -h cmds.ml bin/toplevel.top` ... arguments...

4.3 Exemples

Quelques exemples d'utilisation peuvent être trouvées dans le répertoire de test. Les deux fichiers suivant fournissent quelques fonctions qui facilitent l'utilisation :

- `libSelect.ml` propose une interface simplifiée pour des sélections simples,
- `libAnim.ml` permet de générer des représentations graphiques du projet à différentes étapes de sa construction afin de visualiser les relations entre les spécialisations.

Chapitre 5

Conclusion

En conclusion, on peut dire que les fonctionnalités de base de l'outil n'ont pas beaucoup évoluées en 2008, mais il a gagné en robustesse, et en précision : ce qui était le principal objectif de l'année.

La principale évolution concerne la gestion des annotations que ce soit lors de la production du résultat (que garde-t-on ?) que comme critère de *slicing*. Ce point est encore en développement car il nécessite l'utilisation de fonctions externes au module qui n'existent pas encore.

La boîte à outils de *slicing* peut être considéré comme stable, même si elle peut encore évoluer pour répondre à de nouveaux besoins,

L'annexe D présente en particulier certains projets qui ont été évoqués, et dont certain pourrait éventuellement venir compléter l'outil.

Annexe A

Algorithmes

A.1 Hypothèses

On se donne quelques types et fonctions de base :

```
(* Point de programme *)
type t_program_point ;;

(* Instruction *)
type t_stmt ;;

(* l'instruction située àun point de contrôle (après) *)
val get_pp_stmt : t_program_point -> t_stmt ;;

(* CFG : control flow graph *)
type t_cfg ;;
(* successeurs dans le cfg. *)
val get_cfg_succ : t_cfg -> t_stmt -> t_stmt list ;;
(* predecesseurs dans le cfg. *)
val get_cfg_prev : t_cfg -> t_stmt -> t_stmt list ;;

(* PDG : program dependences graph *)
type t_pdg ;;
(* element composant le PDG *)
type t_elem ;;
(* donne la liste des dépendances directes de l'élément dans le PDG *)
val get_dpds : t_elem -> t_pdg -> t_elem list ;;
val get_all_dpds : t_pdg -> t_elem -> t_elem list ;;
val get_list_all_dpds : t_pdg -> t_elem list -> t_elem list ;;
val get_list_control_dpds : t_pdg -> t_elem list -> t_elem list ;;
val get_list_all_control_dpds : t_pdg -> t_elem list -> t_elem list ;;
val merge : t_elem list -> t_elem list -> t_elem list ;;

val get_pp_elems : t_pdg -> t_program_point -> t_elem list ;;

(* correspondance entre les instructions et les éléments du PDG *)
type t_stmt_elems ;;

(* retrouver l'instruction correspondant àun élément *)
val get_stmt : t_elem -> t_stmt_elems -> t_stmt ;;
```

```
(* retrouver les instructions correspondant aux éléments *)
val get_stmts : t_elem list -> t_stmt_elems -> t_stmt list ;;
(* retrouver les éléments correspondant àune instruction *)
val get_elems : t_stmt -> t_stmt_elems -> t_elem list ;;

type t_state

type t_data

val get_state : t_program_point -> t_state ;;
val get_defs_data : t_state -> t_data -> t_elem list ;;
(* type des marques *)
type t_mark ;;
(* la marque correspondant àmS : superflu. *)
val spare_mark : t_mark ;;
(* combinaison de deux marques *)
val combine_mark : t_mark -> t_mark -> t_mark ;;

(* type correspondant au marquage des instructions d'une fonction. *)
type t_ff ;;
(* lire la marque associée àune instruction dans le marquage *)
val get_stmt_mark : t_stmt -> t_ff -> t_mark ;;
(* remplacer la marque associée àune instruction dans le marquage *)
val replace_stmt_mark : t_ff -> t_stmt -> t_mark -> t_ff ;;
```

A.2 Marquage d'une fonction

Il s'agit de voir comment sont marquées les instructions d'une fonction à partir d'une requête donnant un élément du PDG et une marque.

L'algorithme présenté ici est une version simplifiée de celui qui existe dans l'outil afin d'être plus facilement compréhensible. Il est très inefficace et risque de boucler car on ne teste pas si l'instruction à marquer contient déjà la nouvelle marque (test d'arrêt).

Il s'agit de plus d'une version sans spécialisation des appels de fonction.

```
(* On nomme H le module des hypothèses. *)
module H = AlgoH ;;

(* produit une nouvelle fonction spécialisée en partant de [ff]
   en marquant l'élément [e] et toutes ses dépendances avec la marque [m]. *)
let rec mark_rec_pdg_elem pdg stmt_elems m e ff =
  let new_ff = add_elem_mark pdg stmt_elems m e ff in
  let dpds = H.get_dpds e pdg in
   List.fold_right (mark_rec_pdg_elem pdg stmt_elems m) dpds new_ff
  (* ;; *)
and
(* [add_elem_mark] ajoute la marque [m] àl'instruction correspondant à
   l'élément [e] et marque les autres éléments éventuels comme superflus. *)
      add_elem_mark pdg stmt_elems m e ff =
  let stmt = H.get_stmt e stmt_elems in
  let old_m = H.get_stmt_mark stmt ff in
  let new_m = H.combine_mark old_m m in
```

```
    let new_ff = H.replace_stmt_mark ff stmt new_m in
    let elems = H.get_elems stmt stmt_elems in
    let (_, other_elems) = List.partition (fun elem -> elem = e) elems in
    let mark_spare_elem e ff = mark_rec_pdg_elem pdg stmt_elems H.spare_mark e ff in
      List.fold_right mark_spare_elem other_elems new_ff
```

Annexe B

Exemple de marquage interprocédural

Cette partie présente comment les actions élémentaires précédemment présentées peuvent être utilisées pour répondre à des requêtes utilisateur de plus haut niveau.

B.1 Présentation de l'exemple

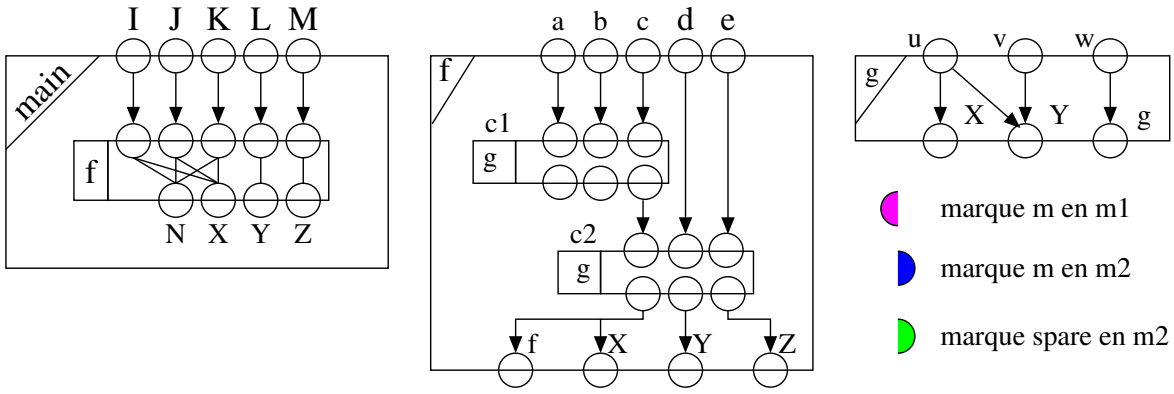
Nous allons voir différents exemple de marquage de l'exemple ci-dessous. Dans tous les cas, on considère que l'utilisateur souhaite n'avoir qu'une spécialisation par fonction source dans le résultat, et qu'il demande systématiquement la propagation de son marquage aux appelants.

Pour simplifier la présentation, comme on ne s'intéresse ici qu'à la propagation interprocédurale, on n'utilise qu'une marque élémentaire m quelconque qui rend un élément visible et la marque *Spare* déjà mentionnée.

```
int X, Y;
int g (int u, int v,
       int w) {
  lg1: X = u;
  lg2: Y = u + v;
  return w;
  }
```

```
int Z;
int f (int a, int b,
       int c, int d,
       int e) {
  int r;
  lf1: r = g (a, b, c);
  lf2: Z = g (r, d, e);
  return X;
  }
```

```
int I, J, K, L, M, N;
int main () {
  lm1: /* ... */
  lm2: N = f (I, J, K, L, M);
  lm3: /* ... */
  }
```



La légende indique comment sont représentées les marques sur les figures suivantes.

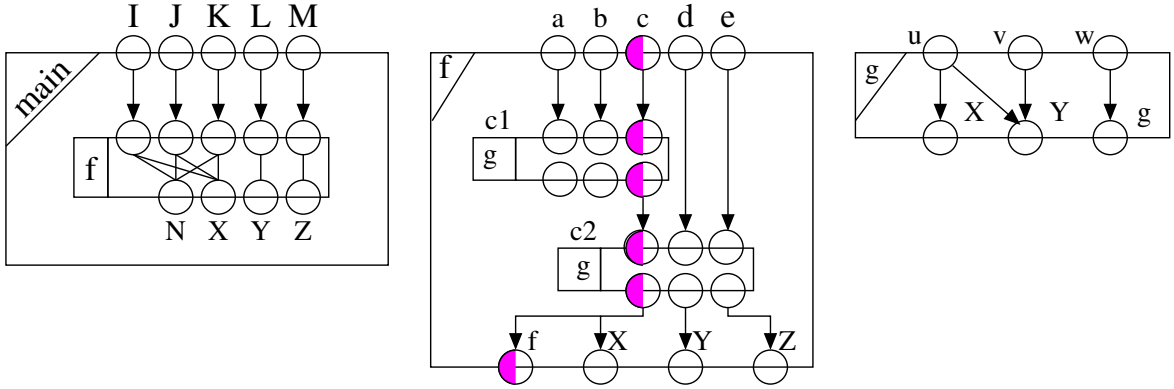
B.2 Cas 1

Supposons tout d'abord que l'utilisateur veuille sélectionner la sortie 0 de f, et voyons comment se déroule le calcul.

On a vu en §3.6.3 que cette requête de l'utilisateur se traduit par la séquence d'actions élémentaires suivantes :

- $f_1 = NewSlice(f_0)$,
- $AddOutputMarks(f_1, (out_0, m))$
- $main_1 = NewSlice(main_0)$,
- $ChangeCall(c, main_1, f_1)$.

On calcule tout d'abord le marquage de f_1 par simple propagation :

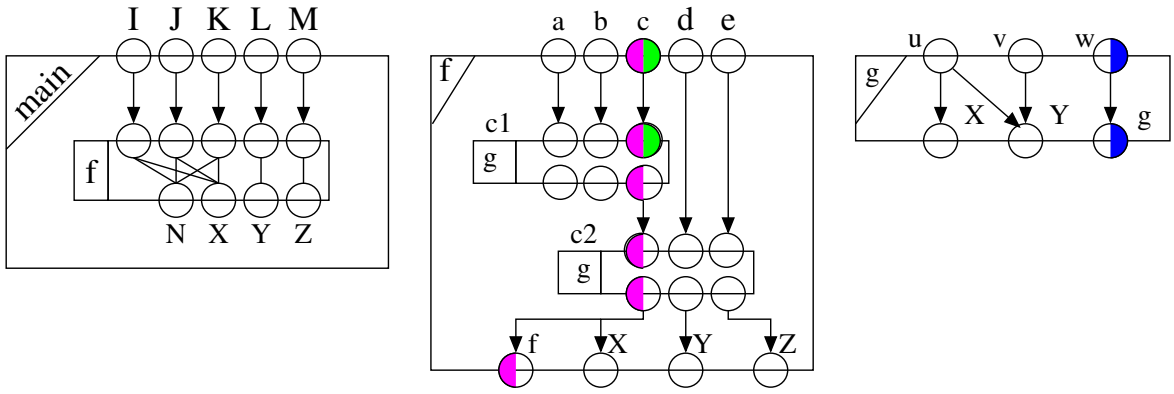


Puis, la seconde action génère $ChooseCall(c_1, f_1)$ et $ChooseCall(c_2, f_1)$.

A l'application de la première de ces nouvelles actions, comme il n'y a pas encore de spécialisation pour g, on génère :

- $g_1 = NewSlice(g_0)$,
- $AddOutputMarks(g_1, (out_0, m))$
- $ChangeCall(c_1, f_1, g_1)$

A l'issue de la construction de g_1, l'entrée w a une marque m_2. L'application du *ChangeCall* va donc déclencher un $ModifCallInputs(c_1, f_1)$ qui va conduire à marquer l'entrée c de f_1 comme *Spare* (en plus de m en m_1).

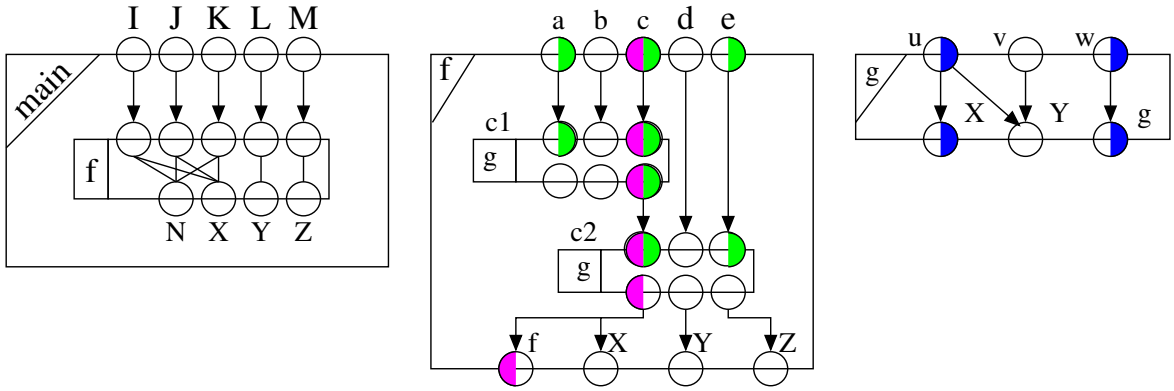


L'application de $ChooseCall(c_2, f_1)$ produit :

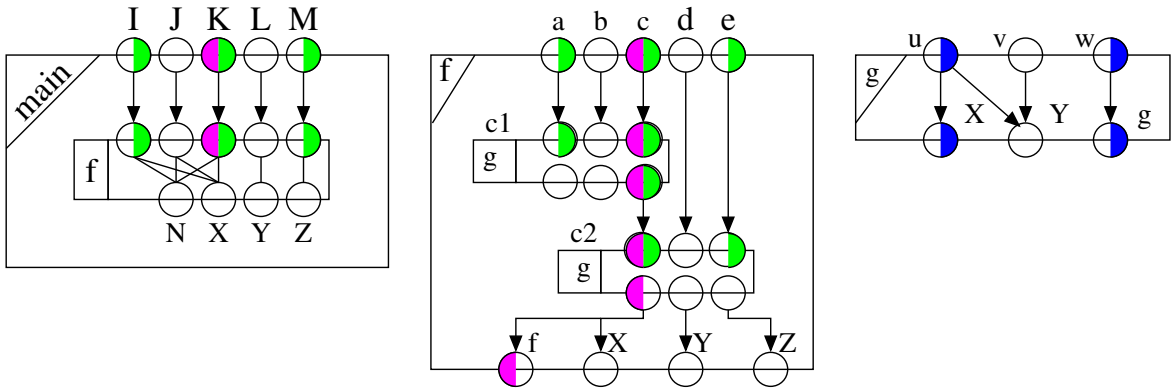
- $AddOutputMarks(g_1, (out_X, m))$ puisqu'on a choisi de n'avoir qu'une spécialisation par fonction source,
- et $ChangeCall(c_2, f_1, g_1)$.

Comme g_1 est appelée en c_1, la modification de son marquage conduit à générer $MissingInputs(c_1, f_1)$, qui, vues les options, est directement traduit par $ModifCallInputs(c_1, f_1)$.

Puis, le *ChangeCall* va déclencher $ModifCallInputs(c_2, f_1)$ qui va marquer e comme *Spare*.



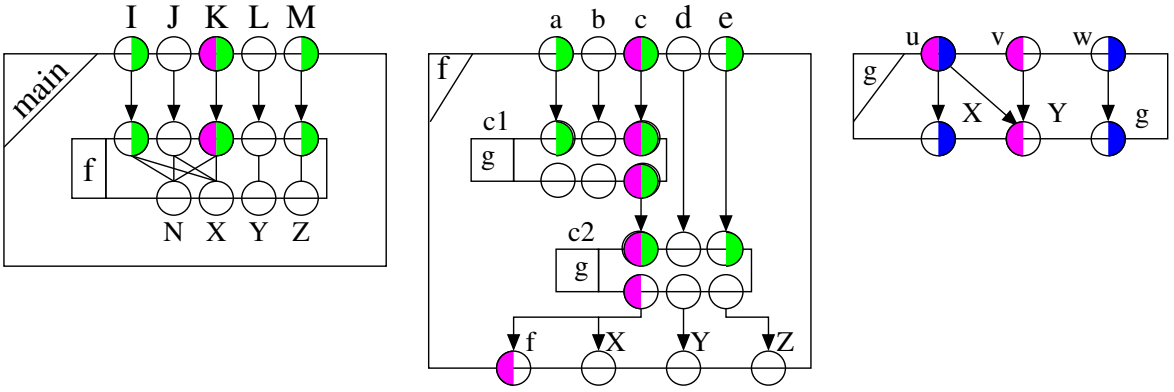
Finalement, il ne reste plus qu'à appliquer $ChangeCall(c, main_1, f_1)$ qui conduit à appliquer $ModifCallInputs(c, main_1)$ et donc à propager le marquage de f_1 dans $main_1$.



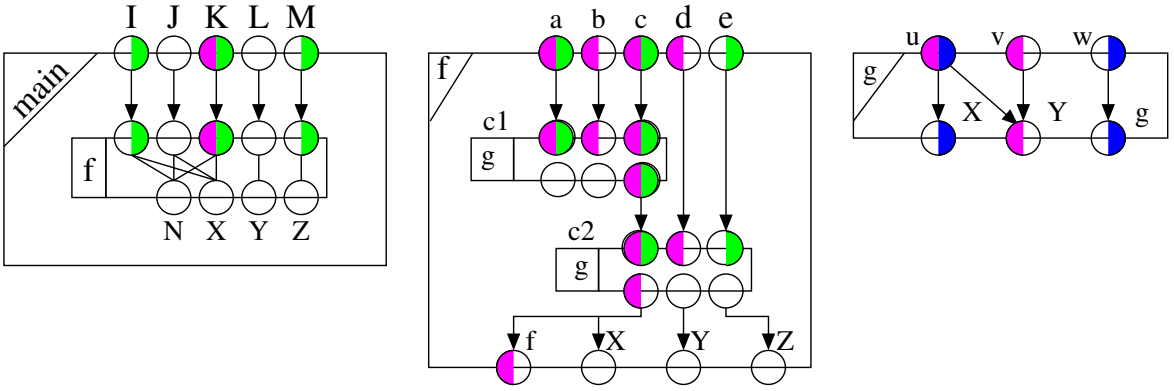
B.3 Cas 2

A partir du résultat du cas 1, l'utilisateur souhaite sélectionner le calcul de Y dans g_1.

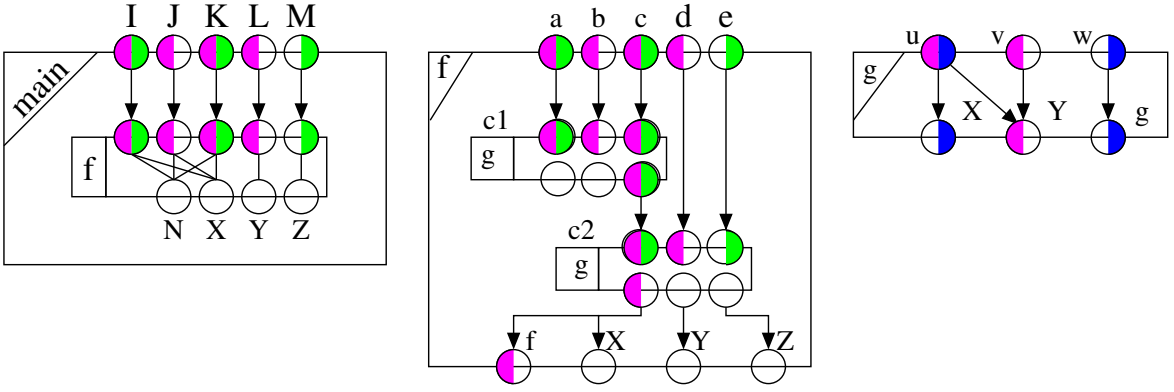
AddUserMark$(g_1, (out_Y, m))$ propage le marquage en m_1 à l'entrée v de g_1 (u est déjà marquée).



Puis comme g_1 est appelée, et qu'il manque des marques, deux actions *MissingInputs* sont générées. Les options indiquent qu'elles doivent être traduites en *ModifCallInputs*(c_1, f_1) et *ModifCallInputs*(c_2, f_1). Ce qui conduit à marquer en m_1 les entrées a, b, d de f_1.



De même, la propagation va être effectuée dans $main_1$ par l'application de *ModifCallInputs*$(c, main_1)$.



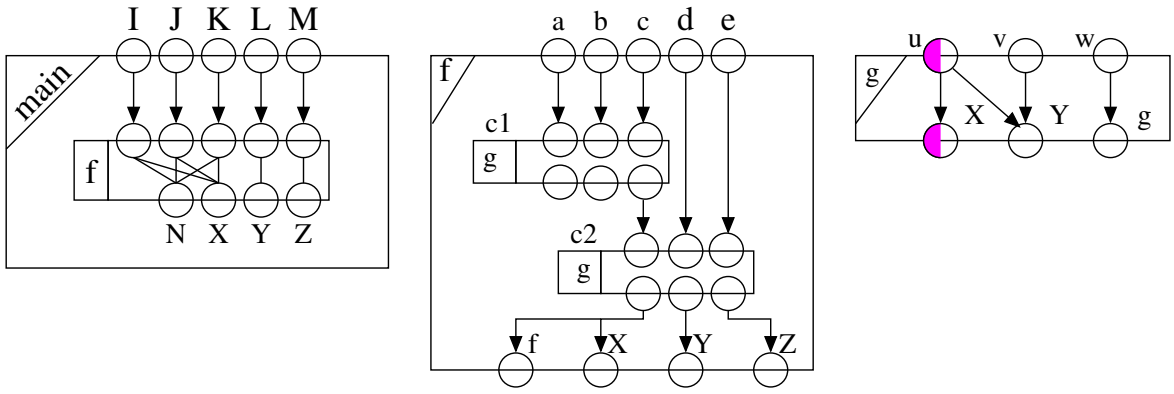
B.4 Cas 3

Dans une nouvelle étude, l'utilisateur souhaite sélectionner le calcul de X dans g.

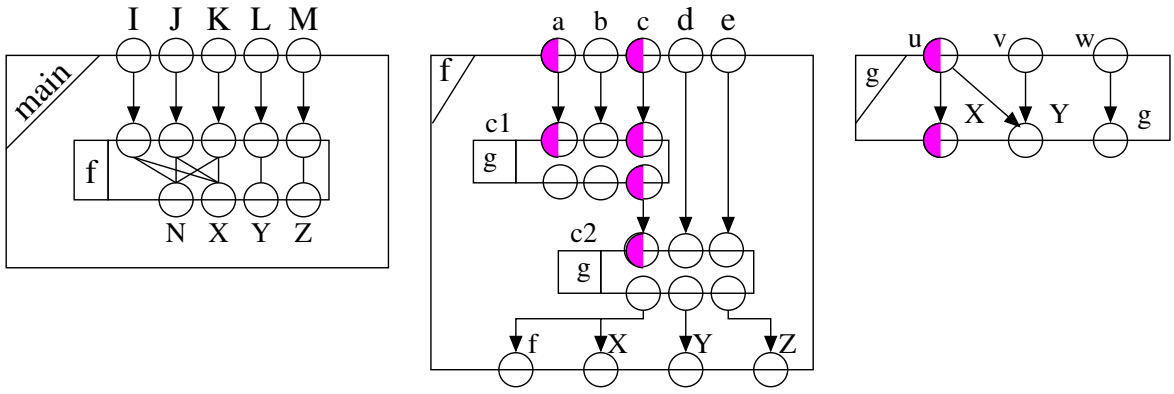
Comme dans le cas 1, cette requête se traduit par la création d'une fonction spécialisée g_1 et la propagation de son marquage à tous ses appels :

- $g_1 = NewSlice(g_0)$,
- $AddOutputMarks(g_1, (out_X, m))$
- $f_1 = NewSlice(f_0)$,
- $ChangeCall(c_1, f_1, g_1)$.
- $ChangeCall(c_2, f_1, g_1)$.
- $main_1 = NewSlice(main_0)$,
- $ChangeCall(c, main_1, f_1)$.

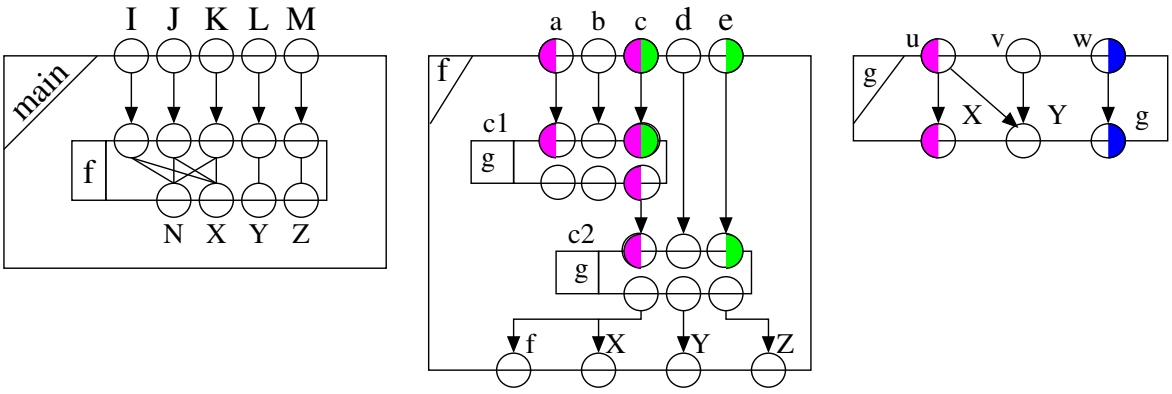
On calcule tout d'abord le marquage de g_1 :



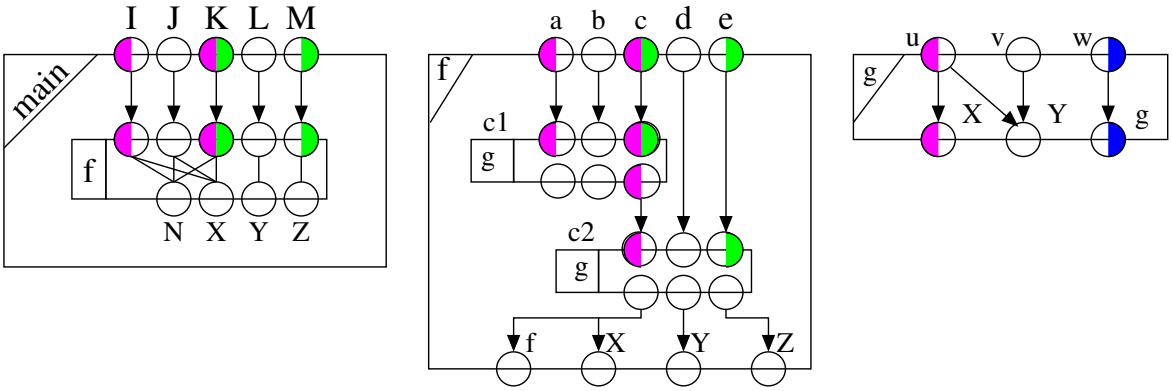
Puis, les deux *ChangeCall* dans f_1 conduisent à propager m_1 :



La seconde propagation (lors de la modification de c_2) déclenche $MissingOutputs(c_1, f_1)$ qui, au vue des options, se transforme en $AddOutputMarks(g_1, outSig_c(c_1))$. Ceci conduit à marquer m_2 la sortie 0 de g_1. Les *MissingInputs* générés se transforme en $ModifCallInputs(c_1, f_1)$ et $ModifCallInputs(c_2, f_1)$, qui propage *Spare* aux entrées c et e de f_1.



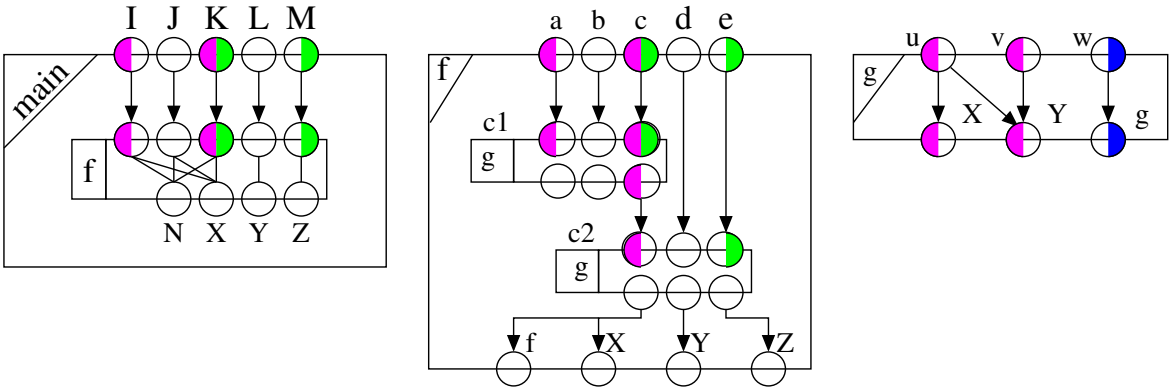
Enfin, le *ChangeCall*$(c, main_1, f_1)$ propage le marquage de f_1 dans $main_1$. On remarque que même si ce changement avait été effectué plus tôt, la propagation du marquage aurait été effectué grâce à des *MissingInputs*.



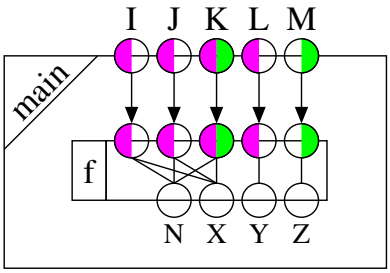
B.5 Cas 4

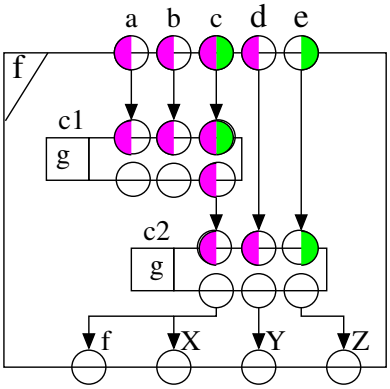
A partir du cas 3, l'utilisateur souhaite ajouter la sélection du calcul de Y dans g_1.

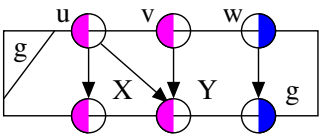
AddUserMark$(g_1, (out_Y, m))$ conduit à marquer Y en m_1, puis, par propagation, v en m_1 également :



Des actions *MissingInputs* transformée en *ModifCallInputs* propage m_1 aux entrées b et d de f_1, puis aux entrées J et L de $main_1$.


I J K L M
main
f
N X Y Z


a b c d e
f
c1
g
c2
g
f X Y Z


u v w
g
X Y g

Annexe C

Détail des actions

Ce chapitre présente une fiche signalétique pour chaque action du calcul interprocédurale présentées au chapitre 3. Elle correspondent aux commandes telles qu'elles ont été spécifiées, et il peut y avoir quelques différences par rapport à ce qui a été implémenté, mais il s'agit de différences mineures (principalement les noms). Pour plus de détails, le lecteur est invité à consulter la document du code qui sera toujours la plus à jour.

Les indications suivantes sont données pour chaque action :

- **Paramètres** : indique le sens des paramètres de l'action,
- **Précondition** : indique les conditions éventuelles de création et/ou d'application,
- **Création par l'utilisateur** : indique si l'utilisateur peut la créer,
- **Modifiable par l'utilisateur** : indique si l'utilisateur peut la modifier lorsqu'elle est dans la liste d'attente,
- **Génération automatique** : indique si cette action peut être générée par l'outil, et précise dans quels cas,
- **Application** : détaille ce qu'il se passe quand cette action est appliquée,
- **Génération** : donne la liste des actions pouvant être générées,
- **Modifications** : indique si une spécialisation peut être crée ou modifiée.

C.1 NewSlice

$$\mathit{NewSlice}(f_0)$$

- **Paramètres** : f_0 est la fonction source pour laquelle on souhaite créer une nouvelle spécialisation.
- **Précondition** : néant.
- **Création par l'utilisateur** : oui.
- **Modifiable par l'utilisateur** : oui (suppression).
- **Génération automatique** : oui, par *ChooseCall*.
- **Application** : crée une nouvelle spécialisation f_i, initialement sans aucun marquage (tout invisible).
- **Génération** : néant.
- **Modifications** : création d'une nouvelle f_i.

C.2 AddUserMark

$$AddUserMark(f_i, (e, m))$$

- **Paramètres** : f_i est la fonction dont on veut modifier le marquage, e un élément à marquer, m la marque à lui ajouter. Différentes version de cette action peut être proposée pour faciliter la désignation de e et le choix de la marque m.
- **Précondition** : f_i doit exister et e désigner un élément valide de f.
- **Création par l'utilisateur** : oui.
- **Modifiable par l'utilisateur** : oui (suppression).
- **Génération automatique** : non.
- **Application** : ajoute m à la marque $m1$ de e dans f_i et propage dans les dépendances.
- **Génération** : *ExamineCalls*.
- **Modifications** : modification de f_i.

C.3 ExamineCalls

$$ExamineCalls(f_i)$$

- **Paramètres** : f_i est la fonction dont le marquage a été modifié et pour laquelle il faut vérifier les appels.
- **Précondition** : f_i doit exister.
- **Création par l'utilisateur** : non.
- **Modifiable par l'utilisateur** : non.
- **Génération automatique** : oui.
- **Application** : vérifie la cohérence du marquage de chaque appel de fonction c de *call*(f) et également des éventuelles appels à f_i.
- **Génération** : *ChooseCall* et/ou *MissingOutputs* si le marquage de certains appels de fonction a été modifié, et *MissingInputs* si f_i est appelée et que les entrées de certains appels sont insuffisamment marquées.
- **Modifications** : rien.

C.4 ChooseCall

$$ChooseCall(c, f_i)$$

- **Paramètres** : appel c dans la fonction spécialisée f_i.
- **Précondition** : néant.
- **Création par l'utilisateur** : non.
- **Modifiable par l'utilisateur** : oui, elle peut être remplacée par un $ChangeCall(c, f_i, g_j)$ (voir les conditions de création de cette action).
- **Génération automatique** : oui, quand le marquage de f_i est modifié et que l'appel c devient visible, cette action est générée pour lui attribuer une fonction à appeler.
- **Application** : détermine la fonction à appeler en tenant compte du mode de fonctionnement.
- **Génération** : *ChangeCall* et éventuellement *NewSlice* et *AddOutputMarks*.
- **Modifications** : rien.

C.5 ChangeCall

$$ChangeCall(c, f_i, g_j)$$

- **Paramètres** : on considère l'appel c de f_i, g_j est la fonction à appeler.
- **Précondition** : $\mathcal{C}all(f_0)(c) = g_0$ et la signature de sortie de g_j doit être compatible avec $sig_c(c, f_i)$.
- **Création par l'utilisateur** : oui.
- **Modifiable par l'utilisateur** : oui, mais seulement s'il l'a initialement créée.
- **Génération automatique** : oui, par un *ChooseCall*.
- **Application** : $\mathcal{C}all(f_i)(c) = g_j$ et les marques des entrées de g_j sont propagées dans f_i ($ModifCallInputs(c, f_i)$).
- **Génération** : l'action *ModifCallInputs* est directement appliquée, mais elle peut générer d'autres actions (voir §C.7).
- **Modifications** : f_i.

C.6 MissingInputs

$$MissingInputs(c, f_i)$$

- **Paramètres** : appel c dans la fonction spécialisée f_i,
- **Précondition** : la fonction appelée nécessite plus d'entrées que n'en calcule f_i.
- **Création par l'utilisateur** : non,
- **Modifiable par l'utilisateur** : oui, elle peut être remplacée par un $ChangeCall(c, f_i, g_j)$ (voir les conditions de création de cette action).
- **Génération automatique** : oui, lorsque la fonction $\mathcal{C}all(f_i)(c) = g_j$ a été modifiée, et qu'elle nécessite le marquage d'entrées qui ne le sont pas $sig_c(c, f_i)$.
- **Application** : équivalente à *ModifCallInputs* ou *ChooseCall* en fonction du mode de fonctionnement.
- **Génération** : dépend de l'application (voir ci-dessus).
- **Modifications** : dépend de l'application (voir ci-dessus).

C.7 ModifCallInputs

$$ModifCallInputs(c, f_i)$$

- **Paramètres** : appel c dans la fonction spécialisée f_i.
- **Précondition** : $\mathcal{C}all(f_i)(c) = g_j$
- **Création par l'utilisateur** : non.
- **Modifiable par l'utilisateur** : non.
- **Génération automatique** : oui, par l'application de *MissingInputs* dans certains modes.
- **Application** : propage les marques des entrées de g_j au niveau de l'appel c de f_i et et dans le dépendances.
- **Génération** : *ChooseCall* et/ou *MissingOutputs* si le marquage de certains appels de fonction a été modifié, et *MissingInputs* si f_i est appelée et que les entrées de certains appels sont insuffisamment marquées.
- **Modifications** : f_i.

C.8 MissingOutputs

$$MissingOutputs(c, f_i)$$

- **Paramètres** : appel c dans la fonction spécialisée f_i
- **Précondition** : $\mathcal{C}all(f_i)(c) = g_j$
- **Création par l'utilisateur** : non.
- **Modifiable par l'utilisateur** : oui, elle peut être remplacée par un $ChangeCall(c, f_i, g_k)$ (voir les conditions de création de cette action).
- **Génération automatique** : oui, lorsque le marquage de c dans f_i est modifié, que l'appel est attribué à g_j, et que les sorties de g_j sont insuffisantes.
- **Application** : dépend du mode :
  - *AddOutputMarks*
  - ou *ChooseCall*
- **Génération** : dépend de l'application (voir ci-dessus).
- **Modifications** : dépend de l'application (voir ci-dessus).

C.9 AddOutputMarks

$$AddOutputMarks(f_i, outSig_f)$$

- **Paramètres** : f_i est la fonction dont on veut modifier le marquage, $outSig_f$ indique les marques qui doivent être ajouté aux sorties.
- **Précondition** : f_i doit exister et $outSig_f$ correspondre à une signature des sorties de f.
- **Création par l'utilisateur** : non.
- **Modifiable par l'utilisateur** : non.
- **Génération automatique** : oui, par l'application de *MissingOutputs*.
- **Application** : les marques de $outSig_f$ sont ajoutées aux marques $m2$ des sorties de f_i et propagées.
- **Génération** : *ChooseCall* et/ou *MissingOutputs* si le marquage de certains appels de fonction a été modifié, et *MissingInputs* si f_i est appelée et que les entrées de certains appels sont insuffisamment marquées.
- **Modifications** : modification de f_i.

C.10 CopySlice

$$CopySlice(f_i)$$

- **Paramètres** : f_i est la fonction spécialisée à copier.
- **Précondition** : f_i doit exister.
- **Création par l'utilisateur** : oui.
- **Modifiable par l'utilisateur** : oui (suppression),
- **Génération automatique** : non.
- **Application** : crée une nouvelle spécialisation f_j de f dont le marquage et les appels sont identiques à ceux de f_i. Attention, à l'issue de cette action, f_j n'est pas appelée.
- **Génération** : non.
- **Modifications** : création d'une nouvelle f_j.

C.11 Combine

$$Combine(f_i, f_j)$$

— **Paramètres** : f_i et f_j sont les fonctions spécialisées à combiner.
— **Précondition** : f_i et f_j doivent exister.
— **Création par l'utilisateur** : oui.
— **Modifiable par l'utilisateur** : oui (suppression),
— **Génération automatique** : non.
— **Application** : calcule une nouvelle spécialisation f_k
— **Génération** : *ChooseCall*
— **Modifications** : création et calcul de f_k.

C.12 DeleteSlice

$$DeleteSlice(f_i)$$

— **Paramètres** : f_i est la fonction spécialisée à supprimer.
— **Précondition** : f_i ne doit pas être appelée.
— **Création par l'utilisateur** : oui.
— **Modifiable par l'utilisateur** : oui (suppression).
— **Génération automatique** : non.
— **Application** : f_i est supprimée.
— **Génération** : néant.
— **Modifications** : f_i est supprimée.

Annexe D

Projets

Ce chapitre regroupe diverses idées qui ont été évoquées à un moment ou un autre, et qui ont soit été abandonnées, soit remises à plus tard.

D.1 Autres critères

D.1.1 Spécifier ce qu'on ne veut plus

Un autre critère intéressant de *slicing*, qui n'apparaît dans aucun papier, serait, je pense, de pouvoir spécifier ce que l'on veut enlever. Par exemple, *"que devient le programme si on ne s'intéresse pas à telle sortie ou à tel traitement"*.

Par ailleurs, l'analyse de dépendance utilisée pour le *slicing* peut également être utilisée pour calculer des informations utiles à d'autres analyses. On peut par exemple s'intéresser aux variables dont la valeur détermine la sortie d'une boucle ; ce qui peut être utiliser comme un indice sur les élargissements à effectuer dans une interprétation abstraite.

D.1.2 Calcul d'une variable globale

Le premier filtrage global vise à sélectionner les instructions qui participent au calcul d'une variable globale donnée. Comme on a accès par ailleurs à la liste des sorties de chaque fonction, il suffit de générer les filtres permettant de calculer cette variable sur chaque fonction l'ayant comme sortie.

D.2 Utilisation de la sémantique

Cette partie regroupe quelques idées d'évolutions pour améliorer le *slicing* d'une fonction en utilisant la sémantique des instructions. Certaines d'entre elles ont été implémentées dans l'outil précédent ; d'autres ne sont que des pistes qui restent à explorer.

D.2.1 Réduction relative à une contrainte

Dans l'outil précédent, une commande permettait de couper une branche en remplaçant le test de la condition de chemin par un `assert`.

Plus généralement, on s'intéresse ici à une contrainte représentant une assertion dans le code. On aimerait déplacer cette contrainte afin de déterminer un point où elle est impossible (réduite à *false*). On pourrait alors *couper* alors la branche correspondante.

Le test conditionnel permettant d'accéder à cette branche peut être transformé en assertion, ou être supprimé (option ?). Dans le premier cas, les données utilisées dans le test sont visibles alors que dans le second, elles peuvent éventuellement être supprimées si elles ne servent pas par ailleurs.

On peut aussi ôter les instructions qui ne sont plus utiles. La détection de code mort permet par exemple de supprimer les éléments qui n'étaient utilisés que dans la branche supprimée.

Exemple 6

Dans la séquence suivante :

$$\texttt{x} = \text{c+1}\,;\ \text{a} = 2\,;\ \text{if (c) x += a}\,;\ \text{y} = \text{x+1}\,;$$

si on demande à sélectionner les dépendances de `y`, puis à supprimer la branche `if (c)`, on obtient :

$$\texttt{x} = \text{c+1}\,;\ \sout{\text{a = 2;}}\ \textbf{assert (!c);}\ \sout{\text{x += a;}}\ \text{y} = \text{x+1}\,;$$

où l'on voit que l'instruction `a=2;` disparaît puis qu'elle ne sert plus.

On verra ci-dessous comment on peut aussi déterminer que `x` vaut alors 1 à l'aide du calcul de WP, et que donc, `y` vaut 2 à l'aide de la propagation de constantes.

Le problème général lorsque l'on impose des contraintes, c'est qu'elles peuvent aussi influencer les résultats d'autres analyses comme l'analyse de valeur. On en verra un cas particulier ci-dessous avec la propagation de constante.

D.2.2 Passage à un point de programme

La contrainte dont nous venons de parler peut également être donnée implicitement en spécifiant un point de programme. Il s'agit alors d'être capable de supprimer ce qui ne sert pas lorsque l'exécution de la fonction passe par ce point. Une telle requête peut par exemple être utilisée dans le cadre d'une analyse d'impact.

Une première étape peut consister à supprimer les branches incompatibles avec le point en ne regardant que la syntaxe.

Exemple 7

Dans la séquence suivante :

$$\texttt{if}\ (\text{c})\ \text{P} : \text{x} \mathrel{+}= \text{a}\,;\ \texttt{else}\ \text{y} = \text{x+1}\,;$$

si l'utilisateur souhaite examiner le passage au point P, on peut éliminer la branche `else` :

$$\textbf{assert (c);}\ \text{P} : \text{x} \mathrel{+}= \text{a}\,;\ \sout{\texttt{else y = x+1;}}$$

Attention, ceci n'est vrai que si la séquence n'est pas dans une boucle...

Mais il peut être plus intéressant de calculer une précondition qui assure que l'on passe par le point donné, et d'utiliser ce résultat comme une contrainte.

Remarque 1 : Attention, il faut noter que cette précondition ne peut généralement pas être calculée par un simple WP car celui-ci assure qu'une propriété est vraie si la fonction est exécutée avec des entrées satisfaisant la précondition, mais il n'assure pas qu'il n'y a pas d'autres cas possibles. Ceci est du à l'approximation introduite par le calcul de WP des boucles.

D.2.3 Propagation de constantes

D'autres réductions peuvent être effectuées en exploitant les constantes du programme.

Exemple 8

Dans la séquence suivante :

```
x = c ? 1 : -1; if (x<0) f(x); else g(x);
```

si l'on étudie cette séquence avec une valeur initiale 0 pour `c` on aimerait savoir déterminer que :
- `x=-1`,
- donc que le second test est vrai,
- que donc `g` ne sera pas appelé,
- et que `f` sera appelé dans ce contexte avec la valeur `-1`.

On souhaite donc obtenir :

```
assert (c == 0); x = -1; f(-1);
```

Ce calcul peut facilement être effectué par une analyse de valeur externe, mais il faut pouvoir lui transmettre la contrainte (ici : $c = 0$;).

Dans l'outil précédent, le module de *slicing* (FLD) avait été couplé à un module de propagation de constante (CST) de la façon suivante :
- pour chaque fonction filtrée, on calcule les états CST correspondants,
- si une branche est supprimée à la demande de l'utilisateur, CST en est informé pour réduire l'état,
- si une constante est associée à une donnée, (par exemple lors de la spécialisation d'une fonction avec un paramètre constant) CST en est également informé,
- lors du calcul d'une fonction filtrée, pour chaque saut conditionnel, on interroge CST pour savoir si une branche est inaccessible ($\perp$), et si c'est le cas, les éléments correspondant sont marqués mM (code mort),
- pour chaque appel de fonction, si l'on sait déterminer qu'une ou plusieurs entrées sont constantes, on demande la spécialisation de cette fonction,
- pour chaque appel de fonction, si l'on sait déterminer qu'une ou plusieurs sorties sont constantes, on fournit l'information à CST pour qu'il puisse la propager.

Exemple 9

Reprenons l'exemple précédent où l'on sait que $c = 0$ en 1 :

```
/*1*/ x = c ? /*2*/1 : /*3*/-1; /*4*/if (x<0) /*5*/f(x); else /*6*/g(x);
```

On a donc :
- en `/*1*/` : $c = 0$,
- en `/*2*/` : $\bot$ puisque le test est faux,
- en `/*4*/` : $x = -1$,
- en `/*5*/` : toujours $x = -1$,
- en `/*6*/` : $\bot$ puisque le test est toujours vrai,

Cette séquence est donc réduite à :

```
x = c ? 1 : -1; if (x>0) f_1(x); else g(x);
```

où `f_1` est la fonction `f` spécialisée avec son entrée à -1.

Quand l'analyse permet de déterminer la valeur précise d'une variable, il peut également être intéressant d'utiliser cette information pour effectuer une transformation de programme juste avant de produire le résultat. On ne peut pas vraiment le faire avant, car on peut être amené à regrouper plusieurs fonctions où la valeur peut être différente.

L'implémentation des points suivants n'a pas été fait dans l'outil précédent, et il faudrait donc voir s'il est possible de les intégrer dans le nouveau :

- supprimer l'argument correspondant à une entrée constante. En effet, lors du calcul de la fonction spécialisée, on ne savait pas supprimer l'argument correspondant, mais on propageait tout de même la valeur pour supprimer les éventuelles branchesmortes.

  Exemple 10

  Dans l'exemple précédent, on peut spécialiser `f` en `f_1` car on sait que $x = -1$.
  Mais on aimerait aussi transformer : `void f_1 (int x) { ... }`
  en : `void f_1 (void) { int x = -1; ... }`

- supprimer le calcul d'une condition de `if` toujours vraie quand il n'y a pas de `else`. Cela n'avait pas été fait car dans le module de propagation de constante, il n'y avait pas de point de programme correspondant à la branche `else` manquante. L'état $\bot$ qui lui aurait été associé n'était donc pas là pour permettre cette suppression.

D.2.4 Utilisation de WP

Comme on l'a vu, la propagation de constante peut permettre de spécialiser les fonctions et de couper des branches de programme.

Ce type calcul s'effectue a priori par propagation avant. Pour faire de la propagation arrière, nous aimerions utiliser un calcul de WP.

Le principe d'utilisation serait le suivant :

Exemple 11

L'utilisateur s'intéresse uniquement à la branche *then* du *if*. On introduit donc l'assertion $y < 0$:

Programme source

```
int f (int c, int x) {

  int y = c ?  1 : -1;
  if (y < 0) {
    int z = y - 1;
    g (z);
    return z + x;
    }
  else
    return 0;
}
```

coupure de branche

```
int f (int c, int x) {

  int y = c ?  1 : -1;
  assert (y < 0); {
  int z = y - 1;
  g (z);
  return z + x;
  }


}
```

WP

```
int f (int c, int x) {
  /* (c == 0); */
  int y = c ?  1 : -1;
  assert (y < 0); {
    int z = y - 1;
    g (z);
    return z + x;
  }

}
```

Propagation de constante

```
int f (int c, int x) {
  /* (c == 0);  */
  int y =  /* 0 ? 1 : */ -1;
  assert ( -1 < 0); {
    int z =  -1 - 1;
    g ( -2);
    return  -2 + x;
  }

}
```

Sur cet exemple, l'utilisation du WP permet de déterminer que c vaut 0, et qu'il est alors possible de propager cette constante.

La principale difficulté de l'utilisation d'un tel calcul est le pilotage des opérations.

Le plus simple est de laisser l'utilisateur contrôler la propagation en utilisant des commandes interactives.

L'idée est de lui permettre de donner une contrainte, de la remonter à l'aide de WP, et d'en déduire d'autres propriétés, utilisable par exemple par l'analyse de valeur.

Attention, ici encore, comme dans la remarque 1 page 67, il faudra préalablement s'assurer que l'approximation introduite par le WP des boucles est bien dans le bon sens relativement à ce que l'on souhaite obtenir...

Bibliographie

[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.

[Chung et al.(2001)Chung, Lee, Yoon, and Kwon] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In SAC '01 : Proceedings of the 2001 ACM symposium on Applied computing, pages 605–609, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-287-5. URL http://doi.acm.org/10.1145/372202.372784.

[CodeSurfeur()] CodeSurfeur. Codesurfeur. URL http://www.codesurfer.com/.

[Comuzzi and Hart(1996)] J. J. Comuzzi and J. M. Hart. Program slicing using weakest preconditions. In FME '96 : Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods, pages 557–575, London, UK, 1996. Springer-Verlag. ISBN 3-540-60973-3. URL http://portal.acm.org/citation.cfm?id=729684.

[Daoudi et al.(2002)Daoudi, Ouarbya, Howroyd, Danicic, Marman, Fox, and Ward] M. Daoudi, L. Ouarbya, J. Howroyd, S. Danicic, M. Marman, C. Fox, and M. P. Ward. Consus : A scalable approach to conditional slicing, 2002. URL http://citeseer.ist.psu.edu/daoudi02consus.html.

[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.

[Fox et al.(2001)Fox, Harman, Hierons, and Danicic] C. Fox, M. Harman, R. Hierons, and S. Danicic. Backward conditioning : A new program specialisation technique and its application to program comprehension. In IWPC '01 : Proceedings of the 9th International Workshop on Program Comprehension, pages 89–97, Washington, DC, USA, 2001. IEEE Computer Society. URL http://citeseer.ist.psu.edu/fox01backward.html.

[Fox et al.(2004)Fox, Danicic, Harman, and Hierons] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. Consit : A fully automated conditioned program slicer. Software : Practice and Experience, 34(1) :15–46, 2004. URL http://dx.doi.org/10.1002/spe.556.

[Harman et al.(2001a)Harman, Hierons, Fox, Danicic, and Howroyd] M. Harman, R. Hierons, C. Fox, S. Danicic, and J. Howroyd. Pre/post conditioned slicing. In ICSM '01 : Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), page 138, Washington, DC, USA, 2001a. IEEE Computer Society. ISBN 0-7695-1189-9.

[Harman et al.(2001b)Harman, Hu, Munro, and Zhang] M. Harman, L. Hu, M. C. Munro, and X. Zhang. GUSTT : An amorphous slicing system which combines slicing and

transformation. In Working Conference on Reverse Engineering, pages 271–280, 2001b. URL http://citeseer.ist.psu.edu/harman01gustt.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.

[Lyle and Wallace(1997)] J. Lyle and D. Wallace. Using the unravel program slicing tool to evaluate high integrity software. In Proceedings of Software Quality Week, May 1997. URL http://citeseer.ist.psu.edu/lyle97using.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. Calcul de dépendances dans un programme c. Technical report, INRIA, juin 2007. (Note : *document de travail en cours d'évolution*).

[Pacalet and Baudin(2005)] A. Pacalet and P. Baudin. Analyse d'impact ; l'outil de slicing. Technical Report DTSI/SOL/05-141, CEA, mai 2005. (Note : *Il existe, en interne, une version plus récente de ce document*).

[Reps(1993)] T. Reps. The wisconsin program-integration system reference manual : Release, 1993. URL http://citeseer.nj.nec.com/reps93wisconsin.html.

[SlicingWebLinks()] SlicingWebLinks. Slicing web links. URL http://www.infosun.fmi.uni-passau.de/st/staff/krinke/slicing/.

[Tip(1995)] F. Tip. A survey of program slicing techniques. Journal of programming languages, 3 :121–189, 1995. URL http://citeseer.nj.nec.com/tip95survey.html.

[Unravel()] Unravel. Unravel. URL http://www.itl.nist.gov/div897/sqg/unravel/unravel.html.

[Ward(2002)] M. Ward. Program slicing via fermat transformations, 2002. URL http://citeseer.ist.psu.edu/ward02program.html.

[Weiser(1979)] M. Weiser. Program slices : formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor, 1979.

[Weiser(1981)] M. Weiser. Program slicing. In ICSE '81 : Proceedings of the 5th international conference on Software engineering, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6.

[Xu et al.(2005)Xu, Qian, Zhang, Wu, and Chen] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. SIGSOFT Softw. Eng. Notes, 30(2) :1–36, 2005. ISSN 0163-5948. URL http://doi.acm.org/10.1145/1050849.1050865.