Détermination de propriétés de flot de données pour améliorer les estimations de temps d'exécution pire-cas

Jordy Ruiz

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *21/12/2017* **par :**

Jordy Ruiz

Détermination de propriétés de flot de données pour améliorer les estimations de temps d'exécution pire-cas

---

JURY

Philippe CLAUSS	Professeur d'Université	Rapporteur
Nicolas HALBWACHS	Directeur de Recherche	Rapporteur
Sandrine BLAZY	Professeur d'Université	Examinateur
Jean-Paul BODEVEIX	Professeur d'Université	Examinateur
Christine ROCHANGE	Professeur d'Université	Directeur de thèse
Hugues CASSÉ	Maître de Conférences	Co-directeur de thèse

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (UMR 5505)
Directeur(s) de Thèse :
Christine ROCHANGE et *Hugues CASSÉ*
Rapporteurs :
Philippe CLAUSS et *Nicolas HALBWACHS*

Détermination de propriétés de flot de données pour améliorer les estimations de temps d'exécution pire-cas

Jordy Ruiz

Résumé

La recherche d'une borne supérieure au temps d'exécution d'un programme est une partie essentielle du processus de vérification de systèmes temps-réel critiques. Les programmes de tels systèmes ont généralement des temps d'exécution variables et il est difficile, voire impossible, de prédire l'ensemble de ces temps possibles. Au lieu de cela, il est préférable de rechercher une approximation du temps d'exécution pire-cas ou *Worst-Case Execution Time* (WCET).

Une propriété cruciale de cette approximation est qu'elle doit être *sûre*, c'est-à-dire qu'elle doit être garantie de majorer le WCET. Parce que nous cherchons à prouver que le système en question se termine en un temps raisonnable, une surapproximation est le seul type d'approximation acceptable.

La garantie de cette propriété de sûreté ne saurait raisonnablement se faire sans analyse statique, un résultat se basant sur une série de tests ne pouvant être sûr sans un traitement exhaustif des cas d'exécution. De plus, en l'absence de certification du processus de compilation (et de transfert des propriétés vers le binaire), l'extraction de propriétés doit se faire directement sur le code binaire pour garantir leur fiabilité.

Toutefois, cette approximation a un coût : un pessimisme – écart entre le WCET estimé et le WCET réel – important entraîne des surcoûts superflus de matériel pour que le système respecte les contraintes temporelles qui lui sont imposées. Il s'agit donc ensuite, tout en maintenant la garantie de sécurité de l'estimation du WCET, d'améliorer sa précision en réduisant cet écart de telle sorte qu'il soit suffisamment faible pour ne pas entraîner des coûts supplémentaires démesurés.

Un des principaux facteurs de surestimation est la prise en compte de chemins d'exécution sémantiquement impossibles, dits *infaisables*, dans le calcul du WCET. Ceci est dû à l'analyse par énumération implicite des chemins ou *Implicit Path Enumeration Technique* (IPET) qui raisonne sur un surensemble des chemins d'exécution. Lorsque le chemin d'exécution pire-cas ou *Worst-Case Execution Path* (WCEP), correspondant au WCET estimé, porte sur un chemin infaisable, la précision de cette estimation est négativement affectée.

Afin de parer à cette perte de précision, cette thèse propose une technique de détection de chemins infaisables, permettant l'amélioration de la précision des analyses statiques (dont celles pour le WCET) en les informant de l'infaisabilité de certains chemins du programme. Cette information est passée sous la forme de propriétés de flot de données formatées dans un langage d'annotation portable, FFX, permettant la communication des résultats de notre analyse de chemins infaisables vers d'autres analyses.

Les méthodes présentées dans cette thèse sont inclues dans le framework OTAWA, développé au sein de l'équipe TRACES à l'IRIT. Elles usent elles-mêmes d'approximations pour représenter les états possibles de la machine en différents points du programme. Ce sont des *abstractions* maintenues au fil de l'analyse, et dont la validité est assurée par des outils de la théorie de l'*interprétation abstraite*. Ces abstractions permettent de représenter de manière efficace – mais sûre – les ensembles d'états pour une classe de chemins d'exécution jusqu'à un point du programme, et de détecter d'éventuels points du programme associés à un ensemble d'états possibles vide, traduisant un (ou plusieurs) chemin(s) infaisable(s).

L'objectif de l'analyse développée, la détection de tels cas, est rendue possible par l'usage de solveurs SMT (Satisfiabilité Modulo des Théories). Ces solveurs permettent essentiellement de déterminer la satisfiabilité d'un ensemble de contraintes, déduites à partir des états abstraits construits. Lorsqu'un ensemble de contraintes, formé à partir d'une conjonction de prédicats, s'avère insatisfiable, aucune valuation des variables de la machine ne correspond à un cas d'exécution possible, et la famille de chemins associée est donc infaisable.

L'efficacité de cette technique est soutenue par une série d'expérimentations sur divers suites de benchmarks, reconnues dans le domaine du WCET statique et/ou issues de cas réels de l'industrie. Des heuristiques sont configurées afin d'adoucir la complexité de l'analyse, en particulier pour les applications de plus grande taille. Les chemins infaisables détectés sont injectés sous la forme de contraintes de flot linéaires dans le système de Programmation Linéaire en Nombres Entiers ou *Integer Linear Programming* (ILP) pilotant le calcul final de l'analyse WCET d'OTAWA. Selon le programme analysé, cela peut résulter en une réduction du WCET estimé, et donc une amélioration de sa précision.

Mots-clés : systèmes temps-réel, WCET, analyse statique, interprétation abstraite, SMT, chemins infaisables, langage machine.

Lookup of data flow properties to improve worst-case execution time estimations

Jordy Ruiz

Abstract

The search for an upper bound of the execution time of a program is an essential part of the verification of real-time critical systems. The execution times of the programs of such systems generally vary a lot, and it is difficult, or impossible, to predict the range of the possible times. Instead, it is better to look for an approximation of the *Worst-Case Execution Time*.

A crucial requirement of this estimate is that it must be *safe*, that is, it must be guaranteed above the real WCET. Because we are looking to prove that the system in question terminates reasonably quickly, an overapproximation is the only acceptable form of approximation.

The guarantee of such a safety property could not sensibly be done without static analysis, as a result based on a battery of tests could not be safe without an exhaustive handling of test cases. Furthermore, in the absence of a certified compiler (and technique for the safe transfer of properties to the binaries), the extraction of properties must be done directly on binary code to warrant their soundness.

However, this approximation comes with a cost : an important pessimism, the gap between the estimated WCET and the real WCET, would lead to superfluous extra costs in hardware in order for the system to respect the imposed timing requirements. It is therefore important to improve the *precision* of the WCET by reducing this gap, while maintaining the safety property, as such that it is low enough to not lead to immoderate costs.

A major cause of overestimation is the inclusion of semantically impossible paths, said *infeasible paths*, in the WCET computation. This is due to the use of the *Implicit Path Enumeration Technique* (IPET), which works on an superset of the possible execution paths. When the *Worst-Case Execution Path* (WCEP), corresponding to the estimated WCET, is infeasible, the precision of that estimation is negatively affected.

In order to deal with this loss of precision, this thesis proposes an infeasible paths detection technique, enabling the improvement of the precision of static analyses (namely for WCET estimation) by notifying them of the infeasibility of some paths of

the program. This information is then passed as data flow properties, formatted in the FFX portable annotation language, and allowing the communication of the results of our infeasible path analysis to other analyses.

The methods hereafter presented are included in the OTAWA framework, developed in TRACES team at the IRIT lab. They themselves make use of approximations in order to represent the possible states of the machine in various program points. These approximations are *abstractions* maintained throughout the analysis, and which validity is ensured by *abstract interpretation* tools. They enable us to represent the set of states for a family of execution paths up to a given program point in an efficient – yet safe – way, and to detect the potential program points associated to an empty set of possible states, signalling one (or several) infeasible path(s).

As the end goal of the developed analysis, the detection of such cases is made possible by the use of *Satisfiability Modulo Theory* (SMT) solvers. Those solvers are notably able to determine the satisfiability of a set of contraints, which we deduct from the abstract states. If a set of constraints, derived from a conjonction of predicates, is unsatisfiable, then there exists no valuation of the machine variables that match a possible execution case, and thus the associated infeasible paths are infeasible.

The efficiency of this technique is asserted by a series of experiments on various benchmarks suites, some of which widely recognized in the domain of static WCET, some others derived from actual industrial applications. Heuristics are set up in order to soften the complexity of the analysis, especially for the larger applications. The detected infeasible paths are injected as *Integer Linear Programming* (ILP) linear data flow constraints in the final computation for the WCET estimation in OTAWA. Depending on the analysed program, this can result in a reduction of the estimated WCET, thereby improving its precision.

Keywords : real-time systems, WCET, static analysis, abstract interpretation, SMT, infeasible paths, machine language.

Remerciements

Je remercie Hugues Cassé, encadrant bienveillant, patient, humain, et remarquable pédagogue. Merci d'être toujours disponible pour m'écouter et m'orienter, plein de pistes et de solutions. Je suis reconnaissant d'avoir été aidé et inspiré par l'excellence de ce savoir-faire.

Je remercie l'ensemble de l'équipe pour leur accueil chaleureux, et pour avoir créé ce cadre de vie et de travail si agréable. Plus précisément, merci à Willie, Vincent, Haluk, Christine, Armelle, Marianne, Thomas, Jakob, Pascal et Pascal.

Je remercie en particulier Christine pour ses méticuleuses relectures, ainsi que Vincent et Marianne pour leur collaboration.

J'adresse aussi ma gratitude à Nicolas Halbwachs et Philippe Clauss, pour avoir accepté la fonction de rapporteur, ainsi qu'à Sandrine Blazy et Jean-Paul Bodeveix, pour leur participation au jury.

Je remercie Mathias pour m'avoir recommendé cette thèse et ces encadrants.

Enfin, à Lotta, pour son confort *vaikeina aikoina*.

Table des matières

Table des figures

Liste des algorithmes

1

Introduction

Problématique, systèmes temps-réel critiques

Les erreurs, pannes, comportements inattendus ou lenteurs excessives des programmes informatiques font partie de la vie quotidienne de nombreuses personnes. Souvent, les conséquences sont bénignes : il suffit d'attendre, de corriger, de relancer le programme voire le système. Parfois, ces problèmes entraînent des désagréments de l'ordre de la perte de données, ou de ressources, ou incapacitent un système jusqu'à l'intervention d'un réparateur.

Toutefois, pour certains systèmes, les conséquences d'un comportement défectueux peuvent avoir des conséquences humainement et/ou financièrement lourdes, voire catastrophiques, tel l'échec du vol inaugural de la fusée Ariane 5, qui explosa en vol. De tels systèmes sont dits *critiques*. Pour les systèmes embarqués comme des satellites, c'est parfois la difficulté d'intervention qui justifie la criticité du système. Certaines applications, dites *temps-réel*, doivent satisfaire en particulier des contraintes temporelles. Lorsque des dépassements de ces contraintes peuvent conduire à des situations critiques, elles sont dites de *temps-réel strict*, comme par exemple pour le système de

pilotage d'un avion.

Après leur développement, les programmes sont généralement testés afin de minimiser ces risques, souvent par une longue série d'exécutions visant à détecter d'éventuels problèmes avant le déploiement. Cependant, tous les cas d'utilisation ne peuvent pas être couverts par ces méthodes expérimentales, qui n'offrent donc aucune garantie. Les systèmes critiques nécessitent une analyse *statique* des programmes, de préférence directement sur leur forme exécutable, pour maximiser la précision et la fiabilité des résultats obtenus, qui ne devront ainsi pas passer par un difficile processus de transfert de propriétés depuis un langage de plus haut niveau.

La recherche d'une borne supérieure au temps d'exécution d'un programme est une partie essentielle du processus de vérification de systèmes temps-réel critiques. Les programmes de tels systèmes ont généralement des temps d'exécution variables et il est difficile, voire impossible, de prédire l'ensemble de ces temps possibles. Au lieu de cela, il est préférable de rechercher une approximation du Temps d'Exécution Pire-Cas ou *Worst-Case Execution Time* (WCET).

Une propriété cruciale de cette approximation est qu'elle doit être *sûre*, c'est-à-dire qu'elle doit être garantie être une majoration du WCET réel. Parce que nous cherchons à prouver que le système en question se termine en un temps raisonnable, une surapproximation est le seul type d'approximation acceptable.

Toutefois, cette approximation a un coût : un pessimisme – l'écart entre le WCET estimé et le WCET réel – important entraîne des surcoûts superflus de matériel pour que le système respecte les contraintes temporelles qui lui sont imposées. Il s'agit donc ensuite, tout en maintenant la garantie de sûreté de l'estimation du WCET, d'améliorer sa précision en réduisant cet écart de telle sorte qu'il soit suffisamment faible pour ne pas entraîner des coûts supplémentaires démesurés.

Un des principaux facteurs de surestimation est la prise en compte de chemins d'exécution sémantiquement impossibles, dits *infaisables*, dans le calcul du WCET. Ceci est dû à l'analyse par énumération implicite des chemins ou *Implicit Path Enumeration Technique* (IPET) qui raisonne sur un surensemble des chemins d'exécution. Lorsque le chemin d'exécution pire-cas ou *Worst-Case Execution Path* (WCEP), correspondant au WCET estimé, porte sur un chemin infaisable, la précision de cette estimation est négativement affectée.

Afin de parer à cette perte de précision, ces chemins infaisables doivent être détec-

tés et cette information passée à d'autres analyses statiques pour le WCET. Afin de garder une portée large, notre analyse doit être aussi indépendante que possible des particularités des différents outils de calcul du WCET, de telle sorte que les résultats obtenus puissent être exploités par toute analyse statique de programmes binaires.

Contributions

Les contributions de cette thèse sont les suivantes.

Nous proposons un domaine abstrait et des outils pour l'interprétation d'un programme binaire avec cette abstraction, et l'adaptons pour gérer les difficultés de l'analyse de code binaire, comme l'adressage des données dans la pile ou l'arithmétique sur n bits. Les états abstraits définis par ce domaine représentent l'exécution de régions du programme, en fonction d'un ensemble de paramètres. Nous montrons comment ces états peuvent être composés pour rendre l'analyse de programme modulaire.

Nous proposons une méthode pour détecter des chemins infaisables par analyse statique sur un programme binaire, en exploitant des techniques modernes de résolution de problèmes Satisfiabilité Modulo des Théories (SMT). En particulier, des avancées récentes dans le domaine des problèmes de satisfiabilité ont permis l'extension des solveurs SMT pour l'extraction efficace de sous-ensembles minimaux insatisfiables (*unsat cores*). Nous utiliserons cette fonctionnalité des solveurs pour factoriser et minimiser des familles de chemins infaisables, facilitant ainsi leur exploitation.

Enfin, nous implémentons l'ensemble de ces techniques dans un outil d'analyse statique, et testons son efficacité, en termes de découverte de chemins infaisables et de leur impact sur le WCET, sur des suites de programmes, dont certaines sont issues d'applications temps-réel pour des systèmes critiques.

Organisation du manuscrit

Ce document est structuré de la façon suivante :

Le Chapitre 2 pose le contexte de la thèse, et détaille le problème ciblé. Nous y dressons un état de l'art des techniques permettant d'effectuer et d'améliorer la précision d'une évaluation sûre du WCET, en particulier grâce aux techniques d'analyse statique, d'interprétation par énumération des chemins et d'interprétation abstraite. Nous moti-

vons le problème des chemins infaisables et détaillons les techniques existantes autour de leur détection, de leur expression et de leur exploitation pour l'amélioration du WCET, en soulignant leurs avantages et inconvénients, notamment en terme de sûreté, de précision et de complexité.

Le Chapitre 3 pose les fondements des travaux de la thèse. Nous y détaillons les structures utilisées pour représenter le programme et la machine, ainsi que des abstractions de ces derniers. Le choix et la construction des abstractions est un point crucial pour l'efficacité d'une telle analyse de programme. Nous procédons alors par raffinements successifs d'abstractions des états de la machine, en partant des états concrets vers un domaine plus complexe et adapté à nos fins. Enfin, des outils d'interprétation abstraite sont mis en place pour permettre la vérification de la validité de l'analyse et de ses résultats, dans une optique de surestimation du WCET.

Le Chapitre 4 développe des méthodes de parcours de programmes binaires, afin de construire une analyse de flot de données, en se basant sur les structures définies au Chapitre 3. Nous procédons une fois de plus par raffinements successifs pour définir l'algorithme de parcours de graphe nous servant à analyser un programme et en extraire des propriétés. Nous exploitons des propriétés de composabilité des structures définies au Chapitre 3 pour développer une analyse efficace, capable de détecter des propriétés dépendantes ou non des contextes d'appels, ou encore valides pour toute itération d'une boucle.

Le Chapitre 5 utilise les informations de flot de données positionnées sur les points importants du programmes par l'analyse développée au Chapitre 4 pour détecter des chemins infaisables. Nous testons pour cela la satisfiabilité des états abstraits issus de l'interprétation de différents chemins du programme, c'est-à-dire que nous vérifions qu'ils indiquent au moins un état possible de la machine prenant ce chemin. Nous transformons ce test en problème de décision, et utilisons un outil de résolution de problème SMT pour y répondre. Dans le cas où ce solveur SMT signale une insatisfiabilité de ce problème, un tel état du programme est garanti impossible, et le chemin associé est par conséquent infaisable. L'usage d'une fonctionnalité particulière à certains solveurs SMT, l'extraction de sous-ensemble minimaux insatisfiables, nous permet d'exprimer les chemins infaisables ainsi obtenus sous la forme de conflits entre arcs. Nous factorisons ainsi les chemins infaisables obtenus en les représentant sous cette forme minimale, en incluant aussi peu d'arcs que possible. Nous facilitons ainsi l'exploitation de ces chemins infaisables, et réduisons le coût de complexité de leur prise en compte

pour l'amélioration du calcul du WCET.

L'efficacité de l'analyse est ensuite attestée par son implantation dans le *framework* d'analyse statique de programme OTAWA, et par une série d'expérimentations sur des benchmarks pertinents au domaine des systèmes temps-réel critiques. Nous démontrons une complexité d'analyse raisonnable, capable d'analyser des programmes de bonne taille, ainsi que, malgré une grande variabilité, des gains de précision sur le WCET estimé après prise en compte des chemins infaisables détectés par injection de contraintes de flot de contrôle dans le calcul du WCET.

2

Contexte

Sommaire

Ce chapitre vise à la fois à établir le contexte et à présenter les travaux fondateurs, connexes ou alternatifs à ceux qui seront développés plus loin dans cette thèse. Il cible le problème et motive notre approche par rapport aux solutions existantes.

2.1 Temps d'exécution pire-cas (WCET)

2.1.1 Introduction : systèmes temps-réel critiques

La garantie de fiabilité (ou certification) d'un système critique passe par trois points majeurs :

- la preuve de *correction* : il s'agit là de prouver que les fonctions du systèmes respectent une spécification qui correspond au comportement attendu ;
- la preuve de *terminaison* : il s'agit là de prouver que le programme termine ;
- la vérification de *contraintes non-fonctionnelles* : il peut s'agir de prouver que le programme respecte des limites de consommation de ressources, des contraintes de disponibilité, des délais d'exécution, etc.

Ce dernier point est critique pour les systèmes temps-réels exécutant des tâches dont le temps d'exécution doit absolument être inférieur à des normes de sûreté. Un dépassement de ces normes, même extrêmement improbable, mettrait en danger la fiabilité de certains systèmes, comme par exemple le système de freinage d'une voiture. C'est sur ce type de système que les batteries de test sont particulièrement inefficaces : une anomalie se produisant dans un milliardième des cas d'exécution ne serait probablement pas détectée, mais elle aurait des conséquences sur un système largement répandu et fréquemment utilisé, comme le système de freinage d'une voiture. Un exemple récent de telles conséquences est celui d'un accident fatal dû à un dysfonctionnement du système de freinage de la 2005 Toyota Camry, en Septembre 2007 [92]. Le processus de certification n'avait pas été intégralement respecté, et la compagnie automobile fut condamnée.

Ce cas souligne l'importance d'une évaluation fiable du Temps d'Exécution Pire-Cas ou *Worst-Case Execution Time* (WCET) pour les systèmes temps-réel critiques.

Remarque. La portée de cette thèse se limitant à cette dernière partie de la certification, il sera fait l'hypothèse que les programme analysés sont corrects et se terminent.

2.1.2 Problème

Le problème de l'évaluation du WCET est un problème beaucoup trop difficile [1] pour qu'une réponse exacte puisse être donnée pour des programmes non-triviaux, en particulier lorsque l'on doit prendre en compte des effets complexes du matériel (caches...). A défaut d'en connaître la valeur exacte, il s'agit donc de trouver une *surestimation* de ce temps d'exécution pire-cas.

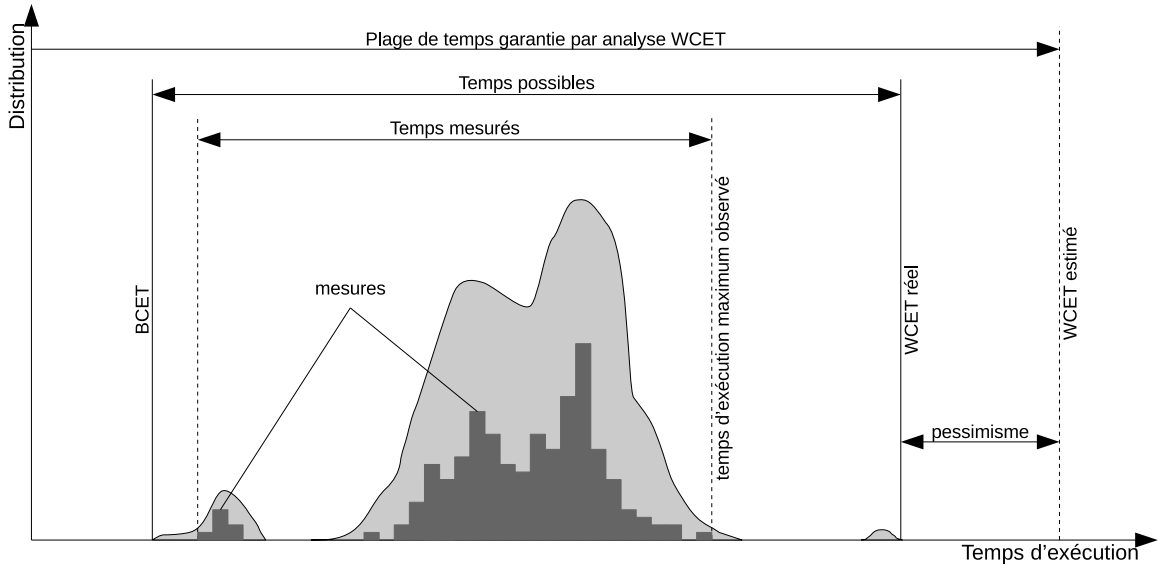


Figure 2.1 – Évaluation du WCET

La Figure 2.1 présente ce problème. La courbe délimitant la zone claire représente la distribution des temps d'exécution, c'est-à-dire les différents temps d'exécution que le programme peut prendre et leur probabilité sur un ensemble défini de cas d'utilisation possibles. L'aire de cette courbe délimite à droite le WCET réel, le temps d'exécution le plus élevé que le programme peut effectivement avoir, et à gauche le plus faible, parfois appelé Temps d'Exécution Meilleur-Cas ou *Best-Case Execution Time* (BCET).

L'aire plus sombre désigne les résultats qui pourraient être obtenus pour une série de tests. Cette série de tests détermine la valeur du temps d'exécution maximum observé : c'est une sous-estimation expérimentale du WCET.

Une analyse visant à évaluer le WCET pour un système critique doit être :

- *sûre*, c'est-à-dire que le WCET obtenu doit être garanti supérieur au WCET réel (cf. Figure 2.2, inspirée de Engblom et al. [39]) ;

1. Sa résolution implique d'ailleurs celle du problème de terminaison, qui est même indécidable dans le cas général, pour des programmes trop complexes.

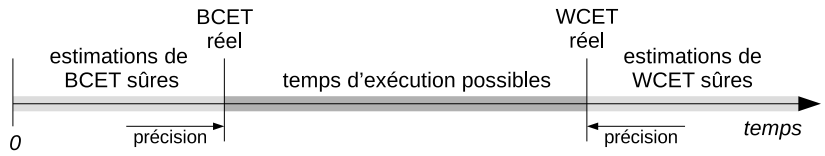


Figure 2.2 – Sûreté de l'estimation du WCET

- *précise*, c'est-à-dire aussi proche du WCET réel que se peut, tout en restant toutefois une surapproximation.

Cette approximation a un coût : afin que le système puisse garantir les délais imposés, une surestimation importante du WCET peut entraîner une surestimation des besoins matériels, et des surcoûts superflus. La différence de temps entre le WCET réel et le WCET surestimé est le *pessimisme*. L'objectif de l'affinage des analyses WCET est donc de réduire ce pessimisme tout en garantissant leur validité, c'est-à-dire en fournissant la preuve que l'estimation est toujours supérieure au WCET réel. L'efficacité de ces analyses est liée au pessimisme qu'elles introduisent (idéalement nul).

Ceci rend l'efficacité des analyses de calcul du WCET difficiles à évaluer sur des programmes non-triviaux : le WCET réel étant inconnu pour de tels programmes, le pessimisme l'est lui aussi. Il est possible de comparer le WCET obtenu par différentes analyses sur un même programme ou encore de comparer le résultat au temps d'exécution maximum *observé*, mais une estimation du WCET éloignée des résultats expérimentaux n'est pas nécessairement due à un pessimisme important. Plutôt que d'être imputable à une imprécision de l'analyse, un tel écart peut avoir pour cause un échec des jeux de tests employés à capturer des scénarios entraînant des temps d'exécution proche du WCET réel.

2.1.3 Approches

Différentes approches existent pour l'estimation du WCET. Wilhem et al. [106] en donnent en 2008 une large vue d'ensemble, en listant de nombreuses techniques d'analyses et outils commerciaux.

Certaines méthodes visent à évaluer le WCET à l'aide d'une série extensive de mesures (techniques dites *measurement-based*), souvent en cherchant à borner le pessimisme à l'aide de calculs probabilistes [28, 16, 35], comme le fait l'outil pWCET [17] (ou RapiTime [85], sa version commerciale). Il est possible de mesurer des temps d'exécu-

tion sur un système de plusieurs manières, soit en injectant du code d'instrumentation, soit de manière non-intrusive en utilisant du matériel externe au système (analyseur logique). Le WCET ainsi estimé n'étant pas garanti supérieur au WCET réel, il s'agit ensuite de prouver les temps d'exécution supérieurs au WCET estimé comme étant d'une probabilité suffisamment faible pour être négligée.

Les limitations de ce type d'analyse viennent de la difficulté de connaître la répartition probabiliste des cas d'utilisation : certaines combinaisons de paramètres peuvent survenir à des fréquences élevées, parfois de manière inattendue. De plus, il n'est pas toujours possible d'initialiser le matériel pour tester certaines conditions.

L'analyse statique apparaît ainsi souvent comme la seule solution capable de garantir une surestimation du WCET.

2.2 Analyse statique pour le WCET

L'analyse statique consiste à obtenir des propriétés d'un programme sans l'exécuter. Elle nécessite de modéliser sa structure ainsi que le système sur lequel il est exécuté.

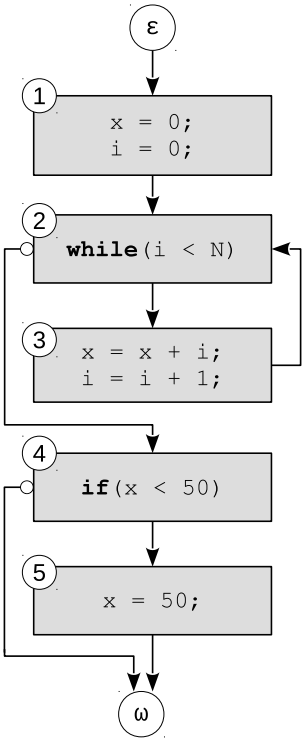


Figure 2.3 – Exemple de CFG d'un programme C

2.2.1 Représentation structurelle d'un programme

En analyse statique, un programme est communément représenté sous la forme d'un graphe, appelé le Graphe de Flot de Contrôle ou *Control Flow Graph* (CFG) [5], illustré sur la Figure 2.3. Le CFG est une abstraction, et une surapproximation dans le sens où il représente un surensemble des chemins d'exécution réellement possibles dans le programme. Les conséquences de cette propriété des CFG sur l'évaluation du WCET sont développées dans la section 2.4 et seront un point central de cette thèse.

Nous définissons d'abord la notion de bloc de base.

Définition 2.1. Un *bloc de base* (parfois abrégé BB) est une séquence maximale d'instructions issue du programme respectant les deux conditions suivantes :

(i) Un branchement (saut d'instruction) ne peut se faire qu'à partir de la dernière instruction (pas de branchement sortant de l'intérieur d'un bloc) ;

(ii) Le programme ne peut entrer dans un bloc de base que par la première instruction (pas de branchement entrant à intérieur d'un bloc).

Les blocs de base sont ainsi des parties de code composées d'instructions s'exécutant toujours séquentiellement. Le programme est découpé en blocs de base de manière à avoir aussi peu de blocs que possible. Nous pouvons maintenant définir un CFG.

Définition 2.2. Le CFG d'un programme est un graphe orienté $G = (V, E, \epsilon, \omega)$, où

- L'ensemble des sommets V correspond à l'ensemble des blocs de base du programme ;
- L'ensemble des arêtes $E \subseteq V^2$ correspond à l'ensemble des transitions possibles : $\forall v_1, v_2 \in V$, $(v_1 \rightarrow v_2) \in E$ si et seulement s'il existe un chemin d'exécution qui passe dans v_2 immédiatement après v_1.
- ϵ et ω sont des blocs de base (virtuels, vides d'instructions) correspondant respectivement à l'entrée et à la sortie du programme. Dans le cas où le programme a plusieurs points d'entrée ou de sortie possibles, ces sommets

sont reliés à chacun de ces points.

Il est généralement admis qu'un CFG ne comporte qu'une seule composante connexe (c'est-à-dire que tous ses nœuds sont atteignables).

Remarque. Lorsqu'un bloc se termine par un branchement conditionnel, et est donc relié à plusieurs arcs sortants, l'arc qui correspond à une exécution séquentielle du programme est dit *non pris* (la condition du branchement est fausse). Les autres arcs, représentant le cas où le branchement a eu lieu, sont dits *pris*.

Dans les CFG de code source, il n'y a pas d'instruction de branchement explicite. Par notation, l'arc où la condition est *fausse* est donc annoté par un inverseur : ⟜.

Nous définissons enfin les notions de chemins et de chemins d'exécution :

Définition 2.3. Un *chemin* dans un CFG $G = (V, E, \epsilon, \omega)$ (ou plus généralement dans un graphe (V, E)) est une séquence d'arêtes $e_1.e_2 \dots e_n \in E^\star$ *consécutives*, c'est-à-dire telles que

$$\forall i \in [1, n-1],\ \exists v, v', v'' \in V,\ \begin{cases} e_i = v \to v' \\ e_{i+1} = v' \to v'' \end{cases}$$

Un *chemin d'exécution* dans un CFG $G = (V, E, \epsilon, \omega)$ est un chemin $e_1.e_2 \dots e_n \in E^\star$ tel qui commence par l'entrée de G et finit par sa sortie, c'est-à-dire tel que

$$\exists v, v' \in V,\ \begin{cases} e_1 = \epsilon \to v \\ e_n = v' \to \omega \end{cases}$$

Le temps d'exécution pire-cas d'un programme correspond nécessairement à un chemin d'exécution : c'est le Chemin d'Exécution Pire-Cas ou *Worst-Case Execution Path* (WCEP).

La construction de CFG peut poser quelques difficultés, ou résulter en des formes difficiles à analyser, nous poussant à effectuer des transformations conservatives des chemins d'exécution. Ce sujet sera traité plus loin en section 3.1.2, en particulier à propos du problème des boucles irrégulières, des branchements dynamiques ou des schémas d'appels récursifs

2.2.2 Niveau de représentation du code

L'analyse d'un programme pose la question du choix du niveau de code analysé.

Plus le code est de haut niveau, plus il est facile d'en extraire la sémantique du programme. Par exemple, il est plus aisé d'obtenir les bornes de boucle sur du code source que sur de l'assembleur ; la structure du programme y est plus évidente. Certaines opérations simples dans du code source (C, par exemple) comme la division par une constante ou des additions flottantes peuvent être traduites à la compilation en code assembleur très difficile à interpréter. Ces obscures transformations peuvent être motivées par un souci d'optimisation (calcul d'une division par une constante sans boucle, par exemple) ou une nécessité due à la pauvreté du jeu d'instructions (absence de calcul flottant natif sur ARM9, par exemple). Propager des informations de flot issues du code source vers le binaire peut se révéler complexe, en particulier en présence d'optimisations [62, 63]. Les effets de ces optimisations ne peuvent pas être ignorés car, si elles améliorent le cas moyen, elles peuvent parfois aussi dégrader le WCET.

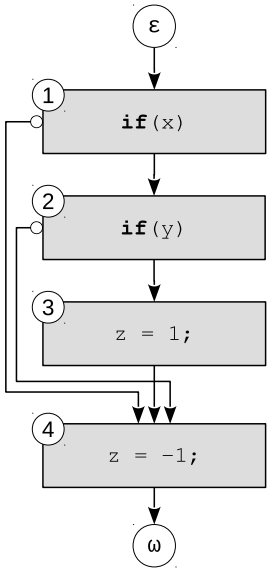


Figure 2.4 – Évaluation en circuit court d'une condition `x && y`

D'un autre côté, l'analyse sur le code binaire est potentiellement plus précise : elle travaille directement sur la forme du programme qui sera exécutée par le processeur, est capable d'exploiter toute optimisation du compilateur et peut tracer précisément l'exécution du programme dans le matériel. Certaines parties du code n'apparaissent même qu'à la compilation : c'est le cas, par exemple, des fonctions d'émulation. Le code binaire fait également apparaître les chemins issus de l'évaluation en circuit court des expressions logiques [2], comme sur la Figure 2.4 qui représente le CFG d'un programme compilé à partir du code C `if(x && y) z = 1; else z = -1;`. Cette figure illustre la décomposition de l'évaluation de l'expression logique `x && y` en deux tests et deux branchements conditionnels, faisant apparaître un chemin d'exécution invisible dans le code source.

Les avantages de cette approche motivent le choix de développer l'intégralité des travaux de cette thèse sous la forme d'analyse sur le code binaire. Un inconvénient de ce

choix est que l'analyse devient propre à un jeu d'instructions assembleur en particulier. Il est toutefois possible de contourner ce problème en raisonnant par l'intermédiaire une abstraction des jeux d'instructions, comme celle qui sera proposée dans la section 3.1.1.

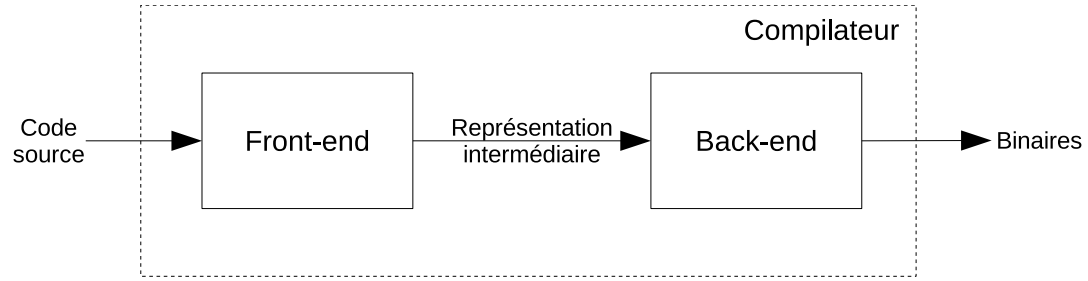


Figure 2.5 – Vue d'ensemble du procédé de compilation

Enfin, d'autres analyses se basent sur une représentation intermédiaire (parfois abrégée IR, pour *Intermediate Representation*) du programme propre à un compilateur (Figure 2.5). Ce code intermédiaire, généré et utilisé par le compilateur pour relier la partie *front-end* (analyse du code source) et *back-end* (génération du code binaire) de la compilation, a une structure beaucoup plus proche de celle du binaire que le code source. Il est indépendant de l'architecture vers laquelle le code est compilé, l'analyse ne perd donc pas en généralité mais reste dépendante d'un compilateur.

Dans tous les cas, la fiabilité de l'analyse dépendra de celle du compilateur. Ainsi, l'usage de compilateurs certifiés (comme par exemple Compcert [61], formellement prouvé avec l'assistant de preuve Coq) peut être requis pour assurer des garanties sur les résultats obtenus par analyse du programme.

2.2.3 La méthode IPET pour le WCET

L'Énumeration Implicite des Chemins ou *Implicit Path Enumeration Technique* (IPET) [64, 84] est une méthode numérique pour estimer le WCET d'un programme à partir d'un CFG. Chaque bloc de base $b \in V$ est annoté par un temps t_b correspondant à son temps d'exécution maximum, et par un entier naturel x_b correspondant au nombre d'exécutions maximum de ce bloc. Le but est d'obtenir dans un premier temps un système de contraintes numériques entières modélisant le flot du programme, puis de chercher le maximum possible de la somme des temps d'exécution t_b de chaque bloc b pondérée par leurs nombres d'exécutions respectifs x_b. Il s'agit donc de maximiser la

fonction objectif

$$\sum_{b \in V} x_b t_b$$

pour obtenir une surestimation du WCET. Si cette technique ne permet pas d'identifier le WCEP, elle nous permet toutefois d'obtenir un surensemble de chemins le contenant.

2.2.3.1 Évaluation du temps d'exécution d'une instruction

L'estimation du temps maximum pris pour l'exécution de chaque instruction du programme est à première vue une tâche assez simple pour des systèmes critiques : nous pouvons nous appuyer sur les renseignements du manuel d'instruction du constructeur pour évaluer cette borne supérieure. Il suffirait donc d'extraire ces informations pour calculer automatiquement le temps d'exécution maximal d'un bloc de base. En pratique, l'architecture d'un système est en beaucoup trop complexe pour être modélisée fidèlement, en particulier pour des machines modernes. Les processeurs sont souvent aidés par plusieurs couches de mémoire cache, et les mécanismes qui dictent les politiques d'éviction dans ces caches rendent souvent l'état du système à un point du programme difficile à prévoir. C'est sans compter les problèmes de pipeline (l'exécution des instructions se chevauche), de prédiction de branchement, d'ordonnancement, etc.

De très nombreuses techniques [6, 38, 72, 40] existent pour la modélisation d'architectures plus ou moins complexes [81]. Plutôt que de chercher à modéliser exactement l'état du système à tout moment, des approximations conservatives sont faites. Ainsi, une approche naïve pour prendre en compte les temps d'accès aux caches est de considérer tous les accès comme des *miss*. Cette approche, sauf en présence d'anomalies temporelles [2] [48], ne met pas en danger la validité du WCET obtenu, mais elle produit un WCET très surestimé. Le sujet des caches est largement traité dans la littérature ; souvent, afin de modéliser plus précisément les effets de cache, les points d'accès à la mémoire sont classifiées selon des catégories [40] : *always hit*, *always miss*, *persistent* (la donnée reste dans le cache une fois chargée, et ne cause donc qu'un seul *miss*), ou *not classified* (on considérera le pire-cas, soit *always miss*). Il est ensuite aisé de déduire de cette catégorisation les pénalités de temps d'accès à appliquer.

2. Les anomalies temporelles sont des comportements contre-intuitifs du programme où un scénario d'exécution peut avoir un temps d'exécution localement faible mais majorer le WCET du programme entier.

Une fois le temps d'exécution pire-cas de chaque instruction déterminé (ou plutôt, surestimé), les contraintes correspondantes sont générées. Par exemple, si le bloc b est garanti s'exécuter en au plus 140 cycles, la contrainte $t_b = 140$ apparaît.

2.2.3.2 Système de contraintes numériques de flot

La génération du système de contraintes représentant les informations de flot issues du CFG se base sur le principe de la loi des nœuds. Pour l'énoncer, chaque arc e du CFG sera également également annoté par un compteur x_e.

Théorème 2.1. *Pour tout bloc du CFG, le nombre d'exécutions de ce bloc est égal à la somme du nombre d'exécutions de chaque arc y entrant, et à la somme du nombre d'exécutions de chaque arc en sortant.*

Si $G = (V, E, \epsilon, \omega)$ est un CFG, alors

$$\forall v \in V, \sum_{v' \in V,\, (v \to v') \in E} x_{v \to v'} = x_v = \sum_{v' \in V,\, (v' \to v) \in E} x_{v' \to v}$$

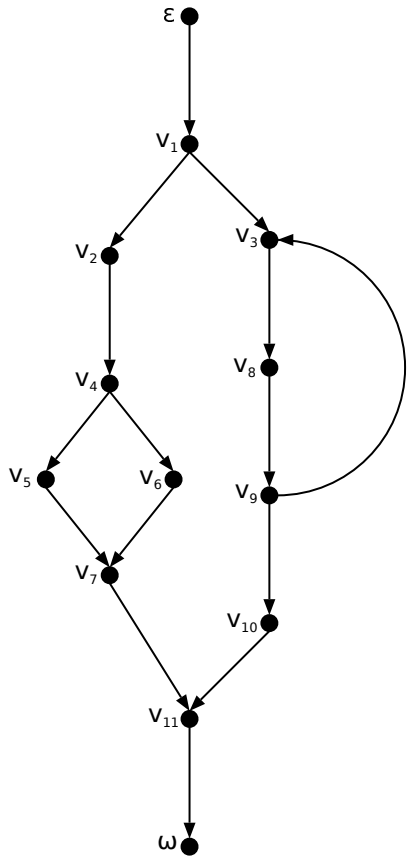
Il est naturel que, pour chaque exécution du programme, le bloc d'entrée ϵ et le bloc de sortie ω du CFG soient chacun exécutés exactement une fois.

$$x_\epsilon = x_\omega = 1$$

Nous cherchons maintenant une technique pour encoder ces contraintes et maximiser le temps d'exécution représenté par la fonction objectif $\sum_b x_b.t_b$. Ce problème est efficacement traité par la méthodologie de Programmation Linéaire en Nombres Entiers (PLNE) ou *Integer Linear Programming* (ILP) [27]. Différents outils apportent déjà une solution à ce problème, comme *lp-solve* [15], *opbdp* [13], ou encore l'optimiseur commercial d'IBM, CPLEX [33]. Une fois toutes les contraintes injectées, la maximisation de la fonction objectif donne une surestimation du WCET.

Les contraintes ainsi générées ne suffisent cependant pas pour des programmes avec boucles ; le WCET ainsi obtenu sera toujours infini, correspondant au cas d'exécution où le programme reste dans une boucle à l'infini. Il faut donc rajouter des contraintes correspondant à des *bornes de boucles* : il s'agit de donner le maximum d'itérations de chaque boucle du programme.

La Figure 2.6 illustre cette méthode en faisant la liste des contraintes de flot de données caractérisant un CFG.



(a) CFG

$$
\begin{aligned}
x_{\epsilon\to 1} &= x_1 &&= x_{1\to 2} + x_{1\to 3}\\
x_{1\to 2} &= x_2 &&= x_{2\to 4}\\
x_{1\to 3} + x_{9\to 3} &= x_3 &&= x_{3\to 8}\\
x_{2\to 4} &= x_4 &&= x_{4\to 5} + x_{4\to 6}\\
x_{4\to 5} &= x_5 &&= x_{5\to 7}\\
x_{4\to 6} &= x_6 &&= x_{6\to 7}\\
x_{5\to 7} + x_{6\to 7} &= x_7 &&= x_{7\to 11}\\
x_{3\to 8} &= x_8 &&= x_{8\to 9}\\
x_{8\to 9} &= x_9 &&= x_{9\to 3} + x_{9\to 10}\\
x_{9\to 10} &= x_{10} &&= x_{10\to 11}\\
x_{9\to 10} + x_{10\to 11} &= x_{11} &&= x_{11\to\omega}
\end{aligned}
$$

(b) Contraintes d'arc dérivées de la loi des nœuds

$$
\begin{aligned}
x_{\epsilon\to 1} &= x_\epsilon = 1\\
x_{11\to\omega} &= x_\omega = 1\\
x_{9\to 3} &\leq 10
\end{aligned}
$$

(c) Contraintes de flot supplémentaires

$$
\begin{aligned}
\text{WCET} = \max\big(&x_1t_1 + x_2t_2 + x_3t_3 + x_4t_4 + x_5t_5\\
&+ x_6t_6 + x_7t_7 + x_8t_8 + x_9t_9 + x_{10}t_{10} + x_{11}t_{11}\big)
\end{aligned}
$$

(d) Fonction objectif

Figure 2.6 – Exemple de système de contraintes extrait d'un CFG

2.2.3.3 Obtention des bornes de boucles

Il est critique dans le cadre de l'évaluation du WCET de pouvoir borner toutes les boucles d'un programme, sans quoi, le nombre de chemins d'un programme est infini.

Les bornes de boucles peuvent être obtenues de deux manières : manuellement, par annotation d'un expert, ou automatiquement, par détection à partir du code. Diverses techniques efficaces permettent la dérivation de bornes de boucles à partir de code source [19, 66, 88, 69], mais il est difficile de faire la correspondance de manière fiable

avec le code binaire, en particulier en présence d'optimisations du compilateur [62].

D'autres méthodes peuvent obtenir automatiquement des bornes de boucles sur le code binaire, comme par exemple celle de Cullmann et Martin [34]. Cette technique, implémentée dans l'outil aiT [101] et adaptée à de nombreuses architectures, semble donner de bons résultats sur des programmes de taille raisonnable. L'outil SWEET permet également la détection de bornes de boucles sur le code binaire [47], mais s'appuie sur des informations issues du compilateur (et en reste donc dépendant). Enfin, il existe des approches visant à valider des bornes de boucles obtenues sur le source sur du code binaire décompilé [97].

2.2.4 Conclusion

Les techniques présentées jusqu'ici suffisent en théorie pour faire l'évaluation statique et fiable du WCET d'un programme, pour autant que l'architecture du système ait pu être modélisée. Pourtant, les résultats sont insatisfaisants :

1. les architectures de systèmes temps réel réalistes sont souvent trop complexes pour être modélisées exactement ;
2. la complexité de l'analyse et de la résolution du (volumineux) système de contraintes qui en découle explose rapidement ;
3. le WCET ainsi obtenu peut être très imprécis, avec des surestimations trop éloignées de la réalité, atteignant parfois le décuple.

La section 2.3 présente les techniques d'interprétation abstraite, qui apportent des solutions aux deux premiers problèmes, en utilisant une abstraction du système, plus efficace à manipuler. Une attention particulière sera apportée à la preuve de la correction de cette abstraction, c'est-à-dire que l'ensemble des états possibles du système doit toujours être représenté par celle-ci, sans quoi le WCET pourrait être sous-estimé.

Kim, Ha et Min [57] exposent en 1999 les causes de surestimation du WCET, éclairant ainsi les pistes de résolution du troisième problème. Leur étude, portée sur un sous-ensemble des benchmarks de Mälardalen [45], compare les résultats obtenus par analyse statique avec d'autres résultats obtenus par batteries de test, tout en faisant varier les pénalités de *miss* dans le cache. Dans le cas d'une architecture sans pipeline, il est montré que la prise en compte de chemins d'exécution sémantiquement

impossibles – dits infaisables – lors de l'analyse est fréquemment le principal facteur de surestimation avec les effets du cache d'instruction et, dans une moindre mesure, du cache de données.

> *The impact of infeasible paths is strongly dependent on the characteristics of the benchmark programs. For example,* `ludcmp` *has a large number of infeasible paths in its static trace and thus suffers from up to 564% overestimation due to infeasible paths. On the other hand, for* `crc`*, the static trace is almost identical to the dynamic trace and thus the overestimation is only 12%.*
>
> Kim, Ha et Min [57]

En plus d'une modélisation fine des effets des caches d'instructions et de données, l'évaluation d'un WCET précis nécessite un moyen d'atténuer l'impact de ces chemins infaisables, de permettre leur élimination dans le processus d'analyse statique. La section 2.4 détaille ce problème et les approches de résolution proposées dans la littérature. C'est par l'étude de ce sujet que les travaux de cette thèse visent à améliorer les techniques statiques d'estimation du WCET.

2.3 Interprétation abstraite

2.3.1 Introduction

Lors de la recherche de propriétés particulières d'éléments dans un ensemble $\mathcal{D}$, il est parfois utile, voire nécessaire, de s'abstraire d'une partie des informations de $\mathcal{D}$ pour faciliter les calculs. L'abstraction est une technique que nous utilisons fréquemment sans y penser. Chacun sait avec certitude que le résultat de 3792×-4012 est de signe négatif[3] (et même pair). Il serait inintelligent de calculer et tester naïvement le signe de -15213504, ce qu'une machine est susceptible de faire. Pour des calculs complexes, il est plus simple de ne considérer que le signe de chaque opérande ($+$, $-$ ou 0 pour une opérande nulle) comme sur la Figure 2.7. Il suffit ensuite de déduire

$$(+) \times (-) = (-)$$

3. Pour un calcul dans $\mathbb{Z}$.

×	**0**	+	−
0	0	0	0
+	0	+	−
−	0	−	+

Figure 2.7 – Signe d'un produit entier

On peut trouver nombre d'exemples de raisonnement similaires : le test de parité, la "preuve par 9", même le simple fait de raisonner sur des bits est une abstraction de l'état électronique d'un système informatique.

La théorie de l'interprétation abstraite, présentée et appliquée à l'analyse statique par Cousot et Cousot [30] en 1977, formalise ce type de raisonnement. Cette théorie est utile pour représenter des propriétés d'un programme et aider à son analyse. Une excellente introduction pédagogue en est faite par l'ouvrage sur l'analyse de programme de Nielson et al. [79], ainsi qu'une publication par ses fondateurs en 2004 [31].

Une abstraction n'a d'intérêt que si les résultats obtenus dans le domaine abstrait garantissent des propriétés dans le domaine concret. L'abstraction de l'exemple présenté plus haut est utile parce qu'elle permet de conclure avec certitude que le résultat de l'opération correspondante dans $\mathbb{Z}$ est inférieur, supérieur ou égal à zéro. Pour permettre cette garantie, la théorie de l'interprétation abstraite établit le concept d'adjonction.

2.3.2 Correspondance de Galois

Définition 2.4. Une *correspondance de Galois*, ou *adjonction*, est un quadruplet $(\mathcal{D}, \alpha, \gamma, \mathcal{D}^\#)$ constitué de :

- un domaine concret $(\mathcal{D}, \sqsubseteq)$, muni d'un ordre partiel
- un domaine abstrait $(\mathcal{D}^\#, \sqsubseteq^\#)$, muni d'un ordre partiel
- une fonction croissante d'abstraction $\alpha : \mathcal{D} \to \mathcal{D}^\#$
- une fonction croissante de concrétisation $\gamma : \mathcal{D}^\# \to \mathcal{D}$

qui respecte une de ces deux propriétés équivalentes, pour tout $x \in \mathcal{D}$ et $a \in \mathcal{D}^\#$:

(i) $x \sqsubseteq \gamma(\alpha(x))$ et $\alpha(\gamma(a)) \sqsubseteq^\# a$

(ii) $x \sqsubseteq \gamma(a) \iff \alpha(x) \sqsubseteq^\# a$

Intuitivement, la propriété (i) signifie que l'on peut perdre en précision en faisant des aller-retours entre les deux domaines, mais que le procédé est toujours correct (on ne perd pas en terme d'éléments considérés). Les domaines concrets et abstraits sont souvent des treillis, ou au moins définissent un plus petit élément, noté $\bot$, et un plus grand, noté $\top$.

On cherchera aussi souvent, dans le cadre de l'analyse statique, à définir un opérateur d'union $\sqcup^\# : \mathcal{D}^\# \times \mathcal{D}^\# \to \mathcal{D}^\#$ pour joindre deux états abstraits, notamment des états venant de deux traces différentes du programme.

Définition 2.5. Soit $(V, \sqsubseteq)$ un ensemble partiellement ordonné. L'opérateur $\sqcup : V \times V \to V$ est un opérateur d'*union* si et seulement si il vérifie la propriété suivante :

$$\forall v, v' \in V,\ (v \sqsubseteq v \sqcup v') \wedge (v' \sqsubseteq v \sqcup v') \tag{2.3.1}$$

Cette propriété garantit que chacun des opérandes est contenu dans son union. Celle-ci est

- *commutative* si elle respecte $\forall v, v' \in V,\ v \sqcup v' = v' \sqcup v$;
- *associative* si elle respecte $\forall v, v', v'' \in V,\ v \sqcup (v' \sqcup v'') = (v \sqcup v') \sqcup v''$.

Si un opérateur d'union est commutatif et associatif, l'union d'un ensemble non vide $\bigsqcup : \mathcal{P}(V) \setminus \{\emptyset\} \to V$ est uniquement définie comme

$$\bigsqcup\{v_1, v_2, \ldots, v_n\} \coloneqq v_1 \sqcup v_2 \sqcup \ldots \sqcup v_n\ \forall (v_1, v_2, \ldots, v_n) \in V^n$$

Idéalement, on souhaiterait également pouvoir vérifier la propriété

$$(v \sqsubseteq w) \wedge (v' \sqsubseteq w) \Longrightarrow v \sqcup v' \sqsubseteq w \tag{2.3.2}$$

Dans ce cas, l'union est dite *minimale*.

Lorsque le domaine abstrait est un treillis, une union minimale, commutative et associative, existe canoniquement. Parfois, on pourra prouver qu'une correspondance de Galois a certaines propriétés plus fortes, qui en font une insertion.

Définition 2.6. Une *insertion de Galois* est une correspondance de Galois $(\mathcal{D}, \alpha, \gamma, \mathcal{D}^\sharp)$ telle que

$$\forall a \in \mathcal{D}^\sharp,\ \alpha(\gamma(a)) = a \tag{2.3.3}$$

Une des conséquences de cette propriété est notamment que γ est nécessairement injective, et donc que le domaine abstrait ne peut pas contenir d'éléments superflus qui ne décrivent aucun élément de l'ensemble concret.

Exemple. En s'inspirant de l'exemple précédent sur les signes numériques, on pourrait établir la correspondance de Galois suivante [4] :

$$(\mathcal{P}(\mathbb{Z}), \alpha, \gamma, \{\bot, 0, +, -, \top\})$$

ordonnée par l'inclusion $\subseteq$ dans l'ensemble concret, et par $\sqsubseteq^\#$ tel que pour tout a dans l'ensemble abstrait, $\bot \sqsubseteq^\# a \sqsubseteq^\# \top$. Les signes représentent maintenant le signe d'un ensemble de valeurs, $\bot$ signifie "pas de signe" (ensemble vide), $\top$ signifie "n'importe quel signe" (l'ensemble est composé de valeurs aux signes hétérogènes). La fonction de concrétisation est définie telle que [5] $\gamma(0) = \{0\}$, $\gamma(+) = \mathbb{Z}^+$, $\gamma(-) = \mathbb{Z}^-$, $\gamma(\bot) = \emptyset$, $\gamma(\top) = \mathbb{Z}$ et la fonction d'abstraction comme suit :

$$\alpha(X) = \begin{cases} \bot & \text{si } X = \emptyset \\ 0 & \text{si } X = \{0\} \\ + & \text{si } X \subseteq \mathbb{Z}^+ \\ - & \text{si } X \subseteq \mathbb{Z}^- \\ \top & \textit{sinon} \end{cases}$$

C'est bien une correspondance de Galois, les démonstrations que α et γ sont croissantes, ainsi que de la propriété (i) sont sans difficulté. C'est même une insertion de Galois, puisque $\alpha(\gamma(a)) = a$ pour tout a du domaine abstrait. □

L'établissement d'une correspondance de Galois permet de garantir la validité des abstractions et des opérations sur l'état abstrait. Pour cela, il suffit d'utiliser le couple de fonctions (α, γ) pour prouver qu'une fonction est une abstraction valide.

4. Souvent, on inclut dans le domaine abstrait deux valeurs supplémentaires, représentant les positifs ou nuls, et les négatifs ou nuls, ce qui permet d'obtenir un meilleur treillis.

5. $\mathbb{Z}^+ \coloneqq]0\,;\ +\infty[$, $\mathbb{Z}^- \coloneqq]-\infty\,;\ 0[$.

Définition 2.7. Soit $(\mathcal{D}, \alpha, \gamma, \mathcal{D}^\sharp)$ une correspondance de Galois, et $f : \mathcal{D} \to \mathcal{D}$ une fonction sur le domaine concret. Une fonction $f^\#$ est une *abstraction valide* de f lorsque

$$\forall x^\# \in \mathcal{D}^\sharp,\ f(\gamma(x^\#)) \sqsubseteq \gamma(f^\#(x^\#))$$

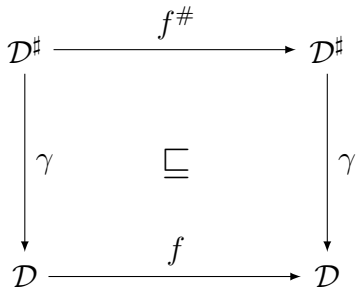


Figure 2.8 – Abstraction d'une fonction

D'un point de vue de l'analyse statique, cela signifie souvent que l'ensemble des états de la machine considérés par l'abstraction maintenue par l'analyse est un surensemble des états réellement possibles.

2.3.3 Construction par fonctions de représentation

Nous allons maintenant voir une approche qui utilise une *fonction de représentation* pour générer une correspondance de Galois, à partir d'un ensemble concret V, d'un domaine abstrait $(\mathcal{D}^\#, \sqsubseteq^\#, \sqcup^\#)$ muni d'une relation d'ordre partiel et d'un opérateur d'union abstraite commutatif et associatif. Cette approche est présentée par Nielson et al. dans leur ouvrage d'analyse statique [79].

Théorème 2.2. *Une fonction de représentation est une application*

$$\beta : V \to \mathcal{D}^\#$$

qui fait correspondre à une valeur de V sa meilleure représentation dans $\mathcal{D}^\sharp$. Cette fonction de représentation génère la correspondance de Galois

$$(\mathcal{P}(V), \alpha, \gamma, \mathcal{D}^\sharp)$$

ayant pour domaine concret $\mathcal{P}(V)$ (naturellement ordonné par $\subseteq$) et où les fonctions d'abstraction et de concrétisation sont définies comme suit :

$$\begin{aligned} \alpha : \mathcal{P}(V) &\longrightarrow \mathcal{D}^\sharp & \gamma : \mathcal{D}^\sharp &\longrightarrow \mathcal{P}(V) \\ V' &\longmapsto \bigsqcup{}^\# \left\{ \beta(v) \mid v \in V' \right\} & a &\longmapsto \left\{ v \in V \mid \beta(v) \sqsubseteq^\# a \right\} \end{aligned}$$

Démonstration. La croissance de α découle de la relation 2.3.1, et la croissance de γ de l'associativité de $\sqsubseteq^\#$. La propriété (ii) de la Définition 2.4 peut être vérifiée grâce à l'Équation 2.3.1 de la Définition 2.5, faisant la preuve que $(\mathcal{P}(V), \alpha, \gamma, \mathcal{D}^\sharp)$ est une correspondance de Galois :

$$\begin{aligned} \alpha(V') \sqsubseteq^\# a &\iff \bigsqcup{}^\# \left\{ \beta(v) \mid v \in V' \right\} \sqsubseteq^\# a \\ &\iff \forall v \in V',\ \beta(v) \sqsubseteq^\# a \\ &\iff V' \sqsubseteq^\# \gamma(a) \end{aligned}$$

□

La Figure 2.9 illustre la construction, et met en évidence que $\alpha(\{v\}) = \beta(v)$ pour tout $v \in V$.

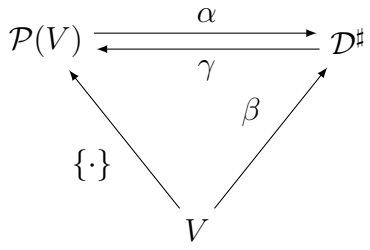


Figure 2.9 – Construction d'adjonctions à partir de fonctions de représentation [79]

2.3.4 Choix du domaine abstrait

L'analyse statique par interprétation abstraite nécessite le choix d'un domaine abstrait approprié. De nombreux domaines sont possibles pour représenter les espaces de valeurs entières des variables d'un programme, ou pour les adresses des accès en mémoire. Rival [89] détaille en 2011 plusieurs domaines numériques pour l'analyse statique par interprétation abstraite. L'implémentation d'une variété de tels domaines

peut être mise à disposition dans des bibliothèques telles Apron [55]. Nous passons en revue quelques espaces numériques connus – des treillis – pour représenter des valeurs entières.

2.3.4.1 Intervalles

Le fonctionnement d'un domaine par intervalles [30] est simple : l'espace de valeurs de chaque variable est représenté par un couple $(m, M) \in \mathcal{D}_I^\# \coloneqq (\mathbb{Z} \cup \{-\infty, +\infty\})^2$, constitué des valeurs minimale et maximale que la variable peut prendre. Les fonctions d'abstraction et de concrétisation vers le domaine concret $\mathcal{P}(\mathbb{Z})$ sont immédiates : $\alpha(A) \coloneqq [\min(A), max(A)]$, et $\gamma([m, M]) \coloneqq [m, M]$. Pour abstraire une fonction $f : \mathbb{Z} \to \mathbb{Z}$ dans ce domaine, on construit la fonction $f^\#$ telle que

$$\forall [m, M] \in \mathcal{D}_I^\#,\ f^\#([m, M]) \coloneqq [\min(\{f(x) \mid m < x < M\}), \max(\{f(x) \mid m < x < M\})]$$

Par exemple $[m, M] + [m', M'] = [m, M] \sqcup^\# [m', M'] = [\min(m, m'), \max(m, M')]$. Le domaine des intervalles est convexe, concis et simple à manipuler. Il fait toutefois de grandes approximations, et manque d'efficacité pour représenter, par exemple, un couple d'entiers.

2.3.4.2 k-sets

Les k-sets sont des ensembles de valeurs de cardinalité maximale k. Ils sont classiquement utilisés dans divers outils d'analyse statique (Jakstab [58] par exemple) pour leur efficacité à représenter des petits ensembles valeurs éparses (ce que les intervalles ne font pas efficacement). Le domaine abstrait est défini comme $\mathcal{D}_k^\# \coloneqq \{A \in \mathcal{P}(\mathbb{Z}) \mid |A| \leq k\} \cup \top$. Lorsqu'un ensemble contient plus de k valeurs, il est représenté par $\top$:

$$\forall A \in \mathcal{P}(\mathbb{Z}),\ \alpha(A) \coloneqq \begin{cases} A & \text{si } |A| \leq k \\ \top & \text{sinon} \end{cases}$$

$$\forall x \in \mathcal{D}_k^\#,\ \gamma(X) \coloneqq \begin{cases} \top & \text{si } X = \top \\ X & \text{sinon} \end{cases}$$

$$\forall x, x' \in \mathcal{D}_k^\#,\ x \sqcup^\# x' \coloneqq \begin{cases} x \cup x' & \text{si } x \neq \top \wedge x' \neq \top \wedge |x \cup x'| \leq k \\ \top & \text{sinon} \end{cases}$$

Les k-sets sont cependant vite limités par leur taille fixe, et leur agrandissement (l'augmentation de k) s'accompagne d'une rapide augmentation de la complexité.

2.3.4.3 CLP

Le domaine des *Circular-Linear Progressions* (CLP) est une abstraction visant à représenter précisément un comportement fréquent dans l'évolution des valeurs de variables de programmes.

```
long t[N];
for(i = 0; i < N; i++)
  t[i] = 0;
```

Figure 2.10 – Accès à un tableau dans une boucle

Un schéma très courant dans les programmes est celui de l'accès à un tableau dans une boucle, comme dans l'exemple de la Figure 2.10. L'adresse de cet accès est généralement compris dans un intervalle entier $[t, t + \delta.(N - 1)]$ où t est l'adresse du premier élément accédé du tableau, δ la taille d'un élément, et N le nombre d'éléments du (sous-)tableau accédé (égal au nombre d'itérations). Cependant, il est possible de donner une description plus précise de l'ensemble des accès que cet intervalle. En effet, si les accès se font par pas de δ comme sur cet exemple ($\delta = 4$ sur la Figure 2.10), leur domaine concret serait donc $\{t, t + \delta, t + 2\delta, \ldots, t + (N - 1)\delta\}$.

Les CLP permettent de représenter ce type d'ensemble de valeurs, qui ont tous le même reste après division par δ. Un CLP est composé d'un triplet (l, δ, m), correspondant (par concrétisation) à l'ensemble $\{l + \delta i \mid 0 \leq i \leq m\} \subseteq \mathbb{Z}$. L'abstraction de fonctions arithmétiques ou bit à bit sur les CLP est assez délicate, mais est largement traitée et illustrée dans plusieurs publications, dont récemment par Källberg [56].

De multiples travaux sur le calcul du WCET utilisent les CLP pour l'interprétation abstraite du programme. Sen et Srikant [96, 95] détaillent l'abstraction de quelques opérations arithmétiques et s'en servent pour faire une analyse d'adresses sur le binaire, avant d'estimer un WCET par ILP. Le domaine des CLP est également utilisé pour l'analyse de code machine dans OTAWA ; il est jugé efficace pour la représentation de variables – notamment d'adresses – par Cassé, Birée et Sainrat [24] qui le démontrent par des expérimentations sur les benchmarks de Mälardalen. Les CLP sont également utilisés dans SWEET, leur implémentation est détaillée par Källberg [56] en 2014.

2.3.4.4 Polyèdres

Cousot et Halbwachs [32] utilisent le modèle d'interprétation abstraite pour détecter des relations linéaires entre variables d'un programme. Le domaine abstrait choisi pour représenter l'espace de valeurs des variables du programme est celui des polyèdres, dont les propriétés sont extensivement utilisées tout au long de l'analyse. Les techniques de programmation linéaire sont utilisées pour déterminer les sommets des polyèdres obtenus par conjonction de contraintes issues du programme.

Un polyèdre convexe de $\mathbb{R}^n$ peut être caractérisé par un système de contraintes linéaires (parfois sous forme matricielle) ou par un système générateur, composé d'une base, d'un ensemble de rayons, et d'un ensemble de sommets. Cette représentation est plus puissante que celle des intervalles : on peut à tout moment obtenir des intervalles de valeurs pour une variable à partir de polyèdres, par projection.

Halbwachs développe largement le domaine des polyèdres convexes pour l'analyse de programmes dans sa thèse [49]. Lisper utilise ces résultats en 2003 pour développer une analyse de WCET paramétrique [65]. Les bornes de boucles sont dérivées automatiquement, des techniques améliorées par rapport à l'IPET classique sont utilisées pour permettre la paramétrisation du WCET, c'est-à-dire qu'il devient fonction d'un vecteur de paramètres. Le système de contraintes ILP symboliques est résolu en utilisant une forme d'ILP paramétrique, appelée *Parameter Integer Programming*. Les polyèdres convexes utilisés dans des états abstraits représentant les valeurs des variables du programme permettent de dériver des contraintes de flot paramétriques.

Une des limites toutefois de ce choix d'abstraction est la nécessité de maintenir la convexité à tout moment ; certaines techniques exigent même que les arêtes des polyèdres ne se coupent qu'à certains angles (60°, ...). L'union abstraite ($\sqcup^\#$) de deux polyèdres entraînera parfois des surapproximations (en particulier s'ils sont disjoints) pour obtenir un plus grand polyèdre convexe qui contienne les deux. Les polyèdres ne sont pas non plus adaptés pour abstraire des ensembles tels ceux représentés par les CLP. Enfin, connaître l'ensemble des valeurs *entières* contenues dans un polyèdre sur un espace vectoriel $\mathbb{R}^n$ est un problème non-trivial qui peut accroître la complexité de l'analyse.

2.3.4.5 Conclusion

Les possibilités d'abstraction et leurs variations sont infinies. Nous en avons présenté les plus basiques ou classiques, mais nombre d'autres abstractions ont été utilisées pour l'analyse statique de programmes. Notamment, Bound-T [53], un outil de calcul de WCET travaillant directement sur le binaire, utilise l'arithmétique de Presburger, un sous-ensemble décidable de l'arithmétique entière. Plus récemment, en 2007, Antoine Miné développe le domaine des Matrices de Différence des Bornes, ou *Difference-Bound Matrices* (DBM) [70], et Péron et Halbwachs [82] l'étendent et proposent une application à l'analyse de programme.

2.3.5 Notre philosophie

Définir une abstraction adaptée aux objectifs d'une analyse de programmes est crucial pour optimiser son efficacité et sa complexité, et en définit une partie des caractéristiques. Une abstraction simple et contraignante permet d'accélérer une analyse, mais les résultats risquent de souffrir des approximations engendrées. Une abstraction utilisant des représentations plus puissantes, plus fines des valeurs du programme sera nécessairement plus coûteuse et complexe à manipuler, mais pourra découvrir des propriétés qui n'auraient pu être découvertes sans cela.

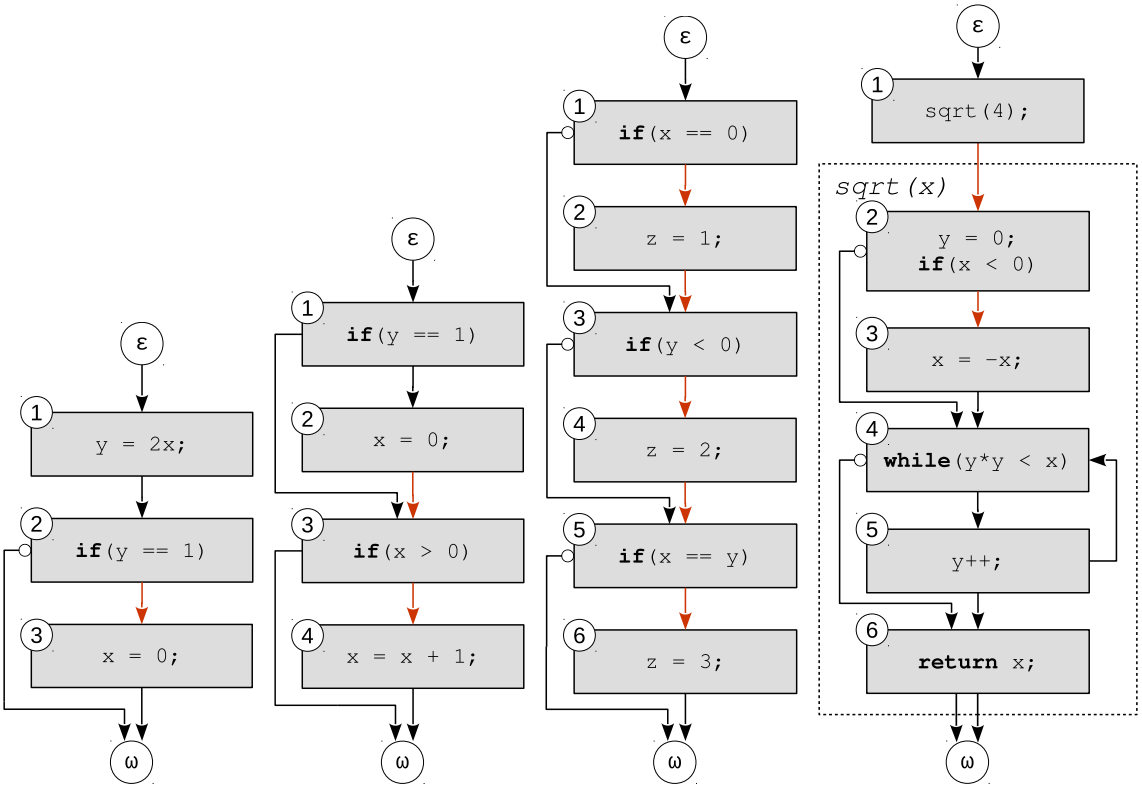
C'est en partant de ce constat que les travaux de cette thèse développeront une solution *hybride*, composée d'un domaine abstrait d'abord très général et puissant – un ensemble de contraintes majoritairement linéaires – mais difficile à manipuler, ensuite enrichi d'un domaine plus structuré, aisé à maintenir et facilitant l'extraction de certaines informations nécessaires à l'analyse de programmes, comme la position d'une donnée adressée dans la pile.

2.4 Chemins infaisables

2.4.1 Problématique

Nous avons vu dans la section 2.2 que l'analyse statique se base sur une représentation du programme sous la forme d'un graphe, le CFG. Sa construction ne prenant pas en compte la sémantique du programme, mais seulement sa structure, les CFG repré-

sentent fréquemment plus de chemins qu'il n'y a réellement de chemins d'exécutions possibles. Ce phénomène peut être observé peu importe le niveau de représentation du programme ; il est illustré sur des programmes C sur la Figure 2.11.



(a) Code mort **(b)** Conflit entre bloc et condition **(c)** Conflit entre conditions **(d)** Chemin contextuellement infaisable

Figure 2.11 – Exemples de chemins infaisables

Sur chacune des figures, l'ensemble des arcs colorés représentent un chemin présent dans le CFG qui ne peut exécuté, quel que soit le contexte dans lequel le programme principal est exécuté. Ces chemins sont dits infaisables.

Définition 2.8. Un *chemin infaisable* est un chemin de CFG sémantiquement impossible, autrement dit, une séquence d'arcs contigus qui n'est incluse dans aucun chemin d'exécution du programme associé.

Nous identifions quatre types de chemins infaisables.

Le cas le plus trivial est celui du *code mort* (Figure 2.11a) : le chemin infaisable est composé d'un seul arc, qui ne peut être exécuté dans aucun scénario, quelles que soient les valeurs d'entrée du programme. Ce type de chemin infaisable témoigne gé-

néralement d'une programmation médiocre et est souvent supprimé à la compilation par des optimisations, mais subsiste parfois lorsqu'il est caché par un grand volume de code (entre le bloc ① et ② pour cet exemple).

Plus souvent, les chemins infaisables proviennent d'un conflit entre plusieurs instructions, mutuellement exclusives, du même contexte. Il peut s'agir d'un conflit entre affectations et conditions (Figure 2.11b), ou entre conditions (Figure 2.11c). Parfois ce sont simplement deux conditions en conflit (par exemple, `x > 0` et `x < 0`) ; parfois, c'est une série de conditions qui forment un ensemble d'arcs ne pouvant être tous empruntés, mais qui ne sont pas deux-à-deux exclusifs (comme sur la Figure 2.11c).

Enfin, il existe un dernier type de chemins infaisables, illustré de la Figure 2.11d. Certains arcs de sous-programmes peuvent ne jamais exécutés dans certains contextes, pour certaines valeurs d'entrée. C'est un chemin contextuellement infaisable, parfois appelé *code dynamiquement mort*. En effet, pour certains cas d'exécutions, comme sur cet exemple dans le cas d'un appel à `sqrt` avec une valeur positive, une partie du code ne s'exécutera jamais.

Par ailleurs, la grande taille d'un programme est un facteur important d'introduction de chemins infaisables, du fait du potentiel d'apparition d'arcs en conflit "éloignés", c'est-à-dire séparés par un grand volume de code. Dans ce cas, il est difficile pour un programmeur de restructurer le programme de façon à éviter qu'il contienne un tel chemin infaisable. Ce constat doit motiver les analyses de chemins infaisables à atteindre une complexité suffisamment bonne (faible) pour traiter des programmes de grande taille, où l'on peut avoir fort à gagner : plus un programme est gros, plus il a de chances d'être sujet à ce facteur d'introduction de chemins infaisables.

Une analyse de WCET par IPET incluant des chemins infaisables comme chemins d'exécution possibles s'expose à des surapproximations qui peuvent augmenter significativement le pessimisme du WCET estimé. En effet, l'évaluation du WCET devra prendre en compte des chemins potentiellement plus coûteux que le WCEP, chemin associé au WCET. Le WCEP qui sera considéré dans l'analyse, par exemple celui qui sera implicitement désigné par la maximisation de la fonction objectif du système ILP, ne correspondra alors pas au WCEP réel. L'inclusion de chemins infaisables dans le calcul du WCET ne met pas en danger la garantie de surestimation du WCET, mais elle *peut* dégrader la précision de l'analyse.

Il faut donc, en premier lieu, détecter ces chemins infaisables, et en second lieu, les

exclure de l'analyse, soit en reconstruisant le CFG (difficile et parfois coûteux), soit en les signalant à l'analyse par le biais d'annotations. Il faut alors convenir d'un moyen pour exprimer, et transférer les propriétés de flot de contrôle (*flow facts* en anglais) que les chemins infaisables représentent, de l'analyse qui les identifie vers l'analyse qui les exploite.

Bien que l'amélioration des analyses de pire-temps d'exécution soient l'objectif principal de cette thèse, le problème des chemins infaisables n'est toutefois pas limité au domaine du WCET, et leur détection a d'autres applications. Outre le calcul du WCET par IPET, ce problème affecte la précision de nombreuses autres analyses statiques, telles que les analyses de cache, ou d'adresses. Nous verrons aussi en 2.4.2.2 que c'est un problème majeur pour la génération de cas de tests. Le processus de compilation bénéficie également de la suppression de chemins infaisables, qui est chose courante dans les phases d'optimisation des compilateurs modernes (bien qu'elle soit souvent indirecte).

2.4.2 État de l'art

2.4.2.1 Pour le WCET

Altenbernd [7] souligne en 1996 l'importance de la détection de chemins infaisables pour l'amélioration de la précision de l'analyse du pire-temps d'exécution. Il y est mentionné que c'est en 1987 que le premier essai sur ce problème est présenté, par Benkoski et al. [14]. Il est ensuite prouvé NP-complet pour des systèmes d'états finis en 1991 par McGeer et Brayton [68]. Alternberd extrait le CFG à la compilation, puis applique une technique d'énumération de chemins combinée avec l'exécution symbolique des branches, qui peuvent être élaguées (*pruning*) afin d'améliorer le WCET. Les boucles y sont déroulées, ce qui implique que les bornes de boucles doivent être connues au moment de l'analyse. Tous les chemins sont énumérés, la complexité de l'analyse augmente donc exponentiellement avec le nombre de conditions ($O(2^{\#if})$), et est multipliée dans chaque boucle par le nombre d'itérations. Cette technique fonctionne sur de petits benchmarks mais est très vite limitée par sa complexité.

Toujours en 1996, Kontouris [60] fait le point sur l'état de l'art et, bien que plusieurs auteurs aient à l'époque noté que le pessimisme était un facteur majeur de surestimation, le problème est, d'après Kontouris, souvent remédié par des annotations de l'utilisateur sur le code source. Kontouris souligne que cette approche n'est pas

acceptable pour des systèmes temps-réel critiques, et présente une technique d'élimination automatique de chemins infaisables pour des programmes écrits dans un langage synchrone particulier, SIGNAL. Le champ d'application est restreint à ce langage, et bien que du code C puisse être généré à partir de tout programme SIGNAL, l'inverse n'est pas vrai. De plus, comme souvent avec les analyses entièrement basées sur le code source, la garantie de correction du transfert de propriétés vers le code binaire est une question en suspens.

En 2012, Ding et. al [36] donnent une vue d'ensemble des techniques et outils de détection de chemins infaisables. Ils nomment les différentes approches possibles au problème (analyse de flot de données, propagation de contraintes...) ainsi que les outils qui les implémentent. Les auteurs désignent les travaux de Gustafsson, Ermedahl et Lisper [46] comme les plus matures en termes de recherche de chemins infaisables par analyse de flot de données. Cette publication de 2006, présente une technique de détection de chemins infaisables avec trois algorithmes séparés, implémentée dans SWEET. Cet outil d'analyse suédois supporte alors les processeurs NECV850E et ARM9, et peut détecter automatiquement des bornes de boucles [47]. Il est intégré à un compilateur et opère sur la représentation intermédiaire du code dans ce compilateur.

Les auteurs catégorisent les chemins infaisables en deux types, (i) ceux issus de "dépendances sémantiques", c'est-à-dire des conflits entre conditions pour toute trace d'exécution ; (ii) ceux dus à des limitations sur les valeurs en entrée, qui dépendent donc du contexte d'exécution. Le programme est divisé en plusieurs *scopes*, analysés séparément et reliés par une représentation hiérarchique, le *scope graph*. Les auteurs utilisent l'exécution abstraite, une forme d'exécution symbolique basée sur l'interprétation abstraite, pour prendre connaissance de l'évolution des valeurs des variables du programme. L'implémentation de cette analyse permet à l'utilisateur de sélectionner un ensemble de valeurs d'entrée pour l'exécution abstraite d'un programme. Les états abstraits étant dupliqués à chaque condition dont le résultat ne peut pas être déterminé statiquement, les auteurs ont recours à des "points de fusion", où les différentes valeurs abstraites sont fusionnées en un seul état, représentant conservativement un chemin unique, mais perdant en précision. Le domaine abstrait utilisé est celui des intervalles, décrit dans la sous-section 2.3.4.1. Les expérimentations (sur les benchmarks de Mälardalen) montrent une faible complexité, mais ne font pas état d'une amélioration du WCET, seulement de la quantité de contraintes de flot, rendant l'efficacité de l'analyse difficile à juger.

Sewell, Kam et Heiser [97] publient également en 2016 une analyse de détection automatique de chemins infaisables et de bornes de boucles, pour l'analyse du WCET. L'outil opère sur seL4, un noyau de système d'exploitation vérifié par l'assistant de preuve Isabelle/HOL, en utilisant l'outil Chronos pour l'analyse de WCET (par IPET). Après compilation d'un programme C, les binaires sont décompilés et validés par rapport au programme original. Ce modèle de traduction-validation (TV) est validé par un solveur SMT. L'analyse de chemins infaisables est une extension d'une analyse de code mort (analyse d'atteignabilité). Les bornes de boucles sont obtenues depuis le code source, et transposées sur le binaire. La recherche de chemins infaisables itère de nombreuses fois en se concentrant sur le(s) chemin(s) le plus long, seul(s) chemin(s) dont l'élimination peut augmenter la précision de l'estimation du WCET. Après une dizaine d'itérations sur le programme étudié, en environ 19 heures pour 9000 lignes de code et 14000 instructions, le WCET se stabilise avec un gain important (environ -50%). Toutefois, ce programme est le seul cas étudié par l'analyse de chemins infaisables. La complexité de l'analyse semble trop explosive pour analyser de plus grandes boucles comme celles qui peuvent être trouvées dans certains benchmarks de Mälardalen, l'analyse exploitant, d'après les auteurs, la taille généralement modeste des fonctions dans seL4.

Trickle [18] est un autre outil de détection de chemins infaisables appliqué sur le noyau seL4, en plus des benchmarks de Mälardalen. Trickle analyse le code binaire de programmes qu'il divise en plusieurs *scopes* individuellement dénués de boucles. Une particularité intéressante de cet outil est son exploitation du solveur SMT Yices, pour obtenir des sous-ensembles minimaux de contraintes insatisfiables ("*unsat cores*") lorsqu'un conflit est détecté. L'algorithme de recherche d'*unsat cores* CAMUS est directement implémenté afin d'obtenir de légers gains en performance par rapport à Yices dans l'obtention de ces "*unsat cores*". Ces sous-ensembles minimaux permettent de générer des chemins infaisables plus courts (composés d'un ensemble d'arcs en conflit minimal), et donc plus expressifs. Nous développerons l'usage de cette technique dans la section 5.2.4.3.

Zwirchmayr et. al [108] présentent une analyse de conditions de branchements qui aide également à se concentrer sur des traces proches du WCEP. Les auteurs cherchent des branchements "déséquilibrées", issus de conditions dont le résultat affecte considérablement le WCET, en raison d'une différence importante entre les chemins suivant l'arc *pris* et ceux suivant l'arc *non pris*. Il est utile d'identifier ces conditions puisque

ce sont les points de code les plus pertinents à étudier pour améliorer la précision de l'analyse de WCET. Ces informations de poids sur les conditions peuvent bénéficier à une multitude d'analyses, dont toute analyse de chemins infaisables. L'approche n'est toutefois pas encore complètement automatisée et dépend encore d'annotations utilisateur. Les résultats ne nécessitent cependant pas d'être garantis pour être utilisé par l'analyse de systèmes critiques, l'intérêt de la technique étant de fournir une heuristique pour améliorer l'efficacité d'autres analyses, en leur indiquant où elles peuvent gagner en précision.

Chen, Mitra, Roychoudhury et Vivy [26, 99] implémentent une technique de détection et d'exploitation de chemins infaisables pour l'évaluation du WCET. Seuls des CFG acycliques (des DAG donc) sont analysés, ce qui sectionne le CFG en plusieurs parties. Le corps de chaque boucle est analysé séparément, pour une seule itération. Deux types de conflits peuvent être détectés : entre deux arcs du DAG ou entre un arc et une instruction d'affectation de variable. La complexité de l'analyse est raisonnablement faible, décrite par les auteurs comme en $O((|V| + |E|) \times |E|)$, ce qui en fait une bonne analyse pour des chemins infaisables simples. Cependant, la classe de chemins infaisables détectables est très limitée, en particulier pour les conflits dont deux arcs seraient suffisamment éloignés l'un de l'autre pour que du code non-acyclique puisse être exécuté entre eux.

Holsti [51] présente une approche différente du problème, où le temps d'exécution est modélisé comme une variable du programme. Son outil Bound-T étant basé sur IPET, il n'y présente pas de résultats expérimentaux. Dans ce papier, Holsti évoque notamment les limites de l'analyse de Cousot et Halbwachs dans le cadre de l'élimination de chemins infaisables du calcul du WCET. Il souligne la nécessité, malgré le problème de complexité engendré (et constaté sur de grands sous-programmes par Bound-T), de travailler avec des disjonctions pour représenter l'état d'un programme afin de permettre la détection de différents types de chemins infaisables. Le *program slicing* [93] et le *join* par intersection convexe en des points arbitraires du programme y sont proposés afin de réduire la complexité de l'analyse.

D'autres techniques existent. Stein et Martin [98] ont publié une tentative d'analyse du code binaire pour la recherche et l'exclusion de chemins infaisables, en utilisant l'arithmétique de Presburger. Les résultats sont limités mais comportent des éléments intéressants, comme la substitution d'opérandes pour gérer les *overflow*/*underflow* sur n bits. D'autres approches cherchent à valider tous les chemins construits, et à ne

construire que les chemins faisables [4].

2.4.2.2 Pour l'amélioration de la génération de cas de tests

Parfois, des techniques de détection de chemins infaisables sont développées pour améliorer l'efficacité d'outils de génération de cas de tests. En éliminant la plupart des chemins infaisables dans les cas de tests générés, autant de calculs inutiles sont évités.

Heldey et Hennel [50] affirment en 1985 qu'il est fréquent qu'une quantité considérable ressources soient gâchées pendant les phases de test pour l'exécution de chemins infaisables. Les auteurs identifient plusieurs types de chemins infaisables, et soulignent l'explosivité du ratio de chemins infaisables en présence de séquence de nombreux sauts conditionnels en séquence. D'après eux, les facteurs principaux de chemins infaisables sont (i) une surestimation du nombre d'itérations des boucles, (ii) des défauts provenant de l'inadéquation du langage utilisé, et (iii) un piètre style de programmation. Les auteurs suggèrent de réécrire les jeux de tests dans un langage fonctionnel, qui permettrait d'écrire des programmes sans chemins infaisables, avec des structures plus simples et moins de chemins d'exécution.

Plus récemment, Ngo et Tann [78] s'attaquent également au problème en proposant des heuristiques pour détecter des chemins infaisables pendant la génération dynamique de jeux de tests. Ces techniques sont implémentées dans jTGEN, un générateur de jeux de tests pour Java basé sur le framework d'optimisation SOOT, qui analyse du bytecode Java. Ces mêmes auteurs présentent dans [77] une classification de quatre schémas (patterns) de chemins infaisables : Identical/complement-decision pattern, Mutually-exclusive-decision pattern, Check-then-do pattern, et Looping-by-flag pattern. Ils présentent ensuite des algorithmes pour détecter ces schémas, et implémentent ces techniques avec SOOT.

2.4.3 Expression et exploitation

Le transport de propriétés de flot entre les modules (*back-end* et *front-end*) de l'analyse statique pour le WCET nécessite l'utilisation d'un *langage d'annotation*. Kirner et al. [59] décrivent en 2007 les caractéristiques d'un langage d'annotation et dressent un comparatif des langages utilisés par plusieurs outils de calcul de WCET. Ceux-ci (aiT, Bound-T, SWEET...) utilisent déjà une multitude de langages d'annotations internes. Si l'usage du modèle d'annotation le plus adapté à chaque outil est avantageux par

certains côtés, cette approche fait perdre toute portabilité.

Ainsi, dans une tentative d'unifier ces langages d'annotation, Bonenfant et al. [21] développent en 2012 FFX, un langage XML. Celui-ci peut notamment exprimer des bornes de boucles, des chemins infaisables – plus généralement des limites de nombre d'exécutions sur une séquence d'arcs – et le tout peut être restreint dans des contextes (architecture, points d'appel de fonctions, etc.). L'objectif de FFX est de supporter la majorité des cas d'utilisation sans ambiguïté, et d'améliorer la portabilité de ces annotations, qui doivent, en l'état, être adaptées pour chaque outil. Chaque analyse peut choisir d'utiliser une partie du langage FFX en sortie, ou n'en considérer qu'une partie en entrée. Les annotations FFX peuvent être produites manuellement par l'utilisateur ou générées automatiquement. Les auteurs implémentent FFX pour faire communiquer les différents modules de l'outil d'analyse statique OTAWA.

```
<context name="arm">
  <function name="f">
    <loop maxcount="10"/>
  </function>
</context>
```

Figure 2.12 – Exemple d'élément FFX décrivant une borne de boucle

Les propriétés de flot de contrôle que les travaux de cette thèse s'appliqueront à identifier seront exprimées dans le format FFX.

Une fois les chemins infaisables détectés et exprimés, se pose le problème de leur exploitation pour l'amélioration du WCET. S'il est souvent possible de modifier le CFG pour en enlever les chemins infaisables, cela demande d'isoler au préalable le chemin en question dans le graphe, ce qui peut largement augmenter sa taille et faire exploser le nombre de chemins à parcourir lors du calcul de WCET. Pour les calculs de WCET par IPET, à base de résolution d'un système ILP (cf. 2.2.3), il est plus efficace d'injecter les informations sur les chemins infaisables sous la forme de contraintes supplémentaires dans le système ILP. Ainsi, Raymond [86] présente en 2014 une approche générale pour exprimer des chemins infaisables comme contraintes ILP, que nous utiliserons nous-mêmes dans nos expérimentations.

Il existe des techniques plus avancées pour exploiter des chemins infaisables. Celles développées par Mussot et al. [73, 74, 76, 75] pour l'amélioration du WCET statique transforment des propriétés de flot données en entrée au format FFX en automates. Cette méthode utilise un formalisme basé sur des automates et des contraintes linéaires pour représenter ces propriétés. Ce formalisme permet plus d'expressivité que les systèmes de contraintes classiques pour exprimer des restrictions sur les chemins pouvant être pris en compte dans l'analyse du WCET, et permet *in fine* l'amélioration de la

précision (par rapport à une génération classique de contraintes ILP).

2.5 Conclusion générale

Nous avons dans ce chapitre présenté le problème du calcul d'une borne supérieure sûre du WCET, et les approches pour sa résolution par analyse statique, le tout dans un contexte de systèmes temps-réel critiques. Après avoir motivé notre choix d'utiliser exclusivement le code binaire pour l'analyse de flot de données de systèmes critiques, nous avons identifié la prise en compte de chemins infaisables comme l'un des facteurs majeurs d'imprécision dans le calcul du WCET.

Nous avons posé les bases de l'interprétation abstraite, théorie sur laquelle se base les développements de cette thèse présentés dans le chapitre éponyme. La section 2.3.4 donne un aperçu des abstractions courantes, dont nous devrons nous inspirer pour définir une abstraction adaptée, point crucial de l'efficacité de notre analyse de flot de données.

Nous avons passé en revue une variété de techniques et outils existants pour l'identification de chemins infaisables en section 2.4. Nous avons identifié les difficultés du développement de telles analyses de programme, en particulier en termes de complexité, de précision et de généralité. En situant la position des analyses de recherche de chemins infaisables dans la chaîne du calcul du WCET, nous avons également motivé l'intérêt d'une analyse portable, dont les résultats soient exprimés de forme à pouvoir être exploités par d'autres modules faisant partie de l'évaluation d'une borne supérieure au WCET.

3

Interprétation abstraite

Sommaire

Ce chapitre présente des méthodes pour représenter l'effet de l'exécution d'un programme sur une machine. Nous présentons les abstractions du programme, de la machine et de l'effet de l'exécution du programme, et garantissons leur correction en nous appuyant sur les résultats de la théorie de l'interprétation abstraite.

3.1 Introduction : représentation du programme

L'usage d'un environnement de développement adéquat est indispensable au développement efficace d'une analyse de programmes, en particulier lorsqu'il s'agit d'analyser directement du code machine. Le framework C++ OTAWA (Open Tool for Adaptive WCET Analyses [10]) fournit un cadre pour l'analyse statique de programmes binaires pour l'évaluation du WCET. Il est composé d'un ensemble de modules interdépendants qui communiquent à l'aide d'annotations internes ou externes (via le langage FFX). OTAWA permet la génération automatique de CFG, diverses transformations de CFG (régularisation de boucles, slicing...), l'obtention automatique d'informations de boucles (identification des têtes de boucles, des arcs de sortie...), ainsi que de nombreuses autres propriétés extraites du programme (constantes chargées dans la pile, propriétés de dominance...).

Les techniques d'analyse statique présentées dans cette thèse ont été implantées sous la forme d'un plugin OTAWA de recherche de chemins infaisables, nommé *PathFinder*.

La première étape du développement de l'analyse statique d'un programme est de définir une représentation de ce programme. Nous avons présenté, dans la section 2.2.1, les graphes de flot de contrôle (CFG), un moyen répandu de représenter la structure d'un programme sous la forme d'un graphe. Chaque sommet du CFG d'un programme binaire est, par définition, une séquence maximale d'instructions machine (Définition 2.1). Ces instructions appartiennent à un *jeu d'instructions* dont il existe une multitude (ARM, x86, PowerPC, SPARC...) qui présentent de nombreuses variations : design (RISC/CISC), taille des registres (32 bits, 64 bits...), *endianness* (gros-boutiste ou petit-boutiste) ; et plus généralement par la sémantique des instructions encodées.

3.1.1 Instructions sémantiques

Non seulement l'écriture d'une analyse de programmes pour un jeu d'instructions spécifique est long, fastidieux, et demande une expertise considérable, mais cela restreint aussi fortement le champ d'application des techniques développées et leur intérêt académique. Il est donc utile pour l'analyse de programmes binaires, en plus d'utiliser la représentation par CFG, de passer par une représentation intermédiaire pour analyser les instructions assembleur à l'intérieur de chaque bloc de base. Nous utilisons

pour ce faire un langage unique, une abstraction des instructions machine construite à la manière des jeux d'instructions RISC, vers laquelle sont traduits [1] des ensembles d'instructions issus de diverses architectures [2].

Remarque. Bien que la grande majorité des techniques que nous emploierons pourraient être facilement généralisées pour k bits, le framework utilisé est conçu pour opérer sur des architectures 32 bits, et les travaux de cette thèse se concentreront sur cette classe de systèmes.

Instruction	Sémantique	Notes
`set` d, a	$d \leftarrow a$	
`seti` d, k	$d \leftarrow k$	
`add` d, a, b	$d \leftarrow a + b$	
`sub` d, a, b	$d \leftarrow a - b$	
`mul` d, a, b	$d \leftarrow a \times b$	
`div` d, a, b	$d \leftarrow a \,/\, b$	
`divu` d, a, b	$d \leftarrow a \,/_+ b$	$/_+$: division entière non signée
`mod` d, a, b	$d \leftarrow a \bmod b$	
`modu` d, a, b	$d \leftarrow a \bmod_+ b$	$\bmod_+$: modulo non signé
`shl` d, a, b	$d \leftarrow a \ll b$	$\ll$: décalage logique à gauche
`shr` d, a, b	$d \leftarrow a \gg b$	$\gg$: décalage logique à droite
`asr` d, a, b	$d \leftarrow a \gg_+ b$	$\gg_+$: décalage arithmétique à droite
`neg` d, a	$d \leftarrow -a$	
`not` d, a	$d \leftarrow \neg a$	$\neg$: négation logique, $\neg a = -a - 1$
`and` d, a, b	$d \leftarrow a \wedge b$	$\wedge$: et logique
`or` d, a, b	$d \leftarrow a \vee b$	$\vee$: ou logique
`cmp` d, a, b	$d \leftarrow a \sim_* b$	$\sim_*$: comparaison signée
`cmpu` d, a, b	$d \leftarrow a \sim_+ b$	$\sim_+$: comparaison non signée
`load` d, a, t	$d \leftarrow M_t[a]$	$M_t[a]$: cellule mémoire de type t à
`store` d, a, t	$M_t[a] \leftarrow d$	l'adresse a
`scratch` d	$d \leftarrow \top$	$\top$: valeur inconnue
`assume` c, a	assert$(c(a))$	La relation c est vérifiée par la comparaison enregistrée dans a

$d, a, b \in Var$, $k \in \mathbb{Z}_{32}$, $t \in \{\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}, \mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}, \mathbb{N}_{64}\}$
$c \in \{=, \neq, <, \leq, >, \geq, <_+, \leq_+, >_+, \geq_+\}$

Table 3.1 – Instructions sémantiques d'OTAWA [24]

1. La traduction se fait dans OTAWA à l'aide de l'outil GLISS, qui met à disposition des outils facilitant le décodage de programmes sous leur forme binaire.

2. Actuellement, ARMv5, Tricore et Sparc sont supportés. Le framework peut cependant être étendu à d'autres jeux d'instructions, que l'ensemble des analyses incluses, étant définies indépendamment de l'architecture, supporteront alors automatiquement.

3.1.1.1 Fondamentaux

Les instructions sémantiques définissent un ensemble d'instructions arithmétiques et logiques élémentaires, avec leurs variantes non signées. Toutes les instructions sémantiques ne sont pas forcément pertinentes pour un jeu d'instructions (par exemple, certains processeurs ne font pas directement de division) et une instruction machine peut se traduire en *plusieurs* instructions sémantiques à interpréter en séquence.

En effet, la sémantique d'une instruction machine pouvant être assez complexe, il est utile de chercher à traduire de telles instructions machines en suites d'opérations élémentaires. En ne devant interpréter qu'un ensemble réduit d'instructions sémantiques, les analyses gagnent en simplicité de développement. Toujours dans l'objectif de minimiser le nombre d'instructions à gérer par les analyses, toutes les instructions sémantiques opèrent sur des variables, sauf `seti` qui permet d'introduire des valeurs immédiates.

Définition 3.1. Chaque instruction issue du jeu d'instructions I_a d'une architecture a a pour image par la *fonction de traduction* t_a une séquence d'instructions sémantiques :

$$t_a : I_a \rightarrow {I_{sem}}^*$$

La séquence obtenue, abstraction d'une instruction machine, est le *bloc d'instructions sémantiques* correspondant.

Un bloc de base étant composé d'une séquence d'instructions machine, il peut être transformé par cette fonction de traduction en une séquence de blocs d'instructions sémantiques.

Remarque. Le registre *pc* (*program counter*), qui indexe chaque instruction machine par son adresse [3], est un cas particulier de la sémantique des instructions assembleur. Ce registre est, par convention, systématiquement incrémenté de la taille d'une instruction assembleur après chaque exécution de celle-ci. Le bloc d'instructions sémantiques issu de la traduction de toute instruction machine devrait donc commencer par une instruction sémantique qui incrémente le registre *pc*. En ARM, par exemple, ce serait `add` $r_{15}, r_{15}, 4$. Afin de minimiser la taille de ces blocs et ainsi réduire le nombre d'instructions sémantiques à analyser, ce changement n'est pas systématiquement traduit.

3. En réalité, ce registre contient généralement l'adresse de l'instruction suivante.

À la place, tout bloc d'instructions sémantiques utilisant le registre *pc* commencera par une instruction `seti` définissant la valeur du *pc* (par exemple, `seti` r_{15}, `0x8004`).

3.1.1.2 Combinaison d'instructions sémantiques

`LDR r3, [r11, #-8]`
- `seti` $t_2, -8$
- `add` t_1, r_{11}, t_2
- `load` $r_3, t_1, \mathbb{N}_{32}$

Figure 3.1 – Bloc d'instructions sémantiques

La Figure 3.1 offre un exemple d'instruction ARM traduite en un bloc de plusieurs instructions sémantiques. L'instruction `LDR` r_3`, [`r_{11}`, #-8]` charge dans le registre r_3 le mot (valeur sur 4 octets) à l'adresse $r_{11} - 8$. Cette instruction machine est traduite en trois instructions sémantiques : les deux premières calculent l'adresse [4], la dernière fait la lecture mémoire et écrit le mot chargé dans r_3.

Remarquons l'introduction de deux variables, t_1 et t_2, absentes de l'instruction originale. Ce sont des *variables temporaires*, et elles sont nécessaires pour faire le lien entre les trois instructions du bloc sémantique. Ces variables sont *locales* à un bloc d'instructions sémantiques, c'est-à-dire que les valeurs qu'elles contiennent ne sont jamais réutilisées d'un bloc à l'autre. Les variables temporaires servent exclusivement à permettre la traduction d'instructions machine en plusieurs instructions sémantiques. Nous définissons ainsi le domaine des variables acceptées par les instructions sémantiques :

Définition 3.2. Soit $Reg \coloneqq \{r_0, \ldots, r_{k-1}\}$ l'ensemble des k registres disponibles sur l'architecture considérée. Soit $Tmp \coloneqq \{t_1, \ldots, t_m\}$ l'ensemble des m variables temporaires nécessaires à la traduction d'instructions. L'ensemble des variables acceptées par les instructions sémantiques est :

$$Var \coloneqq Reg \cup Tmp$$

Le maximum m de variables temporaires nécessaires pour traduire toute instruction est donné par le framework, pour chaque architecture (par exemple, $m = 3$ pour ARM).

3.1.1.3 Instruction `scratch`

4. Il est nécessaire ici d'utiliser deux instructions pour le calcul de l'adresse puisque `add` n'accepte que des variables – pas de valeur immédiate.

Parfois, les opérations élémentaires du jeu d'instructions sémantiques ne suffisent pas pour exprimer des instructions machines particulières ou complexes. C'est le cas par exemple de `REV`, une instruction ARM qui inverse l'ordre des bits dans un mot. Afin d'exprimer conservativement les effets de telles instructions, l'instruction sémantique `scratch` est utilisée (Figure 3.2). Une instruction `scratch` d indique simplement qu'une opération a écrit sur le registre d, et donc que sa valeur est désormais inconnue. Bien que l'usage de `scratch` affaiblisse la précision des analyses, cela est fait de manière contrôlée, de telle sorte à invalider les propriétés d'aussi peu de registres que nécessaire, tout en permettant la traduction de l'intégralité d'un jeu d'instructions.

```
REV r1, r0
| scratch r1
```

Figure 3.2 – Traduction d'une instruction complexe

3.1.1.4 Traduction des branchements

Les instructions assembleur de branchement sont particulières dans le sens où elles affectent le flot d'exécution du programme. Les blocs de base étant, par définition, toujours exécutés séquentiellement, seule la dernière instruction assembleur d'un bloc peut être une instruction de branchement.

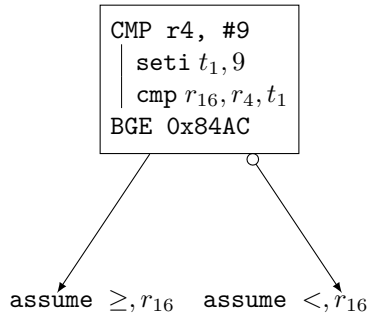


Figure 3.3 – Traduction d'un branchement conditionnel

Le flot de contrôle du programme étant représenté par la structure du CFG, aucune instruction sémantique n'est nécessaire pour traduire un branchement *inconditionnel*. Par exemple, le bloc d'instructions sémantiques traduisant une instruction ARM "`B 0x80C0`" est vide. En revanche, les branchements *conditionnels* donnent lieu à deux arcs sortants, représentant deux chemins d'exécution possibles en sortie du bloc courant. Il faut les distinguer en indiquant à quel cas d'exécution ils correspondent (chemin *pris*, condition vraie ou chemin *non pris*, condition fausse).

L'instruction sémantique `assume` permet d'informer les analyses du résultat d'une comparaison préalablement effectuée. Cette instruction ne peut pas être simplement rajoutée en tête des blocs de base pointés par les arcs sortants (conditionnels), car ceux-ci pourraient également faire partie d'un autre chemin d'exécution (avoir d'autres arcs en entrée) pour lesquels la propriété exprimée par `assume` ne serait pas valide. C'est pourquoi il faut *annoter les arcs* par cette unique instruction conditionnelle afin

de traduire l'ensemble de la sémantique d'un branchement conditionnel.

La dernière instruction machine (`BGE`) de l'exemple de la Figure 3.3 est une instruction de branchement conditionnel. Le bloc d'instructions sémantiques associé est vide, mais une instruction sémantique `assume` est ajoutée sur chacun des deux arcs en sortie. Le chemin d'exécution qui prend l'arc *pris*, et qui pointera vers le bloc de base ayant pour adresse [5] `0x0x84AC`, est annoté par l'instruction sémantique `assume` $\geq r_{16}$, signifiant que le résultat de la comparaison (ici, signée) enregistrée dans le registre r_{16} indique une relation $\geq$. Ici, cela implique que $r_4 \geq t_1$. À l'inverse, l'autre chemin, suivant l'arc *non pris* et continuant en séquence vers le bloc à l'adresse `BXGE` $+ 4$, est annoté par une instruction `assume` $< r_{16}$ (cas $r_4 < t_1$).

Remarque. L'instruction `assume` informe d'une propriété vraie à un point du programme. La plupart des analyses peuvent donc faire le choix conservateur de l'ignorer.

3.1.1.5 Conclusion

Les instructions sémantiques permettent, d'une part, de s'abstraire complètement des spécificités de l'architecture pour laquelle un programme binaire est écrit, et d'autre part de faciliter les analyses en minimisant le nombre d'instructions à traiter. L'interprétation de chaque bloc de base se fera par l'intermédiaire de cette représentation sous la forme de blocs d'instructions sémantiques.

Après avoir précisé les mécanismes d'analyse de blocs de bases, nous présentons maintenant la nature des CFG générés par les programmes, ainsi que diverses méthodes de transformation de ceux-ci.

3.1.2 Graphes de flot de contrôle

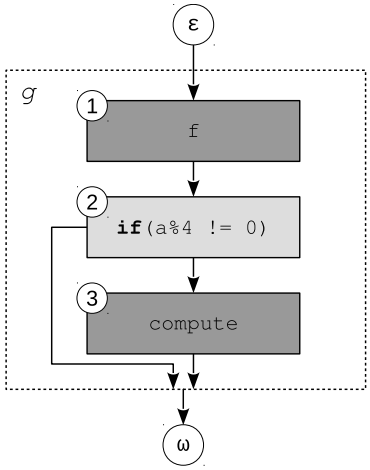
3.1.2.1 Sous-programmes

Pour chaque programme binaire analysé, le framework produit un CFG par sous-programme, et les relie entre eux à l'aide de *blocs virtuels*. Ce mode de fonctionnement est illustré sur l'Exemple 1 de la Figure 3.4a. Nous considérerons que la fonction analysée est `g`. La Figure 3.4b montre ce à quoi le CFG de `g` devrait ressembler (en faisant abstraction des particularités de l'assembleur). Les blocs sombres sont les blocs virtuels, ils représentent une exécution de la fonction en question.

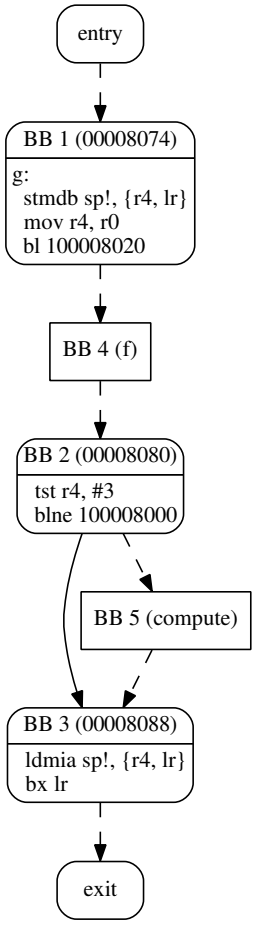
5. L'adresse d'un bloc de base est définie par l'adresse de sa première instruction.

```
void g(int a) {
  f(a);
  if(a % 4 != 0)
    compute();
}
void f(int a) {
  int x = 0, y = 0, i;
  for(i = 0; i < N; i++) {
    x = x + 1;
    y = y + 2;
  }
  if(2 * x == y + a)
    compute();
}
```

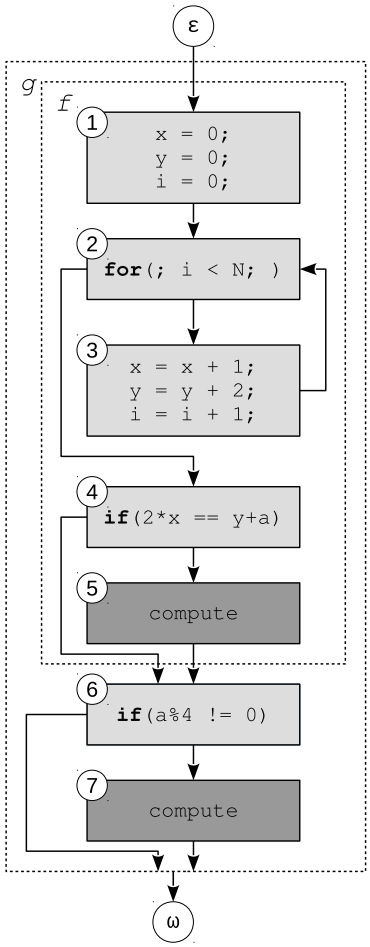
(a) Exemple 1 : programme C



(b) Schéma du CFG de `g`



(c) CFG de `g` généré



(d) Schéma du CFG de `g` après aplatissement de `f`

Figure 3.4 – Construction de CFG illustrée sur l'Exemple 1

Après compilation avec `gcc -O1` pour une architecture ARM5, la Figure 3.4c affiche le CFG de `g` tel qu'il est généré par le framework. Le bloc de base `BB 4` représente

l'exécution de `f`, et `BB 5` l'exécution de `compute`. Le bloc `BB 1` se charge de sauvegarder le contexte, d'appeler `f` et de positionner le paramètre a dans `r4`. Enfin, le bloc `BB 3` rétablit le contexte sauvegardé et exécute le branchement de retour de la fonction `g`. Les CFG de `f` et `compute` ne sont pas représentés sur la Figure 3.4, mais ils sont également générés.

Il est par ailleurs possible de choisir une vue aplatie (*inline*) des CFG, auquel cas le corps des fonctions appelées est inséré au point d'appel. La Figure 3.4d montre l'effet de l'application de cette technique sur l'appel de `f` dans `g`. Celle-ci permet de calculer des propriétés dépendantes du contexte d'appel plutôt que de les agglomérer pour l'ensemble des appels. Cette variation de représentation ne change pas la sémantique exprimée, elle change simplement la vision du programme par une analyse, et peut aider à améliorer sa précision.

3.1.2.2 Boucles

Chaque boucle dans un CFG est identifiée par le bloc de base constituant sa *tête* (ou *loop header*). Sa définition se base sur les propriétés de dominance dans un graphe, dont la définition historique est la suivante :

> *We say box* i *dominates box* j *if every path [...] which passes through box* j *must also pass through box* i. *Thus box* i *dominates box* j *if box* j *is subordinate to box* i *in the program.*
>
> Prosser [83]

Cette définition est évoquée par Cooper et. al [29], qui présentèrent en 2001 un algorithme efficace pour l'identification de ces propriétés.

Nous définissons en plus la propriété duale :

Définition 3.3. Dans un CFG (V, E, ϵ, ω),

(1) un bloc b_1 *domine* un bloc b_2 si et seulement si tous les chemins de ϵ à b_2 contiennent b_1 ;

(2) un bloc b_1 *post-domine* un bloc b_2 si et seulement si tous les chemins de b_2 à ω contiennent b_1. Une condition équivalente est que b_1 domine b_2 sur le CFG

inverse.

Les mêmes propriétés de dominance et post-dominance peuvent être définies pour les arcs du CFG.

Nous pouvons maintenant définir quelques propriétés de boucles, dont celle de tête de boucle :

Définition 3.4. Une *boucle* L d'un CFG (V, E, ϵ, ω) est un sous-ensemble de sommets $L \subseteq V$ vérifiant la propriété $\mathscr{B}$ suivante :

$$\mathscr{B}(L) \iff \forall b \in L,\ \exists b_1, b_2, \ldots, b_n \in L,\\ (b \to b_1 \in E) \wedge (b_n \to b \in E) \wedge \forall k \in [1, n[,\ b_i \to b_{i+1} \in E$$

Une *tête de boucle* est un bloc $h \in L$ qui domine tous les éléments d'une boucle L. Une bloc h peut être tête de boucle pour plusieurs boucles du graphe, mais définit par $\mathscr{L}$ une boucle maximale unique comme l'union de toutes les boucles dont h est la tête :

$$\mathscr{L}(h) \coloneqq \bigcup \{L \subseteq V \mid \mathscr{B}(L) \wedge \forall b \in L,\ h \operatorname{dom} b\}$$

Enfin, $b_1 \to b_2 \in E$ est un *arc de sortie* de la boucle L ssi $b_1 \in L$ et $b_2 \notin L$.

Si toute tête de boucle définit une boucle maximale unique, toute boucle ne définit pas une tête de boucle ! En effet, certains programmes contiennent des boucles qui ont plusieurs point d'entrée, comme par exemple le CFG de la Figure 3.5 (la boucle est faite de deux blocs, tous deux points d'entrée). Ces boucles sont dites *irrégulières*. Heureusement, elles peuvent toujours être transformées en une boucle régulière, contenant une tête de boucle [103].

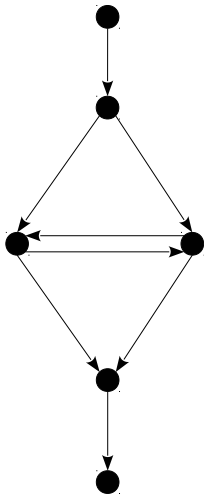


Figure 3.5 – Boucle irrégulière

3.1.2.3 Récursivité

Les appels récursifs sont, du point de vue de l'ensemble du programme, des boucles. Cependant, la récursivité peut

poser quelques difficultés aux analyses de programmes, et demander un traitement particulier. Afin de minimiser ces problèmes, il est possible de transformer des appels de fonctions récursives en boucles, en aplatissant les fonctions appelées. Après remplacement des blocs virtuels d'appel à une fonction récursive par le CFG appelé, une boucle classique apparaît dans le CFG appelant.

Malgré cet effort de transformation, une analyse de programme peut avoir besoin de reconnaître et de faire des traitements spécifiques pour les fonctions récursives. Notamment, le pointeur de pile est généralement différent à chaque appel récursif, et la structure des boucles engendrées par transformation de fonctions récursives est telle qu'il peut être difficile d'identifier la valeur du pointeur de pile en sortie de boucle.

Un programme correct doit être tel que tout appel de fonction se termine avec le pointeur de pile à la même valeur qu'au moment de son appel – il est donc possible de circonvenir ce problème en annotant les boucles concernées avec cette information.

3.1.2.4 CFG sémantique

Il peut être utile de diviser les blocs de base d'un CFG afin de conserver la propriété de séquentialité des blocs de base après traduction des instructions machine en instructions sémantiques. En effet, il existe dans certains jeux d'instructions des *instructions gardées* (*guarded instructions*), telles que les instructions ARM portant les suffixes `-EQ`, `-NE`, `-GE`, `-LO`... Ces instructions sont exécutées en séquence dans un bloc de base, mais n'ont d'effet que si une condition (précédemment évaluée) est vérifiée.

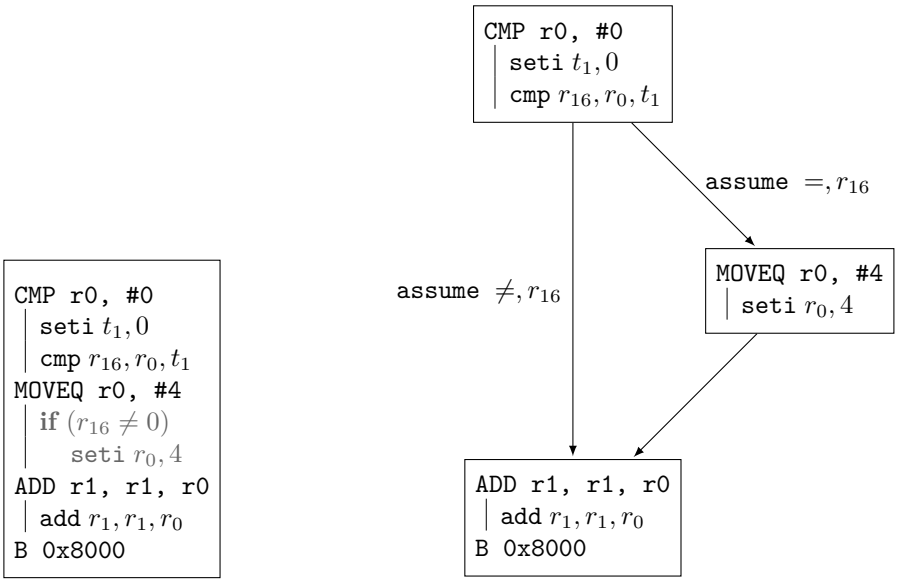
Dans ce cas, il est possible de réécrire le CFG en utilisant un branchement conditionnel virtuel, qui n'a pas réellement lieu à l'exécution du programme, mais qui est exprimé par la sémantique du code assembleur. Le bloc est alors coupé en deux, et les deux moitiés sont reliées par :

(a) un chemin, annoté par une instruction `assume` indiquant que la condition a été évaluée à *vrai*, traduisant l'instruction gardée par un bloc sémantique contenant la ou les instruction(s) sémantiques correspondant au code gardé ;

(b) un chemin, annoté par une instruction `assume` indiquant que la condition a été évaluée à *faux*, passant l'instruction gardée.

La Figure 3.6 illustre cette technique, en dédoublant les chemins d'exécution et

en n'incluant l'instruction `MOVEQ` sur un seul chemin. Cela permet, en préservant la propriété de séquentialité des blocs de base, de simplifier les analyses qui n'auront réellement qu'un seul chemin à traiter par bloc, en leur permettant de ne pas réécrire les algorithmes de parcours de graphes pour l'intérieur des blocs de bases.



(a) Bloc de base assembleur à traduire

(b) Représentation après division en blocs sémantiques séquentiels

Figure 3.6 – Séquentialisation des blocs de base par rapport aux blocs sémantiques

3.1.2.5 Branchements dynamiques

Les branchements dynamiques, sont des branchements à des adresses variables – en pratique restreintes à un certain champ de valeurs – résultant d'un calcul effectué à l'exécution du programme (branchements indirects). Ce problème apparaît généralement à la traduction de code contenant des `switch` ou des pointeurs de fonctions. Il faut dans ce cas identifier les adresses de branchements possibles et construire un arc sortant pour chacune. Diverses techniques résolvent ce problème directement sur le code machine, comme celle d'Holsti et. al [52] (pour les `switch`), ou celle de Sun et Cassé [100], qui utilisent le *slicing* pour réduire la complexité et analyser des programmes de grande taille.

3.1.2.6 Slicing

Le *slicing* de programme est une vieille méthode, ayant pour but de réduire un programme à une forme minimale sans modifier un certain comportement, par exemple sans affecter son flot dans les conditions et les boucles. Il s'agit de le réduire à un sous-ensemble du programme original qui contient toutes les informations nécessaires à un certain usage (analyse statique, mesures, débogage...) tout en le simplifiant par la suppression d'instructions ou de variables. Le *slicing* peut donc être utilisé pour alléger la complexité de certaines analyses de programme.

Cette méthode, présentée à l'origine par Weiser en 1981 [104], a été ensuite développée pour des concepts différents, en de nombreuses variations [102]. L'algorithme, assez coûteux dans sa version originale, est perfectionné par Horwitz, Reps et al. [54, 87]. En 2006, une équipe de Mälardalen propose une version plus adaptée au code binaire [93], mais qui souffre d'importantes pertes de précision. Plus récemment, la publication suscitée de Sun et Cassé [100] définit une technique de *slicing* de code binaire plus précise. Une autre approche, dite d'*analyse statique guidée* [42], améliore la précision de l'analyse des boucles en restreignant le comportement du programme à chaque phase d'analyse.

Après avoir traité de la représentation de programmes assembleur, nous développons une représentation de la machine pertinente à notre analyse. Nous définissons par incréments successifs des abstractions de la machine à chaque section, ce dont la Figure 3.7 donne un aperçu. Pour chaque abstraction, nous fournirons les outils nécessaires à sa validation et à son utilisation pour l'interprétation du programme.

3.2 Représentation de la machine

Nous rappelons que les machines considérées sont des architectures 32 bits, et que les applications étudiées sont mono-tâches, terminent, et sont considérées correctes (exemptes de bogue ou d'erreur). Ce dernier point implique entre autres l'absence d'accès à une mémoire interdite (par exemple à l'adresse nulle), de divisions par zéro ou de toute autre levée d'exceptions. Cela implique également que les zones de la mémoire accédées par des adresses relatives au pointeur de pile ou non sont disjointes. Ainsi, par exemple, nous faisons l'hypothèse qu'une instruction machine d'écriture à

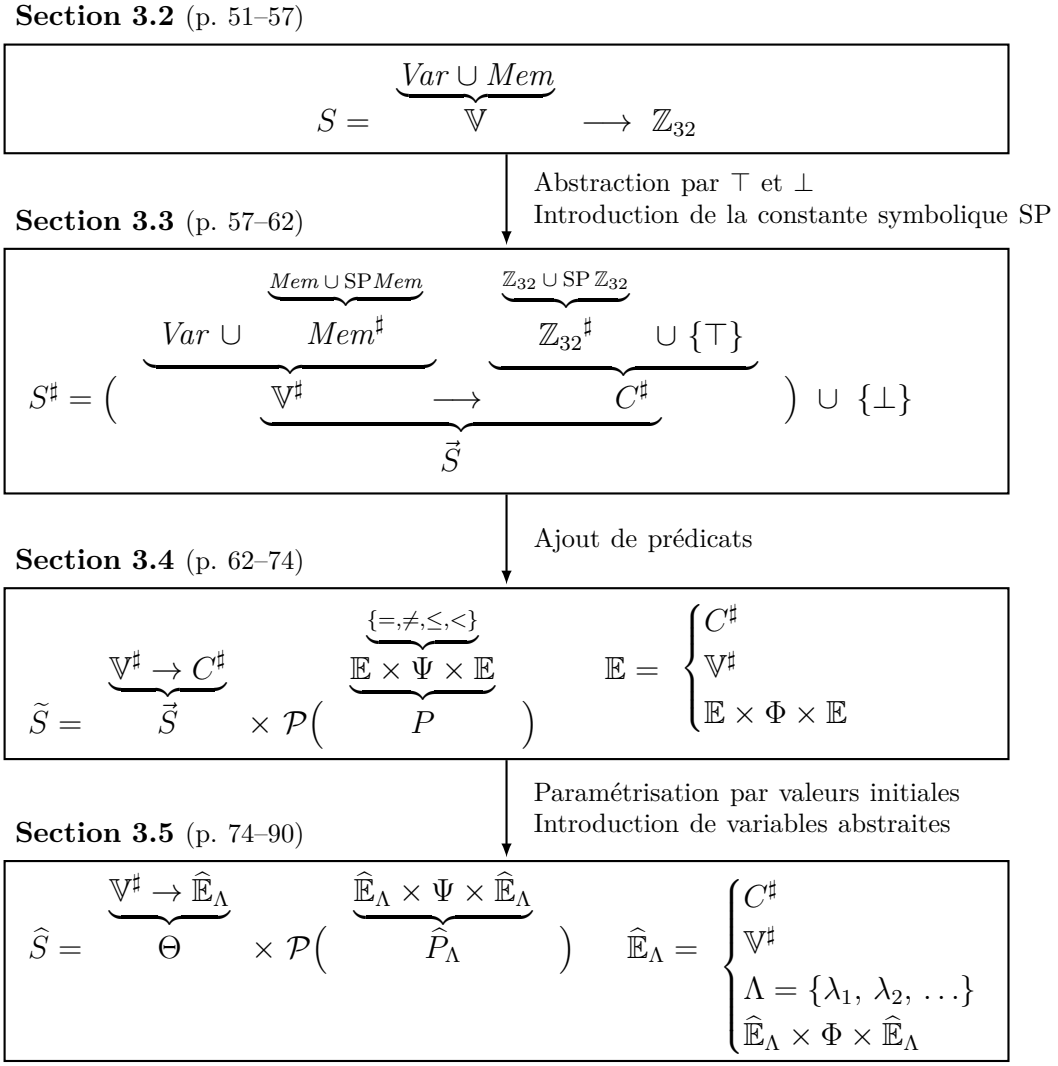


Figure 3.7 – Plan du Chapitre 3

l'adresse `0x8004` ne peut pas affecter une lecture à l'adresse SP-8, où SP est le pointeur de pile[6]. Enfin, nous considérons qu'un calcul d'adresse ne peut faire de dépassement (*underflow* ou d'*overflow*), phénomène, à notre connaissance, absent des programmes corrects.

Les variables (registres et cellules mémoire) dans un programme binaire ne sont pas typées, seuls les opérateurs le sont. Elles seront donc toutes représentées sur $\mathbb{Z}_{32}$, le groupe quotient abélien $(\mathbb{Z}/2^{32}, +)$, isomorphe avec la représentation 32 bits sur $\{0,1\}^{32}$. C'est au moment de la définition des instructions (par exemple, de décalages logiques et arithmétiques) que nous aurons recours à des projections vers l'ensemble des entiers signés et non signés.

Définition 3.5. Nous noterons dans la suite

(1) $Int_n \coloneqq [\![-2^{n-1}, 2^{n-1}-1]\!]$, l'ensemble des entiers signés sur n bits ;

(2) $Int_n^+ \coloneqq [\![0, 2^n-1]\!]$, l'ensemble des entiers non signés sur n bits.

Remarque. Il est intéressant de remarquer que la définition de l'opération de multiplication sur $\mathbb{Z}_{32}$ ne varie pas selon que l'on considère le domaine des entiers signés ou non signés – tout comme l'addition et la soustraction. C'est pourquoi les jeux d'instructions machine (et les instructions sémantiques) ne définissent souvent qu'une instruction indépendante du type considéré.

3.2.1 États concrets

$$S = \mathbb{V}^{\overbrace{Var \cup Mem}} \longrightarrow \mathbb{Z}_{32}$$

Nous représentons la mémoire par *Mem*, l'ensemble des cellules mémoire adressables sur une architecture 32 bits, de la taille d'un mot.

6. Du fait que dans les modèles de programmation prévalents, le programmeur n'a aucune connaissance de la position en mémoire où sera placée la pile, ce qui justifie cette hypothèse.

Définition 3.6. Sur une architecture 32 bits, la mémoire *Mem* est définie comme

$$Mem \coloneqq \left\{ m_a \mid a \in Int_{32}^{+} \cong (\mathbb{Z}_{32})^{2^{32}} \right\}$$

L'état de la machine en tout point du programme est représenté par une application des variables de programme – registres ou mémoire – vers l'ensemble des valeurs sur 32 bits. L'ensemble des états de la machine, dits *concrets*, peut donc être représenté par

$$(Reg \cup Mem) \to \mathbb{Z}_{32}$$

Les états concrets évoluent au fur et à mesure de l'exécution de programme, et seront mis à jour après chaque instruction exécutée. Étant donné que l'interprétation du programme se fera sur les instructions sémantiques, il est nécessaire de compléter les états concrets avec les variables temporaires, et définir S :

Définition 3.7. Soit $\mathbb{V}$ l'ensemble des variables complété avec les variables temporaires.

$$\begin{aligned} \mathbb{V} &\coloneqq Var \cup Mem \\ &= Reg \cup Tmp \cup Mem \end{aligned}$$

Nous enrichissons les états concrets pour considérer également les variables temporaires :

$$S \coloneqq \mathbb{V} \to \mathbb{Z}_{32}$$

3.2.2 Fonction d'interprétation concrète

Dans l'objectif de définir l'évolution des états de la machine au fil d'un programme, nous définissons la *fonction d'interprétation concrète* $\mathbb{I}$, opérant sur le jeu d'instructions sémantiques I_{sem}. $\mathbb{I}$ associe à chaque instruction sémantique une fonction de $S \to \mathcal{P}(S)$ qui met à jour un état concret [7]. Cette fonction peut avoir pour image un ensemble de plusieurs états concrets dans le cas où plusieurs états sont possibles après interprétation

7. L'interprétation du programme se fera *en avant*, c'est-à-dire dans le sens d'exécution du programme.

d'une instruction (cela n'arrive qu'avec `scratch`, qui introduit une valeur inconnue). $\mathbb{I}$ peut aussi envoyer sur un ensemble vide, lorsque l'état ne correspond pas à une condition nécessaire pour emprunter un bloc du CFG (exprimée par `assume`).

Par exemple, les instructions sémantiques `add`, `sub`, `mul` écrivent dans le premier registre le résultat de l'opération arithmétique correspondante sur les deux autres registres. Ainsi, l'interprétation de l'instruction `sub` r_0, r_0, t_1 se définit par :

$$\mathbb{I}[\texttt{sub}\ r_0, r_0, t_1](s) \coloneqq \{s[r_0 \mapsto s(r_0) - s(t_1)]\}\ \ \forall s \in S$$

où $s[x \mapsto k]$ dénote s modifié de façon à ce que l'image de x soit k :

Définition 3.8. Pour tout état concret $s \in S$, pour toute variable $x \in \mathbb{V}$, pour toute valeur $k \in \mathbb{Z}_{32}$, nous définissons la notation suivante :

$$s[x \mapsto k] \coloneqq v \mapsto \begin{cases} k & \text{si } v = x \\ s(v) & \text{sinon} \end{cases}$$

La Définition 3.9 (page 56) définit l'interprétation d'un état concret de S sur l'ensemble des instructions sémantiques. Elle utilise pour cela quelques fonctions intermédiaires :

- $\phi_{Int_{32}} : \mathbb{Z}_{32} \to [\![-2^{n-1}, 2^{n-1} - 1]\!]$ et $\phi_{Int^+_{32}} : \mathbb{Z}_{32} \to [\![0, 2^n - 1]\!]$ sont les projections du groupe quotient $\mathbb{Z}_{32}$ vers, le domaine des entiers sur 32 bits respectivement signés et non signés. Ces projections permettent d'utiliser des opérations définies sur $\mathbb{Z}$, avant de les renvoyer sur $\mathbb{Z}_{32}$ par le morphisme canonique $\phi_{\mathbb{Z}_{32}} : \mathbb{Z} \to \mathbb{Z}_{32}$.
- $\beta_i : \mathbb{Z}_{32} \to \{0, 1\}$ est la fonction renvoyant le i-ème bit d'un nombre. Elle peut être définie numériquement pour tout x par $\beta_i(x) \coloneqq \lfloor \frac{x}{2^i} \rfloor \bmod 2$.

La définition de $\mathbb{I}$ sur S est une étape intermédiaire qui est trivialement généralisée pour des ensembles d'états concrets par, $\forall s' \in \mathcal{P}(S)$, $\forall i \in I_{sem}$,

$$\mathbb{I}[i](s') \coloneqq \bigcup_{s \in s'} \mathbb{I}[i](s)$$

Définition 3.9. L'interprétation d'instructions sémantiques sur les états concrets est définie comme, $\forall s \in S$, $\forall d, a, b \in \mathbb{V}$, $\forall k \in \mathbb{Z}_{32}$, $\forall n > 0$,

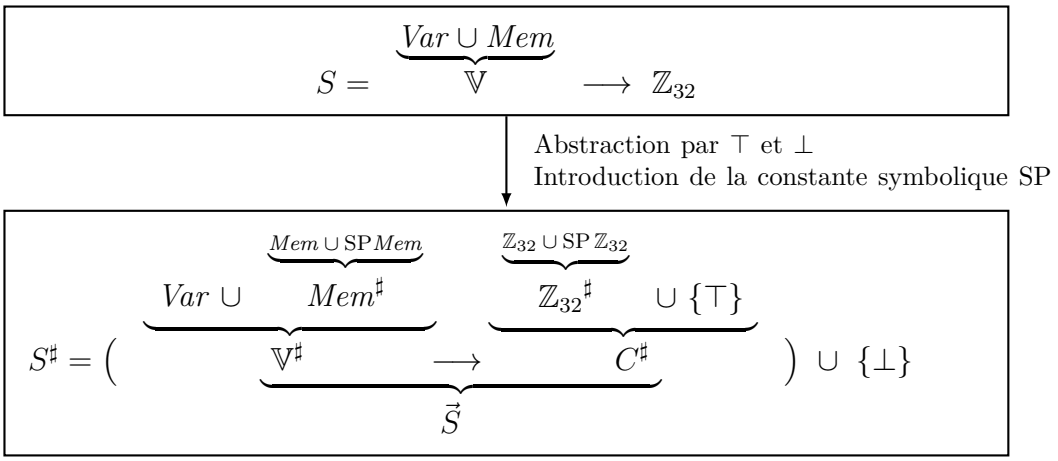
$$
\begin{aligned}
\mathbb{I}[\texttt{set}\ d, a](s) &\coloneqq \{s\,[d \mapsto s(a)]\} \\
\mathbb{I}[\texttt{seti}\ d, k](s) &\coloneqq \{s\,[d \mapsto k]\} \\
\mathbb{I}[\texttt{add}\ d, a, b](s) &\coloneqq \{s\,[d \mapsto s(a) + s(b)]\} \\
\mathbb{I}[\texttt{sub}\ d, a, b](s) &\coloneqq \{s\,[d \mapsto s(a) - s(b)]\} \\
\mathbb{I}[\texttt{mul}\ d, a, b](s) &\coloneqq \{s\,[d \mapsto s(a) \times s(b)]\} \\
\mathbb{I}[\texttt{div}\ d, a, b](s) &\coloneqq \left\{ s \left[d \mapsto \phi_{\mathbb{Z}_{32}} \left(\frac{\phi_{Int_{32}}(s(a))}{\phi_{Int_{32}}(s(b))} \right) \right] \right\} \\
\mathbb{I}[\texttt{divu}\ d, a, b](s) &\coloneqq \left\{ s \left[d \mapsto \phi_{\mathbb{Z}_{32}} \left(\frac{\phi_{Int^+_{32}}(s(a))}{\phi_{Int^+_{32}}(s(b))} \right) \right] \right\} \\
\mathbb{I}[\texttt{mod}\ d, a, b](s) &\coloneqq \left\{ s \left[d \mapsto \phi_{\mathbb{Z}_{32}} \Big(\phi_{Int_{32}}(s(a)) \bmod \phi_{Int_{32}}(s(b)) \Big) \right] \right\} \\
\mathbb{I}[\texttt{modu}\ d, a, b](s) &\coloneqq \left\{ s \left[d \mapsto \phi_{\mathbb{Z}_{32}} \Big(\phi_{Int^+_{32}}(s(a)) \bmod \phi_{Int^+_{32}}(s(b)) \Big) \right] \right\} \\
\mathbb{I}[\texttt{shl}\ d, a, b](s) &\coloneqq \left\{ s \left[d \mapsto s(a) \times 2^{s(b)} \right] \right\} \\
\mathbb{I}[\texttt{shr}\ d, a, b](s) &\coloneqq \left\{ s \left[d \mapsto \phi_{\mathbb{Z}_{32}} \left(\frac{\phi_{Int^+_{32}}(s(a))}{\phi_{Int_{32}}(2^{s(b)})} \right) \right] \right\} \\
\mathbb{I}[\texttt{asr}\ d, a, b](s) &\coloneqq \left\{ s \left[d \mapsto \phi_{\mathbb{Z}_{32}} \left(\frac{\phi_{Int^+_{32}}(s(a))}{\phi_{Int_{32}}(2^{s(b)})} \right) \right] \right\} \\
\mathbb{I}[\texttt{neg}\ d, a](s) &\coloneqq \{s\,[d \mapsto \overline{0} - s(a)]\} \\
\mathbb{I}[\texttt{not}\ d, a](s) &\coloneqq \{s\,[d \mapsto \overline{-1} - s(a)]\} \\
\mathbb{I}[\texttt{and}\ d, a, b](s) &\coloneqq \left\{ s \left[d \mapsto \sum_{0 \le i < 32} \beta_i(s(a)) \times \beta_i(s(b)) \right] \right\} \\
\mathbb{I}[\texttt{or}\ d, a, b](s) &\coloneqq \left\{ s \left[d \mapsto \sum_{0 \le i < 32} \max(\beta_i(s(a)), \beta_i(s(b))) \right] \right\} \\
\mathbb{I}[\texttt{cmp}\ d, a, b](s) &\coloneqq \{s\,[d \mapsto s(a) \sim_* s(b)]\} \\
\mathbb{I}[\texttt{cmpu}\ d, a, b](s) &\coloneqq \{s\,[d \mapsto s(a) \sim_+ s(b)]\} \\
\mathbb{I}[\texttt{load}\ d, a, t](s) &\coloneqq \left\{ s \left[d \mapsto m_{s(a)} \right] \right\} \\
\mathbb{I}[\texttt{store}\ d, a, t](s) &\coloneqq \left\{ s \left[m_{s(a)} \mapsto s(d) \right] \right\} \\
\mathbb{I}[\texttt{scratch}\ d](s) &\coloneqq \left\{ s' \in S \,\middle|\, \forall v \in \mathbb{V},\ (v \neq d) \Rightarrow s'(v) = s(v) \right\} \\
\mathbb{I}[\texttt{assume}\ c, a](s) &\coloneqq \begin{cases} \{s\} & \text{si } c(s(a)),\ a \text{ résultant d'une comparaison} \\ \emptyset & \text{sinon} \end{cases}
\end{aligned}
$$

Lemme 3.1. *Si l'interprétation par $\mathbb{I}$ (en partant en début de programme de $S \in \mathcal{P}(S)$, l'ensemble de tous les états possibles) de la séquence d'instructions correspondant à un chemin d'exécution du programme résulte en un ensemble vide $\emptyset \in \mathcal{P}(S)$, ce chemin est infaisable : il ne sera emprunté pour aucun état en entrée.*

Cette représentation exhaustive des états possibles d'un programme, par l'énumération des valeurs prises par chacune de ses variables, est un outil théorique utile, relativement simple à définir, mais une représentation bien trop lourde pour être implémentée telle quelle.

Plutôt que de maintenir au fil du programme l'ensemble exact d'états concrets possibles en chaque point du programme (représenté par un CFG et des blocs d'instructions sémantiques), il est plus judicieux de définir et manipuler une abstraction adéquate.

3.3 Abstraction de la machine



Cette section présente une abstraction d'ensemble d'états de la machine. Nous partons d'une structure simple, proche des états concrets, et l'enrichissons au fur et à mesure pour améliorer son expressivité.

3.3.1 Le domaine $S_0^\sharp$

Nous définissons une première abstraction naïve, imitant la structure des états concrets : $S_0^\sharp$. Une valeur sur 32 bits est assignée à chaque variable du programme, ou

$\top$ (représentant une valeur inconnue) si plusieurs valeurs sont possibles. S'il existe une variable pour laquelle aucune valeur n'est possible, cela implique qu'il n'y a aucun état possible. Nous notons $\bot \in S_0^\sharp$ l'état abstrait correspondant à ce scénario.

Définition 3.10. Le domaine abstrait $S_0^\sharp$ est défini par :

$$S_0^\sharp := (\mathbb{V} \to \mathbb{Z}_{32} \cup \{\top\}) \cup \{\bot\}$$

La fonction d'interprétation abstraite sur $S_0^\sharp$ est définie de manière similaire à la définition de $\mathbb{I}$ sur le domaine concret. Il suffit d'étendre l'arithmétique sur $\mathbb{Z}_{32}$ à $\mathbb{Z}_{32} \cup \{\top\}$, en définissant toute opération contenant un opérand $\top$ comme ayant pour résultat $\top$: le résultat d'un calcul contenant une inconnue est, dans le cas général, inconnu. Quelques différences : (a) l'instruction `scratch` d est traduite par une mise à $\top$ de d, et (b) l'information apportée par l'instruction `assume` c, a ne peut être utilisée que si c vaut "=", ou si la condition dans a contredit les informations portées par l'état (par exemple, $(c, a) = (\leq, r_1 \sim 0)$ et $s^\sharp(r_1) = 8$), auquel cas l'état obtenu est $\bot$.

Un défaut majeur immédiat de cette simple représentation est son impuissance à travailler avec des variables dans la pile (de type $m_{\mathrm{SP}+k}$) sans connaître la valeur du pointeur de pile, pointeur qui n'est pas forcément connu avant l'analyse ! La Figure 3.8 illustre ce problème sur un code de chargement très courant de variable rangée dans la pile. L'adresse est calculée à partir d'un registre contenant un pointeur vers la pile [8], décalé d'un déplacement constant (ici $k = -16$). En l'absence d'information sur la valeur de ce pointeur de pile, l'adresse sera évaluée à $\top + k = \top$. Ainsi, ce domaine abstrait peut difficilement travailler avec des données rangées dans la pile.

```
ldr r2, [fp, #-16]
  seti t2, −16
  add t1, r11, t2
  load r2, t1, ℕ32
```

Figure 3.8 – Chargement dans le pile

3.3.2 Le domaine $S^\sharp$

Nous étendons $S_0^\sharp$ pour représenter symboliquement la valeur du pointeur de pile en début d'analyse. Le registre qui contient cette information est défini par les conven-

8. En réalité, le registre ARM `fp` (`ebp` en x86) est le *frame pointer*, qui indique le début du contexte de la fonction en cours. C'est par le biais d'adresses relatives à ce pointeur qu'un programme accède typiquement à des informations rangées dans la pile, à l'intérieur d'une fonction.

tions de chaque jeu d'instructions (r_{13} en ARM, r_1 en PowerPC...). Cette constante symbolique, notée SP, correspond à la valeur du pointeur de pile au point où l'analyse du programme a commencé (entrée d'une fonction ou d'un programme).

Définition 3.11. Nous définissons le *domaine des constantes* comme l'ensemble des valeurs sur 32 bits, relatives au pointeur de pile initial SP ou non.

$$\mathbb{Z}_{32}{}^\sharp \coloneqq \mathbb{Z}_{32} \cup \mathrm{SP}\,\mathbb{Z}_{32}$$

Toute l'arithmétique de $\mathbb{Z}_{32}$ n'est pas définie sur $\mathbb{Z}_{32}{}^\sharp$, par exemple l'addition de $\mathrm{SP} + 4$ et $\mathrm{SP} + 0$ n'est pas permise. Nous enrichissons donc dans ce domaine par l'ajout d'un élément $\top$:

$$C^\sharp \coloneqq \mathbb{Z}_{32}{}^\sharp \cup \{\top\}$$

On peut ainsi définir toutes les opérations sur $C^\sharp$:

$$\begin{aligned}
\forall k, k' \in \mathbb{Z}_{32}, \forall c \in C^\sharp, \quad & \forall k, k' \in \mathbb{Z}_{32}, \forall c \in C^\sharp, \\
(\mathrm{SP} + k) + k' &\coloneqq \mathrm{SP} + (k + k') & (\mathrm{SP} + k) - k' &\coloneqq \mathrm{SP} + (k - k') \\
k + (\mathrm{SP} + k') &\coloneqq \mathrm{SP} + (k + k') & k - (\mathrm{SP} + k') &\coloneqq \top \\
(\mathrm{SP} + k) + (\mathrm{SP} + k') &\coloneqq \top & (\mathrm{SP} + k) - (\mathrm{SP} + k') &\coloneqq k - k' \\
c + \top = \top + c &= \top & c - \top = \top - c &= \top
\end{aligned}$$

Et ainsi de suite pour les autres opérateurs arithmétiques. L'application d'opérateurs logiques sur $\mathrm{SP}\,\mathbb{Z}_{32}$ résulte toujours en $\top$: nous n'autorisons pas (conservativement) ce type d'opérateurs pour le calcul d'adresse.

Nous redéfinissons la mémoire, en séparant le tas (*heap*) et les variables globales, accessibles par adresses absolues de la forme k, de la pile (*stack*), accessible uniquement avec des adresses relatives à un pointeur de pile SP, de la forme $\mathrm{SP} + k$.

Définition 3.12. Pour une architecture 32 bits, nous redéfinissons la mémoire par

$$\begin{aligned} Mem^\sharp &\coloneqq \{m_k, k \in \mathbb{Z}_{32}\} \cup \{m_{\mathrm{SP}+k}, k \in \mathbb{Z}_{32}\} \\ &\cong \{m_a, a \in {\mathbb{Z}_{32}}^\sharp\} \end{aligned}$$

et définissons $\mathbb{V}^\sharp$, le domaine des variables avec cette mémoire réorganisée :

$$\mathbb{V}^\sharp \coloneqq Reg \cup Tmp \cup Mem^\sharp$$

Les variables dans la pile (de la forme $m_{\mathrm{SP}+k} \in Mem^\sharp$) sont donc désormais indexées uniquement par rapport à ce SP initial. Ainsi, par exemple, m_{8192} est une cellule mémoire dans le tas, et $m_{\mathrm{SP}-4}$ est une cellule mémoire dans la pile du programme.

Une fois le domaine des constantes définies, nous pouvons redéfinir le domaine abstrait :

Définition 3.13. Un état abstrait peut être :

(i) une application de chaque variable vers une constante, dans $\vec{S} \coloneqq \mathbb{V}^\sharp \to C^\sharp$

(ii) $\bot$, l'état impossible

Le domaine abstrait $S^\sharp$ est donc défini par

$$S^\sharp \coloneqq \vec{S} \cup \{\bot\} = (\mathbb{V}^\sharp \to C^\sharp) \cup \{\bot\}$$

Notons $sp \in Reg$, le registre correspondant au pointeur de pile donné par le jeu d'instructions. Il suffit maintenant, pour pouvoir traiter des données dans la pile, de s'assurer que l'état initial $s^\sharp$ utilisé en début d'analyse de programme assigne à sp la constante symbolique SP : $s^\sharp(sp) = \mathrm{SP} + 0$. Cette propriété est, par définition, toujours valide en début d'analyse.

Afin de permettre la concrétisation de ce domaine abstrait vers S, nous notons κ_{SP} l'adresse de départ de la pile. Cette adresse peut être différente à chaque exécution et est un paramètre de l'exécution d'un programme inconnu pendant l'analyse statique. Cette constante $\kappa_{\mathrm{SP}} \in \mathbb{Z}_{32}$ représente le contexte d'exécution du programme et paramétrise désormais les fonctions de concrétisation.

Définition 3.14. Nous définissons la fonction de concrétisation d'une constante dans un contexte d'exécution :

$$\gamma_{\mathbb{Z}_{32}{}^\sharp} : \mathbb{Z}_{32}{}^\sharp \longrightarrow \mathbb{Z}_{32}$$

$$\kappa_{\mathrm{SP}} \vdash \quad c \longmapsto \begin{cases} k & \text{si } c = k, \quad k \in \mathbb{Z}_{32} \\ \kappa_{\mathrm{SP}} + k & \text{si } c = \mathrm{SP} + k, \ k \in \mathbb{Z}_{32} \end{cases}$$

Nous considérerons par la suite ce contexte d'exécution κ_{SP} comme implicite. La concrétisation de $S^\sharp$ se définit simplement :

Définition 3.15. La fonction de concrétisation $\gamma_{S^\sharp} : S^\sharp \to \mathcal{P}(S)$ est définie par :

$$\kappa_{\mathrm{SP}} \vdash \ \forall s^\sharp \in S^\sharp, \gamma_{S^\sharp}(s^\sharp) :=$$

$$\begin{cases} \left\{ s \in S \;\middle|\; \begin{array}{c} \forall v \in \mathit{Var}, s^\sharp(v) = s(v) \vee \ s^\sharp(v) = \top \\ \wedge \forall m_a \in \mathit{Mem}^\sharp, s^\sharp(m_a) = s(m_{\gamma_{\mathbb{Z}_{32}{}^\sharp}(a)}) \vee \ s^\sharp(m_a) = \top \end{array} \right\} & \text{si } s^\sharp \in \vec{S} \\ \emptyset & \text{si } s^\sharp = \bot \end{cases}$$

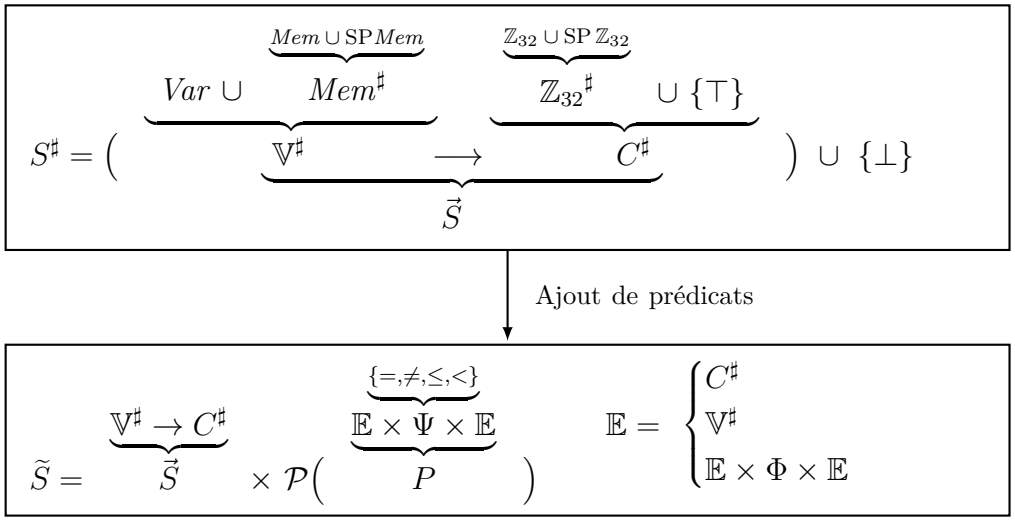
Bien que ce premier problème de représentation de la pile soit adressé par le domaine abstrait $S^\sharp$, celui-ci reste trop simple pour trouver des propriétés non triviales sur le programme, notamment celles qui peuvent permettre la détection de chemins infaisables non triviaux, autres que du code (dynamiquement) mort. Les seuls ensembles de valeurs considérées pour chaque variable du programme sont des singletons ou la valeur inconnue $\top$ – en réalité, ce domaine abstrait s'approche d'un 1-set (cf. Section 2.3.4.2).

```
void f(const int x) {
  if(x > 10) {
    /* a */
  }
  if(x == 2) {
    /* b */
  }
}
```

Figure 3.9 – Conflit indétectable par $S^\sharp$

La Figure 3.9 donne un exemple de programme (représenté en C, pour simplifier) où l'interprétation avec $S^\sharp$ souffre du manque d'expressivité de cette abstraction. Lorsque x est inconnu, le chemin qui passe par a et b est infaisable, mais l'information $x > 10$ issue du décodage du code machine, certainement traduit avec des instructions sémantiques `assume`, ne pourra pas être représentée dans $S^\sharp$.

3.4 Abstraction par prédicats



Nous avions présenté dans la Section 2.3 différents domaines abstraits suffisamment puissants pour détecter le conflit sur cet exemple (intervalles, polyèdres...). Cependant, d'une part, de telles abstractions peuvent ne pas être adaptées à l'analyse du binaire, notamment à cause de contraintes dues à la gestion de la mémoire, et d'autre part, que notre analyse se polarise, pour la recherche de chemins infaisables, sur les propriétés issues des conditions, et nous ne souhaitons pas nous limiter à l'expression d'une classe particulière de propriétés (par exemple, se restreindre aux contraintes linéaires). Nous choisissons donc d'effectuer l'analyse de programme avec une abstraction plus libre et expressive : une conjonction de prédicats. Si cette liberté de représentation entraîne une complexité trop élevée (au niveau de l'interprétation du programme, ou de la recherche de conflits), nous pourrons chercher à restreindre ou adapter cette abstraction pour rendre l'analyse plus évolutive sans perdre (trop) en termes de découverte de chemins infaisables. Cette approche nous paraît adaptée à la recherche de chemins infaisables en raison de l'inconnu qui les entoure généralement, et de la difficulté à anticiper le type de chemins infaisables qu'un programme peut contenir.

3.4.1 Le domaine $\widetilde{S}$

3.4.1.1 Définition

Nous souhaitons étendre $S^\sharp$ pour pouvoir inclure dans les états abstraits les informations sur les variables du programme découvertes au fur et à mesure de son

interprétation, sous la forme d'une liste de prédicats. L'ensemble de ces prédicats (et donc leur conjonction) sera valide pour tout exécution du programme parcourant le chemin analysé. Nous définissons d'abord le domaine des prédicats :

Définition 3.16. Un prédicat de P est constitué d'une relation binaire (opérateur de comparaison) et de deux opérandes :

$$P \coloneqq \mathbb{E} \times \Psi \times \mathbb{E}$$

L'ensemble des relations binaires Ψ est défini comme $\Psi \coloneqq \{=, \neq, \leq, <\}$. Les relations binaires $\leq$ et $<$ étant des relations d'ordre, les relations réciproques $\geq$ et $>$ sont superflues. L'ensemble des expressions sur les variables du programme $\mathbb{E}$ est défini inductivement :

$$\mathbb{E} \coloneqq \begin{array}{|c} C^\sharp \\ \mathbb{V}^\sharp \\ \mathbb{E} \times \Phi \times \mathbb{E} \end{array}$$

où $\Phi \coloneqq \{+, -, \times, /, \mathrm{mod}, \sim\}$.

Exemple. $r_1 > 0$, $m_{\mathrm{SP}-4} \leq r_3$ ou encore $2 \times t_1 = r_0 + 1$ sont des prédicats.

Nous étendons enfin $S^\sharp$ à $\widetilde{S}$ pour y inclure un ensemble de prédicats :

Définition 3.17. Le domaine abstrait $\widetilde{S}$ est défini par :

$$\widetilde{S} \coloneqq \vec{S} \times \mathcal{P}(P) = (\mathbb{V}^\sharp \to C^\sharp) \times \mathcal{P}(P)$$

Remarque. Il est également possible de raisonner avec seulement $\mathcal{P}(P)$ comme domaine abstrait. En effet, l'information portée par une application $\vec{s} \in \vec{S}$ peut être entièrement représentée par un ensemble de prédicats $p_{\vec{s}} \in \mathcal{P}(P)$ défini comme tel :

$$p_{\vec{s}} \coloneqq \bigcup_{\substack{v \in \mathbb{V}^\sharp \\ \vec{s}(v) \neq \top}} \{v = \vec{s}(v)\}$$

Il est toutefois bien plus efficace de représenter les valeurs constantes dans une table. Nous verrons plus loin (section 3.4.2) que la recherche d'une valeur constante associée à

une variable est une opération très fréquente lors de l'interprétation d'un programme, notamment pour l'évaluation d'adresses en mémoire, et $\vec{S}$ fournit la représentation canonique dont les prédicats manquent. Il faudrait sinon parcourir tous les prédicats à la recherche d'un élément de la forme $v = c$, avec $c \in {\mathbb{Z}_{32}}^\sharp$, une opération coûteuse.

3.4.1.2 Concrétisation

Nous définissons d'abord la concrétisation d'un ensemble de prédicats :

Définition 3.18. La fonction de concrétisation $\gamma_P : \mathcal{P}(P) \to \mathcal{P}(S)$ pour un ensemble de prédicats est définie par :

$$\forall p \in \mathcal{P}(P),\ \gamma_P(p) \coloneqq \left\{ s \in S \;\middle|\; \begin{array}{l} \quad \forall (e_1, =, e_2) \in p,\ \exists x_1 \in X_1,\ \exists x_2 \in X_2,\ x_1 = x_2 \\ \wedge\, \forall (e_1, \neq, e_2) \in p,\ \exists x_1 \in X_1,\ \exists x_2 \in X_2,\ x_1 \neq x_2 \\ \wedge\, \forall (e_1, \leq, e_2) \in p,\ \exists x_1 \in X_1,\ \exists x_2 \in X_2,\ x_1 \leq x_2 \\ \wedge\, \forall (e_1, <, e_2) \in p,\ \exists x_1 \in X_1,\ \exists x_2 \in X_2,\ x_1 < x_2 \end{array} \;\middle|\; \begin{array}{l} \\ X_1 \coloneqq \gamma_{\mathbb{E}}(s, e_1) \\ X_2 \coloneqq \gamma_{\mathbb{E}}(s, e_2) \\ \\ \end{array} \right\}$$

où $\gamma_{\mathbb{E}} : \widehat{S} \times \mathbb{E} \to \mathcal{P}(\mathbb{Z}_{32})$ est la fonction de concrétisation d'une expression dans un état, définie inductivement par :

$$\forall e \in \mathbb{E},\ \gamma_{\mathbb{E}}(s, e) \coloneqq \begin{cases} \mathbb{Z}_{32} & \text{si } e = \top \\ \{\gamma_{{\mathbb{Z}_{32}}^\sharp}(e)\} & \text{si } e \in {\mathbb{Z}_{32}}^\sharp \\ \{s(e)\} & \text{si } e \in \mathit{Var} \\ \{s(m_{\gamma_{{\mathbb{Z}_{32}}^\sharp}(a)})\} & \text{si } e \in \mathit{Mem}^\sharp, \text{ avec } e = m_a \\ \bigcup_{a \in \gamma_{\mathbb{E}}(s, e_1)} \bigcup_{b \in \gamma_{\mathbb{E}}(s, e_2)} \phi(a, b) & \text{si } e = (e_1, \phi, e_2), \text{ avec } e_1, e_2 \in \mathbb{E},\ \phi \in \Phi \end{cases}$$

La concrétisation d'un état abstrait de $(\vec{s}, p) \in \widetilde{S}$ est l'ensemble des états abstraits respectant à la fois les contraintes de l'application $\vec{s}$ de chaque variable vers un élément de $C^\sharp$ et de l'ensemble des prédicats p (tel une conjonction). C'est donc l'intersection des états représentés par $\vec{s}$ et p.

Définition 3.19. La fonction de concrétisation $\gamma_{\widetilde{S}} : \widetilde{S} \to \mathcal{P}(S)$ se définit par :

$$\forall \widetilde{s} = (\vec{s}, p) \in \vec{S} \times \mathcal{P}(P),\ \gamma_{\widetilde{S}}(\widetilde{s}) \coloneqq \gamma_{S^\sharp}(\vec{s}) \cap \gamma_P(p)$$

$\vec{S}$ étant un sous-ensemble de $S^\sharp$, il suffit de réutiliser $\gamma_{S^\sharp}$.

3.4.1.3 Interprétation

La fonction d'interprétation $\widetilde{\mathbb{I}}$ de $\widetilde{S}$ doit à la fois maintenir (a) l'application de $\vec{S}$ et (b) l'ensemble de prédicats de $\mathcal{P}(P)$. Nous pouvons la décomposer en deux fonctions [9] $\widetilde{\mathbb{I}}_{C^\sharp} : \vec{S} \to \vec{S}$ et $\widetilde{\mathbb{I}}_P : \vec{S} \times \mathcal{P}(P) \to \mathcal{P}(P)$:

$$\widetilde{\mathbb{I}} : (\vec{s}, p) \mapsto (\widetilde{\mathbb{I}}_{C^\sharp}(\vec{s}), \widetilde{\mathbb{I}}_P(\vec{s}, p))$$

La fonction d'interprétation abstraite sur une telle application (a) a déjà été discutée dans les sections précédentes (Section 3.3.1, Section 3.3.2), et sera définie de manière quasi-identique à la fonction d'interprétation concrète (Définition 3.9), étendue sur $C^\sharp$. Voici les instructions qui diffèrent :

$$\forall \vec{s} \in \vec{S},$$

$$\widetilde{\mathbb{I}}_{C^\sharp}[\texttt{scratch}\ d](\vec{s}) \coloneqq \vec{s}[d \mapsto \top]$$

$$\widetilde{\mathbb{I}}_{C^\sharp}[\texttt{sub}\ d, a, b](\vec{s}) \coloneqq \begin{cases} \vec{s}[d \mapsto 0] & \text{si } a = b \\ \vec{s}[d \mapsto \vec{s}(a) - \vec{s}(b)] & \text{sinon} \end{cases}$$

$$\widetilde{\mathbb{I}}_{C^\sharp}[\texttt{div}\ d, a, b](\vec{s}) \coloneqq \begin{cases} \vec{s}[d \mapsto 1] & \text{si } a = b \\ \vec{s}[d \mapsto \vec{s}(a) \,/\, \vec{s}(b)] & \text{sinon} \end{cases}$$

$$\widetilde{\mathbb{I}}_{C^\sharp}[\texttt{mod}\ d, a, b](\vec{s}) \coloneqq \begin{cases} \vec{s}[d \mapsto 0] & \text{si } a = b \\ \vec{s}[d \mapsto \vec{s}(a) \,/\, \vec{s}(b)] & \text{sinon} \end{cases}$$

$$\widetilde{\mathbb{I}}_{C^\sharp}[\texttt{assume}\ c, a](\vec{s}) \coloneqq \vec{s}$$

9. Nous ferons usage d'informations sur les variables identifiées à une valeur constante pour l'interprétation par $\widetilde{\mathbb{I}}_P$ (prenant en compte les informations valides *avant* l'exécution de l'instruction considérées). À l'opposé, les prédicats de l'état n'ont pas d'utilité pour la fonction d'interprétation $\widetilde{\mathbb{I}}_{C^\sharp}$.

Distinguer les cas $a = b$ pour certaines opérations arithmétiques permet de traiter les cas où $\vec{s}(a)$ est inconnu ($\top$) et éviter une perte de précision lorsque possible. Par exemple, dans le cas d'une l'instruction `sub` d, a, a, si $\vec{s}(a) = \top$, il est préférable d'assigner 0 à $\vec{s}(d)$ que $\top - \top = \top$.

Refléter l'évolution du programme sur un ensemble de prédicats (b) dont la sémantique dépend des valeurs de chaque variable *au point actuel de l'analyse* est plus délicat. En effet, cela implique de mettre à jour, pour toute instruction modifiant une variable x, tous les prédicats relatifs à x.

3.4.2 Interprétation par les prédicats

Nous commençons par introduire un outil utile pour la manipulation de prédicats : la fonction d'invalidation d'une variable. Cette fonction vise à étendre l'effet d'une instruction opérant une modification inconnue sur une variable, sur l'ensemble des prédicats. Une implémentation conservatrice serait simplement de supprimer tous les prédicats affectés (exprimés en fonction de cette variable donc). Après tout, un prédicat apportant une information supplémentaire, il est toujours correct de le supprimer : il n'en résultera qu'une surapproximation (perte de précision mais pas de validité), comme l'énonce le Lemme 3.2. Nous souhaitons toutefois préserver autant d'information que se peut.

Lemme 3.2. *La suppression d'un prédicat dans un état est* sûre *: celle-ci entraîne toujours une surapproximation. Autrement dit,* $\forall p \in \mathcal{P}(P)$, $\forall p_0 \in P$ *:*

$$\gamma_P(p \cup \{p_0\}) \subseteq \gamma_P(p)$$

Démonstration. Pour tout ensemble de prédicats p, la définition de γ_P peut s'écrire comme

$$\gamma_P(p) = \{s \in S \mid \forall p_i \in p,\ \mathscr{P}(s, p_i)\}$$

où $\mathscr{P}$ est une propriété validant les prédicats de p_i par rapport aux valeurs des variables de $\mathbb{V}$ dans l'état s.

Nous montrons ainsi que

$$\begin{aligned}
&\gamma_P(p \cup \{p_0\}) \subseteq \gamma_P(p) \\
\Leftrightarrow\ &\{s \in S \mid \forall p_i \in p \cup \{p_0\},\ \mathscr{P}(s, p_i)\} \subseteq \{s \in S \mid \forall p_i \in p,\ \mathscr{P}(s, p_i)\} \\
\Leftrightarrow\ &\forall s \in S,\ \forall p_i \in p \cup \{p_0\},\ \mathscr{P}(s, p_i) \Longrightarrow \forall p_i \in p,\ \mathscr{P}(s, p_i)
\end{aligned}$$

Or, $p \cup \{p_0\} \supseteq p$. □

Considérons l'instruction sémantique `scratch` r_1. Si l'état avant son interprétation contient les prédicats

$$\{r_0 = r_4,\ r_1 + 1 \neq t_1\}$$

l'information du deuxième prédicat n'a plus de sens après `scratch` r_1, et nous ne pouvons pas faire mieux que le supprimer, et ne garder que $\{r_0 = r_4\}$. Supposons maintenant que l'état avant l'interprétation de l'instruction `scratch` r_1 contienne :

$$\{r_0 = r_1,\ r_1 + 1 \neq t_1\}$$

Nous pouvons alors faire mieux, et transformer cet ensemble de prédicats en

$$\{r_0 + 1 \neq t_1\}$$

Définition 3.20. Soit l'*ensemble des variables* $\mathscr{U}_{\mathbb{E}}$ utilisées par une expression et l'*ensemble des variables* $\mathscr{U}_P$ utilisées par un prédicat, les sous-ensembles de $\mathbb{V}^\sharp$ définis par :

$$\forall p = (e_1, \psi, e_2) \in P, \qquad \mathscr{U}_P(p) \coloneqq \mathscr{U}_{\mathbb{E}}(e_1) \cup \mathscr{U}_{\mathbb{E}}(e_2)$$

$$\forall e \in \mathbb{E}, \qquad \mathscr{U}_{\mathbb{E}}(e) \coloneqq \begin{cases} \emptyset & \text{si } e \in C^\sharp \\ \{e\} & \text{si } e \in \mathbb{V}^\sharp \\ \mathscr{U}_{\mathbb{E}}(e_1) \cup \mathscr{U}_{\mathbb{E}}(e_2) & \text{si } e = (e_1, \phi, e_2) \end{cases}$$

Nous définissons alors la *fonction d'invalidation* d'une variable sur un ensemble de prédicats :

$$\iota_d(p) \coloneqq \begin{cases} \{p_i[e/d] \mid p_i \in p\} & \text{si } \exists a \in \mathbb{V}^\sharp, \exists e \in \mathbb{E}, \\ & (d = e) \in p \vee (e = d) \in p \\ \{p_i \in p \mid d \notin \mathscr{U}_P(p_i)\} & \text{sinon} \end{cases}$$

où $p_i[e/d]$ dénote le prédicat $p_i \in P$ où e remplace toutes les occurrences de d.

Remarque. Cette fonction d'invalidation n'est pas déterministe (plusieurs expressions e différentes peuvent correspondre), et peut être améliorée en utilisant l'associativité des relations d'ordre ($<$ et $\leq$), ou en recherchant des prédicats de la forme $(d = e)$ *impliqués* par p, plutôt que directement inclus.

Pour l'ensemble des instructions sémantiques, notre stratégie sera de ne pas rajouter des informations constantes (du type $x = 4$, $x \in \mathbb{V}^\sharp$) dans les prédicats, redondantes avec l'application de $\vec{S} = \mathbb{V}^\sharp \to C^\sharp$ et plus difficilement exploitables, les prédicats étant plus librement structurés. Nous distinguerons donc les cas où les paramètres de l'instruction (opérandes en lecture) sont des constantes connues.

Nous commençons par définir l'interprétation des instructions de base `set`, `seti` et `scratch`. $\forall d, a \in \mathbb{V}^\sharp$ tq $d \neq a$:

$$\widetilde{\mathbb{I}}_P[\texttt{set}\ d, d](\vec{s}, p) \coloneqq p$$

$$\widetilde{\mathbb{I}}_P[\texttt{set}\ d, a](\vec{s}, p) \coloneqq \begin{cases} \iota_d(p) \cup \{d = a\} & \text{si } \vec{s}(a) = \top \\ \iota_d(p) & \text{sinon} \end{cases}$$

$$\widetilde{\mathbb{I}}_P[\texttt{seti}\ d, k](\vec{s}, p) \coloneqq \iota_d(p)$$

$$\widetilde{\mathbb{I}}_P[\texttt{scratch}\ d](\vec{s}, p) \coloneqq \iota_d(p)$$

Pour le reste des instructions, nous chercherons, lorsque possible, à mettre à jour les prédicats en remplaçant toute occurrence de variable modifiée par cette variable à laquelle on applique l'opération inverse de l'instruction interprétée.

Pour l'interprétation d'instructions arithmétiques à deux paramètres (en lecture), du type $inst_\phi\ d, a, b$, où d est modifié en fonction de a et b, nous distinguons typiquement trois cas :

(a) $d \neq a$ et $d \neq b$: on invalide les prédicats affectés par d et

(i) si la valeur d'au moins une des deux opérandes a, b est inconnue ($\vec{s}(a) = \top \vee \vec{s}(b) = \top$) : on rajoute un prédicat du type $d = a \,\phi\, b$;

(ii) si a et b sont des constantes connues ($\vec{s}(a) \neq \top \wedge \vec{s}(b) \neq \top$) : on ne fait rien, cette information sera représentée dans $\vec{s}$.

(b) $d = a = b$:

(i) si la fonction $f \coloneqq (d \mapsto d \,\phi\, d) \in C^\sharp \to C^\sharp$ est bijective, appliquer le remplacement $[d/(d \,\phi\, d)^{-1}]$ sur les prédicats, où $(d \,\phi\, d)^{-1}$ est l'expression représentant la fonction f^{-1} ;

(ii) si la fonction $f \coloneqq (d \mapsto d \,\phi\, d) \in C^\sharp \to C^\sharp$ n'est pas bijective, on invalide les prédicats affectés par d.

(c) $d \neq a$ et $d = b$ (et de même pour $d = a$ et $d \neq b$) :

(i) si la fonction $f \coloneqq (d \mapsto a \,\phi\, d) \in C^\sharp \to C^\sharp$ est bijective pour tout a, appliquer le remplacement $[d/(d \,\phi\, a)^{-1}]$ sur les prédicats, où $(d \,\phi\, d)^{-1}$ est l'expression représentant la fonction f^{-1} ;

(ii) si la fonction $f \coloneqq (d \mapsto a \,\phi\, d) \in C^\sharp \to C^\sharp$ n'est pas bijective pour tout a, on invalide les prédicats affectés par d : on ne peut rien faire.

Des prédicats "collatéraux" peuvent parfois être rajoutées indépendamment : par exemple, si l'instruction opère $d \mapsto d + d$, on sait que le résultat sera toujours pair et il est intéressant de garder cette information en rajoutant un prédicat $d \bmod 2 = 0$.

Soit $j_{\vec{s}}(v) \coloneqq \begin{cases} \vec{s}(v) & \text{si } \vec{s}(v) \neq \top \\ v & \text{si } \vec{s}(v) = \top \end{cases}$ $\forall \vec{s} \in \vec{S}, \forall v \in \mathbb{V}^\sharp$ la fonction faisant correspondre à une variable v une expression la caractérisant le mieux (sa valeur constante si elle est connue). Alors pour toutes variables $d, a, b \in \mathbb{V}^\sharp$ telles que $d \neq a$ et $d \neq b$,

$$\widetilde{\mathbb{I}}_P[\texttt{add}\ d,a,b](\vec{s},p) \coloneqq \begin{cases} \iota_d(p) & \text{si } \vec{s}(a) \neq \top \wedge \vec{s}(b) \neq \top \\ \iota_d(p) \cup \{d = j_{\vec{s}}(a) + j_{\vec{s}}(b)\} & \text{sinon} \end{cases}$$

$$\widetilde{\mathbb{I}}_P[\texttt{add}\ d,d,d](\vec{s},p) \coloneqq \begin{cases} \iota_d(p) & \text{si } \vec{s}(d) \neq \top \\ p[(d\ /\ 2)\ /\ d] \cup \{d \bmod 2 = 0\} & \text{sinon} \end{cases}$$

$$\widetilde{\mathbb{I}}_P[\texttt{add}\ d,d,b](\vec{s},p) \coloneqq \begin{cases} \iota_d(p) & \text{si } \vec{s}(d) \neq \top \wedge \vec{s}(b) \neq \top \\ p[(d - j_{\vec{s}}(b))\ /\ d] & \text{si } \vec{s}(d) = \top \\ p \cup \{d = \vec{s}(d) + b\} & \text{si } \vec{s}(d) \neq \top \wedge \vec{s}(b) = \top \end{cases}$$

$$\widetilde{\mathbb{I}}_P[\texttt{add}\ d,a,d] \coloneqq \widetilde{\mathbb{I}}_P[\texttt{add}\ d,d,a]$$

$$\widetilde{\mathbb{I}}_P[\texttt{sub}\ d,a,b](\vec{s},p) \coloneqq \begin{cases} \iota_d(p) & \text{si } \vec{s}(a) \neq \top \wedge \vec{s}(b) \neq \top \\ \iota_d(p) & \text{si } a = b \\ \iota_d(p) \cup \{d = j_{\vec{s}}(a) - j_{\vec{s}}(b)\} & \text{sinon} \end{cases}$$

$$\widetilde{\mathbb{I}}_P[\texttt{sub}\ d,d,d](\vec{s},p) \coloneqq \iota_d(p)$$

$$\widetilde{\mathbb{I}}_P[\texttt{sub}\ d,d,b](\vec{s},p) \coloneqq \begin{cases} \iota_d(p) & \text{si } \vec{s}(d) \neq \top \wedge \vec{s}(b) \neq \top \\ p[(d + j_{\vec{s}}(b))\ /\ d] & \text{si } \vec{s}(d) = \top \\ p \cup \{d = \vec{s}(d) - b\} & \text{si } \vec{s}(d) \neq \top \wedge \vec{s}(b) = \top \end{cases}$$

$$\widetilde{\mathbb{I}}_P[\texttt{sub}\ d,a,d](\vec{s},p) \coloneqq \begin{cases} \iota_d(p) & \text{si } \vec{s}(d) \neq \top \wedge \vec{s}(a) \neq \top \\ p[(j_{\vec{s}}(a) - d)\ /\ d] & \text{si } \vec{s}(d) = \top \\ p \cup \{d = a - \vec{s}(d)\} & \vec{s}(d) \neq \top \wedge \vec{s}(a) = \top \end{cases}$$

$$\widetilde{\mathbb{I}}_P[\texttt{mul}\ d,a,b](\vec{s},p) \coloneqq \begin{cases} \iota_d(p) & \text{si } \vec{s}(a) \neq \top \wedge \vec{s}(b) \neq \top \\ \iota_d(p) \cup \{d = j_{\vec{s}}(a) \times j_{\vec{s}}(b)\} & \text{sinon} \end{cases}$$

$$\widetilde{\mathbb{I}}_P[\texttt{mul}\ d,d,d](\vec{s},p) \coloneqq \iota_d(p)$$

$$\widetilde{\mathbb{I}}_P[\texttt{mul}\ d,d,b](\vec{s},p) \coloneqq \begin{cases} \iota_d(p) & \text{si } \begin{array}{l}\vec{s}(d) \neq \top \wedge \vec{s}(b) \neq \top \text{ ou} \\ \vec{s}(d) = \top \wedge \vec{s}(b) = \top\end{array} \\ \iota_d(p) \cup \{d \bmod 2^k = 0\} & \text{si } \vec{s}(d) = \top \wedge \vec{s}(b) \neq \top \wedge k = \\ & \max\{x \in \mathbb{N} \mid \vec{s}(b) \bmod 2^x = 0\} \\ p \cup \{d = \vec{s}(d) \times b\} & \text{si } \vec{s}(d) \neq \top \wedge \vec{s}(d) = \top \end{cases}$$

$$\widetilde{\mathbb{I}}_P[\texttt{mul}\ d,a,d](\vec{s},p) \coloneqq \widetilde{\mathbb{I}}_P[\texttt{mul}\ d,d,a](\vec{s},p)$$

Remarque. Nous n'ajoutons pas de prédicat $d \geq 0$ pour une instruction $\texttt{mul}\ d,d,d$ ni de prédicat $d \bmod b = 0$ pour une instruction $\texttt{mul}\ d,d,b$, à cause des risques d'*overflow*.

Exemple. Soit l'instruction sémantique $i \coloneqq \texttt{sub}\ r_2, r_0, t_1$. Supposons $\vec{s}(r_0) = \top$ et $\vec{s}(t_1) = 4$. Alors, $\widetilde{\mathbb{I}}_P(\vec{s}, \{r_0 \geq 0,\ r_1 > r_2 + 1\}) = \{r_0 \geq 0,\ r_2 = r_0 - 4\}$, le deuxième prédicat du résultat étant obtenu à partir de la formule $d = j_{\vec{s}}(a) - j_{\vec{s}}(b)$.

Le processus est légèrement optimisé en considérant que l'appel à ι_d est inutile lorsque d est une constante connue, puisque l'interprétation par prédicats est conçue de manière à éviter ce type de redondance, c'est-à-dire qu'elle maintient la propriété $\forall d \in \mathbb{V}^\sharp,\ \vec{s}(d) \neq \top \implies \forall p_i \in p,\ d \notin \mathscr{U}_P(p_i)$ pour tout état $(\vec{s}, p)$: l'information de la valeur constante de d aura été représentée dans $\vec{s}$.

Nous pouvons faire la preuve générale de la validité d'une telle technique d'interprétation d'instructions sur les prédicats par substitution de la fonction inverse. Ce principe est énoncé par le Théorème 3.1.

Théorème 3.1. *Soit $i \in I_{sem}$ une instruction sémantique. Si*

(1) il existe une fonction $\mathbb{I}[i]^{-1}$, inverse de la fonction d'interprétation concrète :

$$\forall s \in S,\ \mathbb{I}[i]^{-1}(\mathbb{I}[i](s)) = s$$

(2) il existe $v \in \mathbb{V}^\sharp$, unique variable modifiée par i :

$$\forall s \in S, \forall v' \neq v, \mathbb{I}[i](s)(v) = s(v)$$

(3) il existe une expression $e_i^{-1} \in \mathbb{E}$ modélisant l'effet de $\mathbb{I}[i]^{-1}$ sur v :

$$\forall s \in S, \gamma_{\mathbb{E}}(s, e_i^{-1}) = \mathbb{I}[i]^{-1}(s)(v)$$

Alors, le remplacement de v par e_i^{-1} dans les prédicats est une abstraction valide et précise (sans approximation) de $\mathbb{I}[i]$, c'est-à-dire

$$\forall p \in \mathcal{P}(P),\ \gamma_P(p[v \mapsto e_i^{-1}]) = \mathbb{I}[i](\gamma_P(p))$$

La preuve complète de ce théorème est détaillée dans l'Annexe C.

3.4.3 Conclusion

L'abstraction $\widetilde{S}$ se révèlera suffisamment précise et efficace pour détecter un nombre significatif de chemins infaisables [91], comme nous le verrons en Section 5.3. Elle souffre toutefois de nombreux défauts.

```
LDR r0, [r0]
```
| `load` $r_0, r_0, \mathbb{N}_{32}$

Figure 3.10 – Chargement dans la variable contenant l'adresse

Premièrement, l'interprétation par prédicats est sévèrement limitée par sa capacité à trouver une opération inverse à appliquer, pour chaque instruction interprétée. Beaucoup d'opérations ne sont pas bijectives, et causent des pertes (partielles ou non) d'information sur les variables affectées. Certaines opérations courantes, comme le chargement dans une variable de la valeur à l'adresse contenue dans cette même variable (Figure 3.10), sont impossibles à modéliser avec le domaine des prédicats. Les approximations engendrées peuvent rendre les états abstraits trop imprécis pour détecter des états impossibles, et donc, des chemins infaisables.

Deuxièmement, l'interprétation est coûteuse. Les prédicats sont des objets construits inductivement, à taille variable, qui doivent être parcourus de nombreuses fois à chaque interprétation d'une instruction. Même pour des opérations simples à traduire, il faut déterminer les prédicats à invalider ou ceux qui nécessitent une substitution de

variable. L'invalidation doit à nouveau parcourir l'ensemble des prédicats pour chercher un opérande égal à la variable supprimée, afin de préserver l'information. L'abstraction par prédicats s'accompagne d'algorithmes d'interprétation (et de manipulation des prédicats) à complexité quadratique ($O(n^2)$), du fait que le traitement pour chaque prédicat peut demander le parcours de tous les autres prédicats. La taille des états devient rapidement un problème pour l'analyse de programmes de taille, d'autant que les prédicats tels que ceux décrivant des constantes stockées en mémoire (car trop volumineuses pour être encodées dans les instructions) ne seront jamais supprimés, sauf en utilisant une analyse de vivacité (présentée dans [3]).

Enfin, certaines parties des programmes – les boucles et les fonctions – doivent être analysées plusieurs fois. Afin d'éviter de répéter le travail d'analyse, il est utile de pouvoir représenter l'état de la machine en fin d'exécution d'une région du programme, en fonction de l'état en entrée. Les prédicats de P n'expriment que des propriétés par rapport aux valeurs courantes de chaque variable, et ne permettent donc pas de morceler l'analyse en plusieurs parties composables.

Cette absence de modularité engendre aussi un problème de contextualisation (Figure 3.11). Si l'on découvre une propriété (chemin infaisable) dans une fonction g appelée à partir d'une fonction f, il est difficile, voire impossible de déterminer si cette propriété est particulière à l'appel depuis f, ou si elle est générale à tout appel de g. Pour le savoir, il faudrait analyser g deux fois, avec les informations issues du contexte d'appel (pour chercher une propriété locale), et sans (pour chercher une propriété plus forte). La Figure 3.11 illustre ce problème de recherche d'un contexte minimal pour une propriété découverte sur un arc dans le corps de la fonction g.

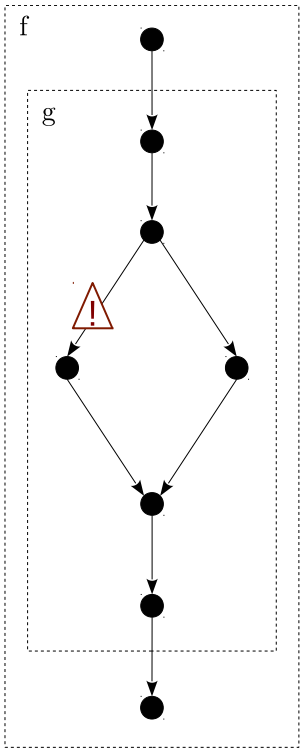
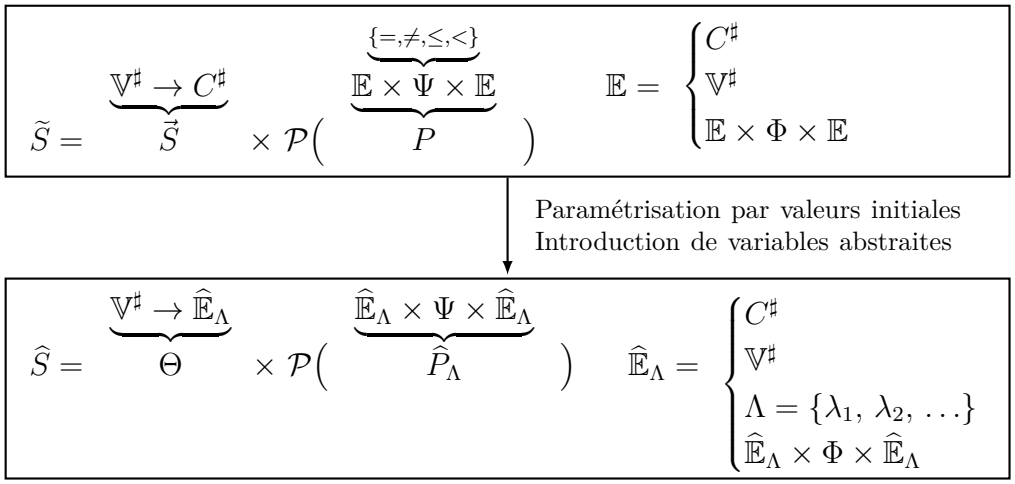


Figure 3.11 – Problème de la contextualisation des propriétés



3.5 Vers une abstraction paramétrée et composable

3.5.1 Le domaine $\widehat{S}_0$

Nous avons jusqu'ici représenté les propriétés des variables du programme dans les états abstraits sous la forme de relations entre elles, valides en un point du programme particulier représenté par l'état. Dans un effort de rendre l'analyse du programme plus modulaire, tout en améliorant sa précision et sa complexité, nous allons désormais exprimer chaque variable du programme en fonction des valeurs initiales de ces variables en entrée du (sous-)programme – des paramètres donc – plutôt que de chercher à maintenir des relations entre leurs valeurs courantes.

3.5.1.1 Définition

Nous noterons $\widehat{\mathbb{E}}$ l'ensemble des expressions $\mathbb{E}$ où l'ensemble des valeurs initiales des variables $\widehat{\mathbb{V}}$ ($= \widehat{Var} \times \widehat{Mem}$) remplace $\mathbb{V}^\sharp$. De manière similaire, $\widehat{P}$ sera l'ensemble des prédicats modifié pour utiliser $\widehat{\mathbb{E}}$. Il s'agit là purement d'une convention : $\widehat{\mathbb{V}} \cong \mathbb{V}^\sharp$, $\widehat{\mathbb{E}} \cong \mathbb{E}$ et $\widehat{P} \cong P$, la différence tient dans le fait que les variables dans $\widehat{\mathbb{E}}$ et $\widehat{P}$ représentent maintenant leurs valeurs avant la partie de code analysée. Par convention, les variables de $\widehat{\mathbb{V}}$ seront différenciées de celles de $\mathbb{V}^\sharp$ par un * : par exemple, $r_0^* \in \widehat{\mathbb{V}}$ représente la valeur de $r_0 \in \mathbb{V}$ au début de l'analyse.

Nous définissons en premier lieu $\widehat{S}_0$:

Définition 3.21. Soit Θ_0 l'ensemble des applications de variables vers des expressions, fonctions de leurs valeurs initiales :

$$\Theta_0 \coloneqq \mathbb{V}^\sharp \to \widehat{\mathbb{E}}$$

Les éléments de Θ_0 seront appelés des *tables de variables*. L'état abstrait paramétré $\widehat{S}_0$ est défini comme :

$$\begin{aligned}\widehat{S}_0 &\coloneqq \Theta_0 \times \mathcal{P}(\widehat{P}) \\ &= (\mathbb{V}^\sharp \to \widehat{\mathbb{E}}) \times \mathcal{P}(\widehat{P})\end{aligned}$$

3.5.1.2 Vue fonctionnelle

Cette abstraction est paramétrée par un ensemble de valeurs initiales associées aux variables de $\mathbb{V}^\sharp$. Nous pouvons donc voir $\widehat{S}$ comme une fonction de $(\mathbb{V}^\sharp \to {\mathbb{Z}_{32}}^\sharp) \to \widetilde{S}$, c'est-à-dire $S \to \widetilde{S}$, où les variables de $\widehat{\mathbb{V}}$ sont simplement remplacées par la valeur qu'elles représentent.

Exemple. Ainsi, si $\theta \in \Theta$ est une table de variables telle que

$$\theta(r_0) = r_0^*, \qquad \theta(r_1) = \top, \qquad \theta(r_2) = r_1^* \times r_2^*$$

nous pouvons définir la fonction associée à θ qui, prenant en paramètres une valuation des valeurs initiales s telle que

$$s(r_0) = 10, \qquad s(r_1) = 20, \qquad s(r_2) = 30$$

donnerait l'état

$$\widetilde{s}(r_0) = 10, \qquad \widetilde{s}(r_1) = \top, \qquad \widetilde{s}(r_2) = 20 \times 30$$

Le même raisonnement s'applique pour la transformation de variables dans les prédicats de $\widehat{P}$: un ensemble de prédicats $\{r_0^* = 10,\ r_2^* - r_1^* \geq 5\}$ deviendrait $\{10 = 10,\ 30 - 20 \geq 5\}$.

Comme l'état obtenu après application d'un ensemble de valeurs initiales ne contient

pas de variables, nous pouvons évaluer dans l'état de $\widetilde{S}$ obtenu toutes les expressions $\mathbb{E}$ vers une constante de ${\mathbb{Z}_{32}}^\sharp$ ou $\top$, et l'ensemble des prédicats de P vers $\top$ ou $\bot$. Par exemple, dans le cas de l'exemple ci-dessus, $\widetilde{s}(r_2)$ deviendrait simplement $20 \times 30 = 600$. Plutôt que de définir une fonction associée à un état de $\widehat{S}_0$ sur

$$S \to \widetilde{S} = \vec{S} \cup \mathcal{P}(P)$$

nous pouvons donc la définir sur

$$S \to (\mathbb{V}^\sharp \to C^\sharp) \times \{\top, \bot\}$$

Des prédicats tautologiques ($\top$) ne contiennent pas d'information, et des prédicats indiquant une contradiction ($\bot$) représentent un état impossible $\bot$. Or, $S^\sharp = (\mathbb{V}^\sharp \to C^\sharp) \cup \{\bot\}$; nous pouvons donc simplement définir la fonction associée sur

$$S \to S^\sharp$$

Nous introduisons pour cela le foncteur $\mathcal{F}_0$:

Définition 3.22. Pour tout état abstrait de $\widehat{S}_0$, nous définissons le foncteur donnant sa *fonction de valuation* associée

$$\mathcal{F}_0 : \widehat{S}_0 \longrightarrow (S \to S^\sharp)$$

$$(\theta, p) \longmapsto s \mapsto \begin{cases} v \mapsto \begin{cases} c \text{ tq } \{c\} = \gamma_{\mathbb{E}}(s, \theta(v)) & \text{si } \theta(v) \neq \top \\ \top & \text{si } \theta(v) = \top \end{cases} & \text{si } \gamma_P(p) \neq \emptyset \\ \bot & \text{si } \gamma_P(p) = \emptyset \end{cases}$$

Pour rappel, $\gamma_{\mathbb{E}}$ est la fonction de valuation d'une expression d'après un état concret s, donnant sa valeur constante si l'expression en paramètre est dénuée de $\top$, et ${\mathbb{Z}_{32}}^\sharp$ sinon.

La définition de $\mathcal{F}_0$ donne ainsi, pour tout ensemble de prédicats p satisfiable, et pour tout θ, s, v,

- $\top$ si l'expression à évaluer contient un $\top$, c'est-à-dire si

$$\top \in \mathscr{U}_{\mathbb{E}}(\theta(v))$$

- Le résultat de l'expression à valeur où chaque variable est remplacée par sa valeur dans s, que nous donne $\gamma_{\mathbb{E}}(s, \theta(v))$ sous la forme d'un singleton.

Exemple. Le foncteur $\mathcal{F}_0$ appliqué à l'exemple ci-dessus donne, pour un ensemble de prédicats p satisfiable,

$$\begin{aligned}\mathcal{F}_0(\theta, p)(s)(r_0) &= 10\\ \mathcal{F}_0(\theta, p)(s)(r_1) &= \top\\ \mathcal{F}_0(\theta, p)(s)(r_2) &= 600\end{aligned}$$

3.5.1.3 Concrétisation

Cette vue fonctionnelle du domaine $\widehat{S}_0$ rend la définition d'une fonction de concrétisation très simple :

$$\begin{aligned}\gamma'_{\widehat{S}_0} : \widehat{S}_0 &\longrightarrow \mathcal{P}(S)\\ \hat{s} &\longmapsto \gamma_{S^\sharp}(\mathcal{F}_0(\hat{s})(\top))\end{aligned}$$

Il suffit d'appliquer le foncteur à $\top = (v \mapsto \top) \in S$ et de concrétiser l'état de $S^\sharp$ obtenu selon les règles habituelles pour obtenir l'ensemble des états concrets de S possibles.

Ce type de concrétisation manque toutefois de précision pour s'appliquer à des états paramétriques comme ceux de $\widehat{S}_0$. En effet, les états de $\widehat{S}_0$ contiennent plus que de simples relations entre valeurs courantes de variables [10], ils contiennent des relations entre les valeurs initiales et valeurs courantes de celles-ci. Conserver ces relations lors de la concrétisation vers $\mathcal{P}(S)$ sera crucial pour la validation de l'interprétation par états paramétriques. Nous définissons en Annexe E une concrétisation plus précise, concrétisant un état abstrait de $\widehat{S}_0$ vers une *fonction* de $S \mapsto \mathcal{P}(S)$, plutôt qu'un ensemble d'état concrets de $\mathcal{P}(S)$.

10. Certaines relations entre variables sont de plus perdues lors de la concrétisation vers $\mathcal{P}(S)$, par exemple une égalité tacite entre r_0 et r_1 issue d'un θ tel que $\theta(r_0) = \theta(r_1)$ $(= r_2^* + 1$ par exemple).

3.5.1.4 Notations

Afin de faciliter la lecture, nous représenterons parfois les tables de variables dans Θ comme pseudo-prédicats. Ainsi, nous noterons $x = x^* + 1$ si la variable x (registre ou cellule mémoire) a été incrémentée de un depuis l'entrée de la région du programme analysée, x^* représentant la valeur de x au début de celle-ci. Pour $\theta \in \Theta$, $\theta(x) = x^* + 1$ serait ainsi une notation équivalente à $(x = x^* + 1) \in \theta$. L'opérande de gauche de ces pseudo-prédicats sera donc toujours une variable de $\mathbb{V}^\sharp$.

```
int x, y, z;
void f() {
  x = y + 1;
  y = 0;
  z = z << x;
  /* ŝ_f */
}
```

(a) La fonction `f`

$\hat{s}_f$		
θ	x	$y^* + 1$
	y	0
	z	$\top$
	...	

(b) Représentation de $\hat{s}_f$

Figure 3.12 – Notation d'états abstraits paramétrés

Sur la Figure 3.12, l'état $\hat{s}_f$ représente l'effet de l'exécution de la fonction `f` de la Figure 3.12a, que nous obtenons à la fin de son analyse. La Figure 3.12b montre une autre représentation, schématique, de l'état abstrait paramétré $\hat{s}_f$, donnant pour toute variable modifiée par `f` sa valeur dans θ. On pourrait également écrire

$$\{x = y^* + 1,\ y = 0,\ z = \top\}$$

3.5.2 Interprétation abstraite sur $\widehat{S}_0$

Nous définissons maintenant la fonction d'interprétation $\hat{\mathbb{I}}_0 : I_{sem} \to \widehat{S}_0 \to \widehat{S}_0$ pour chaque instruction sémantique, avec en premier quelques opérations de base :

$$\hat{\mathbb{I}}_0[\texttt{seti}\ d, k](\theta, p) \coloneqq (\theta[d \mapsto k], p)$$
$$\hat{\mathbb{I}}_0[\texttt{set}\ d, a](\theta, p) \coloneqq (\theta[d \mapsto \theta(a)], p)$$

L'ensemble des prédicats $p \in \mathcal{P}(P^*)$ n'est pas modifié, puisque ces prédicats portent sur les valeurs initiales des variables ($\widehat{\mathbb{V}}$) et les propriétés qu'ils décrivent sont donc, une fois découvertes, vraies pour le reste du programme quoi qu'il arrive. Par exemple, un prédicat $(r_2^* = 0)$ n'est pas affecté par une instruction `seti` $r_2, 4$; le changement

de valeur de r_2 est représenté par $\theta(r_2) = 4$, mais le prédicat porte sur r_2^*, la valeur initiale de r_2. Il est donc invariant, c'est-à-dire qu'il reste vrai pour l'intégralité de l'interprétation de la région analysée.

L'instruction `scratch` d met simplement la variable d à $\top$ dans la table des variables. Une fois de plus, les prédicats ne sont pas affectés avec cette représentation.

$$\hat{\mathbb{I}}_0[\texttt{scratch}\ d](\theta, p) \coloneqq (\theta[d \mapsto \top], p)$$

Le traitement des opérations arithmétiques est trivial :

$$\begin{aligned}
\hat{\mathbb{I}}_0[\texttt{add}\ d, a, b](\theta, p) &\coloneqq (\theta[d \mapsto \theta(a) + \theta(b)], p) \\
\hat{\mathbb{I}}_0[\texttt{sub}\ d, a, b](\theta, p) &\coloneqq (\theta[d \mapsto \theta(a) - \theta(b)], p) \\
\hat{\mathbb{I}}_0[\texttt{mul}\ d, a, b](\theta, p) &\coloneqq (\theta[d \mapsto \theta(a) \times \theta(b)], p) \\
\hat{\mathbb{I}}_0[\texttt{div}\ d, a, b](\theta, p) &\coloneqq (\theta[d \mapsto \theta(a) \,/\, \theta(b)], p) \\
\hat{\mathbb{I}}_0[\texttt{mod}\ d, a, b](\theta, p) &\coloneqq (\theta[d \mapsto \theta(a) \bmod \theta(b)], p) \\
\hat{\mathbb{I}}_0[\texttt{neg}\ d, a](\theta, p) &\coloneqq (\theta[d \mapsto 0 - \theta(a)], p) \\
\hat{\mathbb{I}}_0[\texttt{cmp}\ d, a, b](\theta, p) &\coloneqq (\theta[d \mapsto \theta(a \sim \theta(b)], p)
\end{aligned}$$

Les instructions d'accès à la mémoire (`load` d, a et `store` d, a) nécessitent de distinguer deux cas selon que l'adresse a est une constante connue ($\theta(a) \in {\mathbb{Z}_{32}}^\sharp$) ou non. Si elle est connue, c'est un simple `set` entre d et $m_{\theta(a)}$, la cellule mémoire à l'adresse $\theta(a)$. Si elle est inconnue, nous utilisons $\top$ pour approcher conservativement :

$$\hat{\mathbb{I}}_0[\texttt{load}\ d, a, t](\theta, p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(m_{\theta(a)})], p) & \text{si } \theta(a) \in {\mathbb{Z}_{32}}^\sharp \\ (\theta[d \mapsto \top], p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}_0[\texttt{store}\ d, a, t](\theta, p) \coloneqq \begin{cases} (\theta[m_{\theta(a)} \mapsto \theta(d)], p) & \text{si } \theta(a) \in {\mathbb{Z}_{32}}^\sharp \\ \left(v \mapsto \begin{cases} \theta(v) & \text{si } v \in \mathit{Var} \\ \top & \text{si } v \in \mathit{Mem}^\sharp \end{cases}, p \right) & \text{sinon} \end{cases}$$

Le cas du `store` est le plus délicat, puisqu'une écriture dans une cellule mémoire inconnue nous fait perdre toute certitude sur le contenu de la mémoire [11], toute cellule

11. Il existe néanmoins de multiples techniques [9, 24, 95] qui permettent de limiter la perte d'information dans ce scénario.

mémoire dans la table θ est donc associée à $\top$ dans le cas où l'évaluation de l'adresse échoue.

Les instructions de décalage de bits ne peuvent être traitées qu'avec un décalage connu (constant), du fait que les expressions de $\widehat{\mathbb{E}}$ ne supportent pas de variables en exposant.

$$\hat{\mathbb{I}}_0[\texttt{shl}\ d, a, b](\theta, p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(d) \times 2^{\theta(b)}], p) & \text{si } \theta(b) \in [\![0, 31]\!] \\ (\theta[d \mapsto \top], p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}_0[\texttt{asr}\ d, a, b](\theta, p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(d)\ /\ 2^{\theta(b)}], p) & \text{si } \theta(b) \in [\![0, 31]\!] \\ (\theta[d \mapsto \top], p) & \text{sinon} \end{cases}$$

Ici, $2^{\theta(b)}$ représente la constante résultant d'un calcul constant, et non une expression.

Les opérations logiques bit à bit, étant donné l'absence d'opérateurs logiques dans $\mathbb{E}$, entraînent une perte d'information sur le registre de destination, à l'exception du cas trivial où l'instruction opère sur des constantes entières, et de l'instruction `not` qui a une fonction arithmétique équivalente ($x \mapsto 1 - x$) pour des variables codées en complément à deux :

$$\hat{\mathbb{I}}_0[\texttt{not}\ d, a](\theta, p) \coloneqq (\theta[d \mapsto 1 - \theta(a)], p)$$

$$\hat{\mathbb{I}}_0[\texttt{and}\ d, a, b](\theta, p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(a)\ \&\ \theta(b)], p) & \text{si } \theta(a) \in \mathbb{Z}_{32} \wedge \theta(b) \in \mathbb{Z}_{32} \\ (\theta[d \mapsto \top], p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}_0[\texttt{or}\ d, a, b](\theta, p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(a)\ |\ \theta(b)], p) & \text{si } \theta(a) \in \mathbb{Z}_{32} \wedge \theta(b) \in \mathbb{Z}_{32} \\ (\theta[d \mapsto \top], p) & \text{sinon} \end{cases}$$

Enfin, l'interprétation de l'instruction sémantique `assume` ajoute simplement la propriété renseignée par l'instruction en question dans l'ensemble des prédicats :

$$\hat{\mathbb{I}}_0[\texttt{assume}\ c, a, b](\theta, p) \coloneqq \begin{cases} (\theta, p \cup \{\theta(a)\ \psi_c\ \theta(b)\}) & \text{si } c \in \{=, \neq, <, \leq\},\ \psi_c \coloneqq c \\ (\theta, p \cup \{\theta(b)\ \psi_c^{-1}\ \theta(a)\}) & \text{sinon, avec } \psi_>^{-1} \coloneqq <,\ \psi_\geq^{-1} \coloneqq \leq \end{cases}$$

3.5.3 Le domaine $\widehat{S}$: Introduction de variables abstraites

```
void f1(int x, y) {
  x = exp(x);
  y = x % 2;
  /* ŝ_f1 */
}
```

(a) La fonction `f1`

$\hat{s}_{f_1}$		
θ	x	$\top$
	y	$\top$
	$\ldots$	

(b) $\hat{s}_{f_1}$ sans introduction de variable

$\hat{s}_{f_1}$		
θ	x	λ_1
	y	$\lambda_1 \bmod 2$
	$\ldots$	

(c) $\hat{s}_{f_1}$ avec introduction de variable

```
void f2(unsigned a[4]) {
  a[0] = a[0] & 0x1110;
  a[1] = a[1] - a[0];
  read(a);
  a[1] = a[1] + a[0];
  /* ŝ_f2 */
}
```

(d) La fonction `f2`

$\hat{s}_{f_2}$		
θ	m_a	$\top$
	m_{a+4}	$\top$
	m_{a+8}	m^*_{a+8}
	m_{a+12}	m^*_{a+12}
	$\ldots$	

(e) $\hat{s}_{f_2}$ sans introduction de variable

$\hat{s}_{f_2}$		
θ	m_a	λ_1
	m_{a+4}	m^*_{a+4}
	m_{a+8}	m^*_{a+8}
	m_{a+12}	m^*_{a+12}
	$\ldots$	
p	$\lambda_1 \leq m^*_a$	
	$\lambda_1 \leq$ `0x1110`	

(f) $\hat{s}_{f_2}$ avec introduction de variable

Figure 3.13 – Remède à la perte des relations entre variables après affectation à $\top$

En contrepartie des multiples avantages que procure une abstraction raisonnant sur les valeurs initiales des variables du programme plutôt que sur leurs valeurs courantes, il arrive cependant que le domaine abstrait $\widehat{S}_0$ échoue à représenter des relations entre variables au fil de l'exécution d'un programme.

Prenons l'exemple de la fonction `f1`, décrite sur la Figure 3.13a. Un lien clair existe entre les variables `x` et `y` en fin de fonction, mais une interprétation en avant classique avec $\widehat{S}_0$ ne le verrait pas : après l'appel à la fonction complexe `exp`, l'état abstrait devient $\{x = \top, y = y^*\}$, et y reçoit $\top \bmod 2 = \top$. Nous finissons ainsi avec l'état $\hat{s}_{f_1}$ représenté en Figure 3.13b.

Si nous travaillions avec le domaine des prédicats $\mathcal{P}(P)$ (ou $\widetilde{S}$), une contrainte $y = x \bmod 2$ aurait été facilement découverte et ajoutée à la conjonction de prédicats. Il s'agit donc là d'un problème propre à notre nouvelle abstraction, auquel nous remédions en introduisant le concept de variable abstraite.

Définition 3.23. L'ensemble des *variables abstraites* est

$$\Lambda \coloneqq \{\lambda_1, \lambda_2, \lambda_3, \ldots\}$$

Ces variables représentent une valeur inconnue (mais fixe) correspondant au résultat de l'exécution d'une instruction qui n'a pas pu être représenté par $\widehat{\mathbb{E}}$.

Nous enrichissons les expressions avec, et définissons ainsi les états abstraits paramétrés $\widehat{S}$, notre représentation finale du programme.

Définition 3.24. Soit $\widehat{\mathbb{E}}_\Lambda$ le domaine des expressions $\widehat{\mathbb{E}}$ enrichi par les variables abstraites de Λ.

$$\widehat{\mathbb{E}}_\Lambda \coloneqq \left| \begin{array}{c} C^\sharp \\ \mathbb{V}^\sharp \\ \Lambda \\ \widehat{\mathbb{E}}_\Lambda \times \Phi \times \widehat{\mathbb{E}}_\Lambda \end{array} \right.$$

Soit les tables de variables $\Theta \coloneqq \mathbb{V}^\sharp \to \widehat{\mathbb{E}}_\Lambda$. Soient les prédicats $\widehat{P}_\Lambda \coloneqq \widehat{\mathbb{E}}_\Lambda \times \Psi \times \widehat{\mathbb{E}}_\Lambda$. On définit alors les états de $\widehat{S}$, plus expressifs que ceux de $\widehat{S}_0$ grâce à l'adjonction de ces variables abstraites :

$$\widehat{S} \coloneqq \Theta \times \mathcal{P}(\widehat{P}_\Lambda)$$

Grâce à cette révision des états abstraits, il est désormais possible de mieux représenter l'effet de l'exécution de la fonction `f1` de la Figure 3.13a. Si nous représentons la valeur retournée par l'appel à `exp` (trop complexe pour être modélisée) par une variable λ_1, nous pouvons indirectement exprimer le lien entre x et y en fin de fonction par $\{x = \lambda_1, y = \lambda_1 \bmod 2\}$. Au même titre que les valeurs initiales des variables de $\widehat{\mathbb{V}}$, les variables abstraites de Λ sont des inconnues de l'analyse.

La Figure 3.13d montre un exemple similaire. À travers l'introduction d'une variable λ_1, nous sommes capables de reconnaître que seul `a[0]` est modifié par la fonction (`read` ne modifie pas le tableau). L'adresse de `a` dans la mémoire sera connue après compilation (nous la notons a), et l'interprétation de `f2` résulte d'abord en $\{m_a =$

$\lambda_1, m_{a+4} = m^*_{a+4} - \lambda_1, m_{a+8} = m^*_{a+8}, m_{a+12} = m^*_{a+12}\}$, puis en $\hat{s}_{f_2}$, représenté sur la Figure 3.13f. L'introduction de λ_1 est également utile pour exprimer des informations partielles comme $\lambda_1 < m^*_a$ et $\lambda_1 < \texttt{0x1110}$: le résultat inconnu du *et* binaire est plus petit que chacun de ses opérandes (non signés).

Une bonne stratégie d'interprétation est donc d'introduire une variable abstraite après chaque instruction dont le résultat est inconnu, pour pouvoir exprimer des propriétés sur ce résultat. Partant de ce constat, nous introduisons la fonction d'invalidation :

Définition 3.25. La fonction d'*invalidation* $\hat{\iota}_v : \Theta \longrightarrow \Theta$ est définie pour toute variable $v \in \mathbb{V}^\sharp$ comme

$$(\hat{\iota}_v)\frac{\Gamma_\Lambda \quad \vdash \theta}{\Gamma_\Lambda \cup \{\lambda_{|\Gamma_\Lambda|+1}\} \quad \vdash \theta[v \mapsto \lambda_{|\Gamma_\Lambda|+1}]}$$

pour un contexte $\Gamma_\Lambda \subset \Lambda$ définissant l'ensemble des variables introduites.

L'algorithme d'introduction de variables dans PathFinder implémente cette technique sans dégrader sérieusement la complexité de l'analyse (entre 3% et 4% du temps d'analyse passé dans ce module d'après quelques expériences sur les benchmarks de Mälardalen), grâce à un mécanisme de suppression des variables inutilisées, et en gérant les invalidations complètes de mémoire (dues à des `store` à une adresse inconnue) par l'introduction retardée de variables λ au fil de l'analyse, lorsqu'une valeur inconnue est chargée de la mémoire.

Nous pouvons maintenant adapter la fonction d'interprétation abstraite pour $\widehat{S}$.

3.5.4 Interprétation abstraite sur $\widehat{S}$

Une première application immédiate de cette technique est l'introduction d'une variable abstraite λ_i à chaque instruction `scratch`, représentant le résultat inconnu :

$$\hat{\mathbb{I}}[\texttt{scratch}\ d](\theta, p) \coloneqq (\hat{\iota}_d(\theta), p)$$

De manière similaire, un chargement à une adresse inconnue entraîne l'introduction d'une variable abstraite représentant la valeur chargée. Selon le même état d'esprit, une écriture sur une variable inconnue devrait maintenant affecter une variable abstraite

différente à chaque cellule mémoire :

$$\hat{\mathbb{I}}[\texttt{load}\ d,a,t](\theta,p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(m_{\theta(a)})],p) & \text{si } \theta(a) \in {\mathbb{Z}_{32}}^\sharp \\ (\hat{\iota}_d(\theta),p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}[\texttt{store}\ d,a,t](\theta,p) \coloneqq \begin{cases} (\theta[m_{\theta(a)} \mapsto \theta(d)],p) & \text{si } \theta(a) \in {\mathbb{Z}_{32}}^\sharp \\ (\underbrace{\hat{\iota}_{v_1} \circ \hat{\iota}_{v_2} \circ \ldots \circ \hat{\iota}_{v_k}}_{(v_1,v_2,\ldots,v_k)=Mem^\sharp}(\theta),p) & \text{sinon} \end{cases}$$

Une méthode moins coûteuse et plus réaliste à appliquer pour gérer une écriture à une adresse inconnue consiste à ne pas assigner de variable abstraite à chaque cellule mémoire, seulement un $\top$, comme selon la définition de $\hat{\mathbb{I}}_0[\texttt{store}\ d,a,t]$. En revanche, nous modifions alors l'interprétation du `load` pour introduire cette variable en retard dans la mémoire lors du chargement d'un $\top$. Ainsi, $\hat{\mathbb{I}}(\texttt{load}\ d,a,t)$ devient

$$(\theta,p) \longmapsto \begin{cases} (\theta[d \mapsto \theta(m_{\theta(a)})],p) & \text{si } \theta(a) \in {\mathbb{Z}_{32}}^\sharp \wedge \theta(m_{\theta(a)}) \neq \top \\ \Big(\big(\hat{\iota}_{m_{\theta(a)}}(\theta)\big)[d \mapsto \theta(m_{\theta(a)})],p\Big) & \text{si } \theta(a) \in {\mathbb{Z}_{32}}^\sharp \wedge \theta(m_{\theta(a)}) = \top \\ (\hat{\iota}_d(\theta),p) & \text{sinon} \end{cases}$$

L'introduction de variables permettant de représenter le résultat de toute opération, elle permet l'adjonction de prédicats sur certaines instructions complexes. Par exemple, sur les instructions de décalage, même avec un facteur inconnu, il peut être utile de noter que le décalage à gauche (`shl`) est croissant, et le décalage à droite (`asr`) décroissant pour des valeurs positives (et vice-versa pour des valeurs négatives) :

$$\hat{\mathbb{I}}[\texttt{shl}\ d,a,b](\theta,p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(d) \times 2^{\theta(b)}],p) & \text{si } \theta(b) \in [\![0,31]\!] \\ (\theta',p \cup \{z\,\theta'(d) \geq z\,\theta(a)\}) & \text{sinon si } \theta(a) \in \mathbb{Z}\text{, avec } \theta' \coloneqq \hat{\iota}_d(\theta) \\ & \qquad\qquad \text{et } z \coloneqq \mathit{signe}(\theta(a)) \\ (\hat{\iota}_d(\theta),p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}[\texttt{asr}\ d,a,b](\theta,p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(d)\ /\ 2^{\theta(b)}],p) & \text{si } \theta(b) \in [\![0,31]\!] \\ (\theta',p \cup \{z\,\theta'(d) \leq z\,\theta(a)\}) & \text{sinon si } \theta(a) \in \mathbb{Z}\text{, avec } \theta' \coloneqq \hat{\iota}_d(\theta) \\ & \qquad\qquad \text{et } z \coloneqq \mathit{signe}(\theta(a)) \\ (\hat{\iota}_d(\theta),p) & \text{sinon} \end{cases}$$

Notons que la sémantique des instructions de décalage `shl` et `asr` n'admet pas de décalage négatif. Nous pouvons faire de même pour les opérateurs binaires *et* et *ou*. Une autre propriété intéressante du *et* binaire est celle-ci :

Propriété 3.1.

$$\forall x \in \mathbb{Z},\ x \,\&\, \underbrace{1\ldots111}_{k}{}_2 = x \bmod 2^{k+1}$$

c'est-à-dire

$$\forall x \in \mathbb{Z}, \forall k \in \mathbb{N},\ x \,\&\, (2^{k+1}-1) = x \bmod 2^{k+1}$$

Cette propriété est souvent utilisée par les compilateurs pour calculer rapidement le reste d'une division par une puissance de deux. Il est donc utile de définir :

$$\hat{\mathbb{I}}[\texttt{and}\ d,a,b](\theta,p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(a)\,\&\,\theta(b)],p) & \text{si } \theta(a)\in\mathbb{Z}_{32} \wedge \theta(b)\in\mathbb{Z}_{32} \\ (\theta', p\cup\{\theta'(d)=\theta(b) \bmod 2^k\}) & \text{si } \exists k\le 31,\ \theta(a)=2^k-1\in\mathbb{Z}_{32} \\ & \text{et avec } \theta' \coloneqq \hat{\iota}_d(\theta) \\ (\theta', p\cup\{\theta'(d)=\theta(a) \bmod 2^k\}) & \text{si } \exists k\le 31,\ \theta(b)=2^k-1\in\mathbb{Z}_{32} \\ & \text{et avec } \theta' \coloneqq \hat{\iota}_d(\theta) \\ \left(\theta', p\cup\left\{\begin{array}{c} z_a\theta'(d)\le z_a\theta(a), \\ z_a\theta'(d)\le z_b\theta(b)\end{array}\right\}\right) & \text{sinon si } (\theta(a),\theta(b))\in\mathbb{Z}^2 \\ & \text{avec } \theta' \coloneqq \hat{\iota}_d(\theta), \text{ et } (z_a,z_b)\coloneqq \\ & (\mathit{signe}(\theta(a)),\mathit{signe}(\theta(b))) \\ (\iota_d(\theta),p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}[\texttt{or}\ d,a,b](\theta,p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(a)\mid\theta(b)],p) & \text{si } \theta(a)\in\mathbb{Z}_{32} \wedge \theta(b)\in\mathbb{Z}_{32} \\ \left(\theta', p\cup\left\{\begin{array}{c} z_a\theta'(d)\ge z_a\theta(a), \\ z_a\theta'(d)\ge z_b\theta(b)\end{array}\right\}\right) & \text{sinon si } (\theta(a),\theta(b))\in\mathbb{Z}^2 \\ & \text{avec } \theta' \coloneqq \hat{\iota}_d(\theta), \text{ et } (z_a,z_b)\coloneqq \\ & (\mathit{signe}(\theta(a)),\mathit{signe}(\theta(b))) \\ (\iota_d(\theta),p) & \text{sinon} \end{cases}$$

La sémantique abstraite complète est donnée par l'Annexe B.

3.5.5 Composition

Nous sommes maintenant en mesure de définir l'opération de composition d'états. Chaque état de $\widehat{S}$ représentant l'effet d'une séquence d'instructions sémantiques, correspondant aux instructions de chaque bloc de base successivement emprunté sur un chemin, il est possible de combiner plusieurs états paramétrés. Ainsi, si deux états $\hat{s}, \hat{s}' \in \widehat{S}$ représentent respectivement l'exécution de successions d'instructions $i_1.i_2...i_n$ et $i'_1.i'_2...i'_m$ sur des chemins π et π' consécutifs, alors l'état composé $\hat{s}' \circ \hat{s}$ devra représenter l'exécution de $i_1.i_2...i_n.i'_1.i'_2...i'_m$ sur le chemin $\pi.\pi'$.

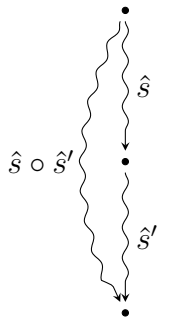


Figure 3.14 – Composition de deux chemins consécutifs

Définition 3.26. L'opérateur de *composition* d'états $\circ$ transforme deux états $\hat{s}, \hat{s}' \in \widehat{S}$ en un état $\hat{s}' \circ \hat{s}$ qui représente l'application successive de $\hat{s}$ puis de $\hat{s}'$. Ainsi, si $\hat{s} = (\theta, p)$, $\hat{s}' = (\theta', p')$, alors

$$(\circ)\ \frac{\Gamma'_\Lambda \vdash \hat{s}' \quad \Gamma_\Lambda \vdash \hat{s}}{\Gamma'_\Lambda \cup \Gamma_\Lambda \vdash \hat{s}' \circ \hat{s}}$$

$$\hat{s}' \circ \hat{s} \coloneqq \Big(v \mapsto \sigma_{\hat{s}}(\theta'(v)),\ \{(\sigma_{\hat{s}}(e_1)\ \psi\ \sigma_{\hat{s}}(e_2)) \mid (e_1\ \psi\ e_2) \in p'\} \cup p\Big)$$

où $\sigma_{\hat{s}}$, la fonction d'application d'un état $\hat{s} = (\theta, p)$ à une expression e est définie comme :

$$\sigma_{\hat{s}}(e) \coloneqq \begin{cases} k & \text{si } e = k,\ k \in \mathbb{Z}_{32} \\ k + \theta(sp) & \text{si } e = \mathrm{SP} + k,\ k \in \mathbb{Z}_{32} \\ \top & \text{si } e = \top \\ \theta(v) & \text{si } e = v,\ v \in \mathbb{V}^\sharp \\ \lambda_i & \text{si } e = \lambda_i,\ \lambda_i \in \Lambda \\ \sigma_{\hat{s}}(e_1)\ \psi\ \sigma_{\hat{s}}(e_2) & \text{si } e = (e_1\ \psi\ e_2),\ e_1, e_2 \in \widehat{\mathbb{E}}_\Lambda, \psi \in \Psi \end{cases}$$

où $sp \in \mathit{Reg}$ désigne le registre pointeur de pile pour l'architecture considérée.

L'opérateur $\circ$ décrit dans la Définition 3.26 effectue en réalité une simple opération, qui, pour appliquer $\hat{s}'$ à $\hat{s}$,

1. remplace dans toutes les expressions de $\hat{s}'$ les variables (qui représentent leur valeur au début de l'analyse) par la valeur de celles-ci d'après $\hat{s}$;
2. effectue dans $\hat{s}'$ une translation du pointeur de pile par rapport aux changements qu'il a subi dans $\hat{s}$;
3. y rajoute les prédicats inchangés de $\hat{s}$.

Intuitivement, les expressions du résultat obtenu, $\hat{s}' \circ \hat{s}$, ainsi que la constante SP, sont maintenant relatives aux valeurs des variables de la machine au début de l'analyse qui a produit $\hat{s}$.

```
int x, y, z;
void g() {
  y = z;
  z = -1;
  /* ŝ_g */
}
void f() {
  x = y + 1;
  y = 0;
  z = z * z;
  /* ŝ */
  g();
  /* ŝ_g ∘ ŝ */
}
```

(a) Le programme

$\hat{s}_g$		
θ	x	x^*
	y	z^*
	z	-1
	$\ldots$	

$\hat{s}$		
θ	x	$y^* + 1$
	y	0
	z	λ_1
	$\ldots$	
p	$W\lambda_1 \geq 0$	

(b) Interprétation séparée des fonctions du programme

$\hat{s}_f = \hat{s}_g \circ \hat{s}$		
θ	x	$y^* + 1$
	y	λ_1
	z	-1
	$\ldots$	
p	$\lambda_1 \geq 0$	

(c) Composition des états

Figure 3.15 – Interprétation d'un appel de fonction par composition d'états

La Figure 3.15 illustre la composition d'états en l'appliquant sur le programme de la Figure 3.15a. La fonction `g` est d'abord interprétée séparément, puis le corps de `f`, jusqu'à l'appel à `g`, résultant en $\hat{s}_g$ et $\hat{s}$ respectivement (Figure 3.15b). La composition de $\hat{s}_g$ et $\hat{s}$ représente l'exécution de la fonction `f`, et équivaut à un état $\hat{s}_f$ qui aurait parcouru l'intégralité de l'exécution de la fonction (en passant par `g`).

Nous définissons également l'état identité :

Définition 3.27. L'état abstrait *identité*, noté $1_{\widehat{S}}$, est défini par le couple d'une liste de prédicats vides et de la table de variables 1_Θ, l'automorphisme identité sur $\mathbb{V}^\sharp$:

$$1_{\widehat{S}} \coloneqq (1_\Theta, \emptyset) = (v \mapsto v^*, \emptyset)$$

Cet état est par définition l'élément neutre de la composition :

$$\forall \hat{s} \in \widehat{S},\ 1_{\widehat{S}} \circ \hat{s} = \hat{s} = \hat{s} \circ 1_{\widehat{S}}$$

Il est aussi, par définition, toujours valide en début d'analyse, du fait qu'il exprime un état du programme où toute variable est inchangée et aucune propriété sur le programme n'a été découverte.

Nous avons, à la suite de plusieurs itérations, défini $\widehat{S}$, une abstraction paramétrée et composable des états possibles de la machine ($\mathcal{P}(S)$), ainsi que les outils pour la manipuler et interpréter un programme avec ($\hat{\mathbb{I}}$). Nous justifions enfin son usage en présentant des techniques de validation de la correction de cette abstraction dans la prochaine section.

3.5.6 Concrétisation de $\widehat{S}$

Nous définissons la fonction de concrétisation d'un état abstrait paramétré $\widehat{S}$ comme une application qui, pour un état de $\widehat{S}$ et un état initial de $\vec{S}$, donne un ensemble d'états de $\mathcal{P}(\vec{S})$, représentation intermédiaire qui peut elle-même être facilement concrétisée vers $\mathcal{P}(S)$. Le tout définit la concrétisation d'une abstraction de $\widehat{S}$ vers un ensemble d'états concrets, paramétrée par un état initial.

Définition 3.28. La fonction de concrétisation $\gamma_{\widehat{S}} : \widehat{S} \to \vec{S} \to \mathcal{P}(\vec{S})$ est ainsi définie :

$$\forall \hat{s} = (\theta, p) \in \widehat{S},\ \forall \vec{s}_0 \in \vec{S},\ \Gamma_\Lambda \vdash \gamma_{\widehat{S}}(\hat{s})(\vec{s}_0) \coloneqq$$
$$\left\{ \vec{s} \in \vec{S} \;\middle|\; \exists L \in (\mathbb{Z}_{32}{}^\sharp)^{|\Gamma_\Lambda|},\ \begin{array}{c} \forall v \in \mathbb{V}^\sharp,\ \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \theta(v)) \wedge \\ \forall (e_1 \,\psi\, e_2) \in p,\ \exists \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \begin{pmatrix} \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, e_1) \\ \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, e_2) \end{pmatrix},\ x_1 \,\psi\, x_2 \end{array} \right\}$$

avec $\gamma_{\widehat{\mathbb{E}}_\Lambda}$, la concrétisation d'une expression e selon un état $\vec{s} \in \vec{S}$ et une valuation $L = \{\kappa_{\lambda_1}, \kappa_{\lambda_2}, \ldots, \kappa_{\lambda_{|\Gamma_\Lambda|}}\} \in (\mathbb{Z}_{32}{}^\sharp)^{|\Gamma_\Lambda|}$ de l'ensemble des variables abstraites définies dans le langage Γ_Λ :

$$\forall e \in \widehat{\mathbb{E}}_\Lambda,\ \Gamma_\Lambda \vdash\ \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}, \{\kappa_{\lambda_1}, \kappa_{\lambda_2}, \ldots, \kappa_{\lambda_{|\Gamma_\Lambda|}}\}, e) \coloneqq$$
$$\begin{cases} \mathbb{Z}_{32} & \text{si } e = \top \\ \{e\} & \text{si } e \in \mathbb{Z}_{32}{}^\sharp \\ \{\vec{s}(e)\} & \text{si } e \in \mathbb{V}^\sharp \\ \{\kappa_e\} & \text{si } e \in \Lambda \\ \bigcup_{a \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}, e_1)} \bigcup_{b \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}, e_2)} \phi(a, b) & \text{si } e = (e_1, \phi, e_2), \text{ avec } e_1, e_2 \in \widehat{\mathbb{E}}_\Lambda,\ \phi \in \Phi \end{cases}$$

Intuition (variables abstraites) Nous fixons pour cette définition un ensemble de valeurs $L = \{\kappa_{\lambda_1}, \kappa_{\lambda_2}, \ldots, \kappa_{\lambda_{|\Gamma_\Lambda|}}\}$ pour tester l'appartenance de chaque $\vec{s}$ à l'ensemble des états constituant le résultat de la concrétisation, afin de s'assurer que l'ensemble des propriétés est conjointement respecté par $\vec{s}$ pour une valeur fixe de chaque λ_i (et pour ne pas accepter, par exemple, un état contenant $0 > \lambda_i$ et $0 < \lambda_i$, valides pour des valeurs de λ_i différentes). Cet ensemble de valeurs L est donné en paramètre à la fonction de concrétisation d'une expression, qui concrétise alors chaque λ_i en un singleton $\{\kappa_{\lambda_i}\}$ pour la valuation L en question.

Cette fonction de concrétisation permet la construction d'une correspondance de Galois et la validation de l'analyse par interprétation sur $\widehat{S}$. Ce processus est décrit dans l'Annexe D.

3.6 Conclusion générale

Nous avons dans ce chapitre traité de la représentation de la machine, de son évolution au fur et à mesure de l'analyse du programme, et de la représentation de celui-ci. Nous avons construit une abstraction adaptée, $\widehat{S}$, et l'avons munie de fonctions d'interprétation abstraite vérifiables grâce à l'établissement d'une correspondance de Galois et de choix de structures appropriés. Cette abstraction raisonnant en fonction de l'état du programme en début d'analyse, elle peut être mise à jour pour refléter l'effet de l'exécution d'instructions sans nécessiter la définition d'une fonction représentant l'effet inverse. Cela simplifie largement le processus d'analyse de programme et limite les pertes de précision dues à l'application de fonctions non bijectives sur les variables. De plus, la composabilité de cette abstraction sera un élément clé dans la construction d'une analyse modulaire qui sera faite au chapitre suivant.

4

Flot du programme

Sommaire

Ce chapitre présente des algorithmes de parcours de graphe de flot de contrôle pour la découverte de propriétés de flot de données. Là où le chapitre précédent portait sur l'analyse des instructions à l'intérieur d'un bloc de base, nous allons désormais traiter de l'analyse de l'ensemble du programme. Dans cette optique, les blocs de bases et arcs des CFG, sont des éléments atomiques du programme.

4.1 Parcours d'un CFG acyclique

Dans cette section, nous détaillons l'algorithme de parcours d'un CFG sans boucles par raffinements successifs. Cet algorithme de parcours en avant doit effectuer une analyse de flot de données : il a pour but d'associer à chaque point du graphe – sommet ou arc – l'ensemble des états possibles.

4.1.1 Parcours d'un graphe acyclique

4.1.1.1 Algorithme de base

Nous commençons par un algorithme classique de parcours de graphe *acyclique*. Celui-ci utilise une *working list* (*wl*), une file d'éléments du graphe à parcourir – ici les sommets, ou blocs de base. Les éléments dans cette file sont uniques (pas de doublon).

Définition 4.1. Pour tout graphe $G = (V, E, \epsilon, \omega)$, et pour tout sommet de ce graphe $v \in V$, nous définissons

- l'ensemble de ses *arcs entrants*, $ins(v) \coloneqq \{(v' \to v) \in E\}$
- l'ensemble de ses *arcs sortants*, $outs(v) \coloneqq \{(v \to v') \in E\}$

De plus, nous notons, pour tout arc $(v_1 \to v_2) \in E$,

$$source(v_1 \to v_2) \coloneqq v_1$$
$$sink(v_1 \to v_2) \coloneqq v_2$$

À chaque itération, nous retirons (*pop*) un élément de la file et y ajoutons la cible (*sink*) de chacun ses arcs sortants (*outs*). Ces itérations se répètent tant que la file, initialisée à l'entrée du graphe ϵ, n'est pas vide. Ce simple, premier algorithme permet de parcourir tous les sommets du graphe.

Parce que nous aurons par la suite [1] besoin de connaître l'ensemble des états du programme possibles au niveau de *sommets* du CFG (et non seulement pour chaque chemin), nous modifions cet algorithme pour y intégrer des points de rendez-vous.

1. Pour les algorithmes d'interprétation de boucles des sections 4.2 et 4.3.

Algorithme 1 : Parcours de graphe acyclique

Données : $G = (V, E, \epsilon, \omega)$, le graphe.
$wl \leftarrow \{\epsilon\}$
tant que $wl \neq \emptyset$ **faire**
  $b \leftarrow pop(wl)$
  pour $e \in outs(b)$ **faire**
    $wl \leftarrow wl \cup \{sink(e)\}$

4.1.1.2 Avec rendez-vous

À la différence de l'Algorithme 1, nous allons désormais retirer à chaque itération l'élément de la file le plus ancien *dont tous les prédécesseurs ont été traités*, c'est-à-dire tous les arcs en entrée (*ins*).

En l'absence de boucles, ce système de rendez-vous ne tombe pas en famine jusqu'à ce que le sommet de fin ω (qui, par définition, n'a pas d'arc sortants) soit atteint par tous les chemins du graphe.

Algorithme 2 : Parcours de graphe acyclique, avec rendez-vous

Données : $G = (V, E, \epsilon, \omega)$, le graphe.
pour $e \in E$ **faire**
  $s_e \leftarrow nil$
$wl \leftarrow \{\epsilon\}$
tant que $wl \neq \emptyset$ **faire**
  $b \leftarrow pop(wl)$
  $pred \leftarrow ins(b)$
  si $\forall e \in pred, s_e \neq nil$ **alors**
    pour $e \in outs(b)$ **faire**
      $s_e \leftarrow \top$
      $wl \leftarrow wl \cup \{sink(e)\}$

Une variable s_e est définie pour chaque arc e, initialement de valeur néante, et permet le marquage des arcs traités (ceux-ci sont mis à $\top$). Pour l'instant, cet algorithme ne fait rien, mais il est aisé de le modifier pour énumérer tous les chemins du graphe.

4.1.1.3 Avec énumération des chemins

Nous modifions les s_e pour représenter l'ensemble des chemins possibles de l'entrée jusqu'à l'arc en question. Si $\Pi(G)$ est l'ensemble des chemins du graphe analysé G,

$s_e \subseteq \Pi(G)$.

Nous considérons par la suite un *unique* arc d'entrée par graphe, noté $\vec{\epsilon}$. Bien que ce soit généralement le cas lorsqu'un CFG représente une fonction, il est de toutes façons possible de se ramener à ce type de graphe à partir d'un CFG ne respectant pas cette contrainte à l'aide de quelques simples transformations de celui-ci.

Le s_e de l'arc entrant du graphe est initialisé à un seul chemin, constitué de ce même arc. Une fois tous les arcs entrants d'un bloc traité, on réunit tous leurs chemins ($s \leftarrow \bigcup_{e \in pred}(s_e)$) et on y concatène, pour chaque arc sortant e, cet arc même ($s \leftarrow \bigcup_{\pi \in s}(\pi \,.\, e)$).

Algorithme 3 : Énumération des chemins d'un graphe acyclique

Données : $G = (V, E, \epsilon, \omega)$, le graphe.
Résultat : $\{s_{v \to \omega} \mid (v \to \omega) \in E\}$, l'ensemble de tous les chemins d'exécution
pour $e \in E$ **faire**
  $s_e \leftarrow nil$
$s_{\vec{\epsilon}} \leftarrow \{\vec{\epsilon}\}$
$wl \leftarrow \{\epsilon\}$
tant que $wl \neq \emptyset$ **faire**
  $b \leftarrow pop(wl)$
  $pred \leftarrow ins(b)$
  si $\forall e \in pred, s_e \neq nil$ **alors**
    $s \leftarrow \bigcup_{e \in pred} s_e$
    $succ \leftarrow outs(b)$
    pour $e \in succ$ **faire**
      $s_e \leftarrow \bigcup_{\pi \in s}(\pi \,.\, e)$
      $wl \leftarrow wl \cup \{sink(e)\}$

Ainsi, pour l'Algorithme 3, quelques invariants bien placés peuvent prouver que chaque élément π des s_e est bien un chemin, et que, après exécution de l'algorithme, l'ensemble des chemins dans les $s_{v \to \omega}$ des arcs de sortie contient tous les chemins de l'entrée ϵ à la sortie ω (c'est-à-dire, pour un CFG, tous les chemins d'exécution).

4.1.2 Parcours avec interprétation abstraite du programme

4.1.2.1 Interprétation sur un CFG acylique indépendant

Nous considérons dans un premier temps des CFG *indépendants* : ceux-ci ne doivent pas avoir d'appels de fonction nécessitant l'analyse d'un autre CFG. Le domaine des

états s_e que nous utiliserons finalement pour l'analyse de programme est décrit par la Définition 4.2.

Définition 4.2. Les états utilisés pour l'analyse du programme sont constitués d'ensembles de couples d'un chemin et de l'état abstrait lui correspondant. Autrement dit, le domaine de ces états, appelés *états du programme*, est

$$X \coloneqq \mathcal{P}(\widehat{S} \times \Pi(G))$$

Remarque. Si les états de $\widehat{S}$ peuvent être vus comme des conjonctions de propriétés (un état $\hat{s} = (\theta, q)$ représente une conjonction de chaque $v = \theta(v)$ et chaque $p_i \in p$) vraies pour un chemin, les ensembles d'états abstraits de X peuvent être vus comme une disjonction de propriétés vraies en un point du programme (pour tout chemin). De la même façon que nous pouvons définir une fonction $a_{\widehat{S}}(\theta, p) = \bigwedge_{v\in\theta} (v = \theta(v)) \wedge \bigwedge_{p_i\in p} p_i$ décrivant les propriétés d'un état de $\widehat{S}$, la fonction $a_X(s) = \bigvee_{(\hat{s},\pi)\in s} a_{\widehat{S}}(\hat{s})$ décrirait les propriétés exprimées par les états abstraits d'un $s \in X$. Un état de X exprime donc une *disjonction de conjonctions* de propriétés vraies en un point du programme.

Nous étendons naturellement la fonction d'interprétation abstraite $\hat{\mathbb{I}}$ pour raisonner sur une séquence d'instructions sémantiques (un bloc de base, donc).
$\forall i_1, i_2, \ldots, i_n \in I_{sem}$, $\forall \hat{s} \in \widehat{S}$,

$$\hat{\mathbb{I}}[i_1; i_2; \ldots; i_n](\hat{s}) \coloneqq \hat{\mathbb{I}}[i_n] \circ \ldots \circ \hat{\mathbb{I}}[i_2] \circ \hat{\mathbb{I}}[i_1](\hat{s})$$

Nous définissons enfin la fonction d'interprétation d'un bloc de base pour un état du programme.

Définition 4.3. Soit $s \in X = \mathcal{P}(\widehat{S} \times \Pi(G))$ un état du programme, et b un bloc de base de G, associé à une séquence d'instructions sémantiques I_b. Alors, nous définissons la fonction

$$\mathbb{I}[b](s) \coloneqq \bigcup_{(\hat{s},\pi)\in s} \left\{\left(\hat{\mathbb{I}}[I_b](\hat{s}), \pi\right)\right\}$$

Soit e un arc de G, et I_e la séquence d'instructions sémantiques associée. Alors

nous définissons également

$$\mathbb{I}[e](s) \coloneqq \bigcup_{(\hat{s},\pi)\in s} \left\{\left(\hat{\mathbb{I}}[I_e](\hat{s}), \pi \,.\, e\right)\right\}$$

Ceci fait, il est simple d'adapter l'Algorithme 3 pour qu'il calcule les états abstraits en tout point du programme. L'état initial $s_{\vec{\epsilon}}$ entre le CFG avec l'état abstrait identité $1_{\widehat{S}}$ et le chemin constitué de l'arc $\vec{\epsilon}$ seul. À chaque itération, nous récupérons les états sur les arcs en entrée dans une variable s, interprétons le bloc b, et écrivons s sur chaque arc en sortie après interprétation de celui-ci.

Algorithme 4 : Interprétation abstraite d'un CFG acylique indépendant

Données : $G = (V, E, \epsilon, \omega)$, le CFG.
Résultat : $\{s_e \mid e \in E\}$, l'ensemble des états associés à chaque arc de G ; s_G, contenant chaque chemin d'exécution de G, associé à l'état abstrait représentant son exécution

```
pour e ∈ E faire
    s_e ← nil
s_ε⃗ ← {(1_Ŝ, ε⃗)}
wl ← {sink(ε⃗)}
tant que wl ≠ ∅ faire
    b ← pop(wl)
    pred ← ins(b)
    si ∀e ∈ pred, s_e ≠ nil alors
        s ← ⋃_{e∈pred} s_e
        s ← 𝕀[b](s)
        succ ← outs(b)
        pour e ∈ succ faire
            s_e ← 𝕀[e](s)
            wl ← wl ∪ {sink(e)}
s_G ← ∪_{e∈ins(ω)} s_e
```

L'exécution de l'Algorithme 4 permet de dégager deux résultats :

1. Chaque arc e du CFG est associé à un état s_e, contenant un ensemble de chemins et d'états abstraits leur correspondant ; ces derniers permettant de vérifier la satisfiabilité des premiers, comme nous le verrons au Chapitre 5. Grâce à ce résultat, nous sommes d'ores et déjà en mesure de détecter des chemins infaisables dans tout CFG acylique indépendant.

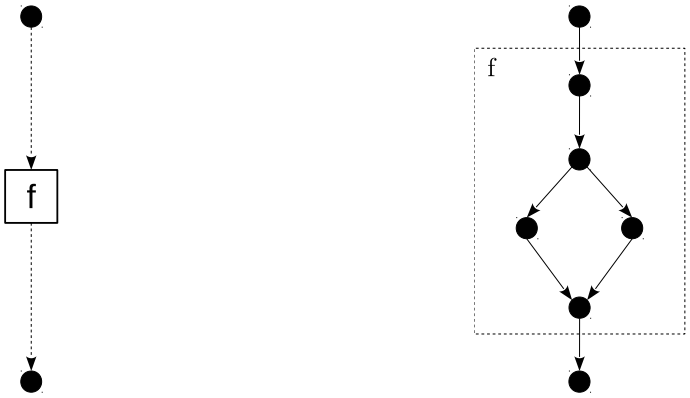
2. La variable $s_G = \cup_{e\in ins(\omega)} s_e$ contient l'ensemble des chemins d'exécution de G

et leur état abstrait correspondant ; cet état modélise l'effet de l'exécution de la fonction représentée par G.

Ce dernier résultat va nous permettre d'étendre l'Algorithme 4 pour traiter les appels (non récursifs) de fonctions.

4.1.2.2 Appels de fonction

Les appels de fonctions sont représentés dans les CFG par des *blocs d'appel*, ou *blocs virtuels*, comme cela a été expliqué en Section 3.1.2.1. Ces blocs, ne contenant eux-mêmes aucune instruction, sont rattachés à une fonction (par son adresse) et représentent son exécution (Figure 4.1). Ces blocs sont munis d'un unique arc de sortie, correspondant à l'arc de retour de la fonction appelée.



(a) Bloc d'appel à une fonction f **(b)** Sémantique décrite par le bloc d'appel

Figure 4.1 – Illustration de la sémantique des blocs d'appel de fonction

Définition 4.4. Pour tout bloc b d'un CFG, nous notons $\mathscr{C}(b)$ l'ensemble des CFG attachés au bloc b. Cet ensemble $\mathscr{C}(b)$ est

- un singleton si b est un bloc d'appel ;
- l'ensemble vide pour tout autre bloc de base.

L'analyse d'un CFG contenant des appels de fonction nécessite donc l'analyse des CFG appelés. Ces dépendances sont établies par le Graphe d'Appel du Programme ou *Program Call Graph* (PCG), mais peuvent aussi être simplement découvertes au fil de l'exécution de l'algorithme (en commençant par l'analyse du CFG principal du programme).

Algorithme 5 : Interprétation abstraite d'un CFG acylique

Données : $G = (V, E, \epsilon, \omega)$, le CFG ;
$\{s_{G_k} \mid \exists b \in V, G_k \in \mathscr{C}(b)\}$, les états correspondant à l'exécution des CFG appelés, préalablement calculés

Résultat : $\{s_e \mid e \in E\}$, l'ensemble des états associés à chaque arc de G ;
s_G, contenant chaque chemin d'exécution de G, associé à l'état abstrait représentant son exécution

pour $e \in E$ **faire**
  $s_e \leftarrow nil$
$s_{\vec{\epsilon}} \leftarrow \{(1_{\widehat{S}}, \vec{\epsilon})\}$
$wl \leftarrow \{sink(\vec{\epsilon})\}$
tant que $wl \neq \emptyset$ **faire**
  $b \leftarrow pop(wl)$
  $pred \leftarrow ins(b)$
  si $\forall e \in pred, s_e \neq nil$ **alors**
    $s \leftarrow \bigcup_{e \in pred} s_e$
    si $\mathscr{C}(b) = \emptyset$ **alors**
      $s \leftarrow \mathbb{I}[b](s)$
    sinon si $\exists G', \mathscr{C}(b) = \{G'\}$ **alors**
      $s \leftarrow s \circ s_{G'}$
    $succ \leftarrow outs(b)$
    pour $e \in succ$ **faire**
      $s_e \leftarrow \mathbb{I}[e](s)$
      $wl \leftarrow wl \cup \{sink(e)\}$
$s_G \leftarrow \bigcup_{e \in ins(\omega)} s_e$

L'Algorithme 5, supportant les appels de programme, requiert donc les états résultant de l'analyse des CFG des fonctions appelées par le CFG analysé G, c'est-à-dire $\{s_{G_k} \mid \exists b \in V, G_k \in \mathscr{C}(b)\}$.

Nous étendons la fonction de composition sur les états du programme comme une forme de produit cartésien :

Définition 4.5. La composition de deux états du programme $s, s' \in X$ est définie par :

$$s \circ s' \coloneqq \left\{ (\hat{s} \circ \hat{s}', \pi.\pi') \;\middle|\; \begin{pmatrix} (\hat{s}, \pi) \\ (\hat{s}', \pi') \end{pmatrix} \in \begin{pmatrix} s \\ s' \end{pmatrix} \right\}$$

Pour un état s en un point d'appel énumérant p chemins, et une fonction f contenant q chemins d'exécution représentés dans s_f, l'interprétation de l'appel à cette fonction résultera logiquement en $p \times q$ chemins représentés dans $s \circ s_f$.

L'Algorithme 5 peut maintenant s'écrire avec une simple modification par rapport à l'Algorithme 4 : au moment de l'interprétation d'un bloc, nous effectuons une composition lorsqu'il s'agit d'un bloc d'appel (qui est, pour rappel, vide d'instructions).

Cet algorithme permet effectivement l'interprétation abstraite de tout programme sans boucle. La gestion des boucles est donc la dernière étape vers une analyse de flot complète.

4.2 Interprétation des boucles par point fixe

4.2.1 Introduction

Nous développons dans cette section une analyse de boucles avec deux objectifs :

- découvrir des propriétés invariantes – vraies pour toute itération – dans les corps de boucles ;
- obtenir un état valide en fin de boucle, pour pouvoir continuer l'analyse du programme.

Par exemple, pour la simple boucle du CFG de la Figure 4.2, nous souhaiterions dégager les propriétés suivantes :

- pour toute itération, à l'entrée de la boucle, $x = 0$ et i est égal au numéro d'itération, soit le nombre total de fois que l'arc de retour ③→② est pris ;
- à la fin de l'exécution de la boucle, $x = 0$ et $i = 10$.

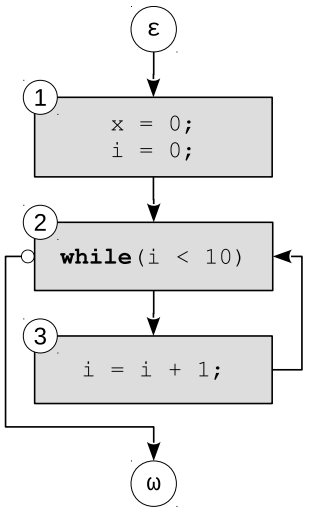


Figure 4.2 – CFG d'une boucle

Bien que les limites de boucles doivent de toute manière être connues pour le calcul du WCET, l'analyse ainsi développée n'utilisera pas de déroulage de boucle, dans le but de préserver une complexité raisonnable et de supporter des applications temps-réel de tailles respectables. De plus, les boucles imbriquées doivent être efficacement supportées.

4.2.2 Union abstraite

Nous avons besoin afin de parcourir les boucles d'un programme de définir un opérateur permettant la réduction d'un ensemble d'états abstraits en un seul. L'union abstraite, est une opération destructrice visant à combiner deux états en conservant autant de propriétés que se peut :

Définition 4.6. Soit $\hat{\sqcup} : \widehat{S} \times \widehat{S} \to \widehat{S}$ l'opérateur d'union abstraite défini par

$$\forall \hat{s} = (\theta, p),\ \hat{s}' = (\theta', p'),$$
$$\hat{s} \,\hat{\sqcup}\, \hat{s}' \coloneqq \left\{ v \mapsto \begin{cases} \theta(v) & \text{si } \theta(v) = \theta'(v) \\ \top & \text{sinon} \end{cases}, \; p \cap p' \right\}$$

Cette définition de l'union abstraite ne conserve que des propriétés égales (ou équivalentes) contenues par les états en opérandes. Nous pouvons réduire la perte de précision entraînée par cette opération en cherchant à conserver des propriétés d'un état (dans θ ou p) qui sont *impliquées* par les propriétés des autres états, par exemple, en

conservant un prédicat $\lambda_1 \geq 0$ pour l'union de deux états, contenant respectivement un prédicat $\lambda_1 \geq 0$ et un prédicat $\lambda_1 \geq 1$. Idéalement, il faudrait même pouvoir déduire, par exemple, $\lambda_1 = 0$ à partir de deux prédicats $\lambda_1 \geq 0$ et $\lambda_1 \leq 0$ issus chacun d'une opérande, un prédicat $\lambda_1 = 0$.

La réalisation de telles déductions sur le domaine abstrait $\widehat{S}$ est complexe, là où des abstractions ciblant un domaine plus restreints de propriétés comme les polyèdres ou les CLP ont l'avantage de définir des opérations d'union plus efficaces. C'est une conséquence de la forte expressivité permise par $\widehat{S}$, du fait que des propriétés peuvent y être représentées très librement (sous la forme de prédicats peu structurés).

Nous définissons par ailleurs l'opérateur $\hat{\sqsubseteq}$, qui, pour deux états abstraits $\hat{s}, \hat{s}' \in \widehat{S}$, est équivalent à une inclusion après concrétisation :

$$\hat{s} \hat{\sqsubseteq} \hat{s}' \iff \forall \vec{s}_0 \in \vec{S},\ \gamma_{\widehat{S}}(\hat{s})(\vec{s}_0) \subseteq \gamma_{\widehat{S}}(\hat{s}')(\vec{s}_0)$$

4.2.3 Notations pour les boucles

Pour rappel, les boucles sont régularisées de telles sorte qu'une tête de boucle soit définie pour chacune d'entre elles (cf. section 3.1.2.2). Nous définissons quelques notations pour accéder aux propriétés ainsi dégagées :

Définition 4.7. Soit G un graphe de flot de contrôle. Alors, nous notons

- H_G l'ensemble des têtes de boucles de G ;
- B_G l'ensemble des arcs de retour de boucle de G.

De plus, pour toute boucle h, nous notons

- $L(h)$ l'ensemble des blocs du corps de h ;
- $\mathit{exits}(h) \coloneqq \{v \to v' \mid v \in L(h) \wedge v' \notin L(h)\}$ l'ensemble des arcs de sortie de h.

Enfin, nous définissons la fonction d'interprétation du corps d'une boucle h, utile à la prochaine partie.

Définition 4.8. Pour toute boucle h, nous notons

$$f_h : \widehat{S} \to \widehat{S}$$

la fonction appliquant l'effet de l'interprétation d'une itération de h sur un état de $\widehat{S}$.

Cette fonction est en réalité définie par l'application par composition ($\circ$) de l'état abstrait $\hat{s}_h$ issu de l'interprétation du corps de la boucle : $\forall \hat{s},\ f_h(\hat{s}) = \hat{s}_h \circ \hat{s}$. Lorsqu'il existe plusieurs chemins dans le corps de boucle (et donc plusieurs états résultant de l'interprétation), f_h donne l'union ($\hat{\sqcup}$) de l'ensemble des états obtenus.

4.2.4 Méthode : calcul de point fixe

La recherche d'un point fixe par unions abstraites successives est une solution simple à l'interprétation de boucles, permettant de découvrir des propriétés vraies indépendamment du numéro d'itération, au prix d'une perte en précision lors des opérations d'union abstraite.

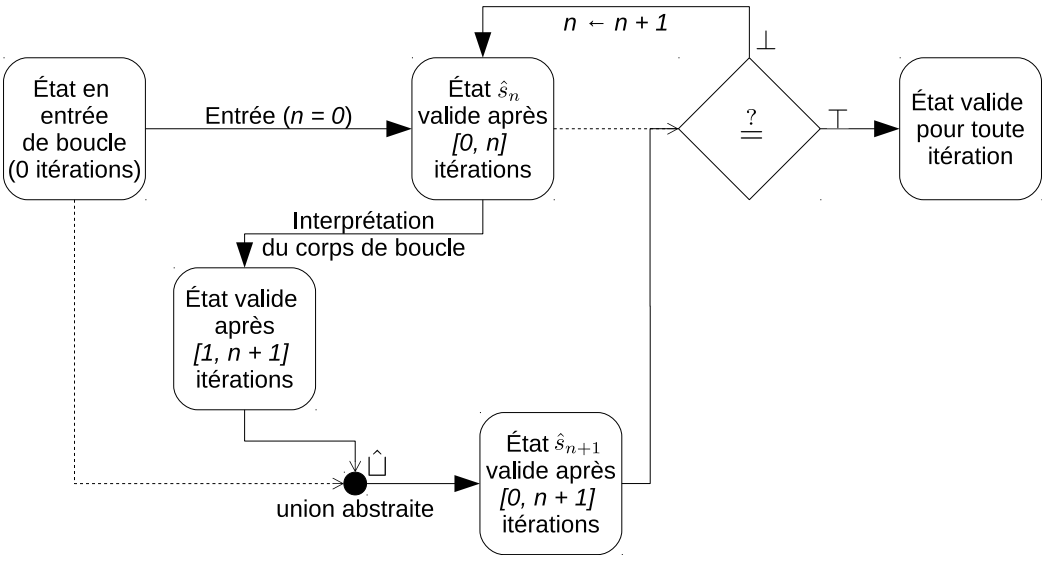


Figure 4.3 – Schéma d'interprétation de boucles par calcul de point fixe

Le schéma de la Figure 4.3 dévoile le fonctionnement de l'interprétation de boucles par calcul de point fixe. L'interprétation commence avec un état $s = \hat{s}_0$, valide seulement en début de boucle (donc pour un nombre d'itérations compris dans $[0, 0]$). À

chaque itération de l'analyse, l'état $\hat{s}$ interprète le corps de la boucle et l'on calcule $\hat{s} \mathbin{\hat{\sqcup}} \hat{s}_0$ jusqu'à obtenir un point fixe.

4.2.5 Formalisation et validation

Ainsi, l'état $\hat{s}_n$, obtenu après n itérations de cette analyse, est défini par induction :

Définition 4.9. Considérons l'interprétation d'une boucle h. Soit $\hat{s}_0$ l'état calculé à l'entrée de la boucle. Alors, $(\hat{s}_n)_n$ est la suite définie par la relation récursive

$$\forall n \geq 0,\ \hat{s}_{n+1} \coloneqq \hat{s}_0 \mathbin{\hat{\sqcup}} f_h(\hat{s}_n)$$

Nous pouvons maintenant énoncer et prouver le lemme central à la validité de cette analyse par point fixe.

Lemme 4.1. *Pour tout n, $\hat{s}_n$ décrit un ensemble de propriétés valides à l'entrée de la boucle h après au plus n ($[0, n]$) itérations de celle-ci. Autrement dit,*

$$\forall n \geq 0, \forall k \leq n,\ f_h^k(\hat{s}_0) \mathbin{\hat{\sqsubseteq}} \hat{s}_n$$

c'est-à-dire que $(f_h^n)_n$ définit une chaîne ascendante.

Sa démonstration est faite à l'Annexe F.

Il en découle le théorème validant cette méthode.

Théorème 4.1. *Soit h une boucle et $\hat{s}_0$ l'état calculé à l'entrée de la boucle. Alors,*

(a) La suite $(\hat{s}_n)_n$ converge, c'est-à-dire

$$\exists m \geq 0, \forall n \geq m,\ \hat{s}_n = \hat{s}_m$$

(b) Les propriétés de l'état vers lequel elle converge sont valides au point d'entrée de boucle, après un nombre quelconque d'itérations, c'est-à-dire

$$\forall i \geq 0,\ \hat{s}_i \mathbin{\hat{\sqsubseteq}} \lim_{n \to +\infty} \hat{s}_n$$

Ce théorème se prouve en montrant que l'opérateur d'union abstraite $\hat{\sqcup}$ résulte toujours en un état soit contenant moins de propriétés significatives (nombre de prédicats et d'éléments de Θ différents de $\top$) que chacun de ses opérandes, soit identique à l'un de ses opérandes. Cette métrique (nombre de propriétés significatives) étant toujours positive, et décroissante à chaque application de $\hat{\sqcup}$, nous en déduisons la convergence de la suite $(\hat{s}_n)$. La validité des propriétés de l'état vers lequel elle converge découle immédiatement du Lemme 4.1.

4.2.6 Algorithme

L'interprétation des boucles par calcul de point-fixe nécessite trois modifications et l'ajout de deux variables à l'Algorithme 5. Le résultat est l'Algorithme 6.

Variables introduites Nous modélisons grossièrement les étapes de l'analyse représentées sur la Figure 4.3 (seulement avant, pendant et après en fait) par une variable q_h attachée à chaque boucle h du CFG, représentant l'état de l'analyse de la boucle h. Trois états sont possibles :

- ENTER : l'analyse de la boucle n'a pas encore commencé ;
- FIX : la recherche d'un point fixe est en cours ;
- LEAVE : un point fixe a été trouvé, les états manipulés sont valides pour toute itération.

Ces états permettent à l'algorithme de situer la progression de l'interprétation de la boucle, et de diriger l'analyse du programme. Ainsi, si le point fixe est trouvé, nous cherchons à sortir de la boucle (LEAVE) et évitons donc les arcs de retour ; à l'inverse, si le point fixe est toujours en cours (FIX), nous ne devons pas prendre les arcs de sortie de la boucle.

Nous associons également à chaque boucle h une variable s_h (un état abstrait de $\widehat{S}$), utilisée pour sauvegarder l'état abstrait en tête de boucle, dans l'état courant du point fixe (correspondant à $\hat{s}_n$ dans le formalisme et la Figure 4.3).

Initialisation Tous les q_h de chaque boucle h sont initialisés à ENTER.

Algorithme 6 : Interprétation abstraite d'un CFG par calcul de point fixe

Données : $G = (V, E, \epsilon, \omega)$, le CFG ; et $\{s_{G_k} \mid \exists b \in V, G_k \in \mathscr{C}(b)\}$
Résultat : $\{s_e \mid e \in E\}$; et s_G
pour $e \in E$ **faire**
  $s_e \leftarrow nil$
pour $h \in H_G$ **faire**
  $q_h \leftarrow$ ENTER
$s_{\vec{\epsilon}} \leftarrow \{(1_{\widehat{S}}, \vec{\epsilon})\}$
$wl \leftarrow \{sink(\vec{\epsilon})\}$
tant que $wl \neq \emptyset$ **faire**
  $b \leftarrow pop(wl)$
  $$pred \leftarrow \begin{cases} ins(b) \text{ if } b \notin H_G \\ ins(b) \setminus B_G \text{ if } b \in H_G \wedge q_b = \text{ENTER} \\ ins(b) \cap B_G \text{ if } b \in H_G \wedge q_b \neq \text{ENTER} \end{cases}$$
  si $\forall e \in pred, s_e \neq nil$ **alors**
    $s \leftarrow \bigcup_{e \in pred} s_e$
    pour $e \in pred$ **faire**
      $s_e \leftarrow nil$
    $succ \leftarrow outs(b)$
    si $b \in H_G$ **alors**
      $s \leftarrow \bigsqcup_{s_i \in s} s_i$
      si $q_b =$ ENTER **alors**
        $q_b \leftarrow$ FIX
        $s_b \leftarrow s$
      sinon si $q_b =$ FIX **alors**
        si $s = s_b$ **alors**
          $q_b \leftarrow$ LEAVE
        sinon
          $s_b \leftarrow s$
      sinon si $q_b =$ LEAVE **alors**
        $q_b \leftarrow$ ENTER
        si $\exists e \in ins(b), s_e \neq nil$ **alors**
          $wl \leftarrow wl \cup \{b\}$
        $succ \leftarrow \emptyset$
    si $\mathscr{C}(b) = \emptyset$ **alors**
      $s \leftarrow \mathbb{I}[b](s)$
    sinon si $\exists G', \mathscr{C}(b) = \{G'\}$ **alors**
      $s \leftarrow s \circ s_{G'}$
    pour $e \in succ \setminus \{exits(h) \mid L(h) \ni b \wedge q_h \neq \text{LEAVE}\}$ **faire**
      $s_e \leftarrow \mathbb{I}[e](s)$
      $wl \leftarrow wl \cup \{sink(e)\}$
$s_G \leftarrow \bigcup_{e \in ins(\omega)} s_e$

Point fixe La variable q_h indiquant l'état du calcul du point fixe suit l'automate de la Figure 4.4.

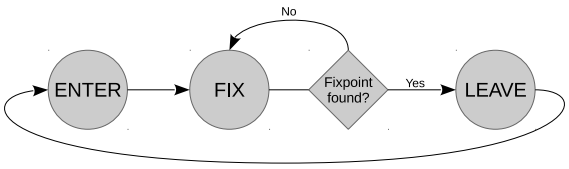


Figure 4.4 – Automate de calcul de point fixe

Lorsque l'analyse entre dans la boucle (ENTER), l'état d'entrée est sauvegardé dans s_h (c'est $\hat{s}_0$ dans le formalisme). Puis, s_h est mis à jour au fur et à mesure du calcul du point fixe (FIX). Enfin, lorsque l'état du point fixe est trouvé (LEAVE, s_h correspond alors à $\lim_{n\to+\infty} \hat{s}_n$ dans le formalisme), nous réinitialisons la boucle (retour à ENTER) et modifions le paramètre *succ* pour en sortir.

Paramètres : prédécesseurs et successeurs Dans le cas du traitement d'une tête de boucle, les paramètres *pred* et *succ* ne sont plus simplement les entrées (*ins*) et sorties (*outs*) du bloc h analysé. Ils suivent au lieu de cela le fonctionnement décrit dans la Table 4.1.

	Têtes de boucle h			Autres blocs
	ENTER	FIX	LEAVE	
pred	arcs d'entrée de h	arcs de retour de h	arcs de retour de h	tout arc d'entrée
succ	tout arc de sortie	tout arc de sortie	aucun	tout arc de sortie

Table 4.1 – Valeur des paramètres *pred* et *succ*

Afin de traiter les boucles imbriquées, ce mode de fonctionnement nécessite une petite modification : lorsque le traitement d'une boucle h est terminé (état LEAVE), il ne faut pas prendre les arcs de sortie qui sont également des arcs de sortie *d'autres boucles* (dont le traitement n'a pas été achevé). C'est-à-dire que

$$\{exits(h) \mid L(h) \ni b \wedge q_h \neq \text{LEAVE}\}$$

doit être exclu des successeurs.

Dans le cas des boucles imbriquées, l'analyse de la boucle interne se termine en premier, et la boucle externe utilise un résultat vrai après un nombre quelconque d'itérations dans la boucle interne. Si h_1 est la boucle externe et h_2 la boucle interne, alors les automates associés à ces boucles évolueront comme illustré sur le Tableau 4.2.

Itération	0	1	2	3	4
q_{h_1}	ENTER	FIX	FIX	FIX	LEAVE
q_{h_2}	ENTER	ENTER	FIX	LEAVE	ENTER

Table 4.2 – Évolution des automates associés à deux boucles imbriquées

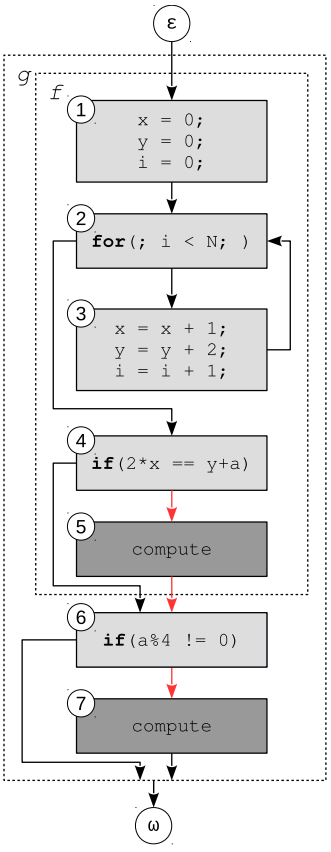
Grâce à la convergence rapide des états par $\hat{\sqcup}$, cet algorithme de parcours de boucle a une faible (bonne) complexité, mais souffre pour la même raison de pertes de précision, étant donné la faiblesse des propriétés invariantes détectables à l'intérieur des boucles.

4.3 Interprétation des boucles par accélération

4.3.1 Le besoin d'une interprétation efficace des boucles

```
void g(int a) {
  f(a);
  if(a % 4 != 0) /* γ2 */
    compute();
}
void f(int a) {
  int x = 0, y = 0, i;
  for(i = 0; i < N; i++) {
    x = x + 1;
    y = y + 2;
  }
  /* (x = i) ∧ (y = 2i) ∧ ¬(i < N) */
  if(2 * x == y + a) /* γ1 */
    compute();
}
```

(a) Programme C de l'Exemple 1



(b) CFG (vue aplatie) de l'Exemple 1

Figure 4.5 – Chemin infaisable entraînant un quasi-doublement de l'estimation du WCET

Reprenons l'Exemple 1 de la section 3.1.2, rappelé sur la Figure 4.5. Ce cas illustre

un chemin infaisable nocif à l'estimation du temps d'exécution pire-cas : la fonction `compute` ne peut être appelée qu'une fois par appel à `g` car le chemin qui rentre dans les deux conditions `if` est infaisable. Dans le cas où `compute` serait une fonction coûteuse avec un WCET élevé par rapport aux opérations dans `f`, détecter ce chemin infaisable permettrait de quasiment réduire par deux le WCET estimé, qui aurait sans cela pris en compte deux appels à `compute`.

Le calcul de point fixe sur la boucle `for` va perdre toute information sur x, y et i : ceux-ci n'étant pas constants, ils vont être mis à $\top$ par l'union $\hat{\sqcup}$. L'utilisation d'opérations dites d'élargissement et de réduction de la théorie de l'interprétation abstraite sur notre domaine abstrait nous permettrait d'encadrer les valeurs prises par ces variables dans la boucle, mais pas de représenter leur évolution, ni le lien entre elles (par exemple, la propriété invariante $y = 2x$).

Une analyse efficace de la fonction `f` doit donner lieu à deux chemins en sortie ; celui qui provoque l'appel à `compute` impliquera que `f` a été appelé avec $a = 0$, l'autre avec $a \neq 0$. En effet, l'analyse de la boucle doit nous permettre de déduire que $y = 2x$, et donc que la condition $2x = y + a$ n'est possible qu'avec $a = 0$. Lorsque la fonction `g` sera analysée, nous devrons observer que la condition `a % 4 != 0` est en conflit avec `2*x == y+a`, grâce au prédicat $y = 2x$. Autrement dit, la conjonction de prédicats $(y = 2x) \wedge (2x = y + a) \wedge (a \bmod 4 \neq 0)$ est insatisfiable, fausse pour toute valeur de x, y, a. Ceci implique que le chemin d'exécution qui passe à travers les deux appels à `compute` ($\{\gamma_1, \gamma_2\}$) est sémantiquement impossible.

```
void f(int a, int b) {
  int i, x = 0;
  for(i = 0; i < a; i++) {
    x = x + b;
    if(x % 2 == 1) /* α */
      a = a - 1;
  }
  if(i > 50) /* β */
    i = 50;
}
f(30,2);
```

Figure 4.6 – Code dynamiquement mort à l'intérieur et après une boucle

Un autre exemple est donné sur la Figure 4.6, dont le code contient deux chemins infaisables constitués d'un seul arc dans `f` (α et β). Ces arcs ne portent pas sur du code mort : leur exécution n'est impossible que dans le contexte de l'appel `f(30,2);`. Le code des blocs pointés par α et β est dit *dynamiquement mort*. La découverte du chemin infaisable constitué par α (et en réalité, l'arc d'appel à `f`) n'est possible que si l'on parvient à détecter dans la boucle des propriétés invariantes comme $x = 2i$.

Afin d'identifier de telles propriétés, qui peuvent, comme sur l'Exemple 1, être à l'origine de chemins infaisables nuisant à la précision de l'estimation du WCET, nous allons améliorer les techniques présentées en section 4.2 et enrichir les états de $\widehat{S}$ pour

trouver de meilleurs invariants, et détecter potentiellement plus de chemins infaisables plus tard.

4.3.2 Le domaine $\breve{S}$

Afin de pouvoir inclure les numéros d'itération des boucles analysées dans les paramètres des propriétés exprimées par les états abstraits, nous enrichissons les expressions de $\widehat{\mathbb{E}}_\Lambda$:

Définition 4.10. Soit, pour un ensemble de têtes de boucles d'un programme $H = \{h_1, h_2, \ldots, h_n\}$, l'ensemble des *indices de boucles*

$$\mathcal{I} \coloneqq \{I_{h_k} \mid h_k \in H\}$$

où chaque I_h représente le nombre d'itérations de h effectuées par le programme depuis la dernière entrée dans h.

Nous étendons les expressions pour inclure ces variables d'indice de boucle :

$$\breve{\mathbb{E}}_\Lambda \coloneqq \left| \begin{array}{c} C^\sharp \\ \mathbb{V}^{\sharp *} \\ \Lambda \\ \mathcal{I} \\ \breve{\mathbb{E}}_\Lambda \times \Phi \times \breve{\mathbb{E}}_\Lambda \end{array} \right.$$

Les états abstraits sont mis à jour pour inclure ces expressions

$$\breve{\Theta} \coloneqq (\mathbb{V}^\sharp \to \breve{\mathbb{E}}_\Lambda)$$
$$\breve{P}_\Lambda \coloneqq \breve{\mathbb{E}}_\Lambda \times \Psi \times \breve{\mathbb{E}}_\Lambda$$
$$\breve{S} \coloneqq \breve{\Theta} \times \mathcal{P}(\breve{P}_\Lambda)$$

Heureusement, l'extension des opérations de $\widehat{S}$ sur $\breve{S}$ immédiate : il suffit de considérer les variables de $\mathcal{I}$ comme des éléments d'un ensemble de variables arbitraires Λ étendu. Les valeurs des variables de $\mathcal{I}$ étant inconnues tout au long de l'analyse, elles peuvent être traitées comme des éléments de Λ par tous les opérateurs et fonctions vus jusqu'ici.

Exemple. Dans le cas de la boucle de l'Exemple 1 (notons-la h_0), notre objectif est d'obtenir en fin d'analyse un état $\{x = I_{h_0},\ y = 2I_{h_0},\ i = I_{h_0}\}$ qui décrit précisément l'évolution des valeurs des variables x, y et i. Nous pourrons plus tard (au Chapitre 5) utiliser l'implication $(x = I_{h_0} \wedge y = 2I_{h_0} \wedge i = I_{h_0}) \implies y = 2x$ pour identifier un chemin infaisable.

Nous étendons trivialement l'état identité sur $\check{S}$, inchangé :

$$1_{\check{S}} \coloneqq \{v \mapsto v^*, \emptyset\}$$

4.3.3 Accélération d'état

4.3.3.1 Principe

Nous allons maintenant exploiter, au bénéfice de l'analyse de boucles, la propriété capitale des états abstraits que nous manipulons depuis la section 3.5 : les états abstraits peuvent être utilisés pour représenter l'exécution indépendante de toute région à une entrée et une sortie. Des régions intéressantes sont les CFG de fonctions (ce qui permet la composabilité de l'analyse lors des appels de fonction), mais aussi les corps de boucle (en considérant la tête de boucle comme nœud d'entrée *et* de sortie) !

Cela signifie que toute boucle du programme peut être analysée séparément, et que l'on peut ainsi obtenir en une seule passe d'analyse une fonction représentant l'exécution d'une itération de la boucle en question. Comme pour toute fonction f dont l'espace de départ coïncide avec l'espace d'arrivée, on peut chercher à déterminer une expression générale de f^n, pour tout $n \geq 0$.

Remarque. Lors de l'analyse d'un corps de boucle, les variables de $\widehat{\mathbb{V}}$ représentent donc non plus la valeur de ces variables en début d'analyse de fonction, mais leur valeur en début d'itération de boucle.

Exemple. Considérons à nouveau la boucle de l'Exemple 1. L'interprétation de la

seule région constituée par ce corps de boucle donnera l'état $\check{s}_{h_0}$

$\check{\mathbf{s}}_{\mathbf{h_0}}$		
θ	x	$x^* + 1$
	y	$y^* + 2$
	i	$i^* + 1$
	$\ldots$	

représentant effectivement l'exécution du code contenu dans le corps de cette boucle. Nous pouvons voir chacune de ses variables comme une suite arithmétique :

$$\begin{array}{cccc} & x_0 \coloneqq x^* & y_0 \coloneqq y^* & i_0 \coloneqq i^* \\ \forall n \geq 0, & x_{n+1} \coloneqq x_n + 1 & y_{n+1} \coloneqq y_n + 2 & i_{n+1} \coloneqq i_n + 1 \end{array}$$

Or, il est aisé de déterminer le cas général d'une suite arithmétique :

$$\forall n \geq 0, \quad x_n \coloneqq x^* + n \qquad y_n \coloneqq y^* + 2n \qquad i_n \coloneqq i^* + n$$

Cette variable n correspond au nombre d'exécutions de la région de programme représentée par $\check{s}_{h_0}$, c'est donc en réalité le nombre d'itérations de h_0, I_{h_0}. Il suffit ensuite de composer ces formules avec tout état $s = (\theta, p)$ en entrée de boucle, remplaçant ainsi les x^*, y^*, i^* par $\theta(x), \theta(y), \theta(i)$, c'est-à-dire $(0, 0, 0)$ dans le cas de l'Exemple 1, pour obtenir un état valide pour tout chemin arrivant en entrée de boucle dans la région l'englobant. Cet état général est $\check{s}^{\curvearrowleft}_{h_0}$:

$\check{\mathbf{s}}^{\curvearrowleft}_{\mathbf{h_0}}$		
θ	x	I_{h_0}
	y	$2I_{h_0}$
	i	I_{h_0}
	$\ldots$	

Un schéma de cette nouvelle approche pour l'interprétation des boucles est présenté sur la Figure 4.7.

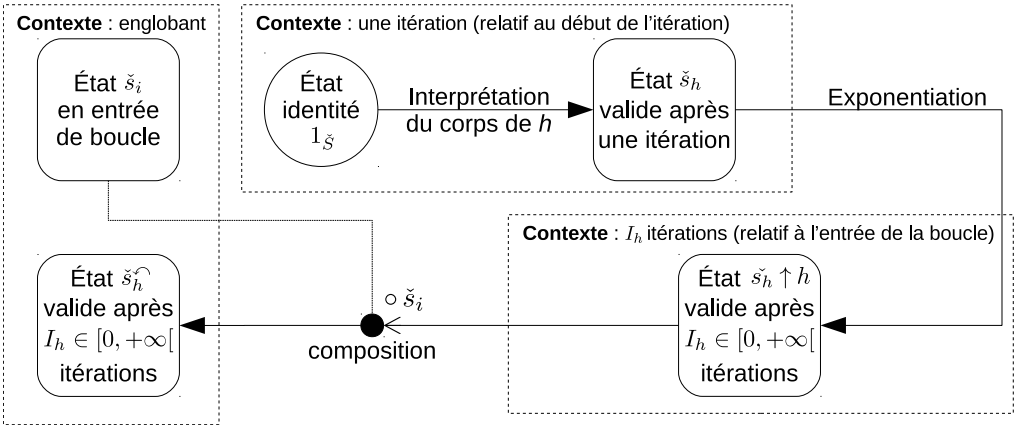


Figure 4.7 – Schéma d'interprétation de boucles par accélération

4.3.3.2 Formalisation

Soit $\curvearrowleft^h$ l'opération de généralisation d'un état sur une boucle h, prenant en paramètre un état $\check{s}_h$ représentant l'exécution d'une itération de h, et opérant sur un état initial $\check{s}_i \in \breve{S}$ représentant l'état en entrée de boucle. Le calcul de $\check{s}_i \curvearrowleft^h \check{s}_h$ se fait en deux étapes :

1. Déduire à partir de $\check{s}_h$ un état valide en tête de boucle pour toute itération, paramétré par la valeur des variables à l'entrée de la boucle h : cet état est obtenu par application d'un opérateur $\uparrow^h$ dit d'*accélération* (cf. Définition 4.11)
2. Appliquer l'état $\check{s}_h\uparrow^h$ obtenu à l'état initial $\check{s}_i$ pour repasser dans le contexte englobant la boucle h.

Les expressions et prédicats dans le résultat final sont donc exprimées en fonction des mêmes valeurs initiales dans $\widehat{\mathbb{V}}$ que $\check{s}_i$. L'analyse peut ensuite reprendre le parcours du CFG normalement (en ignorant les arcs de retour pour ne pas boucler), et identifier des propriétés vraies pour toute itération.

Définition 4.11. L'opérateur d'*accélération d'état* $\uparrow^h$ est défini, pour toute boucle h et pour tout état $\check{s}_h \in \breve{S}$ représentant l'effet d'une itération de h, comme

$$\check{s}_h\uparrow^h := (v \mapsto \check{s}_h(v)\uparrow^h_v, \emptyset)$$

avec $\uparrow_v$ défini pour toute variable $v \in \mathbb{V}^\sharp$ comme

$$\forall e \in \breve{\mathbb{E}}_\Lambda,\ e{\uparrow_v^h} \coloneqq \begin{cases} v^* & \text{si } e = v^* \\ v^* + k\, I_h & \text{si } e = v^* + k,\ k \in \mathbb{Z}_{32} \\ \top & \text{sinon} \end{cases}$$

Intuition. L'opérateur d'accélération $\uparrow^h$ permet d'identifier des constantes et de généraliser des suites arithmétiques, à partir de la représentation d'une itération de h incarnée par $\check{s}_h$. À partir de cet état défini dans le contexte d'une itération de h, nous obtenons un état vrai dans le contexte de I_h itérations de h.

Exemple.

$\check{\mathbf{s}}_\mathbf{h}$		
θ	r_0	$r_0^* + 1$
	r_1	r_0^*
	r_2	5
	r_3	$r_3^* - 2$
	r_4	r_4^*
	r_5	$r_5^* \bmod 2$
	$\ldots$	

$\uparrow^h =$

$\check{\mathbf{s}}_\mathbf{h}{\uparrow^\mathbf{h}}$		
θ	r_0	$r_0^* + I_h$
	r_1	$\top$
	r_2	$\top$
	r_3	$r_3^* - 2I_h$
	r_4	r_4^*
	r_5	$\top$
	$\ldots$	

Remarque. Cet opérateur met à $\top$ les variables affectées à une constante dans la boucle, comme pour r_2 dans l'exemple ci-dessus. Ceci est dû au fait qu'une variable v systématiquement assignée à une constante k dans le corps d'une boucle n'est pas forcément égale à cette même constante k en tête de boucle, à la première itération. Au niveau de la tête de boucle, $v = k$ n'est donc garanti que pour $I_h \neq 0$.

Définissons maintenant l'opérateur de généralisation $\curvearrowleft^h$, selon la méthode présentée plus haut :

Définition 4.12. L'opérateur $\curvearrowleft^h$ de généralisation d'état sur une boucle h à partir d'un état $\check{s}_h \in \breve{S}$ représentant une exécution de h et d'un état $\check{s}_i \in \breve{S}$ en entrée de h est ainsi défini :

$$\check{s}_i \curvearrowleft^h \check{s}_h \coloneqq (\check{s}_h{\uparrow^h}) \circ \check{s}_i$$

Exemple. Nous illustrons le fonctionnement de cet opérateur $\curvearrowleft^h$ sur la Figure 4.8.

La condition de validité de cet opérateur de généralisation est énoncée par le Théorème 4.2.

Théorème 4.2. *Pour tout $\check{s}_h \in \check{S}$, et pour tout $n \geq 0$,*

$$\check{s}_i \curvearrowleft^h \check{s}_h \check{\sqsubseteq} \check{\mathbb{I}}[h]^n(1_{\check{S}})$$

Ce théorème est une conséquence du Lemme 4.2.

Lemme 4.2. *Pour tout $\check{s}_h \in \check{S}$, et pour tout $n \geq 0$,*

$$\check{s}_h{\uparrow}^h[I_h \mapsto n] \check{\sqsubseteq} \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n \text{ fois}}$$

Démonstration. Ce lemme peut être prouvé par récursion.

Soit $(\check{\theta}_h, \emptyset) = \check{s}_h{\uparrow}^h[I_h \mapsto n]$ et pour tout n, $(\check{\theta}_n, \emptyset) = \check{\theta}_h{\uparrow}^h_v[I_h \mapsto n]$. Soit $\check{s}_h = (\check{\theta}_h, p_h)$. Nous noterons abusivement $\underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n \text{ fois}}(v)$ la valeur associée à v dans la table des variables calculée par la composition $\check{s}_h \circ \ldots \circ \check{s}_h$. Nous cherchons en premier lieu à établir que, pour toute variable $v \in \mathbb{V}^\sharp$ et pour tout $n \geq 0$,

$$\check{\theta}_n(v) = \top \ \vee \ \check{\theta}_n(v) = \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{\text{n fois}}(v)$$

- *Initialisation :*

$$\check{\theta}_0(v) = \begin{cases} v^* & \text{si } \check{\theta}_h(e) = v^* \\ v^* + k \times 0 & \text{si } \check{\theta}_h(e) = v^* + k,\ k \in \mathbb{Z}_{32} \\ \top & \text{sinon} \end{cases}$$

Donc

$$\check{\theta}_0(v) = \top \ \vee \ \check{\theta}_0(v) = v^* = 1_{\check{\Theta}}(v^*)$$

- *Hérédité :* Supposons, pour un $n \geq 0$ fixe, que la propriété suivante est vraie :

$$\check{\theta}_n(v) = \top \ \vee \ \check{\theta}_n(v) = \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n \text{ fois}}(v)$$

Alors,

$$\check{\theta}_n(v) = \top \ \vee \ \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n+1 \text{ fois}}(v) = \check{s}_h \circ \check{\theta}_n(v)$$

Or, par définition de $\check{\theta}_n$ et de la composition,

$$\check{s}_h \circ \check{\theta}_n(v) = \begin{cases} v^* & \text{si } \check{\theta}_h(e) = v^* \\ (v^* + k\,n) + k & \text{si } \check{\theta}_h(e) = v^* + k,\ k \in \mathbb{Z}_{32} \\ \top & \text{sinon} \end{cases}$$

De plus,

$$\check{\theta}_{n+1}(v) = \begin{cases} v^* & \text{si } \check{\theta}_h(e) = v^* \\ v^* + k \times (n+1) & \text{si } \check{\theta}_h(e) = v^* + k,\ k \in \mathbb{Z}_{32} \\ \top & \text{sinon} \end{cases}$$

Comme $\check{\theta}_n(v) = \top \Rightarrow \check{\theta}_{n+1}(v) = \top$, et $\check{\theta}_{n+1}(v) = \check{s}_h \circ \check{\theta}_n(v) = \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n+1 \text{ fois}}(v)$, la propriété est vraie au rang $n+1$:

$$\check{\theta}_{n+1}(v) = \top \ \vee \ \check{\theta}_{n+1}(v) = \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n+1 \text{ fois}}(v)$$

- *Généralisation* : Pour toute variable $v \in \mathbb{V}^\sharp$ et pour tout $n \geq 0$,

$$\check{\theta}_n(v) = \top \ \vee \ \check{\theta}_n(v) = \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n \text{ fois}}(v)$$

Par conséquent, pour tout $\vec{s} \in \vec{S}$ et pour toute valuation des variables abstraites L, et pour tout n,

$$\gamma_{\check{\mathbb{E}}_\Lambda}(\vec{s}, L, \check{\theta}_n) = \vec{S} \ \vee \ \gamma_{\check{\mathbb{E}}_\Lambda}(\vec{s}, L, \check{\theta}_n) = \gamma_{\check{\mathbb{E}}_\Lambda}(\vec{s}, L, \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n \text{ fois}})$$

et donc, *a fortiori*,

$$\gamma_{\check{\mathbb{E}}_\Lambda}(\vec{s}, L, \check{\theta}_n) \subseteq \gamma_{\check{\mathbb{E}}_\Lambda}(\vec{s}, L, \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n \text{ fois}})$$

Il en suit que

$$\forall \vec{s}_0 \in \vec{S},\ \gamma_{\check{S}}(\check{\theta}_h{\uparrow}^h[I_h \mapsto n])(\vec{s}_0) \subseteq \gamma_{\check{S}}(\underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{n \text{ fois}})(\vec{s}_0)$$

Et donc, par définition de l'inclusion abstraite,

$$\check{s}_h{\uparrow}^h[I_h \mapsto n] \mathrel{\check{\sqsubseteq}} \underbrace{\check{s}_h \circ \ldots \circ \check{s}_h}_{\text{n fois}}$$

□

Le théorème se déduit immédiatement du Lemme 4.2, en admettant les deux propriétés suivantes :

- La composition est croissante : $\forall \check{s}, \check{s}', \check{s}'' \in \check{S},\ \check{s} \mathrel{\check{\sqsubseteq}} \check{s}' \Longrightarrow \check{s} \circ \check{s}'' \mathrel{\check{\sqsubseteq}} \check{s}' \circ \check{s}''$ (et en particulier pour $\check{s} = \check{s}_i$) ;
- $\forall \check{s} \in \check{S}, \forall f : \check{S} \to \check{S},\ \check{s} = f(1_{\check{S}}) \Longrightarrow \underbrace{\check{s} \circ \ldots \circ \check{s}}_{n \text{ fois}} = f(1_{\check{S}})$ (et en particulier pour $\check{s} = \check{s}_h$ et $f = \check{\mathbb{I}}[h]$).

Par ailleurs, afin de conserver les relations entre variables, nous remplaçons en sortie de toute boucle h tous les I_h par des N_h (et étendons $\mathcal{I}$ en conséquence) représentant le nombre *exact* d'itérations de la boucle h : N_h fixe la valeur de I_h en sortie de boucle. Si une analyse tierce nous communique une borne M pour la boucle h, nous pouvons rajouter un prédicat $N_h \leq M$. Parfois, même le nombre exact d'itérations est connu, N_h peut alors être remplacé par la constante correspondante. Sinon, N_h joue le même role que les variables abstraites de Λ en maintenant des liens entre les variables du programme. N_h représentant un nombre (positif) d'itérations, un prédicat $N_h \geq 0$ peut toujours être ajouté.

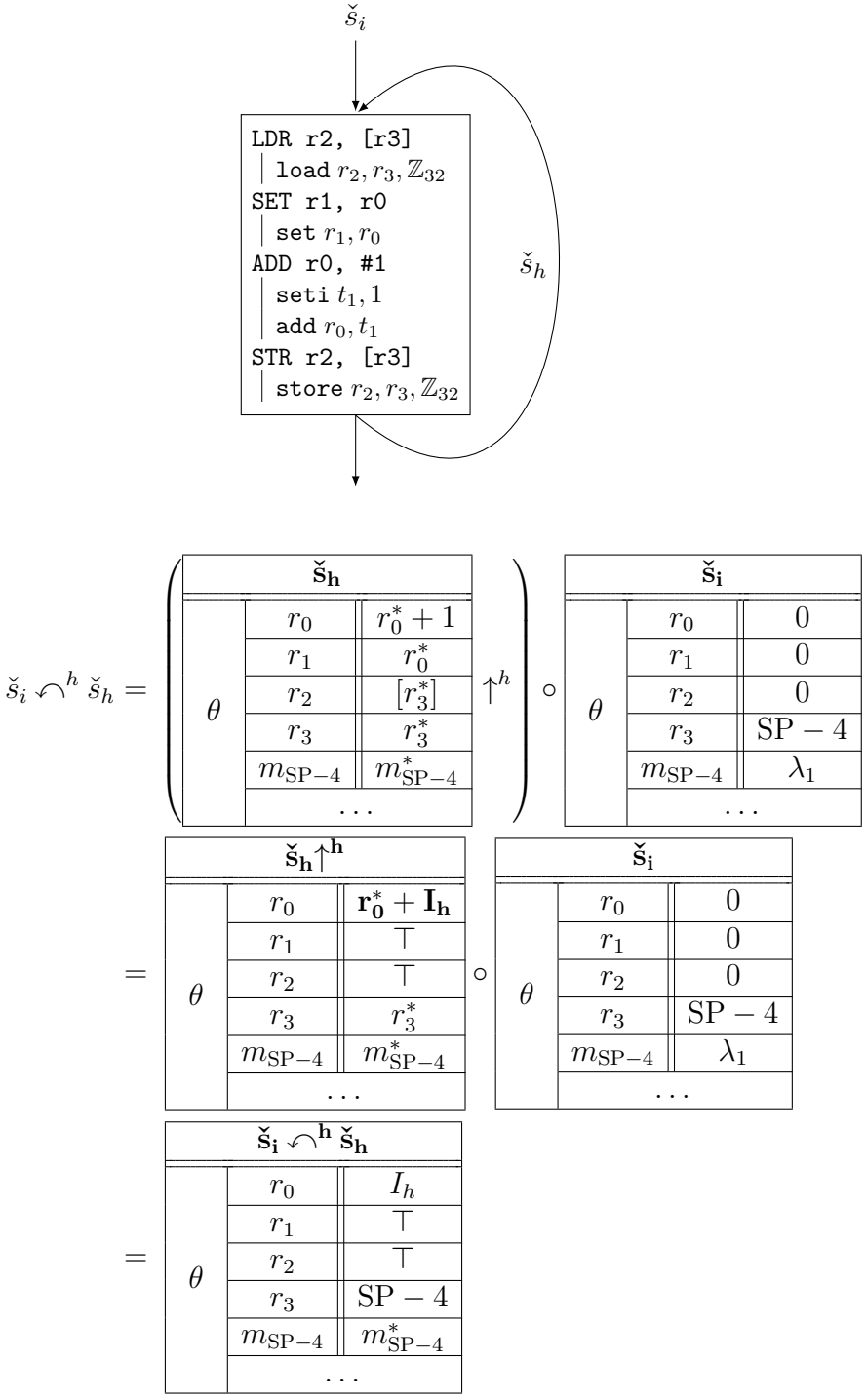


Figure 4.8 – Application de l'opérateur $\curvearrowleft$ sur une simple boucle

4.3.3.3 Calcul de $\hat{s}_h$

$\check{s}_h$ est l'état abstrait tel que la fonction $f_h(\hat{s}) \coloneqq \check{s}_h \circ \hat{s}$ représente le résultat après une itération du corps de boucle h sur $\check{s}$. Cet état est simplement calculé en interprétant le corps de la boucle sur un état initial $1_{\check{S}}$. Si nous notons $\check{\mathbb{I}}[h] : \check{S} \to \check{S}$ la fonction d'interprétation d'une boucle h sur un état, interprétant chaque chemin du corps de boucle et faisant l'union abstraite de l'ensemble des états obtenus, alors :

$$\check{s}_h = \check{\mathbb{I}}[h](1_{\check{S}})$$

Afin que $\hat{s}_h$ puisse représenter exactement l'effet d'une itération, nous pouvons enrichir les expressions avec un opérateur $[e]$ qui représente un accès mémoire à une expression quelconque e. Pendant la généralisation, si l'expression qui représente l'adresse mémoire accédée n'est pas constante, on peut revenir sur $\top$ sans risque pour la validité du résultat.

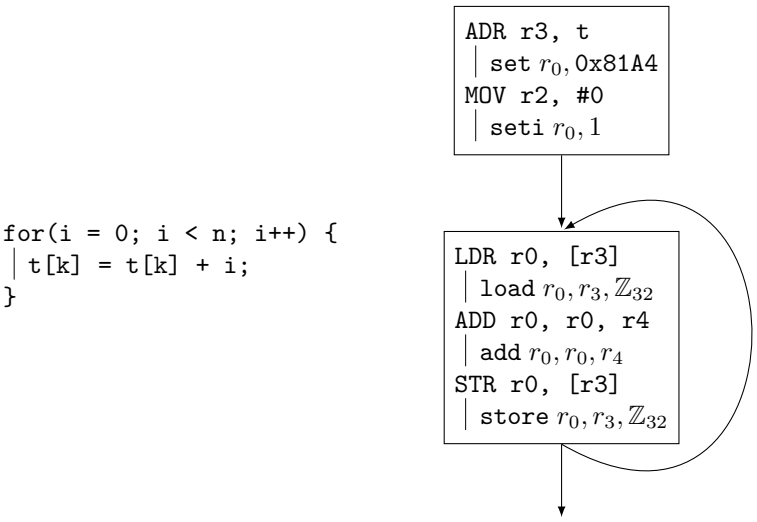


Figure 4.9 – Calcul de la somme d'une suite arithmétique

Toutefois, cela ne suffit pas pour traiter des données à des adresses enregistrées en dehors de la boucle dans des registres. Considérons l'exemple de la Figure 4.9, qui fait la somme de n éléments d'une suite arithmétique de raison k. L'adresse du tableau t est enregistrée dans un registre r_3 avant la boucle. L'utilisation d'un $\check{s}_h$ obtenu par interprétation de la boucle par $\check{\mathbb{I}}[h]$ pour représenter une itération de cette boucle nous fait perdre toute propriété précédemment découverte sur la mémoire. En effet, même si nous parvenions à déduire que les accès à $t[0]$ et $t[1]$ portent sur des emplacements disjoints

dans la mémoire, toute propriété serait détruite par l'écriture aux adresses inconnues $t[0]$ et $t[1]$. La perte de toute information sur les valeurs enregistrées en mémoire, dont les constantes, a des effets dévastateurs sur l'efficacité de l'analyse dans la suite.

Par ailleurs, les compilateurs enregistrent fréquemment les variables d'un programme dans la pile pour libérer des registres (c'est même systématique lorsque les optimisations sont désactivées sur certains compilateurs). Si une de ces variables est une constante utilisée dans la boucle, sa valeur sera une inconnue lors de l'interprétation par $\check{\mathbb{I}}[h]$. Ainsi, sur l'exemple précédent, n aurait pu être enregistré dans une cellule mémoire plutôt que dans r_4. Ce problème peut nous empêcher de détecter des propriétés importantes, en nous privant des informations sur les constantes à l'intérieur de la boucle.

Afin de palier à ce problème, il est intéressant de faire l'interprétation d'une itération de la boucle en partant d'un état enrichi avec des propriétés constantes, plutôt que de l'état identité strictement dépourvu d'information. Nous définissons pour cela la fonction $\phi_{\mathbb{Z}_{32}{}^\sharp}$:

Définition 4.13. La fonction $\phi_{\mathbb{Z}_{32}{}^\sharp} : \check{S} \times \check{S} \to \check{\Theta}$ effectue une projection d'un état $\check{s} \in \check{S}$ en ne conservant que les informations sur les variables constantes, associant le reste des variables à leur valeur initiale (comme pour l'état identité $1_{\check{S}}$) :

$$\forall \check{s} = (\theta, p),\ \check{s}' = (\theta', p') \in \check{S},\ \phi_{\mathbb{Z}_{32}{}^\sharp}(\check{s}, \check{s}') \coloneqq v \mapsto \begin{cases} \theta(v) & \text{si } \theta(v) = v^* \wedge \theta'(v) \in \mathbb{Z}_{32}{}^\sharp \\ v^* & \text{sinon} \end{cases}$$

Nous pouvons alors définir un état $\check{s}'_h$ plus efficace, utilisant des informations sur les constantes en entrée de boucle lorsque $\check{s}_h$ indique qu'elles restent constantes en tête de boucle, pour chaque itération :

$$\check{s}'_h = \check{\mathbb{I}}[h](v \mapsto \phi_{\mathbb{Z}_{32}{}^\sharp}(\check{s}_h, \check{s}_i), \emptyset) \quad \text{avec } \check{s}_h = \check{\mathbb{I}}[h](1_{\check{S}})$$

4.3.4 Algorithme

Nous modifions l'automate de la Figure 4.4 en remplaçant le point fixe par un état ITER pour l'interprétation d'une itération quelconque (partant de $1_{\check{S}}$) du corps de boucle. Nous passons ensuite dans l'état LEAVE, qui permet cette fois l'analyse de

la boucle avec des formules paramétrées par l'indice d'itération I_h, et le calcul d'états valides pour tout itération. L'automate ainsi obtenu est affiché sur la Figure 4.10.

L'algorithme d'analyse de graphe de flot de contrôle en est simplifié. Sur l'Algorithme 7, le traitement de tête de boucle réutilise s_b pour stocker l'état initial (que nous notions $\check{s}_i$ dans les formules) et effectue simplement l'accélération d'état : $s \leftarrow s_b \curvearrowleft^b s$.

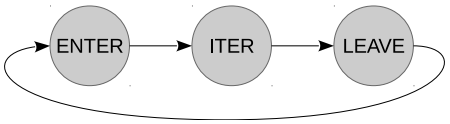


Figure 4.10 – Automate des états de l'analyse d'une boucle par accélération d'état

Cela suffit pour effectuer une analyse complète de flot de données du programme par interprétation abstraite. L'algorithme peut toutefois être complété par une autre analyse identifiant des propriétés du programme à la volée, représentée dans l'Algorithme 7 par la fonction Ξ, grisée. C'est à l'intérieur de cette fonction que se fera par la suite le développement de la recherche de chemins infaisables, utilisant cette analyse de flot de données pour l'amélioration du WCET, *in fine*.

4.4 Discussion et conclusion

4.4.1 Discussion

Étant donné que les algorithmes de parcours de CFG présentés dans ce chapitre énumèrent tous les chemins du programme – au moins pour les régions séquentielles analysées – l'analyse se heurte à un problème de complexité, en particulier en la présence de nombreuses conditions en séquence, chaque condition dédoublant le nombre de chemins à analyser. Ce nombre croît exponentiellement, tout comme la complexité de l'analyse : chaque chemin nécessite une analyse séparée, et génère un problème de satisfiabilité propre à tester.

Nous proposons une solution à ce problème par l'introduction d'un seuil ajustable, noté η, du nombre de chemins analysables en tout point du programme [2], pour des programmes de grande et moyenne taille. Ainsi, dans l'Algorithme 7 au lieu de collecter

2. Nous utilisons $\eta = 250$ ou $\eta = +\infty$ pour la quasi-totalité de nos expérimentations (à l'exception d'un programme pour lequel nous avons dû abaisser le seuil à 80).

Algorithme 7 : Interprétation abstraite d'un CFG par accélération d'état

Données : $G = (V, E, \epsilon, \omega)$, le CFG ; et $\{s_{G_k} \mid \exists b \in V, G_k \in \mathscr{C}(b)\}$
Résultat : $\{s_e \mid e \in E\}$; et s_G
pour $e \in E$ **faire**
  $s_e \leftarrow nil$
pour $h \in H_G$ **faire**
  $q_h \leftarrow$ ENTER
$s_{\vec{\epsilon}} \leftarrow \{(1_{\check{S}}, \vec{\epsilon})\}$
$wl \leftarrow \{sink(\vec{\epsilon})\}$
tant que $wl \neq \emptyset$ **faire**
  $b \leftarrow pop(wl)$
  $$pred \leftarrow \begin{cases} ins(b) \text{ si } b \notin H_G \\ ins(b) \setminus B_G \text{ si } b \in H_G \wedge q_b = \text{ENTER} \\ ins(b) \cap B_G \text{ si } b \in H_G \wedge q_b \neq \text{ENTER} \end{cases}$$
  si $\forall e \in pred, s_e \neq nil$ **alors**
    $s \leftarrow \bigcup_{e \in pred} s_e$
    pour $e \in pred$ **faire**
      $s_e \leftarrow nil$
    $succ \leftarrow outs(b)$
    si $b \in H_G$ **alors**
      $s \leftarrow \bigsqcup_{s_i \in s} s_i$
      si $q_b =$ ENTER **alors**
        $q_b \leftarrow$ ITER
        $s_b \leftarrow s$
      sinon si $q_b =$ ITER **alors**
        $q_b \leftarrow$ LEAVE
        $s \leftarrow s_b \curvearrowleft^b s$
      sinon si $q_b =$ LEAVE **alors**
        $q_b \leftarrow$ ENTER
        si $\exists e \in ins(b), s_e \neq nil$ **alors**
          $wl \leftarrow wl \cup \{b\}$
        $succ \leftarrow \emptyset$
    si $\mathscr{C}(b) = \emptyset$ **alors**
      $s \leftarrow \mathbb{I}[b](s)$
    sinon si $\exists G', \mathscr{C}(b) = \{G'\}$ **alors**
      $s \leftarrow s \circ s_{G'}$
    pour $e \in succ \setminus \{exits(h) \mid L(h) \ni b \wedge q_h \neq \text{LEAVE}\}$ **faire**
      $s_e \leftarrow \mathbb{I}[e](s)$
      $\Xi(s_e, \{(h, q_h) \mid L(h) \ni b\})$
      $wl \leftarrow wl \cup \{sink(e)\}$
$s_G \leftarrow \bigcup_{e \in ins(\omega)} s_e$

les états des arcs en entrée avec

$$s \leftarrow \bigcup_{e \in pred} s_e$$

nous utilisons plutôt

$$s \leftarrow \begin{cases} \bigcup_{e \in pred} s_e & \text{si } \sum_{e \in pred} |s_e| \leq \eta \\ \widehat{\bigsqcup}_{e \in pred} s_e & \text{sinon} \end{cases}$$

en fusionnant par union abstraite ($\widehat{\sqcup}$) l'ensemble des états en entrée dans le cas où le seuil η est dépassé. Ce seuil pousse l'analyse à travailler sur des fenêtres du programme de taille modeste, et améliore la complexité, évitant des temps d'analyse déraisonnables. Cela se fait au prix de ne plus pouvoir différencier les chemins en entrée de chacun de ces points de fusion, et donc d'être incapable de détecter des chemins infaisables entre des arcs avant et après un tel point.

Une autre piste d'amélioration de la complexité est la simplification du graphe de contrôle et de ses blocs par *slicing* (cf section 3.1.2.6). Après avoir implanté cette technique dans notre outil, nous avons constaté des gains de performance, mais avons perdu un nombre significatif de chemins infaisables, du fait que le *slicing* peut supprimer du code dynamiquement mort, et restructurer le graphe de manière à le faire disparaître.

Par ailleurs, il est possible de chercher à détecter des formes de chemins infaisables particulières selon les applications : des comportements plus complexes pourraient être reconnus par la fonction d'accélération (Définition 4.11) pour supporter des schémas adaptés à l'application considérée ; par défaut, seuls les progressions arithmétiques sont traitées. On pourrait par exemple chercher à généraliser des suites arithmético-géométriques (du type $u_{n+1} = au_n + b$, et dont la forme générale est $u_n = (u_0 - \frac{b}{1-a})a^n + \frac{b}{1-a}$) ou d'autres problèmes, qui peuvent nécessiter une extension des expressions de $\breve{\mathbb{E}}_\Lambda$.

D'autres domaines abstraits permettent et facilitent le développement de méthodes d'accélération plus performantes, comme par exemple les travaux de Ancourt et al. [8] dans PIPS, qui utilisent la différentiation discrète sur des contraintes affines entières, représentées par des polyèdres. Le processus d'accélération gagne ainsi en généralité par rapport aux méthodes de reconnaissance de formes que nous utilisons. L'outil FAST inclut également des techniques d'accélération [11] pour des systèmes linéaires représentés par des formules de Presburger sur des entiers positifs. Enfin, Gonnord et Halbwachs [41] cherchent également à calculer l'effet exact des boucles lorsque pos-

sible. Une forme d'accélération abstraite raisonnant sur des approximations polyédrales est définie, généralisant efficacement des fonctions f ainsi représentées à des fonctions vraies après k itérations. Cette méthode est utilisée en complément à des techniques classiques d'élargissement, moins précises mais capables de traiter les cas où l'accélération est impossible.

Enfin, remarquons que les méthodes d'analyse présentées dans la section 4.2 sont applicables pour des abstractions non paramétriques comme $\widetilde{S}$, à condition d'utiliser une vue aplatie des graphes de flot de contrôle.

4.4.2 Conclusion générale

En se basant sur les développements du Chapitre 3, traitant de l'intérieur des blocs de bases, le Chapitre 4 achève notre analyse de programme. Nous avons raffiné un algorithme de parcours de graphe de flot de contrôle pour l'analyse par interprétation abstraite de programme.

Utilisant un système de rendez-vous, le parcours d'un CFG résulte en un état abstrait présent sur chaque arc et sur chaque bloc, représentant une approximation des états possibles de la machine en ces points du programme.

Grâce à la composabilité de l'abstraction utilisée, chaque CFG n'est analysé qu'une fois, donnant des résultats indépendants du contexte d'appel. Les états obtenus sont ensuite spécialisés à chaque contexte d'appel, permettant de bénéficier des avantages d'une vue aplatie des CFG (découverte de propriétés spécifiques à un contexte d'appel, factorisation des propriétés indépendantes du contexte d'appel) sans les inconvénients (duplication de l'analyse, complexité élevée, absence de propriétés générales).

Cette même composabilité nous a permis de définir une opération d'accélération d'état, qui vient remplacer un algorithme de parcours de boucle par point fixe moins précis. Des schémas sont reconnus et des propriétés arithmétiques sont utilisées pour permettre la détermination de propriétés vraies pour toute itération. Ces propriétés sont exprimées en fonction des indices d'itérations, variables de $\mathcal{I}$ dorénavant acceptées dans la syntaxe des expressions utilisées la table des variables et dans les prédicats.

L'algorithme final (Algorithme 7) est complet, à la seule exception de la définition de la fonction Ξ, dernière pièce du puzzle, qui doit en un point du programme utiliser les propriétés découvertes sur l'état de la machine en ce point pour identifier de potentiels

chemins infaisables. C'est de l'exploitation de ces propriétés de flot de données pour la détection de propriétés de flot de contrôle que traitera donc le prochain chapitre.

5

Chemins infaisables

Sommaire

Ce chapitre se base sur l'analyse de flot de données complète précédemment définie (Chapitres 3, 4) pour détecter des chemins infaisables, sous la forme d'ensembles d'arcs du CFG en conflit sémantique. Nous ferons appel à un solveur externe pour résoudre un problème de décision que nous extrairons des informations de flot de données du programme analysé, et qui représentera la satisfiabilité de l'exécution d'un chemin. Afin d'éviter toute perte de généralité, l'implantation n'est pas dépendante d'un outil en particulier, et les informations de flot de contrôle ainsi obtenues sont représentées dans le format portable FFX [21].

5.1 Introduction : un problème SMT

5.1.1 Les chemins infaisables, un problème de décision

L'analyse de flot de données par interprétation abstraite définie au Chapitre 4 permet de transposer la question "ce chemin est-il infaisable" [1] en "n'existe-t-il aucun état de la machine modélisé par l'état abstrait associé à ce chemin". Cette question mathématique est un *problème de décision*, questionnant l'existence d'une solution à un problème, dont la réponse doit être "oui" ou "non".

La résolution de ce problème de décision est complexe. En effet, il n'est pas envisageable de tester l'appartenance de chaque état de la machine à la concrétisation d'un état abstrait afin de déterminer que celle-ci donne un ensemble vide (l'espace des états concrets étant beaucoup trop grand, correspondant à l'ensemble des valuations possibles de chaque registre et cellule mémoire). En revanche, il est possible de raisonner sur la formule logique (de premier ordre) qui définit l'appartenance d'un état concret à l'ensemble des états modélisés par un état abstrait. Par exemple, la question de l'existence d'un état de la machine modélisé par l'état

θ	x	$x^* + 1$
	y	x^*
	i	I_{h_0}
	$m_{\mathrm{SP}-4}$	$\top$
	$m_{\texttt{0x8004}}$	λ_1
	$m_{\texttt{0x8008}}$	λ_1
	$\ldots$	
p	$x = y$	
	$m_{\mathrm{SP}-4} \neq 0$	

peut s'écrire sous la forme de la formule logique [2]

$$(x = x^* + 1) \wedge (y = x^*) \wedge (i = I_{h_0}) \wedge (m_{\texttt{0x8004}} = \lambda_1) \wedge (m_{\texttt{0x8008}} = \lambda_1) \wedge (x = y) \wedge (m_{\mathrm{SP}-4} \neq 0)$$

1. Ou "l'exécution de cette séquence d'arcs contigus du CFG est-elle sémantiquement impossible".
2. "$m_{\mathrm{SP}-4} = \top$" n'apporte aucune information et n'est donc pas traduit.

à laquelle nous cherchons un modèle (une valeur pour chaque variable vérifiant la formule) :

$$\exists ?x, y, z, i, m_{\mathrm{SP}-4}, m_{\texttt{0x8004}}, m_{\texttt{0x8008}}, x^*, \lambda_1, I_h \in \mathbb{Z}_{32}, \begin{cases} & (x = x^* + 1) \\ \wedge & (y = x^*) \\ \wedge & (i = I_{h_0}) \\ \wedge & (m_{\texttt{0x8004}} = \lambda_1) \\ \wedge & (m_{\texttt{0x8008}} = \lambda_1) \\ \wedge & (x = y) \\ \wedge & (m_{\mathrm{SP}-4} \neq 0) \end{cases}$$

Ce problème est un problème de *satisfiabilité* (SAT) bien connu en informatique, plus précisément, un problème de Satisfiabilité Modulo des Théories (SMT), soit un problème SAT étendu à des formules de logique classique du premier ordre, dans laquelle une variété de théories peuvent être implémentées. Celles qui nous intéresseront dans la suite seront plus particulièrement les théories d'arithmétique entière, adaptées à notre problème.

La structure des états abstraits permet même sa représentation sous la forme, plus restrictive, de *conjonctions*, c'est-à-dire, des formules logiques n'usant pas de disjonctions (opérateur $\vee$, ainsi que $\oplus$, $\rightarrow$, ...). Cela facilite la recherche d'une solution à ces questions, étant donné que la complexité de leurs algorithmes de résolution ont tendance à doubler à chaque disjonction (en transformant la formule en une forme normale disjonctive, soit une disjonction de conjonctions).

5.1.2 Des états abstraits aux prédicats SMT

Nous devons tout d'abord définir la fonction de traduction des états abstraits (de $\breve{S}$) en une conjonction de prédicats. La formule ainsi obtenue est sans quantificateur ($\exists, \forall$), composée de variables libres. Ensuite, l'objectif est de prouver, le cas échéant, l'absence de solutions à cette formule (insatisfiabilité), indiquant un chemin infaisable. Définissons en premier lieu le domaine des variables des prédicats de cette formule [3] :

3. En réalité, dans une optique d'optimisation, seules les variables utilisées dans au moins un prédicat seront déclarées au solveur SMT.

Définition 5.1. Le domaine des variables (entières) acceptées par les prédicats arithmétiques SMT sera

$$\mathbb{V}^{\sharp^*} \cup \Lambda \cup \mathcal{I}$$

Ce domaine contient l'ensemble des inconnues exprimables dans les prédicats, sauf $\top$, qui n'est pas une variable correspondant à une valeur fixe : $x = \top$ et $y = \top$ n'entraînent pas $x = y$, par exemple.

La valeur symbolique SP sera confondue avec son registre correspondant dans $\mathbb{V}^*$, étant donné qu'ils représentent la même chose (la valeur initiale du pointeur de pile).

Nous présentons ensuite le mécanisme de traduction des états abstraits en prédicats arithmétiques :

Définition 5.2. Nous définissons la *fonction de traduction* t d'un état abstrait en prédicat comme suit :

$$\forall(\check{\theta}, p),\ t(\check{\theta}, p) \coloneqq \bigwedge_{\substack{p_i \in p \\ \top \notin \mathscr{U}_{\check{P}_\Lambda}(p_i)}} p_i[sp^*/\mathrm{SP}]$$

où sp^* est la variable de $\mathbb{V}^*$ correspondant au pointeur de pile de l'architecture considérée.

Remarque. La traduction des propriétés sur les valeurs courantes des variables des tables $\check{\theta}$ n'est pas utile à la recherche d'états insatisfiables. En effet, chaque contrainte ainsi générée, qui serait de la forme $v = \check{\theta}(v)[sp^*/\mathrm{SP}]$, ne pourrait jamais être en conflit avec une autre, parce que ce serait la seule à faire apparaître la valeur $v \in \mathbb{V}$ (les prédicats de $\check{P}_\Lambda$ ne raisonnant que sur des valeurs initiales de $\widehat{\mathbb{V}}$).

Propriété 5.1. *Soit un état abstrait $\check{s} \in \check{S}$. L'insatisfiabilité de $t(\check{s})$ est une condition suffisante à l'absence d'état concret modélisé par $\check{s}$. Autrement dit,*

$$t(\check{s}) \vdash \bot \Longrightarrow \gamma_{\check{S}}(\check{s}) = (\vec{s}_0 \mapsto \emptyset)$$

Il s'agit maintenant de déterminer l'existence d'une solution à cette formule, c'est-à-dire de résoudre le problème de satisfiabilité.

5.1.3 Solveurs SMT

5.1.3.1 Principe

De nombreux outils existent pour répondre à ces problèmes SMT, implémentant un éventail de théories différentes (raisonnant sur des entiers, des ensembles, des vecteurs de bits, des expressions régulières...). La catégorie de problèmes que nous cherchons à résoudre relève de l'*arithmétique entière*.

L'utilisation de tels solveurs dans ce cadre est simple : il suffit de définir un ensemble de variables sur $\mathbb{Z}$ et une liste d'*hypothèses* ou contraintes (Figure 5.1). Le solveur aura alors deux possibilités de réponse :

- *sat* : le problème est satisfiable, c'est-à-dire qu'il existe un vecteur de valeurs entières associé aux variables définies pour lequel chaque hypothèse est vraie. Un tel vecteur est appelé un *modèle*, et peut être exhibé par le solveur.
- *unsat* : le problème est insatisfiable, c'est-à-dire que quelles que soient les valeurs prises par les variables du problème, l'ensemble (la conjonction) des hypothèses ne pourra pas être vérifié.

Le solveur peut aussi échouer en dépassant la limite des ressources lui étant attribuées – en temps (*timeout*) ou en mémoire – mais nous n'avons jamais rencontré de tel cas pour l'ensemble des applications testées.

Remarque. Si, en pratique, certains de ces solveurs sont capables de résoudre – avec grande difficulté – des problèmes non-linéaires *réels*, ceux de l'arithmétique non-linéaire entière sont indécidables. Les opérations de division et de modulo par une constante peuvent toutefois être traduites par l'introduction d'une fonction symbolique représentant l'effet de l'opération en question, ce qui permet de résoudre des problèmes tels l'insatisfiabilité de $x/7 = x/7 + 1$ (car $\forall f : \mathbb{Z} \to \mathbb{Z}, f(x) \neq f(x) + 1$). Cette technique de *réécriture* permet un support limité des formules non-linéaires.

5.1.3.2 Unsat cores

Des avancées récentes sur le problème de satisfiabilité d'une formule booléenne (problème SAT) ont permis de nouvelles extensions des solveurs traitant le problème difficile de l'extraction d'*unsat cores* ou "noyaux insatisfiables minimaux" [23, 67, 107].

```
a, b, c, d, e: INT;
ASSERT a > b + 2;
ASSERT a = (2 * c) + 10;
ASSERT c + b <= 1000;
ASSERT d >= e;
CHECKSAT;
```

(a) Expression dans CVC4

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Int)
(declare-const e Int)
(assert (> a (+ b 2)))
(assert (= a (+ (* 2 c) 10)))
(assert (<= (+ c b) 1000))
(assert (>= d e))
(check-sat)
```

(b) Expression dans Z3

Figure 5.1 – Un problème d'arithmétique entière linéaire exprimé dans différents solveurs SMT

Pour tout problème SAT insatisfiable, il est possible d'extraire un ou plusieurs sous-ensembles minimaux de contraintes insatisfiables, appelés *unsat cores*. Un *unsat core* est un minimum *local* : il est tel que la suppression de toute contrainte le constituant le rendrait satisfiable, mais il peut exister des ensembles insatisfiables plus petits (non inclus).

Exemple. Considérons les variables entières x, y, z, w et le problème SMT constitué de la liste de contraintes

$$\{x > y,\, y > z,\, z > x,\, z > w,\, w > x\}$$

Alors,

$$\{x > y,\, y > z,\, z > w,\, w > x\}$$

est un ensemble (localement) minimal de contraintes insatisfiables, un *unsat core*. Notons que ce n'est toutefois ni le seul, ni le plus petit :

$$\{x > y,\, y > z,\, z > x\}$$

est également un *unsat core*.

Cette fonctionnalité nous sera utile dans la suite, pour la minimisation des chemins infaisables obtenus (section 5.2.4).

Solveur	API C++ ?	Licence	Arithmétique entière ?		Unsat cores ?
			linéaire	non-linéaire	
Barcelogic	Oui (C++)	Propriétaire	Non	Non	Non
Boolector	Compatible (C)	Libre (GPL)	Non	Non	Non
CVC4	Oui (C++)	Libre (BSD)	Oui	Oui [4]	Oui
VeriT	Compatible (C)	Libre (BSD)	Oui	Non	Non
Yices 2	Compatible (C)	Libre (GPL)	Oui	Non	Non
Z3	Oui (C++)	Libre (MIT)	Oui	Oui	Oui

Table 5.1 – Comparatif de solveurs SMT

5.1.3.3 Choix du solveur

La Table 5.1 liste six grands solveurs SMT d'actualité. Tous ont une API compatible pour l'intégration en C++ dans notre outil PathFinder. Barcelogic [20] et Boolector [80] ne sont pas adaptés à notre problème puisqu'ils n'implantent pas de théorie arithmétique entière. VeriT [22] et Yices 2 [37] traitent les problèmes de l'arithmétique entière linéaire, mais ne permettent pas l'extraction d'*unsat cores*.

En revanche, CVC4 [12] et Z3 [71] sont de bons choix : ils sont sous licences libres, mettent à disposition une API C++, permettent l'extraction d'*unsat cores* et peuvent même parfois raisonner sur des problèmes d'arithmétique entière non-linéaire [5], bien que de manière très limitée (par simples réécritures dans le cas de CVC4).

L'interface de résolution de problème SMT de PathFinder pour l'identification de chemins infaisables intégrera donc ces deux solveurs, CVC4 et Z3, améliorant ainsi l'indépendance des résultats que nous obtiendrons par rapport aux spécificités de ces deux outils, aussi minimes soient-elles.

5.2 Détection et expression de chemins infaisables

La section précédente a présenté un moyen de tester la satisfiabilité d'un état abstrait – c'est-à-dire de déterminer s'il modélise au moins un état concret – et par conséquent du chemin du CFG associé. Nous allons maintenant définir, à l'aide de cette technique, une méthode pour détecter des chemins infaisables dans un programme. Nous nous basons pour cela sur le dernier algorithme défini au Chapitre 4, l'Algorithme 7.

4. Par réécriture.
5. Il faut sélectionner la théorie *Quantifier-Free Linear Integer Arithmetic*(QF-LIA) pour CVC4.

5.2.1 Implémentation de la routine Ξ

La recherche de chemins infaisables se fait à la volée (en même temps que le parcours du CFG par interprétation abstraite) afin de couper les chemins en question de l'analyse le plus tôt possible et d'éviter de dupliquer le chemin et de détecter le même conflit de nombreuses fois. Ainsi, sur l'exemple illustré sur la Figure 5.2, si π est un chemin infaisable, il est préférable de l'identifier ainsi plutôt que de tester $\pi.e_1.e_3.e_5$, $\pi.e_1.e_4.e_6$, $\pi.e_2.e_3.e_5$, et $\pi.e_2.e_4.e_6$ et de trouver quatre chemins infaisables exprimant un seul conflit.

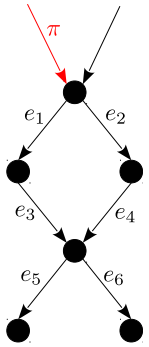


Figure 5.2 – Duplication d'un chemin infaisable π

Nous définissons sat : $\breve{S} \to \{\bot, \top\}$ comme étant la fonction qui, pour tout état abstrait $\breve{s}$, teste la satisfiabilité de la conjonction de prédicats définie par $t(\breve{s})$ grâce à un solveur SMT. Le résultat est tel que

$$\text{sat}(\breve{s}) = \begin{cases} \bot & \text{si } t(\breve{s}) \vdash \bot, \text{ c'est-à-dire } \gamma(\breve{s}) = (\vec{s}_0 \mapsto \emptyset) \\ \top & \text{sinon} \end{cases}$$

Nous pouvons maintenant définir la routine Ξ de l'Algorithme 7, prenant en entrée

- un état du programme s_e issu de l'analyse par interprétation abstraite, soit un couple constitué d'un ensemble de couples
  - d'un état abstrait $\breve{s} \in \breve{S}$,
  - du chemin associé $\pi \in \Pi$

  pour chaque chemin aboutissant à e,
- le contexte d'analyse $\Gamma = \{(h_1, q_{h_1}), \ldots, (h_n, q_{h_n})\}$ où chaque h_k est une boucle englobant l'arc e, et q_{h_k}, donnant l'état de l'analyse de la boucle h_k (ENTER, ITER, LEAVE). De par la conception de l'Algorithme 7, cet état ne sera en réalité jamais ENTER, état signifiant que l'analyse n'est pas dans la boucle.

La routine donne *ips*, l'ensemble des chemins infaisables détectés, et modifie s_e pour la suite de l'analyse, en supprimant les états insatisfiables.

Algorithme 8 : Routine Ξ : détection de chemins infaisables sur un arc e

Données : s_e, l'état sur l'arc e ; Γ, contexte d'analyse des boucles englobantes
Résultat : s_e, l'état sur l'arc modifié ; *ips*, les chemins infaisables détectés

```
ips ← {}
si ∀(h, q_h) ∈ Γ, q_h = LEAVE alors
    pour (š, π) ∈ s_e faire
        si sat(š) = ⊥ alors
            ips ← ips ∪ {π}
            s_e ← s_e \ {(š, π)}
```

L'Algorithme 8 donne une pré-définition de la routine Ξ. L'état d'analyse LEAVE est celui qui indique que nous raisonnons dans le cas général, avec des abstractions valides pour toute itération. Si une des boucles englobantes de l'arc e n'est pas dans l'état final LEAVE, nous ne recherchons pas de chemins infaisables, ceux-ci seront (potentiellement) trouvés plus tard [6], après progression de l'analyse jusqu'à l'état LEAVE.

Si les états q_h indiquent une progression suffisante de l'analyse, l'Algorithme 8 lance le test de satisfiabilité (recherche de contradictions) pour chaque état abstrait $\check{s}$ dans s_e. Si un état $\check{s}$ est effectivement déterminé insatisfiable, son chemin associé π est ajouté à l'ensemble des chemins infaisables *ips*, et le couple $(\check{s}, \pi)$ est supprimé de s_e.

Les éléments de π représentant des chemins, donc des séquences *consécutives* d'arêtes, leur sémantique est claire, et aucune information par rapport aux boucles n'a besoin d'être rajoutée.

Exemple. Ainsi, si nous considérons les blocs de base a, b, c dans une boucle h ($\{a, b, c\} \subseteq L(h)$), comme sur la Figure 5.3, le chemin infaisable $\{a \to b \,.\, b \to c\}$, également noté

$$a \to b \to c$$

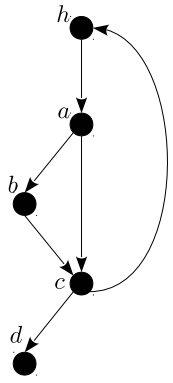


Figure 5.3 – Une boucle

s'applique à pour toute itération de h. En revanche, si d est un bloc hors de la boucle

6. Nous pourrions en réalité rechercher des chemins infaisables pendant que l'analyse est en état ITER, mais c'est inutile puisque l'analyse aura dans l'état LEAVE des propriétés plus puissantes ; les chemins trouvés dans l'état ITER seraient donc redondants.

($d \notin L(h)$), le chemin infaisable

$$a \to b \to c \to d$$

n'a de sens que sur la dernière itération de h.

5.2.2 Chemins abstraits

Nous introduisons le domaine des chemins abstraits. Les chemins abstraits, à l'instar des états abstraits qui représentent un ensemble d'états possibles de la machine, représentent un ensemble de chemins du programme. En revanche, à la différence des états abstraits, cette abstraction n'introduit pas de perte de précision [7], elle sert simplement à *l'expression concise de chemins infaisables*, préparant leur future exploitation [8].

Définition 5.3. Le domaine des *chemins abstraits* Π_G^* d'un CFG $G = (V, E, \epsilon, \omega)$ est simplement défini comme une séquence constituée

- d'arcs de E

  et

- de connecteurs ${}_a\chi_b^l$ représentant l'ensemble des chemins entre deux blocs $a, b \in V$, excluant un ensemble de blocs $l \subseteq V$, possiblement vide.

Exemple. Soient $a, b, c, d, e, f, g, h \in V$ des blocs d'un CFG $G = (V, E, \epsilon, \omega)$. Alors,

- $(a \to b).\,{}_b\chi_c.(c \to d)$ représente tous les chemins de l'arc $a \to b$ à l'arc $c \to d$;
- $(a \to b).\,{}_b\chi_c^{\{a\}}.(c \to d)$ représente le même ensemble de chemins, boucles sur a exclues ;
- ${}_\epsilon\chi_\omega$ représente l'ensemble des chemins d'exécution de G.

Remarque. Toutes les séquences de Π_G^* ne sont pas des chemins abstraits *valides*, c'est-à-dire que certaines ne représentent aucun chemin existant dans G. En effet, afin

7. Seulement dans le sens "concrétisation puis abstraction", certains ensembles de chemins n'étant pas représentables par un seul chemin abstrait.

8. Cette abstraction est par exemple adaptée à l'insertion efficace des informations de chemins infaisables dans un système de contraintes ILP (par injection de contraintes) pour l'amélioration du calcul de WCET, comme nous le verrons dans la section 5.3.1.

d'être valide, chaque connecteur doit coïncider avec les blocs le précédant et le suivant dans le chemin. En outre, une séquence vide n'est pas un chemin abstrait valide.

La sémantique de ces chemins abstraits est formellement décrite par la fonction de concrétisation d'un chemin abstrait.

Définition 5.4. Pour tout CFG $G = (V, E, \epsilon, \omega)$, la fonction de concrétisation d'un chemin abstrait $\gamma_{\Pi_G^*} : \Pi_G^* \to \mathcal{P}(\Pi_G)$ est ainsi définie :

$$\forall \pi^* \in \Pi_G^*,\ \gamma_{\Pi_G^*}(\pi^*) \coloneqq$$
$$\begin{cases} \{e\} & \text{si } \pi^* = e,\ e \in E \\ \gamma_\chi({}_a\chi_b^l) & \text{si } \pi^* = {}_a\chi_b^l,\ \{a,b\}, l \subseteq E \\ \{e \,.\, \pi' \mid \pi' \in \gamma_{\Pi_G^*}(\pi^{*\prime})\} & \text{si } \pi^* = (e \,.\, \pi^{*\prime}),\ e \in E,\ \pi^{*\prime} \in \Pi_G^* \\ \{\pi \,.\, \pi' \mid \pi \in (\gamma_\chi({}_a\chi_b^l),\ \pi' \in \gamma_{\Pi_G^*}(\pi^{*\prime})\} & \text{si } \pi^* = ({}_a\chi_b^l \,.\, \pi^{*\prime}),\ \{a,b\}, l \subseteq E,\ \pi^{*\prime} \in \Pi_G^* \end{cases}$$

avec la concrétisation d'un connecteur γ_χ définie comme :

$$\forall a, b \in E,\ \forall l \subseteq E,\ \gamma_\chi({}_a\chi_b^l) \coloneqq \left\{ \pi \in \Pi_G \;\middle|\; \begin{array}{c} \exists \pi_1 \in \Pi_G,\ \exists v_1 \in V,\ \pi = (a \to v_1) \,.\, \pi_1 \\ \exists \pi_2 \in \Pi_G,\ \exists v_2 \in V,\ \pi = \pi_2 \,.\, (v_2 \to b) \\ \forall (v \to v') \in \pi,\ v' \notin l \end{array} \right\}$$

Remarque. Les chemins d'un graphe étant de simples séquences d'arcs, acceptées dans le domaine des chemins abstraits, la fonction d'abstraction $\alpha_{\Pi_G^*}$ serait une projection triviale ($\alpha_{\Pi_G^*}(\pi) = \pi$).

5.2.3 Expression de chemins infaisables dans FFX

Nous avons brièvement présenté le langage d'annotation portable FFX en section 2.4.3. Les chemins infaisables détectés par l'analyse présentée dans cette thèse sont exprimées au format FFX, permettant leur exploitation par des outils externes capables de lire ce format et d'utiliser des propriétés parmi celles qu'il exprime. Nous allons maintenant présenter le sous-ensemble de ce langage qui sera utilisé pour exprimer des chemins infaisables, sous la forme de conflits entre arcs des CFG du programme.

5.2.3.1 Les conflits dans FFX

La syntaxe du sous-ensemble de FFX utilisé est définie par la grammaire partielle de la Figure 5.4.

```
FLOWFACTS :=
  <flowfacts>
    CONFLICT+
  </flowfacts>

CONFLICT :=
  <conflict seq="true">
    FUNCTION
  </conflict>

FUNCTION :=
  <function address=ADDR>
    ITEM*
  </function>

ITEM :=
  EDGE
  CALL
  LOOP

EDGE :=
  <edge source=ADDR target=ADDR />
```

```
CALL :=
  <call ADDRESS>
    FUNCTION
  </call>

LOOP :=
  <loop address=ADDR>
    ITERATION_ALL
    ITERATION_LAST
  </loop>

ITERATION_ANY :=
  <iteration number="*">
    ITEM+
  </iteration>

ITERATION_LAST :=
  <iteration number="n">
    ITEM+
  </iteration>

ADDR := "INT"
```

Figure 5.4 – Grammaire FFX partielle

Essentiellement, un chemin infaisable est un conflit entre un ensemble d'arcs dans un contexte donné (boucles, points d'appel). Ainsi, dans FFX, chaque chemin infaisable est représenté par un conflit (`CONFLICT`) entre une liste d'éléments (`ITEM`) dans le contexte d'une fonction (`FUNCTION`). Chacun de ces éléments peut être

- un arc (`EDGE`), représenté par l'adresse de la dernière instruction du bloc de départ et l'adresse de la première instruction du bloc d'arrivée ;
- un ensemble d'éléments dans le contexte d'un appel (`CALL`) à une fonction (`FUNCTION`) identifié par l'adresse du point d'appel ;
- un ensemble d'éléments dans le contexte d'une boucle (`LOOP`) identifiée par l'adresse de la tête de boucle, considérés valides
  - soit pour toute itération (`ITERATION_ANY`)

– soit pour la dernière itération (`ITERATION_LAST`)

de la boucle en question.

Il est important de noter que les arcs en conflit seront listés en séquence, dans l'ordre dans lequel ils ont été lus par l'analyse. Cela garantit que, dans un contexte donné (boucle ou fonction), les arcs listés (`EDGE`) appartiennent à un chemin du contexte en question [9]. Cette propriété sera essentielle par la suite (section 5.2.3.3), elle est exprimée dans le langage FFX par l'attribut `seq` de la balise `<conflict>`, systématiquement positionné à `"true"` dans notre cas.

5.2.3.2 Écriture mathématique des conflits FFX

Remarque. Les chemins issus de l'analyse de programme définie dans le chapitre précédent sont un peu particuliers : leurs arcs peuvent provenir de multiples CFG (appelant et appelés). Ceux-ci sont représentés dans notre outil par les adresses des blocs source et destination dans le programme binaire ; il est donc immédiat de déterminer le CFG de la fonction de laquelle chaque arc provient. La génération et la sémantique des balises de contexte de fonction (`<function>`) et d'appels (`<call>`) ne pose ainsi pas de difficulté. Nous nous concentrons donc seulement sur les contextes de boucle (`<loop>` et `<iteration>`).

Le format XML de FFX étant très verbeux, nous ferons usage dans la suite d'une notation mathématique équivalente, plus concise. Nous noterons ainsi un conflit entre trois arcs e_1, e_2, e_3 par

$$e_1, e_2, e_3$$

Si ce conflit est valable, pour toute itération d'une boucle h, nous le noterons

$$\text{loop}_h^*[e_1, e_2, e_3]$$

l'étoile * désignant toute itération, à l'instar du langage FFX. De même, si seul e_1 est dans la boucle h et que le conflit porte sur la dernière itération de celle-ci, nous le noterons

$$\text{loop}_h^n[e_1], e_2, e_3$$

9. Du fait que ces contextes définissent en réalité des régions sans boucles, c'est même le seul ordonnancement vérifiant cette propriété d'appartenance à un chemin du contexte.

Bien entendu, ces contextes loop^* et loop^n peuvent s'emboîter :

$$\text{loop}^*_{h'}[\text{loop}^n_h[e_1], e_2, \text{loop}^*_{h''}[e_3]]$$

Nous notons l'ensemble des conflits ainsi descriptibles $\mathscr{C}_{\text{FFX}}$.

5.2.3.3 Sémantique des conflits FFX

Nous pouvons définir la sémantique des chemins infaisables exprimés sous la forme de conflits entre arcs dans FFX, en les traduisant en un chemin abstrait de Π^*, incluant des arcs de multiples CFG. Cette traduction s'opèrera par une fonction

$$\tau : \mathscr{C}_{\text{FFX}} \to \Pi^*$$

La fonction τ utilise la propriété de séquentialité des arcs exprimés (attribut `seq` = `"true"`). Elle relie systématiquement les listes d'arcs en intercalant entre chaque couple d'arc un connecteur représentant l'ensemble des chemins entre la cible de l'arc précédent et la source de l'arc suivant. Lorsque la liste d'arcs est dans le contexte d'une boucle – pour toutes itérations ou pour la dernière – la tête de boucle est exclue. Les connecteurs reliant un arc en dehors d'une boucle et un arc à l'intérieur de cette boucle n'excluent pas la tête de boucle (on a le droit de boucler jusqu'à s'"installer" dans le contexte d'une itération). La contrainte "dernière itération" exprimée par loop^n est traduite en ne permettant pas de boucler en sortie de boucle : ainsi, sur le deuxième exemple, le connecteur entre e_1 et e_2 ne permet pas de repasser sur la tête de boucle h.

Exemple. Ainsi, pour tout $e_1 = (s_1 \to t_1)$, $e_2 = (s_2 \to t_2)$, $e_3 = (s_3 \to t_3)$ arcs du programme, et pour toutes têtes de boucle h, h',

- Les conflits hors boucles sont traduits par un chemin constitué des arcs du conflit (dans l'ordre énoncé) reliés par des connecteurs :

$$\tau(e_1, e_2, e_3) = e_1 \cdot {}_{t_1}\chi_{s_2} \cdot e_2 \cdot {}_{t_2}\chi_{s_3} \cdot e_3$$

- Les conflits vrais pour toute itération d'une boucle sont traduits par des chemins ne pouvant passer par la tête de boucle (cela en ferait un chemin inter-itération) :

$$\tau(\text{loop}^*_h[e_1, e_2]) = e_1 \cdot {}_{s_1}\chi^{\{h\}}_{t_2} \cdot e_2$$

- L'entrée dans un contexte de boucle est traduite par deux connecteurs, le premier nous permettant d'itérer à loisir sur la boucle en question jusqu'à atteindre l'itération désirée :

$$\tau(e_3, \text{loop}_h^*[e_1, e_2]) = e_3 \cdot {}_{t_3}\chi_h \cdot {}_h\chi_{s_1}^{\{h\}} \cdot e_1 \cdot {}_{s_1}\chi_{t_2}^{\{h\}} \cdot e_2$$

- La sortie de contexte de dernière itération d'une boucle est traduite par une interdiction de reboucler (de repasser par la tête de boucle) :

$$\tau(e_3, \text{loop}_h^n[e_1], e_2) = e_3 \cdot {}_{t_3}\chi_{s_1} e_1 \cdot {}_{t_1}\chi_{s_2}^{\{h\}} \cdot e_2$$

- Le même raisonnement s'applique pour des boucles imbriquées :

$$\begin{aligned}\tau(\text{loop}_{h'}^*[e_3, \text{loop}_h^*[e_1, e_2]]) &= {}_{h'}\chi_{s_3}^{\{h'\}} \cdot e_3 \cdot {}_{t_3}\chi_h^{\{h'\}} \cdot {}_h\chi_{s_1}^{\{h,h'\}} \cdot e_1 \cdot {}_{s_1}\chi_{t_2}^{\{h,h'\}} \cdot e_2 \\ &= \quad e_3 \cdot {}_{t_3}\chi_h^{\{h'\}} \cdot {}_h\chi_{s_1}^{\{h,h'\}} \cdot e_1 \cdot {}_{s_1}\chi_{t_2}^{\{h,h'\}} \cdot e_2\end{aligned}$$

5.2.4 Minimisation des chemins infaisables

5.2.4.1 Le fractionnement des chemins infaisables détectés

Dans son état actuel, l'algorithme de recherche de chemins infaisables procède par points de rendez-vous sur chaque bloc des CFG d'un programme. Une fois que l'analyse a atteint tous les arcs en entrée du bloc dans le contexte (de boucles) considéré, la routine d'identification de chemins infaisables Ξ est déclenchée, pour chaque arc en sortie du bloc.

Exemple. Considérons le programme décrit sur la Figure 5.5. Une fois que l'analyse atteint e, l'arc *pris* de la condition `x == 0`, la routine Ξ est déclenchée avec un état s_e contenant $2^4 = 16$ chemins différents. Notons $\bar{a}$, $\bar{b}$, $\bar{c}$, $\bar{d}$ les arcs *non pris* correspondant respectivement aux arcs a, b, c, d, alors, ces 16 chemins sont (ou terminent par) :

$$\begin{array}{cccc} a.b.c.d.e & a.b.c.\bar{d}.e & a.b.\bar{c}.d.e & a.b.\bar{c}.\bar{d}.e \\ a.\bar{b}.c.d.e & a.\bar{b}.c.\bar{d}.e & a.\bar{b}.\bar{c}.d.e & a.\bar{b}.\bar{c}.\bar{d}.e \\ \bar{a}.b.c.d.e & \bar{a}.b.c.\bar{d}.e & \bar{a}.b.\bar{c}.d.e & \bar{a}.b.\bar{c}.\bar{d}.e \\ \bar{a}.\bar{b}.c.d.e & \bar{a}.\bar{b}.c.\bar{d}.e & \bar{a}.\bar{b}.\bar{c}.d.e & \bar{a}.\bar{b}.\bar{c}.\bar{d}.e \end{array}$$

Les 16 appels subséquents au solveur SMT, vérifiant la satisfiablité des états abs-

traits attachés à chacun de ses chemins, nous apprendrons que les 8 premiers chemins sont infaisables, et ceux-ci seront ajoutés à la liste des propriétés découvertes par l'analyse et inscrites dans un fichier FFX en sortie. Or, ces 8 chemins infaisables expriment en réalité un seul et même conflit, entre les arcs a et e.

Cette méthode est *inefficace*. Ce phénomène de fractionnement des chemins infaisables en raison de divergences des chemins entre les arcs en conflits dans le CFG est fréquent pour des programmes typiques. C'est ainsi que, par exemple, sur un des programmes testés, PathFinder a détecté 118 chemins infaisables alors qu'il n'y avait en réalité que 5 conflits. Sur un autre, ce sont plus de 100.000 chemins infaisables qui ont été détectés, pour un nombre de conflits estimé à 10.000, mais probablement plutôt de l'ordre de la centaine.

Les conséquences d'un tel fractionnement sont l'explosion de la complexité de l'utilisation des chemins infaisables détectés. Par exemple, la complexité de résolution d'un système ILP augmente rapidement avec le nombre de contraintes. Ainsi, si, pour l'amélioration du calcul du WCET d'un programme, les chemins infaisables sont traduits en contraintes ILP et injectés dans le système calculant le WCET, alors le temps de résolution du système augmente excessivement, et le calcul pourra échouer. Ce fut le cas par exemple pour le calcul du WCET sur le benchmark `fft1` (de la suite Mälardalen), dans lequel nous avions injecté des contraintes ILP traduisant 4.999 chemins infaisables.

Il faut donc trouver une solution pour détecter des conflits entre plusieurs arcs, comme a et e pour l'exemple ci-dessus, plutôt que de lister tous les chemins infaisables complets incluant ces deux arcs.

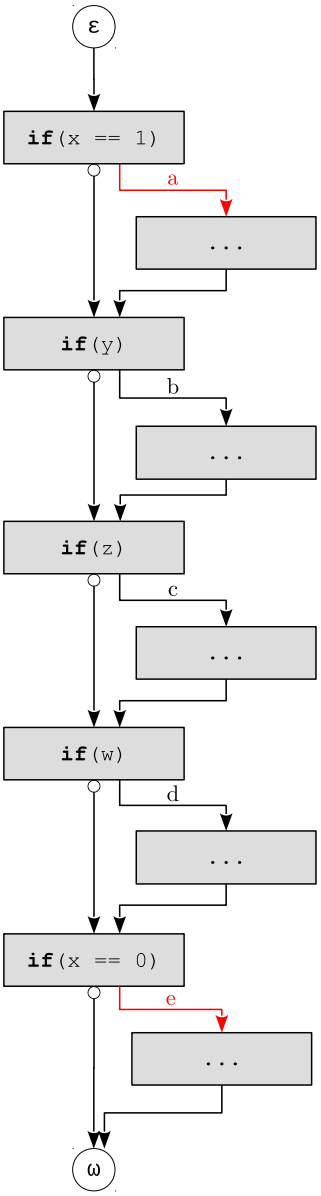


Figure 5.5 – Conflit entre deux arcs éloignés

5.2.4.2 Identification d'ensembles d'arcs en conflit

Une fois la faisabilité de l'ensemble des chemins déterminée, il est aisé de tester des hypothèses. Ainsi, sur l'exemple précédent, après avoir déterminé – à l'aide d'un

solveur SMT – que les 8 chemins

$$\begin{array}{cccc} a.b.c.d.e & a.b.c.\bar{d}.e & a.b.\bar{c}.d.e & a.b.\bar{c}.\bar{d}.e \\ a.\bar{b}.c.d.e & a.\bar{b}.c.\bar{d}.e & a.\bar{b}.\bar{c}.d.e & a.\bar{b}.\bar{c}.\bar{d}.e \end{array}$$

sont infaisables et que les 8 chemins

$$\begin{array}{cccc} \bar{a}.b.c.d.e & \bar{a}.b.c.\bar{d}.e & \bar{a}.b.\bar{c}.d.e & \bar{a}.b.\bar{c}.\bar{d}.e \\ \bar{a}.\bar{b}.c.d.e & \bar{a}.\bar{b}.c.\bar{d}.e & \bar{a}.\bar{b}.\bar{c}.d.e & \bar{a}.\bar{b}.\bar{c}.\bar{d}.e \end{array}$$

sont faisables, il est rapide de vérifier que les 8 chemins infaisables contiennent les arcs $\{a, e\}$ (au contraire des 8 chemins faisables). Cela garantit que les arcs $\{a, e\}$ sont en conflit, pour tout chemin dans le contexte considéré (on peut aussi prouver que cela suffit à représenter l'intégralité des chemins infaisables détectés).

La difficulté est dans *la formulation des hypothèses*. Une piste de résolution tentante serait de chercher à "fusionner" des chemins infaisables entre eux, par exemple par une sorte d'opération $a.b.c.d.e + a.\bar{b}.c.d.e = a.c.d.e$, qui n'est pas sans rappeler celles de la logique booléenne. Malheureusement, la définition d'une telle opération de "fusion" est trop complexe dans le cas général, parce qu'il n'existe simplement pas d'arc opposé "$\bar{a}$" pour tout arc conditionnel a – ce raisonnement ne fonctionne qu'avec des CFG dits "en diamant" comme celui induit par le programme de la Figure 5.5.

Une autre solution doit donc être trouvée, et les solveurs SMT peuvent nous y aider.

5.2.4.3 Extraction d'ensembles d'arcs en conflit à partir de sous-ensembles minimaux insatisfiables

La section 5.1.3.2 a présenté l'extraction d'*unsat cores*, capacité de certains solveurs SMT à donner un sous-ensemble minimal insatisfiable de contraintes. Nous avons fait un critère de choix de solveur de cette fonctionnalité, que CVC4 et Z3 supportent.

Or, il est possible d'améliorer l'analyse d'interprétation du programme pour se souvenir des points du programme (arcs) *responsables* d'une variable, points affectant la valeur d'une variable. Pour cela, nous attachons à chaque prédicat de $\breve{P}_\Lambda$ et à chaque élément de la table de variables $\breve{\Theta}$ un ensemble d'arcs du CFG $\Sigma \in \mathcal{P}(E)$ responsables.

$\hat{\mathbb{I}}^+$ est la fonction d'interprétation définie à partir de $\hat{\mathbb{I}}$ qui met à jour, pour chaque expression dans $\breve{\Theta}$ et à chaque prédicat de $\breve{\mathbb{E}}_\Lambda$, la liste des arcs responsables qui lui est

associée. Notons $A_i \subseteq \mathbb{V}^\sharp$ (resp. $B_i \subseteq \mathbb{V}^\sharp$) est l'ensemble des variables utilisées (resp. modifiées) pour l'interprétation de i. Alors, pour l'interprétation d'une instruction $i \in I_{sem}$ sur un arc [10] e, si $(\check{\theta}_{\hat{\mathbb{I}}}, p_{\hat{\mathbb{I}}})$ est le résultat de l'interprétation classique par $\hat{\mathbb{I}}$ sur $(\check{\theta}, p)$, c'est-à-dire

$$(\check{\theta}_{\hat{\mathbb{I}}}, p_{\hat{\mathbb{I}}}) \coloneqq \hat{\mathbb{I}}[i](\check{\theta}, p)$$

alors, le résultat pour chaque variable $v \in \mathbb{V}^\sharp$, le Σ_v attaché à l'expression $x^{(\Sigma_v)} \coloneqq \check{\theta}(v)$ de la table des variables

- reste inchangé si l'instruction i n'affecte pas v ($v \notin B_i$) :

$$x_{\hat{\mathbb{I}}}{}^{(\Sigma_v)} \quad \text{avec } x_{\hat{\mathbb{I}}} \coloneqq \check{\theta}_{\hat{\mathbb{I}}}(v)$$

- devient l'ensemble des $\Sigma_{v'}$ attachés aux expressions de chaque variable v' lue par l'instruction i ($v' \in A_i$) (car le résultat de l'instruction dépend de la valeur de ses opérandes, et donc hérite des dépendances des opérandes en question), auquel on ajoute l'arc courant e (qui est responsable pour l'exécution de i), si l'instruction i affecte v ($v \in B_i$) :

$$x_{\hat{\mathbb{I}}}{}^{(\Sigma_{A_i} \cup \{e\})} \quad \begin{array}{l} \text{avec } x_{\hat{\mathbb{I}}} \coloneqq \check{\theta}_{\hat{\mathbb{I}}}(v) \\ \text{avec } \Sigma_{A_i} \coloneqq \bigcup\limits_{v' \in A_i} \left\{ \Sigma_{v'} \;\middle|\; \exists x,\, \check{\theta}(v') = x^{(\Sigma_{v'})} \right\} \end{array}$$

De plus, le Σ attaché à chaque prédicat $p_i^{(\Sigma)}$ reste inchangé (puisqu'ils sont immuables, et donc eux-mêmes inchangés) :

$$p_i^{(\Sigma)}$$

Comme pour les éléments de $\check{\theta}_{\hat{\mathbb{I}}}$, chaque nouveau prédicat dans $p_k \in p_{\hat{\mathbb{I}}}$ ($p_k \notin p$) est associé à l'ensemble des Σ attachés aux expressions de chaque variable lue par l'instruction i (A_i), auquel on ajoute l'arc e :

$$p_k{}^{(\Sigma_{A_i} \cup \{e\})} \quad \text{avec } \Sigma_{A_i} \coloneqq \bigcup_{v' \in A_i} \left\{ \Sigma_{v'} \;\middle|\; \exists x,\, \check{\theta}(v') = x^{(\Sigma_{v'})} \right\}$$

10. Pour les instructions sur un bloc, on insère en réalité un marqueur qui sera remplacé par le prochain arc pris.

Définition 5.5. La fonction d'interprétation $\hat{\mathbb{I}}^+$ redéfinie pour mettre à jour la liste des arcs responsables à chaque expression et prédicat d'un état, est ainsi définie, pour l'interprétation d'une instruction $i \in I_{sem}$ sur un arc e :

$$\forall(\check{\theta}, p) \in \check{S},\ (\check{\theta}_{\hat{\mathbb{I}}}, p_{\hat{\mathbb{I}}}) = \hat{\mathbb{I}}[i](\check{\theta}, p) \Longrightarrow \hat{\mathbb{I}}^+[i](\check{\theta}, p) \coloneqq$$
$$\left(\begin{array}{l} v \mapsto \begin{cases} {x_0}^{(\Sigma_v)} & \text{si } v \notin B_i, \text{ avec } x_{\hat{\mathbb{I}}} \coloneqq \check{\theta}_{\hat{\mathbb{I}}}(v) \\ {x_0}^{(\Sigma_{A_i} \cup \{e\})} & \text{si } v \in B_i, \text{ avec } \Sigma_{A_i} \coloneqq \bigcup_{v' \in A_i} \left\{ \Sigma_{v'} \;\middle|\; \exists x,\, \check{\theta}(v') = x^{(\Sigma_{v'})} \right\} \end{cases} \\ p \cup \left\{ {p_k}^{(\Sigma_{A_i} \cup \{e\})} \;\middle|\; \begin{array}{l} p_k \in p_0 \\ p_k^{(\Sigma)} \notin p, \end{array} \text{ avec } \Sigma_{A_i} \coloneqq \bigcup_{v' \in A_i} \left\{ \Sigma_{v'} \;\middle|\; \exists x,\, \check{\theta}(v') = x^{(\Sigma_{v'})} \right\} \right\} \end{array} \right)$$

où $A_i \subseteq \mathbb{V}^\sharp$ (resp. $B_i \subseteq \mathbb{V}^\sharp$) est l'ensemble des variables utilisées (resp. modifiées) pour l'interprétation de i, et où Σ_v est tel que $\exists x,\, x^{(\Sigma_v)} \coloneqq \check{\theta}(v)$ et $x_{\hat{\mathbb{I}}} = \check{\theta}_0(v)$.

Exemple.

$$\hat{\mathbb{I}}^+[\texttt{add}\ x, y, z] \left(\begin{array}{|c|c|c|} \hline & x & x^* + 1\ ^{(e_3)} \\ \theta & y & 2\ ^{(e_1, e_2)} \\ & z & {x^*}\ ^{(e_2, e_4)} \\ \hline & \multicolumn{2}{c|}{\ldots} \\ \hline \end{array} \right) = \left(\begin{array}{|c|c|c|} \hline & x & 2 + {x^*}\ ^{(e_1, e_2, e_4)} \\ \theta & y & 2\ ^{(e_1, e_2)} \\ & z & {x^*}\ ^{(e_2, e_4)} \\ \hline & \multicolumn{2}{c|}{\ldots} \\ \hline \end{array} \right)$$

Une fois chaque expression et prédicat attaché à une liste d'arcs dont il dépend, nous utilisons la fonctionnalité d'extraction d'*unsat core* pour déterminer un ensemble minimaux de contraintes (prédicats SMT) causant l'insatisfiabilité de l'état abstrait, et remonter à la liste d'arcs responsables. Ainsi, pour un état $\check{s}$ insatisfiable

$\check{\mathbf{s}}$		
θ	x	$x^* + 1\ ^{(e_3)}$
	y	$2\ ^{(e_1, e_2)}$
	z	${x^*}\ ^{(e_2, e_4)}$
	$\ldots$	
p	$x^* = {z^*}\ ^{(e_5)}$	

traduit en prédicats SMT comme suit :

$$t(\check{s}) = \{x = x^* + 1,\ y = 2,\ z = x^*,\ x^* = z^*\}$$

nous obtiendrons l'*unsat core* composé de ces trois prédicats SMT :

$$\{x = x^* + 1,\ z = x^*,\ x^* = z^*\}$$

et déduirons e_3, e_2, e_4, e_5 comme arcs responsables. Nous noterons [11]

$$\text{core}(\check{s}) = \{e_3, e_2, e_4, e_5\}$$

Cet ensemble d'arcs fournit une très bonne hypothèse d'arcs en conflit pour remplacer les chemins complets associés aux états abstraits.

5.2.4.4 Le problème des effets de bords

La technique de minimisation de chemins infaisables en un ensemble d'arcs en conflit ne reste qu'une hypothèse, à valider selon la méthode vue en section 5.2.4.2. En effet, sur certains chemins infaisables, des arcs nécessaires au conflit ne modifient pas directement de variable clé à l'insatisfiabilité de l'état représentant le chemin, mais permettent au lieu de cela d'éviter un chemin qui modifierait cette variable. Ce phénomène est appelé un *effet de bord*.

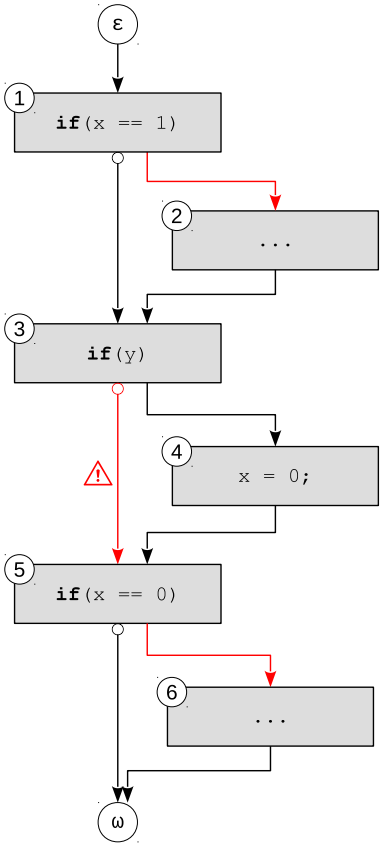


Figure 5.6 – Exemple d'effet de bord

Exemple. Considérons le CFG de la Figure 5.6. Le chemin ① → ② → ③ → ⑤ → ⑥ est infaisable, et ; en particulier, les arcs ① → ②, ③ → ⑤, ⑤ → ⑥ sont nécessaires et suffisants pour former un conflit. L'arc ③ → ⑤ est nécessaire à ce conflit car un chemin ne passant pas par cet arc passerait nécessairement par ③ → ④ et affecterait 0 à x.

En revanche, l'*unsat core* qui serait extrait par l'analyse pointerait seulement vers l'arc ① → ② comme responsable d'un prédicat $x^* = 1$ et vers l'arc ⑤ → ⑥ comme responsable d'un prédicat $x^* = 0$. L'arc ③ → ⑤ n'étant pas inclus, l'ensemble d'arcs

11. Par convention et par souci de complétude, la fonction core donnera $\emptyset$ pour tout ensemble satisfiable.

① → ②, ⑤ → ⑥ n'est pas réellement en conflit. En effet, il inclut le chemin ① → ② → ③ → ④ → ⑤ → ⑥, qui n'est pas un chemin infaisable valide !

En revanche, cette technique est vraisemblablement capable d'identifier le couple d'arcs en conflit ④ → ⑤, ⑤ → ⓦ.

5.2.4.5 D'un ensemble d'arcs en conflits à un conflit FFX

Une fois un ensemble d'arcs en conflits minimaux obtenus grâce à l'extraction d'*unsat core*, la traduction vers les conflits FFX est immédiate : il suffit de

- réordonner les arcs suivant le même ordre que dans le chemin complet correspondant au conflit ;
- placer autour de chaque séquence d'arcs appartenant à une boucle, des contextes de boucles
  - valides pour la dernière itération si ce contexte est suivi par des arcs hors de la boucle,
  - valides pour toute itération sinon.

Formellement, cette fonction est ainsi définie :

Définition 5.6. Soit $\mathcal{X} : \mathcal{P}(E) \times \Pi \to \mathscr{C}_{\text{FFX}}$ la fonction de traduction d'un ensemble d'arcs en conflit vers un conflit FFX, à l'aide du chemin complet associé. Soit $\leq_\pi$ la relation d'ordre partiel induite par l'ordre des arcs dans un chemin π :

$$\forall e_1, e_2 \in E,\ e_1 \leq_\pi e_2 \iff \exists \pi' \in \Pi,\ e_i.\pi'.e_{i+1} \text{ est un sous-chemin de } \pi$$

Alors, si H est l'ensemble des têtes de boucles du programme, et $L(h)$ dénote

(comme au Chapitre 4) l'ensemble des blocs à l'intérieur d'une boucle h,

$$\forall j \in \mathcal{P}(E) \setminus \{\emptyset\},\ \forall \pi \in \Pi,\ \mathcal{X}(j,\pi) \coloneqq$$
$$\begin{cases} e & \text{si } j = \{e\},\ e \in E \\ e_0 \,.\, \mathcal{X}(j \setminus \{e_0\}, \pi) & \text{si } \forall h \in H,\ L(h) \cap j = \emptyset, \\ & \text{pour } e_0 \text{ tq } \forall e \in j,\ e_0 \leq_\pi e \\ \mathcal{X}(\{e \in j \mid e \leq_\pi h_0\} & \text{sinon, pour } h_0 \text{ tq } \begin{array}{l} L(h_0) \cap j \neq \emptyset \\ \forall h \in H,\ h_0 \notin L(h) \end{array} \\ \quad . \mathbf{loop^*_h}\Big[\mathcal{X}(\{e \in j \mid e \in L(h_0)\}, \pi)\Big] & \quad \text{si } \nexists e \in j,\ h_0 \leq_\pi e \\ \mathcal{X}(\{e \in j \mid e \leq_\pi h_0\} & \text{sinon, pour } h_0 \text{ tq } \begin{array}{l} L(h_0) \cap j \neq \emptyset \\ \forall h \in H,\ h_0 \notin L(h) \end{array} \\ \quad . \mathbf{loop^n_h}\Big[\mathcal{X}(\{e \in j \mid e \in L(h_0)\}, \pi)\Big] & \\ \quad . \mathcal{X}(\{e \in j \mid h_0 \leq_\pi e\} & \quad \text{si } \exists e \in j,\ h_0 \leq_\pi e \end{cases}$$

Exemple. Soit $\pi = e_1.e_2.e_3.e_4.e_5$ et $\{e_2, e_3, e_4\} \in L(h_1)$, $\{e_4\} \in L(h_2)$. Alors,

$$\begin{aligned} \mathcal{X}(\{e_1, e_5\}, \pi) &= e_1 \,.\, e_5 \\ \mathcal{X}(\{e_1, e_2, e_3\}, \pi) &= e_1 \,.\, \mathrm{loop}^*_{h_1}[e_2 \,.\, e_3] \\ \mathcal{X}(\{e_1, e_2, e_3, e_4\}, \pi) &= e_1 \,.\, \mathrm{loop}^*_{h_1}[e_2 \,.\, e_3 \,.\, \mathrm{loop}^*_{h_2}[e_4]] \\ \mathcal{X}(\{e_1, e_2, e_3, e_4, e_5\}, \pi) &= e_1 \,.\, \mathrm{loop}^n_{h_1}[e_2 \,.\, e_3 \,.\, \mathrm{loop}^n_{h_2}[e_4]] \,.\, e_5 \end{aligned}$$

Nous pouvons maintenant définir l'algorithme final de la routine Ξ.

5.2.5 Modification de la routine Ξ pour la recherche de conflits

Nous adaptons l'Algorithme 8 en appliquant les changements suivants :

- Le test de satisfiabilité est fait pour chaque couple d'état / prédicat au préalable, ainsi que l'extraction de l'*unsat core* (si l'état est détecté insatisfiable).
- Pour tout état insatisfiable détecté, nous vérifions que les arcs énumérés par l'*unsat core* j suffisent à rendre un chemin infaisable (pour pallier aux problèmes d'effets de bord). Cette validation se fait en vérifiant l'infaisabilité de tout chemin π' contenant les arcs de l'*unsat core* j.

- Si l'ensemble de conflits induit par l'*unsat core* j est valide, on le rajoute ; sinon, on rajoute le chemin infaisable complet comme (volumineuse) contrainte.

Le résultat est l'Algorithme 9.

Algorithme 9 : Routine Ξ : détection de chemins infaisables sur un arc e avec minimisation

Données : s_e, l'état sur l'arc e ; Γ, contexte d'analyse des boucles englobantes
Résultat : s_e, l'état sur l'arc e modifié ; ips, les chemins infaisables détectés

$ips \leftarrow \{\}$
si $\forall (h, q_h) \in \Gamma, q_h = \text{LEAVE}$ **alors**
  $w \leftarrow \{(\pi, \text{sat}(\check{s}), \text{core}(\check{s})) \mid (\check{s}, \pi) \in s_e\}$
  pour $(\pi, b, j) \in w$ **faire**
    si $b = \bot$ **alors**
      si $\forall (\pi', b') \in w,\ (j \subseteq \pi') \Rightarrow (b' = \bot)$ **alors**
        $ips \leftarrow ips \cup \{\mathcal{X}(j)\}$
      sinon
        $ips \leftarrow ips \cup \{\pi\}$
      $s_e \leftarrow s_e \setminus \{(\check{s}, \pi) \mid \pi' = \pi\}$

5.2.6 Discussion

5.2.6.1 Optimisation

Cet algorithme peut encore être amélioré de plusieurs façons.

Tout d'abord, nous pouvons remarquer que les chemins infaisables n'apparaissent qu'au niveau des branchements conditionnels : l'exécution d'une séquence d'instructions ne créant pas de chemins, nous ne découvrons pas de nouveaux chemins infaisables lors de son interprétation. Il y a deux parties de l'interprétation abstraite du programme qui restreignent l'espace des états possibles :

- les branchements conditionnels, introduisant une nouvelle contrainte (via l'instruction sémantique `assume`) ;
- la composition de fonctions, puisqu'elle restreint les propriétés issues de l'exécution d'une fonction à un contexte d'appel en particulier.

Il est donc possible d'économiser, sans perte de résultats, des tests de satisfiabilité (et des appels coûteux au solveur SMT) en n'effectuant ces tests qu'après un branchement conditionnel (plusieurs arcs en sortie de bloc), ou après la composition d'états issue d'un appel de fonction, plutôt que de les faire sur chaque arc.

Lorsque la minimisation d'un chemin infaisable par extraction d'*unsat core* échoue à formuler une hypothèse valide d'un ensemble d'arcs en conflits, nous pouvons malgré tout chercher à réduire le chemin infaisable complet π. Notamment, si parmi les arcs de π, un arc e_1 domine un arc e_2, alors e_1 est superflu et peut être retiré du conflit d'arcs induit par π. De même, e_1 est superflu si e_1 post-domine e_2.

Aussi, la complexité des solveurs SMT ayant tendance à augmenter rapidement avec le nombre de variables et de contraintes, il s'avère bénéfique de ne pas fournir comme assertion dans le solveur les prédicats SMT n'ayant aucune influence sur les autres prédicats. De plus, la construction de l'arbre syntaxique abstrait représentant les prédicats entrés dans de tels solveurs est coûteuse pour des problèmes de petites tailles (mais fréquemment posés). Par exemple, avec CVC4, pour une simple contradiction arithmétique entre deux entiers, le temps pris pour la résolution du problème est estimé à $17\mu s$ et le temps pour la construction de sa représentation interne à $741\mu s$. En rajoutant 100 prédicats de la forme $x =$ constante (sans aucune variable commune, ne pouvant donc pas être en conflit les uns avec les autres), le temps de résolution passe à $46\mu s$ et le temps pour la construction de la représentation interne à $3770\mu s$.

Il peut aussi être noté que les appels au solveur SMT par l'Algorithme 9 se faisant par vagues (autant d'appels que de chemins), et que ceux-ci sont indépendants, cette partie de l'analyse se parallélise bien : la version parallèle de l'interface de PathFinder avec CVC4 donne de bons résultats, avec une accélération de 3 à 4 fois sur 8 cœurs.

Par ailleurs, l'analyse peut être étendue pour l'analyse des premières itérations de boucles sans difficulté majeure. Il suffirait de dérouler une fois chaque boucle – en fait, d'appliquer les états paramétriques issus de l'interprétation du corps de la boucle à l'état en entrée pour obtenir l'état après une itération, et de tester la satisfiabilité des états de chaque chemin. Il faudrait ensuite utiliser un troisième type de contexte de boucle (loop) dans la syntaxe des conflits, et de le traduire par l'attribut FFX correspondant.

5.2.6.2 Limitation à $\mathbb{Z}_{32}$

Enfin, les techniques de résolution de problème SMT utilisées par le biais de solveur fonctionnent pour des entiers, et non sur le groupe des entiers sur 32 bits, sur lequel des prédicats comme $x \times 16 = 0$ sont satisfiables. Il existe différentes solutions pour éviter des faux positifs dans la recherche d'états insatisfiables et ainsi préserver la validité de l'analyse.

Nous pouvons utiliser les théories SMT de vecteurs de bit (*bit vector*). Bien que ce soit un moyen précis de représenter des variables, qui donne même la possibilité d'ajouter (utilement) des opérateurs logiques dans les prédicats, cette représentation est rapidement extrêmement inefficace pour la résolution de problèmes arithmétiques sur 32 bits, prenant parfois plus d'une journée pour un seul problème.

Au lieu de cela, nous pouvons aussi encapsuler chaque variable des prédicats SMT par une opération modulo : remplacer toutes les variables [12] v par $v \bmod 2^{32}$. Malheureusement, cela casse la linéarité des équations et nous fait perdre la grande majorité des résultats (bien que la validité de l'analyse soit préservée).

Stein et Martin [98] proposent une solution à ce problème de perte de linéarité. Ainsi, en plus de la transformation précédente, chaque contrainte c contenant un terme $x \bmod k$ est remplacée par les deux contraintes suivantes

$$\begin{cases} k\lambda_c \leq x < k(\lambda_c + 1) \\ c[x \bmod k \mapsto x - k\lambda_c] \end{cases}$$

où λ_c est une variable arbitraire introduite dans le système de contraintes SMT, pour c, et $c[x \bmod k \mapsto x - k\lambda_c]$ est cette même contrainte où toute occurrence de $x \bmod k$ est remplacée par $x - k\lambda_c$. Ce processus peut être répété à loisir en présence de plusieurs modulos, et permet finalement de détecter correctement les problèmes satisfiables dans $\mathbb{Z}_{32}$ au lieu de $\mathbb{Z}$.

12. Ou plutôt toutes les expressions arithmétiques linéaires, puisque $\forall a, b \in \mathbb{Z},\ (a + b) \bmod \mathbb{Z}_{32} = (a \bmod \mathbb{Z}_{32}) + (b \bmod \mathbb{Z}_{32})$.

5.3 Applications

Cette section présente les résultats expérimentaux obtenus avec notre outil, qui intègre l'ensemble des méthodes développées dans cette thèse.

5.3.1 Exploitation des chemins infaisables pour la réduction du WCET

Nous avons appliqué les méthodes de Raymond [86] pour la traduction de chemins infaisables en contraintes ILP, de manière à influencer le calcul du WCET en les excluant des chemins d'exécution considérés.

Ainsi, par exemple, un chemin infaisable (ou plutôt, une famille de chemins infaisables factorisée) exprimé sous la forme d'une contrainte FFX $e_1.e_2.e_3$ peut être traduit par la contrainte ILP

$$x_{e_1} + x_{e_2} + x_{e_3} < 3$$

où $x_{e_1}, x_{e_2}, x_{e_3}$ représentent respectivement le nombre d'exécutions de e_1, e_2, e_3. Puisque le conflit indique que nous sommes au niveau séquentiel, en dehors de toute boucle, ces coefficients x_e sont 0 ou 1, et la contrainte < 3 indique qu'ils ne peuvent pas être pris tous à la fois.

Pour un conflit $\text{loop}_h^*[e_1.e_2.e_3]$ dont les arcs sont contenus dans une boucle h, nous pouvons ajouter la contrainte ILP

$$x_{e_1} + x_{e_2} + x_{e_3} < 3n_h$$

où n_h représente la borne de la boucle h. Cette fois-ci, les valeurs prises par les coefficients x_e sont logiquement entre 0 et n_h. Cette contrainte est valide mais perd en expressivité par rapport au conflit $\text{loop}_h^*[e_1.e_2.e_3]$: elle autorise par exemple les trois arcs à être pris dans une même itération, à condition qu'une itération "compense" en prenant strictement moins de deux arcs.

Nous ne détaillons pas ici les cas plus complexes de boucles imbriquées et de conflits entre arcs à l'extérieur et à l'intérieur d'une boucle. Toutefois, ce simple cas de conflit entre arcs dans une unique boucle montre déjà les pertes de précision induites par l'expression de chemins infaisables sous la forme de contraintes ILP. Ces pertes de précision ne mettent toutefois pas en danger la sûreté du WCET calculé, qui est toujours

une majoration du WCET réel. En revanche, elles peuvent conduire à l'obtention d'un WCET surestimé correspondant à un WCEP infaisable pourtant exclu par les conflits FFX.

D'autres techniques existent pour l'exploitation des chemins infaisables afin d'améliorer la précision du WCET. En particulier, les méthodes développées par Mussot [73] pour l'expression de tels conflits sous la forme d'automates ont été développées dans un outil capable de traiter les conflits FFX générés par PathFinder. Elles permettent de représenter plus précisément les informations exprimées par les conflits en question, et ont ainsi permis d'améliorer leur impact sur le WCET [75] par rapport à l'intégration des conflits par injection de contraintes ILP. Toutefois, et en partie en raison de limitations en complexité de ces méthodes, les résultats présentés dans cette section feront état de l'amélioration de l'estimation du WCET par ajout de contraintes ILP.

5.3.2 Résultats expérimentaux

5.3.2.1 Benchmarks utilisés

Nous avons principalement testé notre outil de recherche de chemins infaisables sur trois suites de benchmarks :

- La suite de benchmarks de Mälardalen [45], classique pour l'évaluation d'outils dans le domaine du WCET. Cette suite contient une large variété de programmes C, avec de nombreuses boucles imbriquées, ainsi que des opérations de manipulation de bit, de calcul de matrices, de calcul flottant émulé, et un programme constitué d'un volume important de code généré (`nsichneu`) et de conditions imbriquées.

- Les 6 tâches du benchmark *debie1*, basé sur le logiciel de surveillance des débris spatiaux DEBIE-1 [13], écrit en C et développé par *Space Systems Finland Ltd* sous contrat de l'Agence Spatiale Européenne (ESA).

- Les 13 tâches du benchmark *PapaBench*, issu du contrôleur de drone Paparazzi développé à l'ENAC à Toulouse. Ce logiciel est composé de deux programmes : l'un (`fly-by-wire`) est responsable du contrôle des capteurs et de la stabilisation

13. `http://space-env.esa.int/index.php/debie-1.html`

du vol et l'autre (`autopilot`), plus complexe, est chargé de déterminer un plan de vol.

Les programmes sont compilés avec GCC, version 4.7.2, avec le niveau d'optimisation `-O1` et `arm-eabi` pour architecture cible.

Certains des programmes analysés, comme `cover` des Mälardalen qui contient des `switch/case`, ont nécessité une analyse de branchements indirects (cf. section 3.1.2.5). Par ailleurs, nous avons ignoré les programmes récursifs, comme `recursion` des Mälardalen, notre outil ne supportant pas la récursion à ce jour, bien qu'une extension de l'analyse pour la gestion de boucles récursives ne semble pas poser de difficulté majeure.

5.3.2.2 Résultats et impact sur le WCET

La Table 5.2 présente l'ensemble des résultats obtenus pour une implémentation des méthodes présentées (analyse sur $\breve{S}$) dans PathFinder. Les quatre premières colonnes donnent des informations sur le programme ou la tâche considérée. La deuxième colonne et la troisième colonne donnent respectivement le nombre total de blocs de base et d'instructions dans les CFG du benchmark considéré, et la quatrième colonne le nombre de boucles et leur profondeur maximale. Ce sont de bons indicateurs de la complexité d'un benchmark.

Les deux colonnes suivantes évaluent la complexité de l'analyse, en affichant le temps d'exécution pour le benchmark considéré et le ratio de temps passé dans la résolution de problèmes SMT. La machine utilisée pour les expérimentations est équipée d'un processeur Dual core Intel Core i7-6500U 2.50GHz et de 16Go de mémoire vive. Le solveur utilisé pour les résultats dans ce tableau est CVC4 1.4. Les résultats obtenus avec Z3 sont très comparables et ne sont pas présentés ici : temps d'exécution similaires à $\pm 10\%$ près, dans l'ensemble légèrement meilleurs pour Z3, et résultats identiques à l'exception de deux chemins infaisables sur un programme issus d'un conflit entre contraintes non-linéaires, détectés exclusivement par Z3.

Les deux colonnes suivantes donnent le nombre de conflits obtenus, respectivement avec et sans utilisation des *unsat core* pour la factorisation et minimisation des chemins infaisables. Ces deux colonnes soulignent la faible pertinence d'une telle métrique (en nombre de conflits) pour l'évaluation d'une analyse de recherche de chemins infaisables : un nombre de conflits élevé peut être un signe positif, signifiant un grand nombre de chemins infaisables détectés, ou à l'inverse un signe négatif, conséquence d'une mauvaise

Benchmark / Tâche	BB	Inst.	Boucles (prof.)	Temps (s)	SMT %	CI (sans min)	CI (avec min)	Gain WCET
Très petits benchmarks Mälardalen (moins de 30 BB)								
fibcall, insertsort, fdct, bs, jfdctint, janne_complex, ns, duff, bsort100, lcdnum, matmult, crc, fir : *aucun chemin infaisable trouvé*								
Petits benchmarks Mälardalen (sans réduction d'états, $\eta = +\infty$)								
prime	43	193	14(2)	1,491	98%	50	50	27,452%
expint	43	251	12(2)	0,744	98%	56	20	6,553%
select	56	321	4(3)	20,825	88%	2106	1178	*nul*
qsort-exam	58	369	6(3)	24,488	99%	2096	1168	*nul*
edn	70	1123	18(3)	0,428	92%	4	4	*nul*
ndes	83	796	12(2)	2,187	98%	140	140	*nul*
cnt	94	511	7(2)	42,469	99%	1040	1040	*nul*
compress	109	728	17(2)	121,387	96%	23	23	*nul*
cover	205	822	3(1)	0,504	96%	3	3	3,059%
Gros benchmarks Mälardalen (avec réduction d'états, $\eta = 250$)								
ud	122	802	20(3)	14,630	97%	231	57	*nul*
adpcm	171	1797	33(3)	10,236	99%	11	11	0,002%
qurt	245	1390	111(2)	44,929	98%	894	780	15,165%
sqrt	272	1236	11(2)	479,508	99%	6591	1845	10,479%
ludcmp	276	1669	35(4)	215,110	98%	1741	784	*nul*
minver	286	1730	35(4)	140,115	99%	1584	405	3,363%
fft1	337	1882	216(4)	3693,147	93%	134008	11596	14,607%
statemate	387	2530	1(1)	250,279	99%	6433	5458	1,297%
lms	527	2582	116(3)	657,364	97%	1396	615	2,177%
nsichneu	756	8088	1(1)	261,522	74%	19415	19145	*nul*
Benchmarks Debie1								
TM_InterruptService	19	96	0(0)	0,066	98%	0	0	-
HandleHitTrigger	64	291	3(2)	0,969	99%	15	11	41,959%
TC_InterruptService	69	276	0(0)	2,191	98%	30	30	*nul*
HandleTelecommand	190	768	13(2)	3,856	85%	144	141	*nul*
HandleAcquisition	303	1321	19(1)	14,095	99%	126	126	0,156%
HandleHealthMonitoring	425	1670	100(3)	230,381	98%	5094	4538	30,746%
Benchmarks PapaBench (programme fly-by-wire)								
servo_transmit	13	54	1(1)	0,121	99%	10	4	*nul*
send_data_to_autopilot	121	562	10(1)	11,667	98%	6	6	*nul*
check_failsafe	148	639	24(1)	17,695	93%	114	114	*nul*
check_mega128_values	150	655	24(1)	15,629	93%	114	114	*nul*
test_ppm	270	1170	36(1)	36,438	96%	518	518	0,247%
Benchmarks PapaBench (programme autopilot)								
link_fbw_send	3	32	0(0)	0,005	100%	0	0	-
altitude_control	96	388	2(1)	4,673	98%	49	49	*nul*
stabilisation	200	859	13(1)	8,844	97%	239	239	0,534%
climb_control	252	1066	15(1)	23,751	96%	320	320	0,460%
reporting	418	3943	0(0)	70,716	99%	2879	2879	*nul*
radio_control	424	2398	39(1)	71,345	98%	2803	2803	*nul*
receive_gps_data	574	3310	57(1)	89,189	98%	2618	2590	2,590%
navigation	1004	4643	1018(1)	409,404	94%	3273	2899	*nul*

Table 5.2 – Résultats expérimentaux et gains conséquents sur l'estimation du WCET

factorisation des chemins infaisables exprimés, qui se traduit par des conflits nombreux et des propriétés exprimées faibles.

Enfin, la dernière colonne révèle l'impact sur l'estimation du WCET [14] des chemins infaisables détectés, en terme de pourcentage de réduction du WCET obtenu par rapport à une analyse de WCET ne prenant pas en compte l'existence de ces chemins infaisables. En partie grâce au processus de minimisation, l'intégralité des propriétés de flot résultant de l'analyse de chemins infaisables ont pu être intégrés dans le calcul du WCET par ILP, à l'exception de quelques benchmarks (`qurt`...) pour lesquels la complexité du problème ILP engendré nous ont forcé à ne considérer qu'une partie arbitraire des chemins infaisables détectés.

Treize des benchmarks de Mälardalen sont trop petits (moins de 30 blocs de base) et ne contiennent vraisemblablement pas de chemins infaisables. Nous divisons le reste des benchmarks de Mälardalen selon qu'ils nécessitent l'installation d'un seuil maximum (η) d'états acceptés en tout point du programme au delà duquel des réductions par union abstraite sont effectuées. Ainsi, les programmes dans la catégorie "gros benchmarks" échouent sans cette technique, en raison d'une complexité trop élevée en mémoire (utilisation de plus de 8 Go de mémoire). Le temps d'analyse sur les petits benchmarks pourrait être largement réduit par cette technique, mais nous perdrions ainsi quelques chemins infaisables à cause de la perte de précision causée par la réduction d'états. Nous ne faisons pas cette distinction pour les benchmarks de debie1 et PapaBench, qui sont en général assez gros.

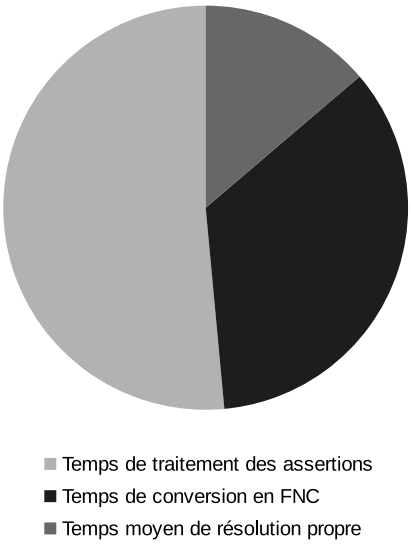


Figure 5.7 – Répartition du temps de résolution moyen pour les problèmes SMT de `prime`

En raison de la fréquence élevée des tests de satisfiabilité et du nombre important de contraintes SMT générées à chaque test, la résolution de problème SMT constitue la majeure partie du temps d'analyse (généralement plus de 90%). Ce temps semble fortement corrélé à la taille du benchmark, mais aussi à son nombre de boucles. La

14. L'estimation du WCET est faite par OTAWA, pour une architecture LPC2138.

Figure 5.7 donne la répartition du temps de résolution de CVC4 pour les 388 problèmes SMT (dont 50 insatisfiables) générés à partir de `prime` : 51% pour le traitement des assertions, 35% pour la conversion du problème en forme normale conjonctive, et 14% pour la résolution du problème elle-même.

Le nombre de chemins infaisables détectés, l'efficacité de la factorisation des chemins infaisables, et leur impact sur le WCET sont très variables et difficilement prévisibles. Par exemple, les 50 chemins infaisables détectés sur `prime` ont un effet important sur le WCET (réduction de 27%) alors que les 1168 chemins infaisables de `qsort-exam` ne l'affectent pas. Ce type de résultats est toutefois normal et attendu : les résultats d'une telle analyse sont pollués par de nombreux chemins infaisables peu coûteux à exécuter, ne faisant donc pas partie du WCEP et n'affectant pas l'estimation du WCET.

Nous pouvons toutefois observer que PapaBench donne des résultats médiocres et que debie1 donne d'excellents résultats pour deux tâches. Le logiciel DEBIE-1 étant une réelle application critique (embarquée sur plusieurs satellites à l'heure actuelle), cela soutient la prémisse que la présence de chemins infaisables a un effet négatif important de surestimation du WCET pour des programmes de systèmes critiques. Par ailleurs, les bons résultats sur l'ensemble des benchmarks contenant un nombre élevé de boucles par rapport à leur taille (à l'exception de `navigation`) renforce l'idée que les boucles renforcent l'impact et l'importance des chemins infaisables pour l'évaluation du WCET.

5.3.2.3 Nature des chemins infaisables détectés

À titre de comparaison, la Table 5.3 montre des résultats obtenus sur les benchmarks de Mälardalen avec une analyse de recherche de chemins infaisables utilisant des états non-paramétriques ($\widetilde{S}$) et par simple point fixe sur les boucles (cf. section 4.2). En l'absence d'évaluation de l'impact sur le WCET, les résultats en termes de chemins infaisables sont difficilement comparables, mais nous observons des larges différences en temps d'analyse sur certains programmes, dans les deux sens, vraisemblablement dues à des variations de la complexité des problèmes SMT générés. La complexité de l'analyse par $\breve{S}$ est particulièrement élevée pour les benchmarks contenant de nombreuses boucles imbriquées, comme `ludcmp`, `minver` et `fft1`, comme l'on pouvait s'y attendre, étant donné que l'analyse de boucles plus fine génère plus de contraintes, et donc des problèmes SMT plus complexes.

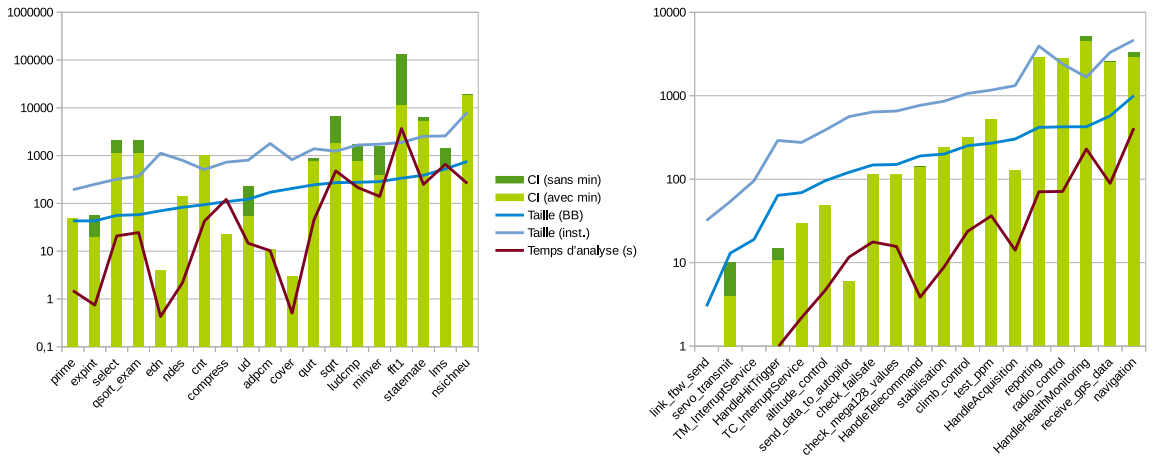
Benchmark	BB	Inst.	Temps (s)	CI (sans min)	CI $\left(\frac{\text{minimisés}}{\text{complets}}\right)$	Longueur moyenne	CI $\left(\frac{\text{1-arc}}{\text{total}}\right)$
Petits benchmarks Mälardalen (sans réduction d'états, $\eta = +\infty$)							
expint	43	251	0,496	13	7/0	1,14	6/7
prime	43	193	1,393	36	21/0	1,29	15/21
select	56	321	99,230	26	4/8	4,33	2/12
qsort-exam	58	369	99,600	32	2/0	1,00	2/2
edn	70	1123	0,329	7	6/0	1,33	4/6
ndes	83	796	6,656	0	0/0	-	-
cnt	94	511	1,760	69	14/0	1,36	9/14
compress	109	728	3,019	39	9/0	1,78	5/9
cover	205	822	0,696	3	3/0	1,00	3/3
Gros benchmarks Mälardalen (avec réduction d'états, $\eta = 250$)							
ud	122	802	13,229	234	30/0	1,70	9/30
adpcm	171	1797	28,148	352	57/0	3,07	3/57
qurt	245	1390	215,679	4750	131/40	3,68	124/171
sqrt	272	1236	38,353	649	54/10	2,16	19/64
ludcmp	276	1669	28,574	464	65/0	1,38	40/65
minver	286	1730	7,580	125	26/0	1,15	22/26
fft1	337	1882	87,001	2837	92/84	3,13	74/176
statemate	387	2530	82,343	1781	70/0	1,77	25/70
lms	527	2582	235,767	8526	554/30	1,96	207/584
nsichneu	756	8088	216,845	19415	3927/8328	5,76	0/12255

Table 5.3 – Résultats expérimentaux pour l'analyse par $\widetilde{S}$ et statistiques sur la nature des chemins infaisables détectés

La sixième colonne de cette table différencie, pour le cas d'une analyse avec minimisation, les chemins infaisables qui ont pu être minimisés en un ensemble d'arcs en conflit et ceux pour lesquels le conflit suggéré par l'*unsat core* ne s'est pas révélé valide (incluant un chemin faisable, probablement à cause d'effets de bords), et pour lesquels l'ensemble des chemins complets est utilisé. L'avant-dernière colonne donne la longueur (nombre d'arcs) moyenne des conflits obtenus.

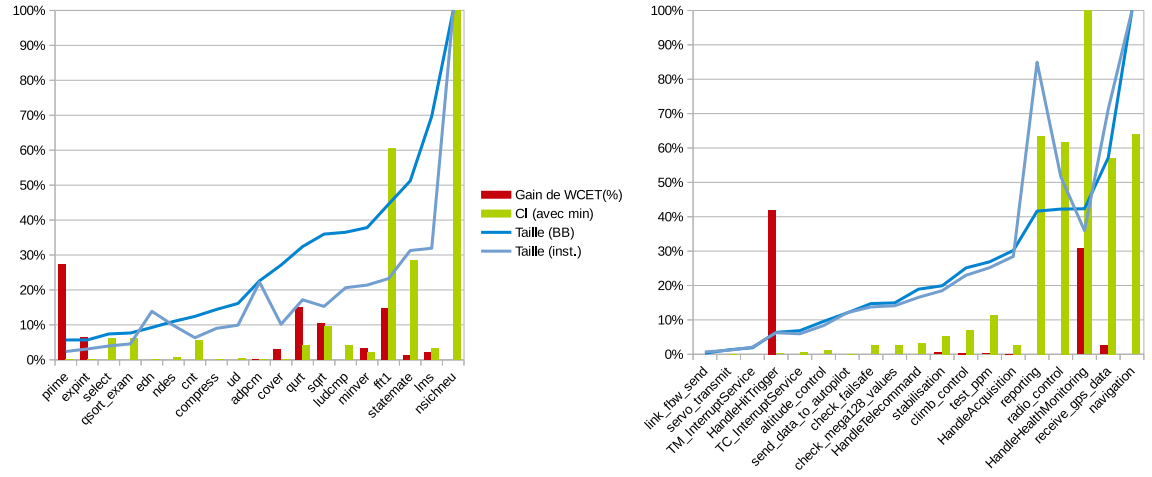
Ainsi, un examen approfondi de `select` nous révèle que l'analyse détecte en réalité 5 conflits, mais que la minimisation échoue pour l'un d'entre eux, générant ainsi 8 chemins infaisables complets (moins quelques arcs supprimés par analyse de dominance *a posteriori*). Nous obtenons donc $4 + 8 = 12$ conflits au lieu de 5, et un nombre moyen d'arcs par conflit élevé. De tels échecs de minimisation, même rares, peuvent rapidement gonfler le nombre de conflits et leur taille, rendant ces résultats plus difficiles à exploiter.

Enfin, la dernière colonne donne la proportion de conflits constitués d'un seul arc. Ces conflits expriment en réalité du code dynamiquement mort (cf. section 2.4.1), et sont une partie importante des chemins infaisables détectés. Notamment, un examen de `cover` nous révèle que trois chemins infaisables d'un seul arc permettent la réduction du WCET estimé de 3%. La présence de code dynamiquement mort joue donc un rôle significatif dans la surestimation du WCET par inclusion de chemins sémantiquement impossibles dans le WCEP.



(a) Mälardalen **(b)** Debie et PapaBench

Figure 5.8 – Complexité de l'analyse (échelle logarithmique)



(a) Mälardalen **(b)** Debie et PapaBench

Figure 5.9 – Impact des chemins infaisables détectés sur l'estimation du WCET

La Figure 5.8 affiche sur une échelle logarithmique le temps d'analyse et le nombre de conflits détectés en fonction de la taille de chaque benchmark. Le temps d'analyse semble corrélé avec la taille des benchmarks, mais aussi avec le nombre de chemins infaisables détectés. L'analyse semble relativement évolutive dans le sens où sa complexité n'augmente pas exponentiellement avec la taille des benchmarks considérés.

La Figure 5.9 présente l'impact des chemins infaisables trouvés sur l'estimation du WCET. Elle souligne la forte variabilité de cet impact et la faible corrélation entre le nombre de conflits obtenus (après minimisation) et l'amélioration de l'estimation du WCET. Ainsi, aucun des nombreux chemins infaisables détectés sur des bench-

marks comme `nsichneu` de Mälardalen ou `navigation` de Papabench ne semblent être sur le WCEP. À l'inverse, un faible nombre de conflits sur `prime` de Mälardalen et `HandleHitTrigger` de PapaBench suffit à améliorer largement la précision du WCET estimé.

5.4 Conclusion générale

Ce chapitre a présenté des techniques de détection de chemins infaisables à partir de propriétés de flot de données décrivant l'état du programme en différents points. Nous avons transformé les états abstraits construits et manipulés dans les chapitres précédents en un ensemble de contraintes SMT, et utilisé des solveurs pour résoudre le problème de décision et détecter d'éventuels états insatisfiables.

Ceci fait, nous avons utilisé l'implantation de méthodes modernes dans deux solveurs SMT, CVC4 et Z3, pour extraire des noyaux insatisfiables minimaux et factoriser, minimiser les chemins infaisables obtenus, et ainsi améliorer leur exploitabilité. Cette partie est importante afin de pouvoir intégrer *efficacement* les informations traduites dans le processus de calcul du WCET.

Les chemins infaisables obtenus sont exprimés dans un format portable, le langage FFX. Ils peuvent ensuite être traduits, entre autres, sous la forme de contraintes ILP qui, bien que souvent imprécises, suffisent pour améliorer l'estimation du WCET pour certains programmes.

La résolution de problèmes SMT domine en pratique largement le temps d'analyse. Les expérimentations sur trois suites de benchmarks, incluant des applications temps-réel ou critiques, donnent des résultats très variables mais significatifs, en particulier sur des tâches denses en boucles, bien que les chemins infaisables "utiles" ne soient pas toujours les plus complexes : une partie de l'amélioration de l'estimation du WCET est due à la suppression de code dynamiquement mort, démontrant par ailleurs l'intérêt du *slicing* (cf. section 3.1.2.6) pour l'analyse de WCET.

6

Conclusion

Résumé

Les travaux de cette thèse s'intègrent dans le cadre de l'estimation sûre du temps d'exécution pire-cas (WCET) pour des systèmes critiques. Les développements présentés ont pour but principal l'amélioration de la précision des méthodes de calcul de WCET par analyse statique, en particulier la réduction du pessimisme engendré par l'énumération implicite de chemins (IPET). Nous recherchons pour cela des chemins infaisables, directement sur le code binaire, représentation du programme la plus proche de celle qui va être exécutée.

Notre approche repose sur une abstraction du programme, sous la forme d'une part de graphes de flot de contrôle (CFG) représentant la structure du programme, qui n'est pas directement apparente sur du code binaire, et d'une autre part d'une représentation intermédiaire des instructions assembleur par un jeu d'instructions sémantiques réduit, plus simple à analyser.

Nous avons utilisé des techniques d'interprétation abstraite pour définir une abstrac-

tion des états de la machine permettant de représenter l'ensemble des états possibles à un point du programme. Des fonctions d'interprétation définissent une sémantique abstraite, et la validité de cette représentation est vérifiable grâce à l'établissement d'une correspondance de Galois. Le but des domaines d'états abstraits ainsi définis est de représenter efficacement les états concrets de la machine, avec le moins de pertes de précision possible.

Nous aboutissons à une abstraction paramétrée, fonction de l'état des variables du programme en début d'analyse. Ce type d'abstraction nous permet de réaliser une analyse de flot de données modulaire, capable de traiter des régions du programme séparément, et de composer leurs résultats. Nous exploitons cette propriété de composabilité des états abstraits pour détecter dans les fonctions des propriétés dépendantes d'un contexte d'appel sans répéter l'analyse de la fonction à chaque point d'appel (sans aplatir le CFG). Ainsi, chaque CFG est analysé une seule fois, donnant des résultats indépendants du contexte, mais qui pourront toutefois être spécialisés pour chaque point d'appel.

Nous utilisons également cette propriété de composabilité pour déterminer l'effet d'une itération de chaque boucle. Nous utilisons cette information pour décrire l'évolution des variables du programme au fur et à mesure de l'exécution de la boucle. Nous appliquons pour cela une opération de généralisation sur l'état paramétrique représentant une unique itération. Cette opération peut reconnaître des schémas de progressions linéaires et déterminer des propriétés des variables du programme vraies pour toute itération de la boucle, et exprimées en fonction du numéro d'itération.

Le parcours des CFG se fait grâce à un système de rendez-vous qui permet de détecter des ensembles de propriétés valides en certains points du programme – sur chaque arc, et en début de chaque bloc de base. Ces propriétés sont associées aux chemins pour lesquels elles sont variables, et sont utilisées pour tester la satisfiabilité des états abstraits, c'est-à-dire pour vérifier qu'ils représentent au moins un état concret. Dans le cas contraire, le chemin associé est infaisable, ces états abstraits représentant l'ensemble des états du programme possibles pour un chemin du CFG : un ensemble d'états possibles vide implique que le chemin associé est infaisable.

La question de la satisfiabilité des états abstraits est un problème de décision complexe qui peut être traduit en un ensemble de contraintes entières et être résolu par un solveur SMT. Ces outils donnent généralement une réponse binaire – "satisfiable" ou "insatisfiable" – mais des extensions modernes de ces outils peuvent, lorsqu'ils détectent

une contradiction dans l'ensemble des contraintes sur lequel ils sont interrogés, en donner un sous-ensemble minimal insatisfiable, appelé *unsat core*. L'outil de recherche de chemins infaisables appliquant les méthodes développées dans cette thèse, PathFinder, intègre ainsi deux solveurs SMT, CVC4 et Z3, capables de raisonner sur des problèmes entiers et permettant l'extraction d'*unsat cores*.

Ces sous-ensembles minimaux insatisfiables nous permettent, lors de l'analyse de programmes, d'éviter le fractionnement des chemins infaisables détectés en formulant des hypothèses sur des ensembles d'arcs en conflit (correspondant à l'ensemble des contraintes en conflit). Cela nous permet d'exprimer les conflit détectés succinctement, plutôt que sous la forme de nombreux, long chemins infaisables, en réalité issus d'un petit nombre de conflits entre arcs. Cette partie de l'analyse, la minimisation de chemins infaisables, est importante pour l'intégration efficace des informations traduites dans le processus de calcul du WCET.

Les chemins infaisables obtenus sont exprimés dans un format portable, le langage FFX. Ils peuvent ensuite être traduits, entre autres, sous la forme de contraintes ILP qui, bien que souvent imprécises, suffisent pour améliorer l'estimation du WCET pour certains programmes, comme l'ont montré les expérimentations sur trois suites de benchmarks. Celles-ci renforcent l'idée que l'analyse de WCET de programmes (y compris ceux intégrés dans des systèmes critiques, comme DEBIE-1) prend en compte des chemins infaisables amplifiant son pessimisme, dégradant ainsi sa précision.

Perspectives

L'analyse de flot de données développée dans cette thèse s'est montrée suffisamment puissante pour trouver des chemins infaisables sur la quasi-totalité des benchmarks testés de taille raisonnablement grande. Parmi ces chemins infaisables, quelques uns affectent l'estimation du WCET, de telle sorte que leur détection permet de réduire le pessimisme. Les développements de cette thèse sont donc utiles, d'autant plus que des résultats ont été obtenus sur de réelles applications pour lesquelles l'estimation précise du WCET est un enjeu important.

Il est cependant très difficile d'évaluer l'efficacité de l'analyse, étant donné que nous ne connaissons ni le WCET réel, ni de l'ensemble des chemins infaisables à détecter. Il est donc difficile de proposer des pistes d'amélioration qui donneraient certainement

lieu à la détection de chemins infaisables supplémentaires, et encore plus difficile de prévoir si ceux-ci auraient un impact sur le WCET estimé.

Ceci dit, toutes les classes de chemins infaisables ne sont pas effectivement visées par l'analyse : des extensions pour rechercher des chemins infaisables sur la première itération de boucles, ou entre deux itérations n et $n+1$ (par déroulage de boucle) étofferaient probablement les résultats. Les chemins infaisables affectant le WCET ne sont toutefois pas nécessairement ceux qui demandent une analyse fine du flot de données. Les expérimentations ont déjà montré que nombre des chemins infaisables "utiles" sont triviaux, par exemple du code dynamiquement mort dû à un cas rare mais cher en temps d'exécution dans une fonction appelée.

Une contribution importante de cette thèse tient dans la minimisation des conflits détectés, réalisée grâce à l'exploitation d'une fonctionnalité moderne des solveurs SMT, qui était par exemple à peine à l'état expérimental dans CVC4 au début de cette thèse. Le système qui assigne à chaque prédicat un ensemble d'arcs dits responsables ne servant qu'à émettre des hypothèses, il n'a pas besoin d'être validé, mais il gagnerait à être formalisé et raffiné ; en effet, de nombreux conflits exprimés sont encore fractionnés et longs. Ce processus de minimisation, s'il ne modifie pas l'ensemble des chemins désignés infaisables, simplifie les propriétés et facilite leur exploitation. La collaboration [75] entre PathFinder et l'outil d'exploitation de chemins infaisables de V. Mussot a mis l'accent sur l'importance de l'expression des chemins infaisables en conflits courts et en nombre réduits, du fait de leur impact sur la complexité d'analyse. Sans cet effort de minimisation, les plus gros des benchmarks de Mälardalen n'auraient pas pu être traités, notamment.

Si l'abstraction utilisée dans cette thèse pour l'analyse de flot de données finale est capable de détecter et représenter des propriétés intéressantes, elle peut aussi échouer sur des cas relativement peu complexes que d'autres abstractions gèrent sans difficultés. En particulier, un état abstrait correspondant à un chemin, ne peut pas représenter de disjonctions dans sa forme actuelle. Par conséquent, nos techniques d'analyse de programme pourraient être complétées par l'intégration d'abstractions plus classiques qui viendraient pallier aux défauts de l'abstraction $\breve{S}$. Les propriétés ainsi découvertes pourraient être rajoutées sous la forme de contraintes SMT à chaque appel au solveur.

Enfin, la complexité de l'analyse est suffisamment faible pour traiter l'intégralité

des trois ensembles de benchmarks utilisés. Ce résultat positif s'obtient toutefois au prix d'union abstraites destructrices sur des points de fusion sur lesquels le nombre d'états abstraits à maintenir est jugée trop importante. Puisque seules les propriétés communes à tous les chemins sont conservées en ces points du programme, une fréquence élevée d'unions abstraites nous empêche de détecter des conflits entre deux arcs trop éloignés dans un CFG. C'est un problème potentiellement important, du fait que certains chemins infaisables sont introduits par le programmeur précisément parce que les arcs en conflit sont "noyés" dans un grand volume de code, et donc masqués au développeur ou difficiles à supprimer (restructuration du programme difficile). De tels chemins infaisables ne peuvent pas, au delà d'une certaine distance, être détectés dans l'état actuel de notre analyse.

Une première piste d'amélioration serait, plutôt que de calculer l'union de n états lorsque n dépasse le seuil η, de séparer ces états en k groupes d'états contenant des propriétés communes intéressantes, et d'appliquer l'union séparément sur chacun de ses groupes de manière à réduire finalement les n états en k états pertinents, plutôt qu'un seul état peu utile. La recherche au préalable de conditions impactant fortement le WCET [108] (en évaluant l'écart de WCET, noté Δ, entre les arcs de sortie pris et non pris de chaque bloc de base) est une bonne heuristique pour nous aider à déterminer quelles propriétés nous devons conserver. Cette approche pose toutefois le problème de la représentation efficace des disjonctions de chemins, un problème de graphe difficile.

Une deuxième piste consiste à s'attaquer à la source de complexité de l'analyse en général. La résolution de problèmes SMT domine largement le temps d'analyse ($> 90\%$ dans la grande majorité des cas). Les solveurs SMT sont, en réalité, peu efficaces pour la résolution d'une longue série de problèmes simples – les conflits entre contraintes détectés sont souvent triviaux. Ces solveurs sont plutôt conçus dans l'optique de la résolution de problèmes plus complexes en moindre nombre, ou au moins définis incrémentalement, ce qui n'est pas notre cas (du moins pour les propriétés issues de la table des variables) puisque les propriétés découvertes au fur et à mesure de l'analyse de programmes peuvent être modifiées ou supprimées. La résolution de problèmes de satisfiabilité étant le goulot d'étranglement de notre analyse, sa complexité pourrait être atténuée par l'utilisation de techniques de résolution plus adaptées (plus rapides sur des problèmes SMT simples), possiblement implantées dans un solveur *ad hoc*.

Annexes

A Structures et abstractions définies dans la thèse

$\mathbb{Z}_{32}$	$\mathbb{Z}/2^{32}\mathbb{Z}$	Groupe des entiers sur 32 bits
${\mathbb{Z}_{32}}^\sharp$	$\mathbb{Z}_{32} \times \{\emptyset, +\mathrm{SP}\}$	Constantes symboliques (absolues ou relatives au pointeur de pile)
$C^\sharp$	${\mathbb{Z}_{32}}^\sharp \cup \{\top\}$	Constantes symboliques et $\top$
Reg	$\{r_0, r_1, \ldots, r_m\}$	Registres
Tmp	$\{t_1, t_2, \ldots, t_n\}$	Variables temporaires
Var	$Reg \cup Tmp$	Variables directement accessibles
Mem	$\{m_a,\ a \in \mathbb{Z}_{32}\}$	Mémoire, tas et pile unifiés
$Mem^\sharp$	$\{m_a,\ a \in {\mathbb{Z}_{32}}^\sharp\}$	Mémoire, tas et pile séparés
$\mathbb{V}$	$Var \cup Mem$	Variables du programme
$\mathbb{V}^\sharp$	$Var \cup Mem^\sharp$	Variables du programme, tas et pile séparés
$\widehat{\mathbb{V}}$	$\mathbb{V}^\sharp$	Notation pour les valeurs initiales des variables
Λ	$\{\lambda_n,\ n \geq 1\}$	Variables abstraites (variables arbitraires)
$\mathcal{I}$	$\{I_{h_k} \mid h_k \in H_G\}$	Indices d'itération
Φ	$\{+, -, \times, /, \mathrm{mod}, \sim\}$	Opérateur arithmétique (ou de comparaison)
$\mathbb{E}$	$\mathrm{gfp}(X = C^\sharp + \mathbb{V}^\sharp + \Phi X^2)$	Expressions
$\widehat{\mathbb{E}}$	$\mathrm{gfp}(X = C^\sharp + \widehat{\mathbb{V}} + \Phi X^2)$	Expressions (valeurs initiales)
$\widehat{\mathbb{E}}_\Lambda$	$\mathrm{gfp}(X = C^\sharp + \widehat{\mathbb{V}} + \Lambda + \Phi X^2)$	Expressions (valeurs initiales, avec variables abstraites)
$\breve{\mathbb{E}}_\Lambda$	$\mathrm{gfp}(X = C^\sharp + \widehat{\mathbb{V}} + \Lambda + \mathcal{I} + \Phi X^2)$	Expressions (valeurs initiales, avec variables abstraites, avec indices d'itération)
Ψ	$\{=, \neq, \leq, <\}$	Relation binaire
P	$\mathbb{E} \times \Psi \times \mathbb{E}$	Prédicats
$\widehat{P}$	$\widehat{\mathbb{E}} \times \Psi \times \widehat{\mathbb{E}}$	Prédicats (valeurs initiales)
$\widehat{P}_\Lambda$	$\widehat{\mathbb{E}}_\Lambda \times \Psi \times \widehat{\mathbb{E}}_\Lambda$	Prédicats (valeurs initiales, avec variables abstraites)
$\breve{P}_\Lambda$	$\breve{\mathbb{E}}_\Lambda \times \Psi \times \breve{\mathbb{E}}_\Lambda$	Prédicats (valeurs initiales, avec variables abstraites, avec indices d'itération)

Θ	$\mathbb{V}^\sharp \to \widehat{\mathbb{E}}_\Lambda$	Table des variables
$\breve{\Theta}$	$\mathbb{V}^\sharp \to \breve{\mathbb{E}}_\Lambda$	Table des variables, avec indices d'itération
S	$\mathbb{V} \to \mathbb{Z}_{32}$	État concret
$\vec{S}$	$\mathbb{V}^\sharp \to C^\sharp$	Application des variables vers une constante
$S^\sharp$	$\vec{S} \cup \{\bot\}$	État abstrait (constantes)
$\widetilde{S}$	$\vec{S} \times \mathcal{P}(P)$	État abstrait (prédicats)
$\widehat{S}_0$	$\Theta_0 \times \mathcal{P}(\widehat{P})$	État abstrait paramétrique
$\widehat{S}$	$\Theta \times \mathcal{P}(\widehat{P}_\Lambda)$	État abstrait paramétrique (avec variables abstraites)
$\breve{S}$	$\breve{\Theta} \times \mathcal{P}(\breve{P}_\Lambda)$	État abstrait paramétrique (avec variables abstraites et indices de boucles)
I_{sem}		Instructions sémantiques
$\mathscr{C}_{\text{FFX}}$		Conflits FFX
Π^*		Chemins abstraits

B Sémantique abstraite complète sur $\widehat{S}$

Définition B.1. La fonction d'interprétation $\hat{\mathbb{I}} : I_{sem} \to \widehat{S} \to \widehat{S}$ est définie pour tout état abstrait $\hat{s} = (\theta, p) \in \widehat{S}$ comme suit :

$$\Gamma_\Lambda \vdash$$

$$\hat{\mathbb{I}}[\texttt{seti}\ d, k](\theta, p) \coloneqq (\theta[\theta(d) \mapsto k], p)$$

$$\hat{\mathbb{I}}[\texttt{set}\ d, a](\theta, p) \coloneqq (\theta[d \mapsto \theta(a)], p)$$

$$\hat{\mathbb{I}}[\texttt{add}\ d, a, b](\theta, p) \coloneqq (\theta[d \mapsto \theta(a) + \theta(b)], p)$$

$$\hat{\mathbb{I}}[\texttt{sub}\ d, a, b](\theta, p) \coloneqq (\theta[d \mapsto \theta(a) - \theta(b)], p)$$

$$\hat{\mathbb{I}}[\texttt{mul}\ d, a, b](\theta, p) \coloneqq (\theta[d \mapsto \theta(a) \times \theta(b)], p)$$

$$\hat{\mathbb{I}}[\texttt{div}\ d, a, b](\theta, p) \coloneqq (\theta[d \mapsto \theta(a) \,/\, \theta(b)], p)$$

$$\hat{\mathbb{I}}[\texttt{mod}\ d, a, b](\theta, p) \coloneqq (\theta[d \mapsto \theta(a) \bmod \theta(b)], p)$$

$$\hat{\mathbb{I}}[\texttt{cmp}\ d, a, b](\theta, p) \coloneqq (\theta[d \mapsto \theta(a) \sim \theta(b)], p)$$

$$\hat{\mathbb{I}}[\texttt{scratch}\ d](\theta, p) \coloneqq (\hat{\iota}_d(\theta), p)$$

$$\hat{\mathbb{I}}[\texttt{load}\ d, a, t](\theta, p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(m_{\theta(a)})], p) & \text{si } \theta(a) \in {\mathbb{Z}_{32}}^\sharp \\ (\hat{\iota}_d(\theta), p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}[\texttt{store}\ d, a, t](\theta, p) \coloneqq \begin{cases} (\theta[m_{\theta(a)} \mapsto \theta(d)], p) & \text{si } \theta(a) \in {\mathbb{Z}_{32}}^\sharp \\ (\underbrace{\hat{\iota}_{v_1} \circ \hat{\iota}_{v_2} \circ \ldots \circ \hat{\iota}_{v_k}}_{(v_1, v_2, \ldots, v_k) = Mem^\sharp}(\theta), p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}[\texttt{shl}\ d, a, b](\theta, p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(d) \times 2^{\theta(b)}], p) & \text{si } \theta(b) \in [\![0, 31]\!] \\ (\theta', p \cup \{z\,\theta'(d) \geq z\,\theta(a)\}) & \text{sinon si } \theta(a) \in \mathbb{Z}, \text{ avec } \theta' \coloneqq \hat{\iota}_d(\theta) \\ & \qquad \text{et } z \coloneqq \mathit{signe}(\theta(a)) \\ (\hat{\iota}_d(\theta), p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}[\texttt{asr}\ d, a, b](\theta, p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(d) \,/\, 2^{\theta(b)}], p) & \text{si } \theta(b) \in [\![0, 31]\!] \\ (\theta', p \cup \{z\,\theta'(d) \leq z\,\theta(a)\}) & \text{sinon si } \theta(a) \in \mathbb{Z}, \text{ avec } \theta' \coloneqq \hat{\iota}_d(\theta) \\ & \qquad \text{et } z \coloneqq \mathit{signe}(\theta(a)) \\ (\hat{\iota}_d(\theta), p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}[\texttt{neg}\ d, a](\theta, p) \coloneqq (\theta[d \mapsto 0 - \theta(a)], p)$$

$$\hat{\mathbb{I}}[\texttt{not}\ d, a](\theta, p) \coloneqq (\theta[d \mapsto 1 - \theta(a)], p)$$

$$\hat{\mathbb{I}}[\texttt{and}\ d,a,b](\theta,p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(a)\ \&\ \theta(b)],p) & \text{si } \theta(a) \in \mathbb{Z}_{32} \wedge \theta(b) \in \mathbb{Z}_{32} \\ (\theta', p \cup \{\theta'(d) = \theta(b) \bmod 2^k\}) & \text{si } \exists k \leq 31,\ \theta(a) = 2^k - 1 \in \mathbb{Z}_{32} \\ & \text{et avec } \theta' \coloneqq \hat{\iota}_d(\theta) \\ (\theta', p \cup \{\theta'(d) = \theta(a) \bmod 2^k\}) & \text{si } \exists k \leq 31,\ \theta(b) = 2^k - 1 \in \mathbb{Z}_{32} \\ & \text{et avec } \theta' \coloneqq \hat{\iota}_d(\theta) \\ \left(\theta', p \cup \left\{ \begin{array}{l} z_a\theta'(d) \leq z_a\theta(a), \\ z_a\theta'(d) \leq z_b\theta(b) \end{array} \right\}\right) & \text{sinon si } (\theta(a),\theta(b)) \in \mathbb{Z}^2 \\ & \text{avec } \theta' \coloneqq \hat{\iota}_d(\theta), \text{ et } (z_a,z_b) \coloneqq \\ & (\mathit{signe}(\theta(a)), \mathit{signe}(\theta(b))) \\ (\iota_d(\theta),p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}[\texttt{or}\ d,a,b](\theta,p) \coloneqq \begin{cases} (\theta[d \mapsto \theta(a) \mid \theta(b)],p) & \text{si } \theta(a) \in \mathbb{Z}_{32} \wedge \theta(b) \in \mathbb{Z}_{32} \\ \left(\theta', p \cup \left\{ \begin{array}{l} z_a\theta'(d) \geq z_a\theta(a), \\ z_a\theta'(d) \geq z_b\theta(b) \end{array} \right\}\right) & \text{sinon si } (\theta(a),\theta(b)) \in \mathbb{Z}^2 \\ & \text{avec } \theta' \coloneqq \hat{\iota}_d(\theta), \text{ et } (z_a,z_b) \coloneqq \\ & (\mathit{signe}(\theta(a)), \mathit{signe}(\theta(b))) \\ (\iota_d(\theta),p) & \text{sinon} \end{cases}$$

$$\hat{\mathbb{I}}[\texttt{assume}\ c,a,b](\theta,p) \coloneqq \begin{cases} (\theta, p \cup \{\theta(a)\ \psi_c\ \theta(b)\}) & \text{si } c \in \{=,\neq,<,\leq\},\ \psi_c \coloneqq c \\ (\theta, p \cup \{\theta(b)\ \psi_c^{-1}\ \theta(a)\}) & \text{sinon, avec } \psi_>^{-1} \coloneqq <,\ \psi_\geq^{-1} \coloneqq \leq \end{cases}$$

C Démonstration du Théorème 3.1

Le Théorème 3.1 énoncé page 72 affirme la validité de la technique d'interprétation d'instructions sur les prédicats par substitution de la fonction inverse utilisée dans la section 3.4. Cette annexe en fait la preuve, après un rappel de l'énoncé.

Théorème 3.1. *Soit $i \in I_{sem}$ une instruction sémantique. Si*

(1) il existe une fonction $\mathbb{I}[i]^{-1}$, inverse de la fonction d'interprétation concrète :

$$\forall s \in S,\ \mathbb{I}[i]^{-1}(\mathbb{I}[i](s)) = s$$

(2) il existe $v \in \mathbb{V}^\sharp$, unique variable modifiée par i :

$$\forall s \in S,\ \forall v' \neq v,\ \mathbb{I}[i](s)(v) = s(v)$$

(3) il existe une expression $e_i^{-1} \in \mathbb{E}$ modélisant l'effet de $\mathbb{I}[i]^{-1}$ sur v :

$$\forall s \in S,\ \gamma_{\mathbb{E}}(s, e_i^{-1}) = \mathbb{I}[i]^{-1}(s)(v)$$

Alors, le remplacement de v par e_i^{-1} dans les prédicats est une abstraction valide et précise (sans approximation) de $\mathbb{I}[i]$, c'est-à-dire

$$\forall p \in \mathcal{P}(P),\ \ \gamma_P(p[v \mapsto e_i^{-1}]) = \mathbb{I}[i](\gamma_P(p))$$

Démonstration. Pour tout état concret $s \in S$, et pour tout $p' \in \mathcal{P}(P)$

$$\begin{aligned} s \in \gamma_P(p') &\Leftrightarrow \forall p_i \in p',\ s \in \gamma_P(p_i) \\ &\Leftrightarrow \forall (e_1\ \psi\ e_2) \in p',\ s \in \gamma_P(e_1\ \psi\ e_2) \\ &\Leftrightarrow \forall (e_1\ \psi\ e_2) \in p',\ \exists x_1 \in \gamma_{\mathbb{E}}(s, e_1),\ \exists x_2 \in \gamma_{\mathbb{E}}(s, e_2),\ x_1\ \psi\ x_2 \end{aligned}$$

Ainsi, $\forall s \in S$, $\forall p \in \mathcal{P}(P)$,

$$\begin{aligned} \mathbb{I}[i](s) \in \gamma_P(p[v \mapsto e_i^{-1}]) \Leftrightarrow \forall (e_1\ \psi\ e_2) \in p[v \mapsto e_i^{-1}], \\ \exists x_1 \in \gamma_{\mathbb{E}}(\mathbb{I}[i](s), e_1),\ \exists x_2 \in \gamma_{\mathbb{E}}(\mathbb{I}[i](s), e_2),\ x_1\ \psi\ x_2 \end{aligned}$$

C'est-à-dire

$$\mathbb{I}[i](s) \in \gamma_P(p[v \mapsto e_i^{-1}]) \Leftrightarrow \forall (e_1 \ \psi \ e_2) \in p,$$
$$\exists x_1 \in \gamma_{\mathbb{E}}(\mathbb{I}[i](s), e_1[v \mapsto e_i^{-1}]),\ \exists x_2 \in \gamma_{\mathbb{E}}(\mathbb{I}[i](s), e_2[v \mapsto e_i^{-1}]),\ x_1 \ \psi \ x_2$$

Nous cherchons d'abord à prouver la propriété $\mathscr{P}$:

$$\forall e \in \mathbb{E},\ \gamma_{\mathbb{E}}(\mathbb{I}[i](s), e[v \mapsto e_i^{-1}]) = \gamma_{\mathbb{E}}(s, e) \qquad (\mathscr{P})$$

La preuve est récursive :

- si $e = \top$,
$$\gamma_{\mathbb{E}}(\mathbb{I}[i](s), e[v \mapsto e_i^{-1}]) = \mathbb{Z}_{32} = \gamma_{\mathbb{E}}(s, e)$$
- si $e \in {\mathbb{Z}_{32}}^\sharp$, alors $e[v \mapsto e_i^{-1}] = e$, et donc
$$\gamma_{\mathbb{E}}(\mathbb{I}[i](s), e[v \mapsto e_i^{-1}]) = \gamma_{{\mathbb{Z}_{32}}^\sharp}(e[v \mapsto e_i^{-1}]) = \gamma_{{\mathbb{Z}_{32}}^\sharp}(e) = \gamma_{\mathbb{E}}(s, e)$$
- si $e \in \mathbb{V}^\sharp$, $e \neq v$, alors $e[v \mapsto e_i^{-1}] = e$. Or d'après l'hypothèse (2), $\mathbb{I}[i](s)(e) = s(e)$, donc
$$\gamma_{\mathbb{E}}(\mathbb{I}[i](s), e[v \mapsto e_i^{-1}]) = \{\mathbb{I}[i](s)(e)\} = \{s(e)\} = \gamma_{\mathbb{E}}(s, e)$$
- si $e \in \mathbb{V}^\sharp$, $e = v$, alors $e[v \mapsto e_i^{-1}] = e_i^{-1}$. O r d'après l'hypothèse (3), $\gamma_{\mathbb{E}}(\mathbb{I}[i](s), e_i^{-1}) = \mathbb{I}[i]^{-1}(\mathbb{I}[i](s))(v)$, soit $s(v)$ d'après l'hypothèse (1), et donc
$$\gamma_{\mathbb{E}}(\mathbb{I}[i](s), e[v \mapsto e_i^{-1}]) = \{s(v)\} = \gamma_{\mathbb{E}}(s, e)$$
- si $e = (f \ \phi \ g)$, alors
$$\begin{aligned}\gamma_{\mathbb{E}}(\mathbb{I}[i](s), e[v \mapsto e_i^{-1}]) &= \bigcup_{a \in \gamma_{\mathbb{E}}(\mathbb{I}[i](s), f[v \mapsto e_i^{-1}])} \ \bigcup_{b \in \gamma_{\mathbb{E}}(\mathbb{I}[i](s), g[v \mapsto e_i^{-1}])} \phi(a, b) \\ &= \bigcup_{a \in \gamma_{\mathbb{E}}(s,e)} \ \bigcup_{b \in \gamma_{\mathbb{E}}(s,e)} \phi(a, b) \\ &= \gamma_{\mathbb{E}}(s, e)\end{aligned}$$
par récursion, en appliquant $\mathscr{P}$ sur f et g.

La propriété $\mathscr{P}$ est ainsi prouvée pour tout $e \in \mathbb{E}$.

Nous pouvons donc l'utiliser pour affirmer que :

$$\mathbb{I}[i](s) \in \gamma_P(p[v \mapsto e_i^{-1}]) \Leftrightarrow \forall (e_1 \; \psi \; e_2) \in p, \\ \exists x_1 \in \gamma_{\mathbb{E}}(s, e_1), \exists x_2 \in \gamma_{\mathbb{E}}(s, e_2), \; x_1 \; \psi \; x_2$$

Et donc :

$$\begin{aligned} \mathbb{I}[i](s) \in \gamma_P(p[v \mapsto e_i^{-1}]) &\Leftrightarrow s \in \gamma_P(p) \\ &\Leftrightarrow \mathbb{I}[i](s) \in \mathbb{I}[i](\gamma_P(p)) \end{aligned}$$

Par conséquent,

$$\gamma_P(p[v \mapsto e_i^{-1}]) = \mathbb{I}[i](\gamma_P(p))$$

□

Notons que nous avons usé de raccourcis d'écriture en utilisant $\mathbb{I}$ comme une fonction de $I_{sem} \to S \to S$, ce qui est possible pour toutes les instructions sauf `scratch`, sur laquelle le Théorème 3.1 ne s'applique de toutes façons pas (il n'existe pas d'inverse à $\mathbb{I}$[`scratch` d]).

D Validation de $\hat{\mathbb{I}}$ par Correspondance de Galois

D.1 Construction de la correspondance de Galois

Simplifions la Définition 3.28. On peut remarquer que $\hat{s}$ étant une variable inefficace de la partie inférieure de l'équation (car les prédicats ne dépendent pas des valeurs courantes des variables de la machine), la fonction de concrétisation $\gamma_{\widehat{S}}$ peut également s'écrire comme :

Propriété D.1. *Pour tout $\hat{s} = (\theta, p) \in \widehat{S}$ et pour tout $\vec{s}_0 \in \vec{S}$,*

$$\Gamma_\Lambda \vdash \gamma_{\widehat{S}}(\hat{s})(\vec{s}_0) = \left\{ \vec{s} \in \vec{S} \;\middle|\; \exists L \in \gamma_{\widehat{P}_\Lambda}(\vec{s}_0, p),\, \forall v \in \mathbb{V}^\sharp,\, \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \theta(v)) \right\}$$

où $\gamma_{\widehat{P}_\Lambda} : \vec{S} \times \mathcal{P}(\widehat{P}_\Lambda) \to \Gamma_\Lambda$ définit l'ensemble des variables abstraites pour lesquelles la conjonction de prédicat est valide, pour une valuation des variables initiales dans $\vec{S}$:

$$\Gamma_\Lambda \vdash \gamma_{\widehat{P}_\Lambda}(\vec{s}_0, p) \coloneqq \left\{ L \in \Gamma_\Lambda \;\middle|\; \forall (e_1 \,\psi\, e_2) \in p,\, \exists \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \begin{pmatrix} \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, e_1) \\ \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, e_2) \end{pmatrix},\, x_1 \,\psi\, x_2 \right\}$$

Nous montrons par le Lemme D.1 que $\hat{\cup}$ est une opérateur d'union dans le domaine ordonné $(\widehat{S}, \hat{\sqsubseteq})$. Ce n'est toutefois pas une union minimale, elle peut donc entraîner une perte de précision.

Lemme D.1. *Soit les opérateurs $\hat{\cup}$ et $\hat{\sqsubseteq}$ définis à la section 4.2.2.*

L'opérateur d'union abstraite $\hat{\cup}$ est croissant par $\hat{\sqsubseteq}$. C'est donc bien une union, selon la Définition 2.5.

Démonstration. La fonction $\gamma_{\widehat{P}_\Lambda}$, telle qu'elle est définie dans l'énoncé de la Propriété D.1, donne l'ensemble des variables abstraites répondant à un critère pour chaque prédicat dans p. Il est donc immédiat que

$$\Gamma_\Lambda \vdash \forall \vec{s}_0 \in \vec{S},\, \forall p, p' \in \widehat{P}_\Lambda,\; \gamma_{\widehat{P}_\Lambda}(\vec{s}_0, p) \subseteq \gamma_{\widehat{P}_\Lambda}(\vec{s}_0, p \cap p')$$

Or,

$$\Gamma_\Lambda \vdash \forall \vec{s}_0 \in \vec{S}, \forall L \in \Gamma_\Lambda, \forall \theta \in \Theta, \forall v \in \mathbb{V}^\sharp,\; \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \theta(v)) \subseteq \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \top)$$

puisque $\top$ peut représenter n'importe quelle constante :

$$\gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \top) = {\mathbb{Z}_{32}}^\sharp$$

Ainsi, pour tout $\hat{s} = (\theta, p)$, $\hat{s}' = (\theta', p') \in \widehat{S}$, et pour tout $\vec{s}_0, \vec{s} \in \vec{S}$,

$$\begin{aligned}
&\Gamma_\Lambda \vdash \vec{s} \in \gamma_{\widehat{S}}(\hat{s})(\vec{s}_0) \\
&\quad \Rightarrow \exists L \in \gamma_{\widehat{P}_\Lambda}(\vec{s}_0, p),\ \forall v \in \mathbb{V}^\sharp,\ \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \theta(v)) \\
&\quad \Rightarrow \exists L \in \gamma_{\widehat{P}_\Lambda}(\vec{s}_0, p),\ \forall v \in \mathbb{V}^\sharp,\ \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \theta(v)) \wedge \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \top) \\
&\quad \Rightarrow \exists L \in \gamma_{\widehat{P}_\Lambda}(\vec{s}_0, p \cap p'),\ \forall v \in \mathbb{V}^\sharp,\ \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \theta(v)) \wedge \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \top) \\
&\quad \Rightarrow \vec{s} \in \gamma_{\widehat{S}}(\hat{s} \mathbin{\hat{\sqcup}} \hat{s}')(\vec{s}_0)
\end{aligned}$$

selon la Définition 4.6. Donc

$$\gamma_{\widehat{S}}(\hat{s})(\vec{s}_0) \subseteq \gamma_{\widehat{S}}(\hat{s} \mathbin{\hat{\sqcup}} \hat{s}')(\vec{s}_0)$$

Par conséquent, et pour tout $\hat{s}$ et $\hat{s}' \in \widehat{S}$,

$$\hat{s} \mathbin{\hat{\sqsubseteq}} \hat{s} \mathbin{\hat{\sqcup}} \hat{s}'$$

□

Construisons la correspondance de Galois qui en suit. Soit $\hat{\beta} : \vec{S} \to \widehat{S}$ la fonction de représentation définie par

$$\forall \vec{s} \in \vec{S},\ \hat{\beta}(\vec{s}) \coloneqq (v \mapsto \vec{s}(v), \emptyset)$$

Soit le domaine concret $(\mathcal{P}(\vec{S}), \subseteq)$ et le domaine abstrait par $(\widehat{S}, \mathbin{\hat{\sqsubseteq}})$. Alors, d'après le Théorème 2.2, $\{\mathcal{P}(\vec{S}), \alpha, \gamma, \widehat{S}\}$ est une correspondance de Galois générée par la fonction de représentation $\hat{\beta}$, avec $\alpha(\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_n) = \hat{\sqcup}\{\hat{\beta}(\vec{s}_i) \mid i \in [1, n]\}$ et $\gamma = \gamma_{\widehat{S}}$.

Remarque. Il est possible de dégager sur chaque fonction d'interprétation abstraite une propriété de croissance, c'est-à-dire que si un état $\hat{s}$ est *plus* précis qu'un autre état $\hat{s}'$, l'interprétation de $\hat{s}$ ne peut pas résulter en un état *moins* précis que celle de $\hat{s}'$.

Propriété D.2. *Les fonctions d'interprétation sont croissantes, c'est-à-dire*

$$\forall I \in I_{sem},\ (\hat{s} \mathbin{\hat{\sqsubseteq}} \hat{s}') \Rightarrow \left(\hat{\mathbb{I}}[I](\hat{s}) \mathbin{\hat{\sqsubseteq}} \hat{\mathbb{I}}[I](\hat{s}')\right)$$

La démonstration de cette propriété est longue et fastidieuse, et ne sera pas faite ici. Elle ne devrait toutefois pas présenter de difficulté majeure, du fait que les fonctions d'interprétations sont définies de façon à ne supprimer des prédicats qu'avec la fonction ι et que la présence de propriétés *supplémentaires* dans un état $\hat{s}$ par rapport à un état $\hat{s}'$ ne déclencherait jamais l'invalidation de propriétés communes aux deux états.

D.2 Validation de $\hat{\mathbb{I}}$

Pour tout ensemble d'états $X \in \mathcal{P}(\vec{S})$, on notera $X(v)$ l'ensemble des valeurs que peut prendre la variable v, parmi les éléments de X :

$$X(v) \coloneqq \{\vec{s}(v) \mid \vec{s} \in X\}$$

Nous commençons par affirmer un lemme découlant trivialement de la Propriété D.1.

Lemme D.2. *Pour tout $\hat{s} = (\theta, p) \in \widehat{S}$, pour tout $\vec{s}_0 \in \vec{S}$, et pour tout $v \in \mathbb{V}^\sharp$,*

$$\Gamma_\Lambda \vdash \ \gamma_{\widehat{S}}(\hat{s})(\vec{s}_0)(v) = \{\gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, \theta(v)) \mid L \in \gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p)\}$$

D.2.1 Instruction `seti`

Débutons avec l'instruction `seti`. Afin de valider son interprétation sur $\widehat{S}$ via $\hat{\mathbb{I}}$, nous cherchons à vérifier, pour tout $\hat{s} \in \widehat{S}$ et $\vec{s}_0 \in \vec{S}$, $\Gamma_\Lambda \vdash$

$$\mathbb{I}[\texttt{seti}\ d, k](\gamma_{\widehat{S}}(\hat{s})(\vec{s}_0)) \overset{?}{\subseteq} \gamma_{\widehat{S}}(\hat{\mathbb{I}}[\texttt{seti}\ d, k](\hat{s}))(\vec{s}_0)$$

c'est-à-dire :

$$\left\{\vec{s} \in \vec{S} \mid \forall v \in \mathbb{V}^\sharp,\ \vec{s}(v) \in \gamma_{\widehat{S}}(\hat{s}[v \mapsto k])(\vec{s}_0)(v)\right\} \overset{?}{\subseteq} \gamma_{\widehat{S}}(\hat{s}[v \mapsto k])(\vec{s}_0)$$

ce qui revient à démontrer :

$$\left\{\vec{s} \in \vec{S} \mid \forall v \in \mathbb{V}^\sharp,\ \vec{s}(v) \in \gamma_{\widehat{S}}(\hat{s}[v \mapsto k])(\vec{s}_0)(v)\right\} \overset{?}{\subseteq} \left\{\vec{s} \in \vec{S} \mid \exists L \in \gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p),\ \forall v \in \mathbb{V}^\sharp,\ \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, \theta[d \mapsto k](v))\right\}$$

soit, grâce au Lemme D.2 :

$$\begin{aligned}\forall v \neq d,\, \vec{s}(v) \in \{\gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, \theta(v)) \mid L \in \gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p)\} \\ \wedge\, \vec{s}(d) \in \{\gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, k) \mid L \in \gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p)\} \\ \overset{?}{\Longleftarrow} \\ \exists L \in \gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p),\, \forall v \neq d,\, \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, \theta(v)) \\ \wedge\, \vec{s}(d) \in \gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, k)\end{aligned}$$

Pour toute constante $k \in \mathbb{Z}_{32}{}^\sharp$, $\gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, k) = \{k\}$, cela revient donc à :

$$\begin{aligned}\forall v \neq d,\, \vec{s}(v) \in \{\gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, \theta(v)) \mid L \in \gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p)\} \wedge \vec{s}(d) = k \\ \overset{?}{\Longleftarrow} \\ \exists L \in \gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p),\, \forall v \neq d,\, \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, \theta(v)) \wedge \vec{s}(d) = k\end{aligned}$$

Or, L n'intervient que dans l'opérande de gauche du $\wedge$, nous avons donc :

$$\begin{aligned}\forall v \neq d,\, \vec{s}(v) \in \{\gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, \theta(v)) \mid L \in \gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p)\} \\ \Longleftrightarrow \\ \exists L \in \gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p),\, \forall v \neq d,\, \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}_\Lambda}}(\vec{s}_0, L, \theta(v))\end{aligned}$$

Nous avons ainsi prouvé que :

$$\mathbb{I}[\texttt{seti}\ d, k](\gamma_{\widehat{S}}(\hat{s})(\vec{s}_0)) = \gamma_{\widehat{S}}(\hat{\mathbb{I}}[\texttt{seti}\ d, k](\hat{s}))(\vec{s}_0)$$

□

En plus de vérifier la relation $f \circ \gamma_{\widehat{S}} \sqsubseteq \gamma_{\widehat{S}} \circ \hat{f}$ validant la correction de l'abstraction $\hat{f}$, nous obtenons pour cette instruction une égalité. Cela signifie que $\hat{\mathbb{I}}[\texttt{seti}\ d, k]$ abstrait l'instruction $\texttt{seti}\ d, k$ sans perte de précision.

D.2.2 Instruction `scratch`

La preuve de l'instruction `scratch` est très similaire à celle de `seti`, en remplaçant $\vec{s}(d) = k$ par $\vec{s}(d) = \lambda_{|\Gamma_\Lambda|+1}$ dans les formules. Le point crucial est l'indépendance de cette condition de l'ensemble des variables abstraites "permises" par $\gamma_{\widehat{P_\Lambda}}(\vec{s}_0, p)$. Elle est justifiée par simple typage : les prédicats p existent dans le contexte du langage Γ_Λ, qui ne contient

pas $\lambda_{|\Gamma_\Lambda|+1}$ ($\Gamma_\Lambda = \{\lambda_0, \lambda_1, \ldots, \lambda_{|\Gamma_\Lambda|}\}$). Effectivement, ces prédicats ont été générés avant que l'instruction `scratch` en question n'introduise le dernier $\lambda_{|\Gamma_\Lambda|+1}$. Nous constaterions donc lors de la validation d'une instruction `scratch` d :

$$\forall v \neq d,\ \vec{s}(v) \in \{\gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \theta(v)) \mid L \in \gamma_{\widehat{P}_\Lambda}(\vec{s}_0, p)\} \wedge \vec{s}(d) = \lambda_{|\Gamma_\Lambda|+1}$$
$$\iff$$
$$\exists L \in \gamma_{\widehat{P}_\Lambda}(\vec{s}_0, p),\ \forall v \neq d,\ \vec{s}(v) \in \gamma_{\widehat{\mathbb{E}}_\Lambda}(\vec{s}_0, L, \theta(v)) \wedge \vec{s}(d) = \lambda_{|\Gamma_\Lambda|+1}$$

Ce qui nous amènerait une fois de plus à vérifier $f \circ \gamma_{\widehat{S}} = \gamma_{\widehat{S}} \circ \hat{f}$. L'égalité a du sens : bien que l'instruction `scratch` introduise une perte de précision par rapport à l'instruction assembleur qu'elle traduit, l'interprétation de cette instruction sémantique par introduction d'un λ n'apporte pas elle-même d'approximation.

D.2.3 Conclusion

L'interprétation par $\hat{\mathbb{I}}$ du reste des instructions sémantiques se valide répétitivement selon le même schéma, parfois à l'aide de quelques artifices arithmétiques. La proximité des définitions des fonctions d'interprétation abstraite et concrète facilite largement les preuves, par rapport à l'interprétation par prédicats qui demande d'exhiber une fonction dont on doit prouver qu'elle applique l'inverse de l'effet désiré sur une expression. Cette simplicité, due à une forme d'interprétation "en avant" induite par le domaine $\widehat{S}$, fait partie de ses avantages.

E Concrétisation de $\widehat{S}_0$

Définition E.1. Nous définissons la concrétisation paramétrée de l'abstraction $\widehat{S}_0$ par

$$\gamma_{\widehat{S}_0} : \widehat{S}_0 \longrightarrow S \to \mathcal{P}(S)$$
$$\forall \hat{s} = (\theta, p) \in \widehat{S}_0,\ \kappa_{\text{SP}} \vdash \gamma_{\widehat{S}_0}(\hat{s}) \coloneqq s_i \mapsto \gamma_{\Theta_0}(s_i, \theta) \cap \bigcap_{p_i \in p} \gamma_{P^*}(s_i, p_i)$$

où γ_{Θ_0} est défini comme

$$\forall s_i \in S,\, \forall \theta \in \Theta_0,\ \kappa_{\text{SP}} \vdash\ \gamma_{\Theta_0}(s_i, \theta) \coloneqq \{s \mid \forall v \in \mathbb{V}^\sharp,\, s(v) \in \gamma_{\mathbb{E}}(s_i, \theta(v))\}$$

en réutilisant la fonction $\gamma_{\mathbb{E}}$ de la Définition 3.18, appelée cette fois-ci avec l'état initial s_i plutôt que l'état courant s pour obtenir les valeurs des variables contenues dans les expressions.

γ_{P^*}, elle, est définie comme suit, de manière similaire à la définition (3.18) de γ_P :

$$\forall s_i \in S,\, \forall p \in \mathcal{P}(P^*),\ \kappa_{\text{SP}} \vdash \gamma_{P^*}(s_i, p) \coloneqq$$
$$\left\{ s \in S \;\middle|\; \begin{array}{l} \forall (e_1, =, e_2) \in p,\, \exists x_1 \in X_1,\, \exists x_2 \in X_2,\, x_1 = x_2 \\ \wedge\, \forall (e_1, \neq, e_2) \in p,\, \exists x_1 \in X_1,\, \exists x_2 \in X_2,\, x_1 \neq x_2 \\ \wedge\, \forall (e_1, \leq, e_2) \in p,\, \exists x_1 \in X_1,\, \exists x_2 \in X_2,\, x_1 \leq x_2 \\ \wedge\, \forall (e_1, <, e_2) \in p,\, \exists x_1 \in X_1,\, \exists x_2 \in X_2,\, x_1 < x_2 \end{array} \;\middle|\; \begin{array}{l} \\ X_1 \coloneqq \gamma_{\mathbb{E}}(s_i, e_1) \\ X_2 \coloneqq \gamma_{\mathbb{E}}(s_i, e_2) \\ \\ \end{array} \right\}$$

On peut remarquer que la variable libre s dans la définition de γ_P est *inefficace*, c'est-à-dire qu'elle n'intervient pas dans l'équation. C'est parce que, une fois s_i connu, l'ensemble des prédicats s'évalue à une constante, *vrai*, $\top$ (les prédicats sont satisfiables), ou *faux*, $\bot$ (les prédicats présentent une contradiction). L'image de γ_{P^*} est donc restreinte à $\{\emptyset, S\}$.

F Démonstration du Lemme 4.1

Lemme 4.1. *Pour tout n, $\hat{s}_n$ décrit un ensemble de propriétés valides à l'entrée de la boucle h après au plus n ($[0, n]$) itérations de celle-ci. Autrement dit,*

$$\forall n \geq 0,\ \forall k \leq n,\ f_h^k(\hat{s}_0) \,\hat{\sqsubseteq}\, \hat{s}_n$$

c'est-à-dire que $(f_h^n)_n$ définit une chaîne ascendante.

Démonstration. Prouvons cette propriété par récursion sur n.

- *Initialisation :* Le cas $n = 0$ est trivial : $f_h^0(\hat{s}_0) = \hat{s}_0 \,\hat{\sqsubseteq}\, \hat{s}_0$.
- *Hérédité :* Supposons la propriété énoncée vraie pour une valeur m quelconque, c'est-à-dire

  $$\forall k \leq m,\ f^k(\hat{s}_0) \,\hat{\sqsubseteq}\, \hat{s}_m$$

  Or, la Propriété D.2 statue que les fonctions d'interprétation abstraite sont croissantes. La fonction f_h, définie par l'application successive de fonctions d'interprétation (et d'unions de chemins divergents) est croissante (l'union d'états de deux chemins respectant la propriété de croissance ne pose pas de problème : $\hat{s}' \,\hat{\sqsubseteq}\, \hat{s} \wedge \hat{s}'' \,\hat{\sqsubseteq}\, \hat{s} \Rightarrow \hat{s}' \,\hat{\sqcup}\, \hat{s}'' \,\hat{\sqsubseteq}\, \hat{s}$ car $\hat{s}' \,\hat{\sqcup}\, \hat{s}'' \,\hat{\sqsubseteq}\, \hat{s}'$ et $\hat{s}' \,\hat{\sqcup}\, \hat{s}'' \,\hat{\sqsubseteq}\, \hat{s}''$). Nous pouvons donc écrire :

  $$\forall k \leq m,\ f_h(f_h^k(\hat{s}_0)) \,\hat{\sqsubseteq}\, f_h(\hat{s}_m)$$

  c'est-à-dire

  $$\forall k \in [1, m+1],\ f_h^k(\hat{s}_0) \,\hat{\sqsubseteq}\, f_h(\hat{s}_m)$$

  Or, étant donné que l'union $\hat{\sqcup}$ est croissante (Lemme D.1),

  $$f_h(\hat{s}_m) \,\hat{\sqsubseteq}\, f_h(\hat{s}_m) \,\hat{\sqcup}\, \hat{s}_0$$

  Trivialement, $f_h^0(\hat{s}_0) = \hat{s}_0 \,\hat{\sqsubseteq}\, \hat{s}_0$, et donc

  $$\left.\begin{array}{rl} f_h^0(\hat{s}_0) \,\hat{\sqsubseteq}\, \hat{s}_0 & \,\hat{\sqsubseteq} f_h(\hat{s}_m) \,\hat{\sqcup}\, \hat{s}_0 \\ \forall k \in [1, m+1], f_h^k(\hat{s}_0) \,\hat{\sqsubseteq}\, f_h(\hat{s}_m) & \,\hat{\sqsubseteq} f_h(\hat{s}_m) \,\hat{\sqcup}\, \hat{s}_0 \end{array}\right\} \Longrightarrow \begin{array}{c} \forall k \in [0, m+1], \\ f_h^k(\hat{s}_0) \,\hat{\sqsubseteq}\, f_h(\hat{s}_m) \,\hat{\sqcup}\, \hat{s}_0 \end{array}$$

  La propriété énoncée est donc vraie pour $n = m + 1$.

- *Généralisation :* Pour toute valeur de $n \geq 0$, la propriété

$$\forall k \leq n,\ f_h^k(\hat{s}_0) \hat{\sqsubseteq} \hat{s}_n$$

est valide.

□

Liste des acronymes

CFG Graphe de Flot de Contrôle ou *Control Flow Graph*

WCET Temps d'Exécution Pire-Cas ou *Worst-Case Execution Time*

BCET Temps d'Exécution Meilleur-Cas ou *Best-Case Execution Time*

IPET Énumeration Implicite des Chemins ou *Implicit Path Enumeration Technique*

WCEP Chemin d'Exécution Pire-Cas ou *Worst-Case Execution Path*

ILP Programmation Linéaire en Nombres Entiers (PLNE) ou *Integer Linear Programming*

CLP *Circular-Linear Progressions*

DBM Matrices de Différence des Bornes, ou *Difference-Bound Matrices*

PCG Graphe d'Appel du Programme ou *Program Call Graph*

SMT Satisfiabilité Modulo des Théories

Bibliographie

[1] *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*. IEEE Computer Society, 2015.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[4] H.A. Aljifri, A. Pons, and M.A. Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In *Conference Proceedings of the 2000 IEEE International Performance, Computing, and Communications Conference (Cat. No.00CH37086)*, page 430, Feb 2000.

[5] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[6] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996.

[7] Peter Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems, RTS 1996, L'Aquila, Italy, June 12-14, 1996*, pages 102–107. IEEE Computer Society, 1996.

[8] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electr. Notes Theor. Comput. Sci.*, 267(1) :3–16, 2010.

[9] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 5–23, 2004.

[10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA : An open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010.

[11] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. Fast : Fast acceleration of symbolic transition systems. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[12] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[13] Peter Barth. A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. 1995.

[14] J. Benkoski, E. Vanden Meersch, L. Claesen, and H. De Man. Efficient algorithms for solving the false path problem in timing verification. In *IEEE International Conference on Computer-Aided Design*, pages 44–47, 1987.

[15] Michel Berkelaar, Kjell Eikland, Peter Notebaert, et al. lpsolve : Open source (mixed-integer) linear programming system. *Eindhoven U. of Technology*, 2004.

[16] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time system. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), Austin, Texas, USA, December 3-5, 2002*, pages 279–288. IEEE Computer Society, 2002.

[17] Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET, a tool for probabilistic WCET analysis of real-time systems. In Gustafsson [44], pages 21–38.

[18] Bernard Blackham, Mark H. Liffiton, and Gernot Heiser. Trickle : Automated infeasible path detection using all minimal unsatisfiable subsets. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 169–178. IEEE Computer Society, 2014.

[19] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. Abc : Algebraic bound computation for loops. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2010.

[20] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic smt solver. In Gupta and Malik [43], pages 294–298.

[21] Armelle Bonenfant, Hugues Cassé, Marianne De Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. FFX : a portable WCET annotation language. In Liliana Cucu-Grosjean, Nicolas Navet, Christine Rochange, and James H. Anderson, editors, *20th International Conference on Real-Time and Network Systems, RTNS '12, Pont a Mousson, France - November 08 - 09, 2012*, pages 91–100. ACM, 2012.

[22] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit : An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[23] Renato Bruni and Antonio Sassano. Finding minimal unsatisfiable subformulae in satisfiability instances. In Rina Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 495–499. Springer, 2000.

[24] Hugues Cassé, Florian Birée, and Pascal Sainrat. Multi-architecture value analysis for machine code. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, volume 30 of *OASICS*, pages 42–52. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[25] Francisco J. Cazorla, editor. *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, volume 47 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[26] Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. Exploiting branch constraints without exhaustive path enumeration. In Wilhelm [105].

[27] Vasek Chvatal. *Linear programming*. Macmillan, 1983.

[28] Antoine Colin and Stefan M. Petters. Experimental evaluation of code properties for WCET analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), 3-5 December 2003, Cancun, Mexico*, pages 190–199. IEEE Computer Society, 2003.

[29] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10) :1–8, 2001.

[30] Patrick Cousot and Radhia Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[31] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In René Jacquart, editor, *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, volume 156 of *IFIP*, pages 359–366. Kluwer/Springer, 2004.

[32] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

[33] IBM ILOG CPLEX. V12. 1 : User's manual for cplex. *International Business Machines Corporation*, 46(53) :157, 2009.

[34] Christoph Cullmann and Florian Martin. Data-flow based detection of loop bounds. In Rochange [90].

[35] Jean-François Deverge and Isabelle Puaut. Safe measurement-based WCET estimation. In Wilhelm [105].

[36] Sun Ding, Hee Beng Kuan Tan, and Kaiping Liu. A survey of infeasible path detection. In Joaquim Filipe and Leszek A. Maciaszek, editors, *ENASE 2012 - Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering, Wroclaw, Poland, 29-30 June, 2012.*, pages 43–52. SciTePress, 2012.

[37] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[38] Jakob Engblom and Andreas Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA '99), 13-16 December 1999, Hong Kong, China*, pages 88–95. IEEE Computer Society, 1999.

[39] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *STTT*, 4(4) :437–455, 2003.

[40] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3) :131–181, 1999.

[41] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2006.

[42] Denis Gopan and Thomas W. Reps. Guided static analysis. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007.

[43] Aarti Gupta and Sharad Malik, editors. *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*. Springer, 2008.

[44] Jan Gustafsson, editor. *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 - a Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July 1, 2003*, volume MDH-MRTC-116/2003-1-SE. Department of Computer Science and Engineering, Mälardalen University, Box 883, 721 23 Västerr$^*_a$s, Sweden, 2003.

[45] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks : Past, present and future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASICS*, pages 136–146. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[46] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Algorithms for infeasible path calculation. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[47] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 57–66. IEEE Computer Society, 2006.

[48] Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multi-core timing analysis. In Alain Plantec, Frank Singhoff, Sébastien Faucou, and Luís Miguel Pinho, editors, *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 299–308. ACM, 2016.

[49] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Grenoble Institute of Technology, France, 1979.

[50] David Hedley and Michael A. Hennell. The causes and effects of infeasible paths in computer programs. In Meir M. Lehman, Horst Hünke, and Barry W. Boehm, editors, *Proceedings, 8th International Conference on Software Engineering, London, UK, August 28-30, 1985.*, pages 259–267. IEEE Computer Society, 1985.

[51] Niklas Holsti. Computing time as a program variable : a way around infeasible paths. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Prague, Czech Republic, July 1, 2008*, volume 8 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[52] Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Analysing switch-case code with abstract execution. In Cazorla [25], pages 85–94.

[53] Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Worst-case execution time analysis for digital signal processors. In *10th European Signal Processing Conference, EUSIPCO 2000, Tampere, Finland, September 4-8, 2000*, pages 1–4. IEEE, 2000.

[54] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1) :26–60, 1990.

[55] Bertrand Jeannet and Antoine Miné. Apron : A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[56] Linus Källberg. Circular linear progressions in SWEET, 2014.

[57] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Analysis of the impacts of overestimation sources on the accuracy of worst case timing analysis. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*, pages 22–31. IEEE Computer Society, 1999.

[58] Johannes Kinder and Helmut Veith. Jakstab : A static analysis platform for binaries. In Gupta and Malik [43], pages 423–427.

[59] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET analysis : The annotation language challenge. In Rochange [90].

[60] Apostolos A. Kountouris. Safe and efficient elimination of infeasible execution paths in wcet estimation. In *Third International Workshop on Real-Time Computing Systems Application (RTCSA '96), October 30 - November 01, 1996, Seoul, Korea*, pages 187–194. IEEE Computer Society, 1996.

[61] Xavier Leroy et al. The CompCert verified compiler. 2004.

[62] Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information : Reconciling compiler optimizations and WCET estimation. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, page 97. ACM, 2014.

[63] Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing flow information for tighter WCET estimation : Application to vectorization. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015* [1], pages 217–226.

[64] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In Bryan Preas, editor, *Proceedings of the 32st Conference on Design Automation, San Francisco, California, USA, Moscone Center, June 12-16, 1995.*, pages 456–461. ACM Press, 1995.

[65] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In Gustafsson [44], pages 99–102.

[66] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*, pages 136–146. IEEE Computer Society, 2009.

[67] Inês Lynce and João P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

[68] Patrick C. MacGeer and Robert King Brayton. *Integrating Functional and Temporal Domains in Logic Design : The False Path Problem and Its Implications.* Kluwer Academic Publishers, 1991.

[69] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, pages 161–166. IEEE Computer Society, 2008.

[70] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. *CoRR*, abs/cs/0703073, 2007.

[71] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[72] Frank Mueller and David B. Whalley. Fast instruction cache analysis via static cache simulation. In *Proceedings 28st Annual Simulation Symposium (SS '95), April 25-28, 1995, Santa Barbara, California, USA*, pages 105–114. IEEE Computer Society, 1995.

[73] Vincent Mussot. *Automates d'annotation de flot pour l'expression et l'intégration de propriétés dans l'analyse de WCET*. PhD thesis, University Toulouse 3 Paul Sabatier, France, 2016.

[74] Vincent Mussot, Armelle Bonenfant, Pascal Sotin, Denis Claraz, and Philippe Cuenot. From relevant high-level properties to WCET computation improvement. In *International Conference on Embedded Real Time Software and Systems (ERTS2)*, page (online). SIA/3AF/SEE, 2014.

[75] Vincent Mussot, Jordy Ruiz, Pascal Sotin, Marianne De Michiel, and Hugues Cassé. Expressing and exploiting conflicts over paths in WCET analysis. In Schoeberl [94], pages 3–1.

[76] Vincent Mussot and Pascal Sotin. Improving WCET analysis precision through automata product. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015* [1], pages 207–216.

[77] Minh Ngoc Ngo and Hee Beng Kuan Tan. Detecting large number of infeasible paths through recognizing their patterns. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 215–224. ACM, 2007.

[78] Minh Ngoc Ngo and Hee Beng Kuan Tan. Heuristics-based infeasible path detection for dynamic test data generation. *Information & Software Technology*, 50(7-8) :641–655, 2008.

[79] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[80] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9 :53–58, 2014 (published 2015).

[81] David A. Patterson and John L. Hennessy. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, 1990.

[82] Mathias Péron and Nicolas Halbwachs. An abstract domain extending difference-bound matrices with disequality constraints. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference,*

VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings, volume 4349 of *Lecture Notes in Computer Science*, pages 268–282. Springer, 2007.

[83] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138, 1959.

[84] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1) :67–91, 1997.

[85] UK Rapita Systems Ltd., York. Rapitime toolkit for dynamic analysis, 2006.

[86] Pascal Raymond. A general approach for expressing infeasibility in implicit path enumeration technique. In Tulika Mitra and Jan Reineke, editors, *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, pages 8–1. ACM, 2014.

[87] Thomas W. Reps, Susan Horwitz, Shmuel Sagiv, and Genevieve Rosay. Speeding up slicing. In David S. Wile, editor, *SIGSOFT '94, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA, December 6-9, 1994*, pages 11–20. ACM, 1994.

[88] Bernhard Rieder, Peter P. Puschner, and Ingomar Wenzel. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement based WCET analysis. In Markus Kucera, Richard Roth, and Massimo Conti, editors, *International Workshop on Intelligent Solutions in Embedded Systems, WISES 2008, Regensburg, Germany, July 10-11, 2008*, pages 1–7. IEEE, 2008.

[89] Xavier Rival. Analyse statique par interprétation abstraite. *Technique et Science Informatiques*, 30(4) :371–380, 2011.

[90] Christine Rochange, editor. *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*, volume 6 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[91] Jordy Ruiz and Hugues Cassé. Using smt solving for the lookup of infeasible paths in binary programs. In Cazorla [25], pages 95–104.

[92] Inc. Safety Research & Strategies. Toyota unintended acceleration and the big bowl of "spaghetti" code. http ://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-"spaghetti"-code, 2013.

[93] Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Faster WCET flow analysis by program slicing. In Mary Jane Irwin and Koen De Bosschere, editors, *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Ontario, Canada, June 14-16, 2006*, pages 103–112. ACM, 2006.

[94] Martin Schoeberl, editor. *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[95] Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *5th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2007), May 30 - June 1st, Nice, France*, pages 39–48, 2007.

[96] Rathijit Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In Christoph M. Kirsch and Reinhard Wilhelm, editors, *Proceedings of the 7th ACM & IEEE International conference on Embedded software, EMSOFT 2007, September 30 - October 3, 2007, Salzburg, Austria*, pages 203–212. ACM, 2007.

[97] Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 185–195. IEEE Computer Society, 2016.

[98] Ingmar Stein and Florian Martin. Analysis of path exclusion at the machine code level. In Rochange [90].

[99] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In Ellen Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 358–363. ACM, 2006.

[100] Wei-Tsun Sun and Hugues Cassé. Dynamic branch resolution based on combined static analyses. In Schoeberl [94], pages 8–1.

[101] Henrik Theiling. Extracting safe and precise control flow from binaries. In *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000), 12-14 December 2000, Cheju Island, South Korea*, pages 23–30. IEEE Computer Society, 2000.

[102] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[103] Sebastian Unger and Frank Mueller. Handling irreducible loops : Optimized node splitting versus dj-graphs. *ACM Trans. Program. Lang. Syst.*, 24(4) :299–333, jul 2002.

[104] Mark Weiser. Program slicing. In Seymour Jeffrey and Leon G. Stucki, editors, *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.*, pages 439–449. IEEE Computer Society, 1981.

[105] Reinhard Wilhelm, editor. *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 5, 2005, Palma de Mallorca, Spain*, volume 1 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[106] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3) :36–1, 2008.

[107] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. *SAT*, 3, 2003.

[108] Jakob Zwirchmayr, Pascal Sotin, Armelle Bonenfant, Denis Claraz, and Philippe Cuenot. Identifying relevant parameters to improve WCET analysis. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, volume 39 of *OASICS*, pages 93–102. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.