ORSAY
N^o d'ordre : 9054

UNIVERSITÉ DE PARIS-SUD 11

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

le grade de docteur en sciences DE L'UNIVERSITÉ PARIS XI

PAR

Thierry HUBERT

—×—

SUJET :

Analyse statique et preuve de programmes industriels critiques

soutenue le 20 juin 2008 devant la commission d'examen

MM. Jean Goubault-Larrecq, président
Xavier Leroy, rapporteur
Claude Marché
Jean-François Monin, rapporteur
Peter Müller
Dillon Pariente

Remerciements

Je remercie chaleureusement **Claude Marché** d'avoir encadré ce travail de thèse, avec beaucoup de compétence, d'enthousiasme et de disponibilité. Merci Claude pour tes conseils, ton optimisme et la confiance que tu m'as accordée au cours de ces années.

Je souhaite exprimer toute ma gratitude à **Dillon Pariente** pour ses conseils avisés.

Je tiens à remercier vivement **Dillon Pariente** et **Emmanuel Ledinot** pour votre accueil chaleureux chez **Dassault Aviation** et pour tous les moyens qu'ils ont mis à ma disposition.

Je tiens à exprimer ma profonde reconnaissance à Messieurs **Xavier Leroy** et **Jean-François Monin** qui ont accepté de juger ce mémoire avec beaucoup d'attention, ainsi qu'à Messieurs **Jean Goubault-Larrecq** et **Peter Müller** qui ont accepté d'en être les examinateurs.

Un grand merci à toute l'équipe **Proval**, pour son accueil chaleureux durant ces années, qui a su créer une ambiance très sympathique d'entraide.

Table des matières

Chapitre 1

Introduction

De nos jours les ordinateurs sont présents dans de plus en plus de secteurs de la vie quotidienne. Que ce soit dans les caisses de supermarchés, dans les avions ou tout simplement dans une cafetière, les «calculateurs électroniques» ont une place de plus en plus importante. Or, au cours de l'histoire de plus en plus de bugs, défaut de conception ou de réalisation se manifestant par des anomalies de fonctionnement, sont apparus avec des conséquences plus ou moins grâves. Par exemple, entre 1985 et 1987, le logiciel de contrôle de l'appareil de radiothérapie Therac-25 fut impliqué dans au moins six accidents durant lesquels des patients reçurent des doses massives de radiation. Au moins cinq patients décédèrent des suites de l'irradiation [82].

Les causes des dysfonctionnements peuvent être multiples et difficiles à identifier. On peut les classer en deux catégories. La première catégorie est celle des dysfonctionnements liés au matériel. Par exemple, en juin 1944, le premier bug répertorié est dû à des insectes qui grillèrent sur les tubes à vide chauds. La deuxième catégorie est celle des dysfonctionnements liés à la programmation. Par exemple, en 1996, la fusée européenne Ariane V a explosé 37 secondes après le décollage. Ce bug est dû à une anomalie dans un élément du programme réutilisé d'Ariane IV et qui ne devait avoir -en théorie- aucun effet sur le guidage [83]. Ensuite, il existe des problèmes dûs à l'association de plusieurs de ces facteurs. Par exemple, en 1962, la fusée Mariner I a dévié de sa trajectoire à cause d'une panne matérielle et d'un bug dû à une erreur humaine de recopie de code [15].

L'informatique est également utilisée dans des domaines à risques majeurs, tels que dans les centrales nucléaires ou les avions... Ces programmes sont dits critiques car ils peuvent menacer la vie humaine et qu'ils nécessitent donc plus de validation et de vérification en regard de leurs spécifications et de leurs exigences.

En premier lieu, on peut se demander comment valider un programme. La première réponse, naïve, est de tester le programme avec *toutes* les données d'entrée possibles, valides ou non. Pour mesurer la tâche, on peut constater que :

- une variable de type entier est généralement codée sur 32 chiffres binaires, ce qui fait un peu plus de $2\ 147\ 483\ 648$ entrées à tester.
- s'il y a deux entiers, on obtient : $4,61 \times 10^{18}$ cas potentiels de test.

En l'état actuel des moyens de calcul, il est donc évident que l'on ne peut vérifier cela

exhaustivement dans un temps raisonnable. Reste donc à utiliser un programme pour tester ce programme. Mais deux problèmes font leur apparition dans cette approche :

1. Si l'on pouvait concevoir un tel programme, il faudrait déjà qu'il ne comporte pas d'erreur lui-même.
2. À la fin des années 30, l'anglais Alan Turing envisageait de créer un programme capable de vérifier si un autre programme risquait de ne pas se terminer. Ces travaux concluaient qu'un tel programme de vérification ne pouvait exister : c'est un problème indécidable.

1.1 Méthodologies de conception de logiciels

La première solution pratique pour renforcer la qualité du developpement logiciel est passé par la proposition de méthodologie de conception. Ainsi, dans les années 70 et 80, des méthodes graphiques de conception ont fait leur apparition :

- SADT (en anglais Structured Analysis and Design Technique) — connue aussi sous le label IDEF [12] (en anglais Integration DEfinition for Function modeling) — est une méthode d'origine américaine, développée par Softech en 1977 puis introduite en Europe à partir de 1982 par Michel Galiner. Elle se répandit vers la fin des années 1980 comme l'un des standards de description graphique d'un système complexe par analyse fonctionnelle descendante, c'est-à-dire que l'analyse chemine du général (dit "niveau A0") vers le particulier (dits "niveaux Aijk"). SADT est une démarche systémique de modélisation d'un système complexe ou d'un processus opératoire.
- Merise [111, 112, 106] est une méthode d'analyse, de conception et de gestion de projet complètement intégrée, ce qui en constitue le principal atout. Elle a fourni un cadre méthodologique et un langage commun et rigoureux à une génération d'informaticiens français.
- UML [19] (en anglais Unified Modeling Language, « langage de modélisation unifié ») est un langage graphique de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la modélisation objet utilisée en génie logiciel. Principalement issu des travaux de Grady Booch, James Rumbaugh et Ivar Jacobson, UML est un standard défini par l'OMG (Object Management Group). L'OMG travaille actuellement sur la version UML 2.1. Il est l'accomplissement de la fusion des précédents langages de modélisation objet Booch, OMT et OOSE.

Les méthodes graphiques nous invitent à découper, cloisonner, concevoir des scénarii, des jeux de tests, des simulations. On essaie de formaliser tout ce qui peut l'être par des schémas, traduits ensuite parfois en langage formel, ce qui permet de générer automatiquement du code C, C++ ou Java. Ces méthodes sont d'une grande aide lors de la conception de logiciels complexes ou volumineux. Dans ces méthodes, rien ne permet de valider l'analyse de manière formelle, c'est-à-dire de manière certaine. L'apparition d'une erreur à l'exécution ou la violation d'une propriété fonctionnelle est donc toujours possible. Cependant, ces méthodes sont les plus utilisées aujourd'hui en génie logiciel de par leur facilité d'apprentissage.

1.2 Méthodes Formelles

Les méthodes graphiques, bien que faciles à utiliser, ne certifient pas l'absence d'erreurs dans un programme. Pour combler ce manque, des méthodes logiques, formelles, ont été introduites. Ce sont des méthodes qui utilisent la logique pour obtenir une preuve qu'un programme est correct. On peut séparer ces méthodes en deux familles : les méthodes dynamiques et statiques.

1.2.1 Méthodes dynamiques

Les méthodes dynamiques représentent l'ensemble des méthodes formelles qui vérifient pendant l'exécution du programme que celui-ci respecte certaines règles, exigences ou propriétés. On peut séparer ces méthodes en deux sous-ensembles : le test formel et la vérification dynamique d'assertions.

1.2.1.1 Test formel

Cette méthode a pour but de vérifier certaines propriétés d'un programme en le testant sur un ensemble de valeurs d'entrées bien choisies de manière à couvrir l'ensemble des erreurs possibles [93]. Malgré cela, le test reste une méthode non exhaustive.

Le test unitaire : En programmation, le test unitaire est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion de programme sur un certain nombre de cas d'exécution.

Il s'agit pour le programmeur de tester un module, indépendamment du reste du programme, ceci afin de s'assurer qu'il répond aux spécifications fonctionnelles et qu'il fonctionne correctement. Cette vérification est considérée comme essentielle, en particulier dans les applications critiques. La métrique de sa complétude s'exprime alors en terme de couverture du code. Cela consiste à s'assurer que le test conduit à exécuter l'ensemble (ou une fraction déterminée) des instructions ou des branches de contrôles présentes dans le code à tester.

L'ensemble des tests unitaires doit être rejoué après une modification du code afin de vérifier qu'il n'y a pas de régressions (apparition de nouveaux dysfonctionnements).

Dans les applications non critiques, l'écriture des tests unitaires a longtemps été considérée comme une tâche secondaire. Cependant, la méthode eXtreme Programming (XP) a remis les tests unitaires, qu'elle nomme maintenant Tests du Programmeur, au centre de l'activité de programmation.

La méthode XP préconise d'écrire les tests en même temps, voire même avant la fonction à tester (Test Driven Development). Ceci permet de définir précisément l'interface du module à développer. En cas de découverte d'un bug logiciel, on écrit la procédure de test qui reproduit le bug. Après correction on relance le test, qui ne devrait plus indiquer aucune erreur.

Le test fonctionnel : Cette méthode revient à générer des tests en fonction d'un modèle du programme.

Ce modèle est habituellement une abstraction de système développé. Les cas de tests issus de ce modèle sont des tests fonctionnels du même niveau d'abstraction que le modèle. Ces cas de test sont connus sous le nom de "abstract test suite". Ils ne peuvent être exécutés directement sur le système à vérifier, car ils sont sur un niveau d'abstraction inadapté. Donc une suite de tests exécutables qui peut communiquer avec le système en cours de test doit être tirée de cette "abstract test suite". Il s'agit là d'une relation entre les tests abstraits et les tests concrets adaptés à l'exécution.

Il existe différentes façons d'extraire des tests à partir d'un modèle mais seules l'expérimentation et quelques heuristiques permettent d'obtenir des résultats satisfaisants. Il est fréquent de consolider tous les essais de dérivation sous la forme d'un ensemble de "test requirements", "test purpose" ou "use case". Ces informations peuvent contenir par exemple des précisions sur la partie du modèle qui devrait être le centre des essais, ou sur les conditions dans lesquelles il est juste d'arrêter les essais (critères d'arrêt des tests).

Du fait que les suites de tests sont dérivées du modèle et non pas du code source, le modèle de test est généralement considéré comme une forme de boîte noire. Notons qu'il est toutefois possible d'extraire un modèle à partir du code source existant.

1.2.1.2 La vérification dynamique d'assertions

Cette méthode a pour but de vérifier certaines propriétés d'un programme en ajoutant dans le code des assertions qui seront vérifiées lors de l'exécution de ce code. Au moment de l'exécution le code ainsi obtenu renverra une erreur plutôt que d'exécuter une action interdite. Le cas le plus simple d'une telle approche est l'introduction d'une instruction `assert` pour tester une valeur booléenne. Cette idée de vérification dynamique a été renforcée en particulier par la vérification d'invariants. Deux langages entre autres étendent une telle approche avec l'ajout des invariants de classe, des pré-conditions et post-condtions :

Eiffel [87] : Eiffel a été développé initialement par Eiffel Software, une société fondée par Bertrand Meyer (à l'origine appelée Interactive Software Engineering Inc ISE). Le livre "Object-Oriented Software Construction" [88] présente les détails des concepts et des théories sur les objets, qui a abouti à la conception d'Eiffel.

L'objectif de ce langage, des bibliothèques et de la méthode de programmation est de créer des modules de façon fiable et réutilisables. Sa contribution la plus importante à l'ingénierie logicielle est la conception par contrat, dans laquelle des assertions, pré-conditions, post-conditions et invariants de classe sont utilisés pour aider à assurer l'exactitude du programme.

La conception d'Eiffel est basée sur la théorie de la programmation orientée objet, avec seulement quelques influences d'autres paradigmes ou de préoccupations pour la maintenance de code existant. Le langage possède des supports formels pour les types abstraits de données. Pour suivre la conception d'Eiffel, un logiciel doit être en mesure de reproduire la conception de sa documentation depuis le programme lui-même.

JML [78] : JML est un langage de spécification pour le langage Java. Il permet de décrire formellement les comportements possibles d'une classe Java. JML hérite des idées du langage Eiffel, de Larch et du calcul de raffinement, dans le but de fournir une sémantique formelle et rigoureuse, tout en étant accessible à tout programmeur Java. Divers outils font usage de l'information supplémentaire afin de vérifier certaines propriétés des programmes. Du fait que les annotations JML ont la forme de commentaires Java, elles peuvent être intégrées dans le code Java ou dans des fichiers séparés.

1.2.2 Méthodes statiques

Les méthodes statiques représentent l'ensemble des méthodes formelles qui vérifient, avant l'exécution d'un programme, que celui-ci respecte certaines propriétés. On peut séparer ces méthodes en deux sous-ensembles : méthodes utilisant une abstraction et les méthodes déductives.

1.2.2.1 Méthodes utilisant une abstraction

Ces méthodes ont besoin d'une abstraction du programme, mais une fois cette abstraction fournie elles travaillent de façon automatique pour assurer la validité du programme. Le principal défaut de ces méthodes est l'importance du choix de l'abstraction ; en effet il est courant de voir apparaître des «fausses alarmes», c'est-à-dire des points du programme abstrait identifiés comme violant une des propriétés à vérifier, alors que cette propriété est valide à ce point dans le programme concret. Les deux principales méthodes sont :

Le model-checking [100] : Il s'agit d'une technique automatique, principalement envisagée pour vérifier les systèmes concurrents avec un nombre d'états finis. Elle est basée sur la simulation, le test et le raisonnement déductif. Cette méthode est utilisée en pratique pour vérifier des circuits complexes et des protocoles de communication. Le principal problème du model-checking est l'explosion du nombre d'états. Ce problème apparaît dans les systèmes avec beaucoup de composants qui peuvent interagir les uns avec les autres ou les systèmes avec des structures de données qui peuvent avoir beaucoup de valeurs différentes. Durant les 15 dernières années des progrès considérables ont été réalisés sur ce problème. Cette approche est implémentée par des outils comme Alloy [1], CWB [9], Csml and Mcb [8], Smv [17], LoTREC [14], Spot [18], APMC [2], Blast [3] et CADP [64, 63].

L'interprétation abstraite [43] : Étant donné un langage de programmation ou de spécification, l'interprétation abstraite consiste à déterminer des sémantiques liées par des relations d'abstraction. La sémantique la plus précise, décrivant l'exécution réelle du programme de manière fidèle, est appelée sémantique concrète. Par exemple, la sémantique concrète d'un langage de programmation impératif peut associer à chaque programme l'ensemble des traces qu'il produit - une trace d'exécution étant une séquence des états consécutifs possibles de l'exécution d'un programme ; un état étant constitué de la valeur du compteur de programme et des allocations mémoire utilisées (globale, pile et tas). Des sémantiques plus abstraites en sont alors déduites ; par exemple, on pourra ne considérer que

l'ensemble des états atteignables lors des exécutions (ce qui revient à ne considérer que les derniers états des traces). Les principaux outils utilisant cette méthode sont ASTRÉE [32] et Polyspace [16].

Pour permettre l'analyse statique, certaines sémantiques calculables doivent être déduites. Par exemple, on pourra choisir de représenter l'état d'un programme manipulant des variables entières en ignorant les valeurs proprement dites et en ne conservant que leurs signes (+, - voire 0). Pour certaines opérations élémentaires comme la multiplication, une telle abstraction ne perd pas de précision : pour connaître le signe d'un produit, il est suffisant de connaître le signe des opérandes. Pour d'autres opérations en revanche, l'abstraction perdra de la précision : ainsi, il est impossible de connaître le signe d'une somme dont les opérandes sont de signe contraire par le seul biais d'abstractions de haut niveau.

De telles pertes de précision ne peuvent pas, en général, être évitées lorsque l'on utilise des sémantiques décidables. Il y a, en général, un compromis à faire entre la précision de l'analyse et sa faisabilité, que ce soit vis-à-vis de la calculabilité ou de la complexité.

1.2.2.2 Méthodes déductives

Ces méthodes ont pour but d'obtenir un programme vérifiant certaines propriétés tout en restant le plus proche du programme possible. Dans une telle approche il n'y a pas d'équivalent des «fausses alarmes» : elles demandent de faire la preuve de ce qu'on appelle des conditions de vérification directement sur le programme concret. L'inconvénient est que cette approche est incomplète : une propriété peut être vraie mais le programme n'est pas suffisament annoté pour la prouver. Ces méthodes peuvent être découpées en deux grands groupes :

Développement de programmes corrects par constructions Ces méthodes permettent de développer la preuve et le programme en même temps. Le principal défaut de cette approche est le fait de ne pouvoir vérifier un code déjà existant. On peut mentionner ici deux approches de ce type.

D'une part, les méthodes basées sur l'isomorphisme de Curry-Howard [70] font correspondre les preuves et les programmes fonctionnels purs. Le fait que les preuves soient des programmes a été découvert en 1930 par Arend Heyting, et ces questions ont été développées par lui-même, puis par L/E/J. Brouwer (maître de Heyting et initiateur de la logique intuitionniste) et plus tard par Kolmogorov. Le principe est connu sous le nom de « sémantique de Heyting » ou encore « interprétation de Brouwer-Heyting-Kolmogorov ». Plus récemment, en 1980, à l'occasion du 80ème anniversaire d'Haskell Curry (le logicien dont le prénom est devenu le nom d'un langage fonctionnel), Howard a énoncé un principe connu aujourd'hui sous le nom d'isomorphisme de Curry-Howard, qui stipule une parfaite similitude (isomorphisme) entre preuves et programmes : un programme fonctionnel est une preuve de sa spécification formelle. Des outils comme Coq [31] et LEGO [79] sont basés sur cette approche.

D'autre part, on peut citer les méthodes reposant sur le raffinement. Un des outils les plus connus dans le domaine est l'atelier B. La méthode B a été introduite par Jean-Raymond Abrial [20], à partir des travaux de E. W. Dijkstra et de C. A. R. Hoare. Le langage

B est une évolution du langage Z, adapté à une utilisation industrielle, et prenant en compte l'ensemble du cycle de développement d'un logiciel. Le langage B est fondé sur le concept mathématique de la théorie des ensembles. Il permet de décrire un programme à travers des étapes successives de raffinement. Il est possible de prouver de manière automatisée que les propriétés exprimées sont cohérentes et préservées par le modèle. La preuve mathématique que ces propriétés sont respectées au fur et à mesure des étapes de conception est garantie. Cette méthode permet :

- d'obtenir des spécifications techniques et des cahiers des charges clairs, structurés, cohérents et sans ambiguïté,
- de développer des logiciels garantis contractuellement sans défaut dans des domaines tels que le temps réel, les automatismes industriels, les protocoles de communication, les protocoles cryptographiques, l'informatique embarquée.

La caractéristique fondamentale de cette méthode est que les logiciels développés sont corrects par construction. La méthode B a été appliquée avec succès sur la réalisation de logiciels critiques de sécurité pour de grands projets industriels dans le domaine du ferroviaire [30].

Vérification par preuve de code source pré-existant Ces méthodes sont basées sur la logique de Floyd-Hoare [62, 69] et permettent de valider un programme déjà existant. Elles s'appuient sur des langages de spécification permettant d'annoter le code des programmes avec diverses formes d'assertions, en particulier des pré-conditions, post-conditions et des invariants de boucle, et cherchent à calculer les conditions de vérification qui permettront de garantir, pour un programme donné, que sa post-condition sera satisfaite. Bien que ces méthodes ne peuvent générer de «fausses alarmes», leur manque d'automatisme les rendent plus difficiles à utiliser. En effet, il faut annoter manuellement les programmes. De nombreux outils utilisant ces méthodes existent : Spec# [26], ESC/JAVA [10], Jack [13], KeY [22] et enfin la plate-forme Why [60] que nous avons utilisé dans cette thèse et dont nous détaillons le fonctionnement dans la section suivante.

1.3 La plate-forme Why

Les outils dont nous parlerons principalement dans ce mémoire sont ceux de la plate-forme Why. Ces outils reposent sur un calcul de plus faible pré-condition [51].

La plate-forme Why est un environnement pour la preuve de propriétés fonctionnelles de programmes C et Java. Les comportements attendus sont spécifiés par des contrats associés aux fonctions C et aux méthodes Java, sous forme de pré-conditions, post-conditions et de clauses décrivant les effets de bords.

La plate-forme Why est composée de trois éléments présentés ci-après.

1.3.1 L'outil Why

L'outil Why [56], issu de la thèse de Jean-Christophe Filliâtre [54], prend en entrée des programmes annotés et produit en sortie des conditions de vérification. Bien qu'il utilise

le calcul de plus faible pré-condition à la Dijkstra, il se fonde sur une technologie et sur quelques choix de conception qui sont originaux.

D'abord, Why ne vient pas avec son propre outil de preuve ; au lieu de cela, il produit des conditions pour un certain nombre d'outils de preuve existants. Cela est particulièrement important, car écrire un prouveur efficace est une activité du seul ressort d'experts du domaine. Actuellement Why décharge ses conditions de vérification vers quatre assistants de preuves (Coq [40], HOL Light[68], Isabelle/HOL [74], et PVS [99]) et au moins huit prouveurs automatiques (CVC3 [28], CVC Lite [27], Ergo [39], HOL 4 [97], haRVey [52], Simplify [49], Yices [47] et Z3 [46]). Ajouter une sortie pour un nouvel outil de preuve ne nécessite normalement qu'un «pretty-printer» et quelques fois un codage du polymorphisme [42].

Si le fait d'ajouter facilement des outils externes de preuve peut augmenter la confiance dans le procédé de vérification, il est également important d'apporter la preuve que l'outil de vérification lui-même est digne de confiance. C'est particulièrement important quand l'outil décharge quelques conditions de vérification lui-même ou implique des traitements complexes pour certaines constructions du langage (les terminaisons abruptes en C ou les exceptions en JAVA, ML ...). Une solution est de prouver le procédé de vérification à l'intérieur d'un assistant de preuve. Why adopte une approche où une preuve que le programme satisfait ses spécifications est vérifiée a posteriori, une fois que toutes les conditions de vérification sont déchargées. Cette vérification est purement automatique (vérification de typage en Coq).

Enfin, Why fournit son propre langage d'entrée, vers lequel on peut compiler des langages de programmation existants. Ce langage est proche du langage ML avec des traits impératifs (des références et des tableaux), des exceptions et des annotations. Dans la tradition de ML, le langage de Why mélange les expressions, les instructions, la liaison avec `let` et les fonctions dans une seule classe syntaxique, ce qui diminue les manipulations symboliques et limite le nombre de cas à considérer pour le calcul de plus faible pré-condition ou les conditions de vérification. De même, les exceptions sont utilisées pour traiter les arrêts abrupts tels que le `return`, le `break` ou le `continue` lors de la traduction de programmes C ou JAVA et il n'y a ainsi aucun besoin de définir des règles spéciales pour ces constructions ; les règles sur les exceptions suffisent. Cet outil sera présenté en détail dans la section 2.3.1.

1.3.2 L'outil Krakatoa

L'outil Krakatoa [61] est un prototype pour la vérification de programmes Java annotés en JML. Il est construit sur l'outil Why. L'outil Krakatoa traduit la sémantique opérationnelle des programmes Java en programme Why et les spécifications JML en assertions. L'outil Krakatoa a aussi besoin de produire une théorie correspondant au programme qui représente la mémoire et les informations dynamiques de typage. Le modèle mémoire de la première version de Krakatoa était construit sur un unique tas mémoire où les objets et les tableaux étaient stockés et la théorie était seulement générée pour l'assistant de preuve Coq. L'outil fut utilisé avec succès pour prouver des petits exemples comme l'algorithme du drapeau Hollandais de Djisktra ou les propriétés de base d'une applet de JAVACARD,

produite par la compagnie Schlumberger, pour le projet IST VerifiCard [37]. L'architecture de preuve modulaire apparaît bien adaptée pour traiter un large éventail de programmes, toutefois le travail manuel nécessaire afin de faire la preuve des conditions de vérifications des programmes en Coq est assez compliqué à cause du manque d'automatisation et de la représentation mémoire trop naïve, qui ne prend pas suffisamment en compte les informations de typage statique.

Pour faire face à ce problème, le modèle mémoire de Krakatoa a été modifié pour adopter un modèle mémoire plus local. Cette approche alternative, déjà présentée par Burstall [36], est mise en avant par Bornat [33] puis utilisée par Mehta and Nipkow [85]. Elle s'adapte parfaitement aux champs d'objet Java. En effet, la case correspondante à un champ ne peut être consultée qu'en utilisant le nom de ce champ. De plus, cette approche est étendue aux tableaux Java et elle peut supporter les nouvelles allocations mémoire. Des approches similaires sont utilisées dans l'outil ESC/JAVA [10] ou l'outil JACK [13].

Une autre amélioration fut de fournir une théorie du premier ordre pour les aspects logiques du programme afin d'utiliser les prouveurs du premier ordre comme Simplify [49] ou Ergo [39] pour prouver les conditions de vérification. Construire une théorie de premier ordre pour les programmes Java n'est pas simple et il faut utiliser Coq pour valider les axiomes du modèle du premier ordre afin de s'assurer de la consistance de la théorie construite [84].

1.3.3 L'outil Caduceus

Comme Krakatoa, l'outil Caduceus a pour but de vérifier les programmes, mais en langage C. Il repose sur l'expérience de Krakatoa et donc reprend les même avancées au niveau du modèle de la mémoire ou de la sortie vers les prouveurs. Comme Krakatoa, Caduceus traduit la sémantique opérationnelle des programmes en C vers des programmes en Why. L'outil Caduceus est présenté en détails dans un chapitre 2 qui lui est dédié.

1.4 Démarche suivie pendant la thèse

Le chapitre 2 présente l'outil Caduceus, ses fonctionnalités et limitations identifiées avant le début de ces travaux de thèse. Nous exposerons tout d'abord les objectifs et les principes d'utilisation de l'outil Caduceus, puis comment, à l'aide d'un langage de spécification propre à l'outil Caduceus, il est possible de spécifier les programmes. Ensuite, nous examinerons les différentes étapes dans le traitement d'un programme C par Caduceus, et les limites de l'approche.

Le chapitre 3 présente une étude de cas effectué entre septembre et décembre 2004 : l'algorithme de Schorr-Waite. Après une présentation de cet algorithme, nous préciserons les raisons à l'origine de ce choix d'étude de cas. Nous présenterons ensuite comment l'algorithme est écrit en C et comment nous l'avons spécifié à l'aide du langage de spécification de Caduceus. Ensuite nous exposerons les différentes étapes dans la preuve de cet algorithme. Enfin, nous comparerons cette preuve avec les autres preuves existantes (réalisées par ailleurs). Ce travail a été publié à la conférence SEFM'05 [71].

Une étude de cas analysée chez Dassault Aviation a été le fil conducteur de ces travaux de thèse. Il s'agit d'un programme critique déstiné à être embarqué, dans une version de développement non encore validée, de 70 000 lignes de code compris dans 350 fonctions. Nous avons commencé à tester l'outil Caduceus, entre janvier 2005 et avril 2005, sur l'étude de cas. Cela nous a amené à réparer plusieurs bugs de l'outil. Par exemple :

- une capture de noms entre deux champs de deux structures différentes ayant le même nom,
- un bug provoquant une trop grande taille des conditions de vérifications,
- l'instruction `switch` qui n'était pas supportée.

Pour traiter plusieurs de ces bugs ainsi que pour simplifier l'analyse du langage C, le chapitre 4 présente un langage intermédiaire ajouté dans Caduceus qui a pour but de factoriser les expressions ayant des sémantiques équivalentes, mais des syntaxes différentes. Nous présenterons donc comment supprimer les opérateurs $\&$, $*$ et les notations $t[i]$, $t.f$. Puis nous exposerons comment l'instruction `switch` est remplacée par des conditions classiques (`if`, `then`, `else`). Enfin nous examinerons les limitations de cette transformation.

Ensuite, le principal problème restant était d'annoter manuellement l'ensemble des 350 fonctions de l'étude de cas. Deux types d'annotations manquaient :

- Premièrement les annotations qui touchent à la séparation entre pointeurs. Entre avril 2005 et juin 2006, nous avons mis en place des analyses de séparation des pointeurs. Ceci sera présenté dans le chapitre 5. Notre première version de cette analyse, par génération automatique de prédicat, nous a conduits à une explosion combinatoire bloquante. Une deuxième version, à l'aide d'une analyse statique de la séparation, a permis de réduire un peu cette explosion combinatoire, puis beaucoup plus à l'aide d'une troisième version de l'analyse, utilisant un polymorphisme de région. Nous avons validé cette approche sur un code significatif extrait de l'étude de cas analysée chez Dassault Aviation. Cet extrait est un code de 3000 lignes et 21 fonctions. Ce travail a été publié à la conférence HAV'07 [73].
- Deuxièmement les annotations qui touchent à l'absence de menace. Pour résoudre ce problème, un outil, pour générer automatiquement les annotations des programmes, a été développé et est présenté section 7.1.

Entre juin 2006 et août 2007, nous avons essayé notre approche sur l'étude de cas au complet. Un des principaux problèmes restant est que certaines conditions de vérifications ne sont pas prouvées. Ce problème, issu du calcul de plus faible pré-condition : la multiplication des hypothèses inutiles dans les conditions de vérification, est exposé au chapitre 6. Nous y présenterons donc comment à l'aide de deux méthodes différentes de sélection d'hypothèses, nous avons obtenu une approche permettant de simplifier les conditions de vérification afin de faciliter la tâche des prouveurs. Ce travail a été réalisé en collaboration avec Jean-François Couchot, alors en post-doctorat dans l'équipe.

Le chapitre 7 quant à lui présente la validation des travaux de recherche sur l'étude de cas fil conducteur de la thèse. Nous exposerons donc la méthode utilisée pour valider ce code embarqué. Dans un premier temps nous présenterons comment sont générées les annotations du code, puis comment nous avons réussi à améliorer les résultats à l'aide de l'analyse de séparation obtenue au chapitre 5 et la sélection d'hypothèses du chapitre 6.

Chapitre 2

Préliminaires

Ce chapitre préliminaire présente l'approche de preuve de programmes C qui a servi de base aux travaux de cette thèse. Cette approche correspond à ce qui était implémenté dans l'outil Caduceus à la fin de l'année 2004 (date de début de ces travaux de thèse).

L'outil Caduceus est un outil de vérification de programmes C annotés qui utilise l'outil Why [55] pour décharger ses conditions de vérifications sur un ou plusieurs assistants de preuve ou procédures de décision. Dans ce chapitre nous verrons tout d'abord dans la section 2.1, les objectifs de Caduceus et ses principes d'utilisation, puis dans la section 2.2, son langage de spécification. Enfin dans la section 2.3, nous verrons les fondements du fonctionnement interne de Caduceus.

2.1 Objectifs et principes d'utilisation de Caduceus

Dans cette section nous verrons les objectifs de l'outil Caduceus ainsi que son fonctionnement du point de vue de l'utilisateur.

2.1.1 Objectifs de Caduceus

Dans l'outil Caduceus les propriétés qui doivent être vérifiées sont de deux sortes : premièrement l'absence de menaces (division par 0, déréférencement de pointeur nul ou dépassement des bornes d'un tableau) et deuxièmement il doit satisfaire les propriétés fonctionnelles données comme post-conditions des fonctions. Ces deux propriétés ne sont pas indépendantes, car elles nécessitent l'insertion d'annotations appropriées (pré-, post- conditions de fonctions, invariants globaux, invariants de boucle, ...) ce qui est habituel dans le cadre de la logique de Hoare. En pratique, ces annotations sont insérées dans le code source à l'aide de commentaire de la forme `/*@...*/`. Le langage de spécification utilisé dans ces commentaires est largement inspiré par le Java Modeling Language(JML) [35]. Les différences entre ces langages sont principalement dues à la méthode de vérification. En effet, il est nécessaire d'exécuter le code en JML, tandis que nous nous intéressons à de la vérification statique.

Une fois le programme C annoté par l'utilisateur, l'exécution de Caduceus sur le code

source fournit des conditions de vérification : ce sont des formules logiques du premier ordre dont la validité implique la sûreté du programme c'est-à-dire l'absence de menaces et la vérification des propriétés fonctionnelles. Une fois ces conditions de vérification générées on utilise un prouveur pour établir leurs validités. L'outil Caduceus est indépendant du prouveur utilisé.

Une part significative du C ANSI est supportée. Toutes les structures de contrôles sont acceptées à part le `goto`. Les programmes avec pointeurs sont supportés, incluant l'arithmétique de pointeur et la possibilité d'aliasing de pointeur. Le principal trait non supporté est le `cast` de pointeur (par exemple le codage des fonctions polymorphiques à l'aide d'un pointeur sur `void`), les unions et les pointeurs de fonction.

2.1.2 Principes d'utilisation de Caduceus

Dans cette section nous illustrons l'utilisation de Caduceus sur un petit exemple qui modélise un porte-monnaie électronique (le langage d'annotation sera décrit plus en détail dans la section 2.2).

2.1.2.1 Post-conditions et absence de menace

Voici le code de notre exemple :

```
typedef struct {
  int balance;
} purse;

/*@ requires  s >= 0
  @ ensures p->balance == \old(p->balance) + s
  @*/
void credit(purse *p,int s) {
  p->balance = p->balance + s;
}

/*@ requires 0 <= s <= p->balance
  @ ensures p->balance == \old(p->balance) - s
  @*/
void withdraw(purse *p,int s) {
  p->balance = p->balance - s;
}
```

Sur cet exemple, la structure `purse` représente le porte-monnaie électronique, la fonction `credit` ajoute de l'argent sur le porte-monnaie et la fonction `withdraw` retire de l'argent du porte-monnaie. On remarque que juste avant chacune des fonctions un commentaire commençant par `/*@` est présent. Ces commentaires sont lus par Caduceus et spécifient la fonction qui suit. Le mot-clé `requires` introduit un prédicat appelé la pré-condition de la

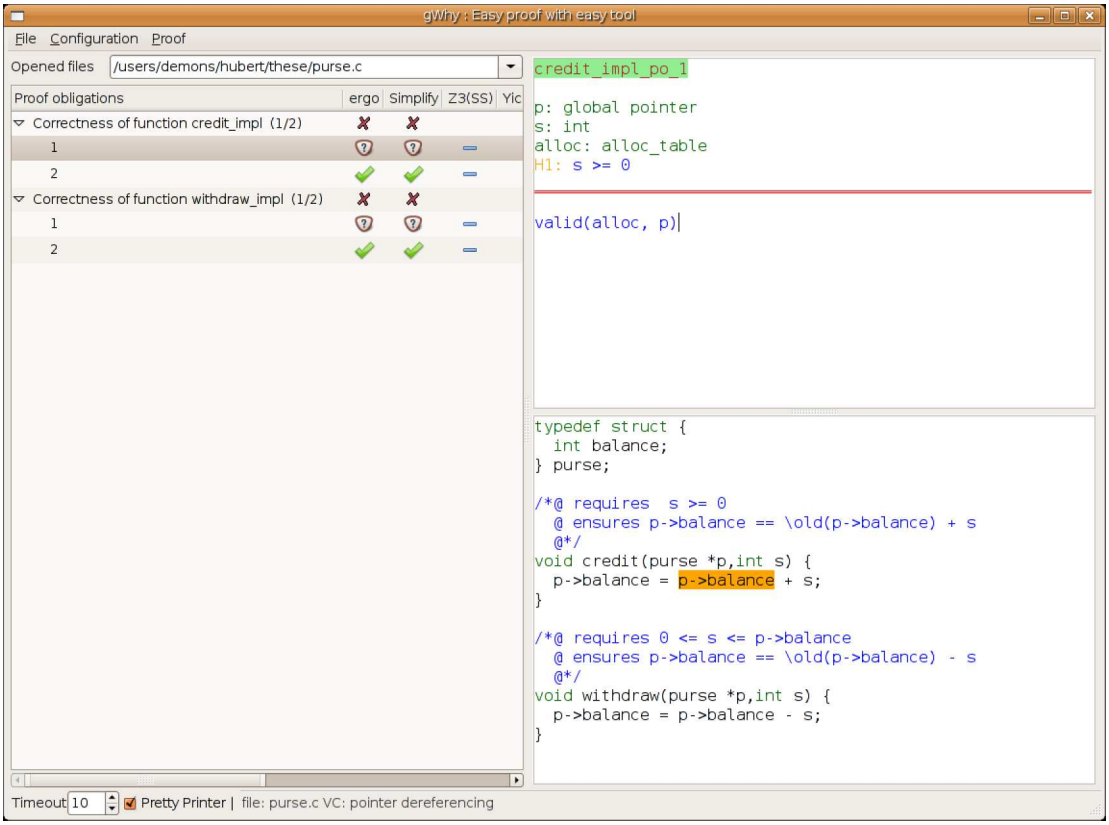


FIG. 2.1 – Visualisation des conditions de vérification pour le programme purse

fonction et le mot-clé `ensures` introduit un prédicat appelé la post-condition de la fonction. Donc sur cet exemple `credit` ne peut être appelé que si `s >= 0` et après son appel on aura : `p->balance == \old(p->balance) + s` où `\old(p->balance)` désigne la valeur de `p->balance` avant l'appel de la fonction. Une fois passé dans Caduceus, ce code génère quatre conditions de vérification, montrées sur la figure 2.1. Sur la partie gauche de la figure ont été exécutés les prouveurs automatiques Ergo et Simplify. La deuxième et la quatrième conditions de vérification ont été prouvées mais pas la première ni la troisième. Plus précisément la première condition de vérification demande de vérifier, dans l'expression surlignée dans la vue en bas à droite de la figure 2.1 `p->balance`, que le pointeur `p` est valide (ce qui est le but affiché en haut à droite). Effectivement rien n'empêche un programme d'appeler la fonction `credit` sur un pointeur nul ou plus généralement qui pointe en dehors d'une zone mémoire allouée. Dans le langage de spécification de Caduceus on peut renforcer la pré-condition pour exiger que le pointeur soit valide. Ce qu'on écrit :

```
/*@ requires  \valid(p) && s >= 0
  @ ensures p->balance == \old(p->balance) + s
```

```
  @*/
void credit(purse *p,int s) {
```

où le predicat prédéfini `\valid(p)` signifie précisément que le pointeur `p` pointe sur une zone mémoire correctement allouée.

On effectue le même ajout pour la fonction `withdraw` et alors toutes les conditions de vérification sont établies par les prouveurs automatiques. Ce qui signifie que les fonctions de ce code respectent leurs contrats et ne contiennent pas de menace.

2.1.2.2 Aliasing de pointeur

Pour illustrer la gestion des alias de pointeurs par Caduceus, rajoutons la fonction suivante à notre code :

```
/*@ requires \valid(p1) && \valid(p2)
  @ ensures \result == 0
  @*/
int test(purse *p1, purse *p2) {
    p1->balance = 0;
    credit(p2,100);
    return p1->balance;
}
```

Cette fonction rajoute deux conditions de vérification, la première est la post-condition de cette nouvelle fonction et la deuxième est la pré-condition de `credit` lors de son appel. En effet, avant d'appeler une fonction, il est nécessaire de vérifier que sa précondition est valide pour les valeurs effectives de ses paramètres.

La condition de vérification venant de la post-condition de la fonction `test` est :

```
p1       :pointer.
p2       :pointer.
alloc    :alloc_table.
balance  :int memory.
H1       :valid(alloc, p1) and valid(alloc, p2))
balance_0:int memory.
H2       :balance_0 = upd(balance, p1, 0)
balance_1:int memory.
H3       :acc(balance_1,p2) = (acc(balance_0, p2) + 100)
result   :int.
H4       :(result = acc(balance_1, p1)
------------------------------------------------------------
(result = 0)
```

Ici on retrouve la pré-condition $H1$ de la fonction `test` et sa post-condition dans le but. De plus on voit apparaître comme hypothèse $H3$ la post-condition de `credit`.

Cette condition de vérification n'est pas prouvable, car il est nécessaire de savoir que p_1 et p_2 sont différents. En effet, si p_1 est alias de p_2, alors le résultat de la fonction `test` sera 100 et non 0 comme l'on cherche à le montrer.

Si on sait que `test` ne sera jamais appelé avec le même pointeur on peut rajouter en pré-condition cette information comme ceci :

```
/*@ requires \valid(p1) && \valid(p2) && p1 != p2
  @ ensures \result == 0
  @*/
int test(purse *p1, purse *p2) {
    p1->balance = 0;
    credit(p2,100);
    return p1->balance;
}
```

Ainsi la condition de vérification non prouvée contient en plus à la ligne 6 l'hypothèse `(p1 <> p2)`. Maintenant le code est correct mais l'outil n'arrive pas à le prouver. Ceci est dû à la modularité de l'approche utilisée. En effet, lors du traitement de la fonction `test` Caduceus ne regarde plus que la spécification de la fonction `credit` et pas son code. De ce fait, il lui est impossible de savoir que $p1 \rightarrow balance$ n'est pas modifié par la fonction `credit` car rien ne le dit dans la post-condition.

Pour remédier à ce problème il existe un autre clause pour spécifier les fonctions : la clause `assigns`. Elle est utilisée pour lister les effets de bords de la fonction, comme illustré ci-dessous :

```
/*@ requires  \valid(p) && s >= 0
  @ assigns p->balance
  @ ensures p->balance == \old(p->balance) + s
  @*/
void credit(purse *p,int s) {
  p->balance = p->balance + s;
}
```

En effet, la seule modification effectuée par la fonction `credit` est sur le champ `balance` du pointeur p passé en paramètre. Avec cette dernière modification notre condition de vérification est vérifiée automatiquement établissant ainsi que le code donné est sans menace et qu'il vérifie bien la spécification donnée.

Il faut noter que l'ajout de la clause `assigns` introduit une nouvelle condition de vérification pour la fonction `credit` : l'approche Caduceus assure aussi la correction des programmes vis à vis de ces clauses. Cette nouvelle condition de vérification est aussi prouvée automatiquement.

2.2 Langage de spécification de Caduceus

Dans cette section nous introduisons le fragment du langage de spécification de Caduceus dont nous avons besoin dans cette thèse. Le langage complet est décrit dans le ma-

nuel [58]. Essentiellement, nous utilisons la logique du premier ordre classique qui considère une syntaxe pour les *propositions* où les atomes sont des *termes*. Les termes sont les expressions sans effets de bord du C auxquelles on ajoute des constructions supplémentaires. Propositions et termes sont décrits par des grammaires données par la figure 2.2. La notation : $\langle term \rangle^{+}_{,}$ est une abréviation pour $\langle term \rangle$`, ...,` $\langle term \rangle$ où $\langle term \rangle$ apparaît au moins une fois.

2.2.1 Expressions logiques

Les connecteurs logiques sont notés && pour la conjonction, `||` pour la disjonction, `=>` pour l'implication et les quantifications sont notées : `\forall` τ `x; P` et `\exists` τ `x; P`. Le langage est étendu par :

- Des prédicats, fonctions et constantes prédéfinis :
  - `\valid(p)` : ce prédicat signifie que le pointer `p` est correctement alloué dans la mémoire. C'est-à-dire que l'on peut lire et écrire à l'adresse de ce pointeur.
  - `\valid_range(p, a, b)` : ce prédicat signifie que le pointer `p` est valide entre les indices `a` et `b` c'est-à-dire que l'on peut lire et écrire à toutes les adresses des pointeurs `p+i` avec `a` $\leq$ `i` $\leq$ `b`.
  - `\base_addr(p)` : cette formule représente l'adresse de base du pointeur `p` et sera détaillée dans la section 2.3.3.
  - `\null` : ce mot clef représente le pointeur nul.
- Enfin nous utiliserons les constructions spéciales suivantes :
  - `\result` : cette notation représente la valeur retournée par une fonction. De ce fait elle ne peut être utilisée que lors d'une post-condition de fonction et si le type de résultat n'est pas `void`.
  - `\old(x)` : cette notation représente la valeur de l'expression `x` au point d'entrée de la fonction.
  - `\at(x,L)` : cette notation représente la valeur de l'expression `x` au label `L`.

2.2.2 Clauses de spécification

Ce sont les clauses qui découpent les annotations logiques en plusieurs parties.

- Clauses de spécification des fonctions :
  - `requires` : cette clause indique que le prédicat qui le suit est la pré-condition de la fonction dont dépend l'annotation logique. Sa syntaxe est :
    `requires` $\langle proposition \rangle$
  - `assigns` : cette clause indique que la liste de variables qui le suit est la liste des effets de bord de la fonction. Sa syntaxe est : `assigns` ($\langle term \rangle^{+}_{,}$ | `\nothing`).
  - `ensures` : cette clause indique que le prédicat qui le suit est la post-condition de la fonction. Sa syntaxe est : `ensures` $\langle proposition \rangle$
- Clauses de spécification des boucles :
  - `invariant` : est suivit d'une proposition qui représente l'invariant inductif de cette boucle. Sa syntaxe est :
    `invariant` $\langle proposition \rangle$

⟨*term*⟩ ::= ⟨*constant*⟩
| ⟨*term*⟩ ⟨*arith_op*⟩ ⟨*term*⟩
| `-` ⟨*term*⟩ | `+` ⟨*term*⟩
| `*` ⟨*term*⟩
| ⟨*term*⟩ `->` ⟨*identifier*⟩
| ⟨*term*⟩ `.` ⟨*identifier*⟩
| ⟨*identifier*⟩ `(` ⟨*term*⟩$^{+}_{,}$ `)`
| ⟨*term*⟩ `[` ⟨*term*⟩ `]`
| `(` ⟨*term*⟩ `)`
| `(` ⟨*logic_type*⟩ `)` ⟨*term*⟩
| `\old (` ⟨*term*⟩ `)` | `\at (` ⟨*term*⟩ `,` ⟨*identifier*⟩ `)`
| `\result` | `\null`
| `\base_addr (` ⟨*term*⟩ `)`

⟨*constant*⟩ ::= ⟨*integer-constant*⟩ | ⟨*floating-point-constant*⟩

⟨*arith_op*⟩ ::= `+` | `-` | `*` | `/` | `%`

⟨*logic_parameter*⟩ ::= ⟨*logic_type*⟩ ⟨*identifier*⟩
⟨*proposition*⟩ ::= `\true`
| `\false`
| ⟨*identifier*⟩
| ⟨*identifier*⟩ `(` ⟨*term*⟩$^{+}_{,}$ `)`
| ⟨*term*⟩ ⟨*relation*⟩ ⟨*term*⟩ [⟨*relation*⟩ ⟨*term*⟩]
| ⟨*proposition*⟩ `=>` ⟨*proposition*⟩
| ⟨*proposition*⟩ `<=>` ⟨*proposition*⟩
| ⟨*proposition*⟩ `||` ⟨*proposition*⟩
| ⟨*proposition*⟩ `&&` ⟨*proposition*⟩
| `!` ⟨*proposition*⟩
| `if` ⟨*term*⟩ `then` ⟨*proposition*⟩ `else` ⟨*proposition*⟩
| `\forall` ⟨*logic_parameter*⟩$^{+}_{,}$ `;` ⟨*proposition*⟩
| `\exists` ⟨*logic_parameter*⟩$^{+}_{,}$ `;` ⟨*proposition*⟩
| `(` ⟨*proposition*⟩ `)`
| `\old (` ⟨*proposition*⟩ `)`
| `\at (` ⟨*proposition*⟩ `,` ⟨*identifier*⟩ `)`
| `\valid (` ⟨*term*⟩ `)`
| `\valid_range (` ⟨*term*⟩ `,` ⟨*term*⟩ `,` ⟨*term*⟩ `)`
| ⟨*identifier*⟩ `: :` ⟨*proposition*⟩

⟨*relation*⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=`

FIG. 2.2 – Gammaire des termes et propositions

– `variant` est suivi d'un terme et éventuellement d'une relation d'ordre. Sa syntaxe est :
`variant` ⟨*term*⟩ [`for` ⟨*identifier*⟩]
Cette clause spécifie que ce terme décroît pour chaque tour de boucle suivant la relation d'ordre donnée. Si l'ordre n'est pas présent Caduceus utilisera l'ordre inférieur strictement sur les entiers naturels. Et si l'indicateur `variant` n'est pas présent sur une boucle, la preuve de terminaison de cette boucle ne sera pas effectuée.
– `loop_assigns` indique que la liste de variables qui le suit est la liste des effets de bord de la boucle. Sa syntaxe est :
`loop_assigns` (⟨*term*⟩$^{+}_{,}$ | `\nothing`)

2.2.3 Les déclarations logiques

Ce sont des déclarations qui permettent de définir de nouveaux types logiques et des prédicats, des fonctions ou des constantes dans la logique uniquement.

– `type` : ce déclarateur permet de déclarer un type abstrait logique qui restera abstrait pour les prouveurs de sortie, ou éventuellement dans le cas des prouveurs interactifs sera défini concrètement dans la syntaxe du prouveur correspondant. Sa syntaxe est :
`type` ⟨*identifier*⟩
– `logic` : ce déclarateur permet de déclarer des fonctions logiques qui sont définies concrètement ou bien abstraitement. Sa syntaxe est :
`logic` ⟨*logic_type*⟩ ⟨*identifier*⟩ `(` (⟨*logic_type*⟩ ⟨*identifier*⟩)$^{\star}_{,}$ `)`
[`{` ⟨*term*⟩ `}` | `reads` ⟨*term*⟩$^{+}_{,}$]
– `predicat` : ce déclarateur permet de déclarer des prédicats qui sont définies concrètement ou bien abstraitement. Sa syntaxe est :
`predicat` ⟨*identifier*⟩ `(` (⟨*logic_type*⟩ ⟨*identifier*⟩)$^{\star}_{,}$ `)`
[`{` ⟨*proposition*⟩ `}` | `reads` ⟨*term*⟩$^{+}_{,}$]

Dans le cas d'une définition abstraite, la présence de la clause `reads` indique que le prédicat ou la fonction correspondante dépend d'un état de la mémoire. La clause liste les locations lues. Dans les prouveurs de sorties, ce prédicat ou cette fonction possèdera des arguments supplémentaires lié au modèle de la mémoire qui sera présenté dans la section 2.3.3. Ce point technique sera illustré dans le chapitre 3.

– `axiom` : ce déclarateur permet de déclarer des axiomes. C'est-à-dire que la formule logique qui suit ce mot clef sera admise comme toujours vraie. Sa syntaxe est :
`axiom` ⟨*identifier*⟩ `:` ⟨*proposition*⟩

Les axiomes permettent une alternative à la définition des prédicats et des fonctions abstraites dans un prouveur externe : ils permettent de construire une *axiomatisation*.

2.3 Fondements du fonctionnement interne

L'outil Caduceus est un des outils de la plate-forme Why dédiée à la vérification par preuve des programmes C et Java. Dans cette plate-forme un unique *générateur de condi-*

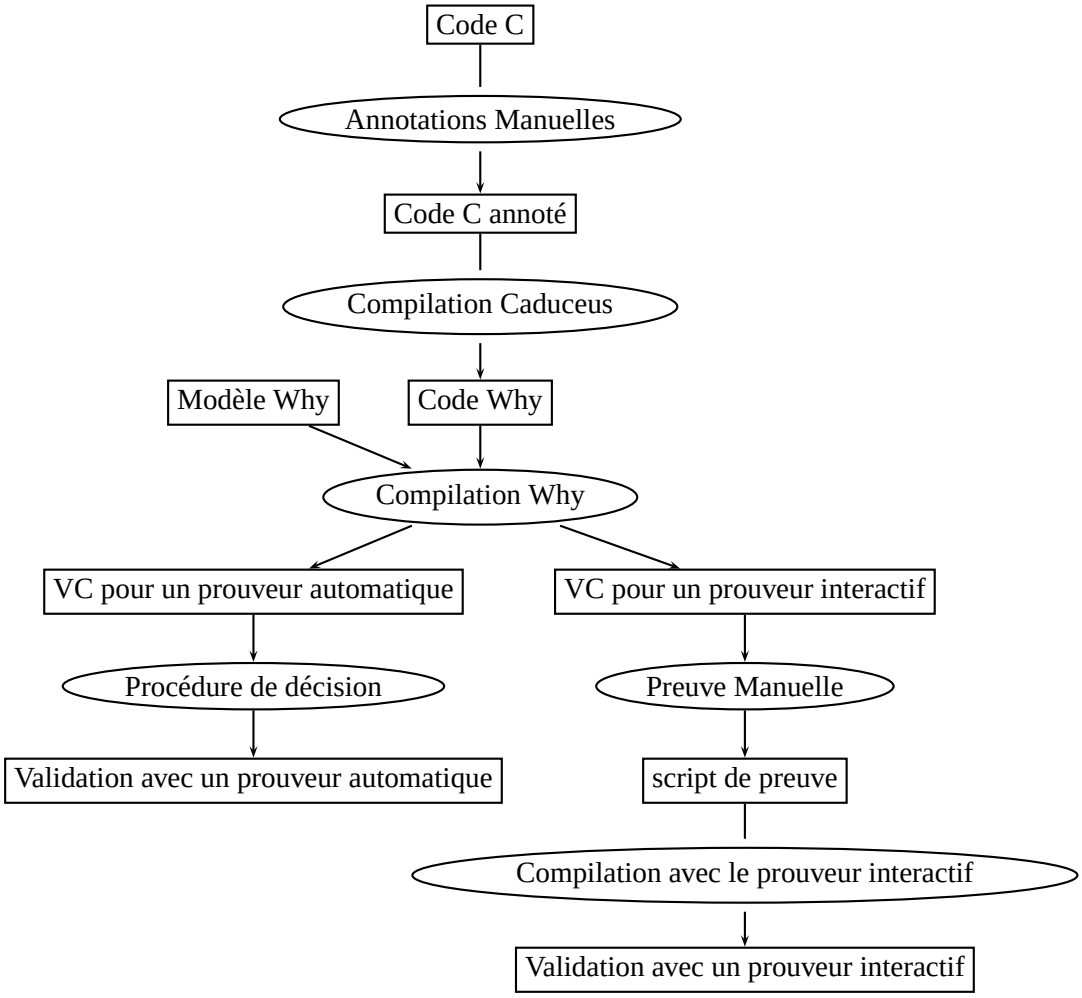


FIG. 2.3 – Schématisation du fonctionnement de Caduceus

tions de vérification est utilisé. Il s'appelle également Why. L'interaction entre l'outil Caduceus et Why est schématisé sur la figure 2.3.

Dans la section 2.3.1 nous décrivons l'outil Why. Puis dans la section 2.3.2 nous présenterons l'interaction entre Caduceus et Why.

2.3.1 L'outil Why

L'outil Why est un générateur de condition de vérification qui opère sur des programmes annotés écris dans une syntaxe dédiée à la vérification par preuve (un langage que nous appelerons également le langage Why).

Le langage Why est de style ML. Il ne fait pas de distinction entre les expressions et les instructions. Les seuls effets de bord possibles sont des affectations sur des variables appelées *références*.

La génération des conditions de vérification procède par un calcul de plus faible précondition relativement classique. Néanmoins une originalité importante est l'intégration d'un mécanisme d'exceptions. Ainsi, une expression de ce langage sera annotée à la fois par une post-condition normale et un ensemble de post-conditions pour les cas de terminaison exceptionnelle.

Dans cette sous-section nous décrivons le langage Why ainsi que le fonctionnement de l'outil de vérification Why [56].

2.3.1.1 Grammaire

La grammaire abstraite du langage Why est présentée sur la figure 2.4. La syntaxe concrète de ce langage est présentée dans le manuel de Why [55]. En particulier la boucle `while` classique ou bien la séquence d'expression peuvent se réduire dans cette syntaxe abstraite.

2.3.1.2 Le calcul de plus faible pré-condition

Nous notons $wp(e, q; r)$ la plus faible pré-condition pour une expression e de programme et une post-condition $q; r$ où q est la propriété à prouver si le programme se termine normalement et r est un ensemble de (E_i, q_i) où chaque E_i est une exception possible de e et chaque q_i est la post-condition associée à cette exception. Nous notons $r = E1 \Rightarrow q1; ...; En \Rightarrow qn$. Exprimer l'exactitude d'un programme e revient simplement à une question de calcul de la formule $wp(e, q; r)$ puis à vérifier que la pré-condition implique cette formule.

Les règles pour les constructions de base sont :

– cas des expressions pures (sans effet de bord) :

$$wp(t, q; r) = q[result \rightarrow t]$$

avec $q[result \rightarrow t]$ la substitution dans q de $result$ par t

– cas du let :

$$wp(let\ x\ =\ e_1\ in\ e_2, q; r)\ =\ wp(e_1, wp(e_2, q; r)[x \rightarrow result]; r)$$

t	::=	$constant$	constant
	|	$t\ arith\_op\ t$	operateur binaire
	|	`-` t	opposé
	|	id`@`id	variable avec label
	|	id`(` $t^+_{,}$ `)`	application
	|	`if` t `then` t `else` t	conditionnelle
p	::=	`true`	true
	|	`false`	false
	|	id`(`$t^+_{,}$`)`	application
	|	$t\ relation\ t$	comparaisons
	|	p `->` p	implication
	|	p `<->` p	equivalence
	|	p `or` p	ou
	|	p `and` p	et
	|	`not` p	négation
	|	`if` t `then` p `else` p	branchement conditionnel
	|	`forall` $id^+_{,}$ `:` $primitive\_type$ `.` p	pour tout
	|	`exists` $id^+_{,}$ `:` $primitive\_type$ `.` p	existe
ep	::=	$id \mid$ `!`$id \mid f$`(`$ep, \ldots, ep$`)`	expression simple
e	::=	ep	expression simple
	|	let id = e in e	déclaration de variable
	|	let id = ref e in e	déclaration de variable référencé
	|	if e then e else e	branchement conditionnel
	|	loop $\{$invariant p variant t for $R\}$ e	boucle infinie
	|	L`:`e	label
	|	raise $(E\ e) : \theta$	levé d'exception
	|	try e with $E\ id \rightarrow e$ end	capture d'exception
	|	assert $\{p\};\ e$	assertion
	|	$e\ \{q\}$	«boite blanche»
	|	$e\ \{\{q\}\}$	«boite noire»
	|	fun $(id : \theta) \rightarrow \{p\}\ e$	déclaration de fonction
	|	rec $f\ (id : \theta) \ldots (id : \theta) : \theta\ \{$variant $t\} = \{p\}\ e$	déclaration de fonction récursive
	|	$(e\ e)$	application

FIG. 2.4 – Syntaxe Abstraite des expressions

– cas du let avec une référence :

$$wp(let\ x\ =\ ref\ e_1\ in\ e_2, q; r)\ =\ wp(e_1, wp(e_2, q; r)[x \rightarrow result]; r)$$

La cas du let avec référence se traite donc de la même façon que le let sans référence.

– cas du if :

$$wp(if\ e_1\ then\ e_2\ else\ e_3, q, r)\ =\ wp(e_1, if\ result\ then\ wp(e_2, q; r)\ else\ wp(e_3, q; r); r)$$

– cas du label :

$$wp(L : e, q; r)\ =\ wp(e, q; r)[x@L \rightarrow x]$$

– cas des exceptions :

$$wp(raise(E\ e), q; r)\ =\ wp(e, r(E); r)$$

$$wp(try\ e_1\ with\ E\ x \rightarrow e_2\ end, q; r)\ =\ wp(e_1, q; E \Leftarrow wp(e_2, q; r)[x \rightarrow result]; r)$$

– cas des assertions :

$$wp(assert\{p\}; e, q; r)\ =\ p \wedge wp(e, q; r)$$

– cas des post-conditions (boite blanche) :

$$wp(e\{q'; r'\}, q; r)\ =\ wp(e, q' \wedge q; r' \wedge r)$$

– cas des post-condition (boite noire) :

$$wp(e\{\{q'; r'\}\}, q; r)\ =\ wp(e, q'; r') \wedge \forall\, \omega.\, \forall\, result.\, (q' \Rightarrow q) \wedge (r' \Rightarrow r)$$

La dernière implication $r' \Rightarrow r$ est en fait un abus de langage pour la conjonction de toutes les implications pour chaque partie de la post-condition.

– cas d'une boucle infinie :

$$wp(\mathsf{loop}\ \{\mathsf{invariant}\ p\ \mathsf{variant}\ t\ \mathsf{for}\ R\}\ e, q; r)\ =\ p \wedge \forall\, \omega.\, p \Rightarrow wp(L : e, p \wedge R(t, t@L); r)$$

où ω représente l'ensemble de références pouvant être modifiées par le corps de la boucle et $t@L$ est un abus de notation pour indiquer que l'on attache le label L à toutes les variables de t. Ici la plus faible pré-condition exprime que l'invariant doit être vérifié initialement et que pour chaque tour de la boucle, soit p est préservé inductivement par e et e fait décroître la valeur de t (pour assurer la terminaison), soit une exception est levée (devant établir r). La formule inductive est quantifiée universellement sur les références modifiées par le corps de boucle. La règles pour la boucle `while` se déduit des règles sur la boucle infinie, le `if` et le `raise`.

– cas pour les fonctions et appels de fonctions.
La plus faible pré-condition pour les constructeurs de fonctions *fun* et *rec* expriment seulement l'exactitude du corps de la fonction :

$$wp(\mathsf{fun}(x : \theta) \rightarrow \{p\}\ e, q; r)\ =\ q \wedge \forall\, x. \forall\, \rho.\, p\ \Rightarrow wp(e, True)$$

$$wp(\textsf{rec}\ f(x_1:\theta_1)...(x_n:\theta_n):\theta\{variant\ t\}=\{p\}\ e,q;r)$$

$$=\ q\wedge\forall\ x_1...\forall\ x_n.\forall\ \rho.\ p\ \Rightarrow\ wp(L:e,True)$$

où ρ représente l'ensemble des références accessibles par le corps de la boucle. Dans le cas de fonctions récursives, $wp(L\ :\ e,True)$ doit être compilée avec un environnement où f est supposé avoir le type $(x_1,\theta_1)\ \rightarrow\ ...\ \rightarrow\ (x_n\ :\ \theta_n)\ \rightarrow\ \{p\ t\ <\ t@L\}\ \theta\ \epsilon\ \{q\}$ c'est-à-dire où la diminution du variant t a été ajoutée à la pré-condition de f.

– cas de l'appel de fonctions $(e_1\ e_2)$.
Il peut être simplifié au cas d'une application x_1x_2 d'une variable à l'autre, en utilisant la transformation suivante si nécessaire :

$$e_1\ e_2\ \equiv\ let\ x_1\ =\ e_1\ in\ let\ x_2\ =\ e_2\ in\ x_1\ x_2$$

Alors, supposons que x_1 a le type $(x:\omega)\ \rightarrow\ \{p'\}\ \theta'\ \epsilon\ \{q'\}$, nous définissons :

$$wp(x_1\ x_2,q)\ =\ p'[x\leftarrow x_2]\wedge\forall\ \omega.\ \forall\ result.\ (q'[x\leftarrow x_2]\Rightarrow q)[t@\rightarrow t]$$

cela signifie que la pré-condition de la fonction doit être vérifiée et sa post-condition doit impliquer la propriété prévue q quelque soit la modification des variables et du résultat. Noter que q et q' peuvent contenir des exceptions. Cette implication est donc encore un abus de langage pour la conjonction de toutes les implications pour chaque partie de la post-condition.

Remarque : Le cas d'une affectation $x\ :=\ e$ n'est pas dans la grammaire car $:=$ est seulement une fonction prédéfinie, dont le type Why est : $(x\ :\ \alpha ref)\ (y\ :\ \alpha)\ \rightarrow\ \{true\}\ unit\ writes\ x\ \{x\ =\ y\}$ on peut néanmoins expliciter le calcul de pré-condition pour ce cas :

$$wp(x:=e,q;r)\ =\ wp(e,x=result\Rightarrow q;r)$$

Dans le cas où l'expression affectée n'a pas d'effet de bord, la règle est simplifiée en :

$$wp(x:=t,q)\ =\ (x=t\Rightarrow q)$$

ce qui est équivalent à la règle habituelle de la logique de Hoare :

$$wp(x:=t,q)\ =\ q[x\leftarrow t]$$

2.3.1.3 Correction du calcul de plus faible pré-condition

Les programmes Why sont typés par un typage à la ML. Une particularité existe pour les références : deux arguments de fonction de type référence n'ont jamais le droit d'être *aliasés* lors d'un appel de cette fonction.

Exemple 2.3.1. *Considérons le programme Why suivant :*

```
let incr2(x: int ref)(y:int ref) =
x:=!x+1;
y:=!y+1
x=x@+1 and y=y@+1

let f(z: int ref) = incr2 z z z=z@+2
```

La fonction `incr2` est parfaitement accéptée (et prouvée automatiquement) par Why. Par contre `f` est rejeté au typage car crée un alias entre x et y.

Le calcul de plus faible pré-condition de Why est correct dans le sens où si un programme est bien typé alors la validité des conditions de vérification entraine la validité du code Why vis à vis de ses spécifications.

2.3.2 La traduction du langage C vers le langage Why

L'outil Caduceus peut être vu comme un compilateur du langage C vers le langage Why. La méthode utilisée pour cette compilation est le plongement superficiel (shallow embedding), ce qui signifie que pour chaque fonction du programme en C on obtiendra une fonction dans le programme en Why. Par opposition au plongement profond (deep embedding en anglais), qui consisterait à définir un type de donnée pour les programmes dans le langage cible.

Revenons sur notre premier exemple :

```
typedef struct {
  int balance;
} purse;

/*@ requires  \valid(p) && s >= 0
  @ assigns p->balance
  @ ensures p->balance == \old(p->balance) + s
  @*/
void credit(purse *p,int s) {
  p->balance = p->balance + s;
}

/*@ requires \valid(p) && 0 <= s <= p->balance
  @ assigns p->balance
  @ ensures p->balance == \old(p->balance) - s
  @*/
void withdraw(purse *p,int s) {
  p->balance = p->balance - s;
}
```

Ce code une fois traduit en Why donne le code de la figure 2.5. On remarque ici plusieurs choses :

```
(* heap variables *)
parameter balance : int memory ref

(* functions specifications *)
parameter credit_parameter :
 p:pointer -> s:int ->
   {(valid(alloc, p) and ge_int(s, 0))}
   unit reads alloc,balance writes balance
   { eq_int(acc(balance, p), add_int(acc(balance@, p), s)))
     and not_assigns(alloc@, balance@, balance,
         pset_singleton(p)) }

parameter withdraw_parameter :
 p:pointer -> s:int ->
   {(valid(alloc, p)
     and (le_int(0, s) and le_int(s, acc(balance, p))))}
   unit reads alloc,balance writes balance
   { eq_int(acc(balance, p), sub_int(acc(balance@, p), s))
     and not_assigns(alloc@, balance@, balance,
         pset_singleton(p)) }

let credit_impl =
 fun (p :pointer) (s : int) ->
  { (valid(alloc, p) and ge_int(s, 0)) }
  init:
  (let caduceus_2 = p in
  (((upd_ balance) caduceus_2)
    (let caduceus_1 = ((acc_ balance) p) in
                       ((add_int caduceus_1) s))))
  { eq_int(acc(balance, p), add_int(acc(balance@, p), s))
    and not_assigns(alloc@, balance@, balance,
        pset_singleton(p)) }

let withdraw_impl =
 fun (p :pointer) (s : int) ->
  { (valid(alloc, p)
     and (le_int(0, s) and le_int(s, acc(balance, p)))) }
  init:
  (let caduceus_2 = p in
  (((upd_ balance) caduceus_2)
    (let caduceus_1 = ((acc_ balance) p) in
                       ((sub_int caduceus_1) s))))
  { eq_int(acc(balance, p), sub_int(acc(balance@, p), s))
     and not_assigns(alloc@, balance@, balance,
         pset_singleton(p)) }
```

FIG. 2.5 – Code Why engendré pour le programme `purse`

1. pour chaque fonction en C il y a une fonction et une interface de cette fonction en Why. Par exemple pour la fonction `credit` il y a l'interface `credit_parameter` et la fonction `credit_impl`. Ce qui permet de traiter les fonctions récursives (mais sans prouver leur terminaison).
2. la variable globale `balance` apparaît, ceci sera expliqué dans la section 2.3.3
3. on voit apparaître des `reads` et `writes` dans les interfaces qui représentent les effets de la fonction qui sont soit donnés par l'utilisateur à l'aide du `assigns` (voir 2.2), soit calculés par Caduceus comme une surestimation des effets. C'est-à-dire qu'il considère toutes les variables présentes dans la fonction comme étant présentes dans les effets de bord.

2.3.3 Le modèle de la mémoire

La mémoire de C est représentée par un ensemble fini de variables Why. Dans Caduceus a été adopté l'approche de Burstall [36], mis en avant par Bornat [33] puis Mehta and Nipkow [85]. L'idée de base est d'avoir une variable Why pour chaque champ de structure C. En effet, deux champs de structure ne peuvent être aliasés, s'il n'y a jamais de *cast* de pointeur. Le principal avantage de cette approche est que lorsqu'on modifie un champ de structure, seule la variable Why correspondante est modifiée, les autres champs de structures restant de faits inchangés. Ce modèle est également étendu aux tableaux et plus généralement à l'arithmétique de pointeur [59].

L'ensemble des variables représentant la mémoire du C peut-être représenté sous la forme de modélisation telle qu'exposée sur la figure 2.6. Toutes ces variables contiennent des tableaux indexés par les adresses de base a_1, a_2, ... appartenant au type de données abstrait `addr`. La variable `alloc` du côté gauche de la figure est l'endroit où sont stockées les allocations qui indiquent pour chaque adresse si la case du tableau associée à cet indice existe et, quand c'est le cas, la taille du bloc alloué. Toutes les autres variables possèdent un tableau à une dimension de la taille définie par la valeur précisée dans la variable `alloc`. Les variables du type `pointer` représentent les pointeurs du C. Une valeur de type `pointer` est soit le pointeur `null` soit une paire composée d'une adresse de base et d'un entier représentant son offset. Par exemple, l'adresse de base a_2 référence un bloc qui selon `alloc` est de taille 5 et le pointeur $(a_2, 3)$ référence ainsi le quatrième élément de chaque tableau $f_1(a_2)$, $f_2(a_2)$,... La variable `intM` est utilisée pour les tableaux d'entiers c'est-à-dire les pointeurs sur des `int`, `intMP` pour les pointeurs sur des pointeurs sur des `int`, etc...De même la variable `pointerM` est utilisé pour les pointeurs sur structures, `pointerMP` pour les pointeurs sur des pointeurs sur des structures, etc...

Vu du côté de Why, chacun de ces tableaux est de type α `memory`, un type de donnés polymorphe et abstrait utilisé avec des fonctions sans effets de bords, pour accéder ou mettre à jour une cellule.

```
acc: α memory, pointer → α
shift : pointer, integer → pointer
upd: α memory, pointer, α → α memory
base_addr: pointer → addr
```

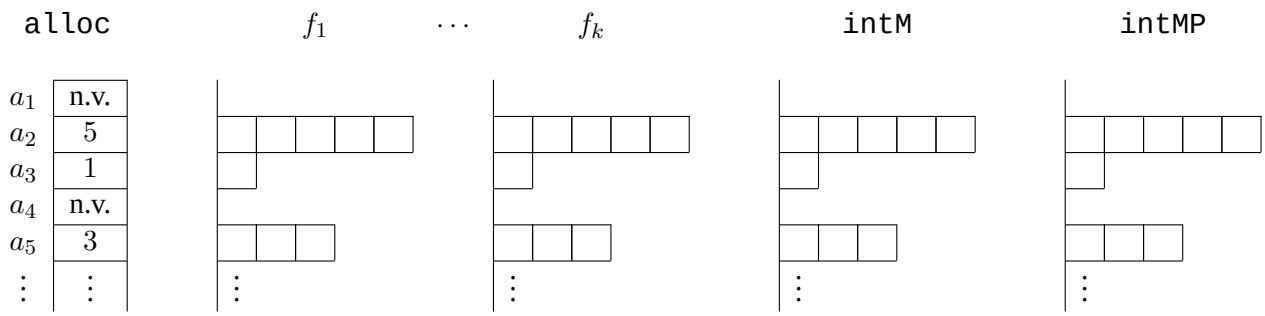


FIG. 2.6 – Modélisation de la mémoire du C

Selon la mémoire utilisée, α est soit `int`, soit `real`, soit `pointer`. On a également une variable Why pour chaque variable globale ou variable locale statique. Pour finir, les variables locales du C sont représentées en utilisant des variables locales Why puisqu'elles ne peuvent être aliasées. La fonction `base_addr` renvoie l'adresse de base d'un pointeur.

Remarque : on considère un modèle idéal pour les entiers et les réels. Cependant, il existe une option à Caduceus qui permet de traiter précisément les entier machine , en préservant l'information de la taille (en nombre de bits). Dans cette thèse, nous ne nous intérressons pas à ce deuxième cas.

Ensuite on axiomatise le modèle de la figure 2.6 :

$$acc(upd(m,i,v),i) = v \quad (2.1)$$
$$acc(upd(m,i,v),j) = acc(m,j) \text{ si } i \neq j \quad (2.2)$$
$$shift(p,0) = p \quad (2.3)$$
$$shift(shift(p,i),j) = shift(p,i+j) \quad (2.4)$$
$$base_addr(shift(p,i)) = base_addr(p) \quad (2.5)$$
$$offset(shift(p,i)) = offset(p) + i \quad (2.6)$$
$$(2.7)$$

Remarque : Ceci n'est qu'un extrait de l'axiomatique complète.

Ainsi le langage C est interprété dans notre langage intermédiaire Why par les règles de transformation :

$$[e \to f] \mapsto acc(f,[e]) \qquad [v = e] \mapsto v := [e]$$
$$[e_1 \oplus e_2] \mapsto shift([e_1],[e_2]) \qquad [e_1 \to f = e_2] \mapsto f := upd(f,[e_1],[e_2])$$

Cette présentation n'est qu'un survol rapide de la méthode. Nous donnerons dans le chapitre 5 des formules précises de transformation qui tiendront compte de notre analyse de séparation.

2.3.3.1 Exemple d'utilisation de ce modèle mémoire sur les tableaux :

```
// recherche l'indice de x dans t entre les indices 0 et n
/*@ requires n >= 0 && \valid_range(t,0,n-1)
```

```
  @ ensures (\result >= 0 => t[\result] == x) &&
  @     (\result == −1 => (\forall int i; 0 <= i < n => t[i] != x))
*/
int search(int t[], int n,  int x) {
   int i = 0;
   int *p = t;
   /*@ invariant 0 <= i <= n &&
     @    (\forall int j; 0 <= j < i => t[j] != x) &&
     @   p == t+i
     @*/
   while (i < n) {
     if (*p == x) return i;
     p++; i++;
  }
  return (−1);
}
```

Cette fonction cherche si un entier est présent dans le tableau t. Elle prend en argument le tableau t, sa taille n, ainsi que l'élément à chercher x et renvoi l'indice où est présent la première occurrence de x ou -1 s'il n'est pas présent.

La pré-condition de cette fonction indique que n doit être supérieur à 0 et que le tableau t doit être valide entre les indices 0 et $n-1$. Cette pré-condition indique bien que n est inférieur à la taille du tableau t.

La post-condition de cette fonction indique que si la valeur renvoyée `\result` est supérieure à 0, alors la valeur du tableau t à cet indice est x tandis que si la valeur renvoyée est -1 alors aucune case du tableau t n'a la valeur x.

Juste avant la boucle le commentaire logique avec le mot `invariant` permet de définir un invariant pour la boucle. Cet invariant indique qu'à chaque exécution de la boucle, i sera compris entre 0 et n, qu'aucune des cases d'indice inférieur à i n'a pour valeur x et que p est toujours égal à $t+i$.

Pour ce code Caduceus génère cinq conditions de vérification :

1. montrer que l'invariant de boucle est vrai avant le début de la boucle.
2. montrer que le pointeur p est valide à l'intérieur de la boucle (lors de $*p$).
3. vérifier la post-condition pour le premier `return` (celui à l'intérieur de la boucle)
4. montrer que l'invariant de boucle est préservé à chaque tour de boucle
5. vérifier la post-condition pour le deuxième `return` (celui après la boucle)

La première condition de vérification est :

```
t    :pointer
n    :int
x    :int
alloc:alloc_table
```

```
intM :int memory
H1   :(n >= 0) and valid_range(alloc, t, 0, (n - 1))
------------------------------------------------------
((0 <= 0) and (0 <= n) and
    (forall j:int. (((0 <= j) and (j < 0))->
         (acc(intM, shift(t, j)) <> x)))) and
     (t = shift(t, 0))
```

Cette condition de vérification contient deux parties : la première, rassemblant les 3 première lignes du but, demande la preuve que `*(t+j)` $\neq$ `x` pour tout `0<=j<0` ce qui ne représente aucun `j` donc cette partie est triviale. La seconde partie consiste à montrer que `t = shift(t, 0)`. Pour ce faire, il suffit d'utiliser l'axiome 2.3 pour réduire `shift(t,0)` à `t`.

La deuxième condition de vérification demande de prouver la validité du pointer p. Pour ce faire, on utilise l'invariant de boucle qui nous dit que $p == t + i$. Montrer la validité de p revient à montrer la validité de $t + i$. La pré-condition `\valid_range(t,0,n-1)` indique que $t + i$ est valide si `0<=i<n` ce que l'on a à l'intérieur de la boucle.

La troisième condition de vérification est :

```
t     :pointer
n     :int
x     :int
alloc :alloc_table
intM  :int memory
H1    :(n >= 0) and valid_range(alloc, t, 0, (n - 1))
i     :int
p     :pointer
H2    :(((0 <= i) and (i <= n)) and
         (forall j:int. (((0 <= j) and (j < i)) ->
         (acc(intM, shift(t, j)) <> x)))) and
         (p = shift(t, i))
H3    :i < n
H4    :valid(alloc, p)
result:int
H5    :result = acc(intM, p)
H6    :result = x
------------------------------------------------------
((i >= 0) -> (acc(intM, shift(t, i)) = x)) and
  ((i = -1) ->
   (forall i:int.
    (((0 <= i) and (i < n)) ->
      (acc(intM, shift(t, i)) <> x))))
```

Cette condition de vérification est la post-condition à prouver lors de la sortie de la fonction par le `return i`. En effet, on s'aperçoit à l'aide de $H5$ et $H6$ que `acc(intM, p) = x`, ce qui est la traduction de `*p = x`. La post-condition est coupée en deux parties : la première est la première ligne du but et dit que si `i >= 0` alors `*(t+i) = x`, la seconde, les quatre dernières lignes du but, disent que si `i = -1` alors $\forall$ `j, 0<=j<n -> *(t+j)` $\neq$ `x`. La seconde partie est prouvée, grâce à $H2$ on sait que `0<=i` donc `i` $\neq$ `-1`. Pour la première partie, il suffit de réécrire les hypothèses $H5$ et $H6$ en enlevant le `result` et de remplacer `p` par sa valeur ($H2$) pour obtenir `acc(intM,shift(t, i)) = x` ce qui est exactement ce que l'on doit prouver.

La quatrième condition de vérification est :

```
t      :pointer
n      :int
x      :int
alloc  :alloc_table
intM   :int memory
H1     :(n >= 0) and valid_range(alloc, t, 0, (n - 1))
i      :int
p      :pointer
H2     :(((0 <= i) and (i <= n)) and
         (forall j:int.(((0 <= j)and(j < i))->
         (acc(intM, shift(t,j)) <> x)))) and
         (p = shift(t, i)))
H3     :i < n
H4     :valid(alloc, p) ->
result :int
H5     :result = acc(intM, p)
H6     :result <> x
result0:pointer
H7     :result0 = shift(p, 1)
p0     :pointer
H8     :p0 = result0
i0     :int
H9     :i0 = (i + 1)
------------------------------------------------------
(((0 <= i0) and (i0 <= n)) and
    (forall j:int.(((0 <= j) and (j < i0)) ->
       (acc(intM, shift(t,j)) <> x))))
         and (p0 = shift(t, i0))
```

Cette condition de vérification est coupée en trois parties :

1. pour la première ligne du but, il faut montrer que `0<=i0<=n`,
2. pour les deux lignes suivantes dans le but, il faut montrer que `*(t+j)` $\neq$ `x` pour `0 <= j < i0`

3. pour la dernière ligne du but, il faut montrer que `p0 = shift(t, i0)`.

La première partie se montre grâce à $H9$: `i0 = i+1` et $H2$: `0 <= i < n` ce qui permet d'affirmer `0 <=i0 <=n`. La deuxième partie doit être coupée en 2 cas : soit `j = i`, dans ce cas grâce à $H2$ on a `p = shift(t, i)` et grâce à $H5$ et à $H6$ on a `acc(intM, p) <> x` donc on obtient aisément `(acc(intM, shift(t,i)) <> x)`, soit `j < i` dans ce cas on a exactement ce qu'il nous faut dans $H2$. Pour montrer la dernière partie on doit d'abord remplacer `p0` et `i0` par leur valeur à l'aide de $H7$ et de $H8$ pour `p0` et à l'aide de $H9$ pour `i0`, il nous reste à prouver ainsi `shift(p,1)=shift(t,i+1)`. Puis on remplace p par sa valeur vue dans $H2$ et il nous reste donc à prouver `shift(shift(t,i),1)=shift(t,i+1)`. Il ne reste qu'à utiliser l'axiome 2.4 pour obtenir `shift(t,i+1)=shift(t,i+1)` ce qui est trivial.

la cinquième et dernière condition de vérification :

```
t     :pointer
n     :int
x     :int
alloc :alloc_table
intM  :int memory
H1    :(n >= 0) and valid_range(alloc, t, 0, (n - 1))
i     :int
p     :pointer
H2    :(((0 <= i) and (i <= n)) and
          (forall j:int.(((0 <= j) and (j < i)) ->
          (acc(intM, shift(t, j)) <> x)))) and
          (p = shift(t, i))
H3    :i >= n
------------------------------------------------------
(((-1) >= 0) -> (acc(intM, shift(t, (-1))) = x)) and
    (((-1) < 0) ->
     (forall i:int.(((0 <= i) and (i < n)) ->
       (acc(intM, shift(t, i)) <> x))))
```

Cette condition de vérification est la post-condition à établir lors de la sortie de la fonction par le `return -1`. Donc comme la troisième condition de vérification elle se coupe en deux parties la première ligne du but d'un côté et les trois dernières lignes du but de l'autre. La première partie se prouve, car `((-1) >= 0)` est faux. La deuxième partie nécessite de prouver grâce à $H3$ et à $H2$ que `i = n` puis grâce à cela il suffit d'utiliser $H2$ pour obtenir notre preuve.

Toutes les cinq conditions de vérification sont déchargées automatiquement par Simplify.

2.3.3.2 Exemple d'utilisation du modèle mémoire sur les structures

```
typedef struct S {
```

```
  int x;
} S;

S s1,s2;


/*@ ensures  \result == x*/
int two_struct ( int y, int x){
  s1.x = x;
  s2.x = y;
  return s1.x;
}
```

Cette fonction écrit juste dans deux structures différentes et renvoie la valeur écrite dans la première structure. Cette fonction permet de mettre en avant l'avantage du modèle mémoire pour le problème d'aliasing sur une structure.

La post-condition de cette fonction est que le résultat de la fonction doit être la valeur écrite dans la première structure `s1`.

Sur cette fonction, Caduceus génère deux conditions de vérification :

1. la **validité de s1** en effet, dans notre modèle mémoire les structures sont représentées par des pointeurs. Il est donc nécessaire que ces pointeurs soient valides. Cette validité est donnée en invariant par Caduceus, car on peut toujours accéder à une structure globale donc on la considère comme toujours valide. En effet, pour chaque structure s Caduceus génère un invariant `\valid(s)`. Ce problème sera revu et détaillé au chapitre 4.
2. la post-condition de `two_struct` lors du `return s1.x` :

```
y     :int
x     :int
alloc :alloc_table
s1    :pointer
s2    :pointer
x_    :int memory
H1    :(base_addr(s2) <> base_addr(s1)) and
         (valid(alloc, s2) and valid(alloc, s1))
x_0   :int memory
H2    :x_0 = upd(x_, s1, x)
x_1   :int memory
H3    :x_1 = upd(x_0, s2, y)
H4    :valid(alloc, s1)
result:int
H5:   :result = acc(x_1, s1)
----------------------------------------------
result = x
```

Remarquons qu'à l'hypothèse $H1$ de cette condition de vérification, nous avons `(base_addr(s2) <> base_addr(s1))`. Comme expliqué plus haut, une valeur de type `pointer` est soit le pointeur `null` soit une paire composé d'une adresse et d'un `offset`. Or ici le prédicat `base_addr` fait référence à l'adresse du pointeur. Ainsi s1 et s2 ne pourrons jamais être aliasés.

Le but à vérifier est la post-condition `result = x`. Pour montrer cela, il est d'abord nécessaire de remplacer `result` par sa valeur vue dans $H5$ et ainsi d'obtenir `acc(x_1, s1) = x`. Puis, remplacer `x_1` par sa valeur dans l'hypothèse $H3$ et faire de même avec `x_0` grâce à l'hypothèse $H2$. Ainsi on obtient `acc(upd(upd(x_,s1,x),s2,y),s1) = x`. Ensuite, il suffit d'utiliser l'axiome 2.2 pour obtenir `acc(upd(x_,s1,x),s1) = x`, car on sait que `s1` $\neq$ `s2`. Et enfin, utiliser l'axiome 2.1 pour arriver à `x = x`. Cet travail est réalisé automatiquement par Simplify.

2.4 Bilan et conclusion de ce chapitre

Dans ce chapitre nous avons vu que l'outil Caduceus est une catégorie de compilateur vers l'outil Why qui génère, via un calcul de plus faible pré-condition, les conditions de vérification à prouver pour vérifier un programme.

Deux points de cette traduction du langage C vers le langage Why restent à approfondir :

1. La traduction du C vers le Why. En effet, cette traduction a été modifée au cours de cette thèse. Les détails sont explicités au chapitre 4.
2. Le modèle mémoire utilisé par Caduceus pour représenter la mémoire du C. Le modèle mémoire a aussi été modifié au cours de cette thèse. Les détails sont explicités au chapitre 5.

Ces deux modifications ont pour but de permettre le passage à l'échelle de l'outil Caduceus.

Chapitre 3

Validation de l'approche déductive sur un programme à pointeurs complexe

The Schorr-Waite algorithm is the first moutain that any formalism for pointer aliasing should climb.
— Richard Bornat ([33], page 121)

L'algorithme de Schorr-Waite [108] est un algorithme de parcours de graphe prévu pour les ramasses-miettes (garbage collector). Il effectue un parcours en profondeur d'une structure arbitraire de graphe (par conséquent une structure où l'aliasing est possible), sans utiliser de mémoire supplémentaire, mais en utilisant les pointeurs de la structure elle-même comme pile de backtrack. Les premières preuves (sans l'assistance d'un ordinateur) de l'exactitude de cet algorithme ont été données dans le même journal par Gries [66] et Topor [114] en 1979, et en 1982, Morris [90] présente une preuve semi-formelle en utilisant un mécanisme général pour traiter les structures de données inductives.

En 2000, Bornat [33] publia la première preuve formelle assistée par ordinateur de l'algorithme de Schorr-Waite en utilisant le système Jape [34]. En 2003, l'algorithme de Schorr-Waite a été employé encore comme étude de cas par Mehta et Nipkow [85], cette fois pour la vérification de programmes avec pointeurs dans la logique d'ordre supérieur du système Isabelle/HOL. La même année, Abrial [21] a effectué une autre vérification de cet algorithme, cette fois basée sur le raffinement, en utilisant le système B.

Pour démontrer que Caduceus est une approche puissante pour la vérification formelle de programmes source en C, selon la citation de Bornat, nous avons effectué une vérification de l'algorithme de Schorr-Waite et ce chapitre rend compte de cette expérience. Avec Caduceus, nous avons en effet formellement prouvé plus de propriétés de l'algorithme de Schorr-Waite que les études précédentes. La première propriété additionnelle que nous avons montrée est l'absence de *menaces* (défini dans la section 2.1.1). La deuxième propriété addition-

```
void schorr_waite(node root) {
  node t = root; node p = NULL;
  while (p != NULL || (t != NULL && ! t->m)) {
    if (t == NULL || t->m) {
      if (p->c) { // pop
        node q = t; t = p; p = p->r; t->r = q;
      }
      else { // swing
        node q = t; t = p->r; p->r = p->l; p->l = q;
p->c = 1;
      }
    }
    else { // push
      node q = p; p = t; t = t->l; p->l = q; p->m = 1;
      p->c = 0;
    }
  }
}
```

FIG. 3.1 – La version C de l'algorithme de Schorr-Waite

nelle, que nous montrons, est une propriété comportementale supplémentaire : tout ce qui est en dehors du graphe reste inchangé. La troisième propriété additionnelle est la terminaison : des arguments informels sont connus grâce à Topor [114] puis en 2003 Abrial [21] fit une preuve de terminaison intégrée dans le processus de raffinement de B, mais nous avons fournit la première preuve formelle de terminaison sur l'implémentation elle-même. Depuis une nouvelle preuve en Java a été faite avec le système Key [29].

Ce chapitre est essentiellement une version étendue de l'article [72]. Il est organisé comme ceci : dans la section 3.1, nous décrirons l'algorithme de Schorr-Waite et lui donnerons une spécification dans le langage de Caduceus. Puis nous décrirons la preuve formelle dans la section 3.2, en ajoutant les annotations dans le code source : l'invariant de boucle. Nous conclurons dans la section 3.3 avec une comparaison avec d'autres approches.

3.1 L'algorithme de Schorr-Waite

L'algorithme de Schorr-Waite effectue un parcours en profondeur d'un graphe orienté, il commence à un nœud particulier du graphe appelé racine. Sa caractéristique principale est d'utiliser directement les pointeurs du graphe pour implémenter le backtrack. C'est pourquoi sa sûreté est difficile à établir.

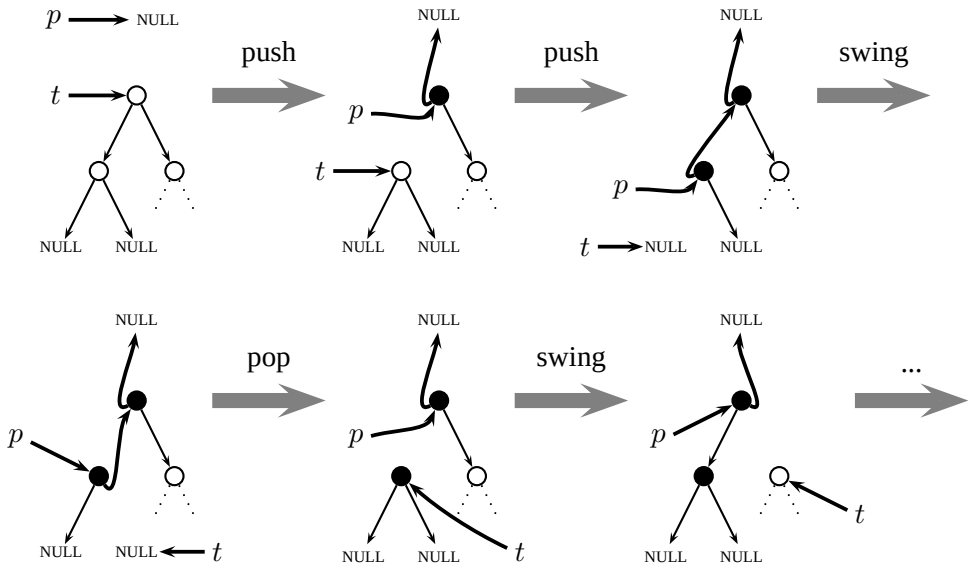


FIG. 3.2 – L'algorithme de Schorr-Waite : une exécution

3.1.1 Le code source C

Dans la version considérée ici, comme dans [33, 85], nous admettons que chaque nœud a au plus 2 fils[1]. Le cas de deux fils est la situation des ramasses-miettes d'un langage interprété comme LISP, où les seules allocations mémoire sont faite à l'aide de $cons$ de listes. En C, ces structures $cons$ peuvent être définies comme ceci :

```
typedef struct struct_node {
  unsigned int m :1, c :1; /* booleans */
  struct struct_node *l, *r;
} * node;
```

où l et r sont respectivement le pointeur vers le fils gauche ou le pointeur vers le fils droit (ils peuvent être nuls). Les champs m et c sont des entiers sur 1 bit, pour coder des booléens. Le champ m est une marque : initialement, tous les nœuds du graphe ne sont pas marqués et à la fin de l'algorithme, ils seront tous marqués. Le champ c est utilisé en interne, pour savoir lequel des fils est actuellement en cours de visite.

L'algorithme de marquage de graphe de Schorr-Waite, écrit directement en code C, est donné dans la figure 3.1. Le début d'une exécution simple est présenté dans la figure 3.2, avec chaque mouvement possible 'push', 'swing' et 'pop'. Les nœuds en noir représentent les nœuds marqués, inversement les nœuds en blanc représentent les nœuds non marqués. Le graphe présenté en exemple est un arbre pour simplifier, mais bien sûr l'algorithme fonctionne sur les graphes quelconques. Le nœud t est le prochain nœud à être exploré et

[1] l'extension vers un nombre arbitraire de fils est essentiellement la même chose, à condition que ce nombre soit limité par une valeur N et que le champ c soit suffisant pour représenter N valeurs différentes comme fait par [29]

```
/*@ requires
  @  \forall node x;
  @     x != \null && reachable(root,x) => \valid(x) &&! x->m
  @ ensures
  @  (\forall node x; \old(x->l) == x->l && \old(x->r) == x->r)
  @ &&
  @  (\forall node x; x != \null && reachable(root,x) => x->m)
  @ &&
  @  (\forall node x;!reachable(root,x) => x->m == \old(x->m))
  @*/
void schorr_waite(node root) {
  ...
}
```

FIG. 3.3 – La spécification de l'algorithme de Schorr-Waite en syntaxe Caduceus

le nœud p est le sommet de la pile de backtrack. Le mouvement 'push' marque un nouveau nœud et explore son fils gauche. Le mouvement 'swing' se produit lorsque le fils gauche a été exploré : la recherche continue sur le fils droit. Le mouvement 'pop' se produit lorsque le fils droit a été exploré : la recherche remonte dans le graphe.

3.1.2 La spécification formelle de l'algorithme

La première étape dans le processus de vérification formelle est de donner une spécification formelle à la fonction `schorr_waite`. La spécification informelle dit que tous les nœuds du graphe accessibles depuis la racine, doivent être marqués. De plus la structure du graphe doit être restaurée comme à l'origine.

Il apparaît immédiatement que pour la spécification formelle de cet algorithme, on aura besoin de parler d'accessibilité de certains nœuds depuis un autre nœud dans le graphe. En effet le raisonnement à propos d'accessibilité est un travail important de cette vérification, comme précisé dans les précédentes études de l'algorithme de Schorr-Waite.

Dans la méthodologie Caduceus, seulement la notion logique a besoin d'être déclarée, à l'aide de l'annotation `predicate` vu à la section 2.2.3, et elle sera supposé définie ou axiomatisée plus tard. Ceci est très similaire à la possibilité de déclarer des fonctions C bien avant de donner leur implémentation : de tels prédicats doivent être implémentés plus tard, soit en donnant des données additionnelles à Caduceus (à l'aide de l'annotation `axiom`), soit dans le prouveur utilisé en sortie. Pour le prédicat d'accessibilité, nous écrivons :

```
/*@ predicate
  @   reachable(node p1, node p2)
  @   reads p1->l,p1->r */
```

Cette déclaration est un prédicat binaire sur les nœuds, mais sa définition n'est pas donnée. La clause `reads` qui suit la déclaration du prédicat fournit des informations sur les données dont dépend le prédicat. Ici, le prédicat dépend de `p1` et `p2` mais aussi des fils de `p1`.

Cela peut paraître confus au début, parce que le prédicat ne dépend pas seulement des fils de `p1`, mais aussi de tous les descendants possibles de `p1`[2]. Pour vraiment comprendre la signification de la clause `reads` vu dans la section 2.2, on a besoin de se rappeler que nous utilisons un modèle 'component-as-array' : cette déclaration signifie en effet que le prédicat aura, pour le prouveur, des arguments de mémoire supplémentaire `l` et `r` représentant respectivement le fils gauche et le fils droit de tous les nœuds. Nous y reviendrons dans la section 3.2.3 où nous définirons `reachable` en Coq.

Une fois ce prédicat `reachable` déclaré, il est possible de donner une spécification formelle de l'algorithme, même si nous ne donnons pas la sémantique de `reachable` immédiatement. La spécification, dans la syntaxe Caduceus, est donnée à la figure 3.3. Notez que dans notre langage d'annotation hérité du C les entiers et les booléens ne sont pas distingués clairement : quand une expression entière e est utilisé comme un atome logique, alors elle peut être vue comme un booléen $e \neq 0$, de ce fait on peut parfaitement écrire `!x->m` ou `x->m == 0`. Notez aussi que lorsque que l'on utilise la logique standard du premier ordre : toutes les fonctions sont totales de ce fait une expression `p->f` est définie même si `p` est le pointeur nul, mais pas sa signification (ceci est très similaire a la division par zéro par exemple).

La pré-condition spécifie à l'aide du langage vu à la section 2.2 que pour tout nœud x qui est non nul et accessible depuis la racine, il est correctement alloué et son champ m est faux (cela signifie que x n'est pas marqué).

La post-condition est la conjonction de trois assertions. La première assertion spécifie donc que pour tous les nœuds (même ceux non accessibles), les fils sont les mêmes avant et après l'algorithme. Ceci est bien sûr très important à spécifier, parce que durant l'exécution de l'algorithme les fils sont modifiés et l'on cherche à prouver que la structure du graphe initiale est restaurée. La deuxième assertion spécifie que tous les nœuds non nuls x accessibles depuis la racine sont marqués. Cela signifie que tout le graphe a été parcouru. La troisième et dernière assertion spécifie que pour tous les nœuds qui ne sont pas accessibles depuis la racine, leurs marques sont inchangées par l'algorithme, ce qui signifie que l'algorithme ne traverse pas les nœuds non accessibles.

Contrairement à la preuve donné par Bornat [33] ou la preuve faite dans Isabelle/HOL [85], nous avons ajouté deux choses : premièrement la pré-condition que tous les pointeurs du graphes sont régulièrement alloués (à l'aide du prédicat `\valid`) au début, sans cela on ne peut prouver qu'il n'y aura pas de déréférencement de pointeur nul. Deuxièmement, nous avons aussi parlé des nœuds non accessibles du graphe : nous sommes capables de montrer qu'ils ne sont ni visités ni modifiés. Dans la section 3.2.4, nous allons ajouter plus d'annotations afin d'établir la terminaison de la fonction `schorr-waite`.

3.2 Vérification

La fonction C `schorr_waite` est maintenant formellement spécifiée. Maintenant le vrai travail commence : nous devons prouver que l'implémentation de la figure 3.1 satis-

[2]Cette confusion sera levée dans une prochaine version du langage de spec : ACSL `http://www.frama-c.cea.fr/acsl.html`

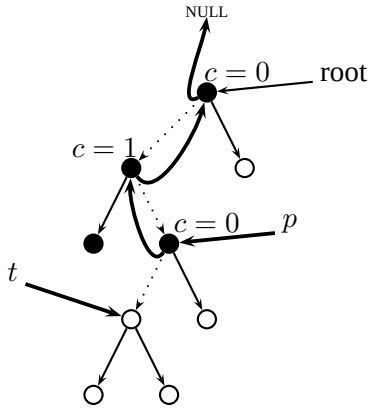


FIG. 3.4 – La pile de backtrack dans la structure du graphe

fait bien la spécification. Si nous lancions l'outil Caduceus sur le code annoté, il générerait certaines conditions de vérification improuvable : comme d'habitude avec un système basé sur la logique de Hoare, quand nous avons une boucle comme la boucle `while`, il est obligatoire (exception faite des cas les plus simples) de manuellement ajouter un *invariant de boucle*, c'est une assertion vraie pour chaque itération de la boucle. Trouver un invariant de boucle approprié est la part la plus difficile du travail de vérification, mais des invariants de boucle appropriés ont été trouvés lors de travaux précédents [33]. Donc, quelques difficultés mise à part, le travail principal fut d'exprimer l'invariant de boucle dans la syntaxe de Caduceus, qui est limité à la logique du premier ordre. La spécification en logique d'ordre supérieur de [85] ne peut être utilisé tel quel dans Caduceus. Mais nous nous en sommes inspirés pour notre approche.

3.2.1 Conception de l'invariant de boucle

L'idée principale pour l'invariant de boucle est que même si la structure de graphe est modifiée, l'accessibilité des nœuds est préservée. Plus précisément, chaque nœud qui est initialement accessible depuis la racine sera toujours accessible soit depuis p soit depuis t. Nous avons aussi besoin de décrire dans l'invariant de boucle comment la pile de backtrack est implémentée à l'aide des pointeurs du graphe. Sur la figure 3.4, nous montrons un état du graphe lors de l'itération de la boucle. Les flèches en pointillé correspondent à la structure initiale du graphe, les flèches épaisses correspondent à l'état courant du graphe et les autres flèches correspondent aux liens non modifiés. La pile de backtrack est la liste des pointeurs, commençant à p, puis on suit le fils gauche lorsque c est à 0, où on suit le fils droit lorsque c est à 1.

L'invariant de boucle écrit dans le langage d'annotation de Caduceus est présenté sur la figure 3.5. Pour comprendre cet invariant de boucle et particulièrement la pile, nous avons besoin de parler de listes de pointeurs dans l'annotation. Avec Caduceus, un type de données peut être importé depuis la logique, implicitement. Alors, de nouveaux prédicats et de nouveaux symboles de fonctions logiques peuvent être déclarés, comme le prédicat

```
void schorr_waite(node root) {
  node t = root; node p = NULL;
  /*@ invariant
    @  (I1 :: \forall node x;
    @    \old(reachable(root,x)) =>
    @                          reachable(t,x) || reachable(p,x))
    @ &&
    @  (I2 :: \forall node x; x != \null =>
    @         (reachable(t,x) || reachable(p,x)) =>
    @                                    \old(reachable(root,x)))
    @ &&
    @  (I3 :: \forall node x;
    @       ! \old(reachable(root,x)) => x->m == \old(x->m))
    @ &&
    @  \exists plist stack;
    @    (I4a :: clr_list (p,stack))
    @  &&
    @    (I4b :: \forall node p; in_list (p,stack) => p->m)
    @  &&
    @    (I4c :: \forall node x; \valid(x) &&
    @                        \old(reachable(root,x)) &&!x->m =>
    @          unmarked_reachable(t,x) ||
    @          (\exists node y; in_list(y,stack) &&
    @                                unmarked_reachable(y->r,x)))
    @  &&
    @    (I4d :: \forall node x;
    @             !in_list(x,stack) => (x->r == \old(x->r) &&
    @                                        x->l == \old(x->l)))
    @  &&
    @    (I4e ::
    @      \forall node p1; \forall node p2;
    @        pair_in_list(p1,p2,cons(t,stack)) =>
    @            (p2->c => \old(p2->l) == p2->l &&
    @                                           \old(p2->r) == p1)
    @          &&
    @            (!p2->c => \old(p2->l) == p1
    @                                    && \old(p2->r) == p2->r))
    @*/
  while (p != NULL || (t != NULL &&! t->m)) {
    ...
```

FIG. 3.5 – L'invariant de boucle

`reachable` déjà introduit, ils peuvent prendre comme arguments des types C ou des types logiques. Ici nous déclarons :

```
//@ type plist
//@ logic plist cons(node p, plist l)
//@ predicate in_list(node p, plist l)
/*@ predicate
  @   pair_in_list(node p1, node p2,
  @                plist l) */
```

ce qui introduit le type logique `plist` pour les listes finies de pointeurs, déclare le symbole de fonction logique `cons` utilisé pour ajouter un élément à une liste, déclare le prédicat `in_list` qui est supposé tester si un pointeur appartient à une liste, et finalement le prédicat `pair_in_list`(p_1, p_2, l), qui dit que p_1 et p_2 sont deux éléments consécutifs de la liste l. Comme pour `reachable`, ils sont seulement déclarés, les définitions ou l'axiomatisation de ces fonctions et prédicats seront donné plus tard.

L'invariant de boucle a besoin de la déclaration de ces deux prédicats :

```
/*@ predicate
  @   unmarked_reachable(node p1,node p2)
  @   reads p1->r,p1->l,p1->m */
```

qui spécifie que p_2 est accessible depuis p_1 en ne traversant que des nœuds non marqués ; et

```
/*@ predicate
  @   clr_list(node p,plist stack)
  @   reads p->c,p->l,p->r */
```

qui spécifie que `stack` est une liste de pointeurs obtenue depuis p suivant la figure 3.4. La définition précise sera donnée dans la section 3.2.3.

Dans la figure 3.5, nous utilisons les notations de Caduceus pour nommer les formules à l'aide des doubles deux points. L'invariant de boucle est fait de huit assertions :

I1 indique que tous les nœuds qui étaient initialement accessibles depuis la racine, sont accessibles depuis soit t soit p.

I2 indique que réciproquement, chaque nœud (non nul) accessible depuis soit t soit p était initialement accessible depuis la racine (`root`). Ceci est vrai seulement pour les nœuds non nul, parce que le nœud `null` peut être inaccessible depuis la racine si le graphe est cyclique.

I3 indique que les nœuds initialement inaccessible depuis la racine, voient leurs marquages inchangés.

I4a définit la pile de backtrack comme indique par la figure 3.4, en utilisant le prédicat `clr_list` déclaré précédemment.

I4b indique que tous les nœuds dans la pile (`stack`) sont marqués.

I4c indique que tous les nœuds valides, initialement accessibles depuis la racine, et pas encore marqués, sont accessibles par un chemin fait uniquement de nœuds non marqués depuis t ou depuis un fils droit d'un des nœuds de la pile (`stack`).

I4d indique que tous les nœuds en dehors de la pile (`stack`) ont leurs fils inchangés.

I4e est essentielle pour être en mesure de récupérer correctement les fils des nœuds de la pile, comme on peut le voir sur la figure 3.4 : Si p_1 est dans la pile, suivi par p_2 (ou si p_1 est t et p_2 est la tête de la pile), alors : si p_2`->c` est 1 alors son fils gauche est le même que initialement et son fils droit est p_1 ; et si p_2`->c` est 0 alors son fils gauche est p_1 et son fils droit est le même qu'initialement.

3.2.2 La vérification en utilisant un prouveur automatique

Le lancement de Caduceus sur le code source annoté donne douze conditions de vérification. Cinq d'entre elles sont liées à la spécification comportementale de l'algorithme : l'invariant de boucle vrai quand on entre dans la boucle, la préservation de l'invariant de boucle pour chacune des branches 'pop', 'swing' et 'push', et la validité de la post-condition. Les sept autres sont utilisées pour établir l'absence de déréférencement de pointeur invalide : la première demande de validité est sur `t` à cause de l'accès au champ de structure `t->m` dans la condition d'arrêt du `while`, et les six autres sont similairement des demandes de validité à cause des déréférencement `t->m` dans le premier `if`, `p->c` dans le deuxième `if`, `t->r` dans la branche 'pop', `p->l` dans la branche 'swing', `t->l` et `p->l` dans la branche 'push'. On peut remarquer que certains accès ne nécessitent pas de vérification de validité, car la simplicité de ces validités leur permet d'être déchargés automatiquement par Why, par exemple la validité de `p` dans `p->r` de la branche 'pop' est déchargée automatiquement, car elle est précédée de `p->c` et que `p` n'a pas changé entre temps.

Une première passe est de tenter de résoudre ces conditions de vérification à l'aide d'un prouveur automatique. Ici nous présentons les résultats obtenus avec Simplify, qui apparaît être le meilleur sur cet exemple. Quatre des douze conditions de vérification sont prouvées automatiquement, ce sont les validités des déréférencements qui apparaissent dans les branches 'pop', 'swing' et 'push'. En effet, une observation rapide de ces conditions de vérification nous permet de voir qu'elles peuvent être résolues par un raisonnement équationnel : par exemple l'accès de `t->r` dans la branche 'pop' est valide car `t` est égale à l'ancienne valeur de `p`, or la validité de `p` est déjà la condition de vérification rattachée au `p->c` du `if` elle est donc admise ici. Ce n'est pas vraiment une surprise de voir que Simplify ne peut pas prouver toutes les conditions de vérification, car jusqu'à maintenant nous ne lui avons donné aucun définition ou axiomatisation pour les prédicats, et particulièrement pour le prédicat d'accessibilité(`reachable`). En effet, pour établir par exemple la validité de `t->m` dans la condition du `while`, un des besoins est d'utiliser la pré-condition qui dit que tous les pointeurs non nul initialement accessibles depuis la racine(`root`) sont valides. Pour établir que `t` est initialement accessible, nous avons besoin d'utiliser l'assertion `I1` de l'invariant de boucle, en effet `t` est accessible depuis lui-même. Pour aider à la preuve un axiome peut être ajouté facilement pour cet exemple :

```
/*@ axiom reachable_refl :
```

```
@   \forall node p; reachable(p,p) */
```

En effet grâce à ce nouvel axiome, Simplify prouve automatiquement les trois conditions de vérification restantes touchant à la validité du déréférencement de pointeurs : ainsi nous avons certifié automatiquement l'absence de menace.

A ce point, toutes nos tentatives d'axiomatisation pour résoudre le reste des conditions de vérification automatiquement ont échouées. Nous avons aussi essayé sur d'autres prouveurs automatiques supportés comme haRVey et CVC-lite, mais ils prouvent encore moins que Simplify. Cette preuve faite automatiquement par Simplify est bien sûr incomplète et les conditions de vérification que nous avons sont apparement trop compliquées pour elles, alors nous essayons maintenant l'utilisation d'un prouveur interactif : l'assistant de preuve Coq.

3.2.3 La vérification en utilisant le prouveur interactif Coq

Contrairement à Simplify, nous avons maintenant toute la puissance du langage de spécification de Coq ; en particulier nous n'avons pas besoin d'axiomatiser les prédicats du code source C, mais nous pouvons les *définir*, et plus particulièrement pour le prédicat d'accessibilité `reachable`, il est naturel d'utiliser une définition inductive.

3.2.3.1 Compléments à la librairie des listes de Coq

– Le prédicat **llist** qui permet de définir une liste de pointeur à l'aide d'une fonction `next` qui permet de connaitre le pointeur qui suit un pointeur donné :
```
Definition plist := list pointer.
```

où `list` est le type des listes polymorphes de Coq. On rappelle qu'en Coq `nil` désigne la liste vide, `: :` l'ajout en tête, `In` l'appartenance à une liste et `++` la concaténation.

```
Definition in_list := (@In pointer).

(** * Paths *)

(** [(lpath t p1 l p2)] :
    there is a path from pointer p1 to pointer p2 using links
    in store t, and the list of pointers along this path is l. *)

Inductive lpath (a : alloc_table)
 (next : pointer → pointer) : pointer → plist → pointer → Prop :=
  | Path_null : ∀ p :pointer, lpath a next p nil p
  | Path_cons :
      ∀ p1 p2 :pointer,
        valid a p1 →
        ∀ l :list pointer,
```

```
          lpath a next (next p1) l p2 →
            lpath a next  p1 (cons p1 l) p2.

Lemma lpath_eq_fun : ∀ (a : alloc_table)
  (f g :pointer → pointer) (p q : pointer)
(l : list pointer),
 (∀ p, In p l → f p = g p) →
 lpath a f p l q  → lpath a g p l q.

(** * Lists *)

(** [(llist t p l)] : there is a (finite) linked list
    starting from pointer p using links in store t,
    and this list of pointers is l *)

Definition llist (a : alloc_table) (next : pointer → pointer)
  (p :pointer) (l :plist) :=
  lpath a next p l null.


Lemma split_list :
  ∀ (A :Set)(x :A) l, In x l →
  ∃ l1, ∃ l2, l = l1 ++ x :: l2.
```

– Le prédicat **no_rep** qui permet de définir une liste de pointeur sans répétition :

```
Fixpoint no_rep (l :list pointer) {struct l} : Prop :=
match l with
| nil ⇒ True
| (a::l) ⇒ ¬In a l  ∧ no_rep l
end.

Lemma no_rep_p : ∀ (p1 :pointer)(lp1 lp2 :list pointer),
  no_rep(lp1++p1::lp2)→  ¬In p1 lp1.

Lemma no_rep_false : ∀ (p1 : pointer)(lp1 lp2 : list pointer),
¬ no_rep (p1 ::lp1 ++ p1 :: lp2).
```

– On prouve quelques lemmes montrant que les sous listes d'une listes sans répétition sont sans répétition :

```
Lemma no_rep_sublist_left : ∀ (l2 l1  : list pointer),
  no_rep (l1 ++ l2) → no_rep l1.

Lemma no_rep_sublist_right :
  ∀ (l1 l2 : list pointer),
  no_rep (l1 ++ l2) → no_rep l2.
```

```
Lemma no_rep_remove_one : ∀ (l1 l2 : list pointer)
  (a : pointer), no_rep (l1 ++a::l2) →
  no_rep (l1++l2).

Lemma no_rep_exchange : ∀ (l1 l2 : list pointer)
  ( a : pointer), no_rep (l1 ++a::l2) →
  no_rep (a::l1++l2).
```

– On termine par deux lemmes de préservation de l'inclusion de liste sans répétition par suppression d'un élément :

```
Definition included(l1 l2 : list pointer) :=
  ∀ q : pointer, In q l1 → In q l2.

Lemma no_rep_in_app :
  ∀ p : pointer,
  ∀ lp1 lp2a lp2b : list pointer,
  no_rep (p::lp1) →
  included (p::lp1) (lp2a++p::lp2b) →
  included lp1 (lp2a ++ lp2b).

Lemma no_rep_in_app2 :
  ∀ p : pointer,
  ∀ lp1 lp2a lp2b : list pointer,
  no_rep (lp2a++p::lp2b) →
  included (lp2a++p::lp2b) (p::lp1) →
  included (lp2a ++ lp2b) lp1.
```

3.2.3.2 La vérification du programme

La première étape est de définir le prédicat d'accessibilité `reachable` dans Coq. Nous procéderons légèrement différemment que Mehta et Nipkow [85], qui utilisent une axiomatisation de la théorie des ensembles : nous utiliserons les définitions inductives, qui sont plus appropriées pour le raisonnement dans Coq. Nous commençons par introduire un prédicat général `path` qui précise le chemin entre deux nœuds c'est-à-dire : $path(p_1, p_2, lp)$ indique qu'il existe un chemin lp entre p_1 et p_2. Il est défini avec ces trois clauses :

```
Inductive path (a : alloc_table) (l : memory pointer)
(r : memory pointer) :
pointer →  pointer → list pointer → Prop :=
  | Path_null : ∀ p :pointer, path a l r p  p nil
  | Path_left :
      ∀ p1 p2 :pointer,
      ∀ lp : list pointer,
        valid a p1 →
          path a l r (acc l p1) p2 lp →
            path a l r p1  p2 (p1::lp)
```

```
  | Path_right :
     ∀ p1 p2 :pointer,
      ∀ lp : list pointer ,
        valid a p1 →
          path a l r (acc r p1)  p2 lp →
            path a l r p1 p2 (p1::lp).
```

le premier dit que l'on peut aller d'un nœud a lui-même par le chemin vide, le deuxième (resp. le troisième) dit que s'il y a un chemin s depuis le fils gauche (resp. la droite) de p_1 à p_2 alors $cons(p_1, s)$ est un chemin de p_1 à p_2. Le prédicat `reachable` est défini grâce à l'existence d'un chemin (resp. `unmarked_reachable` par un chemin avec des nœuds non marqués) :

```
Definition reachable (a : alloc_table)
  (l : memory pointer)(r : memory pointer)
    (p1 :pointer) (p2 :pointer) : Prop :=
       ∃ lp : list pointer, path a l r p1 p2 lp.

Definition unmarked_reachable (a : alloc_table) (m :memory ℤ)
  (l r : memory pointer) (p1 p2 :pointer) : Prop :=
       ∃ lp : list pointer,
        (∀ x : pointer, In x lp → (acc m x) = 0)
           ∧ path a l r p1 p2 lp.
```

On remarque qu'il est maintenant facile d'établir la validité de l'axiome `reachable_refl`. Il est surprenant au début de voir que le prédicat `reachable` défini ici a cinq arguments au lieu de deux, comme déclaré dans la section 3.1.2. Ceci est l'effet de la clause `reads` : comme dit dans la section 2.2, ceci est dû au modèle 'component-as-array' : le prédicat `reacheable` a deux arguments en C, mais dans le modèle on lui donne des arguments supplémentaires a, l, r qui sont les tableaux indexés par les pointeurs (avec acc comme fonctions d'accès). a correspond à une table d'allocation, qui donne pour chaque pointeur s'il est alloué ou non, et l et r correspondent aux champs de la structure `node`. Il est évident que pour effectuer une vérification d'un programme C avec Caduceus et un assistant de preuve, il faut en comprendre plus sur le modèle vu dans la section 2.3.3.

La prochaine étape est de s'occuper du type `plist` utilisé pour modéliser la pile : nous utilisons simplement les listes finies de la librairie standard de Coq. Le prédicat `clr_list` peut être défini inductivement :

```
Definition clr_list (a : alloc_table) (c :memory ℤ)
  (l : memory pointer) (r : memory pointer) :
  pointer →list pointer→ Prop :=
    let next t :=
      if Z_eq_dec (acc c t) 0
        then (acc l t)
```

```
      else (acc r t)
    in
  llist a next.
```

Quelques lemmes doivent être prouvés avant la preuve des conditions de vérifications, le principal, réutilisé plusieurs fois dans la preuve, est de montrer que quand p_2 est accessible depuis p_1, alors il y a un chemin entre p_1 et p_2 sans cycle :

```
Lemma path_no_cycle : ∀ (a : alloc_table)
  (p1 p2 : pointer) (l r : memory pointer)
  (pa : list pointer), path a l r p1 p2 pa →
    ∃ pa' :list pointer,
      incl pa' pa ∧ no_rep pa' ∧ path a l r p1 p2 pa'.
```

D'autres lemmes encore sont utiles comme le lemme `split_path` qui permet de couper un chemin en deux :

```
Lemma split_path : ∀ (a : alloc_table) (p1 p2 : pointer)
  (l r : memory pointer) (path2 path1 : list pointer),
    path a l r p1 p2 (path1++path2) → ∃ p3 : pointer,
    path a l r p1 p3 path1 ∧ path a l r p3 p2 path2.
```

les lemmes `path_upd_right` et `path_upd_left` qui affirment que si on modifie les mémoires r ou l en dehors des pointeurs présent dans un chemin alors ce chemin est toujours là.

```
Lemma path_upd_right :
  ∀ (alloc : alloc_table) (l r : memory pointer)
  (p :pointer) (lp : list pointer) (p1 p0 p2 :pointer),
    ¬ In p lp → path alloc l r p1 p0 lp →
    path alloc l (upd r p p2) p1 p0 lp.

Lemma path_upd_left :
  ∀ (alloc : alloc_table) (l r : memory pointer)
  (p :pointer) (lp : list pointer) (p1 p0 p2 : pointer),
    ¬ In p lp → path alloc l r p1 p0 lp →
    path alloc (upd l p p2) r p1 p0 lp.
```

Avec ces définitions et ces lemmes, nous sommes capables de compléter manuellement toutes les conditions de vérification en Coq. Nous donnons quelques chiffres à la fin de la section suivante.

3.2.4 Terminaison

Pour prouver la terminaison de cet algorithme il suffit de montrer la terminaison de la boucle `while`. Nous utilisons la clause *variant* (vu à la section 2.2) avec une expression qui décroît suivant un ordre bien fondé. Il n'est pas difficile de trouver quel est cette expression

```
//@ type weight_type
/*@ logic weight_type weight(node p, node t)
                    reads p->m,p->c,p->l,p->r  */
/*@ predicate reachable_elements(node root, plist s)
                    reads root->l,root->r */
/*@ requires
       \exists plist s; reachable_elements(root,s) &&
     ... */
void schorr_waite(node root) {
  node t = root; node p = NULL;
  /*@ invariant ...
    @ variant weight(p,t) for order_mark_m_and_c_and_stack
    @*/
  while (p != NULL || (t != NULL && ! t->m)) {
   ...
```

FIG. 3.6 – Annotation supplémentaire pour la terminaison

pour l'algorithme de Schorr-Waite. Il y a un triplet d'entier qui décroît lexicographiquement à chaque itération de la boucle : le nombre de nœuds accessibles non marqués ; le nombre de nœuds accessibles qui ont leur $c = 0$; la taille de la pile. Il reste une question mineure : dans notre modèle, il n'y a aucune règle sur la finitude de la mémoire, alors rien ne dit que l'ensemble des nœuds accessibles est fini. Donc nous avons dû admettre ceci en l'insérant dans la pré-condition. Cette annotation supplémentaire pour la terminaison est donnée dans la figure 3.6, que nous définissons en Coq avec :

```
Definition reachable_elements (a :alloc_table)
  (l r :memory pointer) (p t : pointer)
  (lp : list pointer) : Prop :=
  no_rep lp ∧
  ∀ p2 : pointer, p2 ≠ null →
    (reachable a l r p p2 ∨ reachable a l r t p2)
    ↔ In p2 lp.
```

Nous devons définir la fonction `weight` et la relation `order_mark_m_and_c_and_stack` conformément à la définition informelle de la mesure précédente. Pour cela il faut exprimer les trois quantités du triplet d'entiers. Pour y parvenir on définit en Coq, le type `weight_type` comme un record qui contient toutes les données du programme C nécessaires au calcul de ces trois quantités.

```
Record weight_type : Set := weight
  {
    wa : alloc_table;
    wm : memory ℤ;
    wc : memory ℤ;
```

```
    wl : memory pointer;
    wr : memory pointer;
    wp : pointer;
    wt : pointer
  }.
```

Cette définition introduit automatiquement la fonction `weight` avec le profil voulu conformément à sa déclaration logique Caduceus donnée figure 3.6.

La difficulté est que la fonction qui a un tel `record` associe le triplet d'entiers cherché n'est pas totale : le nombre d'éléments non marqués accessibles n'est pas directement calculable.

Pour s'en sortir, on va définir la relation d'ordre voulue comme *l'image inverse* de l'ordre lexicographique sur les triplets d'entiers par une *relation* entre un type `weight_type` et un triplet d'entiers. La preuve qu'un tel ordre est bien fondé se trouve maintenant dans la librairie standard de Coq dans le module `Wellfounded.Inverse_Image`.

```
Variables A B : Set.
Variable R : B → B → Prop.
Variable F : A → B → Prop.
Let RoF (x y :A) : Prop :=
  exists2 b : B, F x b & (∀ c :B, F y c → R b c).

Theorem wf_inverse_rel : well_founded R → well_founded RoF.
```

Il faut remarquer que lors de notre preuve de `Schorr-Waite` nous avions réalisé nous même un tel développement qui a été repris dans la librairie standard de Coq.

Ici, pour `B` nous utilisons les triplets de `nat` et pour `R` l'ordre lexicographique bien fondé standard. Nous commençons par définir nos trois fonctions partielles :

```
Fixpoint mesure_mark (m : memory ℤ) (l : list pointer)
  {struct l} : nat :=
match l with
| nil ⇒ 0
| p::l ⇒ plus ( if Z_eq_dec (acc m p) 0 then 1 else 0)
  (mesure_mark m l)
end.

Definition interp_mark_m (e : weight_type) (n :nat) : Prop :=
 ∃ lp : list pointer,
   reachable_elements e.(wa) e.(wl) e.(wr) e.(wp)
     e.(wt) lp ∧ (mesure_mark e.(wm) lp)=n.

Definition interp_mark_c (e : weight_type) (n :nat) : Prop :=
  ∃ lp : list pointer,
    reachable_elements e.(wa) e.(wl) e.(wr) e.(wp) e.(wt) lp ∧
```

```
      (mesure_mark e.(wc) lp)=n.

Definition interp_stack (e : weight_type) (n :nat) : Prop :=
  ∃ stack : list pointer,
    clr_list e.(wa) e.(wc) e.(wl) e.(wr) e.(wp) stack ∧
      (List.length stack)=n.
```

Nous les combinons en une fonction partielle sur les triplets :

```
Definition interp_mark_m_and_c_and_stack (e : weight_type)
  (t : natnatnat) : Prop :=
    let (i,p) := t in
    let (j,k) := p in
      interp_mark_m e i ∧ interp_mark_c e j ∧ interp_stack e k.
```

Et notre relation bien fondée est alors définie par :

```
Definition order_mark_m_and_c_and_stack : weight_type →
weight_type → Prop :=
   RoF lex_natnatnat interp_mark_m_and_c_and_stack.
```

Avec les annotations de code complétées, Caduceus génère deux conditions de vérification supplémentaires. La première est de montrer que l'ordre est bien fondé, puis considérant qu'il l'est la seconde condition de vérification est trivialement résolu par un raisonnement équationnel. La vérification que la mesure décroît à chaque itération de la boucle est ajouté à la condition de vérification qui préserve l'invariant de boucle.

La condition de vérification sur la bonne fondaison ne passe pas au prouveur automatique comme Simplify, car c'est une formule du second ordre. Donc au final, le prouveur Simplify prouve huit des treize conditions de vérification, ce qui est un bon score, mais bien sûr les cinq qui restent sont les plus difficiles.

Avec Coq, nous complétons toutes les conditions de vérification même la bonne fondaison. Voici quelques résultats sur la preuve Coq entière : les définitions de prédicats sur les listes, l'accessibilité et le reste, plus les lemmes correspondants représentent 317 lignes de définitions et 589 lignes de tactiques de preuve. Les conditions de vérification génèrent 985 lignes et 2411 lignes de tactiques de preuves ont été insérées manuellement pour les prouver. 40% du travail a été dévolu à la preuve de terminaison, en termes de lignes de texte Coq et de temps passé : prouver les conditions de vérifications sans la terminaison a nécessité à peu près trois semaines de travail et la preuve de terminaison a nécessité deux semaines de plus. Ce temps prend en compte les nombreuses itérations du processus de modification des annotations, régénération des conditions de vérification et modification du script de preuve.

Ces résultats peuvent sembler trop imposants pour un programme de seulement 10 lignes de code C. Mais bien sûr cet algorithme est particulièrement subtil et ce n'est pas

surprenant que sa justification soit difficile. Une autre raison est que nous avons dû définir dans Coq beaucoup de définitions et lemmes préliminaires (concernant l'accessibilité) à partir de zéro : nous aurions pu nous passer de ceci si Coq possédait une librairie riche sur les graphes. En effet, une partie de nos préliminaires a été incluse dans les versions suivantes de Coq. Comparé à la preuve similaire en Isabelle/HOL [85], nous avons trois fois plus de lignes de preuve : nous croyons que ceci est dû à la faiblesse des tactiques automatiques de Coq (mais il faut se rappeler également que nous avons les preuves supplémentaires de l'arrêt et de l'absence des menaces).

3.3 Bilan et comparaison avec d'autres approches

Nous avons présenté une preuve formelle de l'algorithme de Schorr-Waite entièrement vérifié par l'ordinateur, directement sur une version du code source réel en C. Nous avons prouvé ces propriétés comportementales, comme les études précédentes [33, 85, 21], mais nous avons aussi prouvé que le déréférencement de pointeur invalide ne peut se produire, et la terminaison de l'algorithme.

La preuve formelle de terminaison de cette algorithme n'est pas complètement nouvelle : la terminaison est en effet établie par Abrial [21] comme une condition pour le raffinement. Le point important est que un mécanisme de preuve de terminaison est intégrée à notre méthodologie Caduceus, grâce à la puissante clause `variant`, qui permet l'utilisation d'une relation bien fondée arbitraire.

Concernant l'absence de déréférencement de pointeur invalide, les techniques avancées d'analyse statique peuvent aussi l'établir. L'algorithme de Schorr-Waite est un des exemples automatiquement manipulé par l'approche de TVLA [107, 103], où une puissante analyse d'accessibilité dans la structure des pointeurs est faite. Mais jusqu'à présent, autant que nous en sachions, uniquement notre approche peut manipuler en même temps les vérifications de déréférencement de pointeur et la validité de propriétés comportementales donné par l'utilisateur. Remarquons qu'avec Caduceus, comme vu dans la Section 3.2.2, l'absence de déréférencement de pointeur invalide est établie automatiquement, même si Caduceus ne fait aucune analyse d'accessibilité lui-même.

En conclusion, nous soulignons encore que nous avons utilisé un outil générique pour des programmes C arbitraires, sans aucun ajout particulier pour ce cas d'étude, qui de plus donne le choix du prouveur final. Au début, il n'était pas clair que le langage de spécification de Caduceus soit suffisant, en particulier la langage de spécification de Caduceus ne traite que la logique du premier ordre, mais finalement nous avons réussi. Ce cas d'étude est un véritable succès, qui montre de façon évidente que la méthodologie de Caduceus [59] est puissante.

Chapitre 4

Un langage C normalisé

L'outil Caduceus, comme vu dans le chapitre 2, est similaire à un compilateur. En général, pour séparer les différentes phases de la compilation, un compilateur manipule des langages intermédiaires internes. Ce n'était pas le cas auparavant dans Caduceus, mais pour les besoins de l'analyse de séparation, qui sera présentée au chapitre 5, nous avons éprouvé le besoin de définir un tel langage intermédiaire.

Utiliser des langages intermédiaires, dans un cadre de vérification formelle n'est pas nouveau, par exemple Cminor [5] est un langage de programmation impératif et des compilateurs de C à Cminor ont été prouvés, ainsi que de Cminor au langage machine.

Notre langage intermédiaire que nous appelons le C normalisé est un sous-ensemble du langage C. Une partie du langage C peut se réduire à ce langage normalisé.

Dans ce chapitre nous présentons (section 4.1) les motivations qui nous ont amenées à créer ce nouveau langage, le sous-ensemble du C que nous traitons (section 4.2), la définition du C normalisé (section 4.3), la traduction du C vers le C normalisé (section 4.4), la traduction de C normalisé vers le Why (section 4.5) et enfin nous décrirons (section 4.6) les limitations de cette normalisation.

4.1 Motivations

Dans le langage C, il existe plusieurs variantes d'accès à la mémoire ($*p$, $t.f$, $t \rightarrow f$ et $t[i]$). Pour l'analyse statique des programmes, distinguer ces variantes n'a pas d'intérêt que ce soit pour la présentation théorique, ou pour l'implémentation. De ce fait, Caduceus transforme le fragment du langage C traité que nous présentons à la section 4.2 pour le transformer en C normalisé où les opérateurs d'adressages `&`, de déréférencements `*`, les accesseurs aux tableaux `t[i]` et les accesseurs aux champs de structures `t.f` ont disparu. Le fait de supprimer toutes ces notations différentes permet d'uniformiser les traitements et ainsi diminuer les erreurs possibles.

4.2 Le sous-ensemble du C ANSI traité

Nous ne traitons pas l'ensemble du langage C ANSI. En effet, certaines constructions de C ne sont pas traitées tel que les `goto` (section 4.2.1), les pointeurs de fonction (section 4.2.2) ou les `unions` et `cast` (section 4.2.3). La grammaire du langage C que nous traitons est présenté dans la figure 4.1 avec :

- e : représente les expressions du langage
- s : représente les instructions du langage
- lv : représente les valeurs gauche du langage
- d : représente les déclarations globales du langage
- prog : représente le programme en entier

Pour une question de lisibilité, les affectations sont mises en position d'instruction ainsi que les incrémentations, mais ceci n'est pas une limitation, notre implémentation supporte celle-ci en position d'expression. De plus, les instructions `for` et `do ... while` ne sont pas présentes dans cette gramaire, car elles sont analogues au `while`. Ceci non plus n'est pas une limitation. L'instruction `continue` quant à elle n'est pas significative pour la suite de l'étude.

4.2.1 Instruction `goto` arbitraire

Le `goto` est une instruction C qui permet de réaliser des sauts vers des points en avant ou en arrière du programme comme sur cet exemple :

```
void f(int n){
  int i;
  a :
  for (i = 0; i < n; i++){
    if (i = 5)
      goto a;
  }
}
```

Cette fonction, qui sans le `goto` est très simple, ne terminera jamais pour les `n` supérieurs à 5. Le support de l'instruction `goto` est délicat en Caduceus car cette instruction n'est pas traduisible directement en `Why` (voir la section 2.3.1).

Dans cette thèse, nous ignorons l'instruction `goto`. Cette instruction n'est pas, ou très peu, utilisée sur les codes critiques du fait des nombreux bugs qu'elle peut provoquer, elle est , de plus, souvent interdite par les règles industrielles d'écriture de code critique.

Remarquons que l'outil Caduceus traite les `goto` vers l'avant, qui ne sautent pas à l'intérieur d'un bloc. Il ne serait pas impossible de supporter des `goto` arbitraires : on peut envisager des méthodes de transformation de code comme celle de Lyle Ramshaw [101], mais celles-ci peuvent avoir l'inconvénient de dupliquer du code (donc des conditions de vérification). Il existe des méthodes de calculs de plus faible pré-condition qui peuvent s'appliquer directement (sans transformer le code) et pour lequel on peut supporter un langage de spécification qui permet de placer des invariants à des points arbitraires du code (voir le

e	::=	c	(constantes)
	|	v	(variables)
	|	$e[e]$	(accès aux tableaux)
	|	$e.f$	(accès aux champs de structures)
	|	$e \rightarrow f$	(accès aux champs de structures)
	|	$id(e, \ldots, e)$	(appel de fonction)
	|	$e\ op\ e$	(opérateur +,-,*,/,%, &&, etc.)
	|	$\&lv$	(adressage d'une valeur gauche)
s	::=	$lv = e;$	(affectation)
	|	$\tau\ v = e;$	(introduction de variable locale)
	|	$\tau\ v[e] = e, \ldots,\ e;$	(introduction de tableau local)
	|	$lv + +;$	(incrémenteur)
	|	$lv - -;$	(décrémenteur)
	|	**return** $e;$	(retour de fonction)
	|	**if**(e) s **else** s	(branchement de condition)
	|	**while**(e) s	(boucle while)
	|	**switch**(e) **case** $c : s; \ldots$ **case** $c : s;$ **default** $: s;$	(branchement de condition switch)
	|	$s\ s$	(sequence)
	|	**break**;	(sortie d'un bloc)
lv	::=	v	(variable)
	|	$*e$	(déréférencement)
	|	$e.f$	(déréférencement)
	|	$e \rightarrow f$	(déréférencement)
	|	$e[e]$	(accès dans un tableau)
d	::=	$\tau x;$	(introduction de variable globale)
	|	$\tau x[e];$	(introduction de tableau global)
	|	$\tau x = e;$	(introduction de variable globale initialisée)
	|	$\tau x[e] = e, \ldots,\ e;$	(introduction de tableau global initialisé)
	|	**struct** $id\{\tau\ x; ...\};$	(introduction de structure)
	|	$\tau\ f(...)\{s\}$	(introduction de fonction)
	ϵ	(vide)	
$prog$	::=	$d\ prog$	(le programme)
	|	ϵ	

FIG. 4.1 – Grammaire du C ANSI traité

langage de spécification ACSL `http://www.frama-c.cea.fr/acsl.html`). Dans la plate-forme Why ceci a été mis en oeuvre pour la preuve de programmes Mix [57].

4.2.2 Pointeurs de fonction

Les pointeurs de fonction en langage C permettent d'avoir l'ordre supérieur. Or l'outil `Why`, vu à la section 2.3.1, vers lequel nous traduisons le langage C ne permet pas l'ordre supérieur sur les fonctions avec effet. La traduction des pointeurs de fonction du langage C vers le langage `Why` nécessiterait un travail supplémentaire. De ce fait, nous avons, dans un premier temps, dû restreindre la portion du langage C accepté par Caduceus au programme sans pointeur de fonction. De fait il n'y a pas de pointeur de fonction dans les codes industriels que nous avons analysés.

4.2.3 Types `union` et `cast`

Les `unions` et les `cast` sont des instructions qui permettraient de modifier le type d'une donnée. En effet, sur cet exemple :

```
typedef struct {
  short x;
  short y;
} S;

typedef union {
  S s;
  long z;
} U;


int f(U u){
  u.s.x = 0;
  u.s.y = 0;
  return u.z;
}

int g(S s){
  s.x = 0;
  s.y = 0;
  return (long) s;
}
```

les instructions `union` et `cast` devraient permettre de transformer deux variables de type `short` en une variable de type `long`. Or cette transformation ne respecte pas la norme ANSI [76]. En effet, rien n'indique que les deux champs de la structure `s` se suivent en mémoire. Ce genre de transformations dépendra du compilateur que l'on utilise. De ce fait,

nous avons décidé de rejeter les instructions `cast` et les instructions `union` dans cette thèse.

Contrairement aux cas précédents les `cast` de pointeur peuvent apparaître dans les code critique. La présence de ces constructions dans du code embarqué sera discuté dans la section 7.4.

Dans l'outil Caduceus, les `cast` et les `unions` ne sont pas supportés. Il y a seulement des expérimentations qui ont été ou sont encore menées, par J. Andronick [24] et par Y. Moy [91].

4.3 Définition du C normalisé

Le C normalisé est un sous-ensemble du C : il est compilable par un compilateur C et la transformation du C en C normalisé ne modifie pas, sur une large classe de programmes C, la sémantique des programmes. Les cas qui modifient la sémantique des programmes seront traités dans la section 4.6.2.

Les seuls types de données dans ce langage sont les types de base et les pointeurs vers des structures. Les expressions du langage sont les constantes, les variables, les opérateurs standards, les appels de fonction, les accès aux champs de structures et l'arithmétique de pointeurs.

La grammaire de ce langage est vue à la figure 4.2. On considère qu'il s'agit là d'une syntaxe abstraite en particulier on lève l'ambiguité sur l'addition d'un pointeur et d'un entier en notant cette opération $\oplus$ et sur la soustraction entre pointeurs en la notant $\ominus$. Ici encore, on ne met pas ni la boucle `for`, ni la boucle `do ... while`, ni le `continue` pour simplifier la lecture.

Ce qui a été supprimé est l'opérateur d'adressage `&x`, l'accesseur direct `t.f`, l'accès au tableau `t[i]` et l'accesseur `*p`. Ces constructions du langage C peuvent être traduites vers le C normalisé et c'est l'objet de la section suivante.

4.4 Traduction du C vers le C normalisé

Dans cette section, nous présentons le passage du C au C normalisé. Pour ce faire, nous présenterons, en section 4.4.1, un certain nombre de règles de simplification. Nous traitons ensuite, dans la section 4.4.2 et 4.4.3, des traitements spécifiques relatifs à l'opérateur d'adressage et à la déréférenciation.

4.4.1 Simplification systématique

Les règles de réécriture de la figure 4.3 sont systématiquement appliquées au cours de la normalisation lorsque cela est possible. Ces règles peuvent être appliquées sur un programme C sans en changer la sémantique. Elles nous permettent donc de simplifier le langage en supprimant les syntaxes redondantes.

On remarque notamment qu'après utilisation de ces règles l'expression `e[e]` a totalement disparu du langage.

e	::=	c	(constantes)
	|	v	(variables)
	|	$e \rightarrow f$	(accès aux champs de structures)
	|	$id(e, \ldots, e)$	(appel de fonction)
	|	$e\ op\ e$	(opérateur +,-,*,/,%, &&, etc.)
	|	$e \oplus e$	(addition pointer+entier)
	|	$e \ominus e$	(soustraction de pointeur-pointeur)
	|	$e\ op\ e$	(comparaison de pointeurs)
s	::=	$lv = e;$	(affectation)
	|	$\tau\ v = e;$	(introduction de variable locale)
	|	$\tau\ v[e] = \{e, \ldots, e\};$	(introduction de tableau local)
	|	$e++;$	(incrémenteur)
	|	$e--;$	(décrémenteur)
	|	**return** $e;$	(retour de fonction)
	|	**if**$(e)\ s$ **else** s	(branchement de condition)
	|	**while**$(e)\ s$	(boucle while)
	|	$s\ s$	(sequence)
	|	**break**;	(sortie du bloc)
lv	::=	v	(variable)
	|	$e \rightarrow f$	(déréférencement)
d	::=	$\tau\ x;$	(introduction de variable globale)
	|	$\tau\ x[e];$	(introduction de tableau global)
	|	$\tau\ x = e;$	(introduction de variable globale initialisée)
	|	$\tau\ x[e] = \{e, \ldots, e\};$	(introduction de tableau global initialisé)
	|	$struct\ id\{\tau\ x; ...\};$	(introduction de structure)
	|	$\tau\ f(...)\{s\}$	(introduction de fonction)

FIG. 4.2 – Grammaire du C normalisé

$t[e]$	$\Rightarrow$	$*(t+e)$		(1)
$*(\&lv)$	$\Rightarrow$	lv	(avec lv l'ensemble des valeurs gauches du C)	(2)
$\&(*e)$	$\Rightarrow$	e	(avec e l'ensemble des expressions du C)	(3)
$(*e).f$	$\Rightarrow$	$e \rightarrow f$	(avec e l'ensemble des expressions du C)	(4)

FIG. 4.3 – Règles de simplification du C normalisé

Déclarations :

$$\Gamma \vdash \tau\ x; \rightsquigarrow \tau\ x[1] \quad x \in \Gamma \quad (1)$$
$$\Gamma \vdash struct\ id\ \{\ldots;\ \tau\ f;\ldots\}; \rightsquigarrow struct\ id\ \{\ldots;\tau\ f[1];\ldots\}; \quad f \in \Gamma \quad (2)$$

Expressions :

$$\Gamma \vdash x \rightsquigarrow *x \quad x \in \Gamma \quad (3)$$
$$\Gamma \vdash \&x \rightsquigarrow x \quad x \in \Gamma \quad (4)$$
$$\Gamma \vdash e.f \rightsquigarrow *(e.f) \quad f \in \Gamma \quad (5)$$
$$\Gamma \vdash e \rightarrow f \rightsquigarrow *(e \rightarrow f) \quad f \in \Gamma \quad (6)$$

FIG. 4.4 – Règles de transformation du C normalisé pour traiter l'opérateur d'adressage

4.4.2 Le traitement de l'opérateur d'adressage

L'opérateur & permet de caractériser l'adresse mémoire d'une variable, ce qui n'est pas possible dans le modèle mémoire utilisé par Caduceus (comme vu section 2.3.3). En effet, les variables de types de base n'ont pas d'adresse mémoire dans notre modèle. Pour pouvoir traiter cet opérateur dans notre modèle mémoire une transformation est donc nécessaire. Cette transformation consiste à changer globalement la variable adressée par l'opérateur & en pointeur sur une case mémoire (c'est-à-dire un bloc mémoire de taille 1).

Les règles pour traiter cette transformation sont données dans la figure 4.4. Où Γ est l'environnement contenant :

- toutes variables `x` dont l'expression `&x` apparaît dans le programme
- tous les champs `f` tel que l'expression `&(e.f)` ou `&(e->f)` apparaît dans le programme
- toutes variables `x` de type structure

En effet, comme vu dans la section 2.3.3, les structures ne peuvent être accédées que par l'intermédiaire d'un pointeur. Ainsi c'est comme si toutes les variables de type structure étaient utilisées avec l'opérateur &.

Remarque : Pour toutes les expressions et les instructions qui ne sont pas dans les règles, on applique ces mêmes règles à leurs sous-termes.

Exemple 4.4.1. *Le code :*

```
int x

void f(){
  int *z = &x;
  x = 5;
}
```

devient par l'application des règles (1), (3) et (4) de la figure 4.4 :

```
int x[1];       // regle (1)
```

```
void f(){
  int *z = x; // regle (4)
  *x = 5;     // regle (3)
}
```

Sur cet exemple nous voyons bien la transformation de la variable de type `int : x` *en variable de type pointeur sur* `int`*. De plus, l'opérateur* `&` *disparaît alors que l'accès à la variable* `x` *devient l'appel au contenu du pointeur* `x : *x`*.*

Dans un premier temps, nous pensions traduire cet exemple comme ceci :

```
int *x;

void f(){
  int *z = x;
  *x = 5;
}
```

Or ceci pose un problème. En effet, l'accès $*x$ *va nécessiter une preuve que le nouveau pointeur* x *est alloué, alors que* x *est à l'origine une variable de type* int*. Le pointeur créé par la normalisation est donc forcément alloué. C'est pourquoi nous avons choisi de remplacer la déclaration par un tableau de taille 1, qui permet d'allouer le pointeur.*

Le cas des structures

Comme on le voit dans la figure 4.4 plusieurs règles ne concernent que les structures. En effet, nous considérons les structures comme toujours adressées. Donc l'accesseur `t.f` disparaît.

Exemple 4.4.2. *Le code :*

```
struct S{
  int x;
};

struct S y;

void f(){
  y.x = 5;
}
```

sera remplacé grâce à l'application des règles (1) et (5) de la figure 4.4 et de la règle (4) de la figure 4.3 par :

```
struct S{
  int x;
};
```

```
struct S y[1]; // regle (1)

void f(){
  y->x = 5;     // regle (5)
}
```

De plus, prendre l'adresse d'une structure revient à prendre la structure elle-même. De ce fait, il suffit de supprimer l'opérateur `&` du code sans rien changer d'autre (comme vu dans la règle (4) de la figure 4.4).

Exemple 4.4.3. *L'utilisation des règles (1), (4) et (5) de la figure 4.4 et de la règle (4) de la figure 4.3 permet de réécrire le code :*

```
struct S{
  int x;
} s;

void f (){
  struct S* z = &s;
  s.x = 5;
}
```

en :

```
struct S{
  int x;
} s[1];             // regle (1)

void f (){
  struct S* z = s; // regle (4)
  s->x = 5;         // regle (5)
}
```

où les `&e` *et* `e.f` *ont été supprimés.*

4.4.3 Élimination des déréférenciations

L'opérateur `*` et l'accesseur `t->f` représentent tous deux des déréférencements de pointeurs. Or un des buts du C normalisé est justement de supprimer les notations multiples. Pour ce faire, nous avons choisi de ne garder que la notation `t->f`.

Pour supprimer l'opérateur de déréférencement `*` nous considérons les types des pointeurs sur Θ comme des pointeurs sur structures. Avec Θ l'ensemble des types défini par :

$$\Theta ::= \texttt{int}|\texttt{char}|\texttt{short}|\texttt{long}|\texttt{long long}|\texttt{float}|\texttt{double}|\tau\ \texttt{*}|\tau\ \texttt{[ ]}$$

où τ est un type quelconque.

Déclarations :

$$\begin{array}{rcll} \tau\ *x & \rightsquigarrow & structure(\tau)\ *x & \text{où } \tau \in \Theta \quad (1)\\ \tau x[n] & \rightsquigarrow & structure(\tau)\ x[n] & \text{où } \tau \in \Theta \quad (2)\\ struct\ id\{\ldots;\tau\ *f;\ldots\}; & \rightsquigarrow & struct\ id\{\ldots;structure(\tau)\ *f;\ldots\}; & \text{où } \tau \in \Theta \quad (3)\\ struct\ id\{\ldots;\tau\ f[n];\ldots\}; & \rightsquigarrow & struct\ id\{\ldots;structure(\tau)\ f[n];\ldots\}; & \text{où } \tau \in \Theta \quad (4) \end{array}$$

Expressions :

$$*e \quad \rightsquigarrow \quad e \rightarrow field(\tau) \qquad \text{où } \tau \text{ est le type de e et } \tau \in \Theta \quad (5)$$

FIG. 4.5 – Règle de transformation du C pour supprimer l'opérateur de déréférencement

Premièrement, on a besoin d'une fonction de traduction des typ→
es de Θ vers le nom des structures :

$$\begin{array}{rcl} trad(\texttt{int}) & \Rightarrow & "integer"\\ trad(\texttt{char}) & \Rightarrow & "integer"\\ trad(\texttt{long}) & \Rightarrow & "integer"\\ trad(\texttt{short}) & \Rightarrow & "integer"\\ trad(\texttt{long long}) & \Rightarrow & "integer"\\ trad(\texttt{float}) & \Rightarrow & "real"\\ trad(\texttt{double}) & \Rightarrow & "real"\\ trad(\texttt{struct}S) & \Rightarrow & S\\ trad(\tau\ \texttt{*}) & \Rightarrow & trad(\tau)"P"\\ trad(\tau\ \texttt{[ ]}) & \Rightarrow & trad(\tau)"P" \end{array}$$

et

$$\begin{array}{c} structure(\tau) = trad(\tau)"P"\\ field(\tau) = trad(\tau)"M" \end{array}$$

Remarque : Comme indiqué dans la section 2.3.3 on considère, ici, un modèle idéal pour les entiers et les réels. Le passage à un modèle gérant les entiers machines ne pose pas de problème ici.

On définit, ensuite, une structure pour chaque type de pointeur sur Θ présent dans le programme. Si τ est le type sur lequel pointe le pointeur alors la nouvelle structure définie sera :

$$\texttt{struct}\ structure(\tau)\{\tau\ field(\tau);\};$$

Enfin, il suffit d'appliquer les règles de la figure 4.5 pour terminer l'élimination des déréférencement.

Exemple 4.4.4. *Le code :*

```
int *x;

void f(){
```

```
  *x = 5;
}
```

possède un pointeur sur `int`*. On créé donc la structure* `integerP` *contenant le champ de type* `int integerM` *et on applique les règles (1) et (5) de la figure 4.5 :*

```
struct integerP {     // règle de construction
  int integerM;       // de la structure
};

struct integerP *x;  // règle (1)

void f(){
  x -> integerM = 5; // règle (5)
}
```

Ainsi x est devenu une structure et son accesseur est devenu `->`*.*

Exemple 4.4.5. *L'exemple suivant illustre la traduction des tableaux de pointeurs sur structure. Le code C :*

```
struct S{int a;};

struct S* t[10];

int f(){
  return t[5]->a;
}
```

est normalisé en :

```
struct S {
  int a;
};

struct SPP{
  struct S* SPM;
}

struct SPP t[10];

int f()
{
  return ((t + 5)->SPM)->a;
}
```

Une fois cette transformation de code effectuée, le seul moyen pour accéder au tas mémoire est l'opérateur `t->f`. De ce fait, on se rapproche de la traduction en `Why`, vu dans la section 2.3.1, où il n'y a que deux fonctions : une pour modifier le tas mémoire et une pour y accéder.

4.5 Traduction du C normalisé vers le langage Why

Dans cette section, nous présentons notre nouveau schéma de traduction du C normalisé vers le langage Why. La démarche utilisée avant l'introduction du C normalisé intermédiaire reste la même : en préalable nous avons besoin d'effectuer un calcul des effets (section 4.5.1) pour chaque fonction du programme C normalisé puis nous décrivons des règles d'interprétation (section 4.5.2) qui procèdent par récursion sur la structure du programme. Enfin, nous décrirons à part le cas du `switch` (section 4.5.3) qui est nouveau car n'était pas supporté par Caduceus avant cette thèse.

4.5.1 Le calcul des effets

Rappelons que la démarche de vérification est modulaire : on prouve une fonction en ne considérant que les spécifications des fonctions qu'elle appelle. Dans le programme Why obtenu par traduction, chaque fonction donnera lieu d'une part à une interface Why : une fonction déclarée avec son type avec effet mais sans code, d'autre part une fonction Why avec son implémentation. Le calcul des effets préalable est nécessaire pour déterminer les interfaces Why des fonctions.

Pour calculer les effets des fonctions, il suffit de regarder les fonctions dans l'ordre du graphe d'appel (les fonctions feuilles en premier puis en remontant le graphe d'appel). Puis pour chaque fonction il suffit de traverser les instructions et les expressions récursivement, en récoltant les variables et les champs auxquels on a accédé ou qui ont été modifiés et ajoutant les effets des fonctions appelées.

Pour les fonctions récursives ou mutuellement récursives le calcul des effets est obtenu par une itération de l'inférence d'effet sur le corps de la fonction jusqu'à obtenir un point fixe. Un point fixe sera forcément atteint, car l'ensemble des variables est fini et l'ensemble des effets d'une fonction grandit strictement durant ce processus.

Les annotations logiques doivent aussi être parcourues dans ce calcul des effets pour rajouter des effets en lecture lors de l'appel d'un `logic` ou d'un `predicat` dans les annotations.

Exemple 4.5.1. *Sur ce code :*

```
int x;

struct S{int a;};

int f(struct S *p){
  p->a +=x;
}
```

le calcul d'effets de f sur cet exemple nous donnera en `reads` *: a, x et en writes a.*

4.5.2 Les règles d'interprétations du C normalisé vers Why

La traduction est réalisée à l'aide de six interprétations mutuellement récursives :

1. L'interprétation des expressions du C normalisé. On note donc $[\![e]\!]$ l'interprétation de l'expression e.
2. L'interprétation des expressions du C normalisé comme expressions booléennes Why. Bien sûr, le langage C normalisé ne possède pas d'expression booléennes, mais il est plus naturel d'utiliser une expression booléenne pour traduire les expressions du C normalisé utilisées comme booléens. On note donc $[\![e]\!]_b$ l'interprétation de l'expression e.
3. L'interprétation des instructions du C normalisé. On note donc $[\![s]\!]$ l'interprétation de l'instruction s.
4. L'interprétation des prédicats. On note donc $[\![A]\!]_p$ l'interprétation du prédicat A.
5. L'interprétation des effets de bords. On note donc $[\![A]\!]_a$ l'interprétation de l'ensemble de valeurs gauches A.
6. L'interprétation des types. On note donc $[\![\tau]\!]$ l'interprétation du type τ.

Le traitement des expressions. Nous donnons seulement les constructions les plus significatives. La traduction des accès ou modifications de la mémoire s'effectue à l'aide de deux fonctions abstraites du programme Why :

```
parameter acces : m : α memory ref → p : pointer →
  {valid(alloc,p)} α reads alloc,m {result=acc(m,p)}
parameter update : m : α memory ref → p : pointer → v :α →
  {valid(alloc,p)} unit reads alloc,m writes m {m = upd(m@,p,v}
```

Les expressions du C normalisé sont donc interprétées par une expression Why simple :

- $[\![x]\!] \Rrightarrow !x$
- $[\![e \rightarrow f]\!] \Rrightarrow$ `acces` $f\ [\![e]\!]$
- $[\![e_1 \oplus e_2]\!] \Rrightarrow$ `shift` $[\![e_1]\!]\ [\![e_2]\!]$, lorsque e_1 est un pointeur
- $[\![e_1 \ominus e_2]\!] \Rrightarrow$ `sub_pointer` $[\![e_1]\!]\ [\![e_2]\!]$, lorsque e_1 et e_2 est un pointeur
- $[\![e_1\ op\ e_2]\!] \Rrightarrow [\![e_1]\!] op [\![e_2]\!]$, avec $op \in \{+, -, *, /, \%, \&, \hat{}, |\}$
- $[\![e_1\ op\ e_2]\!] \Rrightarrow$ `if` $[\![e_1\ op\ e_2]\!]_b$ `then` 1 `else` 0, avec $op \in \{==, !=, >, >=, <, <=, \&\&, ||\}$
- $[\![f(e_1, ..., e_n)]\!] \Rrightarrow f\_parameter([\![e_1]\!], ..., [\![e_n]\!])$

Ici, $f\_parameter$ designe *l'interface* Why de la fonction f qui est décrite dans le traitement des instructions. Cette interface ne mentionne que la spécification de la fonction f et pas son corps, conformément à l'approche modulaire (section 2.1.2.2).

Le traitement des expressions booléennes. L'interprétation des expressions booléennes est définie comme ceci :

- $[\![e_1\ op\ e_2]\!]_b \Rrightarrow [\![e_1]\!]\ op\ [\![e_2]\!]$, avec $op \in \{==, !=, >, >=, <, <=\}$
- $[\![e_1\ \&\&\ e_2]\!]_b \Rrightarrow \texttt{if}\ [\![e_1]\!]_b\ \texttt{then}\ [\![e_2]\!]_b\ \texttt{else}\ false$
- $[\![e_1\ ||\ e_2]\!]_b \Rrightarrow \texttt{if}\ [\![e_1]\!]_b\ \texttt{then}\ true\ \texttt{else}\ [\![e_2]\!]_b$
- $[\![!e]\!]_b \Rrightarrow\ not\ [\![e]\!]_b$
- $[\![e]\!]_b \Rrightarrow\ [\![e]\!]\ <>\ 0$, dans tout les autres cas.

Le traitement des instructions. L'interprétation des instructions du C normalisé est définie comme ceci :

- $[\![x = e_2;]\!] \Rrightarrow x := [\![e_2]\!]$
- $[\![e_1 \rightarrow f = e_2;]\!] \Rrightarrow \texttt{update}\ f\ [\![e_1]\!]\ [\![e_2]\!]$
- $[\![s_1\ s_2]\!] \Rrightarrow [\![s_1]\!]; [\![s_2]\!]$
- $[\![\beta_1\ x_1 = e_1;\ ...\ ; \beta_n\ x_n = e_n; s]\!] =$
  $\texttt{let}\ x_1 =\ ref\ [\![e_1]\!]\ \texttt{in}\ ...\ \texttt{let}\ x_n =\ ref\ [\![e_n]\!]\ \texttt{in}\ [\![s]\!]$
  La traduction au-dessus admet que chaque variable x_i possède un initialisateur e_i. Si une variable locale n'est pas initialisée, elle sera en effet initialisé en utilisant l'appel à une fonction non spécifiée, ce qui simule le fait que la valeur d'initialisation est inconnue.
- $[\![/*\ @\texttt{invariant}\ A\ \texttt{loop\_assigns}\ B\ \texttt{variant}\ C\ */\ while(e)\ s]\!] \Rrightarrow$
  $\texttt{try}$
  $\texttt{while}\ [\![e]\!]\ \texttt{do}\ \{\texttt{invariant}\ [\![A]\!]_p\ \texttt{and}\ [\![B]\!]_a\ \texttt{variant}\ [\![C]\!]_p\}$
  $\texttt{try}\ [\![s]\!]\ \texttt{with}\ Continue\ \rightarrow\ void\ \texttt{end done}$
  $\texttt{with}\ Break\ \rightarrow\ void\ \texttt{end}$

- $[\![return\ e;]\!] \Rrightarrow\ raise\ (Return\_int\ [\![e]\!])$ avec e de type int.
- $[\![return\ e;]\!] \Rrightarrow\ raise\ (Return\_real\ [\![e]\!])$ avec e de type $real$.
- $[\![return\ e;]\!] \Rrightarrow\ raise\ (Return\_pointer\ [\![e]\!])$ avec e de type $pointer$.
- $[\![break;]\!] \Rrightarrow\ raise\ (Break)$
- $[\![continue;]\!] \Rrightarrow\ raise\ (Continue)$

Pour finir, une définition de fonction C normalisé sera interprétée en deux éléments :

1. Une définition de fonction Why où l'exception $Return$ est correctement rattrappée :
   $[\![/*\ @\texttt{requires}\ A\ \texttt{assigns}\ B\ \texttt{ensures}\ C\ */\ \tau\ f(\tau_1\ x_1,\ ...,\ \tau_n\ x_n)\{s\}]\!] \Rrightarrow$
   $\texttt{let}\ f\_impl(x_1 : [\![\tau_1]\!])\ ...\ (x_n : [\![\tau_n]\!])\ \rightarrow$
   $\{[\![A]\!]_p\} try [\![s]\!]\ with\ Return\ v\ \rightarrow\ v\ end \{[\![B]\!]_a\ \texttt{and}\ [\![C]\!]_p\}$
2. Une interface de fonction Why :
   $[\![/*\ @\texttt{requires}\ A\ \texttt{assigns}\ B\ \texttt{ensures}\ C\ */\ f(\tau_1\ x_1,\ ...,\ \tau_n\ x_n)\{s\}]\!] \Rrightarrow$
   $\texttt{parameter}\ f\_parameter(x_1 : [\![\tau_1]\!])\ ...\ (x_n : [\![\tau_n]\!])\ \rightarrow$
   $\{[\![A]\!]_p\} [\![\tau]\!] \texttt{reads}\ reads\_effect(f)\ \texttt{writes}\ writes\_effect(f)\ \{[\![B]\!]_a\ \texttt{and}\ [\![C]\!]_p\}$

Le traitement des prédicats et des termes. L'interprétation des prédicats du C normalisé est définie comme suit. Le paramètre L en position de puissance indique le label courant de l'interprétation. Par défault, l'interprétation démarre avec le label $None$.

- $[\![\texttt{\textbackslash valid}(p)]\!]^L \Rrightarrow \texttt{valid}\ (alloc@L, [\![p]\!]^L)$
- $[\![\texttt{\textbackslash valid\_range}(p,a,b)]\!]^L \Rrightarrow \texttt{valid\_range}(alloc@L, [\![p]\!]^L, [\![a]\!]^L, [\![b]\!]^L)$
- $[\![A \&\& B]\!]^L \Rrightarrow [\![A]\!]^L\ \texttt{and}\ [\![B]\!]^L$
- $[\![A || B]\!]^L \Rrightarrow [\![A]\!]^L\ \texttt{or}\ [\![B]\!]^L$
- $[\![A => B]\!]^L \Rrightarrow [\![A]\!]^L \rightarrow [\![B]\!]^L$
- $[\![\texttt{\textbackslash forall}\ \tau\ i;\ P]\!]^L \Rrightarrow \texttt{forall}\ i : [\![\tau]\!].[\![P]\!]^L$
- $[\![\texttt{\textbackslash exists}\ \tau\ i;\ P]\!] \Rrightarrow \texttt{exists}\ i : [\![\tau]\!].[\![P]\!]^L$
- $[\![\texttt{\textbackslash old}(A)]\!]^L \Rrightarrow [\![A]\!]^{Init}$
- $[\![\texttt{\textbackslash at}(A, L')]\!]^L \Rrightarrow [\![A]\!]^{L'}$

et celle des termes est :

- $[\![x]\!]^{None} \Rrightarrow x$
- $[\![x]\!]^{Init} \Rrightarrow x@$
- $[\![x]\!]^L \Rrightarrow x@L$
- $[\![\texttt{\textbackslash base\_addr}(p)]\!]^L \Rrightarrow \texttt{base\_addr}(alloc@L, [\![p]\!]^L)$
- $[\![\texttt{\textbackslash null}]\!]^L \Rrightarrow \texttt{null}$
- $[\![\texttt{\textbackslash result}]\!]^{None} \Rrightarrow \texttt{result}$
- $[\![\texttt{\textbackslash old}(t)]\!]^L \Rrightarrow [\![t]\!]^{Init}$
- $[\![\texttt{\textbackslash at}(t, L')]\!]^L \Rrightarrow [\![t]\!]^{L'}$
- $[\![f(t_1, \ldots, t_n)]\!]^L \Rrightarrow f([\![t_1]\!]^L, \ldots, [\![t_n]\!]^L)$ où f est un symbole de fonction utilisateur ou bien une opération prédéfinie (+,-,...)

Le traitement des effets de bord. L'interprétation des effets de bord du C normalisé $[\![B]\!]_a$, où B est une liste d'effets, présenté sous la forme matricielle suivante (un nom de champ par ligne) :

$$\begin{pmatrix} x_1^1 \rightarrow f^1, & \ldots, & x_{n_1}^1 \rightarrow f^1 \\ \vdots & \vdots & \vdots \\ x_1^m \rightarrow f^m, & \ldots, & x_{n_m}^m \rightarrow f^m \end{pmatrix}$$

est définie comme ceci :

- $[\![B]\!]_a \Rrightarrow$
  $\texttt{not\_assigns}(alloc, f^1@, f^1, [\![x_1^1, .., x_{n_1}^1]\!]_{set})\ \texttt{and}\ ...$
  $\texttt{and not\_assigns}(alloc, f^m@, f^m, [\![x_1^m, .., x_{n_m}^m]\!]_{set})$

  L'interprétation des ensembles est définie par :
- $[\![x, l]\!]_{set} \Rrightarrow \texttt{pset\_union}([\![x]\!]_{set}, [\![l]\!]_{set})$
- $[\![x[..]]\!]_{set} \Rrightarrow$
  $\texttt{pset\_all}([\![x]\!]_{set})$
- $[\![x[a..b]]\!_{set}] \Rrightarrow$
  $\texttt{pset\_range}([\![x]\!]_{set}, a, b)$
- $[\![x[a..]]\!]_{set} \Rrightarrow$
  $\texttt{pset\_range\_right}([\![x]\!]_{set}, a)$
- $[\![x[..b]]\!]_{set} \Rrightarrow$
  $\texttt{pset\_range\_left}([\![x]\!]_{set}, b)$
- $[\![x[..] \rightarrow t]\!]_{set} \Rrightarrow$
  $\texttt{pset\_acc\_all}([\![x]\!]_{set}, t)$

- $[\![x[a..b] \rightarrow t]\!]_{set} \Rrightarrow$ `pset_acc_range`$([\![x]\!]_{set}, t, a, b)$
- $[\![x[a..] \rightarrow t]\!]_{set} \Rrightarrow$ `pset_acc_range_right`$([\![x]\!]_{set}, t, a)$
- $[\![x[..b] \rightarrow t]\!]_{set} \Rrightarrow$ `pset_acc_range_right`$([\![x]\!]_{set}, t, b)$
- $[\![x]\!]_{set} \Rrightarrow$ `pset_singleton`$([\![x]\!])$

Les fonctions logique ainsi introduite sont axiomatisées. L'ensemble d'axiomes correpondant est donné en annexe 9.1.4.

Interprétation des types L'interprétation des types est définie par :

- $[\![int]\!] \Rrightarrow integer$
- $[\![char]\!] \Rrightarrow integer$
- $[\![long]\!] \Rrightarrow integer$
- $[\![short]\!] \Rrightarrow integer$
- $[\![float]\!] \Rrightarrow real$
- $[\![double]\!] \Rrightarrow real$
- $[\![\tau$ `*`$]\!] \Rrightarrow pointer$
- $[\![\tau$ `[ ]`$]\!] \Rrightarrow pointer$

Remarque : Ici encore on utilise un modèle idéal des entiers, il n'y a pas de difficulté à l'étendre aux entiers machine.

4.5.3 Instruction `switch`

L'instruction `switch` peut être transformée en une série d'instructions `if` sous certaines conditions que nous verrons section 4.6.1.

La règle de traduction est :

$$[\![\text{switch}(e)s_1 \dots s_n]\!] \Rrightarrow \text{try let } i = eval(e) \text{ in } [s_1 \dots s_n]^i_\emptyset \text{ with } \texttt{Break} \Rightarrow \text{ void}$$

où `eval` est la fonction d'évaluation statique des expressions et $[..]^i_\gamma$ est défini par les cas qui suivent.

Si C est :

- soit `case i :`
- soit `default`.

Si les s_i sont des instructions différentes de

- `break;`
- `case i :`
- `default`

Enfin si Γ_{all} est l'ensemble des cas présents pour ce `switch`.

1. code mort

   $$[s_1\ s_2 \dots s_n]^i_\Gamma \rightsquigarrow [s_2 \dots s_n]^i_\Gamma$$

   Avec $n \geq 1$

2. case sans break

$$[\text{ case } x_1 : s_1 \dots s_n \text{ C } s_{n+1} \dots s_m]^i_\Gamma \rightsquigarrow$$
$$\texttt{if}(i = y_1 \; or \; \dots \; or \; i = y_n) \; \texttt{then}(s_1 \dots s_n); \; [\text{ case } x_2 : s_{n+1} s_m]^i_{\Gamma \cup x_1}$$

où $\{y_1, ...y_n\} = \Gamma \cup x_1$ et $n \geq 1$

3. dernier case sans break

$$[\text{ case } x_1 : s_1 \dots s_n]^i_\Gamma \rightsquigarrow \texttt{if}(i = y_1 \; or \; \dots \; or \; i = y_n) \; \texttt{then}(s_1 \dots s_n)$$

où $\{y_1, ...y_n\} = \Gamma \cup x_1$

4. case avec break

$$[\text{ case } x_1 : s_1 \dots s_k \text{ break ; } s_{k+1} \dots s_n \text{ C } x_2 : s_{n+1} \dots s_m]^i_\Gamma \rightsquigarrow$$
$$\texttt{if}(i = y_1 \; or \; \dots \; or \; i = y_n) \; \texttt{then}(s_1 \dots s_k) \; \texttt{else}([\text{ case } x_2 : s_{n+1} s_m]^i_\emptyset)$$

où $\{y_1, ...y_n\} = \Gamma \cup x_1$. De plus ici $s_{k+1} \dots s_n$ est du code mort.

5. dernier case avec break

$$[\text{ case } x_1 : s_1 \dots s_k \text{ break ; }]^i_\Gamma \rightsquigarrow \texttt{if}(i = y_1 \; or \; \dots \; or \; i = y_n) \; \texttt{then}(s_1 \dots s_k)$$

où $\{y_1, ...y_n\} = \Gamma \cup x_1$

6. default sans break

$$[\text{ default } : s_1 \dots s_n \text{ case } x_1 : s_{n+1} \dots s_m]^i_\Gamma \rightsquigarrow$$
$$\texttt{if}(i <> y_1 \; and \; \dots \; and \; i <> y_n) \; \texttt{then}(s_1 \dots s_n); [\text{ case } x_1 : s_{n+1} \dots s_m]^i]_\Gamma$$

où $\{y_1, \dots y_n\} = \Gamma_{all} - \Gamma$ c'est-à-dire qu'on execute $s_1 \dots s_n$ si i n'est pas dans Γ_{all} ou si i est dans Γ

7. default avec break

$$[\text{ default } : s_1 \dots s_k \text{ break ; } s_{k+1} \dots s_n \text{ case } x_1 : s_{n+1} \dots s_m]^i_\Gamma \rightsquigarrow$$
$$\texttt{if}(i <> y_1 \; and \; \dots \; and \; i <> y_n) \; \texttt{then}(s_1 \dots s_n) \texttt{else}([\text{ case } x_1 : s_{n+1} \dots s_m]^i]_\Gamma)$$

où $\{y_1, \dots y_n\} = \Gamma_{all} \neg \Gamma$. De plus, ici $s_{k+1} \dots s_n$ est du code mort.

8. default final

$$[\text{ default } : s_1 \dots s_n]^i_\Gamma \rightsquigarrow s_1 \dots s_n$$

Ici pas besoin de test, car on se retrouve dans le `else` de tout les `case` avec `break`.

9. case case

$$[\text{case } x_1 : \text{ case } x_2 : s_1 \dots s_n]^i_\Gamma \rightsquigarrow [\text{case } x_2 : s_1 \dots s_n]^i_{\Gamma \cup x_1}$$

Remarque : Notons que l'instruction `break` peut apparaître à l'intérieur d'un bloc présent pour un s_n donné. Ce cas sera traduit par la règle générale en `raise Break`.

Nous allons illustrer toutes ces règles sur l'exemple suivant :

```
int f(int i){
  int a;
  switch (i) {
    case 0 :
      a = 0;
      break;
    case 1 :
      a = 1;
      break;
    case 2 :
    case 3 :
      a = 3;
      break;
    case 4 :
      a = 4;
    case 5 :
      a = 5;
      break;
    default :
      a = 6;
      break;
    case 6 :
      a = 7;
  }
  return a;
}
```

On applique d'abord la règle (1) d'introduction afin de faire disparaitre le `switch`. Sur les motifs du `case 0` et du `case 1`, on peut ensuite appliquer la cinquième règle pour obtenir :

```
try
let i = eval(e)  in
  if(i = 0)
    then(a := 0)
    else(
      if(i = 1)
      then(a := 1)
      else( ... ))
with  Break
```

Deuxièmement le motif du `case 2` permet d'utiliser la dixième règle. L'instruction `case 3` qui suit permet une nouvelle fois d'utiliser la cinquième règle et ainsi obtenir :

```
  if(i = 2 or i = 3)
  then(a := 3)
  else( ... )
```

Ensuite, les motifs du `case 4` et du `case 5` seront traduits identiquement grâce aux règles 3 et 5 :

```
  if(i = 4)
  then(a := 4);
  if(i = 4 or i = 5)
  then(a := 5)
  else( ... )
```

Enfin, le motif du `default :` sera traduit par la huitième règle :

```
  if(i <> 0 and i <> 1 and i <> 2
     and i <> 3 and i <> 4 and i <> 5)
  then(a := 6)
  else( ... )
```

Le dernier `case`, quant à lui, sera traité par la règle 4 pour devenir :

```
  if(i = 6)
  then(a := 7)
```

Ainsi, le programme du début deviendra une fois traduit :

```
try
let i = eval(e) in
  if(i = 0)
    then(a := 0)
    else(
      if(i = 1)
      then(a := 1)
      else(
        if(i = 2 or i = 3)
        then(a := 3)
        else(
          if(i = 4)
          then(a := 4);
            if(i = 4 or i = 5)
            then(a := 5)
            else(
              if(i <> 0 and i <> 1 and i <> 2
                and i <> 3 and i <> 4 and i <> 5)
              then(a := 6)
              else(
                if(i = 6)
                then(a := 7))))))
with  Break => void
```

On obtient donc un code ne comprenant plus d'instruction `switch`. Cette transformation prend en compte la majorité des instructions `switch` utilisées en pratique 4.6.1.

4.6 Limitations

Le C normalisé permet de condenser le C dans un langage plus petit et donc plus facile à analyser. Cependant cette transformation peut entrainer des pertes. En effet, certains `switch`, comme nous le verrons dans la section 4.6.1, sont inexprimables en C normalisé. De plus, certaines transformations dues au C normalisé, peuvent faire perdre la sémantique du programme, comme nous le verrons dans la section 4.6.2

4.6.1 `switch` irrégulier

L'instruction `switch` est une instruction qui permet des constructions confuses. En effet, certaines constructions permettant l'imbrication de l'instruction `switch` avec d'autres instructions est parfaitement possible en C.

Exemple 4.6.1. *L'algorithme de Duff's device :*

```
switch (count % 8)  /* count > 0 assumed */
  {
    case 0 :        do {  *to = *from++;
    case 7 :              *to = *from++;
    case 6 :              *to = *from++;
    case 5 :              *to = *from++;
    case 4 :              *to = *from++;
    case 3 :              *to = *from++;
    case 2 :              *to = *from++;
    case 1 :              *to = *from++;
                    } while ((count -= 8) > 0);
  }
```

Cet algorithme permet de minimiser le nombre de tests pendant une copie. Ce mécanisme est du C correct. Malheureusement on s'aperçoit rapidement que la transformation en série de `if` sans duplication de code est impossible.

Ce genre de `switch` modifie le flot de contrôle de la même façon que le `goto`, vu à la section 4.2.1. De plus, ce type d'utilisation de l'instruction `switch` est très rare. Nous avons donc fait le choix de ne pas l'accepter dans `Caduceus`.

4.6.2 Les structures passées par valeur

En langage C, les structures peuvent être utilisées comme directement comme des valeurs : affectation d'une structure, passée par valeur à une fonction ou retournée par une fonction. Or, comme nous l'avons vu dans la section 4.4.2, la traduction de Caduceus vers Why les structures ne sont accédées que par l'intermédiaire de pointeur. Du coup la traduction ne préserve pas la sémantique du programme C.

Exemple 4.6.2. *Le code :*

```
struct s{
  int i;
};

struct s t[1] = {2};

void f(struct s y){
  y.i = 5;
}

int g(){
  f(t[0]);
  return t[0].i;
}
```

deviendra une fois normalisé :

```
struct s{
  int i;
};

struct s t[1] = {2};

void f(struct s *y){
  y->i = 5;
}

int g(){
  f(t+0);
  return (t+0)->i;
}
```

Or sur le programme original le résultat de la fonction `g` *est 2 alors que sur le programme normalisé son résultat est 5. Ceci est dû au fait que le passage par valeur des structures n'a pas été traité dans le C normalisé.*

Il a donc été choisi, dans un premier temps, de refuser à la normalisation les structures passées par valeur. Par contre, le fait de mettre directement une structure comme champ d'une autre structure est accepté. Même si une indirection est rajoutée à la normalisation, on détecte bien que les opérations sur cette structure ont une sémantique inchangée quand on rajoute cette indirection.

Exemple 4.6.3. `struct S{int i;};`

```
struct T{int j; struct S s;};
```

```
int f(struct S *s){
  return s->i;
}

int g(struct T *t){
  return f(&(t->s));
}
```

Comme ce code ne passe pas les structures par valeur et bien qu'une structure soit utilisée à plat dans une autre, ce code est accepté et normalisé en :

```
struct S{int i;};

struct T{int j; struct S s[1];};

int f(struct S *s){
  return s->i;
}

int g(struct T *t){
  return f(t->s);
}
```

4.7 Bilan

Le langage C normalisé est donc un sous-ensemble du langage du C, dans lequel les constructions suivantes ont été supprimées :

- l'opérateur adressage `&` : cet opérateur a complètement disparu du langage. Dans le C normalisé il est interdit de prendre l'adresse d'une variable.
- l'accesseur au tableau `t[i]` : cette notation a complètement disparu dans le langage C normalisé. Elle a été remplacée par la notation C équivalente : `*(t+i)`.
- l'accesseur au champ de structure `s.f` : cette notation a complètement disparu dans le langage C normalisé. Cela est dû au fait que les variables de type structure ont complètement disparu de ce langage, remplacées par des pointeurs sur structures. Du coup, la notation `s.f` devient `t->f`.
- l'opérateur de déréférencement `*x` : cet opérateur a complètement disparu du langage C normalisé. Cela est dû au fait que dans ce langage les types de pointeurs sont remplacés par des types de pointeurs sur structures. Ainsi l'opérateur `*x` devient $x \rightarrow \tau$ avec τ étant la traduction du type de x.
- par ailleur l'instruction `switch`, qui n'était pas supportée dans les version préliminaire de Caduceus, est maintenant traduite en Why en utilisant des séquences de `if`.

Nous obtenons ainsi un sous-ensemble langage de C tout en étant beaucoup plus simple et donc plus facile à analyser. Les limitations dues à ce langage n'ont pas d'impact sur l'analyse du code embarqué étudié dans le chapitre 7.

Chapitre 5

Analyse de séparation

Pour effectuer la vérification d'un programme avec pointeurs, il est connu que la détection d'aliasing de pointeur est essentielle. Une analyse de séparation est une technique automatique pour détecter si deux pointeurs donnés sont ou non aliasés. De nombreuses analyses de séparation ont été proposées.

La plus connue des approches pour l'analyse de séparation est la logique de séparation proposée par Reynolds [104]. Cependant, pour autant que nous sachions, les outils implémentant cette logique proposent souvent des analyses très complexes efficaces sur de petits codes difficiles mais souvent trop lourdes pour les études de cas concrètes telles que des codes industriels embarqués.

L'outil Caduceus, comme il est présenté dans le chapitre 2, a besoin d'annotations manuelles pour indiquer que chaque pointeur pointe sur une zone qui lui est propre. Or, sur un code de taille conséquente, cette annotation manuelle se révèle vite longue et fastidieuse. Pour aider l'utilisateur et rendre la tâche d'annotation plus aisée, nous proposons, dans la section 5.1, de générer automatiquement ces annotations. Une fois cette analyse utilisée en pratique, il en a découlé un nouveau problème. En effet, nous nous sommes aperçus que la quantité de prédicats générés à l'aide de cette méthode était trop importante pour que les fichiers de condition de vérification puissent être lus par les prouveurs. Pour résoudre ce problème nous avons donc, dans la section 5.2, utilisé une inférence de régions afin d'obtenir une partie de la séparation sans l'aide de prédicats. Cette inférence sera utilisée lors de la traduction vers Why.

5.1 La séparation à l'aide de prédicats et d'invariants

Dans cette section, nous verrons comment, à l'aide de prédicats, nous avons effectué une analyse de séparation. Celle-ci s'effectue sur les variables globales du code une fois normalisé comme vu chapitre 4. D'après la norme ANSI [76] tous les tableaux globaux sont séparés en mémoire.

Dans un premier temps, nous verrons les prédicats dépendants des déclarations de structures, puis nous verrons les invariants dépendants des déclarations de variables globales.

Pour ce faire nous avons créé deux fonctions. L'une, `IS`, qui prend un tableau en argu-

ment et renvoie le prédicat qui affirme que tous les tableaux déclarés à l'intérieur du type de ce tableau sont séparés. L'autre ES qui prend deux tableaux en argument et renvoie le prédicat qui affirme que tous les tableaux déclarés dans le type du premier argument sont différents de tous les tableaux déclarés dans le type du second argument. Ces prédicats seront utilisés sur des variables globales. On peut les utiliser, car on sait que tous les tableaux globaux sont séparés. Ces fonctions sont présentées à la figure 5.1. Le type pointeur est considéré comme un type de base parce qu'on ne peut rien affirmer sur la séparation des variables ou des champs de structures déclarés comme pointeur.

Exemple 5.1.1. *En effet sur ce code :*

```
struct list {
  int x;
  struct list *next;
};
```

on ne peut rien dire sur la séparation de p *et* $p \rightarrow next$.

5.1.1 Les prédicats dépendants des déclarations de structures

Dans un premier temps, nous regardons toutes les structures du programme une par une. Nous cherchons à maximiser la séparation à l'intérieur même de chaque structure. Pour chaque structure s de type τ nous créons le prédicat :

$$internal\_separation\_\tau(s) \ : \ IS(s) \tag{5.1}$$

Exemple 5.1.2. *Appliquons cette règle sur le code suivant :*

```
typedef struct{
  int i;
} s2;


typedef struct {
  int x[1];
  int y[3][5];
  s2 z[2];
} s1;
```

Cela nous conduit à générer le prédicat :

1. `internal_separation_s1(s1 *p)` *qui indique grâce à la règle 5.1 :*

$$\begin{array}{l}
\forall\, i : int,\ 0 \leq i < 2 \Rightarrow p \rightarrow x \neq p \rightarrow z[i] \wedge \\
\forall\, i : int,\ 0 \leq i < 3 \Rightarrow p \rightarrow x \neq p \rightarrow y[i] \wedge \\
\forall\, i : int, \forall\, j : int,\ 0 \leq i < 3 \Rightarrow 0 \leq j < 2 \Rightarrow p \rightarrow z[j] \neq p \rightarrow y[i] \wedge \\
\forall\, i\ j : int,\ i \neq j \Rightarrow 0 \leq i < 2 \Rightarrow 0 \leq j < 2 \Rightarrow p \rightarrow z[i] \neq p \rightarrow z[j] \wedge \\
\forall\, i\ j : int,\ i \neq j \Rightarrow 0 \leq i < 3 \Rightarrow 0 \leq j < 3 \Rightarrow p \rightarrow y[i] \neq p \rightarrow y[j]
\end{array}$$

$ES(\tau_1\ p_1[n_1], \tau_2\ p_2[n_2]) =$
Si $\tau_1 = \tau_2$ alors $\forall i_1, i_2 : int, 0 \leq i_1 < n_1 \Rightarrow 0 \leq i_2 < n_2 \Rightarrow p_1 + i_1 \neq p_2 + i_2$
else $True \wedge$
Soit $\tau_1 =$ type de base ou pointeur et $\tau_2 =$ type de base ou pointeur alors
$True$
Soit $\tau_1 =$ type de base ou pointeur et $\tau_2 =$ struct S_2 alors
$\forall i_2 : int, 0 \leq i_2 < n_2 \Rightarrow \bigwedge_{f_2 \in \text{champ}(S_2)} ES(p_1, (p_2 + i_2) \rightarrow f_2)$
Soit $\tau_1 =$ Struct S_1 et $\tau_2 =$ type de base alors
$\forall i_1 : int, 0 \leq i_1 < n_1 \Rightarrow \bigwedge_{f_1 \in \text{champ}(S_1)} ES((p_1 + i_1) \rightarrow f_1, p_2)$
Soit $\tau_1 =$ Struct S_1 et $\tau_2 =$ Struct S_2 alors
$\forall i_2 : int, 0 \leq i_2 < n_2 \Rightarrow \bigwedge_{f_2 \in \text{champ}(S_2)} ES(p_1, (p_2 + i_2) \rightarrow f_2) \wedge$
$\forall i_1 : int, 0 \leq i_1 < n_1 \Rightarrow \bigwedge_{f_1 \in \text{champ}(S_1)} ES((p_1 + i_1) \rightarrow f_1, p_2) \wedge$
$\forall i_1, i_2 : int, 0 \leq i_1 < n_1 \Rightarrow 0 \leq i_2 < n_2 \Rightarrow \bigwedge_{f_1 \in \text{champ}(S_1)} \bigwedge_{f_2 \in \text{champ de}(S_2)}$
$ES((p_1 + i_1) \rightarrow f_1, (p_2 + i_2) \rightarrow f_2)$

$IS(\tau\ p[n]) =$
Soit $\tau =$ type de base ou pointeur alors $True$
Soit $\tau =$ struct S alors
$(\forall i, j : int, 0 \leq j < n \Rightarrow 0 \leq i < n \Rightarrow$
$\bigwedge_{f \in \text{champ}(S)} ES(p + j, (p + i) \rightarrow f)) \wedge$
$(\forall i, j : int, 0 \leq j < n \Rightarrow 0 \leq i < n \Rightarrow \bigwedge_{f,g \in \text{champ}(S),\ g \neq f}$
$\Rightarrow ES((p + j) \rightarrow g, (p + i) \rightarrow f)) \wedge$
$(\forall i : int, 0 \leq i < n \Rightarrow \bigwedge_{f \in \text{champ}(S)} IS((p + i) \rightarrow f))$

FIG. 5.1 – Fonctions de génération de prédicat de séparation

Dans un second temps, nous créons des invariants pour séparer les structures différentes deux à deux. Pour deux structures différentes s_1 et s_2 de type respectif τ_1 et τ_2 nous créons le prédicat suivant :

$$separation\_\tau_1\_\tau_2(s_1, s_2) : ES(s_1, s_2) \tag{5.2}$$

Exemple 5.1.3. *Si l'on applique cette règle sur ce code :*

```
typedef struct {
  int a;
} s11;

typedef struct {
  s11 x[1];
} s1;

typedef struct {
  int b;
} s22;

typedef struct {
  s22 y[1];
} s2;

typedef struct {
  s22 z[2];
} s3;
```

on obtient les prédicats :

1. `separation_s1_s2(s1 *p,s2 *q)` *qui indique :*

$$p \rightarrow x[0] \neq q \rightarrow y[0]$$

2. `separation_s2_s3(s2 *p,s3 *q)` *qui indique :*

$$\forall\, i : int,\ 0 \leq i < 2 \Rightarrow q \rightarrow z[i] \neq p \rightarrow y[0]$$

3. `seperation_s1_s3(s1 *p,s3 *q)` *qui indique :*

$$\forall\, i : int,\ 0 \leq i < 2 \Rightarrow q \rightarrow z[i] \neq p \rightarrow x[0]$$

Ainsi, nous obtenons dans un premier temps un prédicat pour chaque structure du programme qui assure la séparation à l'intérieur de cette structure, puis dans un second temps, un prédicat pour chaque couple de structures qui assure la séparation entre ces deux structures.

5.1.2 Les invariants dépendants des déclarations de variables globales

Une fois ces prédicats sur les types créés, il ne reste plus qu'à s'intéresser aux variables globales du programme. Plus exactement aux variables globales de type tableau, car depuis le chapitre 4 les variables globales ne peuvent plus être que des variables globales de type de base, de type pointeur ou de type tableau. Pour chaque tableau global t défini par :

$$\tau\ t[n];$$

nous ajoutons un invariant :

$$separation\_t : \forall i : int, 0 \leq i < n \Rightarrow internal\_separation\_\tau(t+i)$$

Nous regardons ensuite les tableaux globaux deux par deux :

$$\tau_1\ t_1[i_1];$$
$$\tau_2\ t_2[i_2];$$

et pour chaque couple de tableaux globaux nous ajoutons un invariant :

$$separation\_t_1\_t_2 : \forall i, j : int, 0 \leq i < i_1 \Rightarrow 0 \leq j < i_2 \Rightarrow (t_1 + i \neq t_2 + j \wedge separation\_\tau_1\_\tau_2(t_1+i, t_2+j))$$

Exemple 5.1.4. *Appliquons ces règles sur le code suivant :*

```
typedef struct{
  int c;
} s2;

typedef struct{
  int a;
  s2 b;
} s1;

s2 t1[4];

s1 s[3];
```

Ce code va générer un invariant par couple de variables globales, soit deux invariants :

1. *le premier est associé à la variable s et indique que :*

$$separation\_s : \forall i : int, 0 \leq i < 3 \Rightarrow internal\_separation\_s1(t+i)$$

2. *le second est associé au couple (t_1, s) et indique que :*

$$separation\_t_1\_s : \forall i, j : int, 0 \leq i < 4 \Rightarrow 0 \leq j < 3 \Rightarrow (t_1 + i \neq s + j \wedge separation\_s1\_s2(t_1+i, s+j))$$

Comme on peut le voir sur cet exemple une fois les prédicats de types générés, il est très facile d'obtenir les invariants associés aux variables globales du programme.

Cependant, une fois cette méthode implémentée et utilisée, nous nous sommes aperçus que l'augmentation significative du nombre d'invariants dûs à cette méthode entraîne sur les codes de taille importante des fichiers de taille trop importante pour les prouveurs.

Nous avons donc combiné cette approche à une approche par typage, qui permet d'obtenir la séparation de pointeurs statiquement et donc de supprimer la majorité de ces annotations.

5.2 Analyse statique de séparation

Suite au problème posé par notre première méthode, nous avons cherché une approche qui permettrait de supprimer, ou tout au moins réduire, le nombre d'annotations générées automatiquement. Pour ce faire, nous proposons une nouvelle analyse de séparation, adaptée au modèle de Caduceus vu dans le chapitre 2. L'idée principale de cette analyse est que les régions de la mémoire qui sont garanties séparées sont modélisées par des composants distincts du modèle du tas mémoire. La méthode présentée ici est généralisable et applicable à tous les outils basés sur le modèle «component as array», que ce soit pour le C, pour le Java comme ESC/JAVA.

La section 5.2.1 présente notre nouvelle analyse de séparation, et définit le raffinement du modèle mémoire de Caduceus nécessaire. Le premier ingrédient est un système de type pour notre C normalisé vu au chapitre 4, où les types des pointeurs sont paramétrés par des régions. Une particularité importante est que pour un argument de fonction qui est du type pointeur, sa région est elle-même un paramètre de la fonction : ce qui conduit à un système de type *polymorphique* à la Milner [89]. Le résultat principal est le théorème 5.2.1 : Le système de type est *relativement sûr*, c'est-à-dire que si le programme est bien typé alors l'interprétation du même programme dans le modèle classique vu au chapitre 2 a la même sémantique.

Le deuxième ingrédient est un algorithme pour l'inférence de types (vu dans la section 5.2.4). Nous ne prouverons pas sa correction dans ce chapitre, mais nous obtenons la correction par l'intermédiaire de l'outil Why (le générateur de conditions de vérification de Caduceus). Le langage d'entrée de Why [56] est un langage de type polymorphe à la ML et nous utilisons ce système de type puissant pour obtenir la pleine correction de notre analyse de séparation : l'outil Why vérifie a posteriori, pour chaque programme C, que son interprétation dans notre modèle mémoire raffiné est bien typée.

Dans la section 5.2.5, nous montrerons une application de cette technique ainsi que des résultats expérimentaux obtenus avec Caduceus. Deux bénéfices sont présentés : premièrement, cela montre que notre analyse de séparation fournit des améliorations significatives pour des programmes de grandes tailles. Deuxièmement, plus surprenant, cela montre que cette analyse peut être un ingrédient très utile pour raisonner sur le comportement avancé des programmes. Enfin, nous comparons cette approche à d'autres approches existantes dans la section 5.3.

```
struct S { int i; };

/*@ requires \valid(x) && \valid(y)
  @ assigns x->i, y->i
  @ ensures x->i == 1 && y->i == 2
  @*/
void f(struct S *x, struct S *y) {
  x->i = 1; y->i = 2;
}

struct S t1[1], t2[1];

//@ ensures t1->i == 1 && t2->i == 2
void g(){
  f(t1,t2);
}
```

FIG. 5.2 – Un cas simple d'analyse de séparation

5.2.1 Principe de l'analyse de séparation

Nous illustrons sur un simple exemple l'intégration de cette analyse de séparation dans le modèle présenté au chapitre 2. Considérons le code source C de la figure 5.2, où nous utilisons la syntaxe du langage de spécification de Caduceus (voir la section 2.2). (L'annotation `\valid(x)` indique que `x` est correctement alloué dans la mémoire.)

La post-condition de la fonction `f` ne peut être établie : elle est en effet fausse en cas d'aliasing : les pointeurs `x` et `y` peuvent être égaux. Or pour l'appel de `f` dans la fonction `g` ils sont différents. Mais, comme nous utilisons un raisonnement modulaire (fonction par fonction, comme pour toutes les techniques basées sur le calcul de plus faible pré-condition), le code entier ne peut être prouvé correct. Une solution possible est d'ajouter à la pré-condition de `f` l'hypothèse `x != y`, mais notre but dans cette section est justement de supprimer ce genre d'annotations. L'interprétation Why du code de `f` est :

$$i := upd(i, x, 1); \qquad i := upd(i, y, 2);$$

ainsi l'établissement de la post-condition $acc(i, x) = 1$ remontant le calcul de plus faible pré-condition correspond à la formule :

$$acc(upd(upd(i, x, 1), y, 2), x) = 1$$

qui est une conséquence des axiomes 2.1 et 2.2 (page 33) si $x \neq y$.

Notre but est d'interpréter le code de `f` en utilisant deux variables distinctes pour représenter le même champ des structures de `x` et `y` et ainsi obtenir le code Why :

$$i_x := upd(i_x, x, 1); \qquad i_y := upd(i_y, y, 2);$$

```
struct S { int i; };

/*@ requires \valid(x) && \valid(y)
  @ assigns x->i, y->i
  @ ensures x->i == 1 && y->i == 2
  @*/
void f(struct S *x, struct S *y) {
  x->i = 1; y->i = 2;
}

struct T {
  struct S t1[2];
  struct S t2[2];
};

/*@ ensures s->t1->i == 1 && s->t2->i == 2 &&
  @         s->(t1+1)->i == 2 && s->(t2+1)->i == 1
  @*/
void h(struct T s) {
  f(s->t1,s->t2);
  f((s->t2)+1,(s->t1)+1);
}
```

FIG. 5.3 – Cas des régions paramétriques

où i_x et i_y sont deux *régions* distinctes pour le champ `i`. Avec cette interprétation, les post-conditions $acc(i_x, x) = 1$ et $acc(i_y, y) = 2$ découlent de l'axiome 2.1 sans besoin de rajouter $x \neq y$ en pré-condition.

Considérons maintenant le code de la figure 5.3, où la fonction f est appelée deux fois. Lors du premier appel, x pointe sur le tableau $s \rightarrow t_1$ et y pointe sur le tableau $s \rightarrow t_2$. Pour le deuxième c'est l'inverse. Pour allouer $s \rightarrow t_1$ et $s \rightarrow t_2$ dans des régions différentes, nous avons besoin de fabriquer des *paramètres* de régions pour `f`, et nous les appelons des régions *paramétriques*. Les interprétations complètes de `f` et `h` sont :

$$
\begin{array}{l}
\texttt{void}\ f(i_x, i_y, x, y)\ \{ \\
\quad i_x := upd(i_x, x, 1); \quad i_y := upd(i_y, y, 2); \\
\} \\
\texttt{void}\ h(t_1, t_2, i_1, i_2, s)\ \{ \\
\quad f(i_1, i_2, acc(t_1, s), acc(t_2, s)); \\
\quad f(i_2, i_1, shift(acc(t_2, s), 1), shift(acc(t_1, s), 1)); \\
\}
\end{array}
$$

et leurs post-conditions peuvent être établies par un raisonnement simple du premier ordre.

Notre but est d'intégrer la notion de séparation dans le modèle du code C, en attachant

les régions aux pointeurs et aux variables de mémoire. L'interprétation d'un accès à une mémoire $e \rightarrow f$ est maintenant $acc(f\_r, e)$ où r est la région de e. Voyons maintenant comment ajouter ces régions.

5.2.2 Système de types avec régions

Nous voyons les régions comme un système de type riche pour les pointeurs. Pour la simplicité, nous ne considérons que les pointeurs et le type de base `int`. Le type Why des expressions est alors donné par la grammaire :

(types)	τ	$::=$	`int`	
		$\mid$	$r\ pointer$	(pointeur de région r)
		$\mid$	$(\tau, r)memory$	(mémoire de valeurs de type τ dans la région r)
(régions)	r	$::=$	ρ	(région variable)
		$\mid$	R	(région constante)

Les régions variables sont nécessaires pour les régions passées aux fonctions en paramètre. En effet, les régions qui sont des paramètres de fonctions ont un type *polymorphe*. Notre système de type est juste un cas particulier de système de type à la Milner [89].

Si nous considérons la fonction f de l'exemple au-dessus, son profil est :

$$f(i_x : (int, \rho_1)memory, i_y : (int, \rho_2)memory, x : \rho_1\ pointer, y : \rho_2\ pointer)$$

Cette fonction est donc polymorphe pour ρ_1, ρ_2 : pour chaque appel de f ces régions peuvent être instanciées différemment.

Règles de typage

Nous exprimons maintenant la séparation en donnant les règles de typage, utilisant les types avec régions. L'environnement de type dépend de deux ensembles dénotés Γ et Δ. Γ est un environnement de type classique qui associe les identificateurs de variable à leur type : nous écrivons $x : t \in \Gamma$ quand Γ associe la variable x au type t. Δ est un *environnement de région* qui associe chaque paire (r, f), où r est une région et f un identificateur de champ, au type t de $p \rightarrow f$ quand p est un pointeur de région r. Nous l'écrivons comme ceci $(r, f) : t \in \Delta$.

Les règles de typage ci-dessous comportent un aspect inhabituel qui est que l'environnement Γ, Δ est fixé dès le départ y compris pour les variables locales (remarque : nous supposons, ici, que le typage habituel du C a déjà eu lieu y compris la résolution des portées, et donc une variable locale donnée apparaît de manière unique et non ambiguë dans l'environnement). Ces règles de typage expriment donc si oui ou non un programme est bien typé dans un tel environnement fixé.

Règle de typage pour les expressions

– Pour les constantes nous avons :

$$\frac{}{\Gamma, \Delta \vdash n : \texttt{int}} \qquad \frac{}{\Gamma, \Delta \vdash \textbf{null} : r\ \texttt{pointer}}$$

pour n'importe quelle région r (avec donc un polymorphisme de région pour **null**).

– Le type des variables suit l'environnement Γ et le type des accès aux champs suit l'environnement Δ :

$$\frac{}{\Gamma, \Delta \vdash x : t} \text{ si } x : t \in \Gamma \qquad \frac{\Gamma, \Delta \vdash l : r\ pointer}{\Gamma, \Delta \vdash l \to f : t} \text{ si } (r, f) : t \in \Delta$$

– L'appel de fonction utilise le typage polymorphe :

$$\frac{\Gamma, \Delta \vdash e_1 : t_1 \quad \cdots \quad \Gamma, \Delta \vdash e_n : t_n}{\Gamma, \Delta \vdash id(e_1, \ldots, e_n) : t}$$

si $id : \forall \rho_1 \ldots \rho_k, (\tau_1, \ldots, \tau_n) \to \tau \in \Gamma$ et il existe une substitution de régions σ sur $\rho_1 \ldots \rho_k$ telle que $t = \tau\sigma$ et pour chaque i, $t_i = \tau_i\ \sigma$. De plus, on exige que $\forall i\ j, i \neq j \Rightarrow \rho_i\sigma \neq \rho_j\sigma$. Cet dernière contrainte est essentielle : il est interdit sur une instance donnée de id d'associer la même région effective à deux régions paramètres distinctes.

– Pour l'expression conditionnelle :

$$\frac{\Gamma, \Delta \vdash e_1 : \tau \quad \Gamma, \Delta \vdash e_2 : \tau \quad \Gamma, \Delta \vdash e_3 : int}{\Gamma, \Delta \vdash e_3 ? e_1 : e_2 : \tau}$$

En particulier, si e_1 et e_2 sont des pointeurs, ils doivent être dans la même région.

– Les opérations sur un pointeur ne changent pas sa région, mais exigent que ces arguments soient dans la même région :

$$\frac{\Gamma, \Delta \vdash e_1 : r\ pointer \quad \Gamma, \Delta \vdash e_2 : r\ pointer}{\Gamma, \Delta \vdash e_1\ op\ e_2 : int}$$

où $op \in \{\ominus, ==, <=, =>, <, >, ! =\}$.

$$\frac{\Gamma, \Delta \vdash e_1 : r\ pointer}{\Gamma, \Delta \vdash e_1\ op : r\ pointer}$$

où $op \in \{++, --\}$.

$$\frac{\Gamma, \Delta \vdash e_1 : r\ pointer \quad \Gamma, \Delta \vdash e_2 : int}{\Gamma, \Delta \vdash e_1\ \oplus\ e_2 : r\ pointer}$$

Règles de typage pour les instructions

– Pour l'affectation de variable :

$$\frac{\Gamma, \Delta \vdash e : t}{\Gamma, \Delta \vdash v = e} \text{ si } v : t \in \Gamma$$

– Pour l'affectation d'un champ :

$$\frac{\Gamma, \Delta \vdash e_1 : r\ pointer \quad \Gamma, \Delta \vdash e_2 : t}{\Gamma, \Delta \vdash e_1 \to f = e_2} \text{ si } (r, f) : t \in \Delta$$

– Pour l'introduction de variable locale :

$$\frac{\Gamma, \Delta \vdash e : \tau'}{\Gamma, \Delta \vdash \tau\ v = e;} \text{ si } v : \tau' \in \Gamma$$

Remarque : τ n'est pas égal à τ' car τ est un type du C alors que τ' est un type de notre langage de type avec régions. Ils sont néanmoins forcément liés : si τ est int alors τ' aussi sinon τ est un type pointeur et τ' est nécessairement de la forme $r\ pointer$.

$$\frac{\Gamma, \Delta \vdash e_i : \tau' \quad \Gamma, \Delta \vdash n : int}{\Gamma, \Delta \vdash \tau\ v[n] = \{e_1, \ldots, e_n\};} \text{ si } \begin{cases} v : r\ pointer \in \Gamma \\ f = field(\tau) \\ (r, f) \in \Delta \end{cases}$$

où `field` est la fonction définie à la section 4.4.3.

– Pour le retour de fonction id :

$$\frac{\Gamma, \Delta \vdash e : \tau}{\Gamma, \Delta \vdash return\ e} \text{ si } \texttt{result}_{id} : \tau \in \Gamma$$

où $\texttt{result}_{id}$ est une pseudo variable représentant le retour de la fonction courante.

– Pour le branchement conditionnel :

$$\frac{\Gamma, \Delta \vdash i_1 \quad \Gamma, \Delta \vdash i_2 \quad \Gamma, \Delta \vdash e : int}{\Gamma, \Delta \vdash if(e) i_1\ else\ i_2}$$

– Pour la boucle `while` :

$$\frac{\Gamma, \Delta \vdash s \quad \Gamma, \Delta \vdash e_3 : int}{\Gamma, \Delta \vdash while(e_3) s}$$

– Pour le bloc :

$$\frac{\Gamma, \Delta \vdash s_1 \ldots \Gamma, \Delta \vdash s_n}{\Gamma, \Delta \vdash \{s_1 \ldots s_n\}}$$

– Pour le `break` :

$$\frac{}{\Gamma, \Delta \vdash break;}$$

Le typage des déclarations globales de variables s'effectue de manière identique à celui des variables locales. Le typage de chaque fonction du programme consiste à typer son corps.

5.2.3 Changement dans le schéma de traduction vers Why

L'analyse de région induit quelques changements dans les règles de traduction du C normalisé vers Why de la section 4.5.

Comme indiqué informellement précédemment, les traductions des accès et mises à jour des champs de structures tiennent maintenant compte des régions de la façon suivante :

– $[\![e \rightarrow f]\!] \Rrightarrow acces\ f^r\ [\![e]\!]$ si e a le type $r\ pointer$.

– $[\![e_1 \rightarrow f = e_2]\!] \Rrightarrow \texttt{let}\ v_1 = [\![e_1]\!]\ \texttt{in}\ \texttt{let}\ v_2 = [\![e_2]\!]\ \texttt{in}\ \texttt{update}\ f^r\ v_1\ v_2;\ v_2$ si e a le type $r\ pointer$.

L'autre changement est dans la traduction des fonctions et de leurs appels.

Si $id(x_1, \ldots, x_n)$ a, comme vu précédemment, pour type : $\forall \rho_1 \ldots \rho_k, (\tau_1, \ldots, \tau_n) \rightarrow \tau$ alors :

– l'interface Why engendrée :
$[\![/ * @ \texttt{requires}\ A\ \texttt{assigns}\ B\ \texttt{ensures}\ C\ * / id(\tau_1\ x_1,\ ...,\ \tau_n\ x_n)\{s\}]\!] \Rrightarrow$

$$\begin{array}{lllll}\texttt{parameter}\ id\_parameter(& f_1^{\rho_1} : (t_{f_1}, \rho_1)memory, & \ldots, & f_{m_1}^{\rho_1} : (t_{f_{m_1}}, \rho_1)memory, \\ & f_1^{\rho_2} : (t_{f_1}, \rho_2)memory, & \ldots, & f_{m_2}^{\rho_2} : (t_{f_{m_2}}, \rho_2)memory, \\ & \vdots & \vdots & \vdots \\ & f_1^{\rho_k} : (t_{f_1}, \rho_k)memory, & \ldots & f_{m_k}^{\rho_k} : (t_{f_{m_k}}, \rho_k)memory, \\ & x_1, & \ldots, & x_n) \rightarrow\end{array}$$

$\{[\![A]\!]_p\}[\![\tau]\!]\texttt{reads}\ reads\_effect(id)\ \texttt{writes}\ writes\_effect(id)\ \{[\![B]\!]_a\ \texttt{and}\ [\![C]\!]_p\}$
où t_{f_i} est l'interprétation Why du type du champ f_i.
La post-condition de la fonction id peut alors utiliser les $f_1^{\rho_i},\ \ldots,\ f_{m_i}^{\rho_i}$ et $x_1 \ldots x_n$.
L'implémentation Why est changée de manière analogue.

– chaque appel de id de la forme $id(e_1, \ldots, e_n)$ avec la substitution de régions σ sera interprété comme ceci :

$$\begin{array}{lllll}id(& f_1^{\rho_1\ \sigma}, & \ldots, & f_{m_1}^{\rho_1\ \sigma}, \\ & f_1^{\rho_2\ \sigma}, & \ldots, & f_{m_2}^{\rho_2\ \sigma}, \\ & \vdots & \vdots & \vdots \\ & f_1^{\rho_k\ \sigma}, & \ldots & f_{m_k}^{\rho_k\ \sigma}, \\ & e_1, & \ldots, & e_n)\end{array}$$

Remarque : La condition de typage des appels qui interdit d'identifier deux régions par σ nous assure, ici, que les paramètres de mémoires passés à id sont deux à deux distincts, donc le programme Why obtenu est bien typé du point de vue des alias 2.3.1.3.

Le résultat théorique que nous donnons maintenant est la propriété de correction de cette traduction pour les programmes bien typés : cette correction est relative dans le sens où nous montrons que l'interprétation des programmes avec séparation a la même sémantique que celle sans séparation.

Théorème 5.2.1 (Correction relative)**.** *Si un programme est bien typé dans un environnement donné Γ, Δ, alors l'interprétation logique avec des variables de mémoires de régions a la même sémantique que l'interprétation avec le modèle classique component-as-array vue au chapitre 2.*

Démonstration. La preuve s'effectue comme ceci :

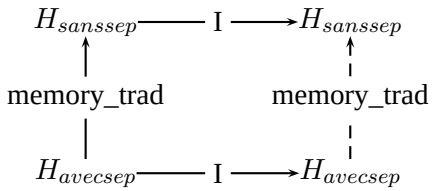
1. Définition d'un état mémoire avec ou sans l'analyse de séparation.
   Les états mémoire sans analyse de séparation H sont :
   – les valeurs associées aux variables $[v]_H : H(v)$.
   – les valeurs des différents champs f de structure $[p \rightarrow f]_H : H^f(p)$
   Les états mémoires avec analyse de séparation sont :

– les valeurs associées aux variables $[v]_H : H(v)$.
– les valeurs des différents champs `f` de structure $[p \rightarrow f]_H : H^{f^r}(p)$ avec $p : r\ pointer$.

De plus, pour passer d'un état mémoire avec séparation à un état mémoire sans séparation, il suffit de faire l'union des f^r. On appelle cette transformation `memory_trad`.

2. On prouve qu'il existe une bisimulation entre les étapes d'exécution pour des états mémoire sans séparation et des états mémoire avec séparation.

   La preuve suit la figure suivante :



   En effet, il suffit de montrer que la flèche en pointillée existe si le code est bien typé pour les régions.

   Pour ce faire, on prend une mémoire sans séparation H_s et une mémoire avec séparation H_a telles que $memory\_trad(H_a) = H_s$. On considère toutes les instructions possibles I qui agissent différemment suivant qu'on utilise la séparation ou pas : mise à jour de champs de structure et appel de fonction. Puis on applique sur ces deux mémoires les instructions qui peuvent les modifier et sur l'appel de fonction pour traiter le cas du polymorphisme. Ensuite on vérifie que la fonction de traduction permet toujours de passer de la mémoire avec séparation à la mémoire sans séparation.

   Dans ce qui suit, $H_s = H_s^{f^1} \cup \ldots \cup H_s^{f^k}$ et $H_a = H_a^{f^1} \cup \ldots \cup H_a^{f^k}$ où $H_a^{f^i} = H_a^{f^i_{r_1}} \cup \ldots \cup H_a^{f^i_{r_{n_i}}}$.

   Il y a donc deux cas à traiter :

   (a) Mise à jour d'un champ de structure :

$$
\begin{array}{l}
H_s \xrightarrow{e \rightarrow f = e';} H'_s \\
\quad \text{avec } H'^f_s = upd(H^f_s, [e]_{H_s}, [e']_{H_s}) \text{ et } H'^g_s = H^g_s si f \neq g \\
H_a \xrightarrow{e \rightarrow f = e';} H'_a \\
\quad \text{avec } e : r_0\ pointer,\ H'^{f_{r_0}}_a = upd(H^{f_{r_0}}_a, [e]_{H_a}, [e']_{H_a}), \\
\forall i, i \neq r_0 \Rightarrow H'^{f_i}_a = H^{f_i}_a \text{ et } H'^g_a = H^g_a si f \neq g
\end{array}
$$

   La partition de l'état mémoire suivant les noms de champs étant la même dans les deux cas, pour montrer $memory\_trad(H'_a) = H'_s$ il suffit de montrer que $memory\_trad(H'^f_a) = H'^f_s$.

$$
\begin{array}{l}
memory\_trad(H'^f_a) = \\
memory\_trad(H'^{f_{r_0}}_a \cup H'^{f_{r_1}}_a \cup \ldots \cup H'^{f_{r_n}}_a) = \\
memory\_trad(upd(H^{f_{r_0}}_a, [e]_{H_a}, [e']_{H_a}) \cup H^{f_{r_1}}_a \cup \ldots \cup H^{f_{r_n}}_a)
\end{array}
$$

Les $H_a^{f_{r_i}}$ sont une union disjointe dont le $memory\_trad$ vaut H_s^f, la mise à jour de $H_a^{f_{r_0}}$ se traduit dans H_s^f par $upd(H_s^f, [e]_{H_s}, [e']_{H_s})$. Par conséquent :

$$memory\_trad(H'^f_a) = upd(H_s^f, [e]_{H_s}, [e']_{H_s}) = H'^f_s$$

(b) appel de fonction :

$H_s \xrightarrow{id(e_1,\ldots,e_n)} H'_s$
avec id du type $(\tau_1, \ldots, \tau_n) \rightarrow \tau$et e_i de type τ_i
$H_a \xrightarrow{id(e_1,\ldots,e_n)} H'_a$
avec id du type $\forall \rho_1 \ldots \rho_k, (\tau_1, \ldots, \tau_n) \rightarrow \tau, e_i$ de type t_i
et $\exists \sigma$ une substitution des τ_i vers t_i sur $\rho_1 \ldots \rho_k$

Une petite difficulté, ici, est que l'exécution d'un appel en Why n'est pas «déterministe» dans le sens où la seule chose que l'on sait de H'_s (resp. H'_a) est que c'est un état vérifiant la post-condition P (resp. P') de la fonction id (resp. avec séparation) (plus précisement les instances respectives des post-conditions obtenues par substitution des paramètres formels par les arguments effectifs). On va donc montrer, ici, que $memory\_trad(H'_a)$ est l'un des états H'_s possibles, c'est-à-dire que P est vérifié dans cet état.

Premièrement, la correction du typage, qui impose que les régions ne sont pas identifiées par σ, nous assure que H'_a satisfait bien P' (cf 2.3.1.3). P' est de la forme $B' \wedge C'$ où B' est l'interprétation de la clause `assigns` et C' l'interprétation de la clause `ensures`. De même P est de la forme $B \wedge C$. C est directement obtenu à partir de C' en remplacant les f^r par f. On a donc directement que C est vrai dans $memory\_trad(H'_a)$. Pour B la situation est plus compliquée, car pour chaque champ f, B' contient une conjonction de la forme $\texttt{not\_assigns}(alloc, f^{r_1}@, f^{r_1}, S_1) \wedge \ldots \wedge \texttt{not\_assigns}(alloc, f^{r_k}@, f^{r_k}, S_k)$, et B contient une formule atomique de la forme $\texttt{not\_assigns}(alloc, f^r@, f^r, \texttt{pset\_union}(S_1, \ldots, S_k))$. Ces deux formules vraies toutes deux ou fausses toutes deux, grâce à l'axiomatisation utilisée pour `not_assigns`(voir annexe 9.1.4). On a donc bien aussi que B est vrai dans $memory\_trad(H'_a)$.

On a donc prouvé que quelle que soit l'instruction du code C, il est toujours possible de traduire le code Why avec séparation vers le code Why sans séparation.

□

Ce résultat nous permet de conclure que toute propriété que l'on prouve sur le programme Why obtenu avec analyse de séparation est également vraie sur le programme Why obtenu sans analyse de séparation.

5.2.4 Inférence des types avec régions

L'étape restante est maintenant de fournir un système d'inférence de types. Permettant de construire, quand cela est possible, un environnement Γ, Δ rendant un programme donné

bien typé. Notre système de type est un cas particulier du système de type polymorphique à la Milner, nous pouvons donc dériver une méthode d'inférence depuis un algorithme d'inférence de type connu comme dans l'algorithme W [44]. La seule caractéristique particulière est la manipulation de la partie de l'environnement Δ.

Dans un premier temps, nous fournissons une région constante nouvelle à chaque variable globale de type pointeur. Aux paramètres de fonction et aux variables locales de type pointeur nous assignons une région variable. Ceci produit un Γ initial. Nous donnons dans un même temps un Δ initial qui fait que tout est a priori séparé.

Dans un deuxième temps, les fonctions sont analysées, dans l'ordre du graphe d'appel, pour déterminer leur type polymorphe. Pour chaque fonction, le code est exploré et les règles de typage données ci-dessus mènent aux contraintes d'égalité entre les régions, ce qui nous permet une unification des régions. Chaque fois qu'une fonction est parcourue nous déterminons quelles variables de région restent non instanciées, ce qui nous permet de trouver le type des fonctions polymorphiques en quantifiant sur ces régions.

L'unification des régions est standard, excepté pour la manipulation du Δ : chaque fois que deux régions r_1 et r_2 sont désignées comme identiques, nous avons besoin d'effectuer une fusion sur Δ : pour chaque champ f tel que Δ associe (r_1, f) à t_1 et (r_2, f) à t_2, nous avons besoin de fusionner les couples et par conséquent d'unifier les types t_1 et t_2, ce qui peut récursivement mener à une unification avec d'autres régions. Durant cette phase d'unification, Γ est modifié par effet de bord.

À la fin du processus, nous finissons avec un Γ et un Δ pour lesquels le programme est bien typé. Comme dit dans l'introduction de ce chapitre, nous ne fournissons pas de preuve de ce fait. C'est un algorithme classique d'inférence, seul l'unification des régions est un ajout, et nous sommes convaincus qu'il est correct. De plus, en pratique, la correction est obtenue d'une manière fiable : nous générons un programme typé Why et si l'inférence de type est erronée alors le typage de Why échouera.

Voici l'algorithme qui permet l'inférence de type :

1. Au début, on a $\Gamma = \emptyset$ et $\Delta = \emptyset$.
2. Pour toutes les variables globales de type tableau ou pointeur, on rempli Γ en affectant à chaque variable une nouvelle région différente de toutes les autres.
3. Pour toutes les fonctions dans l'ordre du graphe d'appel :
   (a) On introduit des régions variables pour les paramètres, les variables globales et la variable de retour de type pointeurs ou tableaux de la fonction.
   (b) On spécialise le type de chaque fonction appelée. Puis on rajoute des contraintes d'égalité entre les paramètres et les arguments effectifs de la fonction appelée.
   (c) On parcourt le code et la spécification de la fonction pour typer chaque expression à l'aide des règles de typage vues dans la section 5.2.2 et ainsi obtenir les contraintes d'égalité. Ces dernières nous permettent d'unifier les régions et les types.
   (d) On généralise les régions variables non instanciées des paramètres et de la variable de retour de la fonction en cours de traitement.

Remarque : Dans le cas de fonctions mutuellement récursives, on analyse l'ensemble des fonctions qui s'appellent mutuellement avant de généraliser leurs types.

Exemple 5.2.2. *Utilisons cet algorithme sur ce code :*

```
struct S1{
  int i;
}

struct S2{
  struct S1 x[1];
}

struct S2 a[1];
struct S2 b[1];

struct S1* f(struct S2 *c; struct S2 *d){
  c->x->i = 0;
  d->x->i = 0;
  return c->x;
}

void g(){
  a->x=f(a,b);
}
```

1. *Au début,* $\Gamma = \emptyset$ *et* $\Delta = \emptyset$.
2. *Il y a deux variables globales de type tableau. Donc :* $\Gamma = \{a : R_1\ pointer,\ b : R_2\ pointer\}$ *et* $\Delta = \emptyset$.
3. *Il y a deux fonctions :*
   - *Pour la fonction* f *:*
     (a) *Il y a deux paramètres et la variable de retour. Donc* $\Gamma = \{a : R_1\ pointer,\ b : R_2\ pointer,\ c : \rho_1\ pointer,\ d : \rho_2\ pointer,\ result_f : \rho_3\ pointer\}$ *et* $\Delta = \emptyset$.

     (b) *Il n'y a pas de fonction appelée dans* f.

     (c) *Il y a six expressions à typer dans* f *:*
     - $c : \rho_1\ pointer$
     - $c \rightarrow x : \rho_4\ pointer$
     - $c \rightarrow x \rightarrow i : int$
     - $d : \rho_2\ pointer$
     - $d \rightarrow x : \rho_5\ pointer$
     - $d \rightarrow x \rightarrow i : int$

     l'instruction `return c->x` *amène une contrainte d'égalité :*
     - $\rho_3 = \rho_4$

Donc : $\Delta = \{(\rho_1, x) : \rho_3\ pointer,\ (\rho_2, x) : \rho_5\ pointer,\ (\rho_3, i) : int,\ (\rho_5, i) : int\}$.

(d) Les régions à généraliser sont ρ_1, ρ_2 *et* ρ_3 *donc pour la suite :* $\Gamma = \{\ f : \forall \rho_1,\ \rho_2,\ \rho_3,\ (\rho_1\ pointer,\ \rho_2\ pointer) \rightarrow \rho_3\ pointer\}$

– *Pour la fonction* g *:*

(a) Il n'y a pas de paramètre et de variable de retour. Donc : $\Gamma = \{\ a : R_1\ pointer,\ b : R_2\ pointer, f : \forall \rho_1,\ \rho_2,\ \rho_3,\ (\rho_1\ pointer,\ \rho_2\ pointer) \rightarrow \rho_3\ pointer\}$

(b) Il y a un appel de fonction dans g *: la fonction* f*. On spécialise donc le type de* f*, c'est-à-dire que l'on crée, pour chaque région généralisée, une région nouvelle et que l'on associe à cette nouvelle région les mêmes associations que dans* Δ*. Donc :*
– *Pour* ρ_1 *on crée* ρ_7
– *Pour* ρ_2 *on crée* ρ_8
– *Pour* ρ_3 *on crée* ρ_9

(c) Le typage du corps de g *amène les types suivants pour les expressions :*
– $a : R_1\ pointer$
– $a \rightarrow x : \rho_6\ pointer$
– $b : R_2\ pointer$
– $f(a, b) : \rho_9\ pointer$

avec les contraintes d'égalité suivante pour les paramètres et des arguments effectifs de f *ainsi que du résultat de* f *:*
– $\rho_7 = R_1$
– $\rho_8 = R_2$
– $\rho_9 = \rho_6$

(d) Il n'y a pas de régions à généraliser pour g*.*

On obtient ainsi par typage que a*,* b*,* $a \rightarrow x$ *et* $b \rightarrow x$ *sont dans des régions différentes.*

Remarque sur la complétude : l'algorithme W est connu pour trouver le type principal. En termes de régions, cela indique qu'il trouve la séparation avec le plus grand nombre possible de régions permis par les règles de typage données. On obtient ainsi la meilleure séparation possible à l'aide de ces règles de typage.

5.2.5 Expérimentations

Notre analyse de séparation est implémentée dans l'outil Caduceus, comme une option. Le choix de cette option demande à exécuter l'inférence des régions et alors la génération du modèle et l'interprétation en Why du code C est modifié convenablement. Nous montrons ici quelques expériences et applications.

5.2.5.1 Le benchmark de Caduceus

Le benchmark de Caduceus est un ensemble de petits programmes C qui sont utilisés comme tests de non régression. Ces exemples sont petits, et pour la plupart motivés par une raison autre que la séparation.

Sans l'analyse de séparation, pour l'ensemble complet d'exemples il y a un nombre total de 1324 conditions de vérification générées. Une fois passées dans le prouveur automatique Simplify [50] 1287 d'entre elles sont validées, soit 97,2%.

Avec l'analyse de séparation, il y a 1349 conditions de vérification générées. De premier abord, le nombre de ces conditions de vérification devrait être le même parce que l'analyse de séparation mène à une interprétation Why des programmes avec exactement la même structure. Cependant, le nombre de ces conditions de vérification est différent pour deux raisons :

1. Les conditions de vérification triviales (principalement des tautologies propositionnelles) sont silencieusement validées, et dans certains cas, avec l'analyse de séparation, quelques conditions de vérification sont des tautologies alors que sans cette analyse elles ne le sont pas. Ceci fait donc diminuer le nombre des conditions de vérification.
2. Comme indiqué dans la section 4.5.2 et comme vu dans la preuve du théorème de correction 5.2.1 l'interprétation de la clause **assigns** produit autant de propositions qu'il y a de variables de mémoires impliquées dans la clause. Or l'analyse de séparation augmente le nombre de variables de mémoire. Ceci peut donc augmenter le nombre de conditions de vérification, mais celles-ci s'en trouvent simplifiées.

Avec l'analyse de séparation, le prouveur automatique Simplify valide 1327 conditions de vérification, ce qui représente 98,3%.

Cela indique que l'analyse de séparation aide parfois sur ces petits exemples, mais, cela indique que l'analyse de séparation ne gêne pas la preuve des exemples où la séparation n'est pas le souci. Nous remarquons aussi que l'analyse de séparation est rapide (nous pensons qu'elle est linéaire en la taille du code), ainsi l'analyse de séparation pourra probablement être utilisée par défaut dans l'avenir.

5.2.5.2 Les régions et les annotations logiques

L'exemple suivant est inspiré d'un code Java proposé par P. Müller [92]. Ce code extrait l'ensemble des éléments positifs d'un tableau et les met dans un nouveau tableau.

```
int *m(int t[], int length) {
  int count = 0; int i; int *u;
  for (i=0; i < length; i++) if (t[i] > 0) count++;
  u = (int*)calloc(count,sizeof(int));
  count = 0;
  for (i=0; i < length; i++) if (t[i] > 0) u[count++] = t[i];
  return u;
}
```

La vérification que l'affectation `u[count++] = ...` est à l'intérieur des bornes du tableau est difficile : elle suppose un raisonnement "sémantique". Notons que la deuxième boucle compte exactement le même nombre d'éléments que la première, alors l'index `count` doit être plus petit que la valeur de `count` utilisée pour allouer le tableau `u`. Pour rendre ce raisonnement explicite, il est naturel d'annoter la boucle avec un invariant, le même pour les deux boucles :

```
/*@ invariant
  @    0 <= i && i <= length &&
  @    0 <= count && count <= i &&
  @    count == num_of_pos(0,i-1,t)
  @ variant length - i
  @*/
for (i=0; i < length; i++) { ...
```

où `num_of_pos` est un cas particulier du constructeur en JML `\num_of`.

`\num_of` est une construction JML [77] qui donne le nombre d'éléments satisfaisant le prédicat donné comme argument. Cependant, l'exemple d'origine de Müller était précisément un challenge pour les outils de vérification statique du fait qu'aucun de ces outils ne supportent le constructeur `\num_of`.

Pour le cas considéré, il est possible de donner un prédicat qui simule une utilisation de `\num_of`. Nous avons fait cela pour ce code de C, nous introduisons la fonction logique :

```
logic int num_of_pos(int i,int j,int a[]) reads a[..]
```

dont le but est de donner le nombre d'éléments positifs dans le tableau `a` entre les indices `i` et `j`, inclus. Cette fonction est formalisée par l'introduction des axiomes :

```
axiom num_of_pos_empty :
  \forall int i, int j, int a[];
      i > j => num_of_pos(i,j,a) == 0
axiom num_of_pos_true_case :
  \forall int i, int j, int k, int a[];
      i <= j && a[j] > 0 =>
        num_of_pos(i,j,a) == num_of_pos(i,j-1,a) + 1
axiom num_of_pos_false_case :
  \forall int i, int j, int k, int a[];
      i <= j &&! (a[j] > 0) =>
        num_of_pos(i,j,a) == num_of_pos(i,j-1,a)

axiom num_of_pos_strictly_increasing :
  \forall int i, int j, int k, int l, int a[];
      j < k && k <= l && a[k] > 0 =>
         num_of_pos(i,j,a) < num_of_pos(i,l,a)
```

Le point clé est maintenant que la vérification de la correction ne peut pas être faite facilement, car, dans la seconde boucle, bien que nous sachions que `count` est inférieur au

nombre d'éléments positifs de `t`, il faut un raisonnement permettant d'établir que ce nombre d'éléments n'a pas changé entre les deux boucles. Intuitivement, c'est évident car l'écriture dans le tableau `u` nouvellement alloué ne peut pas changer le contenu de `t`. Le problème est qu'avec une seule variable de tas `intM` dans le modèle mémoire pour représenter tous les tableaux d'entiers, ceci est loin d'être facile. En effet, la fonction logique `num_of_pos` n'est pas définie concrètement mais uniquement par l'axiomatisation ci-dessus, à partir de laquelle on peut envisager de montrer que `num_of_pos(i,j,a)` dépend uniquement des valeurs de `a[i..j]`, mais cette démonstration n'est certainement pas triviale (nécessite probablement une récurrence sur j).

Part contre, avec notre modèle mémoire fin découlant de l'analyse statique de séparation mémoire, les choses sont beaucoup plus simples. Il est détecté statiquement que `t` et `u` sont séparés, et qu'ils peuvent être modélisés dans deux variables de tas séparées `intM_t` et `intM_u`. De plus, la fonction logique `num_of_pos` devient paramétrique sur les variables mémoires à cause de l'argument `a` : la fonction logique `num_of_pos` obtenue en Why prend en entrée un paramètre qui est la mémoire de `a`.

```
logic num_of_pos:
  (int, 'rho) memory, int, int, 'rho pointer -> int
```

Et les axiomes sont proprement quantifiés sur le paramètre de mémoire supplémentaire. Par exemple :

```
axiom num_of_pos_strictly_increasing :
 forall intM_rho:(int, 'rho) memory.
  forall i:int.
   forall j:int.
    forall k:int.
     forall l:int.
      forall a:'rho pointer.
       j < k and k <= l
        and acc(intM_rho, shift(a, k)) > 0 ->
        num_of_pos(intM_rho, i, j, a) < num_of_pos(intM_rho, i, l, a)
```

Donc si on accède ou modifie un tableau `a`, la mémoire qui sera modifiée sera la mémoire passée en paramètre. De ce fait, si on appelle `num_of_pos` avec `t`, puis `u`, alors le paramètre mémoire de cette fonction sera respectivement `intM_t`, puis `intM_u`, ce qui nous permet de garder la séparation statique entre `t` et `u`. Il devient syntaxiquement vrai que `num_of_pos(0,j-1,t)` est le même dans les deux boucles de notre partie de code C.

Sur cet exemple, chaque condition de vérification est automatiquement vérifiée par le prouveur Simplify.

5.3 Comparaison avec d'autres travaux

Talpin et Jouvelot ont proposé en 1994 [110] un calcul pour l'analyse des effets des programmes basés sur un système de type polymorphe. Le principe est le même que le nôtre, mais leurs travaux sont limités aux références : aucun partage en profondeur dans les

structures de données n'est possible. Tofte et Talpin ont proposé de mettre en œuvre le calcul avec une pile de régions et sans ramasse-miettes [113]. Leur idée de base est d'associer une région pour chaque bloc et de recueillir la région sur le bloc de sortie.

Dans le contexte de l'analyse statique, l'analyse «points-to» est une technique avancée pour la recherche d'information sur les pointeurs. Cette analyse a débuté avec Andersen en 1994 [23] puis étendu en 1996 par Steensgaard [109] et en 2000 par Das [45]. Nous avons employé leurs idées de concevoir un type système pour l'analyse de séparation, mais il est évident que notre analyse est beaucoup moins précise que la leur.

Le système Cyclone [67] propose un nouveau langage de programmation analogue au C, mais dans lequel le programmeur peut spécifier les régions manuellement. De la même façon que notre approche, il alloue des régions paramétriques pour les appels de fonction. Notre approche est appliquée sur du code C réel, et de plus les régions y sont *automatiquement inférées* au lieu d'être données par l'utilisateur.

Comparée à la séparation logique, notre analyse de séparation est clairement moins précise. Il semble que la séparation logique est très puissante dans certain cas : elle permet par exemple d'indiquer qu'une liste chaînée ne peut pas être circulaire, ou qu'un graphe est un arbre, et peut raisonner sur ces informations. C'est quelque chose que notre analyse ne peut faire : dès que nous traversons une liste chaînée, une seule région est inférée pour toute la liste. Notre analyse est clairement plus adaptée aux programmes de grande taille, comme vu dans la section 5.2.5.2. La combinaison de la puissance de la logique de séparation et de notre analyse de séparation serait une prolongation future intéressante de notre travail.

En 2006, Nanevski, Morrisett et Birkedal [94] montrèrent l'importance des régions paramétriques dans le cadre de la *théorie des types de Hoare*. Nous avons obtenus le même résultat dans le contexte classique de la vérification déductive basée sur le calcul de plus faible pré-condition, qui est directement applicable à plusieurs outils pratiques existants.

Globalement parlant, nous pensons que notre approche nous a permis d'étudier des applications beaucoup plus complexes que les travaux précédents, comme le cas d'étude industriel du chapitre 7.

5.4 Bilan de ce chapitre

Nous avons donc proposé une analyse de séparation qui est potentiellement utilisable pour tous les outils de vérification déductive basés sur le calcul de plus faible pré-condition et le modèle mémoire component-as-array. Notre expérimentation sur les programmes en C est très positive, comme vu par les applications effectuées.

Il reste quelques défauts que nous avons prévu de traiter dans le futur. Premièrement, l'analyse de séparation doit être effectuée sur l'ensemble du code et n'est donc pas modulaire. Ce n'est pas un problème important puisque l'analyse de séparation est rapide, et parce que, après l'exécution de l'analyse de séparation, la suite de la vérification peut rester modulaire, fonction par fonction. Mais dans le cas où le code complet n'est pas fourni, comme par exemple dans le cas d'utilisation de bibliothèques, cela reste un problème. Nous voudrions ajouter un nouveau constructeur dans le langage de spécification pour permettre à l'utilisateur une spécification de la séparation : par exemple, pour une fonction d'une librai-

rie comme `memcpy`, l'utilisateur pourrait spécifier que la source et la cible sont séparées.

Notre analyse de séparation est adaptée pour être utilisée sur le modèle mémoire component-as-array. Dans ce modèle, un tableau C donné doit être entièrement dans la même région. Pourtant, par une analyse statique avancée, il est possible de découvrir qu'un tableau donné peut être découpé en plusieurs régions, par exemple dans le code suivant :

```
struct S { int x };
struct S t[10];
void f(S *p, int n) { p->x = n; }
void main() { f(t+0,2); f(t+1,3); }
```

nous n'avons pas gratuitement que $t[0] \rightarrow x == 2$.

De plus, notre analyse de séparation pèche encore sur un problème particulier :

```
int t[1];

void f(int *p, int *q);

void g() { f(t,t);}
```

Sur cet exemple, `f` a besoin en paramètres de deux pointeurs `p` et `q` et donc de deux mémoires associées respectivement `intM_p` et `intM_q`. Cependant, la fonction `g` l'appelle avec deux fois le même pointeur et donc deux fois la même mémoire ce qui est refusé par la règle de typage des applications 2.3.1.3. Pour resoudre ce problème une solution possible serait de dupliquer le code de `f` pour ce cas précis, en créant ainsi une fonction `f_same_pointer` et une fonction `f_two_pointeurs` afin d'avoir une seule mémoire en paramètre de la fonction dans le cas où l'appel de `f` se fait avec deux paramètres identiques.

Chapitre 6

La sélection d'hypothèses

Après avoir utilisé l'outil Caduceus sur plusieurs études de cas, il en est ressorti que plusieurs conditions de vérification ne semblent pas prouvables à l'aide des différents prouveurs à notre disposition. Après avoir procédé à une étude en Coq de ces conditions de vérification, nous nous sommes aperçus qu'elles étaient prouvables.

Nous avons donc cherché un moyen de valider automatiquement ces différentes conditions de vérification qui sont valides mais non-prouvables avec les outils automatiques dont nous disposons. Une condition de vérification est exprimée comme un but et un contexte. Le but encode l'exécution du programme, et peut-être vu comme des hypothèses et une propriété que le programme devrait satisfaire, à savoir la conclusion. Le contexte est une extension de la théorie de base (habituellement une combinaison de l'égalité avec des symboles non-interprétés de fonction et l'arithmétique linéaire) avec un grand ensemble d'axiomes. Le contexte décrit plusieurs aspects du langage tel que le modèle mémoire.

Les conditions de vérification peuvent donc contenir des axiomes inutiles (comme par exemple les axiomes sur les pointeurs pour un programme qui ne manipule pas de pointeur) et un grand nombre d'hypothèses inutiles introduite par le calcul de pré-condition. Un grand nombre d'axiomes et d'hypothèses inutiles agrandit l'espace de recherche des prouveurs SMT (Satisfiability Modulo Theories), et dégrade leurs performances.

Au lieu d'appeler aveuglément les prouveurs SMT sur l'ensemble des hypothèses et sur l'ensemble du contexte, nous présentons une méthode pour supprimer autant d'hypothèses que possible par une sélection d'hypothèses appropriée, ce qui nous permet de tailler de manière significative l'espace de recherche des prouveurs SMT. Ce travail a été réalisé en collaboration avec Jean-François Couchot, alors en post-doctorat dans l'équipe.

L'idée de la stratégie développée ici est naturelle : une hypothèse est pertinente si elle contient les variables et les prédicats nécessaires pour vérifier la conclusion. Pour trouver ces hypothèses pertinentes, nous analysons d'une part les dépendances entre les variables de la conclusion et les variables de chaque hypothèse, puis les prédicats de chaque hypothèse et les prédicats utilisables pour prouver la conclusion dans la théorie.

Nous présenterons d'abord dans la section 6.1 la forme des conditions de vérification. Puis dans la section 6.2 la façon dont sont sélectionnées les variables pertinentes. Suivie dans la section 6.3 de la façon dont sont sélectionnés les prédicats pertinents. Enfin dans la

$$
\begin{array}{lcl}
\Gamma \vdash H \Rightarrow C & \rightsquigarrow & \Gamma, H \vdash C \\
 & & \text{si } H \text{ n'est pas de la forme } H_1 \wedge H_2. \\
\Gamma \vdash H_1 \wedge H_2 \Rightarrow C & \rightsquigarrow & \Gamma \vdash H_1 \Rightarrow (H_2 \Rightarrow C) \\
\Gamma \vdash \forall x.C & \rightsquigarrow & \Gamma \vdash C
\end{array}
$$

FIG. 6.1 – La fonction introduction qui transforme les conditions de vérification

$$\Gamma \vdash C_1 \wedge C_2 \quad \rightsquigarrow \quad \Gamma \vdash C_1, \Gamma \vdash C_2$$

FIG. 6.2 – La fonction split qui découpe les conditions de vérification

section 6.4 la façon dont nous utilisons cette ensemble de variables et de prédicats obtenus pour sélectionner les hypothèse pertinentes.

6.1 La forme réduite des conditions de vérification

Cette section présente la forme réduite des conditions de vérification que nous considérons dans le reste de ce chapitre. Un premier point à noter est celui dû à l'application d'un calcul de plus faible pré-condition sur des programmes impératifs, la forme habituelle des conditions de vérification est :

$$\forall X.\, H_1 \Rightarrow H_2 \Rightarrow \ldots \Rightarrow H_n \Rightarrow C$$

où X est l'ensemble des variables dans la formule rassemblée pendant une étape du calcul de wp, H_i, $1 \leq i \leq n$ et C sont des formules de la logique du premier ordre.

6.1.1 La forme réduite «simple»

Tout d'abord il nous faut transformer ces conditions de vérification afin de rendre leur lecture plus simple. C'est le rôle de la fonction «introduction» 6.1. Elle introduit les hypothèses et supprime les «pour tout» ($\forall$) en admettant que les variables sont toutes quantifiées universellement.

6.1.2 La forme réduite avec « split »

Une fois cette introduction effectuée, nous utilisons une fonction qui réduit la taille, mais augmente le nombre des conditions de vérification en découpant les disjonctions dans la conclusion.

Ceci est effectué par la fonction split donnée dans la figure 6.2, où la dernière règle n'est considérée que si celles du dessus ne peuvent être appliquées. On en déduit que si l'ensemble des conditions de vérification obtenue par la fonction split est valide, alors la condition de vérification d'où cet ensemble est issu est valide. On remarque que la condition de vérification obtenue est sans conjonction dans la conclusion.

Remarque : Caduceus ne génère jamais de $\vee$ dans les hypothèses, de ce fait, à moins que l'utilisateur ne l'exprime explicitement dans des annotations, il n'y a pas de $\vee$ dans les conditions de vérification.

La forme réduite de nos conditions de vérification est donc :

$$H_1, .., H_n \vdash C \tag{6.1}$$

6.2 Sélection des variables pertinentes

Dans ce chapitre, nous proposons une méthode pour évaluer la pertinence des variables par rapport au but. Pour ce faire, nous donnons une graduation de la notion de pertinence, les variables sont plus ou moins pertinentes suivant cette distance.

Pour calculer cette distance, nous présentons dans la section 6.2.1 le graphe de dépendance entre les variables. Puis, dans la section 6.2.2,une méthode qui se base sur le graphe pour calculer la pertinence de chaque variable de la condition de vérification.

6.2.1 Dépendance des variables

Notre but ici est de construire un graphe qui permet un rapprochement entre les variables d'une condition de vérification. Nous construisons donc un graphe G_V de la façon suivante :

- Dans un premier temps nous aplatissons les hypothèses des conditions de vérification de cette façon : $H_1, \ldots l[g(x)] \ldots H_n \vdash C$ est remplacé par : $H_1, \ldots y = g(x), l[y], \ldots H_n \vdash C$ c'est-à-dire que si dans l'hypothèse H, un terme fonctionnel $f(t_1, ..., t_n)$ est présent comme le paramètre d'un prédicat ou d'une fonction, alors ce terme est remplacé par une variable fraiche x de la théorie. L'hypothèse $x = f(t_1, ..., t_n)$ est alors ajouté juste avant H. Cependant, l'aplatissement n'est pas appliqué quand le symbole de fonction est un paramètre d'un prédicat d'égalité.
- Les nœuds du graphe sont étiquetés par les variables de la condition de vérification et les variables résultant de l'aplatissement.
- On ajoute une arête entre $v_1 \leftrightarrow v_2$ deux nœuds v_1 et v_2 du graphe si les deux variables auxquelles correspondent les deux nœuds du graphe sont dans une même hypothèse de la condition de vérification.

Chaque condition de vérification a donc son graphe dont les sommets correspondent aux variables de la condition de vérification et les composantes fortement connexe correspondent aux hypothèses.

Remarque : l'aplatissement et le calcul des arêtes ne concernent pas la conclusion de la condition de vérification. En effet, notre but est de trouver une méthode pour sélectionner les hypothèses. Pour ce faire, ici, nous cherchons à savoir si les variables présentes dans le but sont aussi présentes dans les hypothèses. Donc dans tous les cas, lors de notre calcul de l'ensemble des variables pertinentes les variables du but seront dans l'ensemble.

Exemple 6.2.1. *Utilisons notre méthode de construction de graphe sur cette condition de*

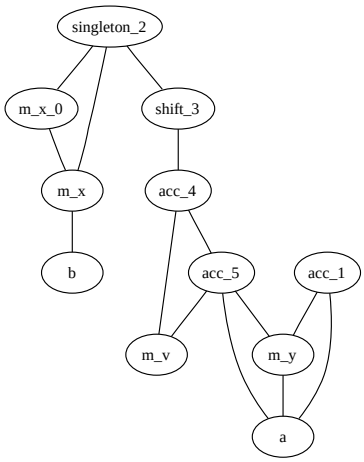


FIG. 6.3 – Graphe de dépendance des variables pour la condition de vérification (6.2)

vérification :

$$
\begin{array}{ll}
H1 & \mathsf{valid}(a) \\
H2 & \mathsf{valid}(b) \\
H3 & \mathsf{valid}(\mathsf{acc}(m_y, a)) \\
H4 & \mathsf{valid\_acc\_range}(m_v, 2) \\
H5 & \mathsf{separation1\_range}(m_v, 2) \\
H6 & \mathsf{not\_assigns}(m_x, m_x\_0, \mathsf{pset\_singleton}(\mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a)), 0))) \\
\hline
 & \mathsf{not\_assigns}\left(\begin{array}{c} m_x, \mathsf{upd}(m_x\_0, \mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a), 1)), \mathsf{acc}(m_x, b)), \\ \mathsf{pset\_range}(\mathsf{pset\_singleton}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a))), 0, 1)) \end{array}\right)
\end{array}
\tag{6.2}
$$

La figure 6.3 représente le graphe associé à la condition de vérification 6.2. Dans ce graphe, $singleton\_2$*,*$shift\_3$*,* acc_4 *et* acc_5 *sont des variables fraiches introduites par l'étape d'aplatissement de l'hypothèse :*

$$\mathsf{not\_assigns}(m_x, m_x\_0, \mathsf{singleton}(\mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a)), 0))).$$

Ce qui conduit à cinq hypothèses correspondant aux 5 composantes fortements connexes du graphe $\{m_x, m_x\_0, singleton\_2\}$*,* $\{singleton\_2, shift\_3\}$*,* $\{shift\_3, acc\_4\}$*,* $\{acc\_4, acc\_5, m_v\}$ *et* $\{acc_5, m_y, a\}$*.*

De même la variable acc_1 *correspond à l'hypothèse* $\mathsf{valid}(\mathsf{acc}(m_y, a))$*.*

6.2.2 Pertinence des variables

En nous basant sur le graphe vu à la section précédente, nous proposons une métrique de la distance entre une variable et la conclusion. Pour ce faire, nous appellerons $\mathcal{V}^n$ l'ensemble des variables d'une condition de vérification à distance n ou inférieur.

Nous commençons avec $\mathcal{V}^0$: les variables qui sont dans la conclusion de la condition de vérification. Puis on exécute un algorithme de parcours en largeur sur le graphe G_V calculé à la section 6.2.1 jusqu'à trouver le point fixe $\mathcal{V}^*$ c'est-à-dire l'ensemble des variables accessibles depuis $\mathcal{V}^0$.

Nous proposons de définir les $\mathcal{V}^n$ à l'aide de la séquence des $\left(\mathcal{V}^n\right)_{n\in\mathbb{N}}$ défini comme ceci :

$$\begin{cases} \mathcal{V}^{2n+1} &= \mathcal{V}^{2n} \cup \{v \mid \exists v_1, v_2 \,.\, v_1 \neq v_2 \wedge v_1 \in \mathcal{V}^{2n} \wedge v_2 \in \mathcal{V}^{2n} \wedge v \notin \mathcal{V}^{2n} \wedge \\ & \qquad\qquad (v \leftrightarrow v_1, v \leftrightarrow v_2 \text{ sont des arêtes de } G_V)\} \\ \mathcal{V}^{2n+2} &= \mathcal{V}^{2n+1} \cup \{v \mid \exists v' \,.\, v' \in \mathcal{V}^{2n} \wedge v \notin \mathcal{V}^{2n+1} \wedge (v \leftrightarrow v' \text{ est une arête de } G_V)\} \end{cases}$$

Cette séquence utilise une heuristique qui consiste à prendre d'abord les nœuds qui sont doublement liés avant les nœuds qui sont simplement liés. Cela permet une meilleure graduation dans le calcul d'accessibilité des variables. Sémantiquement, on privilégie les variables qui sont fortement connectées (c'est-à-dire liées par plus d'une hypothèse avec les variables déjà sélectionnées). Pour finir, on ajoute les variables inaccessibles à l'ensemble $\mathcal{V}^*$ pour des raisons de complétude et ainsi obtenir l'ensemble $\mathcal{V}^\infty$.

Exemple 6.2.2. *Si on reprend l'exemple vu à la section 6.2.1, l'ensemble des variables de la conclusion de la condition de vérification (6.2) est :*

$$\mathcal{V}^0 = \{m_x, m_x\_0, m_v, m_y, a, b\}$$

Grâce à cet ensemble et au graphe vu dans la figure 6.3 on obtient en appliquant l'algorithme vu plus haut la séquence :

$$\begin{array}{lcl} \mathcal{V}^0 &=& \{m_x, m_x\_0, m_v, m_y, a, b\} \\ \mathcal{V}^1 &=& \mathcal{V}^0 \cup \{acc\_1, acc\_5, singleton\_2\} \\ \mathcal{V}^2 &=& \mathcal{V}^1 \cup \{acc\_4\} \\ \mathcal{V}^3 &=& \mathcal{V}^2 \cup \{shift\_3\} \\ \mathcal{V}^* &=& \mathcal{V}^3 \\ \mathcal{V}^\infty &=& \mathcal{V}^* \end{array}$$

Ce qui nous permet de classer les variables par ordre d'importance pour la réussite de la preuve.

6.3 Sélection des prédicats pertinents

De même que pour les variables, dans cette section, nous donnons une méthode pour évaluer la pertinence des prédicats. Pour ce faire, nous présentons une métrique de la notion de pertinence.

Pour calculer cette distance, nous verrons, dans la section 6.3.1, le graphe de dépendance entre les prédicats, puis, dans la section 6.3.2, comment à l'aide de ce graphe nous calculons la pertinence de chaque prédicat de la condition de vérification.

6.3.1 Dépendance des prédicats

Nous proposons de créer un graphe orienté et pondéré représentant les relations entre les prédicats. Intuitivement, dans le graphe, chaque nœud représente un prédicat et s'il existe une arête du nœud p au nœud q alors il existe un axiome dans la théorie de Caduceus qui indique que p peut impliquer q. En suivant cette idée, nous construisons un graphe nommé G_P de la façon suivante :

En pratique, pour chaque prédicat présent dans la théorie, exception faite de l'égalité et l'inégalité, on crée un nœud dans le graphe étiqueté par ce prédicat. On note que si un prédicat r apparaît dans une occurrence négative (par exemple dans un axiome de la forme $\neg r$), elle est néanmoins représentée comme sommet marqué avec r. Voyons maintenant comment la dépendance entre les prédicats est représentée à l'aide des arêtes du graphe.

Dans un premier temps, chaque axiome de la théorie est écrit en CNF, mais d'une manière direct (par opposition à la CNF raffinée [96]) : les axiomes sont de petite taille et leur transformation sous la forme CNF n'entraînent pas d'explosion combinatoire. Ainsi, chaque clause résultante C (vu comme un ensemble de littéraux) peut être vu comme l'union de C_- et C_+ contenant respectivement les littéraux négatifs de C et les littéraux positifs de C. Intuitivement, chaque symbole de prédicat de C_- est un prémice de l'axiome et chaque symbole de prédicat de C_+ est une conséquence de l'axiome.

Une fois cette transformation effectuée. On prend chaque axiome de la théorie un par un puis pour chaque paire dans $C_- \times C_+$ ne contenant ni égalité ni inégalité une arête est ajoutée de la façon suivante :

Soit p un symbole de prédicat dans C_- et q un symbole de prédicat dans C_+ du même axiome C. Nous créons donc une arête de p à q pondérée avec un certain k. Plus cette pondération k est petite, plus la probabilité que p puisse établir q est élevée. En effet, les arêtes sont pondérées avec le nombre de symbole de prédicat présent dans l'axiome C minoré de 1 ($card(C) - 1$). Un axiome avec de nombreuse de prémices, un nombre important desymboles de prédicat p de C_-, de conséquences et de symboles de prédicat q de C_+, a moins de chance d'être utilisable dans un pas de déduction pour obtenir q que l'axiome $p \Rightarrow q$ (écrit en CNF $\{\neg p, q\}$). S'il existe 2 arêtes $p \xrightarrow{w_1} q$ et $p \xrightarrow{w_2} q$, nous laisserons dans le graphe seulement l'arête $p \xrightarrow{min(w_1,w_2)} q$. On note que même si les prédicats d'égalité et d'inégalité ne sont pas représentés comme nœud, ils sont cependant considérés dans le graphe de la dépendance puisqu'ils sont impliqués dans le calcul des pondérations.

Remarque : Contrairement au graphe vu dans la section 6.2.1 sur les variables le graphe obtenue ici est totalement indépendant de la condition de vérification que l'on cherche à prouver. Ce graphe ne dépend que du modèle de Caduceus et peut donc être construit statiquement.

Exemple 6.3.1. *Prenons par exemple l'axiome de la théorie de Caduceus :*

$$\mathsf{not\_assigns}(m_1, m_2, l) \Leftrightarrow \big(\forall p\,.\,\mathsf{valid}(p) \wedge \neg\mathsf{mem}(p, l) \Rightarrow \mathsf{acc}(m_1, p) = \mathsf{acc}(m_2, p)\big) \tag{6.3}$$

où p est un pointeur et $\mathsf{mem}(p, l)$ indique que p est un membre de la liste l.

Sa forme en CNF est donnée par :

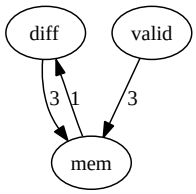


FIG. 6.4 – Graphe de dépendance de l'axiome(6.3)

$$\{\{\neg\mathsf{not\_assigns}(M_1, M_2, L), \neg\mathsf{valid}(P), \mathsf{mem}(P, L), \mathsf{acc}(M_1, P) = \mathsf{acc}(M_2, P)\},\\ \{\mathsf{valid}(p_0), \mathsf{not\_assigns}(M_1, M_2, L)\},\\ \{\neg\mathsf{mem}(p_0, L), \mathsf{not\_assigns}(M_1, M_2, L)\},\\ \{\mathsf{acc}(M_1, p_0) \neq \mathsf{acc}y(M_2, p_0), \mathsf{not\_assigns}(M_1, M_2, L)\}\} \tag{6.4}$$

où les variables en capitales sont universellement quantifiées et p_0 est une nouvelle variable constante résultante de la skolemization de p. La figure 6.4 représente le graphe de dépendance correspondant à cet axiome. C'est un extrait du graphe représentant le modèle mémoire de Caduceus.

6.3.2 Pertinence des prédicats

Dans cette partie, nous décrivons comment utiliser le graphe vu dans la section précédente afin de mesurer une distance entre un prédicat et la conclusion. Pour ce faire, nous appellerons $\mathfrak{L}[n]$ l'ensemble des variables d'une condition de vérification à distance n ou inférieur.

Comme vu dans la section 6.1, la conclusion peut être réduite à un littéral c sans perte de généralité. Dans ce qui suit, nous ne faisons pas la distinction entre un symbole de prédicat et le nœud correspondant du graphe G_P.

On dit qu'un prédicat p est pertinent pour prouver le prédicat q s'il existe un chemin dans le graphe G_P du nœud p au nœud q. Intuitivement, plus le poids du chemin est petit plus la probabilité pour p d'établir q est forte.

Pour trouver les prédicats pertinents pour une condition de vérification donnée, on calcule l'ensemble de prédécesseurs de n'importe quel prédicat d'occurrence positive apparaissant dans la conclusion de la condition de vérification et des successeurs de n'importe quel prédicat d'occurrence négative apparaissant dans la conclusion de la condition de vérification. Cet ensemble une fois obtenu est stocker dans la liste $\mathfrak{L}$ ordonné par la pondération

du chemin. On construit donc cette liste comme ceci :

$$\begin{aligned}
\mathcal{L}[0] &= \text{l'ensemble des prédicats de la conclusion de la condition de vérification} \\
\mathcal{L}[1] &= \mathcal{L}[0] \cup \{p \mid \exists p_1, p_1 \in \mathcal{L}[0] \wedge (p_1 \xrightarrow{1} p \text{ est une arête de } G_P)\} \\
\mathcal{L}[2] &= \mathcal{L}[1] \cup \{p \mid \exists p_1, p_1 \in \mathcal{L}[1] \wedge (p_1 \xrightarrow{1} p \text{ est une arête de } G_P)\} \\
&\quad \cup \{p \mid \exists p_1, p_1 \in \mathcal{L}[0] \wedge (p_1 \xrightarrow{2} p \text{ est une arête de } G_P)\} \\
&\ldots
\end{aligned}$$

Pour finir, la complétude de la sélection est obtenue en ajoutant les prédicats inaccessible à la fin de la liste : $\mathcal{L}[\infty]$.

Exemple 6.3.2. *Si on reprend l'exemple vu à la section 6.2.1 la condition de vérification est :*

$$\begin{array}{ll}
H1 & \mathsf{valid}(a) \\
H2 & \mathsf{valid}(b) \\
H3 & \mathsf{valid}(\mathsf{acc}(m_y, a)) \\
H4 & \mathsf{valid_acc_range}(m_v, 2) \\
H5 & \mathsf{separation1_range}(m_v, 2) \\
H6 & \mathsf{not_assigns}(m_x, m_x_0, \mathsf{pset_singleton}(\mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a)), 0)))
\end{array}$$

$$\mathsf{not\_assigns}\left(\begin{array}{r} m_x, \mathsf{upd}(m_x\_0, \mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a), 1)), \mathsf{acc}(m_x, b)), \\ \mathsf{pset\_range}(\mathsf{pset\_singleton}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a))), 0, 1)) \end{array}\right) \tag{6.5}$$

Les prédicats présents dans la conclusion de la condition de vérification sont :

$$\mathcal{L}[0] = \{\mathsf{not\_assigns}, \mathsf{upd}, \mathsf{shift}, \mathsf{acc}, \mathsf{range}, \mathsf{singleton}\}$$

Donc, si on utilise le graphe de la figure 6.4 vu à la section 6.3.1 qui correspond à un extrait du modèle de Caduceus. On obtient la liste suivante :

$$\begin{aligned}
&\mathcal{L}[0] = \{\mathsf{not_assigns}, \mathsf{upd}, \mathsf{shift}, \mathsf{acc}, \mathsf{range}, \mathsf{singleton}\} \\
&\mathcal{L}[1] = \mathcal{L}[0] \cup \{\mathsf{mem}\} \\
&\ldots
\end{aligned}$$

Remarque : le prédicat valid_acc_range n'apparaît ni dans $\mathcal{L}[0]$, ni dans $\mathcal{L}[1]$.

6.4 Sélection des hypothèses pertinentes

Dans la section 6.2 nous avons vu une distance sur les variables de chaque condition de vérification et dans la section 6.3 nous avons vu une distance sur les prédicats pour chaque condition de vérification. Nous allons maintenant utiliser ces distances pour sélectionner les hypothèses pertinentes.

Pour ce faire nous demandons dans un premier temps à l'utilisateur de nous donner deux nombres. Le premier correspondant à la distance qui sera appliquée sur les prédicats et le deuxième sera appliqué sur les variables. Pour chaque condition de vérification on utilise la méthode vue dans la section 6.3.2 avec premier nombre i donné par l'utilisateur pour

calculer l'ensemble des prédicats pertinents appelé $\mathcal{L}[i]$. Puis de la même façon on utilise la méthode vue dans la section 6.2.2 avec le deuxième nombre j donné par l'utilisateur pour obtenir l'ensemble $\mathcal{V}^j$ des variables pertinentes. Pour chaque condition de vérification on a donc un ensemble $\mathcal{L}[i]$ des prédicats pertinents et un ensemble $\mathcal{V}^j$ des variables pertinentes.

Pour une condition de vérification donnée, on cherche les hypothèses pertinentes. Pour chaque hypothèse de cette condition de vérification on regarde l'ensemble des prédicats P et l'ensemble des variables V présents dans cette hypothèse. Plusieurs critères peuvent être utilisés pour savoir si cette hypothèse est pertinente ou pas :

1. le plus faible : $V \cap \mathcal{V}^j \neq \emptyset$ ou $P \cap \mathcal{L}[0 \dots i] \neq \emptyset$: c'est-à-dire qu'il y a au moins une variable pertinente ou un prédicat pertinent dans l'hypothèse.
2. le critère intermédiaire : $card(V \cap \mathcal{V}^j)/card(\mathcal{V}^j)$ et $card(P \cap \mathcal{L}[0 \dots i])/card(\mathcal{L}[0 \dots i])$ sont supérieurs à un taux donné.
3. le plus fort : $V \subseteq \mathcal{V}^j$ et $P \subseteq \mathcal{L}[0 \dots i]$: c'est-à-dire que tous les prédicats et les variables de l'hypothèse sont pertinentes.

Nous avons donc expérimenté ces différents critères et il en est resorti qu'un critère trop faible sélectionne rapidement trop d'hypothèses. Nous avons donc opté pour le critère le plus fort et il sera le seul considéré dans la suite.

Exemple 6.4.1. *Si on reprend l'exemple vu à la section 6.2.1 la condition de vérification est :*

$$\frac{\begin{array}{ll} H1 & \mathsf{valid}(a) \\ H2 & \mathsf{valid}(b) \\ H3 & \mathsf{valid}(\mathsf{acc}(m_y, a)) \\ H4 & \mathsf{valid\_acc\_range}(m_v, 2) \\ H5 & \mathsf{separation1\_range}(m_v, 2) \\ H6 & \mathsf{not\_assigns}(m_x, m_x\_0, \mathsf{singleton}(\mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a)), 0))) \end{array}}{\mathsf{not\_assigns}\left(\begin{array}{c} m_x, \mathsf{upd}(m_x\_0, \mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a), 1)), \mathsf{acc}(m_x, b)), \\ \mathsf{range}(\mathsf{singleton}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a))), 0, 1)) \end{array}\right)} \tag{6.6}$$

Pour $i = 0$ on a :

$$\mathcal{L}[0] = \{\mathsf{not\_assigns}, \mathsf{upd}, \mathsf{shift}, \mathsf{acc}, \mathsf{range}, \mathsf{singleton}\}$$

Si on fait varier j on a :

$$\begin{array}{lcl} \mathcal{V}^0 & = & \{m_x, m_x\_0, m_v, m_y, a, b\}, \\ \mathcal{V}^1 & = & \mathcal{V}^0 \cup \{acc\_1, acc\_5, singleton\_2\}, \\ \mathcal{V}^2 & = & \mathcal{V}^1 \cup \{acc\_4\}, \\ \mathcal{V}^3 & = & \mathcal{V}^2 \cup \{shift\_3\}, \\ \mathcal{V}^* & = & \mathcal{V}^3 \textit{ and} \\ \mathcal{V}^\infty & = & \mathcal{V}^*. \end{array}$$

donc pour $i = 0$ et $0 \leq j \leq 2$ aucune hypothèse n'est sélectionner. Mais pour $i = 0$ et $j = 3$ la condition de vérification devient :

$$\begin{array}{l}\mathsf{not\_assigns}(m_x, m_x\_0, \mathsf{singleton}(\mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a)), 0))) \Rightarrow \\ \mathsf{not\_assigns}(m_x, \mathsf{upd}(m_x\_0, \mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a), 1)), \mathsf{acc}(m_x, b)), \\ \qquad \mathsf{range}(\mathsf{singleton}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a))), 0, 1))\end{array} \tag{6.7}$$

Ainsi la seule hypothèse sélectionnée est la seule nécessaire pour faire la preuve.

La méthode que nous venons de voir demande encore à l'utilisateur de donner deux nombres afin d'être utilisée. Cette méthode n'est donc pas entièrement automatique et demande donc à l'utilisateur une connaissance précise de la génération des conditions de vérification et du fonctionnement de la méthode de sélection. Pour résoudre ce problème et rendre la méthode entièrement automatique, nous avons mis au point un algorithme d'utilisation de la sélection d'hypothèses.

Pour une condition de vérification donnée on lance d'abord la sélection d'hypothèses avec $i = 0$ et $j = 0$. Ensuite, suivant la réponse du prouveur plusieurs choix s'offre à nous :

1. La formule est déclarée valide alors la procédure est terminée et la condition de vérification est prouvée.
2. La formule est déclarée non valide, peut être à cause d'un manque d'hypothèse, dans ce cas, trois cas sont possibles :
   (a) si $\mathcal{L}[i] = \mathcal{L}[\infty]$ et $\mathcal{V}^j = \mathcal{V}^\infty$ alors la procédure ce termine et la condition de vérification est non prouvée.
   (b) si $\mathcal{V}^j = \mathcal{V}^\infty$ alors on incrémente i, on remet j à 0 et on relance l'algorithme.
   (c) sinon on incrémente j et on relance l'algorithme.

Une évaluation de cette approche sera présentée à la section 7.3.

6.5 Bilan et Comparaison avec d'autres approches

Dans ce chapitre nous avons présenté une nouvelle stratégie pour sélectionner les hypothèses dans les formules issues de la vérification de programme. Pour ce faire, nous avons combiné deux analyses de dépendance séparée basées sur des graphes. De plus, nous avons donné une heuristique pour calculer le graphe avec une granularité suffisante.

Des stratégies pour simplifier le travail du prouveur ont été largement étudiées depuis l'existence des prouveurs automatiques [118], les propositions principales sont des systèmes déductifs plus efficaces [118, 117, 116].

Le travail présenté ici peut être comparé avec la stratégie de sélection sos (set of support) [118, 98].

Notre travail peut être aussi vu comme une conjecture automatique du sos initial guidé par la formule à prouver. Dans ce sens, il est proche de [86] où les clauses pertinentes initiales sont sélectionnées suivant un critère syntaxique, c'est-à-dire un taux de correspondance entre les symboles de n'importe quelle clause et les symboles des clauses de la conclusion. En considérant le filtrage syntaxique sur les clauses issues des axiomes et des

hypothèses, qui plus tard ne considérera pas la relation entre les hypothèses, formalisées par les axiomes de la théorie : il fournit une preuve vers l'avant limitée. Par contre, en soulignant le partage entre la dépendance statique et la dépendance dynamique, nous ne sommes pas si loin d'une preuve en arrière.

En se concentrant sur la partie prédicative de la condition de vérification, nos objectifs sont proches de ceux développés dans [65].

Le travail restant à faire est d'appliquer une stratégie pour choisir des axiomes appropriés de la théorie, car, la majeure partie du temps, chaque condition de vérification exige seulement une partie infime d'une plus grande théorie. Dans [102, 48], un exemple d'une telle stratégie est présenté, mais il a besoin d'une classification manuelle préliminaire des axiomes, laquelle, dans une optique de passage à l'échelle (en terme de volume de code), nécessiterait un effort d'automatisation.

Chapitre 7

Validation de l'approche déductive avec analyse statique sur un code industriel critique

Dans ce chapitre nous présentons les résultats des différentes améliorations effectuées sur Caduceus pour traiter les codes de grande taille. Pour cela Dassault Aviation nous a donné accès sur son site à un programme critique embarqué de 70 000 lignes de code compris dans 350 fonctions, sans fonction récursive et sans allocation dynamique. Ce code rassemble l'essentiel de ses données dans des structures globales. Comme on peut s'y attendre, sur un programme de cette taille, la moindre action effectuée manuellement peut s'avérer rédhibitoire. L'un des buts de cette thèse a donc été d'automatiser la majeure partie du travail de vérification de code.

Cette étude de cas nous a occupé tout au long de la thèse. Les travaux effectués sur l'étude peuvent se découper chronologiquement en quatre phases :

1. Nous avons commencé à tester l'outil Caduceus, entre janvier et avril 2005, sur l'étude de cas. Cela nous a amené à réparer plusieurs bugs de l'outil. Par exemple :
   – une capture de noms entre deux champs de deux structures différentes ayant le même nom,
   – un bug provoquant une trop grande taille des conditions de vérifications,
   – instruction `switch` non traitée.

   Après avoir corrigé tous ces bugs deux gros problèmes restaient à traiter. Premièrement, il nous fallait annoter manuellement l'ensemble des 350 fonctions de l'étude de cas. Pour résoudre ce problème, un outil, pour générer automatiquement les annotations des programmes, a été créé et est présenté section 7.1. Deuxièmement, il nous fallait des informations de séparation, ce qui a motivé l'approche par prédicats logiques, présentée section 5.1.
2. Dans une seconde phase, nous avons testé notre approche de séparation par prédicats sur un code significatif extrait de l'étude de cas analysé chez Dassault Aviation. Cet extrait est un code de 3000 lignes et 21 fonctions. Il en est ressorti que notre approche pour la séparation pouvait générer des conditions de vérification d'une taille

déraisonnable pour les prouveurs automatiques (allant jusqu'à un Go).

3. Nous avons alors mis en place l'analyse statique, vue à la section 5.2, qui a été implémentée dans Caduceus puis testée sur l'extrait de l'étude de cas. Cela a permis de passer d'un taux de 84% à 99,5% de conditions de vérification automatiquement prouvées par Simplify. Les 10 conditions de vérification restantes ont été prouvées en Coq. Ces expérimentations sont détaillées dans la section 7.2.
4. Entre juin 2006 et aout 2007, nous avons expérimenté notre approche sur l'étude de cas au complet. Plusieurs problèmes sont apparus :
   – certaines fonctions sont rejetées par Caduceus. Cela est dû à certains `cast` non ANSI présents dans l'étude de cas. Notons toutefois que la présence d'instructions non C ANSI dans le code étudié est parfaitement justifiée, notamment pour les opérations de conversion de type nécessaires lors de la manipulation de messages de bus de données.
   – pour certaines fonctions le calcul de plus faible pré-condition fait s'effondrer les performances de la machine par manque de mémoire.
   – certaines conditions de vérifications ne sont pas prouvées. Plusieurs raisons peuvent expliquer cela. Premièrement, Gena ne génère pas forcément toutes les annotations nécessaires et certaines d'entre elles doivent être corrigées manuellement. Deuxièmement, comme pour l'extrait de 3000 lignes, certaines conditions de vérification sont vraies mais non prouvées par un prouveur automatique.
5. Enfin, à partir d'août 2007, une option a été rajoutée à Why implémentant la sélection d'hypothèses, présentée au chapitre 6, pour aider les prouveurs automatiques. Ces expérimentations sont détaillées dans la section 7.3.

7.1 Génération automatique d'annotations

Comme nous l'avons vu au chapitre 2, Caduceus prend en entrée du code C annoté. Or l'étude de cas proposée est bien entendu un code C sans annotation. Le premier travail consiste donc à annoter ces 70 000 lignes de code. Pour ne pas effectuer cette tâche manuellement un outil de génération automatique d'annotation a été développé chez Dassault Aviation. Il existe des outils comme Houdini [95], Daikon [53] qui donnent des résultats précis sur les programmes plus court. Mais pour la classe de programmes traitée, c'est-à-dire des codes de grandes tailles mais syntaxiquement simples, un outil plus simple est plus efficace.

Dans cette section nous présentons cet outil de génération automatique d'annotation appelé "Gena". Cet outil a pour but de générer les annotations nécessaires à Caduceus pour prouver l'absence de menace (le périmètre de cette étude de cas s'est en effet limité à la recherche de menace à l'exécution). Cet outil exploite les règles de programmation de code critique définies par Dassault Aviation afin de générer le plus grand nombre d'annotations pour les codes critiques analysés chez Dassault Aviation. Il génère les pré-conditions (section 7.1.1) et les effets de bord (section 7.1.2) des fonctions ainsi que certains invariants simples, les variants et effets de bord des boucles (section 7.1.3).

7.1.1 Génération des pré-conditions des fonctions

Pour prouver l'absence de menace d'un code, Caduceus nécessite uniquement en pré-condition des fonctions la validité des pointeurs globaux ou passés en paramètre utilisés dans chaque fonction, et non alloués dans le corps de la fonction. De ce fait, pour chaque variable globale ou paramètre de fonction de type pointeur ou tableau auquel on accède dans le corps de la fonction :

- si cette variable v est déclarée de type $\tau *$ ou $\tau[]$, Gena génére la pré-condition `\valid`(v)
- si cette variable v est déclarée de type $\tau[n]$, Gena génére la pré-condition `\valid_range`$(v, 0, n-1)$.

De plus les pré-conditions sont remontées le long du graphe d'appel en substituant les noms des variables, c'est-à-dire que la pré-condition d'une fonction appelante est l'union du calcul de sa propre pré-condition avec toutes les pré-conditions de l'ensemble de ses fonctions appelées.

Exemple 7.1.1. *Pour ce code très simple :*

```
int *x;
char *c;

void f(char *d){
  *x = 5;
  *d = 'a';
}

void g(short *s){
  *s = 4;
  f(c);
}
```

le générateur automatique d'annotation Gena fournira comme pré-condition :

```
int *x;
char *c;

/*@ // requires de f
requires \valid(x)
  &&  \valid(d)
*/
void f(char* d){
  *x = 5;
  *d = 'a';
}

/*@ // requires de g
```

```
requires \valid(s)
  && \valid(x)
  && \valid(c)
*/
void g(short *s){
  *s = 4;
  f(c);
}
```

Cette méthode est très simple et possède, bien sûr, plusieurs limitations importantes :

1. Tableaux de taille variable passés en paramètre : la taille de ces tableaux peut varier suivant l'appel. Or, Caduceus nécessite de connaître cette taille afin d'assurer en pré-condition la validité du tableau sur l'ensemble de son "range".

   Exemple 7.1.2. *Le code :*

   ```
   void f (int t[];int taille){
     int i;

     for (i = 0; i < taille; i++){
       t[i] = 0;
     }
   }
   ```

   nécessite une annotation en pré-condition :

   ```
   /*@ // requires de f
   requires \valid_range(t,0,taille-1)
   */
   void f (int t[];int taille){
     int i;

     for (i = 0; i < taille; i++){
       t[i] = 0;
     }
   }
   ```

   Cependant, dans l'état actuel des développements, Gena est incapable d'associer la variable $taille$ à la taille réelle du tableau t.

   Nous avons dû donc corriger manuellement les annotations sur les quelques fonctions qui prenaient en paramètre un tableau de taille variable.

2. Tableaux multi-dimensionels : les tableaux à deux dimensions ou plus nécessitent en pré-condition une annotation qui permet d'affirmer que chaque case du tableau est un pointeur valide.

   Exemple 7.1.3. *Le code :*

```
int t[5][10];

void f(){
  int i;
  int j;

  for (i = 0; i < 5; i++){
    for (j = 0;j < 10; j++){
      t[i][j] = 0;
    }
  }
}
```

devra être annoté de la façon suivante :

```
int t[5][10];

/*@ // requires de f
requires \valid_range(t,0,4)
&& \forall int i; 0 <= i < 5 => \valid_range(t[i],0,9)
*/
void f(){
  int i;
  int j;

  for (i = 0; i < 5; i++){
    for (j = 0;j < 10; j++){
      t[i][j] = 0;
    }
  }
}
```

Or ici Gena ne fournira que la première pré-condition ($\texttt{\valid\_range}(t, 0, 4)$*).*

L'étude de cas analysée chez Dassault Aviation ne contient aucun tableau à deux dimensions ou plus. Gena était donc suffisant sur ce point.

3. Allocation dynamique : en cas d'allocation dynamique de mémoire d'un pointeur, la validité de celui-ci en pré-condition de la fonction peut s'avérer inutile, voire invérifiable.

   Exemple 7.1.4. *Sur le code :*

```
void f(int *x){
  x = (int *) malloc( sizeof(int));
  *x =0;
}
```

aucune annotation n'est nécessaire en pré-condition. En effet, le pointeur x n'est pas forcément valide au début de la fonction : c'est le $malloc$ qui le rend valide. Pour ce code Gena va générer l'annotation potentiellement invérifiable suivante :

```
/*@ // requires de f
requires \valid(x)
*/
void f(int *x){
  x = (int *) malloc( sizeof(int));
  *x =0;
}
```

Ce qui de toute évidence restreint beaucoup trop la spécification de la fonction f.

Cette limitation n'a pas posé de problème particulier car l'étude de cas analysée chez Dassault Aviation ne réalise pas d'allocation de mémoire dynamique.

7.1.2 Génération des effets de bord des fonctions

Pour prouver l'absence de menace avec Caduceus un listage des effets de bord de chaque fonction est nécessaire. Comme pour les pré-conditions, Gena va donc rechercher dans la fonction courante analysée et dans chaque fonction appelée la liste des pointeurs globaux ou passés en paramètre qui ont été modifiés (c'est-à-dire placés en valeur gauche).

Exemple 7.1.5. *Sur ce code très simple :*

```
int *x;
int t[5];

void f(int *a){
  *x = 0;
  *a = 1;
  t[0] = 2;
}

void g(int *b){
  int i;

  f(b);
  for (i =0; i< 5; i++){
    t[i] = 2;
  }
}
```

les annotations produites par Gena seront les suivantes :

```
int *x;
```

```
int t[5];

/*@ // assigns de f
assigns *x, *a, t[..]
*/
void f(int *a){
  *x = 0;
  *a = 1;
  t[0] = 2;
}

/*@ assigns de g
assigns *x,
  *b,
  t[..]
*/
void g(int *b){
  int i;

  f(b);
  for (i =0; i< 5; i++){
    t[i] = 2;
  }
}
```

Cette méthode est très simple, mais elle possède une limitation importante, l'analyse d'alias n'est pas effectuée.

Exemple 7.1.6. *Dans le code suivant :*

```
void f(int *x){
  int *y = x;
  *y = 5;
}
```

la liste des effets de bord de f est $\{x\}$ mais sur cette exemple Gena ne lui donnera aucun effet de bord.

Mais du fait du nombre restreint de variables locales sur le code analysé chez Dassault Aviation, ce problème ne s'est pas présenté.

7.1.3 Le traitement des boucles

Pour prouver l'absence de menace avec Caduceus il est nécessaire d'annoter les boucles. Les boucles qui seront traitées ici ne représentent qu'une sous-catégorie des boucles possibles.

En effet Gena ne traitera que les boucles de la forme :

```
for (i=0; i<n; i++){
....
}
```

ou de la forme :

```
while(i<n){
...
}
```

où i est une variable et n est une expression arbitraire. De plus le corps de la boucle devra être sans effet de bord sur n pour les boucles for et $while$, et sur i pour la boucle for.

Comme vu au chapitre 2, les boucles ont besoin de trois annotations différentes : l'invariant de boucle qui permet d'abstraire la boucle, le variant de boucle qui permet de prouver la terminaison de la boucle et enfin la liste des effets de bord de la boucle. Les invariants générés se limitent à spécifier que l'indice de boucle i est bien compris entre 0 et $n-1$. Les variants générés stipulent simplement la différence entre le nombre d'exécution de la boucle n et l'indice i (laquelle est bien décroissante, ce qui assure la terminaison de la boucle). Enfin les effets de bord sont calculés comme dans la section 7.1.2 sauf que les variables locales ne sont plus ignorées.

Exemple 7.1.7. *Sur les exemples simples que traite Gena la génération des annotations s'effectue comme suit :*

```
void f(int t[],int n){
  int i;

  for (i=0;i<n;i++){
    t[i] = 0;
  }
}
```

Cet exemple sera annoté par Gena par :

```
void f(int t[],int n){
  int i;

  /*@
    @ invariant 0 <= i <= n
    @ loop_assigns
    @   t[..],i
    @ variant n-i
  */
  for (i=0;i<n;i++){
    t[i] = 0;
  }
}
```

Gena génère des annotations suffisantes uniquement sur les boucles simples, et dans le seul objectif de s'assurer de l'absence de menaces à l'exécution. Mais pour le code analysé chez Dassault Aviation, ce genre d'annotation s'est avéré suffisant.

7.2 Intérêt de l'analyse de séparation

Le chapitre 5 traite de l'analyse de séparation c'est-à-dire de savoir si deux pointeurs sont égaux ou non. C'est en effet un problème important parce qu'en moyenne les annotations qui traitent de la séparation représentent 90% des annotations d'un code. Ce chapitre se découpe en deux parties : la première traite d'une analyse faite sur les variables globales à l'aide d'ajout automatique de prédicat, la deuxième quant à elle traite tous les pointeurs du programme à l'aide d'une analyse de typage statique. Dans cette section nous verrons donc les progrès apportés par ces analyses sur le code analysé chez Dassault Aviation.

7.2.1 Intérêt de l'analyse de séparation par prédicat

Le code analysé chez Dassault Aviation comporte, pour des raisons d'efficacité (optimisation de la taille de la pile, ...), de nombreuses variables globales de type structure. De ce fait, il est nécessaire d'annoter ce code de façon à ce que chacune de ces variables soient séparées l'une de l'autre. Au vu du nombre de variables globales présentes dans ce code, l'annotation manuelle stipulant que toutes les variables globales sont séparées deux à deux a vite été abandonnée, compte tenu des aspects combinatoires. À la place, l'analyse automatique de séparation de la section 5.2 a été mise en œuvre. Cette analyse considère les variables globales deux à deux et produit un invariant qui assure que ces deux variables sont séparées.

Mais nous avions alors sous-estimé les aspects de passage à l'échelle. En effet, suite à l'implémentation de cette analyse vue dans la section 5.1, les fonctions qui manipulent de nombreuses structures ont généré des conditions de vérification allant jusqu'à 1 Go. Ce type de conditions de vérification ne peut être traité par les prouveurs disponibles, et reste donc invérifiable automatiquement.

Nous nous sommes donc confrontés à un problème de passage à l'échelle plus important qu'il nous semblait de prime abord. En effet, nous souhaitions résoudre ce problème de passage à l'échelle en générant automatiquement les annotations nécessaires, mais cela a conduit à un problème différent mais tout aussi contraignant : la taille des conditions de vérification. Il nous fallait donc une méthode qui permette de résoudre le problème de la séparation de façon statique c'est-à-dire sans générer de condition de vérification.

7.2.2 Intérêt de l'analyse de séparation par typage

Comme vu précédemment le code analysé chez Dassault Aviation comporte de nombreuses variables globales de type structure. L'annotation manuelle - voire même automatique - du code est de ce fait impossible dans des conditions de temps de calcul et de mémoire raisonnables. Pour résoudre ce problème nous avons mis au point une méthode

Fonction	nombre de VC	
	séparation	sans
1	0/0	0/0
2	3/3	3/3
3	51/51	30/30
4	66/66	52/52
5	0/0	0/0
6	0/0	0/0
7	0/0	0/0
8	37/37	27/27
9	7/7	7/7
10	0/0	0/0
11	0/0	0/0
12	0/0	0/0
13	98/98	86/86
14	64/64	**28/32**
15	542/542	**217/282**
16	698/698	**323/398**
17	59/59	**40/49**
18	3/3	3/3
19	**69/79**	**46/73**
20	95/95	**44/47**
21	180/180	**59/62**
total	1972/1982	965/1151
pourcentage	99,5%	83,8%

FIG. 7.1 – résultats de l'analyse de séparation sur l'extrait de code

de typage présentée dans la section 5.2 qui permet de séparer les pointeurs sans ajouter d'annotation ou générer de condition de vérification.

Voici les résultats obtenus avec et sans l'analyse de séparation sur l'extrait de l'étude de cas analysé chez Dassault Aviation. Ce code ne comprend que 21 fonctions et il est donc plus facile de tester notre approche dessus que sur l'étude de cas au complet. Le tableau 7.1 présente pour chaque fonction le nombre de conditions de vérification prouvées automatiquement par rapport au total de celles-ci, ceci avec ou sans l'option de séparation. Les chiffres en gras représentent les fonctions qui ne sont pas totalement prouvé automatiquement. De plus, il faut noter que le temps laissé au prouveur pour chaque condition de vérification est de 600 secondes. Pour information le nombre de régions calculées pour les données globales du programme est de 376. De plus, le nombre de paramètres de région des fonctions Why produites (qui est égal au nombre de variables de régions dans leurs types) est de 242. Les résultats obtenus sur l'extrait de l'étude de cas sont encourageants, les dix conditions de vérification restantes ont été prouvées en Coq.

La version de Caduceus avec séparation a ensuite été expérimentée sur l'étude de cas

complète. Cette approche bien qu'imparfaite sur de nombreux points a fourni plusieurs résultats satisfaisants :

- Tout d'abord, il est maintenant possible de passer ce code dans l'outil Caduceus sans plus d'annotations que celles fournies par Gena vu à la section 7.1, et d'obtenir des conditions de vérification lisibles par un prouveur. Ce point peut sembler anodin mais représente déjà une avancée importante dans le traitement de ce code. En effet jusqu'à présent il nous avait été impossible de faire passer dans l'outil Caduceus sur 320 fonctions des 352 du code. les 32 fonctions restantes ne passent pas à cause de `cast` de pointeurs. Il faut noter aussi que 38 fonctions des 320 qui sont traités par Caduceus ne passent pas le calcul de plus faible pré-condition de Why (dans un temps et un espace mémoire acceptable).
- De plus, une amélioration des résultats s'est faite ressentir avec l'utilisation de cette analyse. En effet, plusieurs fonctions dont les conditions de vérification n'étaient pas validées par le prouveur Simplify, sont maintenant prouvées automatiquement.

Sans cette analyse, il est nécessaire d'annoter manuellement tous les couples de pointeurs pour savoir s'ils sont oui ou non aliasés, ce qui provoque de manière significative une augmentation de la taille des conditions de vérification (seules 20% des conditions de vérification sont alors validées). Une fois les améliorations mises en œuvre, les résultats de la preuve sur le code de 70 000 lignes analysé chez Dassault Aviation avec le prouveur Simplify sont les suivantes :

Valid	:	115774	$96,09\%$
Invalid	:	5	$0,00\%$
Unknown	:	37	$0,03\%$
Timeout	:	4672	$3,88\%$
Total	:	120488	100%

Il faut noter cependant que certaines fonctions du code ne sont pas intégrées dans ce résultat et ce pour deux raisons. D'une part, certaines fonctions (casting de messages de bus de données) du code analysé chez Dassault Aviation ne respectent pas la norme ANSI (principalement pour un problème de "padding" entre les champs de structures) et sont donc rejetées par Caduceus (qui ne traite que des codes strictement ANSI). Pour résoudre ce problème de "padding" il est nécessaire de posséder un modèle mémoire bas niveau (dépendant de l'architecture de la machine et du compilateur). Certains outils possèdent ce type de modèle comme : VCC [115] qui utilise différentes axiomatisations pour définir l'architecture, Frama-C [11] et Caveat [38] qui exploitent tous les deux des données fournies par l'utilisateur sur la machine et le compilateur, ou le "Verifying Compiler" [81] basé sur le code assembleur du PowerPC. D'autre part, certaines fonctions de ce code possèdent de nombreux branchements, c'est-à-dire de `if-else` imbriqués, ce qui nécessite parfois un espace mémoire supérieur aux capacités des machines de développement standard (le calcul de la plus faible pré-condition repose sur la "remontée" à travers toutes les branches if-else du code, occasionnant une consommation importante de ressources dédiées à la mémorisation des résultats intermédiaires). Ce problème peut être résolu à l'aide d'une nouvelle version du calcul de plus faible pré-condition dans Why, appelé `fast-wp`, qui ne décompose pas les

calculs pour chacun des branchements [80], diminuant ainsi singulièrement le nombre de conditions de vérification mais les rendant par la même plus difficiles à prouver.

7.3 Intérêt de la sélection d'hypothèses

Une fois les conditions de vérification générées par Caduceus, aucun des prouveurs utilisés n'a été en mesure de valider l'ensemble des conditions de vérification. Pour réduire le nombre des conditions de vérification qui peuvent être prouvées manuellement, mais qui ne le sont pas automatiquement, les résultats du chapitre 6 ont permis de simplifier les conditions de vérification et ainsi de faciliter la tâche du prouveur.

Le premier résultat expérimental obtenu est très positif : sur le code extrait de 3000 lignes, les 10 conditions de vérification qui n'avaient pas été prouvées par Simplify, sont maintenant prouvées grâce à la sélection d'hypothèses. Le code extrait est donc prouvé à 100% automatiquement.

Pour expérimenter la sélection d'hypothèses sur le code complet, nous avons eu besoin de mettre en place une classification des conditions de vérification en fonction de leur statut (prouvée, non-prouvée). Nous avons rajouté une option à Why pour permettre cette classification. Cette option appelée `-multi-why` permet de générer les conditions de vérification en sortie de Why sous la forme suivante : un fichier pour le contexte, et un fichier par condition de vérification. Cela représente plusieurs avantages :

1. le travail de simplification peut être fait directement sur les conditions de vérification une par une.
2. l'utilisation de différents prouveurs est facilitée, en effet les conditions de vérification étant séparées, il est plus facile d'utiliser tel ou tel prouveur qui s'avère plus efficace sur une condition de vérification donnée.

Cette approche a cependant aussi soulevé un certain nombre de problèmes. En effet sur le code analysé chez Dassault Aviation, nous générons 120488 conditions de vérification, ce qui nous conduit aux limites des systèmes de fichiers des systèmes d'exploitation. En effet un système de fichier comme FAT32 est limité à 65 534 fichiers par répertoire. Bien que ce problème soit entièrement pratique, il n'en reste pas moins un point potentiellement bloquant qui a pris quelques temps pour être contourné.

Une fois ces problèmes pratiques résolus, nous avons lancé la simplification des conditions de vérification vue au chapitre 6 sur les conditions de vérification indiquées en `timeout` (c'est-à-dire lorsque le prouveur n'a pas pu se prononcer sur la validité d'une condition de vérification dans un temps raisonnable) dans la section précédente. Nous avons donc obtenu les résultats suivants à l'aide du prouveur Simplify :

Valid	:	1593	$34,10\%$
Invalid	:	11	$0,24\%$
Unknown	:	331	$7,08\%$
Timeout	:	2737	$58,58\%$
Total	:	4672	100%

On remarque une augmentation d'environ un tiers des conditions de vérification jugées valides. Les résultats obtenus à ce jour sur cette étude de cas analysée chez Dassault Aviation sont les suivants, avec sélection d'hypothèses :

Valid	:	117367	$97,41\%$
Invalid	:	5	$0,0\%$
Unknown	:	37	$0,03\%$
Timeout	:	3079	$2,56\%$
Total	:	120488	100%

Ces résultats, bien que perfectibles, restent encourageants. Notons également qu'une tâche d'identification et de résolution des causes des validités en timeout est en cours. Quant aux résultats "invalid" et "unknown", ils sont souvent dûs à des problèmes d'annotations.

7.4 Bilan

Cette étude de cas réalisée chez Dassault Aviation nous a permis de valider nos différentes approches et améliorations. En effet, bien que l'objectif des 100% de conditions de vérification valides n'ait pas été atteint, les avancées présentées dans ce chapitre sont jugées satisfaisantes et encourageantes.

Pour valider l'étude de cas de Dassault Aviation, il reste plusieurs problèmes à résoudre :

- Les fonctions C rejettées par Caduceus ou Why. Cela est dû à deux causes distinctes :
  1. Fonctions non C ANSI (casting de messages de bus de données). De fait, la correction du code analysé dépend du compilateur utilisé à cause de la non spécification du "padding" entre les champs de structures. Pour résoudre ce problème il sera nécessaire de construire un modèle bas niveau qui dépendrait du compilateur et de l'architecture de la machine.
  2. Fonctions dont le temps de calcul de leur plus faible pré-condition est trop grand. Ce problème est partiellement résolu à l'aide d'une nouvelle mouture du calcul de plus faible pré-condition dans Why, appelé `fast-wp`, qui ne découpe pas les branchements [80]. En effet, grâce a cette option, sur les 38 fonctions dont le calcul de plus faible pré-condition ne terminait pas dans un temps et un espace mémoire raisonnable, 31 terminent maintenant en temps et en espace mémoire raisonnable. Seules 7 d'entres elles posent encore des problèmes.
  3. L'une des fonctions ne peut pas être traitée spécifiquement à cause du problème d'alias de mémoire présenté à la fin de la section 5.4.
- Les conditions de vérification qui ne sont pas prouvées. Elles peuvent se classer en deux groupes :
  1. Conditions de vérification invalides. Cela peut s'expliquer de deux façons différentes :
     (a) Une erreur dans le code analysé chez Dassault Aviation. Ce cas est possible, dans la mesure où il s'agit d'une version de développement non encore testée.

(b) Une annotation insuffisante ou erronée. Comme cela a été présenté dans la section 7.1, certaines annotations ne sont pas complètes ou ne peuvent pas être prouvées, et doivent être corrigées manuellement.

2. Conditions de vérification prouvables à l'aide d'un assistant de preuve, mais non validées par les prouveurs automatiques. Bien que l'analyse effectuée dans la section 7.3 diminue le nombre de ces conditions de vérification, il en reste un certain nombre. Pour parvenir à une bonne couverture, plusieurs problématiques et pistes d'investigation doivent être envisagées :
   – inefficacité des prouveurs disponibles pour les objectifs de preuve considérés. Il existe deux grandes classes de prouveurs du premier ordre qui pourraient nous intéresser : les prouveurs TPTP (Thousands of Problems for Theorem Provers) comme vampire [105] et les prouveurs SMT (Satisfiability Modulo Theories) comme Simplify, haRVey et Ergo que nous utilisons depuis le début de ces travaux de thèse. La classe TPTP rassemble des prouveurs "arrière", c'est-à-dire des prouveurs qui prennent pour point de départ la partie conclusion de la condition de vérification considérée pour choisir efficacement les axiomes à exploiter. Et ce en opposition à la classe des prouveurs SMT qui propose des mécanismes en "avant", c'est-à-dire qui essayent de prouver que la conjection de la négation du but et du contexte est insatisfaisable. Comme vu dans la section 7.3 le nombre d'hypothèses inutiles est très important ce qui défavorise grandement les prouveurs "avant". Mais les prouveurs TPTP quant à eux ne résolvent pas ou de façon très sommaire l'arithmétique, ce qui pour nous les rend relativement inefficaces. Il apparaît donc qu'un "mixte" de ces deux approches permettrait peut-être de valider un nombre supérieur de conditions de vérification.
   – Mauvaise formulation du modèle mémoire. Il pourrait être intéressant d'envisager de reformuler le modèle mémoire vu dans le chapitre 2 et en annexe 9.1.4 de façon à ce que les prouveurs puissent mieux les exploiter. En effet, en changeant la formulation des axiomes du modèles il devrait être possible d'améliorer l'efficacité des prouveurs.

Chapitre 8

Conclusion

8.1 Résumé des contributions

D'un point de vue général, nous pouvons considérer notre contribution comme une approche complémentaire des méthodes formelles destinée à être utilisée pour des programmes de taille importante. Notre travail est dès à présent distribué dans la plate-forme Why et peut-être utilisé sur un grand nombre de types de codes ANSI-C.

Notre approche repose sur plusieurs points.

Dans un premier temps, sur une simplification du code C comme vue au chapitre 4. Cette simplification a pour objectif de supprimer du langage C toutes les notations redondantes, ambigües, ou complexes, pour aboutir à un langage C simplifié reposant sur un nombre limité de constructions syntaxiques :

- l'opérateur adressage `&`,
- l'accesseur au tableau `t[i]`,
- l'accesseur au champ de structure `s.f`,
- l'opérateur de déréférencement `*x`,
- les instructions `switch`, `case` et `default` (remplacées par une construction à base d'instructions `if` équivalentes).

Ce langage C normalisé une fois obtenu a pour avantage d'être un sous-ensemble de C tout en étant beaucoup plus "clair" et donc plus facile à analyser. Certes ce langage C normalisé possède quelques limitations exprimées dans la section 4.6 qui ne permettent pas de traiter l'ensemble de la syntaxe du langage C. Néanmoins, ces limitations ne se présentent pas sur les programmes embarqués étudiés dans le contexte de ces travaux de thèse.

Dans un deuxième temps nous avons proposé une analyse de séparation des variables globales basée sur l'ajout automatique de prédicats vue dans la section 5.1. Cette analyse avait pour but originel de facilité la tâche de l'utilisateur en supprimant une bonne partie des annotations qu'il devait ajouter manuellement. Bien qu'efficace sur ce point, elle a mis en évidence un autre problème lié au passage à l'échelle : la taille des conditions de vérification. En effet, une fois cette analyse mise en œuvre sur le code analysé chez Dassault Aviation, la taille des fichiers de condition de vérification s'est avérée incompatible des prouveurs.

Pour résoudre cette difficulté nous avons mis au point une extension du modèle mémoire

à la Floyd-Hoare de Caduceus vue à la section 5.2. Cette extension a consisté à découper la mémoire en régions, le découpage étant le plus fin possible, réunissant les pointeurs pouvant être aliasés. Cette approche ne découpe pas la mémoire aussi finement qu'on aurait pu l'espérer, mais permet de traiter la majorité des conditions de vérification liées à l'aliasing des pointeurs. Cela est fait à l'aide d'une inférence de régions qui nous permet de valider notre analyse grâce aux capacités de typage de Why. Cette approche nous a permis non seulement de résoudre le problème du nombre de prédicats générés par la première méthode, mais aussi de séparer les variables locales et les champs de structure. Comparée aux méthodes de la logique de séparation, notre analyse reste moins précise, mais, compte tenu de son efficacité, elle reste plus adaptée aux programmes de grande taille. Un article résume notre approche et son caractère innovant [73].

L'association de ces deux approches, nous a permis une avancée particulièrement sensible pour l'étude de cas de Dassault Aviation. En effet, grâce à ces deux analyses nous avons pu valider plus de 96% des conditions de vérification générées pour le code critique analysé. Cela démontre l'efficacité de notre approche pour le traitement de code de grande taille.

Un troisième point consiste dans la simplification des conditions de vérification. En effet les conditions de vérification sont souvent encombrées d'éléments de contexte de preuve : un grand nombre des hypothèses des conditions de vérification sont inutiles pour leur validation. De ce fait beaucoup de conditions de vérification ne peuvent être traitées dans des conditions acceptables (en termes de temps d'exécution, et d'espace mémoire disponible) par les outils de preuve automatique. C'est précisément pour traiter de cette difficulté, en supprimant une bonne partie des hypothèses inutiles dans les conditions de vérification, que la simplification des conditions de vérification a été mise au point.

Cette nouvelle stratégie présentée dans l'article [41] de sélection des hypothèses dans les formules issues de la vérification de programme combine deux analyses de dépendance séparée. La première analyse permet d'estimer la pertinence des variables à partir d'une analyse du graphe de dépendance. Pour ce faire, nous donnons une gradation de cette notion de pertinence : les variables seront plus ou moins pertinentes suivant une mesure de distance. Pour calculer cette distance nous utilisons le graphe de dépendance de la section 6.2 entre les variables qui permet un rapprochement entre les variables d'une condition de vérification. De la même façon, la deuxième analyse permet de connaître la pertinence des prédicats à l'aide du graphe orienté et pondéré représentant les relations entre les prédicats de la section 6.3.

Cette nouvelle approche a permis une diminution du nombre de conditions de vérification, qui n'étaient pas validées automatiquement, de l'ordre d'un tiers. Ce résultat met en évidence l'impact des limitations propres des prouveurs dans la validation des conditions de vérification, et l'importance de simplifier dans la mesure du possible la tâche des procédures de décision.

Enfin, après avoir implanté notre approche dans la plate-forme Why, nous l'avons testé sur l'ensemble du code embarqué fourni par Dassault Aviation. Cette utilisation pratique a permis d'éprouver le passage à l'échelle de notre approche. De plus cette expérimentation a été l'occasion de confronter nos approches théorique et pratique à des cas industriels représentatifs de plusieurs milliers de lignes de code.

8.2 Perspectives

Bien que des avancées sensibles et des résultats d'intérêt aient été obtenus dans le cadre de ces travaux de thèse, il nous semble pertinent d'identifier ici des axes d'amélioration complémentaires, dont l'objectif principal reste la mise à disposition de méthodes et d'outils capables de traiter l'ensemble de la problématique de la preuve de programmes de grande taille.

Méthodologie de gestion des preuves Nous avons constaté lors de notre étude de cas le besoin d'une méthodologie de gestion des preuves. En effet, lors de la mise au point des spécifications, on est amené à rejouer le processus de vérification et il est souhaitable de ne pouvoir valider de nouveau que ce qui est nécessaire. Il y a besoin de mettre au point d'une part une méthode modulaire de preuve et d'autre part d'intégrer cela dans une interface utilisateur afin de faciliter l'utilisation de cette méthodologie.

Nous avons prévu d'intégrer ultérieurement une méthodologie de gestion des preuves dans l'outil Éclipse [75]. Pour ce faire il faut créer des plugins Éclipse pour le langage C annoté, le langage Why et les conditions de vérification. De plus il est nécessaire de pouvoir lancer tous les prouveurs automatiques sur chacune des conditions de vérification ainsi que d'avoir un lien entre les conditions de vérification et les lignes du programme Why associés puis les lignes du programmes C associés. On obtiendra ainsi soit un environnement de développement qui permettra de faire directement la preuve de chacun des programmes écrits, soit un environnement de preuve où il est facile de modifier les annotations du langage source.

Jessie La plate-forme Why peut être utilisée pour faire la preuve de code C ou de code Java. Nous avons donc pensé qu'il serait intéressant de voir comment utiliser l'ensemble des travaux développés pendant ma thèse sur du code Java.

En effet dans la plate-forme Why, il existe un outil appelé Krakatoa qui a pour but de prouver les programmes Java. Cet outil, comme Caduceus, transforme le langage Java en Why. Il existe de nombreuses similitudes entre l'outil Krakatoa et l'outil Caduceus. Notre objectif a donc consisté à éviter d'effectuer la même analyse dans les deux outils. Dans ce sens, l'extension de langage C normalisé vu dans le chapitre 4 pour le traitement du code Java est en cours de développement au sein du laboratoire. Ce nouveau langage a été appelé Jessie (JC). À quelques détails près, le langage Jessie est proche du C normalisé. La principale différence consiste en l'ajout des exceptions dans Jessie.

Ce nouveau langage nous permet donc d'utiliser les résultats des travaux de thèse sur les codes Java en plus des codes C, donc de factoriser les différentes analyses faites par Krakatoa et Caduceus. Il est ainsi plus simple de rajouter une nouvelle analyse qui s'appliquera au Java comme au C.

Analyse de séparation modulaire L'analyse de séparation, bien qu'elle permette de prouver des codes de taille importante, présente une limitation importante. Comme le précise le chapitre 5, notre analyse de séparation nécessite d'accéder au code complet pour

pouvoir être utilisée. Pour rendre cette analyse plus modulaire, il paraît indispensable d'être en mesure d'exprimer dans le langage d'annotation des informations concernant les pointeurs et leurs régions. En effet si les annotations permettent d'indiquer les associations entre les pointeurs paramètres et leurs regions (plus précisément, il faudrait donner le profil avec région de chaque fonction dont on ne connait pas le code) alors il est possible d'effectuer une analyse de séparation sans avoir pour autant accès au code complet. Cette approche bien que réalisable présente l'inconvénient de demander à l'utilisateur de rajouter lui-même des annotations manuellement alors que l'un des buts de notre analyse de séparation était justement de diminuer le nombre de ces annotations manuelles. Néanmoins on pourrait envisager d'utiliser notre analyse pour générer les annotation en question.

Étude de cas fournie par Dassault Aviation Un des buts de ces travaux de thèse était de parvenir à vérifier par preuve le logiciel de l'étude de cas de Dassault Aviation. Malgré des avancées remarquables, cet objectif n'est pas rempli dans son intégralité.

Pour y parvenir, il faudrait traiter, dans un premier temps, le problème des fonctions rejetées par Caduceus comme vu au chapitre 7. Ces rejets sont dus à certains `cast` de pointeur non ANSI. Pour traiter ces fonctions, il nous faut donc une analyse qui permette de traiter les `cast` voire même les `unions` dans Caduceus (les `cast` et les `unions` ont une sémantique commune). Notre principale idée pour résoudre ce problème est de demander à l'utilisateur de fournir deux fonctions de traduction pour chaque `cast`, la première traduirait les variables du type d'origine vers une variable du type d'arrivé du `cast`, la deuxième ferait l'inverse. Ainsi à l'aide de ces deux fonctions de traduction nous conserverions deux variables logiques pour chaque variable «castée» ce qui nous permettrait de garder les deux visions de cette variable tout au long de l'analyse du programme.

Le deuxième problème pour obtenir la preuve de cette étude de cas analysée chez Dassault Aviation est la correction et la complétude des annotations. En particulier, bien que l'outil Gena vu au chapitre 7 fournisse automatiquement la grande majorité des annotations, certaines de celles-ci peuvent s'avérer incomplètes ou invalidables en l'état (compte tenu des approximations de l'analyse statique effectuée par cet outil en cours de développement). Une nouvelle version du générateur d'annotation est à l'étude afin de pallier les diverses limitations rencontrées.

Coopération entre les différentes approches Il existe aujourd'hui une offre importante en termes d'outils de vérification de code C. On peut distinguer au moins cinq types de technologies sous-jacentes à ces outils :

- Outils à base d'interprétation abstraite (Polyspace [16], Astrée [32]) : ces outils complètement automatiques permettent de détecter un sur-ensemble des instructions susceptibles de provoquer des erreurs à l'exécution. Le problème majeur est de limiter les fausses alarmes. Le projet RNTL Astrée a apporté la preuve que cette méthode peut devenir très efficace quand elle est adaptée par des experts à un domaine d'application spécifique.
- Outils à base de logique de Hoare (Caduceus, Caveat [38]) : ces outils de preuve de programmes C sont très interactifs et ne traitent qu'un sous-ensemble du langage

C. Leur utilisation nécessite l'écriture d'une spécification formelle du programme étudié. En l'absence d'une telle spécification, des interprétations abstraites simples et générales sont utilisées.
- Outils à base de modèle (Blast [3], SLAM [25]) : dans des domaines spécifiques d'application (pilotes matériels), des règles de codages peuvent être exprimées et vérifiée avec ces outils.
- Outils à base de flot de données (CCured [4], Coverity [7]) : ces outils permettent d'instrumenter du code C afin de vérifier dynamiquement certaines propriétés, et peuvent détecter statiquement la présence d'erreurs d'exécution et la violation de règles de codage, mais ne permettent pas de spécifier ou de vérifier des propriétés arbitraires.
- Outils de spécialisation (CodeSurfer [6]) : ils fournissent des analyses de flots de données et de contrôle dont la visualisation assistée est une forme de spécialisation de logiciel. Ces analyses ne sont pas utilisables par des outils de preuve et ne sont pas très précises.

Au-delà des limitations propres à chaque système existant, il n'existe pas d'intégration entre ces systèmes alors que de toute évidence cette coopération est pertinente. C'est précisément l'objectif du projet ANR CAT (C Analysis Toolbox) et de la plate-forme Frama-C [11] : permettre une interaction et une interopérabilité entre les principales techniques existantes (principalement par combinaison d'analyses par interprétation abstraite et vérification déductive).

Chapitre 9

Annexe

9.1 Modèle Caduceus

Nous donnons ici le détail de l'axiomatisation en Why du modèle mémoire utilisé par Caduceus.

9.1.1 Axiomatisation des opérations sur les pointeurs

Définition du type `pointer` :

```
type 'z pointer
```

Définition du type `addr` :

```
type 'z addr
```

Définition du type `alloc_table` :

```
type alloc_table
```

Fonction qui retourne n'importe quel pointeur (utilisé comme initialiseur par défaut) :

```
parameter any_pointer: unit -> {} 'z pointer { true }
```

Fonction qui donne la taille du bloc mémoire alloué à un pointeur donné :

```
logic block_length: alloc_table, 'z pointer -> int
```

Fonction qui donne l'adresse de base du bloc mémoire alloué à un pointeur donné :

```
logic base_addr: 'z pointer -> 'z addr
```

Fonction qui donne le decalage dans le bloc mémoire alloué à un pointeur donné :

```
logic offset: 'z pointer -> int
```

Fonction qui permet de deplacer un pointeur dans un bloc mémoire :

```
logic shift: 'z pointer,int -> 'z pointer
```

Fonction qui permet de soustraire deux pointeurs.

```
logic sub_pointer: 'z pointer, 'z pointer -> int
```

Prédicats qui permettent de comparer deux pointeurs :

```
predicate lt_pointer(p1:'z pointer,p2: 'z pointer) =
  base_addr(p1) = base_addr(p2) and offset(p1) < offset(p2)

predicate le_pointer(p1:'z pointer,p2: 'z pointer) =
  base_addr(p1) = base_addr(p2) and offset(p1) <= offset(p2)

predicate gt_pointer(p1:'z pointer,p2: 'z pointer) =
  base_addr(p1) = base_addr(p2) and offset(p1) > offset(p2)

predicate ge_pointer(p1: 'z pointer,p2: 'z pointer) =
  base_addr(p1) = base_addr(p2) and offset(p1) >= offset(p2)
```

Fonction qui permet de tester l'égalité entre deux pointeurs :

```
parameter eq_pointer :
  p:'z pointer -> q: 'z pointer ->
  {} bool { if result then p=q else p<>q }
```

Fonction qui permet de tester l'inégalité entre deux pointeurs :

```
parameter neq_pointer :
  p: 'z pointer -> q: 'z pointer ->
  {} bool { if result then p<>q else p=q }
```

Définition de la variable `alloc`, la table d'allocation globale :

```
parameter alloc : alloc_table ref
```

Prédicat qui définit la validité d'un pointeur p dans une table d'allocation a donnée :

```
predicate valid(a:alloc_table, p:'z pointer) =
0 <= offset(p)
and offset(p) < block_length(a,p)
```

Prédicat qui définit la validité d'un pointeur décalé de i, soit $p+i$, dans une table d'allocation a donnée :

```
predicate valid_index(a:alloc_table, p:'z pointer, i:int) =
0 <= offset(p)+i
and offset(p)+i < block_length(a,p)
```

Prédicat qui définit la validité du tableau p entre les index i et j dans une table d'allocation a donnée :

```
predicate valid_range(a:alloc_table, p:'z pointer, i:int, j:int)=
0 <= offset(p)+i
and offset(p)+j < block_length(a,p)
```

L'axiome qui relie le `shift` et l'`offset` :

```
axiom offset_shift :
  forall p: 'z pointer. forall i:int [offset(shift(p,i))].
  offset(shift(p,i)) = offset(p)+i
```

Les axiomes sur les décalages successifs :

```
axiom shift_zero :
  forall p: 'z pointer [shift(p,0)]. shift(p,0) = p

axiom shift_shift :
  forall p:'z pointer. forall i:int. forall j:int
  [shift(shift(p,i),j)].
    shift(shift(p,i),j) = shift(p,i+j)
```

L'axiome qui indique que l'operateur `shift` n'influence pas la fonction `base_addr` :

```
axiom base_addr_shift :
  forall p:'z pointer. forall i:int [base_addr(shift(p,i))].
  base_addr(shift(p,i)) = base_addr(p)
```

L'axiome qui indique que l'operateur `shift` n'influence pas la fonction `block_length` :

```
axiom block_length_shift :
  forall a:alloc_table. forall p:'z pointer. forall i:int
  [block_length(a,shift(p,i))].
    block_length(a,shift(p,i)) = block_length(a,p)
```

L'axiome qui indique que si deux pointeurs ont la même `base_addr` alors ils ont la même `block_length` :

```
axiom base_addr_block_length :
  forall a:alloc_table. forall p1: 'z pointer.
  forall p2: 'z pointer.
    base_addr(p1) = base_addr(p2) ->
    block_length(a,p1) = block_length(a,p2)
```

Les axiomes qui expriment que si deux pointeurs ont la même `base_addr` et le même `offset` alors ils sont égaux et inversement :

```
axiom pointer_pair_1 :
  forall p1:'z pointer. forall p2:'z pointer.
  (base_addr(p1) = base_addr(p2) and offset(p1) = offset(p2)) ->
  p1 = p2

axiom pointer_pair_2 :
  forall p1:'z pointer. forall p2:'z pointer.
  p1 = p2 ->
  (base_addr(p1) = base_addr(p2) and offset(p1) = offset(p2))
```

L'axiome qui indique que si deux pointeurs ont des `base_addr` différents alors quelque soit leur decalage ils seront différents.

```
axiom neq_base_addr_neq_shift :
  forall p1:'z pointer. forall p2: 'z pointer.
  forall i:int. forall j:int.
  base_addr(p1) <> base_addr(p2) -> shift(p1,i) <> shift(p2,j)
```

Les axiomes qui associe l'`offset` et le `shift` :

```
axiom neq_offset_neq_shift :
  forall p1: 'z pointer. forall p2: 'z pointer.
  forall i:int. forall j:int.
  offset(p1)+i <> offset(p2)+j -> shift(p1,i) <> shift(p2,j)

axiom eq_offset_eq_shift :
  forall p1:'z pointer. forall p2: 'z pointer.
  forall i:int. forall j:int.
  base_addr(p1) = base_addr(p2) ->
offset(p1)+i = offset(p2)+j -> shift(p1,i) = shift(p2,j)
```

Les axiomes qui définissent le `valid_range` :

```
axiom valid_range_valid_shift :
  forall a:alloc_table. forall p:'z pointer. forall i:int.
  forall j:int. forall k:int.
    valid_range(a,p,i,j) -> i <= k <= j -> valid(a,shift(p,k))

axiom valid_range_valid :
  forall a:alloc_table. forall p:'z pointer.
  forall i:int. forall j:int.
    valid_range(a,p,i,j) -> i <= 0 <= j -> valid(a,p)

axiom valid_range_valid_index :
  forall a:alloc_table. forall p:'z pointer. forall i:int.
  forall j:int. forall k:int.
    valid_range(a,p,i,j) -> i <= k <= j -> valid_index(a,p,k)
```

L'axiome qui défini la soustraction de pointeur :

```
axiom sub_pointer_def :
  forall p1:'z pointer. forall p2:'z pointer.
  base_addr(p1) = base_addr(p2) ->
    sub_pointer(p1,p2) = offset(p1)-offset(p2)
```

Définition des opérations Why sur les pointeurs avec des pré- et post-conditions (la comparaison de pointeurs n'a de sens que pour des pointeurs dans le même bloc) :

```
parameter shift_ : p:'z pointer -> i:int ->
  { } 'z pointer { result = shift(p,i) }

parameter sub_pointer_ : p1:'z pointer -> p2:'z pointer ->
  { base_addr(p1) = base_addr(p2) } int
  { result = offset(p1) - offset(p2) }

parameter lt_pointer_ : p1:'z pointer -> p2:'z pointer ->
  { base_addr(p1) = base_addr(p2) } bool
  { if result
        then offset(p1) < offset(p2)
        else offset(p1) >= offset(p2) }

parameter le_pointer_ : p1:'z pointer -> p2:'z pointer ->
  { base_addr(p1) = base_addr(p2) } bool
  { if result
        then offset(p1) <= offset(p2)
        else offset(p1) > offset(p2) }

parameter gt_pointer_ : p1:'z pointer -> p2:'z pointer ->
  { base_addr(p1) = base_addr(p2) } bool
  { if result
        then offset(p1) > offset(p2)
        else offset(p1) <= offset(p2) }

parameter ge_pointer_ : p1:'z pointer -> p2:'z pointer ->
  { base_addr(p1) = base_addr(p2) } bool
  { if result
        then offset(p1) >= offset(p2)
        else offset(p1) < offset(p2) }
```

9.1.2 Axiomatisation des mémoires

Définition du type des mémoires :

```
type ('a, 'z) memory
```

`'a` est le type des cellules et `'z` le type représentant la région.

Définition du type de la fonction logique d'accès aux mémoires :

```
logic acc: ('a,'z) memory, 'z pointer -> 'a
```

Définition de la fonction Why d'accès aux mémoires avec la pré-condition de validité du pointeur :

```
parameter acc_ : m: ('a,'z) memory ref -> p:'z pointer ->
  { valid(alloc,p) }
  'a reads alloc,m
  { result = acc(m,p) }
```

Définition du type de la fonction logique de modification des mémoires (retourne une nouvelle mémoire) :

```
logic upd: ('a,'z) memory, 'z pointer, 'a -> ('a,'z) memory
```

Définition de la fonction Why de modification en place des mémoires avec pré-condition de validité du pointeur :

```
parameter upd_ : m:('a,'z) memory ref -> p:'z pointer -> v:'a ->
  { valid(alloc,p) }
  unit reads alloc,m writes m
  { m = upd(m@,p,v) }
```

Les axiomes qui définissent les accès et modifications des mémoires :

```
axiom acc_upd :
  forall m : ('a,'z) memory. forall p : 'z pointer. forall a: 'a
  [acc(upd(m,p,a),p)].
    acc(upd(m,p,a),p) = a

axiom acc_upd_neq :
  forall m : ('a,'z) memory. forall p1: 'z pointer.
  forall p2 : 'z pointer. forall a: 'a [acc(upd(m,p1,a),p2)].
    p1 <> p2 -> acc(upd(m,p1,a),p2) = acc(m,p2)
```

9.1.3 Les exceptions nécessaire pour traiter les programmes C

L'exception associée au `break` :

```
exception Break
```

L'exception associée au `Continue` :

```
exception Continue
```

Les exceptions associées au `return` en fonction du type de la valeur retournée :

```
exception Return

exception Return_int of int

exception Return_real of real

exception Return_pointer
```

9.1.4 Axiomatisation du prédicat `not_assigns`

Le prédicat `not_assigns` sert à traduire en Why les clauses `assigns` de Caduceus. Il est définit par :

```
predicate not_assigns (a:alloc_table,m1:('a,'z) memory,
  m2:('a,'z) memory,l:'z pset) =
    forall p:'z pointer.
       valid(a,p) -> not_in_pset(p,l) -> acc(m2,p)=acc(m1,p)
```

c'est-à-dire que `not_assigns`(a, m_1, m_2, L) signifie que tout pointeur valide p en dehors de l'ensemble L pointe sur la même valeur dans les mémoires m_1 et m_2. Le prédicat `not_in_pset` est axiomatisé par cas suivant les différentes constructions de `pset`.

Le terme `pset_empty` désigne l'ensemble vide :

```
logic pset_empty : -> 'z pset

axiom pset_empty_intro :
  forall p:'z pointer. not_in_pset(p, pset_empty)
```

Le terme `pset_singleton`(p) désigne l'ensemble réduit à l'élément p :

```
logic pset_singleton : 'z pointer -> 'z pset

axiom pset_singleton_intro :
  forall p1:'z pointer.forall p2:'z pointer
  [not_in_pset(p1, pset_singleton(p2))].
    p1 <> p2 -> not_in_pset(p1, pset_singleton(p2))

axiom pset_singleton_elim :
  forall p1:'z pointer. forall p2:'z pointer
  [not_in_pset(p1, pset_singleton(p2))].
    not_in_pset(p1, pset_singleton(p2)) -> p1 <> p2
```

Le terme `pset_union`(L_1, L_2) désigne l'ensemble issu de l'union des ensembles L_1 et L_2 :

```
logic pset_union : 'z pset, 'z pset -> 'z pset

axiom pset_union_intro :
  forall l1:'z pset. forall l2:'z pset. forall p:'z pointer
[not_in_pset(p, pset_union(l1,l2))].
     not_in_pset(p, l1) and not_in_pset(p, l2) ->
     not_in_pset(p, pset_union(l1,l2))

axiom pset_union_elim1 :
  forall l1:'z pset. forall l2:'z pset. forall p:'z pointer
[not_in_pset(p, pset_union(l1,l2))].
    not_in_pset(p, pset_union(l1,l2)) -> not_in_pset(p,l1)

axiom pset_union_elim2 :
  forall l1:'z pset. forall l2:'z pset. forall p:'z pointer
[not_in_pset(p, pset_union(l1,l2))].
    not_in_pset(p, pset_union(l1,l2)) -> not_in_pset(p,l2)
```

Le terme `pset_star`(L, m) désigne l'ensemble des accès à la mémoire m par les éléments de l'ensemble L :

```
logic pset_star :
  'z pset, ('y pointer,'z) memory -> 'y pset (* l->x *)

axiom pset_star_intro :
  forall l:'z pset. forall m:('y pointer,'z) memory.
  forall p:'y pointer
[not_in_pset(p, pset_star(l, m))].
    (forall p1:'z pointer. p = acc(m,p1) ->
    not_in_pset(p1, l)) ->
    not_in_pset(p, pset_star(l, m))

axiom pset_star_elim :
  forall l:'z pset. forall m:('y pointer,'z) memory.
  forall p:'y pointer
[not_in_pset(p, pset_star(l, m))].
    not_in_pset(p, pset_star(l, m)) ->
    (forall p1:'z pointer. p = acc(m,p1) -> not_in_pset(p1, l))
```

Le terme `pset_all`(L) désigne $L[..]$ c'est-à-dire l'ensemble des cases des tableaux t pour t dans l'ensemble L :

```
logic pset_all : 'z pset -> 'z pset (* l[..] *)

axiom pset_all_intro :
  forall p:'z pointer. forall l:'z pset
  [not_in_pset(p, pset_all(l))].
    (forall p1:'z pointer.
       (not not_in_pset(p1,l)) ->
       base_addr(p) <> base_addr(p1)) ->
    not_in_pset(p, pset_all(l))

axiom pset_all_elim :
  forall p:'z pointer. forall l:'z pset
  [not_in_pset(p, pset_all(l))].
    not_in_pset(p, pset_all(l)) ->
    (forall p1:'z pointer.
       (not not_in_pset(p1,l)) -> base_addr(p) <> base_addr(p1))
```

Le terme `pset_range`(L, a, b) désigne $L[a..b]$ c'est-à-dire l'ensemble des cases, entre les indices a et b, des tableaux t, pour t dans l'ensemble L :

```
logic pset_range : 'z pset, int, int -> 'z pset (* l[a..b] *)

axiom pset_range_intro :
  forall p:'z pointer. forall l:'z pset. forall a:int.
  forall b:int
[not_in_pset(p, pset_range(l,a,b))].
    (forall p1:'z pointer. not_in_pset(p1, l) or
forall i:int. (a <= i <= b) -> (p <> shift(p1,i))) ->
    not_in_pset(p, pset_range(l,a,b))

axiom pset_range_elim :
  forall p:'z pointer. forall l:'z pset. forall a:int.
  forall b:int
[not_in_pset(p, pset_range(l,a,b))].
    not_in_pset(p, pset_range(l,a,b)) ->
      forall p1:'z pointer. (not not_in_pset(p1, l)) ->
forall i:int. a <= i <= b -> shift(p1,i) <> p
```

Le terme `pset_range_left`(L, b) désigne l'ensemble des cases, d'indice inférieur à b, des tableaux t, pour t dans l'ensemble L :

```
logic pset_range_left : 'z pset, int -> 'z pset (* l[..b] *)

axiom pset_range_left_intro :
  forall p:'z pointer. forall l:'z pset. forall b:int
[not_in_pset(p, pset_range_left(l,b))].
    (forall p1:'z pointer. not_in_pset(p1, l) or
forall i:int. (i <= b) -> (p <> shift(p1,i))) ->
    not_in_pset(p, pset_range_left(l,b))

axiom pset_range_left_elim :
  forall p:'z pointer. forall l:'z pset. forall b:int
[not_in_pset(p, pset_range_left(l,b))].
    not_in_pset(p, pset_range_left(l,b)) ->
      forall p1:'z pointer. (not not_in_pset(p1, l)) ->
forall i:int. i <= b -> shift(p1,i) <> p
```

Le terme `pset_range_right`(L, a) désigne l'ensemble des cases, d'indice supérieur à a, des tableaux t, pour t dans l'ensemble L :

```
logic pset_range_right : 'z pset, int -> 'z pset (* l[a..] *)

axiom pset_range_right_intro :
  forall p:'z pointer. forall l:'z pset. forall a:int
[not_in_pset(p, pset_range_right(l,a))].
    (forall p1:'z pointer. not_in_pset(p1, l) or
forall i:int. (a <= i) -> (p <> shift(p1,i))) ->
    not_in_pset(p, pset_range_right(l,a))

axiom pset_range_right_elim :
  forall p:'z pointer. forall l:'z pset. forall a:int
[not_in_pset(p, pset_range_right(l,a))].
    not_in_pset(p, pset_range_right(l,a)) ->
      forall p1:'z pointer. (not not_in_pset(p1, l)) ->
forall i:int. a <= i -> shift(p1,i) <> p
```

Le terme `pset_acc_all`(L, m) désigne l'ensemble des mémoire accèdés par les cases des tableaux t, pour t dans l'ensemble L :

```
logic pset_acc_all :
  'z pset, ('y pointer,'z) memory -> 'y pset (* l(..)->m *)

axiom pset_acc_all_intro :
  forall p:'y pointer. forall l:'z pset.
  forall m:('y pointer,'z) memory
[not_in_pset(p, pset_acc_all(l, m))].
    (forall p1:'z pointer.
      (not not_in_pset(p1,l)) ->
      forall i:int. p <> acc(m, shift(p1,i))) ->
    not_in_pset(p, pset_acc_all(l, m))

axiom pset_acc_all_elim :
  forall p:'y pointer. forall l:'z pset.
  forall m:('y pointer,'z) memory
[not_in_pset(p, pset_acc_all(l, m))].
    not_in_pset(p, pset_acc_all(l, m)) ->
    forall p1:'z pointer.
      (not not_in_pset(p1,l)) ->
    forall i:int. acc(m, shift(p1,i)) <> p
```

Le terme `pset_acc_range`(L, m, a, b) désigne l'ensemble des mémoire accèdés par les cases, entre les indices a et b, des tableaux t, pour t dans l'ensemble L :

```
logic pset_acc_range :
  'z pset, ('y pointer,'z) memory, int, int ->
  'y pset (* l[a..b]->m *)

axiom pset_acc_range_intro :
  forall p:'y pointer. forall l:'z pset.
  forall m:('y pointer,'z) memory.
  forall a:int. forall b:int
  [not_in_pset(p, pset_acc_range(l, m, a, b))].
    (forall p1:'z pointer.
      (not not_in_pset(p1,l)) ->
      forall i:int. a <= i <= b -> p <> acc(m, shift(p1,i))) ->
    not_in_pset(p, pset_acc_range(l, m, a, b))

axiom pset_acc_range_elim :
  forall p:'y pointer. forall l:'z pset.
  forall m:('y pointer,'z) memory.
  forall a:int. forall b:int.
    not_in_pset(p, pset_acc_range(l, m, a, b)) ->
    forall p1:'z pointer.
      (not not_in_pset(p1,l)) ->
 forall i:int. a <= i <= b -> acc(m, shift(p1,i)) <> p
```

Le terme `pset_acc_range_left`(L, m, b) désigne l'ensemble des mémoire accèdés par les cases, d'indice inférieur à a, des tableaux t, pour t dans l'ensemble L :

```
logic pset_acc_range_left :
  'z pset, ('y pointer,'z) memory, int -> 'y pset (* l[..b]->m *)

axiom pset_acc_range_left_intro :
  forall p:'y pointer. forall l:'z pset.
  forall m:('y pointer,'z) memory.
  forall b:int [not_in_pset(p, pset_acc_range_left(l, m, b))].
    (forall p1:'z pointer.
      (not not_in_pset(p1,l)) ->
      forall i:int. i <= b -> p <> acc(m, shift(p1,i))) ->
    not_in_pset(p, pset_acc_range_left(l, m, b))

axiom pset_acc_range_left_elim :
  forall p:'y pointer. forall l:'z pset.
  forall m:('y pointer,'z) memory.
  forall b:int [not_in_pset(p, pset_acc_range_left(l, m, b))].
    not_in_pset(p, pset_acc_range_left(l, m, b)) ->
    forall p1:'z pointer.
      (not not_in_pset(p1,l)) ->
 forall i:int. i <= b -> acc(m, shift(p1,i)) <> p
```

Le terme `pset_acc_range_right`(L, m, a) désigne l'ensemble des mémoire accèdés par les cases, d'indice supérieur à a, des tableaux t, pour t dans l'ensemble L :

```
logic pset_acc_range_right :
  'z pset, ('y pointer,'z) memory, int -> 'y pset (* l[a..]->m *)

axiom pset_acc_range_right_intro :
  forall p:'y pointer. forall l:'z pset.
  forall m:('y pointer,'z) memory.
  forall a:int [not_in_pset(p, pset_acc_range_right(l, m, a))].
    (forall p1:'z pointer.
      (not not_in_pset(p1,l)) ->
      forall i:int. a <= i -> p <> acc(m, shift(p1,i))) ->
    not_in_pset(p, pset_acc_range_right(l, m, a))

axiom pset_acc_range_right_elim :
  forall p:'y pointer. forall l:'z pset.
  forall m:('y pointer,'z) memory.
  forall a:int [not_in_pset(p, pset_acc_range_right(l, m, a))].
    not_in_pset(p, pset_acc_range_right(l, m, a)) ->
    forall p1:'z pointer.
      (not not_in_pset(p1,l)) ->
 forall i:int. a <= i -> acc(m, shift(p1,i)) <> p
```

9.1.5 Axiomatisation permettant les allocations dynamique de mémoire

Fonction permettant de savoir si un pointeur est alloué sur le tas :

```
logic on_heap : alloc_table, 'z pointer -> prop
```

Fonction permettant de savoir si un pointeur est alloué sur la pile :

```
logic on_stack : alloc_table, 'z pointer -> prop
```

Fonction qui indique que le bloc mémoire associé à p n'est pas alloué dans la table d'allocation a :

```
logic fresh : alloc_table,'z pointer -> prop

axiom fresh_not_valid :
  forall a:alloc_table. forall p:'z pointer.
    fresh(a,p) -> not(valid(a,p))

axiom fresh_not_valid_shift :
  forall a:alloc_table. forall p:'z pointer.
    fresh(a,p) -> forall i:int. not(valid(a,shift(p,i)))
```

Fonction qui indique qu'une tablea d'allocation $a2$ alloue au moins tout ce que la table d'allocation $a1$ alloué :

```
logic alloc_extends : alloc_table, alloc_table -> prop

axiom alloc_extends_valid :
  forall a1:alloc_table. forall a2:alloc_table.
    alloc_extends(a1,a2) -> forall q:'y pointer. valid(a1,q) ->
    valid(a2,q)

axiom alloc_extends_valid_index :
  forall a1:alloc_table. forall a2:alloc_table.
    alloc_extends(a1,a2) ->
    forall q:'y pointer. forall i:int.
      valid_index(a1,q,i) -> valid_index(a2,q,i)

axiom alloc_extends_valid_range :
  forall a1:alloc_table. forall a2:alloc_table.
    alloc_extends(a1,a2) ->
    forall q:'y pointer. forall i:int. forall j:int.
      valid_range(a1,q,i,j) -> valid_range(a2,q,i,j)

axiom alloc_extends_refl :
  forall a:alloc_table. alloc_extends(a,a)

axiom alloc_extends_trans :
  forall a1,a2,a3:alloc_table
  [alloc_extends(a1,a2), alloc_extends(a2,a3)].
    alloc_extends(a1,a2) ->
    alloc_extends(a2,a3) ->
    alloc_extends(a1,a3)
```

La fonction spécifie la relation entre les états de la table d'allocation avant et après une exécution de corp de fonction :

- $a1$: la table d'allocation au debut de la fonction
- $a2$: la table d'allocation avec le `return`
- $a3$: la table d'allocation après le `return`

```
logic free_stack : alloc_table, alloc_table, alloc_table -> prop

axiom free_stack_heap :
  forall a1 : alloc_table. forall a2 : alloc_table.
  forall a3 : alloc_table.
    free_stack(a1,a2,a3) ->
      (forall p : 'z pointer. valid(a2, p) -> on_heap(a2,p) ->
        valid(a3, p))

axiom free_stack_stack :
  forall a1 : alloc_table. forall a2 : alloc_table.
  forall a3 : alloc_table.
     free_stack(a1,a2,a3) ->
      (forall p : 'z pointer. valid(a1, p) -> on_stack(a1,p) ->
        valid(a3, p))
```

Les fonctions d'allocation de mémoire dynamique :

```
parameter alloca_parameter : n:int ->
{ n >= 0 }
'z pointer writes alloc
{ valid(alloc,result) and offset(result) = 0 and
  block_length(alloc,result) = n and
          valid_range(alloc,result,0,n-1) and
          fresh(alloc@,result) and on_stack(alloc,result) and
          alloc_extends (alloc@,alloc) }

parameter malloc_parameter : n:int ->
{ n >= 0 }
'z pointer writes alloc
{ valid(alloc,result) and offset(result) = 0 and
  block_length(alloc,result) = n and
          valid_range(alloc,result,0,n-1) and
          fresh(alloc@,result) and on_heap(alloc,result) and
          alloc_extends (alloc@,alloc) }
```

9.1.6 Axiomatisation du pointeur `null`

Définition de pointeur `null` avec région polymorphe :

```
logic null : 'z pointer
```

Axiome qui indique que le pointeur `null` n'est pas valide :

```
axiom null_not_valid: forall a:alloc_table. not valid(a,null)
```

Bibliographie

[1] Alloy. http://alloy.mit.edu/.

[2] Apmc. http://apmc.berbiqui.org.

[3] Blast. http://mtc.epfl.ch/software-tools/blast/.

[4] Ccured. http://manju.cs.berkeley.edu/ccured.

[5] Cminor. http://pauillac.inria.fr/~xleroy/compcert/html/Cminor.html.

[6] Codesurfer. http://www.grammatech.com.

[7] Coverity. http://www.coverity.com.

[8] Csml and mcb. http://www.cs.cmu.edu/~modelcheck/csml.html.

[9] Cwb. http://homepages.inf.ed.ac.uk/perdita/cwb/.

[10] ESC/Java. http://research.compaq.com/SRC/esc/.

[11] Frama-c. http://www.frama-c.cea.fr/.

[12] Idef0. http://www.idef.com/IDEF0.html.

[13] Jack :java applet correctness kit. http://www.gemplus.com/smart/r_d/trends/jack.html.

[14] Lotrec. http://www.irit.fr/ACTIVITES/LILaC/Lotrec/.

[15] Mariner 1. http://nssdc.gsfc.nasa.gov/nmc/masterCatalog.do?sc=MARIN1.

[16] Polyspace. http://www.polyspace.com.

[17] Smv. http://www-2.cs.cmu.edu/~modelcheck/smv.html.

[18] Spot. http://spot.lip6.fr.

[19] Uml. http://www.uml.org/.

[20] J.-R. Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.

[21] J.-R. Abrial. Event based sequential program development : Application to constructing a pointer program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003 : Formal Methods*, volume 2805, pages 51–74. Springer, 2003.

[22] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4 :32–54, 2005.

[23] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[24] J. Andronick. *Modélisation et vérification formelles de systèmes embarqués dans les cartes à microprocessur. Plateforme Java Card et Système d'exploitation*. Thèse de doctorat, Université Paris-Sud, Mar. 2006.

[25] T. Ball and S. K. Rajamani. Slam. `http://research.microsoft.com/slam`.

[26] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System : An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[27] C. Barrett and S. Berezin. CVC Lite : A new implementation of the cooperating validity checker. `http://verify.stanford.edu/CVCL/`.

[28] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany*, Lecture Notes in Computer Science. Springer, 2007.

[29] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software : The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[30] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. METEOR : A successful application of B in a large project. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99 : World Congress on Formal Methods*, Lecture Notes in Computer Science (Springer-Verlag), pages 369–387. Springer Verlag, Sept. 1999.

[31] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[32] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE static analyzer. `http://www.astree.ens.fr/`.

[33] R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[34] R. Bornat and B. Sufrin. Animating formal proof at the surface : The Jape proof calculator. *The Computer Journal*, 42(3) :177–192, 1999. `http://jape.org.uk`.

[35] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.

[36] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7 :23–50, 1972.

[37] N. Cataño, M. Gawlowski, M. Huisman, B. Jacobs, C. Marché, C. Paulin, E. Poll, N. Rauch, and X. Urbain. Logical techniques for applet verification. Deliverable 5.2, IST VerifiCard project, 2003. `http://www.cs.kun.nl/VerifiCard/files/deliverables/deliverable_5_2.pdf`.

[38] CAVEAT project. `http://www-drt.cea.fr/Pages/List/lse/LSL/Caveat/index.html`.

[39] S. Conchon and E. Contejean. The Ergo automatic theorem prover. `http://ergo.lri.fr/`.

[40] The Coq Proof Assistant. `http://coq.inria.fr/`.

[41] J.-F. Couchot and T. Hubert. A Graph-based Strategy for the Selection of Hypotheses. In *FTP 2007 - International Workshop on First-Order Theorem Proving*, Liverpool, UK, Sept. 2007.

[42] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.

[43] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 1992.

[44] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82 : Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[45] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI '00 : Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[46] L. de Moura and N. Bjørner. Z3, An Efficient SMT Solver. `http://research.microsoft.com/projects/z3/`.

[47] L. de Moura and B. Dutertre. Yices : An SMT Solver. `http://yices.csl.sri.com/`.

[48] D. Deharbe and S. Ranise. Satisfiability Solving for Software Verification. available at `http://www.loria.fr/~ranise/pubs/sttt-submitted.pdf`, 2006.

[49] D. Detlefs, G. Nelson, and J. B. Saxe. The Simplify decision procedure (part of ESC/Java). `http://research.compaq.com/SRC/esc/simplify/`.

[50] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52(3) :365–473, 2005.

[51] E. W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

[52] D. Déharbe and S. Ranise. harvey prover. `http://harvey.loria.fr/`.

[53] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[54] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, July 1999.

[55] J.-C. Filliâtre. The Why verification tool, 2002. `http://why.lri.fr/`.

[56] J.-C. Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

[57] J.-C. Filliâtre. Formal Verification of MIX Programs. In *Journées en l'honneur de Donald E. Knuth*, Bordeaux, France, October 2007.

[58] J.-C. Filliâtre, T. Hubert, and C. Marché. The Caduceus tool for the verification of C programs. `http://why.lri.fr/caduceus/`.

[59] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In J. Davies, W. Schulte, and M. Barnett, editors, *6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, Nov. 2004. Springer.

[60] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer.

[61] J.-C. Filliâtre, C. Marché, C. Paulin, and X. Urbain. The KRAKATOA proof tool, 2003. `http://krakatoa.lri.fr/`.

[62] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[63] H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST*, Newsletter 4 :13–24, 2002.

[64] H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In M. B. Dwyer, editor, *8th international SPIN workshop on Model checking of software*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer-Verlag New York, 2001.

[65] E. P. Gribomont. Simplification of boolean verification conditions. *Theoretical Computer Science*, 239(1) :165–185, 2000.

[66] D. Gries. The schorr-waite graph marking algorithm. *Acta Inf.*, 11 :223–232, 1979.

[67] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI '02 : Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, New York, NY, USA, 2002. ACM Press.

[68] J. Harrison, M. Gordon, T. Melham, R. Milner, L. Paulson, and K. Slind. The hol light theorem prover. `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580 and 583, Oct. 1969.

[70] W. A. Howard. The formulae-as-types notion of construction. In J. R. H. Jonathan P. Seldin, editor, *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[71] T. Hubert and C. Marché. A case study of C source code verification : the Schorr-Waite algorithm. In B. K. Aichernig and B. Beckert, editors, *3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, Koblenz, Germany, Sept. 2005. IEEE Comp. Soc. Press.

[72] T. Hubert and C. Marché. A case study of C source code verification : the Schorr-Waite algorithm. 2005.

[73] T. Hubert and C. Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, pages 81–93, Braga, Portugal, Mar. 2007. `http://www.lri.fr/~marche/hubert07hav.pdf`.

[74] The ISABELLE system. `http://isabelle.in.tum.de/`.

[75] J.-M. L. Jeff McAffer. *Eclipse Rich Client Platform : Designing, Coding, and Packaging Java(TM) Applications (The Eclipse Series) (Paperback)*. The Eclipse Serie, 2005.

[76] B. Kernighan and D. Ritchie. *The C Programming Language (2nd Ed.)*. Prentice-Hall, 1988.

[77] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML : notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.

[78] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. *JML Reference Manual*, Apr. 2003. draft.

[79] The LEGO proof assistant. `http://www.dcs.ed.ac.uk/home/lego/`.

[80] K. R. M. Leino. Efficient weakest preconditions. Technical Report MSR-TR-2004-34, Microsoft Research, 2004.

[81] X. Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *Conference Record of the 33rd Symposium on Principles of Programming Languages*, Charleston, South Carolina, Jan. 2006. ACM Press.

[82] N. G. Leveson and C. S. Turner. *An Investigation of the Therac-25 Accidents*. IEEE Computer Society, 1993.

[83] J. L. LIONS. Ariane 5 flight 501 failure. `http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html`.

[84] C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*. Springer, Aug. 2005.

[85] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *19th Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer, 2003.

[86] J. Meng and L. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In *ESCoR : Empirically Successful Computerized Reasoning*, 2006.

[87] B. Meyer. *Eiffel : The Language*. Prentice Hall, Hemel Hempstead, 1992.

[88] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000.

[89] R. Milner. A theory of type polymorphismn programming. *J. Comput. Syst. Sci.*, 17, 1978.

[90] J. M. Morris. A proof of the Schorr-Waite algorithm. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–51. Reidel, 1982. Proceedings of the 1981 Marktoberdorf summer school.

[91] Y. Moy. Union and cast in deductive verification. In *Proceedings of the C/C++ Verification Workshop*, volume Technical Report ICIS-R07015. Radboud University Nijmegen, July 2007.

[92] P. Müller. Specification and verification challenges. Exploratory Workshop : Challenges in Java Program Verification, Nijmegen, The Netherlands, Sept. 2006. `http://www.cs.ru.nl/~woj/esfws06/slides/Peter.pdf`.

[93] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing, Second Edition*. Wiley, June 2004.

[94] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In J. H. Reppy and J. L. Lawall, editors, *11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 62–73, Portland, Oregon, USA, 2006. ACM.

[95] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking : an empirical evaluation. *SIGSOFT Softw. Eng. Notes*, 27(6) :11–20, 2002.

[96] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science, 2001.

[97] M. Norrish. Hol 4 kananaskis-4. `http://hol.sourceforge.net/`.

[98] D. A. Plaisted and A. H. Yahya. A relevance restriction strategy for automated deduction. *Artificial Intelligence*, 144(1-2) :59–93, 2003.

[99] The PVS system. `http://pvs.csl.sri.com/`.

[100] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[101] L. Ramshaw. Eliminating go to's while preserving program structure. *J. ACM*, 35(4) :893–920, 1988.

[102] W. Reif and G. Schellhorn. Theorem proving in large theories. In M. P. Bonacina and U. Furbach, editors, *Int. Workshop on First-Order Theorem Proving, FTP'97*, pages 119–124. Johannes Kepler Universität, Linz (Austria), 1997.

[103] T. W. Reps, S. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In P. Degano, editor, *12th European Symposium on Programming, ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*, pages 380–398. Springer, 2003.

[104] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[105] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 292–296, Trento, Italy, July 1999. Springer.

[106] A. Rochfeld and J. Morejon. *La méthode Merise - Tome 3 Gamme opératoire.* Editions d'organisation (Paris), 1989. ISBN 2-7081-1057-8.

[107] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst.*, 24(3) :217–298, 2002.

[108] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10 :501–506, 1967.

[109] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[110] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. 2(3) :245–271, 1992.

[111] H. Tardieu, A. Rochfeld, and R. Colletti. *La méthode Merise - Tome 1 Principes et outils.* Editions d'organisation (Paris), 1983. ISBN 2-7081-1106-X.

[112] H. Tardieu, A. Rochfeld, R. Colletti, G. Panet, and G. Vahéee. *La méthode Merise - Tome 2 Démarches et pratiques.* Editions d'organisation (Paris), 1985. ISBN 2-7081-0703-8.

[113] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[114] R. W. Topor. The correctness of the Schorr-Waite list marking algorithm. *Acta Inf.*, 11 :211–221, 1979.

[115] J. S. Wolfram Schulte, Songtao Xia and F. Piessens. A glimpse of a verifying c compiler.

[116] L. Wos. Conquering the meredith single axiom. *Journal of Automated Reasoning*, 27(2) :175–199, 2001.

[117] L. Wos and G. W. Pieper. The hot list strategy. *Journal of Automated Reasoning*, 22(1) :1–44, 1999.

[118] L. Wos, G. A. Robinson, and D. F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4) :536–541, 1965.

Table des figures

Résumé

Dans cette thèse, nous avons contribué au développement de la plate-forme Why afin de fournir une méthode de preuve de la sûreté des programmes industriels critiques. Dans un premier temps, cette thèse présente la plate-forme Why telle qu'elle existait. Cette plate-forme, basée sur le calcul de plus faible pré-condition, s'utilise directement sur le code source et fournit en sortie les conditions de vérification qui doivent être validées pour assurer la sûreté du programme. La première contribution consiste à montrer la méthode de fonctionnement de cette plate-forme en effectuant la preuve d'un programme mettant en œuvre un algorithme complexe sur les graphes : Schorr-Waite. La deuxième contribution consiste en une analyse de séparation des pointeurs. Cette analyse, basée sur une séparation en régions de la mémoire, est une analyse par typage, donc entièrement statique. La troisième contribution consiste en une analyse de simplification des conditions de vérification. En effet, les conditions de vérification contiennent souvent plein d'hypothèses inutiles à la validation de celles-ci. Pour résoudre ce problème une analyse de pertinence des hypothèses a été développé afin de simplifier les conditions de vérification. Cette thèse se termine sur l'étude de cas d'un programme industriel critique développé chez Dassault Aviation afin de valider notre approche.

Mots clefs. Méthodes Formelles. Spécification et preuve de programme. Analyse Statique. Analyse de séparation. Langage C. Plate-forme WHY.

Abstract

In this thesis, we contributed to the development of the Why platform to design a proof method for the safety of critical industrial programs. At first, this thesis presents the platform Why such as it existed before the thesis. This platform, based on the calculus of weakest precondition, is directly used on the source code and generates the verification conditions which must be validated to ensure the program safety. The first contribution is a case study of proof of a program implementing a complex algorithm on graphs : Schorr-Waite. The second contribution consists of a separation analysis of pointers. This analysis, based on a separation in regions of the memory, is a type-based analysis, thus completely static. The third contribution consists of a method of simplification of the verification conditions. Indeed, the verification conditions often contain a large number of hypotheses that are useless for the validation of them. To resolve this problem an analysis of relevance of the hypotheses was developed to simplify the verification conditions. This thesis ends with the case study of a critical industrial program developed at Dassault Aviation to validate our approach.

Keywords. Formal Methods. Specification and proof of program. Static Analysis. Separation Analysis. C Language. Why Platform.