Verified static analyzes for low-level languages

Vincent Laporte

▶ **To cite this version:**

Vincent Laporte. Verified static analyzes for low-level languages. Programming Languages [cs.PL]. Université de Rennes, 2015. English. NNT : 2015REN1S078 . tel-01285624

HAL Id: tel-01285624
https://theses.hal.science/tel-01285624

Submitted on 9 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANNĒE 2015

THĒSE / UNIVERSITĒ DE RENNES 1
sous le sceau de l'Universitē Europēenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITĒ DE RENNES 1

Mention : informatique

Ēcole doctorale Matisse

prēsentēe par

Vincent Laporte

prēparēe ā l'Unitē Mixte de Recherche 6074 - IRISA
Institut de recherche en informatique et systēmes alēatoires
UFR Informatique Ēlectronique (ISTIC)

Vērification d'analyses statiques pour langages de bas niveau

Thēse rapportēe par

M. John Gregory MORRISETT
Professeur - Cornell University/rapporteur

M. Anders MØLLER
Professeur - Aarhus University/rapporteur

et soutenue ā Rennes
le 25 novembre 2015

devant le jury composē de :

M. Anders MØLLER
Professeur - Aarhus University / rapporteur

M. Xavier LEROY
Directeur de recherche - INRIA Rocquencourt / examinateur

M. Antoine MINĒ
Professeur des universitēs - Paris VI / examinateur

Mme. Marie-Laure POTET
Professeur des universitēs - ĒNSIMAG / examinatrice

Mme. Sandrine BLAZY
Professeur des universitēs - Univ. de Rennes 1 / directrice de thēse

M. David PICHARDIE
Professeur des universitēs - ĒNS Rennes / codirecteur de thēse

Contents

List of Figures

Rēsumē ētendu en français

1 Vérification statique de programmes

Le logiciel est omniprésent, et partout il contient des erreurs. Une erreur se manifeste lorsqu'un programme se comporte différemment de ce qui est attendu. Cela peut provenir d'un problème de conception ou d'usage : le logiciel est utilisé dans un but différent de celui pour lequel il a été conçu. Cela peut également provenir d'une inadéquation entre la spécification du programme et la fonctionnalité qu'il remplit effectivement. Enfin, cela peut être une erreur à l'exécution ; par exemple, le programme s'interrompt brutalement. Dans de nombreux cas, on s'accommode d'un programme erroné. Cependant, si celui-ci est un programme *critique*, les conséquences d'une erreur peuvent être dramatiques. Un logiciel est dit « critique » lorsqu'un comportement incorrect de ce logiciel peut menacer la sécurité ou l'intégrité de ses utilisateurs ou de son environnement d'exécution. De tels logiciels sont présents dans de nombreux systèmes, de l'airbag aux programmes de commande de vol, en passant par la signalisation des réseaux ferroviaires et les contrôleurs de centrales électriques.

Les erreurs de programmation sont diverses, ont de nombreuses causes, et de nombreux remèdes ont été proposés soit pour les éviter ou pour circonscrire leurs effets. Notamment, les langages de programmations peuvent, dans leur conception même, éviter des familles entières d'erreurs. Par exemple, la *gestion automatique de la mémoire* [JHM11], proposée par de nombreux langages, évite toute erreur de sûreté liée à la gestion de la mémoire. Si les erreurs de programmations ne peuvent être évitées, leurs conséquences peuvent être limitées par une vérification dynamique de politiques de sûreté : lorsqu'une erreur de programmation aboutirait à un comportement incorrect, le programme est interrompu sans provoquer davantage de problèmes. Par exemple, certaines conséquences des *débordements de tampons* [Ale96] peuvent être évitées en utilisant des *canaris* [Cow+98] qui permettent de vérifier, au cours de l'exécution, l'intégrité, dans une certaine mesure, des adresses de retour stockées sur la pile. De même, des protocoles d'accès à la mémoire peuvent être imposés dynamiquement par des tests d'intégrité injectés dans le programme lors de sa compilation [CCH06]. Dans des langages de haut niveau tels Java™, les programmes vérifient systématiquement que les accès aux éléments d'un tableau sont dans les bornes de celui-ci et satisfont une discipline de typage : toute violation produit une erreur qui peut être traitée par le programme ou bien l'interrompt.

S'assurer ainsi lors de l'exécution du bon déroulement d'un programme a un coût, puisque cela peut accroitre le temps d'exécution, la taille du programme ou sa consommation de mémoire. Heureusement, certaines vérifications sont superflues et cela peut être démontré statiquement : avant l'exécution, un simple examen du code source permet de prouver que certains tests ne peuvent pas échouer ; ils peuvent donc être retirés. C'est ce que font certains compilateurs optimisants ; par exemple, le compilateur Java™ Hotspot insère systématiquement des instructions pour vérifier les accès aux éléments des tableaux puis tente de les retirer en prouvant automatiquement qu'elles ne sont pas nécessaires

[WWM07]. Une autre approche consiste à prouver statiquement que des opérations non contrôlées sont sûres (par exemple que les accès aux éléments d'un tableau sont bien dans ses bornes), et d'injecter des instructions de contrôle seulement lorsque le résultat de l'analyse statique n'est pas assez précis. C'est l'approche suivie par exemple dans CCured [Nec+05] et WIT [Akr+08] qui instrumentent des programmes C pour y injecter des vérifications de sûreté. Les langages de bas niveau peuvent aussi être conçus avec des objectifs de sûreté. C'est notamment le cas de Cyclone [Jim+02 ; Gro+05] et de Rust [Rust]. Dans ces langages, tous les programmes sont à priori sûrs : le compilateur rejette les programmes qu'il ne peut prouver sûrs. Le programmeur doit explicitement annoter les parties du programme dans lesquelles les propriétés de sûreté doivent être vérifiées dynamiquement.

Ces approches dynamiques ou hybrides sont très efficaces pour empêcher le détournement d'erreurs de programmation et pour circonscrire les conséquences de ces erreurs, comme par exemple dans le cas de la faille « Heartbleed » [CVE13] : les serveurs affectés ont pu faire fuir des kibi-octets de données secrètes (mots de passes, clefs cryptographiques secrètes…) en accédant hors des bornes de tableaux. Cependant, empêcher dynamiquement que les erreurs ne se manifestent n'est pas toujours suffisant : dans un contexte critique, un programme ne doit pas s'interrompre à cause d'une erreur de programmation. En outre, dans les langages de bas niveau, dans lesquels il est permis de briser les abstractions, les vérifications dynamiques ne sont pas toujours possibles ou alors seulement à un coup exorbitant.

C'est pourquoi, afin d'assurer qu'un programme a du sens et que son exécution ne peut produire d'erreur, ledit programme doit être étudié statiquement. C'est-à-dire qu'il faut prédire tous ses comportements possibles, et ce avant toute exécution du programme, en prenant en compte tout environnement possible. Les analyseurs statiques sont des outils qui inspectent automatiquement le texte d'un programme et infèrent des propriétés concernant les exécutions de ce programme ; par exemple qu'aucune erreur d'une famille donnée ne peut se produire. Il existe un large panel d'analyseurs statiques, avec différents objectifs. Certains tels coverity [Bes+10] visent à trouver autant d'erreurs de programmation que possible : ils s'appliquent aux programmes en cours de développement pour y trouver les erreurs le plus tôt possible. Ces outils ne prétendent pas être corrects : s'ils ne trouvent pas d'erreur, cela ne signifie pas qu'il n'y en a pas. Certains analyseurs font des hypothèses pragmatiques et ne sont corrects que lorsque celles-ci sont satisfaites. Par exemple, l'outil Infer [Cal+15] qui s'applique aux applications pour téléphones mobiles suppose qu'il n'y a ni "concurrence ni *dynamic dispatch*". De même, le Static Driver Verifier [Bal+06] qui s'applique aux pilotes de périphériques écrits en C pour Microsoft Windows, est correct à condition que le programme analysé manipule la mémoire de manière sûre. Cet analyseur vérifie que des programmes spécifiques (des pilotes de périphériques) satisfont des propriétés spécifiques (que l'API des pilotes est correctement utilisée). Enfin, certains analyseurs sont corrects. Par exemple Astrēe [Mau04 ; Cou+05] a pour but de vérifier des programmes, de prouver qu'ils ne peuvent pas produire d'erreur à l'exécution. Il est employé par la société Airbus pour prouver la sûreté de programme de commande de vol électrique de certains de ses appareils.

Dans cette thèse, nous nous concentrons sure l'analyse statique correcte des programmes, qui doit faire face aux deux difficultés suivantes : d'une part le problème est indécidable ; d'autre part un tel analyseur doit être fiable. En vertu du théorème de Rice, aucun analyseur statique correct ne peut produire de résultats précis pour tous les

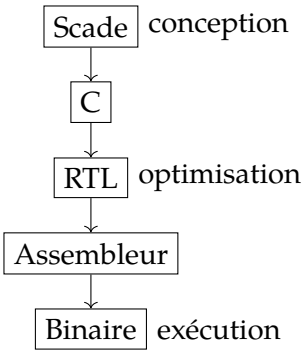
programmes. Un tel analyseur doit recourir à des approximations et échouer dans de nombreux cas. Concevoir une analyse statique requiert donc de trouver un juste milieu : quels programmes seront analysés précisément et dans quels cas l'analyseur abandonnera-t-il. Un analyseur utile doit abandonner dans de nombreux cas (une infinité) mais doit être suffisamment précis pour la catégorie de programme ciblée. Une autre propriété importante pour un analyseur est son efficacité : il doit fournir un résultat précis dans des délais suffisamment brefs. L'interprétation abstraite [CC77] fournit un cadre théorique pour construire des familles d'analyseurs statiques de coûts et précisions variés.

2 Analyse de langages de bas-niveau

Les analyseurs statiques reposent, afin d'obtenir des résultats précis et corrects, sur les abstractions fournies par le langage de programmation ciblé : variables, fonctions, objets, types, gestion automatique de la mémoire etc. Les langages de bas niveau ne fournissent que peu d'abstractions, et quand bien même il en fournirait, le programmeur est autorisé à les briser. Cela tend à suggérer qu'il est préférable d'analyser des programmes écrits dans les langages de plus haut niveau. Cependant, ce n'est pas toujours possible, ni même désirable.

Certains programmes sont directement écrits dans des langages de bas niveau, soit parce qu'ils ont besoin de briser les abstractions, ou afin de fournir des services de bas niveau comme la bibliothèque standard d'un langage de plus haut niveau, un pilote de matériel ou un système d'exploitation. Dans de tels cas, il n'existe pas de version de haut niveau qui puisse être analysée. Et dans les cas où elle existerait, elle ne serait peut-être pas disponible, par exemple pour des raisons de propriété intellectuelle.

Toutefois, les programmes sont en général écrits dans des langages destinés avant tout aux programmeurs et, avant de pouvoir être exécutés par une machine, subissent plusieurs transformations, appelées collectivement *compilation*, pour produire un programme correspondant dans une forme propice à l'exécution par une machine. Le diagramme ci-contre schématise les différentes étapes de la compilation d'un programme d'avionique écrit initialement dans le langage Scade. Ce processus de compilation peut lui-même être erroné et introduire des erreurs dans le programme compilé, ou plus simplement invalider les résultats d'une analyse obtenus sur le code source.



Une analyse peut donc avoir lieu au niveau le plus bas possible, pour manipuler le programme sous sa forme finale, tel qu'il sera exécuté. Cette approche est notamment utilisée par Reps [BR08 ; RBL06] et par nous-mêmes dans le chapitre 4. Son avantage majeur est que la distance entre le programme analysé et le programme exécuté est réduite au minimum. Mais c'est aussi son principal inconvénient : le programme est analysé à un niveau très bas, alors que le langage de programmation (si l'on peut le qualifier de tel) ne propose que peu d'abstractions voire aucune et il est très difficile de faire la moindre hypothèse quant à la forme du programme étudié. Aucune approximation adaptée ne peut être choisie : quelle est la bonne abstraction pour analyser une famille de programmes donnée ?

De plus, la plupart des analyses statiques sont conçues pour des langages de plus

haut niveau qui proposent quelques abstractions (fonctions, variables, types…) ; et un analyseur peut se reposer sur ces abstractions pour être précis : il peut être conçu pour être très imprécis quand le programme analysé ne se conforme pas à ces abstractions. Tant qu'il s'y conforme, l'analyse est aussi précise que possible, et dès qu'il ne s'y conforme plus, le résultat d'analyse, quoique correct, devient très grossier.

Il est donc judicieux d'analyser, lorsque c'est possible (c'est-à-dire lorsque le code source est disponible) un programme sous une forme de plus haut niveau et de prouver que les propriétés établies sur cette forme de haut-niveau sont préservées par le processus de compilation. Il existe deux moyens pour établir une telle preuve : la compilation vérifiée et la *translation validation*.

L'emploi d'un compilateur vérifié permet de justifier que les propriétés établies sur le code source d'un programme sont valides pour toutes les exécutions du programme compilé. Un compilateur vérifié est un compilateur accompagné d'un théorème de correction ; on le prouve une fois et il s'applique à toutes les exécutions du compilateur. Par exemple, le compilateur CompCert garantit que la compilation n'introduit pas de comportements dans les programmes compilés et que celle-ci préserve toute propriété de sûreté établie sur le code source. Le programme compilé doit toutefois être exempt d'erreurs à l'exécution, ce qui est, par chance, une propriété qui peut être démontrée par un analyseur statique.

La *translation validation* [PSS98 ; Nec00] consiste à vérifier qu'un résultat particulier est correct pour un programme particulier. Cela requiert un calcul supplémentaire à chaque exécution d'un analyseur. Cette approche est utilisée pour la vérification de l'analyseur Sparrow [Kan+14].

L'exemple du compilateur CompCert montre que les deux méthodologies peuvent être combinées : certaines passes de compilation sont certifiées alors que d'autres (comme l'allocation de registre, par exemple) sont validées à chaque exécution du compilateur.

3 Peut-on avoir confiance en un analyseur statique ?

Il peut paraître douteux que de reposer sur un analyseur statique pour justifier qu'un programme en particulier est sûr. L'analyseur statique étant lui-même un programme, il peut aussi contenir des erreurs de programmation. Pire, pour s'assurer qu'un programme est sûr, il faut désormais s'assurer qu'un autre programme — l'analyseur — est correct et remplit sa fonction. Cette dernière propriété étant à priori bien plus délicate à établir. À nouveau, l'interprétation abstraite est une méthodologie qui permet de construire des analyseurs statiques corrects.

Pour établir formellement qu'un programme ne peut produire d'erreur à l'exécution, ou qu'il calcule quelque chose de *correct*, il faut au préalable définir formellement le *sens* des programmes. La sémantique formelle d'un langage de programmation est une description mathématique de la signification des programmes. Elle définit précisément comment un programme s'exécute, quelle fonction il calcule.

La preuve de correction d'un analyseur statique, qui lie ses résultats aux comportements des programmes analysés, est généralement un effort manuel distinct de l'effort d'implantation de l'analyseur. Cela pose deux problèmes : d'une part, une preuve manuelle est sujette aux erreurs et est délicate à vérifier ; d'autre part, il y a un écart entre le modèle vérifié et le programme implanté.

Ces deux problèmes peuvent être palliés par un recours à un outil d'aide à la preuve

tel que Coq [Coq15]. Un tel outil est en premier lieu un démonstrateur de théorèmes sceptique : il peut être utilisé pour énoncer des propriétés et des preuves, et étant donné un terme de preuve — un objet syntaxique qui représente la justification qu'un énoncé est valide — Coq peut vérifier la validité de cette preuve. Même si le démonstrateur dispose de moyens de recherche de preuve automatiques et des procédures de décision qui aident l'utilisateur dans la construction des termes de preuve, il est seulement nécessaire de faire confiance au noyau, généralement réduit, qui vérifie les termes de preuve.

Coq est en outre un langage de programmation purement fonctionnel : on peut écrire, dans un même formalisme, des programmes et des propriétés. Ces programmes peuvent être exécutés directement dans Coq, mais ils peuvent aussi être *extraits* pour produire un programme OCaml qui peut être lié à d'autres programmes. Ceux-là peuvent effectuer des entrées-sorties et ainsi communiquer avec le monde extérieur.

Le compilateur CompCert est un exemple notable de programme extrait d'un développement Coq. La syntaxe et la sémantique des langages de programmation manipulés par ce compilateur (du C jusqu'à l'assembleur) ainsi que les passes de compilations sont définies en Coq de sorte que les propriétés, aussi écrites en Coq, peuvent mentionner le code effectif du compilateur pour en énoncer la correction. Le fossé entre le programme vérifié et le programme qui s'exécute est ainsi comblé.

La preuve mécanisée a été récemment appliquée avec succès dans divers domaines : des résultats mathématiques tels que la conjecture de Kepler [Hal+15], le théorème des quatre couleurs [Gon07] et le théorème de Feit-Thompson [Gon13] ; la vérification de l'implantation de systèmes d'exploitations [Kle+10].

La vérification mécanisée d'analyses statiques a été par le passé généralement appliquée au cadre classique de l'analyse de flot de données plutôt qu'à l'interprétation abstraite, plus générale. Klein et Nipkow ont proposé une inférence de type pour le bytecode Java au moyen d'une analyse de flot de données [KN06] ; Coupet-Grimal et Delobel [CD04] ainsi que Bertot *et al.* [BGL06] ont appliqué ce cadre d'analyses à des optimisations de compilateurs et Cachera *et al.* [Cac+05] à l'analyse de flot de contrôle. Vafeiadis *et al.* [VN11] s'appuient sur une simple analyse de flot de données pour vérifier une optimisation qui élimine des barrières de synchronisation dans des programmes C concurrents.

David Pichardie, dans sa thèse de doctorat [Pic05], propose une méthodologie pour vérifier la correction d'analyseurs statiques basés sur l'interprétation abstraite. Il applique cette méthodologie à des analyseurs qui manipulent des programmes écrits, à des fins d'illustration, dans des mini-langages.

Dans cette thèse, nous étendons cette méthodologie pour construire et vérifier des analyseurs statiques corrects, basés sur l'interprétation abstraite, qui manipulent des programmes exécutables : soit du code binaire, soit du code source qui peut être compilé par un compilateur vérifié ; un tel compilateur préserve la validité des résultats de l'analyse. Nos analyses visent des langages réalistes et prennent en considération les subtilités intrinsèques à l'analyse de langages de bas niveau.

4 Contributions et organisation de ce document

Dans cette thèse, nous montrons comment les méthodes de vérification mécanisée peuvent être appliquées à la construction d'analyseurs statiques certifiés qui ciblent des langages de bas niveau et tout particulièrement des représentations intermédiaires du compilateur CompCert : les langages CFG (aux chapitres 3 et 5) et C♯minor (au chapitre 6) ; et un

langage binaire jouet (au chapitre 4). Tout au long de ce document, nous décrivons plusieurs analyseurs statiques qui ont tous été prouvés corrects au moyen de l'assistant à la preuve Coq. Leur correction est établie vis-à-vis d'une sémantique du langage ciblé, et cette sémantique est aussi définie en Coq. Lorsque c'est possible, les définitions de ces sémantiques sont partagées avec le compilateur CompCert.

Le chapitre 2 présente la vérification formelle d'analyseurs statiques en Coq : comment on définit la syntaxe et la sémantique des langages de programmation, quelle est la forme générale d'un analyseur statique basé sur l'interprétation abstraite, et comment on conduit la preuve de correction d'un tel analyseur.

Nous présentons au chapitre 3 une méthodologie générale, que nous appliquons tout au long de ce document. Cette méthodologie est appliquée dans ce chapitre-là à un langage relativement simple (un sous-ensemble de la représentation intermédiaire CFG du compilateur CompCert : pas de mémoire globale, pas d'appels de fonction, pas d'arithmétique en virgule flottante), et nous obtenons néanmoins une analyse de valeur qui produit des résultats comparables à ceux des analyseurs à l'état de l'art.

En suivant la même méthodologie et en réemployant des composants de l'analyseur précédent (en particulier les interfaces et implantations des domaines numériques abstraits), nous étudions au chapitre 4 l'analyse de programmes dont la structure (notamment leur graphe de flot de contrôle) n'est pas connue avant l'analyse. Nous nous intéressons à la problématique du désassemblage correct de programmes binaires et construisons un analyseur statique vérifié pour un langage binaire jouet. Quoique très simple, ce langage, inspiré de l'assembleur x86, dispose d'un encodage des instructions à taille variable et des branchements conditionnels basés sur des drapeaux. L'analyseur est capable de désassembler les programmes, même quand leurs instructions se chevauchent ou sont produites pendant l'exécution du programme (programmes auto-modifiants). Il peut alors prouver que les programmes analysés sont sûrs, c'est-à-dire qu'ils ne peuvent pas produire d'erreur lors de leur exécution (les erreurs pourraient provenir d'une impossibilité à décoder les instructions).

Le chapitre 5 revient sur l'analyse de programmes structurés (présentés sous la forme d'un graphe de flot de contrôle immuable) mais qui opèrent sur une mémoire faiblement structurée : les variables n'y sont pas de simple entités disjointes mais des données agrégées (tableaux, structures) auxquelles on accède partiellement *via* des pointeurs.

Enfin, le chapitre 6 est dévolu à un analyseur pour le langage C♯minor, assez proche de C, qui prouve automatiquement la pré-condition de théorème de correction de CompCert.

Tous les programmes présentés dans ce manuscrit sont disponibles en ligne sur le site web associé [Web]. Ce travail s'inscrit dans le projet ANR Verasco. En particulier, ce document présente par souci de complétude des contributions d'autres doctorants.

— L'évaluation expérimentale de l'analyse de CFG présentée section 3.5 a été menée par André Maroneze [Mar14].

— Le domaine abstrait de polyèdres de l'analyseur Verasco, présenté section 6.4.2 est dû à Alexis Fouilhé [FB14].

— L'itérateur intra-procédural pour C♯minor (§ 6.1), le foncteur de domaines numériques abstraits entre entiers idéaux et entiers machines (§ 6.4.1) ainsi que le mécanisme de canaux de communication entre domaines (§ 6.4.3), présentés chapitre 6 sont dûs à Jacques-Henri Jourdan [Jou+15].

Chapter 1

Introduction

1.1 Static Verification of Software

Software is pervasive; and everywhere it has bugs. A bug manifests when a program behaves differently than what is expected. This may be a design or usage issue: the software is used for a certain purpose whereas it is designed for a different one. The bug may result from a functional error: the program is not functionally correct, it fails at fulfilling its specification. Or the bug can be a run-time error; for instance, the program crashes. In many cases, a bogus software is not an issue; at most an unpleasantness. However, when the failing software is *safety-critical*, consequences of a bug can be dramatic. A software is called "safety-critical" when a mis-behavior of this software would threaten the safety or the integrity of its users or its run-time environment. Such softwares are found in many places including airbags, fly-by-wire avionics, railway signalization, and power plant control.

Programming errors have many causes and there have been many attempts to prevent them or to limit their effects. The design of programming languages itself can help in avoiding whole families of errors. For instance, *automatic memory management* [JHM11], featured by many languages, avoids all safety errors related to memory management (e.g., double free, use after free). If programming errors cannot be avoided, their consequences can be mitigated by a dynamic enforcement of safety policies: in case a programming error would lead to a buggy behavior, the program aborts without causing any more harm. For instance, some consequences of *buffer overflows* [Ale96] can be avoided by using *canaries* [Cow+98], that enable to check at run-time the integrity, to some extent, of return addresses stored on the stack. Similarly, memory access protocols can be dynamically enforced by integrity checks introduced at compile-time [CCH06]. High-level languages like Java™ systematically check that array accesses are in-bounds and obey a typing discipline: any violation results in an error that may be caught by the program itself or halts it.

Such run-time enforcement of safety properties has costs, as it may increase the execution time, code size, or memory footprint of the running program. Fortunately, some checks can be statically proved to be unnecessary: before the execution, simply by looking at the source code, it can be proved that some checks cannot fail: therefore, they can be removed. This is what is done in optimizing compilers; for instance, the Java HotSpot™ compiler systematically inserts checks and then tries to remove them by automatically proving that they are not needed [WWM07]. An other approach is to statically prove that unchecked operations are safe (e.g., array accesses are in-bounds), and insert checks whenever the static analysis result is not precise enough. This approach is taken by CCured [Nec+05] and WIT [Akr+08] which instrument C programs to perform safety checks. Low-level

languages can also be designed with safety in mind, as for instance Cyclone [Jim+02; Gro+05] or Rust [Rust]. In these languages, safety is the default: the compiler will reject programs that it cannot prove safe. The programmer has to explicitly annotate the parts of the programs where safety properties must be dynamically checked.

These dynamic or hybrid approaches are very effective to prevent exploitation of bugs and avoid some bad consequences of programming errors, as in the case of the Heartbleed bug [CVE13]: buggy servers silently leaked kibi-bytes of secret data (user passwords, secret cryptographic keys…) by reading arrays out-of-bounds. Nonetheless, such dynamic avoidance of errors is no panacea: in critical settings, programs should not halt nor crash because of programming errors. Also, in low-level languages, in which it is allowed to break the abstractions (the programmer is trusted), dynamic checks are not possible or extremely costly.

Therefore, to ensure that a program is meaningful and that its execution cannot produce an error, said program must be statically studied. This means to predict all possible behaviors, prior to any execution of the program, taking into account any possible environment. Static analyzers are tools that automatically inspect a program text and infer properties about the executions of the program, for instance that no error of some family can occur. There is a wide range of such tools, with different aims. Some like the coverity tool [Bes+10] aim at finding as many programming errors as possible: they apply to development versions of software so as to find bugs as early as possible in the development process. Such tools do not pretend to be sound: if they find no bug, it does not mean that there are none. Some analyzers make practical assumptions and are sound under them. As an example, the Infer tool [Cal+15] targeting applications for smartphones assumes that there is no "concurrency or dynamic dispatch". Similarly, the Static Driver Verifier [Bal+06] that targets device drivers for Microsoft Windows that are written in C, is sound provided the analyzed program is memory-safe. This analyzer verifies that specific software (device drivers) satisfy specific requirements (that the driver API is correctly used). Finally, some analyzers are sound. For instance, Astrēe [Mau04; Cou+05] aims at verifying software, at proving that they cannot produce any run-time error. It is used by the Airbus company to prove the fly-by-wire program of some of its planes.

In this work, we focus on such sound static verification of programs, which faces the two following issues: the problem at hand is undecidable; and the analyzer should be reliable. By Rice's theorem, no static analyzer can yield precise results for all programs. It must resort to approximations and fail to produce any useful result in many cases. Designing a static analysis involves finding the right trade-off: what kinds of program are precisely analyzed and in which cases the analyzer gives up. A useful analyzer must give up in (infinitely) many cases but be precise enough on the family of programs of interest. An other sensible property of an analyzer is its efficiency: it must yield a meaningful answer fast enough. Abstract interpretation [CC77] provides a theoretical framework to build families of static analyzers with various precisions and costs.

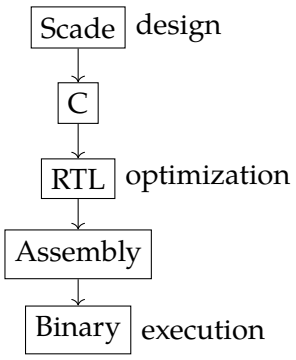
1.2 Analysis of Low-level Languages

Static analyzers rely, to get precise and sound results, on the abstractions provided by the targeted programming language: variables, functions, objects, types, automatic memory management, etc. Low-level languages feature less abstractions, or the programmer is

allowed to break them. This would thus suggest to preferably analyze programs written in higher-level languages. This is however not always possible or even desirable.

Some programs are directly written in low-level languages, because they need the power of breaking the abstractions, so as to implement low-level services like the standard library and run-time support for higher-level languages, hardware drivers or operating systems. In such cases, there is no high-level artifact that can be analyzed. And even if it exists, it may not be available, e.g., for intellectual property reasons.

Anyway, programs are usually written in languages friendly to the programmers and, before they can be actually run on a machine, they undergo several transformations, collectively referred to as *compilation*, to produce a corresponding program in a form friendly to the executing machine. The picture on the right sketches various steps in the compilation of an avionic program initially written in Scade. This compilation process may itself be wrong and introduce bugs, or more simply invalidate analysis results obtained on the source code.



An analysis can therefore take place at the lowest-possible level, operating on the program as it is executed. This approach is notably taken by Reps [BR08; RBL06] and applied in Chapter 4. Its main advantage is that there is no gap between the analyzed program and the run one. This is also its main drawback: the program is analyzed at a very low level, at which the programming language (if that can be called so) features little or no abstraction and it is very hard to do any hypothesis about the program shape. Adequate approximations cannot be chosen: what is the right abstract domain to analyze a given family of programs?

In addition, most static analyses are designed for higher-level languages which feature some abstractions (functions, variables, types…); and an analyzer can rely on these abstractions so as to be precise: it can be designed as very imprecise when the analyzed program does not adhere to the abstractions: while it does, the analysis is as precise as possible, and as soon as it does not, the analysis result, while correct, gets very coarse.

It is therefore judicious to analyze, when this is possible (i.e., when the source code is available) a high-level form of the program and to prove that properties inferred on this high-level form are preserved by the compilation process. There are two approaches to establish such properties: verified compilation and translation valitation.

The use of a verified compiler can justify that properties established on the source program hold for all executions of the compiled program. A verified compiler is a compiler which comes with a correctness theorem; it is proved once and holds for all runs of the compiler. For instance, the CompCert compiler guarantees that the compilation does not introduce behaviors in the compiled program and preserves all safety properties established on the source code. The compiled program must nonetheless be free of run-time error, which fortunately is the kind of properties that a static analyzer may infer.

Translation validation [PSS98; Nec00] consists in verifying that a particular result is correct for a particular program. This involves an additional computation on every execution of an analyzer. Such approach has been applied to the verification of the Sparrow analyzer [Kan+14].

The example of CompCert shows that both method can be combined: some compilation passes are certified while other are validated (as the register allocation, for instance).

1.3 Trusting a Static Analyzer

Relying on a static analyzer to justify that a given program is safe may look dubious. The static analyzer itself being a program, it is also subject to bugs. Worse, to believe that some program is safe, we now have to trust that an other program is functionally correct. This last property being a priori much harder to establish. Again, abstract interpretation is a methodology to design sound static analyzers.

So as to formally establish that a program cannot produce an error, or that it computes *the right thing*, we need to formally give *meanings* to programs. The formal semantics of a programming language is a mathematical description of the meaning of programs. It precisely defines how a program is executed, what function it computes.

The soundness proof of static analyzers, which links their results to the behaviors of the analyzed programs, is often a pen-and-paper effort, distinct than the implementation effort. This raises two issues: first, the proof effort is error-prone and difficult to check; second, there is a gap between the verified model and the implemented program.

Both issues can be addressed by resorting to a proof assistant like Coq [Coq15]. Such a tool is first a skeptical theorem prover: it can be used to write statements and proofs, and given a proof term —a syntactic object that conveys the argument that some statement holds—, Coq can check its validity. Even if the theorem prover features some automated proof search and decision procedures that help the user in the process of building the proof terms, we only need to trust a reduced kernel that performs the proof checking.

Coq is also a purely functional programming language: we can write, in a single formalism, programs and properties. These programs can be directly run inside Coq, but they can also be *extracted* to the OCaml programming language, so as to be linked with other programs which may perform inputs and outputs and communicate with the outer world.

The CompCert compiler is a notable example of program that is extracted from a Coq development. The syntax and semantics of the programming languages manipulated by the compiler (from C to assembly) and the compilation passes are defined in Coq so that properties, also written in Coq, can mention the actual code of the compiler to state its correctness. There is no more gap between the code that is verified and the code which runs.

Mechanized proof has been recently applied with great success to various domains: mathematical results as the Kepler conjecture [Hal+15], the Four Colour Theorem [Gon07], and the Feit-Thompson theorem [Gon13]; verification of the implementation of operating systems [Kle+10].

Previous work on mechanized verification of static analyses has been mostly based on classic data-flow frameworks rather than on the more general abstract interpretation. Klein and Nipkow instantiate the data-flow framework for inference of Java bytecode types [KN06]; Coupet-Grimal and Delobel [CD04] and Bertot *et al.* [BGL06] for compiler optimizations, and Cachera *et al.* [Cac+05] for control-flow analysis. Vafeiadis *et al.* [VN11] rely on a simple data-flow analysis to verify a fence elimination optimization for concurrent C programs.

David Pichardie, in his Ph.D. [Pic05], proposes a methodology to verify the soundness of static analyzers based on abstract interpretation. He applies this methodology to analyzers which operate on mini languages and programs written for the purpose of illustration.

In this work, we extend this methodology and build and verify sound static analyzers, based on abstract interpretation, that analyze programs that can be executed: either binary code, or source code that can be compiled by a verified compiler; said compiler preserves the analysis results. Our analyses target realistic languages and take into account the subtleties inherent to the analysis of low-level languages.

1.4 Contributions and Structure of this Document

In this work we show how mechanized verification methods can be applied to the construction of certified static analyzers that target low-level languages, in particular intermediate representations of the CompCert compiler: CFG (in chapters 3 and 5) and C♯minor (in chapter 6); and a toy binary language (in chapter 4). Throughout this document, we describe several static analyzers that are all proved sound within the Coq proof assistant. The soundness is established with respect to a semantics of the target languages, also defined in Coq. Where relevant, the definitions of theses semantics are shared with the CompCert compiler.

The chapter 2 introduces to the formal verification of static analyzers in Coq: how a programming language syntax and semantics can be defined, what is the general shape of an abstract-interpretation-based static analyzer and of its soundness proof.

We then present in chapter 3 a general methodology that is used throughout this document, and applied to a simple language (a subset of the CFG intermediate representation of CompCert: no memory, no function calls, no floating-point values), and get nonetheless a value analyzer which yields results comparable to state-of-the art analyzers.

Following the same methodology, and reusing parts of the aforementioned analyzer (interfaces and implementations of numerical domains in particular), we study in chapter 4 the analysis of programs whose structure (notably their control-flow graph) is not known before the analysis. We address the issue of sound disassembling of binary programs, and build a verified static analyzer for a toy binary language. Despite being very small, this language, which is inspired from x86 assembly, features a variable-length encoding and flag-based conditional branches. The analyzer is able to disassemble programs, even when instructions overlap or are produced during the program execution. It then proves that analyzed programs are safe, i.e., cannot produce any error when they are run. (Errors correspond to decoding failures.)

The following chapter 5 comes back to the analysis of well-structured programs (given as an immutable control-flow graph) but that operate on a loosely structured memory: variables are not simple disjoint entities but aggregated data (arrays, structures) accessed chunk-wise through pointers.

Bringing all together, chapter 6 is devoted to an analyzer for the C♯minor language, close to C, so as to automatically discharge the precondition of the CompCert correctness theorem.

All programs that are shown in this document are available on the companion web-site [Web]. This is joint work in the Verasco ANR project. In particular, this document presents, for completeness, the following contributions of other Ph.D. students.

- The experimental evaluation of the CFG analyzer presented in Section 3.5 is due to André Maroneze [Mar14].

- The polyedral domain of the Verasco analyzer presented in Section 6.4.2 is due to Alexis Fouilhé [FB14].
- The intra-procedural iterator for C♯minor (§ 6.1), the functor from ideal integers to machine integers (§ 6.4.1), and the channel mechanism for communication between domains (§ 6.4.3) presented in chapter 6 are due to Jacques-Henri Jourdan [Jou+15].

Chapter 2

Context

In this chapter we briefly introduce to the main tools our work is built on: static analysis based on abstract interpretation and the CompCert verified C compiler, both verified with the Coq proof assistant.

2.1 Introduction to abstract-interpretation-based static analysis in Coq

So as to describe the design, implementation and proof of static analyzers in Coq, we will first present how a programming language syntax and semantics can be defined. This will be the opportunity to highlight some aspects of Coq that are used throughout this work. Then we will discuss the abstract-interpretation framework that encompasses all interpreters that will be presented in this document.

All Coq snippets in this section are extracted from a short development (about a thousand lines of Coq) available on the companion web page [Web].

2.1.1 Syntax of a Toy Language

Let's start with a (minimalist) toy language, called GE (as GRAPH & EXPRESSIONS), with (unbounded) integer variables, arithmetic expressions, and basic instructions (assign and branch) arranged in a control-flow graph. Its formal abstract syntax is given in Figure 2.1. Program variables (type VAR) are represented by positive numbers. Expressions (type EXPR) are represented as abstract syntax trees whose leaves are constant values or variable names and nodes represent addition, multiplication or comparison of their subtrees. A

```
Definition var : Type := positive.

Inductive expr : Type :=
| EConst (v: Z)
| EVar (x: var)
| EAdd (e₁ e₂: expr)
| EMul (e₁ e₂: expr)
| ELt (e₁ e₂: expr).
```

```
Definition node : Type := positive.

Inductive instr : Type :=
| IAssn (x: var) (e: expr) (s: node)
| IIf (g: expr) (s₁ s₂: node)
| IStop.

Definition prog : Type := Map [ node, instr ].
```

Figure 2.1: Syntax of GE, a toy control-flow graph language with structured expressions

```
Example fact (x y n: var) : prog := of_list (
   (1, IAssn n (EConst 0) 2)
 :: (2, IAssn y (EConst 1) 3)
 :: (3, IIf (ELt (EVar n) (EVar x)) 4 6)
 :: (4, IAssn n (EAdd (EVar n) (EConst 1)) 5)
 :: (5, IAssn y (EMul (EVar y) (EVar n)) 3)
 :: (6, IStop)
 :: nil).
```

```
(*
1: n ⟵ 0
2: y ⟵ 1
3: while ( n < x ) {
   4: n ⟵ n + 1
   5: y ⟵ y × n }
6: stop
*)
```

Figure 2.2: Example GE program computing the factorial function

program (type prog) is a finite map[1] from program points (type node) to instructions (type instr). Each instruction embeds the name of its successors nodes. An instruction is either an assignment IAssn x e s of an expression e to a variable x (and execution proceeds at node s), a conditional branch IIf g s_1 s_2 to node s_1 or s_2, depending on the value of the guard g, or IStop that terminates the program execution.

As an illustration, we can build the program fact, shown on Figure 2.2, which computes the factorial of x in variable y using auxiliary variable n (the same program is given on the right in an informal concrete syntax). The of_list function builds a map from an association list.

So as to be able to reason about such programs, we need to define the language semantics. To this end, we first introduce two aspects of Coq: how properties about *sets* can be expressed, and how we specify abstract data types, cornerstone of modular developments.

2.1.2 Sets as Propositions

When used in specifications (as opposed to programs and computations), sets of values of some type A can be described by predicates over this type, i.e., functions of type A → Prop that map every value of type A to a proposition (of type Prop) that expresses whether this value is a member of the set. Figure 2.3 shows how various set operations are thus defined. Given a set X, i.e., a predicate over A, the fact that an element a is a member of this set is reflected by the property (X a), that we conveniently write "a ∈ X". Set inclusion, intersection and union are described respectively as implication, conjunction and disjunction of membership facts. The empty set is the predicate that is always false. A singleton is defined as the equality to its element. Given two sets X and Y, their product X × Y is the sets of pairs (x, y) such that x is a member of X and y is a member of Y. The term (Union A B X f) represents the set $\bigcup_{a \in X} f(a)$. We use the Notation keyword to define non-alphabetic symbols and infix notations. The Union operator is defined as usual, with a plain Definition; a notation for it will be defined later (after type classes are introduced).

2.1.3 Abstract Data Types

The Coq infrastructure proposes at least two mechanisms to define and use abstract data types: modules and records. The main difference is that records are first-class objects:

[1] The notation Map [K, V] represents the type of finite maps whose keys have type K and values type V; it comes from the Containers library, contributed by Stēphane Lescuyer.

```
Notation ℘(A) := (A → Prop).
Notation a ∈ P := (P a).
Notation X ⊆ Y := (∀ a, a ∈ X → a ∈ Y).
Notation X ∩ Y := (λ a, a ∈ X ∧ a ∈ Y).
Notation X ∪ Y := (λ a, a ∈ X ∨ a ∈ Y).
Notation ∅ := (λ _, False).
Notation {{ a }} := (eq a).
Notation X × Y := (λ a, let (x, y) := a in x ∈ X ∧ y ∈ Y).
Definition Union (A B: Type) (X: ℘(A)) (f: A → ℘(B)) : ℘(B) := λ b, ∃ a, a ∈ X ∧ b ∈ f a.
```

Figure 2.3: Sets as propositions

abstracting over a module requires a functor (i.e., a construction specific to modules), when abstracting over a record is the usual for-all quantification. A decisive advantage of records is the type-classes mechanism built atop them (briefly described below). Here we focus on describing our use of records to specify, define and use abstract data types in this work.

As an example, we define an abstract type of monads. Given a type constructor M, its monad structure is defined by a record with two fields, named unit and bind. The first one takes as argument a type A, of value a of this type, and builds the corresponding monadic value of type M A. The second takes as arguments two types A and B, and composes a monadic value of type M A with a function from A to M B. These functions are polymorphic: for instance unit takes as argument any type and a value of this type. There is no specific concept of polymorphism is Coq: it is a particular instance of a dependent type. After their definition, the fields of the record take two additional arguments: the type constructor M and the record itself (of type monad M).

```
Record monad (M: Type → Type) : Type := {
  unit : ∀ A, A → M A;
  bind : ∀ A B, M A → (A → M B) → M B
}.
```

This defines a new type constructor, monad, that can be used to define generic programs, over any monad. For instance, given a type constructor M and its monad structure S, we can define a generic lift that takes any function (f: A → B) into the corresponding lifted function (lift f: M A → M B). The Context keyword introduces hypotheses (or variables) that are shared by several definitions; their scope is bounded by *sections*. To ease the readability, we define the usual notations ret and >>= for the two monadic operators. This program does not know what the actual monad is, but it can refer to the fields of the structure S. Underscores _ in terms are placeholders that are automatically filled by the type-inference engine of Coq.

```
Context (M: Type → Type) (S: monad M).
Notation "'ret' x" := (unit M S _ x).
Notation "x >>= f" := (bind M S _ _ x f).

Definition lift (A B: Type) (f: A → B) (m: M A) : M B := m >>= λ a, ret (f a).
```

To define an actual monad structure for some concrete type constructor, we have to define a particular record with the appropriate operators. For instance, the set monad can be defined as follows. The unit operator builds a singleton set and the bind operator is the Union that we have previously defined.

```
Definition set A := ℘(A).
Definition set_monad : monad set := {|
  unit := λ A a, {{ a }};
  bind := λ A B (X: set A) (f: A ⟶ set B), Union _ _ X f
|}.
```

In order to be able to prove facts about programs that use monads, we need to specify the monad abstract data-type. As for the operators, the specification of an abstract data type can be defined as a record, whose fields are invariants of the data type. In the case of monads, the three monad laws can be specified as follows. This specification is parameterized by an equivalence relation e over monadic values (with the infix notation · === ·). The parameter E is a *proof* that this relation is actually an equivalence.

```
Context (e: ∀ A, relation (M A)) (E: ∀ A, Equivalence (e A)).

Record monad_spec : Prop := {
  unit_left : ∀ A B (a: A) (f: A ⟶ M B), (ret a >>= f) === f a;
  unit_right : ∀ A (m: M A), (m >>= (λ x, ret x)) === m;
  bind_assoc: ∀ A B C (m: M A) (f: A ⟶ M B) (g: B ⟶ M C),
    ((m >>= f) >>= g) === (m >>= (λ x, f x >>= g))
}.
```

Then, we can prove properties about generic programs. For instance, that the lifted identity is the identity (up to functional extentionality).

```
Context (S_correct: monad_spec).

Lemma lift_id A : lift A A id === id.
Proof. intros x. apply S_correct. Qed.
```

Finally, to be able to use such a theorem for a particular monad instance, we have to prove the monad laws. In the case of the set monads, we prove them for a particular equivalence relation, eq_set, which is extensional.

```
Definition eq_set A : relation (set A) := λ X Y, ∀ a, a ∈ X ↔ a ∈ Y.
Lemma eq_set_equiv A : Equivalence (eq_set A).
Proof. vm_compute; firstorder. Qed.

Lemma set_monad_correct : monad_spec set set_monad eq_set eq_set_equiv.
Proof. now split; vm_compute; firstorder; subst. Qed.
```

2.1.4 Type Classes

This is a brief introduction to type-classes in Coq [SO08; SW11], that presents sufficient material to understand the uses of this feature in this work. Many aspects are not discussed. In particular how it relates to *canonical structures* — an other Coq feature somehow similar to type-classes — and how to automate proof search thanks to the inference mechanism.

Type classes provide a facility for (static) overloading, i.e., using the same name for different functions, ambiguities being resolved from the context during type inference.

As an example, consider *join*, a binary operator defined on several data-types (say A and B). These operators have similar properties and similar uses, thus deserve the same name. The different operators could simply be named joinA and joinB, but the name of the type of its argument is often easily guessed from the arguments themselves. Also, it is convenient to use a notation — say x ⊔ y — for both operators.

The basic idea is to add a level of indirection and define a generic join function:

```
Definition join (X: Type) (j: X → X → X) : X → X → X := j.
```

Then, we can use (join A joinA a a′) in lieu of (joinA a a′) and similarly (join B joinB b b′) in lieu of (joinB b b′). The type-classes feature provides a powerful inference mechanism that allows to leave arguments X and j of join as *implicit* and simply write (join a a′) and (join b b′), or more conveniently (a ⊔ a′) and (b ⊔ b′) respectively, after the corresponding notation has been defined. Those expressions have holes that are automatically filled.

This mechanism requires some annotations. The types of the holes need to belong to a declared *class* and the value that will fill this hole has to be declared an *instance* of this class. To come back to the example of join, the class of types with such an operator could be defined as follows.

```
Class join_op (X: Type) :=
  join : X → X → X
.
```

This reads as: a type X belongs to the class join_op only if there is an operator with type X → X → X. This operator can then be referred to using the generic name join. The inference mechanism does not pick any term with the right type, but only one that has been explicitly designated. To label a term of the right type as a join operator for a given type, the Instance keyword is to be used as follows.

```
Instance join_op_A : join_op A := joinA.
```

Then, when join is applied to arguments of type A, the inference machinery will kindly find the expected operator. The final step is to define a nice notation for this overloaded operator.

```
Notation "x ⊔ y" := (join x y).
```

Usually, in a given context, there is only one instance for a given class with a given set of arguments. The Global and Local keywords, used with the Section facility, enable to control the scopes of the instance declarations.

Some instances can be parameterized by other instances. As an example, consider the data-type X+⊥ (a notation for botlift X) that adds a new element Bot to an existing type X.

```
Inductive botlift (X: Type) : Type :=
| NotBot (x: X)
| Bot.
```

This element is meant to be an identity with respect to a join operator. Therefore, we can define a generic join operator for *any* lifted data-type that belongs to the join_op class:

```
Instance botlift_join (X: Type) (J: join_op X) : join_op (X+⊥) :=
 λ x y : X+⊥,
 match x, y with
 | Bot, _ => y
 | _, Bot => x
 | NotBot x', NotBot y' => NotBot (x' ⊔ y')
 end.
```

Now, in a context in which terms u and v have type B+⊥, the expression u ⊔ v, which actually is the term (join _ _ u v) with two holes, will be filled with respectively B+⊥ (from the types of u and v) and (botlift_join B join_op_B) from the available instances.

Similarly, given an instance for a lifted type (X+⊥), we could define an instance for the corresponding base type X:

```
Instance botunlift_join (X: Type) (J: join_op (X+⊥)) : join_op X :=
 λ x y : X, NotBot x ⊔ NotBot y.
```

This would be a bad idea in general. Indeed, instance inference may now loop: to find an instance of (join_op X), apply botunlift_join and botlift_join and start again.

In this work, the main type classes are for: decidable equality; for debugging purposes, a to_string function that produces a readable representation of its argument; lattice structure (maximal element, join, widening); concretizations; reductions; abstract domains.

2.1.5 Concrete Semantics

With powerful Coq tools at hand, we can come back to our toy programming language and define its semantics.

The semantics of expressions is given in Figure 2.4. Programs operate on an environment (type env) that maps program variables to integers. Truth values are encoded with integers zero and one. Any non-zero integer is considered true. Semantics of expressions is defined through an evaluation function, that can be seen as a big-step operational semantics.

The meaning of programs is defined as a small-step operational semantics in Figure 2.5. The state of (the machine executing) the program (type exec_state) is made of a program counter (the node, in the program control graph, corresponding to the next instruction to execute) and an environment. The step function returns the next execution state or None when the execution is over or stuck (execution is *stuck* when the program point is not bound in the program control-flow graph, or when an expression evaluates an unbound variable). The run function executes at most n steps of a program. We can then define the set of reachable states from an initial state i when running a program p. A state is said *initial* when the program counter is 1; notice that the environment is not necessarily empty in an initial state (bound variables can be seen as input data). Finally, we define

```
Definition env : Type := Map [ var, Z ].

Definition of_bool (b: bool) : Z :=
 if b then 1 else 0.

Definition val_is_true (v: Z) : bool :=
 if v == 0 then false else true.

Definition zlt (a b: Z) : Z :=
 of_bool (a <? b).
```

```
Fixpoint eval_expr (e: expr) (ρ: env)
 : option Z :=
 let binop op e₁ e₂ :=
    eval_expr e₁ ρ >>= λ v₁,
    eval_expr e₂ ρ >>= λ v₂,
    ret (op v₁ v₂) in
 match e with
 | EConst v => ret v
 | EVar x => ρ[x]
 | EAdd e₁ e₂ => binop Z.add e₁ e₂
 | EMul e₁ e₂ => binop Z.mul e₁ e₂
 | ELt e₁ e₂ => binop zlt e₁ e₂ end.
```

Figure 2.4: Semantics of GE expressions

```
Definition exec_state: Type := (node * env).

Definition step (p: prog) (st: exec_state)
 : option exec_state :=
 let (pc, ρ) := st in
 p [ pc ] >>= λ i,
 match i with
 | IAssn x e s =>
   eval_expr e ρ >>= λ v,
   let ρ' := ρ [ x <- v ] in
   Some (s, ρ')
 | IIf g t e =>
   eval_expr g ρ >>= λ v,
   let pc' := if val_is_true v then t else e in
   Some (pc', ρ)
 | IStop => None end.

Definition stuck (p: prog): ℘(exec_state) :=
 λ e, step p e = None ∧ p[fst e] ≠ Some IStop.
```

```
Fixpoint run (p: prog) (n: nat) (st: exec_state)
 : exec_state :=
 match n with
 | O => st
 | S n′ => match step p st with
   | Some st′ => run p n′ st′
   | None => st end end.

Definition reachable (p: prog) (i: exec_state)
 : ℘(exec_state) :=
 λ e, ∃ n, run p n i = e.

Definition initial: ℘(exec_state) :=
 λ e, fst e = 1.

Definition sem (p: prog): ℘(exec_state) :=
 Union initial (reachable p).

Notation "⟦ p ⟧" := (sem p).
```

Figure 2.5: Semantics of GE programs

the reachability semantics sem of a program p as the set of states reachable (in some finite number of steps) from any initial state. An interesting characterization of this semantics is that it is the least post-fixpoint[2] of the following function, where of_opt casts the result of step into a set of states.

```
Definition SEM (p: prog) := λ s, initial_state ∪ Union s (of_opt ∘ step p).
```

This *collecting* semantics of programs is precise enough to express *safety* properties of programs as "in all reachable states, variable x is non-negative" or "execution cannot be stuck".

In the following, we will describe how abstract interpretation enables the automatic proof of such properties.

2.1.6 Abstract Semantics

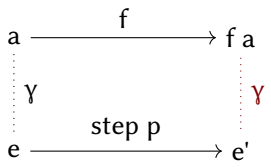
Abstract interpretation is a wide topic that provides a general framework to design and prove correct sound static analyzers [CC77]. The meaning of "interpretation" is twofold. On one hand it is a way to "give a meaning to" programs, i.e., a non-standard semantics. On the other hand it describes a way to execute programs. We use abstract interpretation as a tool for building sound static analyzers; we do not formalize this theory. Also, since we will only prove results about soundness of the analyses (and none about their precision), we do not introduce abstraction functions nor Galois connexions.

A static analysis computes facts about programs. These facts belong to some class specific to the analysis. This class is defined through a so-called "abstract domain", a machine representation of these properties. This data-type is partially-ordered by a relation le, noted $\cdot \sqsubseteq \cdot$. Each element of the abstract domain represents a collection of execution states: this is formalized through a concretization relation, usually called γ. The partial order of the domain reflects the precision of the elements: greater elements denote coarser properties (i.e., larger sets of execution states).

```
Context (A: Type) (le: A → A → bool) (γ: A → ℘ exec_state).
Infix "⊑" := le.
Context (le_sound: ∀ a a', a ⊑ a' → γ a ⊆ γ a').
```

The abstract semantics of a program p is defined as a (usually monotone) function f: A → A. It should be correct w.r.t. the concrete semantics in the following sense: given an abstract value a and a concrete state e in the concretization γ(a) of the abstract value, for every step of the program p from e to some state e', then this final state e' is in the concretization of the abstract result f(a). This soundness condition is depicted by the diagram on the right, in which the right-most γ edge is the conclusion.

```
Context (f_sound: ∀ a e,
  e ∈ γ(a) → step p e ⊆ γ(f a)
).
```



In this setting, any post-fixpoint of the abstract semantics that also over-approximates initial states approximates all reachable states, as summarized by the following theorem.

[2] The least post-fixpoint of a function SEM is the smallest set s that satisfies: (SEM s) ⊆ s.

```
(* Iterates f at most n times from a. *)
Fixpoint iter (n: nat) (a: A): A * nat :=
 match n with
 | O => (a, n) (* give up: no more fuel *)
 | S n' =>
  let a' := f a in (* do one step *)
  if a' ⊑ a (* post-fixpoint reached? *)
  then (a, n)
  else iter n' a' (* continue *)
 end.
```

```
(* The possible analysis outcomes. *)
Inductive outcome: Type := OK | Fuel.

(* Analysis from an initial state. *)
Definition analyzer n (ι: A) : A * outcome :=
 let (π, n) := iter n ι in
 (π, if n == O then Fuel else OK).

(* Soundness of this generic analyzer. *)
Theorem analyzer_pfp n ι π:
 analyzer n ι = (π, OK) → f π ⊑ π.
```

Figure 2.6: Generic post-fixpoint solver

```
Theorem soundness: ∀ a: A, initial_state ⊆ γ(a) → f a ⊑ a → ⟦p⟧ ⊆ γ(a).
```

We can then build a simple iterator, as shown on Figure 2.6, that starts from an initial state ι and repeatedly applies the abstract semantics until a post-fixpoint is found. This iterator may not terminate: if the abstract semantics is not monotone, if $\iota \sqsubseteq f(\iota)$ does not hold (i.e., the sequence is not initially increasing), or if the domain A has infinite ascending chains. Therefore, so as to guarantee the termination of the analyzer, we rely on *fuel*, i.e., a counter that decreases on every recursive call; when the analyzer runs out of fuel, it gives up.

However, we take care to design analyses such that all three conditions hold. In particular, when dealing with domains with infinite (or very long) ascending chains, we use *widening* operators so as to accelerate the iteration and ensure its termination. A widening is a binary operator, usually written $\cdot\nabla\cdot$, that enjoys the following condition: for each sequence $(x_i)_{i\in\mathbf{N}}$ that is increasing (i.e., $\forall i \in \mathbf{N},\ x_i \sqsubseteq x_{i+1}$), the sequence $(y_i)_{i\in\mathbf{N}}$ defined as $y_0 = x_0$ and $\forall i \in \mathbf{N},\ y_{i+1} = y_i \nabla x_{i+1}$ is such that there is a rank k after which the sequence is constant (i.e., $\forall i \in \mathbf{N},\ y_{k+i} = y_k$) [CC77].

2.1.7 Flow-Sensitive Analyzer

The generic iterator that we have just shown is somehow too general. All analyzers that we will describe in this document have a more specific shape: the abstract domain (of some type Σ) only represents environments (rather than execution states). From this domain, we build an abstraction of states by attaching to each program point a value of this domain: the previous framework is instantiated with the type Map [node, Σ].

The abstraction for environments comes with two operators (in addition to the usual partial order and join) that model the two fundamental instructions of the language: assignments and conditional branches. These operators may return no value when they discover a contradiction: for instance, a call to (assume e b σ) may prove that in no concrete environment in the concretization of σ the expression e can evaluate to a value whose boolean interpretation is b; in such a case it returns None. These operators come with soundness conditions (in which γo is the concretization relation of Σ lifted to options, and ρ[x <- v] is the map ρ updated at key x with the value v).

```
Context (assign: var → expr → Σ → option Σ) (assume: expr → bool → Σ → option Σ)
 (assign_sound: ∀ x e σ ρ v,
    ρ ∈ γ(σ) → v ∈ eval_expr e ρ → ρ[x <- v] ∈ γo (assign x e σ))
 (assume_sound: ∀ g σ ρ v,
    ρ ∈ γ(σ) → v ∈ eval_expr g ρ → ρ ∈ γo (assume g (val_is_true v) σ)).
```

The abstract semantics f of a program p performs one execution step from an abstract state π. For every program point n of the program, the new abstract environment at (f π)[n] is computed as the join of all states coming from the predecessors of n. The map predecessors p attaches to each program point n the set of nodes k such that there is an edge in p from k to n. The function mapio applies a function f to each binding in a map to build a map with the results; when f returns None, the binding is removed from the resulting map. The function set_map_reduce applies a function to each elements of a set in parallel, then joins all the results.

```
Context (p: prog).
Definition f (π: Map [ node, Σ ]) : Map [ node, Σ ] :=
 mapio (λ (n: node),
  set_map_reduce (λ k,
   π[k] >>= λ σ,
   match p[k] with
   | Some (IAssn x e _) => assign x e σ
   | Some (IIf g t e) => assume g (t == n) σ
   | _ => None
   end)
 ) (predecessors p).

Definition fs_analyzer (n: nat) (ι: Map [ node, Σ ]) : Map [ node, Σ ] * outcome :=
 analyzer _ fs_leb (λ π, ι ⊔ f π) n ι.

Definition γ_fs (π: Map [ node, Σ ]) : ℘ exec_state := λ e, snd e ∈ γo(π[fst e]).

Theorem fs_analyzer_sound n ι π :
 initial_state ⊆ γ_fs(ι) → fs_analyzer n ι = (π, OK) →
 ⟦p⟧ ⊆ γ_fs(π).
```

The flow-sensitive analyzer is finally defined by specializing the generic analyzer with the function λ π, ι ⊔ f π. This function is the abstract counterpart of the concrete SEM previously defined. It therefore yields a sound analyzer, as stated by the fs_analyzer_sound theorem: if the initial abstract state is a correct approximation of the concrete initial states, the analysis result is a correct abstraction of the program behaviors. The γ_fs concretization relation lifts the concretization γo of abstract environments to flow-sensitive abstract states.

The analysis could be defined as a post-fixpoint of other functions: using λ π, π ⊔ f π would also work. Moreover, if the domain Σ has infinite ascending chains, we could define a widening operator ∇ and compute a post-fixpoint of λ π, π ∇ f π. However, since these two last functions are not known to be monotone, there is no guarantee that their post-fixpoints (if any) are greater than the initial abstract state ι. Therefore, the analyzer should check this fact in order to be sound.

More generally, there are more subtle ways to compute a solution to this analysis problem than iterating the abstract semantics. One of them is described by Bourdoncle [Bou93] along with a suitable widening operator.

Example: definite initialization This flow-sensitive analyzer can be used to define data-flow analyses; for instance, the following one computes, for each program point, the set of variables that are known to be initialized when the execution reaches said point.

In this domain, an abstract value is a set of variables. These sets are ordered by inclusion, the greatest set being the empty one. The join operation is the set intersection. Assignment of variable x with the result of the evaluation of expression e is modeled, in an abstract state m, as adding x to the set m. The assume operator is just the identity.

Finally, to analyze a program, we run the analyzer from an initial state [][1 <- {}] (the empty map in which is added one binding from the node 1 to the empty set {}), which maps the initial node to the greatest element (no variable is (known to be) initialized before the execution starts) and other nodes to nothing (they may be unreachable).

```
Definition di : Type := set var.
Definition di_leb (x y: di) : bool := subset y x.
Definition di_join (x y: di) : di := inter x y.
Definition di_assign (x: var) (e: expr) (m: di) : option di := Some (add x m).
Definition di_assume (g: expr) (b: bool) (m: di) : option di := Some m.

Definition di_analyze (p: prog) (n: nat) : Map [ node, di ] * analyze_t :=
  analyze di di_leb di_join di_assign di_assume p n [][1 <- {}].
```

This abstract domain is specified through the following concretization relation: any variable x in an abstract set a is bound in all environments ρ represented by this set.

```
Definition di_gamma (a: di) (ρ: env) : Prop := ∀ x, x ∈ a ⟶ ρ[x] ≠ None.
```

Given an analysis result, we can perform a safety check and establish, for instance, the following safety property: any program whose analysis result π satisfies the safety check cannot be stuck.

```
Theorem is_safe_not_stuck p n π :
  di_analyzer p n = (π, OK) ⟶
  is_safe p π = true ⟶
  ⟦p⟧ ∩ stuck p ⊆ ∅.
```

The safety check is_safe verifies that the control-flow graph is properly defined (no successor of an instruction is outside the graph) and that all free variables of expressions are definitely initialized when the execution evaluates said expressions. This analyzer coupled with the safety check cannot prove that our fact program is safe: indeed, it is not if the variable x is not initialized in the initial environment. We could refine the analyzer to take into account some additional knowledge about the initial execution state, so as to establish facts like: "the program is safe provided the variable x is initially bound". Notice that the correctness property of the safety analysis does not involve the concretization relation of the abstract domain: it only refers to the semantics of the programming language.

Such an analysis does not need all the power of abstract interpretation. We discuss next a more involved class of analyses.

2.1.8 Non-Relational Value Analysis

This particular class of analyses computes, for each program variable, an abstraction of the value it may hold; for instance, an interval. We will build such analyses on top of the flow-sensitive analyzer that we have just defined.

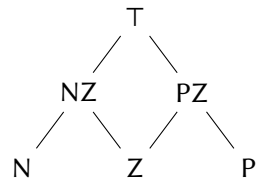
This analyzer is parameterized by the domain of abstract values of type V with a partial order and a join operator. The meaning of abstract values is defined by a concretization relation γ: V $\longrightarrow \wp(Z)$.

The flow-sensitive analyzer is instantiated with the type Σ := Map [var, V], with the convention that variables that are not bound may have any value. We then implement, in addition to the ordering and join operators, the abstract transfer functions assign and assume. To do so, we first define forward and backward evaluations of expressions, relying on abstract forward and backward operators modeling the basic concrete operators.

The forward evaluation function is very similar to its concrete counterpart and enjoys a soundness property: the evaluation of an expression e in an abstract state a yields a result that over-approximates all concrete results of the evaluation of e in any environment represented by a.

The backward evaluation is more interesting. It takes as argument an expression e and an abstraction res of the result of the evaluation of this expression. It is a *reduction*: it receives an abstract state a and tries to return a smaller one after taking into account the information that e is known to evaluate to res.

As an example, suppose a *sign* domain, with six abstract values: Z for zero, P (resp. N) for strictly positive (resp. negative) numbers, and PZ (resp. NZ) for non-negative (resp. non-positive) numbers and $\top$ (read *top*) for all numbers. These values are organized in a lattice as depicted on the right. The abstract backward multiplication operator, when applied to three abstract values x, y and z, returns better approximations for x and y taking into account that the concrete multiplication returns a value represented by z. More specifically, (ab_mul_bw PZ PZ P) returns Some (P, P): the product of two non-negative numbers is non-zero only if both numbers are also non-zero. The call (ab_mul_bw P P Z) returns None: no product of positive numbers can be zero.



Equipped with a backward operator for each primitive of the expression language, we build a backward evaluation function. It works backwards: starting from the result, it applies binary operators until it reaches the expression leaves. To evaluate a compound expression as (EMul e_1 e_2), the backward operator for multiplication is applied first; this yields approximations for the results of the sub-expressions, which are then recursively evaluated. Evaluating a variable sets it in the abstract environment.

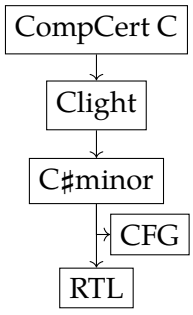
Finally, we can implement the assume function: to take into account that an expression is false when evaluating the else branch of a conditional, the guard is evaluated backwards from the abstraction of the boolean false (i.e., the integer zero). The case of the then branch is more involved, since we have no easy way to abstract all non-zero values. When the guard is a comparison $e_1 < e_2$, we can assume that a negated form of this expression, $e_2 < e_1 + 1$, evaluates to false, and do as in the case of the else branch.

To come back to the example program of Figure 2.2, the value analysis using the sign domain yields in particular the following results: the final value of y is strictly positive, the final value of n is non-negative, and inside the loop (program points 4 and 5), the value of x is strictly positive. This last property is inferred by the backward evaluation of the loop guard, even though x is an input value never set in the program.

2.2 CompCert

CompCert [Ler06] is a full-fledged compiler for the C programming language, targeted at the compilation of safety-critical applications (e.g., fly-by-wire flight control system). It has been formally specified and proved correct within the Coq proof assistant: each intermediate language comes with a formal description of its semantics and, for each compilation pass, a theorem states that this pass does not introduce new behaviors in the compiled program.

Compilation is performed in many steps, as sketched on the right, doing one transformation or optimization at a time, and going through various intermediate representations. The main stages of the front-end are as follows. The source language is called CompCert C, a sibling of C99. Then comes Clight, in which all expressions are pure (i.e., without side effects). The next step is C♯minor, which has a simpler structure (e.g., only one kind of loops) and no overloaded operators. Finally, stack-allocated variables are merged into a single stack block, control is represented as a graph, and expressions are transformed into three-address code: this leads to the RTL language.



This last language of CompCert's front-end is architecture dependent: the instruction set depends on the architecture for which the program is compiled. The back-end, which compiles RTL towards assembly, is less relevant to this document [Ler09].

For the purpose of building static analyzers into CompCert, an additional intermediate representation has been introduced between C♯minor and RTL. It is called CFG[3], is structured as a graph (as RTL) but features a rich expression language (as C♯minor). We will highlight some properties of these languages.

CompCert C

CompCert C deviates from the C99 standard [C99] in various cases. In particular, operations on machine integers are (almost) completely defined in CompCert. Notable exceptions include division by zero, logical shifts by negative quantities. Many more operations have undefined behaviors in C99, for instance overflows in signed arithmetic.

An other important example is the *strict aliasing rule*, that roughly says that no two pointers of different types alias. The following program has undefined behavior in standard C (and its execution leads to an assertion violation when compiled with `gcc` or `clang` at optimization level 2), as in function `check`, pointers `h` and `k` alias but have distinct types. However, in CompCert C, this program is well-defined.

[3] CFG is also a common abbreviation for "control-flow graph"; in this document however, it is used consistently as the name of the CompCert intermediate representation.

```
Inductive expr : Type :=
 | Evar (x: ident)
 | Econst (cst: constant)
 | Eunop (op: unary_operation) (e: expr)
 | Ebinop (op: binary_operation) (ℓ r: expr)
 | Eload (κ: memory_chunk) (addr: expr)
 | Econdition (b ℓ r: expr).

Inductive unary_operation : Type :=
 | Onegint | Oboolval | ...

Inductive binary_operation : Type :=
 | Oadd | Osub | Omul | ...
 | Ocmp (c: comparison)
 | Ocmpu (c: comparison)

Inductive comparison : Type :=
 | Ceq | Cne | Clt | Cle | Cgt | Cge.
```

```
Inductive constant : Type :=
 (* numerical constant *)
 | Ointconst (i: int) | Ofloatconst (f: float)
 (* address of the symbol plus the offset *)
 | Oaddrsymbol (g: ident) (ofs: int)
 (* stack pointer plus the given offset *)
 | Oaddrstack (ofs: int).

Inductive instruction : Type :=
| Iskip (s: node)
| Iassign (x: ident) (e: expr) (s: node)
| Istore (κ: memory_chunk) (a e: expr)(s: node)
| Iifthenelse (g: expr) (s₁ s₂: node)
| Iswitch (e:expr) (tbl: list (int * node)) (s:node)
| Icall (x: option ident) (sig: signature)
    (f: expr) (args: list expr) (s: node)
| Ireturn (r: option expr)
| ...
```

Figure 2.7: Syntax of CFG programs

```
1  void check (int *h, long *k)
2  {
3    *h = 5;
4    *k = 6;
5    assert ( *h == 6 );
6  }
```

```
7  int main (void)
8  {
9    long k;
10   check((int *)&k, &k);
11   return 0;
12 }
```

The CFG Intermediate Language

The syntax of CFG expressions is shown in Figure 2.7. They include reading local variables, constants and arithmetic operations, reading store locations, and conditional expressions (similar to $b\ ?\ e_1 : e_2$ in C). The set of unary and binary operators is quite large so as to handle the diversity of values: machine integers, floats and pointers.

The instructions of the language are arranged in a graph: each program point is bound to an instruction which holds the names of its successors. They feature GOTOs (Iskip), assignments of temporary variables (Iassign), memory writes (Istore), conditional branches (Iifthenelse), multiway branches (Iswitch), and function calls and returns.

The C♯minor Intermediate Language

C♯minor has expressions very similar to CFG; their syntax is therefore not shown. The only difference is that taking the address of a variable is an expression in C♯minor whereas it is a constant in CFG.

The syntax of statements however is very different; it is shown on Figure 2.8. A statement is a tree, and every function is made of only one such statement. This language features

```
Inductive stmt : Type :=
 | Sskip
 | Sset (x: ident) (e: expr)
 | Sstore (κ: memory_chunk) (a e: expr)
 | Sseq (s₁ s₂: stmt)
 | Sifthenelse (g: expr) (s₁ s₂: stmt)
 | Sloop (body: stmt)
 | Sblock : stmt → stmt
 | Sexit (d: nat)
 | Scall (x: option ident) (sig: signature)
     (f: expr) (args: list expr)
 | Sreturn (r: option expr)
 | ...
```

Figure 2.8: Syntax of C♯minor statements

as control structures sequences, conditionals, infinite loops, nested blocks and associated exits, as well as function calls, switches, and GOTOs to named labels (not shown).

Soundness Theorem

The soundness theorems of the compilation passes are of the following form:

$$\forall p : \text{good}, \forall p', \text{Compile}(p) = \lfloor p' \rfloor \rightarrow [\![p']\!] \subseteq [\![p]\!].$$

In this statement, $[\![p]\!]$ is the set of all behaviors of program p, according to the semantics of the language in which p is written, and good is the set of programs whose behavior cannot go wrong.

The behaviors of programs, or their observable semantics, are defined independently of the programming language: this enables to compare the behaviors of programs written in several languages, as is the case of a source program and the corresponding output of the compiler. A program has four possible kinds of behaviors:

- it produces a finite sequence of events and returns a final value;
- it produces a finite sequence of events and its execution continues forever without emitting any further event;
- it produces an infinite stream of events; or
- it produces a finite sequence of events and then goes wrong (i.e., its execution is stuck).

An event is either a call to an external function, a *volatile* memory access (i.e., read from or write to a variable whose declaration is annotated with the C keyword volatile), or the execution of an *annotation* (i.e., a special kind of instruction, whose execution has no other effect that being visible in the semantics).

The soundness property ensures that, for every program whose execution cannot go wrong, the compilation does not introduces new behaviors. The compiler is free however to *remove* behaviors from source programs. Also, it can turn a wrong program (i.e., whose execution can get stuck) into a good one (i.e., whose execution cannot get stuck). Two examples of such programs are given on Figure 2.9. In the first one, the execution gets stuck when evaluating the expression j + 1. The + operator is statically resolved, according to the declared types of its arguments, as the addition on integers. At run-time, variable j

```
1  int
2  main(void)
3  {
4    int i;
5    int *pi = &i;
6    int j = (int)pi;
7    int k = j + 1; /* Error! */
8    return 0;
9  }
```

```
1  int
2  main(void)
3  {
4    int x, y;
5
6    (&y)[0] = 1;
7    (&x)[1] = 2; /* Error! */
8    return y;
9  }
```

Figure 2.9: Two C programs that get *more* defined when compiled by CompCert

evaluates to a pointer, so that the addition cannot proceed. However, in C♯minor, there is only one operator for the addition of integers and pointers: therefore, after a few compilation passes, the program only has good behaviors.

In the program on the right of the figure, the store on line 7 is out-of-bounds. However, during the compilation of this program, CompCert first allocates the two stack variables `x` and `y` contiguously, so that the offending store becomes well-defined as an overwrite of variable `y`. Further optimizations will then remove both variables and yield a very simple program that immediately returns `2`.

An other example is the expression `&(*p)` when `p` is a null pointer. It is undefined at the Clight level but is simplified to `p` in C♯minor[4].

Machine Arithmetic

Most integer types in programming languages are bounded, with arithmetic overflows treated either as run-time errors or by "wrapping around" and taking the result modulo the range of the type. In C♯minor, integer arithmetic is defined modulo 2^N with $N = 32$ or $N = 64$ depending on the operation. Moreover, C♯minor does not distinguish between signed and unsigned integer types: both are just N-bit vectors. Some integer operations such as division or comparisons come in two flavors, one that interprets its arguments as unsigned integers and the other as signed integers; but other integer operations such as addition and multiplication are presented as a single operator that handles signed and unsigned arguments identically.

Memory Model

The concrete memory model of CompCert defines the behavior of the C store (of type `Mem.mem`), that encompasses the heap (dynamically allocated calling `malloc`), the stack (holding local variables that cannot be allocated to machine registers), and global variables.

This store is organized in *blocks*, each having a unique name. A fresh block is allocated by a call to `Mem.alloc` (which never fails) and released by a call to `Mem.free` (which may fail).

A particular byte in a given block can be referred to by an (unsigned) 32 bit machine integer. Therefore, a pointer is a value (`Vptr b ofs`) that represents the address of the byte at offset `ofs` in the block `b`.

[4]This expression is well-defined in C99, and is equivalent to a null pointer [C99, § 6.5.3.2].

An access to the content of a block, either to read from it (Mem.loadv) or to write to it (Mem.storev), is defined by a pointer and a *chunk*, which is one of the following variants.

```
Inductive memory_chunk : Type :=
  | Mint8signed | Mint8unsigned | Mint16signed | Mint16unsigned
  | Mint32 | Mfloat32 | Mfloat64.
```

It describes the type of the data being transferred, its size, and signedness.

Many memory operations may fail, e.g., dereferencing a dangling pointer. This is modeled through a permission system: each pointer (i.e., each block·offset pair) is mapped to a permission. The permissions are, in increasing order:

None dangling pointer;

Nonempty valid pointer, comparisons are permitted (e.g., pointer to a function);

Readable loads are also permitted (e.g., constant static data);

Writable stores are also permitted (e.g., global variable);

Freeable free is also permitted (e.g., stack allocated local variable).

This permission system is very fine-grained: a different permission may be given to every offset in a block. During the execution of a C program, actual permissions do not use all this flexibility. Blocks are like arrays: they have bounds such that there is no permission outside the bounds and all offsets within the bounds have the same permission.

This permission system does not enable modeling const-qualified local variables that are read-only but may be freed on function exit. The Freeable property implies the Writable one. Some other properties cannot be expressed in this model, for instance sizes of nested arrays, as illustrated by the following example. In C99, this program has undefined behavior (the array denoted by a[1] only has five elements), whereas in CompCert it returns 42.

```
1  static int a[4][5];
2
3  int
4  main(void)
5  {
6    a[1][7] = 42; /* Error */
7    return a[2][2];
8  }
```

2.3 Conclusion

Verified construction of sound static analyzers based on abstract interpretation is well understood when applied to small and simple languages [Pic05]. The development of the CompCert compiler has shown that the complexity of programming languages as C is within the scope of formally verified tools. In the following chapters, we describe how we have built and verified static analyzers, on the model of the ones presented in this chapter, for programs written in CompCert intermediate representations: Chapters 3 and 5 present analyses for the CFG language, and Chapter 6 presents an analyzer of C♯minor. This

methodology also applies at the lowest level of programming languages, binary, as shown in Chapter 4.

Chapter 3

Modular construction of static analyzers of a C-like language in Coq

The static analyzer Astrēe, that operates on embedded programs written in C, is an achievement in terms of software engineering. Its soundness is formally assessed in the abstract interpretation framework. Unfortunately, the soundness proof is neither machine-checked nor directly related to the actual implementation.

On the other hand, there have been recently various static analyzers implemented and proved correct within a proof assistant [KN06; Cac+05; VN11; CP10; Nip12; RL12]. However, most of these works apply to simple, ideal, languages. The CompCert compiler verification has shown that providing a machine-checked proof of the soundness of a large software, dealing with the semantics of real-life languages, is a realistic task. Moreover, various components of this compiler are of great interest for implementing a static analyzer and can be reused as-is. In particular, intermediate languages — together with the formal definition of their semantics, defined with respect to a formal memory model [LB08] —, and the (verified) front-end to bring source code (from files) into such representations.

In this chapter, we present the implementation and machine-checked verification of a static analysis for a language close to C in terms of realism and complexity. It is built on CompCert and operates on one of its intermediate languages, CFG. This demonstrates the usability of theorem proving in static analysis of real programs. This implementation focuses on the modularity of its design, with precise interfaces to promote flexibility (various components may be composed in various ways to yield a range of analyses each with its own precision and cost), extensibility (it is easy to replace a component or to add a new one) and reuse (some components may be reused as is or with little modification, e.g., in the other analyzers presented in the following chapters). In addition, the proof effort benefits from such a modularity: each component has a clear interface and a precise specification; the proof of each component is therefore independent of the proofs of the other components.

The implementation of such a value analysis faces the following difficulties.

Machine Arithmetic The object language operates on machine integers rather than on ideal unbounded integers. The analysis has to be sound (and hopefully precise) even in case of arithmetic overflows.

Pointer Arithmetic Low-level languages, as every intermediate representation in CompCert, allow complex expressions to describe memory locations, and more generally feature a subtle arithmetic of pointer values. This mandates the use of dedicated techniques. In

this chapter, we only set down the required architecture along with an implementation that do not track the contents of the memory; the actual implementation of a precise domain devoted to memory will be discussed in Chapter 5.

Extensibility and Reuse All components of the analyzer shall be clearly separated, with precise interfaces and specifications. Therefore, further components may be developed and proved independently, and optionally plugged into the analyzer. In addition, some components that are of independent interest — e.g., a numerical abstract domain — can be reused in different analyzers, as we will do, for instance, in the next chapter.

Mitigation of the proof effort We have at hand various techniques to assess various properties of the analyzer: proof of the implementation, proof of a validator that treats the implementation as an untrusted oracle, testing the implementation on some inputs. Both first methods can be applied to critical properties (soundness), whereas the last method can be applied to non critical properties (termination, precision, cost as analysis time and memory usage). Each method having its own advantages and drawbacks. In particular, quantitative and hardly defined properties (as performance) are empirically evaluated by comparing the behaviors of different implementations on the same inputs.

Overview of the analysis The static analysis that is described in this chapter is a value analysis. It means that it computes an over-approximation of the values held in the variables of the analyzed program. It is flow-sensitive: there is one result for every program-point; the result for a given variable at a given program-point represents at least all possible values of this variable when the execution reaches this program-point. It operates on the CFG intermediate language, described in Section 2.2.

The analyzer is built as sketched on Figure 3.1 (white boxes represent the interfaces of each component; red boxes are the implementations of these interfaces that are discussed in this chapter; dotted boxes are other implementations that will be incorporated in following chapters to enhance the precision of the analyzer). Most of the components share a common interface of abstract domain (adom) that we describe in Section 3.1. The main component, the CFG iterator, is parameterized by the memory abstraction. Our implementation uses an external solver, whose result is validated. This iterator is the subject of Section 3.2. The implementation of the memory abstraction is parameterized by a relational numerical domain. The interface of numerical domains is given in Section 3.3, as well as methods to build them, and in particular how to combine them. Finally, an experimental evaluation of the analyzer is presented in Section 3.5.

A short version of this chapter has been published at the international Static Analysis Symposium (SAS) [Bla+13].

3.1 Abstract Domain Library

This section describes the library we have designed to represent our abstract domains. First, it defines generic abstract domains. Then, it details the interval abstract domain. Last, it explains how to combine abstract domains.

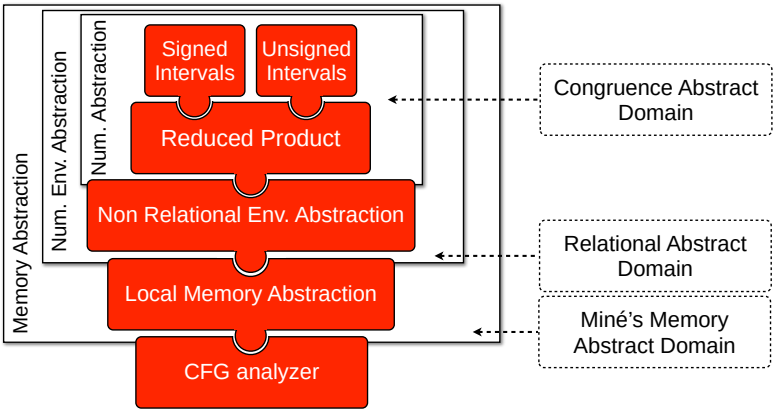


Figure 3.1: Overview of the architecture of the analyzer

3.1.1 Abstract Domain Interface

The basic structure of an abstract domain is a (weak) lattice with widening, as described by the following signature.

```
Class weak_lattice (A: Type) : Type := {
  leb: A ⟶ A ⟶ bool;
  top: A;
  join: A ⟶ A ⟶ A;
  widen: A ⟶ A ⟶ A
}.

Notation "x ⊑ y" := (leb x y).
Notation "⊤" := top.
Notation "x ⊔ y" := (join x y).
Notation "x ∇ y" := (widen x y).
```

Figure 3.2: Signature of weak lattices with widening

Here, A is the type of abstract values. A record of type (weak_lattice A) holds various operators. Values of type A are partially ordered by a comparison operator called leb; a distinguished element of type A is called top and is meant to represent the greatest element of the lattice; two binary operators join and widen compute upper bounds of their arguments, the first being expected to be more precise than the last, whereas the last should ensure termination of the analysis, as discussed in Section 2.1.6.

The use of type-classes allows in particular to define overloaded notations for the various components of the weak_lattice record. For instance, the greatest element of *any* lattice will be written ⊤.

Such a lattice is specified with respect to a concretization relation, which relates the type A of abstract values to the type B of concrete values (see Figure 3.3). The adom record contains only three properties: the monotonicity of the gamma operator, the soundness of the top element and the soundness of the least upper bound operator join. We do not provide formal proof relating the abstract order with top or join. Indeed any *weak-join* will be suitable here. The lack of properties about the widening operator is particularly surprising at first sight. In fact, as we will explain in Section 3.2, the widening operator is used only during fixpoint iteration and this step is validated *a posteriori*. Thus, only

```
Class gamma_op (A B: Type) : Type := γ : A → ℘(B).

Record adom (A B:Type) (WL:weak_lattice A) (G:gamma_op A B) : Prop := {
 (* monotonicity of gamma *)
 gamma_monotone: ∀ a₁ a₂, a₁ ⊑ a₂ → γ a₁ ⊆ γ a₂;
 (* top over-approximates any concrete property *)
 gamma_top: ∀ b, b ∈ γ(⊤);
 (* join over-approximates concrete union *)
 join_sound: ∀ a₁ a₂, γ a₁ ∪ γ a₂ ⊆ γ (a₁ ⊔ a₂)
}.
```

Figure 3.3: Specification of weak lattices

the result of this iteration step is verified and we don't need a widening operator in the verification process.

3.1.2 Example of Abstract Domain: Intervals

Our value analysis operates over compositions of abstract domains. One of the most basic abstract domains is the domain of intervals [CC76]. Using an interval to represent a set of machine integers is not straightforward. Indeed, machine integers form a circle, whereas an interval is a segment on a straight line. Also, depending on the operator applied to a machine integer, it may be interpreted as signed or unsigned. Here we focus on the signed interpretation; this signedness problem will be dealt with in Section 3.3.3.

Figure 3.4 defines an abstract domain of intervals; one element of this domain represents the machine integers whose *signed* interpretation falls into this range. This instance is called signed_itv_adom. The definitions of leb and join are standard: the order corresponds to interval inclusion and the union of two intervals is their convex hull (the boolean operator x <=? y tests if x is less than or equal to y). The widen operator is parameterized by two functions, next_smaller and next_larger that enable a family of widening strategies. In particular, such a strategy may depend on the very program being analyzed. The basic widening operator for the domain of intervals over-approximates a non-stable bound by the largest (resp. smallest) integer. This is captured by a next_larger function (resp. next_smaller) that always returns Int.max_signed (resp. Int.min_signed). A refined strategy would try several thresholds between the current bound and the limit integer. For instance, (next_larger m) may return the smallest power of two larger than m. Also, to capture the frequent pattern of constant for-loops, e.g., when folding an array tab of constant size N, next_larger may return a value that appears as a literal constant in the analyzed program. For instance, when analyzing the following C snippet, trying N as a widening threshold would lead to a correct interval for the variable i.

```
1     for (int i = 0; i < N; ++i) { /* ... */ tab[i] /* ... */ }
```

An interval represents the range of the signed interpretation of a machine integer, as defined by the relation sgamma. Thus, top is defined as the largest interval with bounds Int.min_signed and Int.max_signed. The concretization is defined as follows. A machine integer n belongs to the concretization of an interval itv iff (Int.signed n) belongs to itv.

```
Record itv := { min: Z; max: Z }.

Instance signed_itv_wl : weak_lattice itv := {
 leb := λ i₁ i₂,
   let (min₁, max₁) := i₁ in
   let (min₂, max₂) := i₂ in
    (min₂ <=? min₁) && (max₁ <=? max₂);
 top := { min := Int.min_signed; max := Int.max_signed };
 join := λ i₁ i₂,
   let (min₁, max₁) := i₁ in
   let (min₂, max₂) := i₂ in
   { min := Z.min min₁ min₂; max := Z.max max₁ max₂ };
 widen := λ i₁ i₂,
   let (min₁, max₁) := i₁ in
   let (min₂, max₂) := i₂ in
   { min := if min₁ <=? min₂ then min₁ else next_smaller min₂
   ; max := if max₂ <=? max₁ then max₁ else next_larger max₂ }
}.

Instance sgamma : gamma_op itv int := λ i v,
 min i <= Int.signed v <= max i.

Lemma signed_itv_adom : adom itv int signed_itv_wl sgamma.
Proof. (* proof omitted here *) Qed.
```

Figure 3.4: Domain of intervals, abstracting signed machine integers

We also define a variant of this domain with a concretization using an unsigned interpretation of machine integers: (λ i n, min i ≤ Int.unsigned n ≤ max i). As explained in Section 3.3, combining both domains recovers some precision that may be lost when using only one of them.

The itv record type provides only lower and upper bounds of type Z. Using the expressiveness of the Coq type system, we could choose to add an extra field requiring a proof that min ≤ max holds. While elegant at first sight, this would be rather heavyweight in practice, since we must provide such a proof every time we build a new interval. For the kind of proofs we perform, if such a property was required, we would generally have an hypothesis of the form n ∈ γ(i) in our context and it would imply that min i ≤ max i holds.

3.1.3 Abstract Domain Functors

Our library provides several functors that build complex abstract domains from simpler ones.

Direct Product A first example is the product (adom (A * A′) B WLx Gx) of two abstract domains (adom A B WL G) and (adom A′ B WL′ G′), where the lattice structure WLx is straightforward: maximal element, join and widening are defined component-wise, and two

```
Instance lift_bot_wl (A: Type)
 (WL:weak_lattice A)
 : weak_lattice (A+⊥) := {

 top := NotBot(⊤);

 leb x y :=
    match x, y with
    | Bot, _ => true
    | _, Bot => false
    | NotBot x, NotBot y => x ⊑ y
    end;

 join x y :=
    match x, y with
    | Bot, _ => y
    | _, Bot => x
    | NotBot x, NotBot y => NotBot (x ⊔ y)
    end;
 widen x y :=
    match x, y with
    | Bot, _ => y
    | _, Bot => x
    | NotBot x, NotBot y => NotBot (x ∇ y)
    end
}.
```

Figure 3.5: Lattice structure of a domain with bottom

pairs are ordered if and only if both of theirs components are respectively ordered:

```
Definition leb : (A * A′) ⟶ (A * A′) ⟶ bool :=
 λ x y, let (x, x′) := x in let (y, y′) := y in x ⊑ y && x′ ⊑ y′.
```

The concretization γx of a pair (a, a′) is the intersection (γ a) ∩ (γ a′) of their concretizations.

Lifting a Bottom Element A bottom element is not mandatory in our definition of abstract domains because some sub-domains do not necessarily contain one. For instance, the domain of intervals does not contain such a least element. Still in our development, the bottom element plays a specific and important role since it represents the empty set of concrete values, i.e., any contradiction (for instance, unreachable states). We hence introduce a polymorphic type A+⊥ that lifts a type A with an extra bottom element called Bot[1].

```
Definition botlift (A:Type): Type := Bot | NotBot (x:A).
Notation A+⊥ := (botlift A).
```

We then define a simple functor lift_bot that lifts any domain (adom A B) on a type A to a domain on A+⊥ (see Figure 3.5). The maximal element is the lifted maximal element of the original domain A. The partial order extends the original one such that the new element is actually minimal. The join and widening operators extend the original ones such that the new element is a unit.

In this new domain, the concretization function extends the concretization of the input domain with γ Bot = ∅. We can then prove that the lifted lattice satisfies the specification, provided the original lattice is correct.

```
Instance lift_gamma (A B: Type) (G:gamma_op A B) : gamma_op (A+⊥) B :=
 λ x, match x with Bot => ∅ | NotBot x => γ(x) end.
```

[1]The lifted data-type is an instance of the error monad; we will use the notation "do_bot a <- e; f(a)" for its *bind* operator.

```
Lemma lift_bot (A B: Type) WL G (D: adom A B WL G)
  : adom (A+⊥) B (lift_bot_wl WL) (lift_gamma G).
Proof. (* proof omitted here *) Qed.
```

Finite Reduced Map Finite maps are a pervasive data structure: they are used to bind abstract states to program points, abstract values to program variables, and so on. When building abstract domains, such maps are generally used to represent functions. We thus provide a generic functor, parameterized by an abstract domain of type adom A B WL G, i.e., a domain in which abstract elements have type A and represent sets of values of type B. It then builds an abstract domain whose abstract elements have type Map [K, A] (for some type K of keys) and represent sets of functions of type K → B as described by the following definitions.

```
Definition get (m: Map [ K, A ]) (r: K) : A :=
  match m[r] with None => ⊤ | Some i => i end.

Instance gamma : gamma_op t (K → B) :=
  λ m rs, ∀ r, rs r ∈ γ (get m r).
```

Keys that are not bound in the abstract map are implicitly bound to the greatest element in A. Therefore, bindings to this element are lazily represented and we ensure that no binding to it is actually present in the map. This invariant could be enforced by typing, using a method similar to the above bottom-lifting.

There is a variant of this domain in which keys that are not bound in the map implicitly represent the least element in A. This variant is useful to represent flow-sensitive information, i.e., when keys are program points. In that case, program points that are not bound are known to be unreachable, i.e., dead code.

3.2 Fixpoint Resolution

As many data flow analyses, our value analysis can be turned into the fixpoint resolution of an equation system on a lattice. CompCert already provides a classical Kildall iteration framework [Kil73] to iteratively find the least fixpoint of an equation system. But using such a framework is impossible here for two reasons. First, the lattice of bounded intervals contains very long ascending chains that make standard Kleene iterations too slow. Second, the non-monotonic nature of widening and narrowing makes fixpoint iteration sensible to the iteration order of each equation.

We have therefore designed a new fixpoint resolution framework that relies on the general iteration techniques defined by Bourdoncle [Bou93]. First, Bourdoncle provides a strategy computation algorithm based on Tarjan's algorithm to compute strongly connected subcomponents of a directed graph and find loop headers for widening positioning. This algorithm also sorts each strongly connected subcomponent in order to obtain an iteration strategy that iterates inner loops until stabilization before iterating outer loops. Bourdoncle then provides an efficient fixpoint iteration algorithm that iterates along the previous strategy and requires a minimal number of abstract order tests to detect convergence.

```
Definition check_fxp t instruction wl transfer entry code init (fxp: node ⟶ t+⊥) : bool :=
 NotBot init ⊑ fxp entry
 &&
 map_forall code (λ pc ins,
  match fxp pc with
  | NotBot ab => List.forallb (λ x, let (pc', tf) := x in tf ab ⊑ fxp pc') (transfer pc ins)
  | Bot => true
  end).
```

Figure 3.6: Fixpoint checker

This algorithm relies on advanced reasoning in graph theory and formally verifying it would be highly challenging. This frontal approach would also certainly be too rigid because widening iteration requires several heuristic adjustments to reach a satisfactory precision in practice (loop unrolling, delayed widenings, decreasing iterations). We have therefore opted for a more flexible verification strategy: Bourdoncle strategies and fixpoints are computed by an external tool (represented by the function called get_extern_fixpoint) and we only formally verify a fixpoint checker (called check_fxp).

The external tool has the following type.

```
Parameter get_extern_fixpoint (t: Type) (instruction: Type) (wl: weak_lattice t)
 (transfer: node ⟶ instruction ⟶ list (node * (t ⟶ t +⊥)))
 (P: Map [ node, instruction ]) (entry: node)
 (init: t)
 : node ⟶ t+⊥.
```

It is parameterized by an abstract domain (type t and lattice wl), and the instruction set (type instruction and abstract semantics transfer). The transfer function defines the abstract semantics of the CFG programming language: given a node and the instruction at this node, it yields a list of possible successors with the corresponding abstract transfer function. The output of the oracle is flow-sensitive: at each node of the program is attached an abstract value. Nodes that are not bound in the result are unreachable.

This external oracle is not trusted. Instead, its output is validated after each call. The code of the checker is given on Figure 3.6. Given a program to analyze and the putative solution fxp, the checker ensures two properties:

1. the solution for the entry node is larger than the initial abstract state init; and
2. for every program point pc, at which the instruction is ins, the solution at this program point is either Bot (i.e., this program point is unreachable), or NotBot ab such that any possible abstract step from ab leads to an abstract state that is actually predicted by the solution.

Our fixpoint analyzer is defined below. Given an abstract domain ab, the transfer functions transfer, a program P, its entry point entry, and the initial abstract value init, it calls the external oracle, and checks its answer. If the check succeeds, this answer is returned; otherwise, a trivial result is returned.

```
Definition solve_pfp (ab: adom t B wl G)
  (transfer: node → instruction → list (node * (t → t+⊥)))
  (P: Map [ node, instruction])
  (entry: node)
  (init: t)
  : node → t+⊥ :=
  let fxp := get_extern_fixpoint ab transfer P entry init in
  if check_fxp ab transfer P entry init fxp
  then fxp
  else (λ _, ⊤).
```

The verification of the fixpoint checker yields the following property: the concretization of the result of the solve_pfp function is a post-fixpoint of the concrete transfer function. That is, given the analysis result fxp, for each node pc of the program, applying the corresponding transfer function tf to the analysis result yields an abstract value included in the analysis result.

```
Lemma solve_pfp_postfixpoint: ∀ ab entry P transfer init fxp,
  fxp = solve_pfp ab P entry transfer init →
  ∀ pc i, P[pc] = ⌊i⌋ →
  ∀ pc′ tf,
  In (pc′, tf) (transfer pc i) →
  γ(tf (fxp pc)) ⊆ γ(fxp pc′).
Proof. (* proof omitted here *) Qed.
```

The proof goes as follows. First we examine the outcome of the fixpoint checker. If it failed, the analysis result is $\top$, whose soundness is one of the properties of all abstract domains (gamma_top in Figure 3.2). Otherwise, every possible step has been examined by the checker: in particular, for the step from pc to pc′, the checker enforces that the abstract states are correctly ordered: tf(fxp pc) $\sqsubseteq$ fxp pc′. Then, the monotonicity of the concretization yields the conclusion.

The actual soundness proof of the analysis now relies on the soundness of the abstract semantics, as represented by the transfer function given to the solver. Our implementation of this function is built from a memory abstraction: a type with abstract transformers that model the CFG instructions. This memory abstraction is, in turn, parameterized by a numerical abstraction, which is the topic of the next section.

3.3 Numerical Abstraction

Following the design of the Astrēe analyzer [Cou+05], our value analysis is parameterized by a numerical abstract domain that is unaware of the C memory model. We first present the interface of abstract numerical environments, then how we abstract numerical values in order to build non relational abstract environments. Finally, we show concrete instances of numerical domains and how they can be combined.

3.3.1 Abstraction of Numerical Environments

The first interface captures the notion of numerical environment abstraction. This interface matches with most implementations of a relational abstract domain [CH78; Min04] on machine integers. It relies on a numerical expression language (type nexpr) defined in Figure 3.7. These expressions are parameterized by the type var of their variables. The unary and binary operations are the same as in CompCert's CFG language. The constant constructor NOintunknown denotes any machine integer; it enables to write non-deterministic expressions.

These expressions are associated with a big-step operational semantics eval_nexpr of type (var → int) → nexpr → ℘(int). This semantics depends on a valuation ρ of the variables. It returns a set of machine integers as expressions are non-deterministic and can be stuck. Its formal definition is given on Figure 3.8.

The interface ab_env of relational numerical domains, given in Figure 3.9, includes an abstract domain id_adom with a concretization id_gamma to sets of functions from variables to machine integers, and three operators: range, assign, and assume. Each operator comes with a correctness property, defined in terms of the concretization relation. The range e ab query returns two intervals from the interval domain previously defined (one for each signedness) that represent all values that expression e may evaluate to given any valuation ρ in the concretization of ab. The assign x e ab transformer returns an abstract element that represents functions in which variable x is now bound to the result of the evaluation of expression e. The assume e ab transformer refines the abstract element ab to take into account that expression e evaluates to true.

3.3.2 Building Non-relational Abstraction of Numerical Environments

Implementing a fully verified relational abstract domain is a challenge in itself and it is not the topic of this dissertation. Advances on this subject can be found in the work of Fouilhē *et al.* [FB14; FMP13]. We implement instead the previous interface with a standard non relational abstract environment of the form var → $V^\sharp$ where $V^\sharp$ abstracts numerical values. The notion of abstraction of numerical values is captured by the interface num_dom shown on Figure 3.10. It is defined as a carrier t, an abstract domain structure num_adom and several abstract transformers. Some operators are forward ones: they provide properties about the output of an operation. For instance, the operator const builds an abstraction of a single value. Some operators are backward ones: given some properties about the input and expected output of an operation, they provide a refined property about its input. These backward operators are needed to implement a precise assume transformer (see § 2.1.8). Each operator is required to come with a soundness

```
Variable var : Type.

Inductive nconstant : Type :=
| NOintconst: int → nconstant
| NOintunknown : nconstant.

Inductive nexpr : Type :=
| NEvar (v: var)
| NEconst (cst: nconstant)
| NEunop (op: unary_operation) (e: nexpr)
| NEbinop (op: binary_operation) (e₁ e₂: nexpr)
| NEcondition (b: nexpr) (e₁ e₂: nexpr).
```

Figure 3.7: Syntax of numerical expressions (nexpr)

```
Definition eval_nconstant (c: nconstant) : ℘(int) :=
 match c with
 | NOintconst i => {{ i }}
 | NOintunknown => λ _, True
 end.

Definition Ntrue: int := Int.one.
Definition Nfalse: int := Int.zero.

Definition eval_nunop (op: unary_operation) (i: int) : ℘(int) :=
 match op with
 | Ocast8unsigned => {{ Int.zero_ext 8 i }}
 | Onegint => {{ Int.neg i }}
 | Onegf => ∅
 | ...
 end.

Definition eval_nbinop (op: binary_operation) (i_j: int * int) : ℘(int) :=
 let (i, j) := i_j in
 match op with
 | Oadd => {{ Int.add i j }}
 | Odiv =>
    if Int.eq j Int.zero || Int.eq i (Int.repr Int.min_signed) && Int.eq j Int.mone
    then ∅
    else {{ Int.divs i j }}
 | Oaddf => ∅
 | ...
 end.

Variable ρ: var ⟶ int.
Fixpoint eval_nexpr (ne: nexpr) : ℘(int) :=
 match ne with
 | NEvar v => {{ ρ v }}
 | NEconst cst => eval_nconstant cst
 | NEunop op e =>
    Union (eval_nexpr e) (eval_nunop op)
 | NEbinop op e₁ e₂ =>
    Union (eval_nexpr e₁ × eval_nexpr e₂) (eval_nbinop op)
 | NEcondition b e₁ e₂ =>
    Union (eval_nexpr b) (λ k, eval_nexpr (if Int.eq Int.zero k then e₂ else e₁))
 end.
```

Figure 3.8: Semantics of numerical expressions

```
Inductive signedness := Signed | Unsigned.
Definition ints_in_range (r: signedness → itv+⊥) : ℘(int) := (γs (r Signed)) ∩ (γu (r Unsigned)).
Class ab_env (var t: Type) {E:EqDec var} : Type := {
  id_wl:> weak_lattice t;
  id_gamma:> gamma_op t (var → int);
  id_adom:> adom t (var → int) id_wl id_gamma;

  range: nexpr var → t → signedness → Interval.itv+⊥;
  assign: var → nexpr var → t → t+⊥;
  assume: nexpr var → t → t+⊥;

  range_correct: ∀ e ρ ab,
    ρ ∈ γ ab →
    eval_nexpr ρ e ⊆ ints_in_range (range e ab);
  assign_correct: ∀ x e ρ n ab,
    ρ ∈ γ ab →
    n ∈ eval_nexpr ρ e →
    (upd ρ x n) ∈ γ (assign x e ab);
  assume_correct: ∀ e ρ ab,
    ρ ∈ γ ab →
    Ntrue ∈ eval_nexpr ρ e →
    ρ ∈ γ (assume e ab)
}.
```

Figure 3.9: Signature of relational numerical domains

```
Class num_dom (t:Type) := {
 (* abstract domain structure *)
 num_wl: weak_lattice t;
 num_gamma: gamma_op t int;
 num_adom : adom t int num_wl num_gamma;
 (* over-approximation of the concrete intersection *)
 meet: t ⟶ t ⟶ t+⊥;
 meet_sound: ∀ x y, (γ x) ∩ (γ y) ⊆ γ (meet x y);
 range: t ⟶ signedness ⟶ itv+⊥; (* signed/unsigned range *)
 range_sound: ∀ x:t, γ x ⊆ ints_in_range (range x);
 const: int ⟶ t;
 const_sound: ∀ n, n ∈ γ(const n);
 forward_unop: unary_operation ⟶ t ⟶ t+⊥;
 forward_unop_sound: ∀ op x,
  Eval_unop op (γ x) ⊆ γ (forward_unop op x);
 forward_binop: binary_operation ⟶ t ⟶ t ⟶ t+⊥;
 forward_binop_sound: ∀ op x y,
  Eval_binop op (γ x) (γ y) ⊆ γ(forward_binop op x y);
 backward_unop: (* omitted *); backward_unop_sound: (* omitted *);
 backward_binop: binary_operation ⟶ t ⟶ t ⟶ t ⟶ (t * t)+⊥;
 backward_binop_sound: ∀ op x y z i j k,
  k ∈ eval_binop op i j ⟶ i ∈ γ x ⟶ j ∈ γ y ⟶ k ∈ γ z ⟶
  (i, j) ∈ γ(backward_binop op x y z).
}.
```

Figure 3.10: Signature of non-relational numerical domains

proof.

We also implement a functor that lifts any abstraction of numerical values into a numerical environment abstraction. It relies on the functor for finite reduced maps that we have presented at the end of Section 3.1.3.

```
NonRelDom.make(t): num_dom t ⟶ ab_env ((Map [ var, t])+⊥)
```

The most advanced operator in this functor is the assume function. It relies on a backward abstract semantics of expressions.

```
Fixpoint backward_expr (e: nexpr) (ab: (Map [ var, t])+⊥) (itv: t) : (Map [ var, t])+⊥ :=
 match e with
  | ...
  | NEcond b ℓ r =>
     (backward_expr b (backward_expr r ab itv) (const Nfalse))
     ⊔
     (backward_expr b (backward_expr ℓ ab itv)
        (backward_unop Oboolval (eval_expr b ab) (const Ntrue)))
 end.
```

We just show and comment the case of conditional expressions. Given such an expression NEcond b ℓ r, an abstract environment ab and the expected value itv of this expression, we explore the two branches of the condition. In one case, the condition b evaluated to Nfalse and the *right* branch r evaluated to itv. In the other case, the condition b evaluated to anything whose boolean value is Ntrue and the *left* branch ℓ evaluated to itv. Then we have to consider that any of the two branches might have been taken, hence the join.

Equipped with such backward operators, the analysis is then able to deal with complex conditions like the following:

```
if (0 <= x && x < y && y < z && z < t && t < u && u < v && v < 10).
```

When analyzing the true branch of this if, it is sound to assume that the condition holds. The backward operator will propagate this information and infer one bound for each variable. Since backward evaluation of conditions goes right to left, the following bounds are inferred: $v < 10$, $u < 9$, $t < 8$, $z < 7$, $y < 6$, and $0 \leq x < 5$. Unfortunately, no information is propagated from left to right. However applying again the assume function does propagate information between the various conditions. Iterating this process finally yields the most precise intervals for all variables involved in this condition.

Notice that inferring such precise information is possible thanks to the availability of complex expressions in the analyzed CFG program. Compare for example with Frama-C which, prior to any analysis, and as part of the parsing phase, destructs boolean operations into nested ifs; it is thus unable to give both bounds for each variable. Nevertheless, it is possible to recover similar precision in absence of complex expressions, thanks to dedicated relational symbolic domains [Bla+03; Jou+15]. This is also more general: good precision can be achieved even if the original source code does not feature the rich expressions.

3.3.3 Abstraction of Numerical Values: Instances and Functor

We gave two instances of the numerical value abstraction interface: the intervals of signed integers and the intervals of unsigned integers. In both domains, intervals are represented using unbounded integers (type Z), but we have to take into account that they model bounded integers that wrap around on overflows. Therefore, we systematically test if an overflow may occur and fall back to top when we can't prove the absence of overflow. This logic is implemented in the repr operator, which is systematically used to implement abstract operators, as exemplified by the definition of the addition of intervals.

```
Definition repr  (i: itv): itv := if i ⊑ ⊤ then i else ⊤.
Definition add (i j: itv): itv := repr {| min := min i + min j; max := max i + max j |}.
```

We also rely on a reduction operator when the result of an operation may lead to an empty interval. Since our representation of intervals contains several elements with the same (empty) concretization, it is important to always use a same representative for them.[2] This function is in particular used in the implementation of backward operators (as backward_lt below) that are reductions.

```
Definition reduce (ℓ u:Z): itv+⊥ :=
  if ℓ <=? u then NotBot {| min := ℓ ; max := u |} else Bot.
```

[2]Otherwise the analyzer may encounter two equivalent values without noticing it and lose precision.

```
Definition backward_lt (i j: itv): itv+⊥ * itv+⊥ :=
  (meet i (reduce Int.min_signed (max j - 1)), meet j (reduce (min i + 1) Int.max_signed)).
```

At run-time, there are no *signed* or *unsigned* integers; there are only *machine* integers that are bit arrays whose interpretation may vary depending on the operations they undergo. Therefore choosing one of the two interval domains may hamper the precision of the analysis. Consider the following example C program, in which `(*)` denotes a boolean condition unknown to the analyzer and `INT_MAX` is the maximal signed integer that is called Int.max_signed in CompCert.

```
1  int main(void) {
2    signed s; unsigned u;
3    if (*) u = INT_MAX; else u = INT_MAX + 1;
4    if (*) s = 0; else s = -1;
5    return u + s;
6  }
```

At the end of line 3, an unsigned interval can exactly represent the two values that the variable u may hold. However, the least signed interval that contains them both is top. Similarly, at the end of line 4, a signed interval can precisely approximate the content of variable s whereas an unsigned interval would be extremely imprecise. Moreover, comparison operations can be precisely translated into operations over intervals (e.g., intersections) only when they share the same signedness. Therefore, so as to get as precise information as possible, we need to combine the two interval domains. This is done through reduction.

To combine abstractions of numerical values in a generic and precise way, we implement a functor that takes two abstractions and a sound reduction operator and returns a new abstraction based on their reduced product.

```
Instance reduced_product A₁ A₂ {N₁:num_dom A₁} {N₂:num_dom A₂}
 (P: reduction A₁ A₂) : num_dom (A₁ * A₂) := (* omitted definition *).
```

A reduction is made of an operator ρ and a proof that this operator is a sound reduction.

```
Class reduction A₁ A₂ : Type := ρ: A₁+⊥ ⟶ A₂+⊥ ⟶ (A₁ * A₂)+⊥.

Class reduction_correct A₁ A₂ B
 (G₁: gamma_op A₁ B) (G₂: gamma_op A₂ B) (R: reduction A₁ A₂) :=
 reduction_spec: ∀ (a: A₁+⊥) (b: A₂+⊥), γ a ∩ γ b ⊆ γ (ρ a b).
```

Each operator of this functor is implemented by first using the operator of both input domains and then reducing the result with ρ. We hence ensure that each encountered value is systematically of the form (ρ a b) but we do not prove this fact formally, avoiding the heavy manipulation of quotients. Note also that, for soundness purposes, we do not need to prove that reduction actually reduces (i.e., returns a lower element in the abstract lattice).

In the next section, we discuss how the numerical domains that we have built in this section are connected to the iterator defined in the previous section.

3.4 Memory Abstraction

The last layer of our modular architecture connects the CFG abstract interpreter with numerical abstract domains. It fills the gap between the CFG constructs that manipulate memory locations through pointers and the numerical domain that operates over a set of identified variables. The memory domain aims at translating every C feature into useful information in the numerical world. On the interpreter side, the interface with this *abstract memory model* is called mem_dom. It provides a basic domain structure, a query operator range that returns an interval for each program variable, and four abstract transformers, forget, assign, store and assume.

Our final analyzer (value_analysis, whose signature is given below) is parameterized by a structure of this type. The analysis of a program returns a function that for each node returns either Bot, meaning that this node is unreachable, or some function that attaches an interval to each local variable and each signedness.

```
Definition value_analysis (t:Type) : mem_dom t → program →
  node → (ident → sign_flag → Interval.itv+⊥)+⊥.
```

We first present the specification of this domain then describe a basic implementation that do not track the contents of the memory.

3.4.1 Specification

The signature of this domain is given in Figure 3.11. It features, as every other domain, a sound lattice structure. The query operator range, as before, provides two intervals for each local variable. The four transformers forget, assign, store and assume model the basic operations of the CFG language: updating a local variable or a memory location, and conditional execution. Indeed, the transfer function to be given to the external fixpoint solver (see § 3.2) is defined as shown on Figure 3.12 (given some abmem implementing the interface mem_dom), mapping each CFG instruction to the corresponding abstract semantics.

These transformers are specified as over-approximations of their concrete counterparts (with a capital initial letter) that operate on *sets* of concrete states (defined as strongest post-conditions). For instance, a concrete state em′ is in the set (Assign x q E) if and only if em′ is the result of updating some concrete state (e, m) in E at local variable x with a value v that results from the evaluation of expression q in this concrete state.

```
Definition Assign (x: positive) (q: expr) (E: ℘ memory) : ℘(memory) :=
  λ em′, ∃ e m, (e, m) ∈ E ∧
  ∃ sp, ∃ v,
    eval_expr ge (Vptr sp Int.zero) e m q v ∧
    em′ = (PTree.set x v e, m).
```

Before we describe an implementation of this interface, we describe how to combine the soundness of a memory domain with the soundness of the iterator to yield the final soundness theorem of the analysis. This theorem is stated on Figure 3.13, for a given sound memory domain, not shown here. It means that given a program p, for each reachable state of this program, at program point pc in function f, the analysis result for this program

```
Record mem_dom (t:Type) := {
  (* abstract domain with concretization
     to local environment and global memory *)
  mem_wl : weak_lattice t;
  mem_gamma: gamma_op t (env * mem);
  mem_adom: adom t (env * mem) mem_wl mem_gamma;
  (* consult the range of a local variable *)
  range: t ⟶ (ident ⟶ signedness ⟶ itv+⊥);
  range_sound: ∀ ab e m,
  (e,m) ∈ γ ab ⟶
  ∃ ab', ab = NotBot ab' ∧
  ∀ x, match e ! x with Some (Vint i | Vptr _ i) => i ∈ ints_in_range (range ab' x)
     | _ => True end;
  (* project the value of a local variable *)
  forget: ident ⟶ t ⟶ t+⊥;
  forget_sound: ∀ x ab, Forget x (γ ab) ⊆ γ (forget x ab);
  (* assign a local variable *)
  assign: ident ⟶ expr ⟶ t ⟶ t+⊥;
  assign_sound: ∀ ge x e ab, Assign ge x e (γ ab) ⊆ γ (assign x e ab);
  (* assign a memory cell *)
  store: memory_chunk ⟶ expr ⟶ expr ⟶ t ⟶ t+⊥;
  store_sound: ∀ ge κ a e ab, Store ge κ a e (γ ab) ⊆ γ (store κ a e ab);
  (* assume an expression evaluates to non-zero value *)
  assume: expr ⟶ t ⟶ t+⊥;
  assume_sound: ∀ ge e ab, Assume ge e (γ ab) ⊆ γ (assume e ab)
}.
```

Figure 3.11: Signature of abstract memory domains

```
Definition transfer (n : node) (ins: instruction) : list (node * (abmem ⟶ abmem+⊥)) :=
  match ins with
  | Iskip s => (s, λ ab, NotBot ab) :: nil
  | Iassign x e s => (s, assign x e) :: nil
  | Istore κ e a s => (s, store κ e a) :: nil
  | Iifthenelse e ifso ifno => (ifso, assume e) :: (ifno, assume (Eunop Onotbool e)) :: nil
  | ...
  end.
```

Figure 3.12: Abstract semantics of CFG

```
Theorem value_analysis_sound : ∀ (p: program) (stk: list stackframe) (f: function)
 (sp: val) (pc: node) (e: env) (m: Mem.mem),
   Reach p (State stk f sp pc e m) →
   ∃ range, vanalysis f pc = NotBot range ∧
   ∀ x, match e ! x with
      | Some (Vint i | Vptr _ i) => ints_in_range (range x) i
      | _ => True end.
Proof. (* proof omitted here *) Qed.
```

Figure 3.13: Soundness theorem of the analysis

point is some range function that conservatively approximates the (numerical) content of each local variable x (the property ints_in_range has been defined in Figure 3.9).

This theorem follows from the fact that the following property is a state invariant:

```
Definition gamma_state : ℘(state) :=
 λ s,
   match s with
   | State s f sp pc rs m => (rs, m) ∈ γ (solve_pfp f pc)
   ... (* omitted details *)
   end.
```

This invariant expresses that at program point pc, the contents of the registers (rs) and memory m are over-approximated by the analysis result at this program point. The omitted details deal with call stacks and enable the handling of (external) function calls.

The proof that this is an invariant follows from the soundness of the abstract memory transformers and the solve_pfp_postfixpoint lemma discussed before (§ 3.2).

3.4.2 Basic Implementation

The implementation of this interface leverages the (possibly relational) numerical domains previously introduced to compute over approximations of memory states. A concrete memory state is made of a local environment of type env that maps temporary variables (registers) to values and a global CompCert memory.

The simple implementation that we present here does not recover any information about the global memory. It only keeps track of the content of the registers, which are variables that do not have an address: there cannot be targeted by a pointer expression and are always referred to explicitly. Nonetheless, this implementation features the basic architecture that will be expanded further on (this is the topic of Chapter 5). It is parameterized by a numerical domain over some type N.

The concretization of this domain (of type $N \to \wp(var \to int)$) is extended to represent sets of concrete memory states (the γ in the definition refers to the concretization of the numerical domain N):

```
Instance gamma_mem : gamma_op N memory :=
 λ (ab: N) (em: env * mem),
   let (e,_) := em in
```

```
∃ ρ: var ⟶ int,
 ρ ∈ γ ab ∧
 (∀ x, match e ! x with Some (Vint i | Vptr _ i) => ρ x = i | _ => True end).
```

This means that the memory domain uses the numerical domain to compute, for each temporary variable that holds an integer or pointer value, an over-approximation of the integer component of this value (i.e., the values of integers and the offsets of pointers).

The core operation of this domain is the conversion from CFG expressions (type expr, defined in Figure 2.7) to numerical expressions (type nexpr, defined in Figure 3.7). This transformation is performed in two steps, to remove loads then pointers. The first step produces an intermediate expression of type pexpr, similar to a CFG expression, but without loads (the details of this expression type will be discussed in a following chapter, and in particular in Section 5.3.1). In this basic implementation that does not compute anything about the content of the memory, loads are trivially removed. Removing pointers mostly consists in translating pointer operations into the corresponding operations on their offsets. The only corner case is with comparisons and conditional expressions: to compare two pointers, it is not enough to compare their offsets; no pointer is equal to an integer; and all pointers are "true".

As an example, consider the expression x ==u y (unsigned equality test). If x and y both are integers, it is already a numerical expression. If both are pointers, the analyzer needs some information about the blocks these pointers point to (if the blocks are equal, the offsets are to be compared; if the blocks are different, the pointers are different). Finally, if one is a pointer whereas the other is an integer, then the result is always false. Therefore, to correctly translate comparisons and conditional expressions, the analyzer must be able to predict the types of the involved subexpressions. The numerical analysis is thus complemented with a (parallel, non-relational) type analysis that computes, for each temporary variable, an approximation of its contents: is it definitely an integer, definitely a (non-null) pointer, or any value. The type IOP.t is made of the two constants VI and VP to represent integers and pointers respectively; it is complemented with a generic All element to form the type IOP.t+⊤. This leads to the following type for the memory domain.

```
Definition t := (N * Map [ var, IOP.t ]).
```

Then, each transfer function applies the conversion to feed the numerical domain and the type analysis. For instance, the assign operator, applied to variable x and expression e, takes the following form. First loads are removed from the CFG expression e by the convert function. If this conversion fails (i.e., returns None), everything known about variable x is invalidated (forget). Otherwise, the resulting expression pe given on one hand to the type analysis that infers the type of the result and on the other hand to the numerical domain after a second step of conversion.

```
Definition assign (x: ident) (e: expr) (ab: t) : t+⊥ :=
 match convert e with
 | None => forget x ab
 | Some pe =>
  let (nm, tp) := ab in
  do_bot ty <- Eval.eval_expr IOP.adom tp pe;
  do_bot nm' <- assign x (nconvert tp pe) nm;
```

```
    NotBot (nm', TDom.set tp x ty)
  end.
```

In the previous sections, we have seen how our value analysis is built and proved correct. Next section discusses some unproved properties of this analysis such as its precision.

3.5 Experimental Evaluation

The soundness theorem of the analyzer states that if the analysis produces a result, then this result over-approximates all possible behaviors of the analyzed program. It does not guarantee that the analysis indeed terminates or produces any result; it does not say anything about the precision of the result. A sound analyzer may fail to terminate within a reasonable amount of time or may return imprecise results. Therefore, we have conducted some experiments to evaluate the precision and the efficiency of our analyzer. A more detailed description of this evaluation can be found in the dissertation of A. Maroneze [Mar14, § 6.3].

We compare our analyzer to two interval-based analyzers operating over C programs: a state-of-the-art industrial tool, Frama-C [Cuo+12], and an implementation of a value-range analysis [Nav+12] which will be referred to as *Wrapped*). Frama-C is an industrial-strength framework for static analysis, developed at CĒA. It integrates an abstract interpretation-based inter-procedural value analysis on interval domains with congruence, *k*-sets and memory analysis. It operates over C programs and has a very deep knowledge of its semantics, allowing it to slice out undefined behaviors for more precise results. It currently does not handle recursive functions. The Wrapped value-range analyzer relies on the LLVM compiler infrastructure and operates over its intermediate representation to perform an interval analysis in a signedness-agnostic manner, using so-called "wrapped" intervals (hence its name) to deal with machine-integer issues such as overflows while retaining precision. It is an intra-procedural tool, but can benefit from LLVM's inlining to perform inter-procedurally in the absence of recursion and function pointers.

The three tools have been compared on significant C programs from CompCert test suite. They range from a few dozen to a few thousand statements. To relate information from different analyses, we annotated the programs to capture information on integer variables at function entries and exits and at loops (for iteration variables). This amounts to 545 annotations in the 20 programs considered. For each program point, we counted the number of *bounded* variables. We consider as bounded any variable whose inferred interval has no more than 2^{31} elements, and hence rule out useless intervals like $x \in [-2^{31};\ 2^{31} - 2]$, inferred after a guard like $x < y$. Finally, to be able to compare the results of an inter-procedural analysis with those of two intra-procedural analyses with inlining, we considered for each annotation the union of the intervals of all call contexts. Less than 10% of intervals present a union of different intervals, and among those several preserve the boundedness for all contexts. Overall, its impact on the results is negligible.

The results are shown in Figure 3.14, which displays the number of bounded variables per program and per analyzer. In total, Frama-C bounded 398 variables, our analyzer got 355, and Wrapped ended up with 305. The main differences between our analyzer and Frama-C, especially on the larger benchmarks (lzw, arcode and lzss) result from global variable tracking and congruence information. Such reasoning is not handled by our

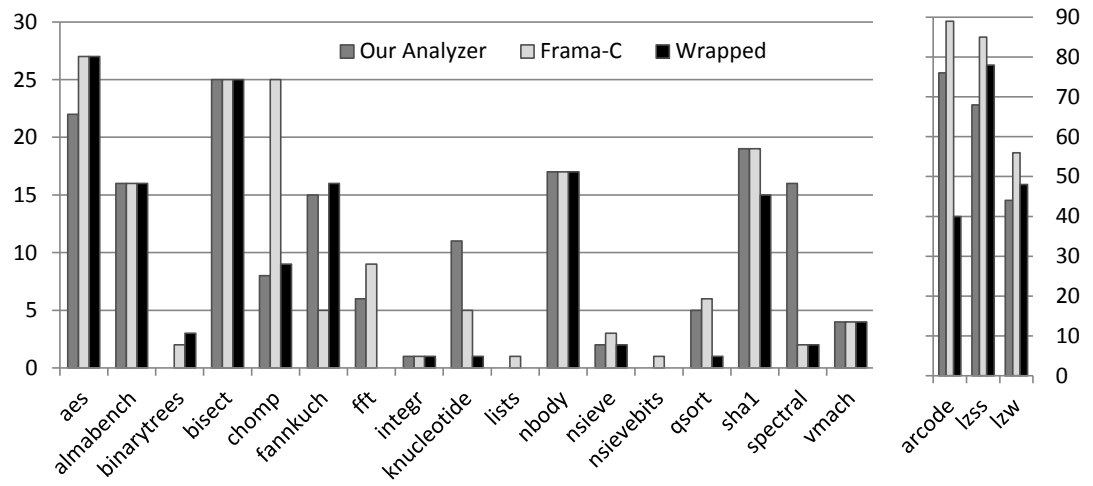


Figure 3.14: Number of bounded intervals (bounded per program and analyzer)

analyzer. On the other hand, the precision of our product of signed and unsigned domains allows us to bound more variables (e.g., on fannkuch), where Wrapped also obtains a good score, mainly due to variables bounded as $[0, 2^{31} - 1]$ and similar values. Some issues with the inlining used by Wrapped explain its worse results in fft, knucleotide and spectral.

We also compared the execution times of the analyses. Overall, our analysis runs faster than Frama-C because we track less information, such as pointers and global variables. For programs without these features, both analyses run in roughly the same time, from a few tenths of seconds for the smaller programs up to a few seconds for the larger ones. Wrapped's analysis is faster than the others. On a larger benchmark (over 3,000 lines of C code and about 10,000 CFG instructions after inlining) our analysis took 34 seconds to perform.

It is hard to draw final conclusions about the precision of our tool from these experiments. Frama-C, for instance, is likely to perform better on specific industrial critical software for which it has been specially tuned. Nevertheless we give evidence that our analyzer performs non-trivial reasoning over the C semantics, close to that of state-of-the-art (non-verified) tools.

3.6 Related Work and Conclusion

So as to be confident in the results of value analyzers, special care has to be given to the verification of such tools. In this chapter, we have described the design, implementation, and soundness proof in the Coq proof assistant of a value analyzer for a realistic language close to C: the CFG intermediate representation of the CompCert compiler.

The current formalization of this analyzer is directly inspired by the design choices of the Astrēe static analyzer [Cou+05], trying to capture some of its key interfaces. Our current abstract memory model is aligned with the model developed by Minē [Min06] because we connect a C abstract semantics with a generic notion of numerical abstract domain. Still our treatment of memory is simplified since we only track values of local variables in the current implementation of our analyzer. This analyzer is also inspired by the static analyses devoted to a precise handling of signed and unsigned integers [SK07;

Nav+12].

The precision of the analysis has been experimentally evaluated and compared on several benchmarks. It performs comparably to existing off-the-shelf (unverified) tools, Frama-C [Cuo+12] and Wrapped [Nav+12].

Our work constitutes the first machine-checked proof of a nontrivial value analysis based on abstract interpretation and a reference implementation of a tool. Previous work on mechanized verification of static analyses has been mostly based on classic data flow frameworks: Klein and Nipkow instantiate this framework for inference of Java bytecode types [KN06]; Coupet-Grimal and Delobel [CD04] and Bertot *et al.* [BGL06] for compiler optimizations, and Cachera *et al.* [Cac+05] for control flow analysis. Vafeiadis *et al.* [VN11] rely on a simple data flow analysis to verify a fence elimination optimization for concurrent C programs. Compared to these prior works, our value analysis relies on fixpoint iterations that are accelerating with widening operators. Cachera and Pichardie [CP10] and Nipkow [Nip12] describe a verified static analysis based on widenings but their technique is restricted to structured programs and targets languages without machine arithmetic nor pointers. Robert and Leroy [RL12] have developed a points-to analysis in the CompCert framework. This static analysis technique is quite orthogonal to what we formalize here. Hofmann *et al.* [HKS10] provide a machine-checked correctness proof in Coq for a generic post-fixpoint solver named RLD. That solver takes as input a set of constraints and finds a suitable solution by implementing an optimized dynamic iteration strategy. The formalized algorithm is not fully executable and cannot be extracted to OCaml code.

The proof effort required to establish the soundness of our analyzer has been mitigated by several design choices. We have adopted a lightweight abstract interpretation framework that requires few properties on the abstract domains. For instance, we do not prove that abstract domains enjoy a lattice structure, nor that the analysis always terminates. Following lessons learned from the CompCert development, we rely on *a posteriori* validation of complex algorithms: the Bourdoncle algorithm that computes the iteration strategy and the actual computation of the analysis result are seen as untrusted external solvers whose results are checked as part of the analysis. Finally, properties that are not critical (but still important, like the precision of the analysis result), are assessed experimentally rather than formally specified and proved.

This analyzer has some weaknesses that we will address in the next chapters. In particular we will implement all the dotted boxes mentioned on Figure 3.1. First, we want to replace the current memory abstraction with a domain similar to Minē's memory model [Min06]; this is the topic of chapter 5. We also intend to connect more numerical domains to the interface for numerical environments, and in particular relational abstract domains. Several such improvements will be discussed in chapter 6.

Meanwhile, we explore in chapter 4 methods for verified static analysis of lower-level programming languages, in which the control-flow graph of programs is unknown to the analyzer and discovered as part of the analysis.

Chapter 4

Verified value analysis & self-modifying programs

The previous chapter has introduced general methodology and techniques to build verified static analyzers. These techniques assumed that the programs to analyze are given as a control-flow graph. However, static analysis can be applied to various kinds of programs: it is generally applied to source code or some intermediate representation as CFG, but it is sometimes more suitable to apply it on lower-level representations as binary code. Indeed, no source code may be available or the compiler might not be trusted. Indeed, even correct compiler may not preserve the property of interest (e.g., if this property is not functional); this is sometimes referred to as the WYSINWYX phenomenon [BR10].

Static analysis techniques should be adapted when they are applied to binary programs, i.e., when the program to be analyzed comes in the form of a sequence of bits and must first be disassembled. Disassembling is the process of translating a program from a machine friendly binary format to a textual representation of its instructions. It requires to *decode* the instructions (i.e., understand which instruction is represented by each particular bit pattern) but also to precisely locate the instructions in memory. Indeed instructions may be interleaved with data or arbitrary padding. Moreover once encoded, instructions may have various byte sizes and may not be well aligned in memory, so that a single byte may belong to several instructions.

The two main disassembling techniques are known as *linear sweep* and *recursive traversal* [LD03]. The first decodes each code segment from its beginning and assumes that each further instruction starts where the previous one ends; as it ignores the control-flow of the program, it may unexpectedly decode data or padding or fail to identify the beginning of instructions. Recursive traversal tries to address these weaknesses by following the control-flow rather than blindly following the linear ordering of the bytes in the binary: after decoding one instruction, disassembling continues for all possible successors of this instruction. This in turn suffers from the limitation that computed jumps have unknown targets. So this technique may fail to disassemble some parts of the code.

To succeed in disassembling all code without performing a linear sweep from each byte, a static analysis must predict a sound but hopefully precise over-approximation of the targets of every reachable computed jump.

In addition, instructions may be produced at run-time, as a result of the very execution of the program. A simple example is the modification of some operands (e.g., registers) of existing instructions. Another example is the creation of new sequences of instructions in an existing code. Such programs are called *self-modifying* programs; they are commonly used in security as an obfuscation technique (e.g., to protect the intellectual property of the program authors, to increase the stealth of a malware) [Szo05], as well as in just-in-time

compilation and in operating systems (mainly for improving performances).

Because the instructions of a self-modifying program are not the instructions that will be executed, most of standard reverse engineering tools (e.g., the IDA Pro disassembler and debugger, a *de facto* standard for the analysis of binary code) cannot disassemble and analyze self-modifying programs. In order to disassemble and analyze such programs, one must very precisely understand which instructions are written and where. And for all programs, one must check every single memory write to decide whether it modifies the program code.

As the real code of a self-modifying program is hidden and varies over time, self-modifying programs are also beyond the scope of the vast majority of formal semantics of programming languages. Indeed a prerequisite in such semantics is the isolation and the non-modification of code in memory. To address the challenge of formal reasoning on self-modifying code, Shao *et al.* [CSV07] propose a Hoare-logic-like framework, called GCAP, and illustrate its use and effectiveness on a dozen of programs that manipulate their code at run-time. They can establish safety properties of all these programs.

In this chapter, we formalize, with the Coq proof assistant, key static analysis techniques to predict the possible targets of the computed jumps and make precise which instructions alter the code and how, while ensuring that the other instructions do not modify the program. This allows automatic reasoning about self-modifying programs and to automatically prove their safety.

Our formalization effort is divided in three parts. Firstly, we formalize a small binary language (called Goto*) in which code is handled as regular mutable data. This language has little constructions so as to limit the cost of its formal definition. Nonetheless, it features properties that are inspired from real languages like x86 assembly: computed memory accesses and jumps, variable-length encoding of instructions, and conditional branches based on flags. We formally define an executable concrete semantics for this language and a non-trivial class of "safe" programs. Secondly, we formalize and prove correct an abstract interpreter that takes as input an initial memory state, computes an over-approximation of the reachable states that may be generated during the program execution, and then checks that all reachable states maintain memory safety. Good precision of the analyzer is achieved through specific techniques including a simple form of trace partitioning. Finally, we extract from our formalization an executable OCaml tool that we run on several self-modifying challenges [CSV07].

A short version of this chapter has been published at the international conference on Interactive Theorem Proving (ITP) [BLP14], and submitted for publication at the Special issue for ITP 2014 of the Journal of Automated Reasoning (JAR).

4.1 Disassembling by Abstract Interpretation

We now present the main principles of our analysis on the program shown in Figure 4.1. It is printed as a sequence of bytes (on the extreme left) as well as under a disassembled form (on the extreme right) for readability purposes. This program, as we will see, is self-modifying, so these bytes correspond to the initial content of the memory from addresses 0 to 11. The remainder of the memory (addresses in $[-2^{31}; -1] \cup [12; 2^{31} - 1]$), as well as the content of the registers, is unknown and can be regarded as the program input.

All our example programs target a machine operating over a low-level memory made

Initial program	Possible final program	Initial assembly listing
07000607	07000607	0: cmp R6, R7
03000000	03000000	1: gotoLE **5**
00000005	**00000004**	2:
00000000	00000000	3: halt R0
00000100	00000100	4: halt R1
09000000	09000000	5: cst 4 → R0
00000004	00000004	6:
09000002	09000002	7: cst 2 → R2
00000002	00000002	8:
05000002	05000002	9: store R0 → *R2
04000000	04000000	10: goto 1
00000001	00000001	11:

Figure 4.1: A self-modifying program: as a byte sequence (left); after some execution steps (middle); assembly source (right).

of 2^{32} cells, eight registers, and flags — boolean registers that are set by comparison instructions. Each memory cell or register stores a 32 bits integer value[1], that may be used as an address in the memory. Programs are stored as regular data in the memory; their execution starts from address zero.

Nevertheless, throughout this chapter we write the programs using the following custom syntax. The instruction cst v → r loads register r with the given value v. The instruction cmp r, r' denotes the comparison of the contents of registers r and r'. The instruction gotoLE d is a conditional jump to d, that is taken if in the previous comparison (cmp r, r') the content of r' was less than or equal to the one of r; goto d is an unconditional jump to d. The instructions load *r → r' and store r' → *r denote accesses to memory at the address given in register r; and halt r halts the machine with as final value the content of register r.

The programming language we consider is inspired from x86 assembly; notably instructions have variable size (one or two bytes, e.g., the length of the instruction gotoLE 5 stored at line 1 is two bytes, the byte 03000000 for goto and one byte for 5) and conditional jumps rely on flags. In this setting, a program is no more than an initial memory state, and a program point is simply the address of the next instruction to execute.

In order to understand the behavior of this program, one can follow its code as it is executed starting from the entry point (byte 0). The first instruction cmp R6, R7 compares the (statically unknown) content of two registers. This comparison modifies only the states of the flags. Then, the gotoLE 5 instruction is executed and, depending on the outcome of this comparison, the execution proceeds either on the following instruction (stored at byte 3), or from byte 5. Since the analysis cannot predict which branch will be taken, both branches must be analyzed.

Executing the block from byte 5 will modify the byte 2 belonging to the gotoLE instruction (highlighted in Figure 4.1); more precisely it will change the jump destination from 5 to 4: the store R0 → *R2 instruction writes the content of register R0 (namely 4) in memory

[1] Each byte — addressable unit of storage data — is therefore four octets long.

at the address given in register R2 (namely 2). Notice that a program may directly read from or write to any memory cell: we assume that there is no protection mechanism as provided by usual operating systems. After the modification is performed, the execution jumps back to the modified instruction, jumps to byte 4 then halts, with final value the content of register R1.

This example highlights that the code of a program (or its control-flow graph) is not necessarily a static property of this program: it may vary as the program runs. To correctly analyze such a program, one must discover, during the fixpoint iteration, the two possible states of the goto instruction at program points 1 and 2 and its two possible targets (i.e., 4 and 5). More generally, we need at least to know, for each program point (i.e., memory location), which instructions may be decoded from there when the execution reaches this point. This in turn requires to know what are the values that the program operates on. We therefore devise a value analysis that computes, for each reachable program point (i.e., in a *flow sensitive* way) an over-approximation of the content of the memory and the registers, and the state of the flags, when the execution reaches that point.

The analysis relies on a numeric abstract domain $\mathsf{N}^\sharp$ that provides a representation for sets of machine integers and abstract arithmetic operations. $\gamma_\mathsf{N} : \mathsf{N}^\sharp \to \wp(\mathsf{int})$ denotes the associated concretization function. Relying on such a numeric domain, one can build abstract transformers. They model the execution of each instruction over an abstract memory that maps locations (i.e., memory addresses[2] and registers) to abstract numeric values. An abstract state is then a mapping that attaches such an abstract memory to each program point of the program, and thus belongs to $\mathsf{addr} \to \big((\mathsf{addr} + \mathsf{reg}) \to \mathsf{N}^\sharp\big)$.

To perform one abstract execution step, from a program point pp and an abstract memory state $\mathsf{m}^\sharp$ that is attached to pp, we first enumerate all instructions that may be decoded from the set $\gamma_\mathsf{N}(\mathsf{m}^\sharp(\mathsf{pp}))$. Then for each of such instructions, we apply the matching abstract transformer. This yields a new set of successor states whose program points are dynamically discovered during the fixpoint iteration.

The abstract interpretation of a whole program iteratively builds an approximation executing all reachable instructions until nothing new is learned. This iterative process may not terminate, since there might be infinite increasing chains in the abstract search space. As usual in abstract interpretation, we accelerate the iteration using widening operations [CC77]. Once a stable approximation is finally reached, an approximation of the program listing or control-flow graph can be produced.

To illustrate this process, Figure 4.2 shows how the analysis of the program from Figure 4.1 proceeds. We do not expose a whole abstract memory but only the underlying control-flow graph it represents. On this specific example, three different graphs are encountered during the analysis. For each program point pp, we represent a node with same name and link it with all the possible successor nodes according to the decoding of the set $\gamma_\mathsf{N}(\mathsf{m}^\sharp(\mathsf{pp}))$. The array shows the construction of the fixpoint: each line represents a program point and the columns represent the iterations of the analysis. In each array cell lies the name of the control-flow graph representing the abstract memory for the given program point during the given iteration; the symbol $\bot$ stands for an unreachable program point. The shaded array cells highlight the program points that need to be analyzed: they are the worklist.

Initially, at iteration 0, only program point 0 is known to be reachable and the memory

[2] Type addr is a synonym of int, the type of machine integers.

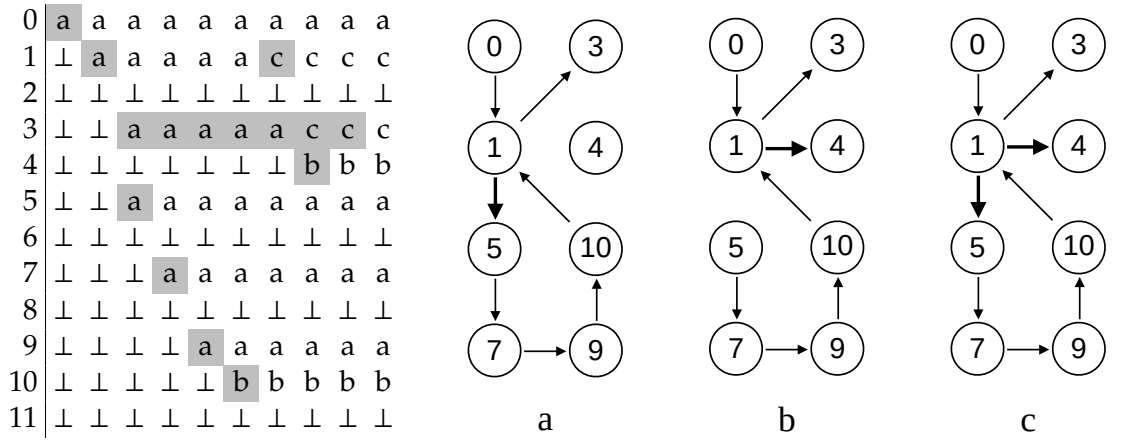


Figure 4.2: Iterative fixpoint computation

is known to exactly contain the program, denoted by the first control-flow graph (called *a* in Figure 4.2 and corresponding to the initial program of Figure 4.1). The only successor of point 0 is point 1 and it is updated at the next iteration. After a few iterations, point 9 is reached and the abstract control-flow graph *a* is updated into the control-flow graph *b* that is propagated to point 10. This control-flow graph corresponds to the possible final program of Figure 4.1, where program-point 5 became unreachable. At the next iteration, program point 1 (i.e., the loop condition) is reached again and the control-flow graph *b* is updated into the control-flow graph *c* that corresponds to the union of the two previous control-flow graphs. After a few more iterations, the process converges.

In addition to a control-flow graph or an assembly listing, more properties can be deduced from the analysis result. We can prove safety properties about the analyzed program, like the fact that its execution is never stuck. Since the semantics only defines the good behaviors of programs, unsafe programs reach states that are not final and from which no further execution step is possible (e.g., the byte sequence at current program point is not the valid encoding of an instruction).

The analysis produces an over-approximation of the set of reachable states. In particular, a superset of the reachable program points is computed, and for each of these program points, an over-approximation of the memory state when the execution reaches this program point is available. Thus we can check that for every program point that may be reached, the next execution step from this point cannot be stuck. This verification procedure is formally verified, as described in the following section.

4.2 Semantics of Goto⋆

This section defines the abstract syntax and semantics of the low-level language our static analyzer operates over. The semantics uses a decoding function from binary code to our low-level language. The semantics is presented as a small-step operational semantics that can observe self-modifying programs.

4.2.1 Abstract Syntax

```
Definition addr := int.
Inductive reg  := R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7.
Inductive flag := FLE | FLT | FEQ.

Inductive comparison  :=  Ceq | Cne | Clt | Cle | Cgt | Cge.
Inductive binop := OpAdd | OpSub | OpMul | OpDivs | OpShl | OpShr | OpShru
| OpAnd | OpOr  | OpXor | OpCmp (c: comparison) | OpCmpu (c: comparison).

Inductive instruction :=
(* arithmetic *)
| ICst (v:int) (dst:reg) | ICmp (src dst: reg) | IBinop (op: binop) (src dst: reg)
(* memory *)
| ILoad  (src dst: reg) | IStore (src dst: reg)
(* control *)
| IGoto (tgt: addr) | IGotoInd (r: reg) | IGotoCond (f: flag) (tgt: addr) | ISkip | IHalt (r: reg).
```

Figure 4.3: Abstract syntax of Goto⋆

The programming language in which are written the programs to analyze is formalized using the abstract syntax shown on Figure 4.3. In the Coq formalization, the abstract syntax is presented as inductive data types. Machine integers (type int) are those of the CompCert library Int of 32 bits machine integers [BL09]. The eight registers of our language are called R0, … R7 and there are three register flags called FLE (for "less or equal" comparisons), FLT and FEQ.

Instructions are either arithmetic expressions, or memory accesses or control-flow instructions. Instructions for accessing memory are ILoad and IStore; their operands are registers. So as to keep the language simple, memory accesses are limited to these two instructions: the other instructions, which are described next, only operate on registers. Arithmetic expressions consist of integer constants, signed comparisons and binary operations. Control-flow instructions consist of unconditional and conditional jump instructions, the empty instruction ISkip and the IHalt instruction which halts the program execution. For unconditional jumps, we distinguish register-indirect jumps (IGotoInd r instructions, where r is a register) from other jumps (i.e., absolute jumps, written as IGoto v, where v is a literal constant address).

In a binary language, there is no distinction between code and data: a value stored in memory can be interpreted either as representing data or as encoding instructions. So as to model a binary language, we first introduce a decoding function called decode_from. Its type is (addr→int) → pp → option(instruction×nat). Given a memory mem of type addr → int (i.e., a function from addresses to values) and an address pp of type addr, this function yields the instruction stored at this address along with its byte size (so as to know where the next instruction begins). This size is of type nat, the Coq type for natural numbers. Since not all integer sequences are valid encodings, this decoding may fail (hence the option type). In order to be able to conveniently write programs, there is also a matching encoding function called enc, whose type is instruction → list int. However the development does not depend on it at all: properties are stated in terms of already

```
Definition decode_register (v: Z) : option register :=
 match v with
 | 0 => ⌊R0⌋ | 1 => ⌊R1⌋ | ... | 7 => ⌊R7⌋
 | _ => None end.

Definition decode_flag (v: Z) : option flag :=
 match v with
 | 0 => ⌊FLE⌋ | 1 => ⌊FLT⌋ | 2 => ⌊FEQ⌋
 | _ => None end.

Definition decode_binop (v: Z) : option binop :=
 match v with
 | 0 => ⌊OpAdd⌋ | 1 => ⌊OpSub⌋ | ... | 9 => ⌊OpXor⌋
 | _ => None end.

Definition split_instruction (v: int) : Z × Z × Z × Z :=
 let v := Int.unsigned v in
 let (v, dst) := Z.div_eucl v 256 in
 let (v, src) := Z.div_eucl v 256 in
 let (v, arg) := Z.div_eucl v 256 in
 let (v, typ) := Z.div_eucl v 256 in
 (typ, arg, src, dst).

Definition decode_from (m: addr ⟶ int) (base: addr) : option (instruction × nat) :=
 match split_instruction (m base) with
 | (0, 0, src, 0) => do rs <- decode_register src; ⌊(IHalt rs, 1)⌋
 | (1, 0, 0, 0) => ⌊(ISkip, 1)⌋
 | (2, 0, src, 0) => do rs <- decode_register src; ⌊(IGotoInd rs, 1)⌋
 | (3, flg, 0, 0) => do f <- decode_flag flg; ⌊(IGotoCond f (m (base+1)), 2)⌋
 | (4, 0, 0, 0) => ⌊(IGoto (m (base+1)), 2)⌋
 | (5, 0, src, dst) => do rs <- decode_register src;
                         do rd <- decode_register dst; ⌊(IStore rs rd, 1)⌋
 | (6, 0, src, dst) => do rs <- decode_register src;
                         do rd <- decode_register dst; ⌊(ILoad rs rd, 1)⌋
 | (7, 0, src, dst) => do rs <- decode_register src;
                         do rd <- decode_register dst; ⌊(ICmp rs rd, 1)⌋
 | (8, o, src, dst) => do op <- decode_binop o;
                         do rs <- decode_register src;
                         do rd <- decode_register dst; ⌊(IBinop op rs rd, 1)⌋
 | (9, 0, 0, dst) => do rd <- decode_register dst; ⌊(ICst (m (base+1)) rd, 2)⌋
 | _ => None end.
```

Figure 4.4: Decoding binary code

```
Definition flag_state : Type := flag → bool.
Definition register_state : Type := register → int.
Definition memory : Type := addr → int.

Record machine_config : Type := {
  pc : int; mc_flg : flag_state; mc_reg : register_state; mc_mem : memory
}.
Inductive machine_state := Halted (v: int) | Running (c: machine_config).
Notation "⌈ v ⌉" := (Halted v).
Notation "〈 pp , f , r , m 〉" := (Running {| pc := pp; mc_flg:= f; mc_reg:= r; mc_mem := m |}).
```

Figure 4.5: Concrete execution states of Goto⋆

encoded programs.

The decoding function is defined in Figure 4.4. The binary decoding of a sequence of bytes stored in memory m at program point pp is written (decode_from m pp). A successful decoding yields a pair ⌊(i,sz)⌋, where sz is the size of i, the instruction stored at address pp in m[3]. The notation ⌊·⌋ denotes a successful result as opposed to a failure represented by None.

The binary format of instructions is arbitrary and has little impact on the design of the analyzer. We rely on the fact that the encoding length can be inferred from the first byte of any encoded instruction[4]. The self-modifying programs that we consider rely on the particular encoding that we chose. This encoding works as follows. Instructions that hold a value (e.g., IGoto 5) require two bytes: the value occupies the second byte; other instructions require one byte. The first byte is made of four fields of one octet each: the decoding of this first byte first extracts the content of each field using Euclidean divisions (performed by the split_instruction function). The first field (typ) corresponds to the constructor of the instruction data type. From its value, one can deduce the size of the instruction and how to interpret the next fields. The second field (flg) holds a flag (only used in the IGotoCond instruction). The third and fourth fields hold respectively the source and destination registers. Depending on the instruction, none, both or only one of them may be relevant. Unused fields always have the value zero. This encoding is very sparse: many byte sequences do not represent any valid instruction. Moreover, in the decoding function, errors are propagated by the bind operator of the error monad, written do a <- m; b.

4.2.2 Semantics

The language semantics is given as a small-step transition relation between machine states (see Figure 4.5). A machine state may be ⟨pp, f, r, m⟩ where pp is the current program point (address of the next instruction to be executed), f is the current flag state, r is the current register state, and m is the current memory. Such a tuple is called a machine

[3]The size cannot be deduced from the instruction as: 1. the encoding function is not known; and 2. they may be several encodings, of various sizes, for a single instruction.

[4]This is not the case, for instance, of the encoding of x86 instructions, that may begin with an arbitrary number of one-byte prefixes

```
Definition compare (i j: int) (f: flag) : bool :=
 match f with
 | FLE => negb (Int.lt j i) | FLT => Int.lt i j | FEQ => Int.eq i j end.

Definition step (ms: machine_state) : option machine_state :=
 match ms with
 | ⟨ pp, f, r, m ⟩ =>
   do i_sz  <- decode_from m pp;
   let (i, sz) := i_sz in
   ⌊match i with
    | ICst v rd => ⟨pp+sz, f, r#rd ⟵ v, m⟩
    | ICmp rs rd => ⟨pp+sz, compare (r rd) (r rs), r, m⟩
    | IBinop op rs rd => ⟨pp+sz, f, r#rd ⟵ (r rs [op] r rd), m⟩
    | ILoad rs rd => ⟨pp+sz, f, r#rd ⟵ m (r rs), m⟩
    | IStore rs rd => ⟨pp+sz, f, r, m # r rd ⟵ r rs⟩
    | IGoto v => ⟨v, f, r, m⟩
    | IGotoCond c v => ⟨if f(c) then v else pp+sz, f, r, m⟩
    | IGotoInd rd => ⟨r(rd), f, r, m⟩
    | ISkip => ⟨pp+sz, f, r, m⟩
    | IHalt rd => ⌈r rd⌉
   end⌋
 | ⌈ _ ⌉ => None end.

Definition small_step : relation machine_state := λ a b, step a = Some b.
Infix "⟹" := small_step.
```

Figure 4.6: Concrete semantics

configuration (type machine_config). Otherwise, a machine state is ⌈v⌉, meaning that the program stopped returning the value v. Values are machine integers (type int).

The semantics is defined in Figure 4.6 by a partial function code from machine states to machine states. This function is then lifted to the small_step relation. A step can only be performed from a running configuration ⟨pp, f, r, m⟩. From such a configuration, the contents of the memory m from address pp is first decoded. If this decoding fails, the execution is stuck. Otherwise, the decoding yields a pair made of an instruction i and its size sz. Then, depending on the actual instruction, the step function builds the state resulting from its execution. In each case, most of the state is kept unchanged. Instructions that are not branching proceed their execution at program point pp+sz (since sz is the size of the instruction that has been decoded). In the code, the notation s # id ⟵ v stands for the update of s with a new value v for register or memory cell id; and the notation [op] stands for the denotation of the binary operator op on machine integers.

Instruction ICst v rd updates destination register rd with value v. Instruction ICmp rs rd updates the flag state according to the comparison (compare) of the values held by the two involved registers. Instruction IBinop op rs rd applies the denotation [op] of the given binary operator op to the contents r(rs) and r(rd) of registers rs and rd. Then, it updates the

state of register rd: in r, the new value of rd thus becomes (r rs) [op] (r rd). Instruction ILoad rs rd updates register rd with the value m(r(rs)) found in memory at the address given in register rs. Instruction IStore rs rd updates the memory at the address given in register rd with the value given in register rs.

Instruction IGoto v sets the program point to v. Instruction ISkip does nothing: execution proceeds at next program point. Conditional jump instruction IGotoCond c v jumps to address v or falls through to pp+sz depending on the current state of flag c. Indirect jump instruction IGotoInd rd proceeds at the program point found in register rd. Instruction IHalt rs terminates the execution, returning the content of register rs.

Finally, we define the semantics $[\![P]\!]$ of a program P as the set of states s that are reachable from an initial state $\langle 0, f, r, P\rangle$, with current program point zero and memory P (where $\Rightarrow^\star$ denotes the reflexive-transitive closure of the small-step relation):

$$[\![P]\!] = \{\, s \mid \exists f\ r,\ \langle 0, f, r, P\rangle \Rightarrow^\star s \,\}\,.$$

Notice that the program P belongs to the state: it is initially known, but can be modified as the execution goes on.

4.3 Abstract Interpreter

The static analysis that we devise in this chapter is parameterized by an abstract memory domain (of type ab_mc) and an abstract numerical domain (of type $\mathsf{int}^\sharp$). The interface of the memory abstract domain is presented in Section 4.3.1 as well as its implementation. The numerical domain specification is the same as the (non-relational) one from the previous chapter (§ 3.3.2). This memory domain interface enables the definition of an abstract semantics that is discussed in Section 4.3.2. The analysis is flow-sensitive: it computes an abstract state for each (reachable) program point. It also computes a numerical approximation of the final value (produced by instruction IHalt). Thus, the result of the analysis has the following type.

```
Definition AbEnv : Type := (Map [ addr, ab_mc ] * int♯+⊥).
```

The iterative computation of the analysis result, that will be described in Section 4.3.3, follows a different algorithm than in the previous chapter. Indeed, the control-flow graph of the program is not known until the analysis completes; the dependencies between the abstract states at different program points are discovered during the analysis. A pre-computation of an iteration strategy is therefore not possible.

4.3.1 Memory Abstract Domain

An abstract memory domain is a carrier type along with some primitive operators whose signatures are given in Figure 4.7. The carrier type ab_mc is equipped with a lattice structure (defined in § 3.1.1). An object of this type represents a set of triples flag-state × register-state × memory, as described by the primitive as_gamma. Such a triple ultimately represents any machine configuration with matching components at any program point (see gamma_to_mc).

A memory domain can be queried for the values stored in some register (var) or at some known memory address (load_single); these operators return an abstract numeric value.

```
Definition pre_machine_config: Type := flag_state × register_state × memory.
Instance gamma_to_mc A (G:gamma_op A pre_machine_config): gamma_op A machine_config :=
 λ a mc, (mc_flg mc, mc_reg mc, mc_mem mc) ∈ γ(a).

(* [mem_dom] is parameterized by a numerical domain. *)
Context (int♯: Type) (ab_num: num_dom int♯).

Record mem_dom (ab_mc: Type) : Type :=
 (* abstract domain with concretization to machine configurations *)
 { as_wl: weak_lattice ab_mc
 ; as_gamma : gamma_op ab_mc pre_machine_config
 ; as_adom : adom ab_mc machine_config as_wl as_gamma
 (* consult the contents of a register *)
 ; var: ab_mc → reg → int♯
 ; var_sound: ∀ ab:ab_mc, ∀ m: machine_config,
    m ∈ γ(ab) → ∀ r, mc_reg m r ∈ γ(var ab r)
 (* consult the contents of the memory at a particular address *)
 ; load_single: ab_mc → addr → int♯
 ; load_sound: ∀ ab:ab_mc, ∀ m: machine_config,
    m ∈ γ(ab) → ∀ a:addr, mc_mem m a ∈ γ(load_single ab a)
 (* update the contents of the memory at a particular address *)
 ; store_single: ab_mc → addr → int♯ → ab_mc
 ; store_sound: ∀ ab:ab_mc, ∀ dst v,
    Store (γ ab) dst v ⊆ γ (store_single ab dst v)
 (* update the flag state *)
 ; compare: ab_mc → register → register → ab_mc
 ; compare_sound: ∀ ab:ab_mc, ∀ rs rd,
    Compare (γ ab) rs rd ⊆ γ(compare ab rs rd)
 (* update the contents of a register *)
 ; assign: ab_mc → reg → int♯ → ab_mc
 ; assign_sound: ∀ ab:ab_mc, ∀ rd v,
    Assign (γ ab) rd v ⊆ γ(assign ab rd v)
 (* assume a flag is in a known state *)
 ; assume: ab_mc → flag → bool → ab_mc+⊥
 ; assume_sound: ∀ ab:ab_mc, ∀ f b,
    Assume (γ ab) f b ⊆ γ(assume ab f b)
 (* abstraction of a fragment of a concrete memory *)
 ; init: memory → list addr → ab_mc
 ; init_sound: ∀ (m: memory) (dom: list addr) f r (m′: memory),
    (∀ a, List.In a dom → m a = m′ a) → (f, r, m′) ∈ γ(init m dom)
 }.
```

Figure 4.7: Signature of abstract memory domains for analysis of Goto*

Other operators enable us to alter an abstract state, like assign that sets the contents of a register to a given abstract numeric value, and store_single that similarly updates the memory at a given address.

The operator compare updates the abstract counterpart of the flag state when two given registers are compared. We can also use the operator assume when we know the boolean value of a flag. This operator is a reduction. It is always sound to return the same abstract state as the first argument, but a more precise information may allow to gain precious information when reaching a conditional branch. Indeed, our first implementation of this operator is the identity; we will then refine it in our first extension (§ 4.4.2). The operator init is used when initializing the abstract interpreter with an abstraction of the initial memory. Part of the initial memory is exactly known: the initial program text, static data and so on. Therefore the init operator gets a list dom of addresses and a function m that gives the values of the actual initial memory m′ at these addresses.

All these operators obey some specifications. As an example, the load_sound property states that given a concrete state m in the concretization of an abstract state ab, the concrete value stored at any address a in m is over-approximated by the abstract value returned by the matching abstract load. The γ symbol is overloaded through the use of type classes: its first occurrence refers to the concretization from the abstract memory domain (the as_gamma field of record mem_dom) and its second occurrence is the concretization from the numeric domain ab_num.

This signature is similar to the one of the previous chapter (Figure 3.11 on page 41) as it features a lattice structure and basic operators to model the programming language instructions. The main difference is that operators do not take expressions as arguments but directly values. Consider for instance the load_single operator; it takes as argument a concrete address whereas the assign operator of Figure 3.11 receives an expression (and that expression may contain loads). This prevents the implementation to reason symbolically about addresses: this would be mandatory to deal with dynamically allocated memory (either with a system call like mmap or VirtualAlloc, or from a memory manager that is part of the program) but is beyond the scope of this work.

Such an abstract memory domain is implemented using two maps: from registers to abstract numeric values to represent the register state and from values to abstract numeric values to represent the memory.

```
Record ab_machine_config :=
  { ab_reg: Map [ reg, int# ] ; ab_mem: Map [ addr, int# ] }.
```

To prevent the domain of the ab_mem map from infinitely growing, we bound it by a finite set computed before the analysis: the analysis will try to compute some information only for the memory addresses found in this set. The content of this set does not alter its soundness: the values stored at addresses not in it are unknown and the analyzer makes no assumptions about them. On the other hand, the success of the analysis and its precision depend on it. In particular, the analyzed set must cover the whole code segment. To compute this set, one possible method [BR10] is to start from an initial guess and, every time the analysis discovers that the set is too small (when it infers that control may reach a point that is not is the set), the analysis is restarted using a larger set. In practice, for all our examples, running the analysis once was enough, taking as initial guess the addresses of the instructions of the initial program.

```
Definition load_many (m: ab_mc) (a: int♯) : int♯+⊥ :=
 match concretize a with
 | Just addr_set => IntSet.fold
    (λ acc addr, acc ⊔ NotBot (T.(load_single) m addr)) addr_set Bot
 | All => NotBot top
 end.
```

Figure 4.8: Example of abstract transformer

4.3.2 Abstract Semantics

As a second layer, we build abstract transformers over any such abstract domain. Consider for instance the abstract load called load_many and presented in Figure 4.8; it is used to analyze any ILoad instruction (T denotes a record of type mem_dom int♯ ab_mc). The source address may not be exactly known, but only represented by an abstract numeric value a. Since any address in $\gamma(a)$ may be read, we have to query all of them and take the least upper bound of all values that may be stored at any of these addresses: $\bigsqcup \{T.(load\_single)\ m\ x \mid x \in \gamma(a)\}$. However the set of concrete addresses may be huge and care must be taken: if the size of this set exceeds some threshold, the analysis gives up on this load and yields top, representing all possible values.

We build enough such abstract transformers to be able to analyze any instruction (function ab_post_single, shown in Figure 4.9). This function returns a list of possible next states, each of which being either Hlt v (the program halts returning a value approximated by v) or Run pp m (the execution proceeds at program point pp in a configuration approximated by m) or GiveUp (the analysis is too imprecise to compute anything meaningful).

The computed jump (IGotoInd) also has a dedicated abstract transformer (inlined in Figure 4.9): in order to know from where to continue the analysis, we have to enumerate all possible targets; the assign operation in each branch refines the knowledge about the value of the branching register rs). The abstract transformer for the conditional jump IGotoCond f v returns a two-element list. The first element means that the execution may proceed at pp + sz (i.e., falls through) in a state where the branching flag f is known to evaluate to false; the second element represents the case when the branch is taken: the flag is known to evaluate to true, and the next program point, v, is the one given in the instruction. Since each assume may return ⊥ meaning that the considered branch cannot be taken, we use the combinator bot_cons that propagates this information: the returned list does not contain the unreachable states.

Then, function ab_post_many performs one execution step in the abstract. To do so, we first need to identify what is the next instruction, i.e., to decode in the abstract memory from the current program point (function abstract_decode_at, not shown). This may require to enumerate all concrete values that may be stored at this address. Therefore this abstract decoding either returns a set of possible next instructions or gives up. In such a case, the whole analysis will abort since the analyzed program is unknown.

```
Inductive ab_post_res := Hlt (v: int♯) | Run (pp: addr) (m: ab_mc) | GiveUp.

Definition bot_cons A B (f: A ⟶ B) (a: A+⊥) (b: list B) : list B :=
 match a with NotBot a′ => f a′ :: b | Bot => b end.

Definition ab_post_single (m: ab_mc) (pp: addr) (instr: instruction × nat)
 : list ab_post_res :=
 match instr with
 | (IHalt rs, sz) => Hlt (T.(var) m rs) :: nil
 | (ISkip, sz) => Run (pp + sz) m :: nil
 | (IGoto v, sz) => Run v m :: nil
 | (IGotoInd rs, sz) =>
    match concretize (T.(var) m rs) with
    | Just tgt => IntSet.fold (λ acc addr,
        Run addr (assign m rs (const_int addr)) :: acc) tgt nil
    | All => GiveUp :: nil
    end
 | (IGotoCond f v, sz) =>
    bot_cons (Run (pp + sz)) (T.(assume) m f false)
     (bot_cons (Run v) (T.(assume) m f true) nil)
 | (IStore rs rd, sz) =>
    Run (pp + sz) (store_many m (T.(var) m rd) (T.(var) m rs)) :: nil
 | (ILoad rs rd, sz) =>
    match load_many m (T.(var) m rs) with
    | NotBot v => Run (pp + sz) (T.(assign) m rd v) :: nil
    | Bot => nil
    end
 | (ICmp rs rd, sz) => Run (pp + sz) (T.(compare) m rs rd ) :: nil
 | (ICst v rd, sz) => Run (pp + sz) (T.(assign) m rd v) :: nil
 | (IBinop op rs rd, sz) =>
    match T.(forward_int_binop) op (T.(var) m rs) (T.(var) m rd) with
    | NotBot v => Run (pp + sz) (T.(assign) m rd v) :: nil
    | Bot => nil
    end
 end.

Definition ab_post_many (pp: addr) (m:ab_mc) : list ab_post_res :=
 match abstract_decode_at pp m with
 | Just instr => flat_map (ab_post_single m pp) instr
 | All => GiveUp :: nil
 end.
```

Figure 4.9: Abstract small-step semantics of Goto⋆

```
Record analysis_state := {
  worklist : list int (* list of program points remaining to visit *);
  result_fs : Map [ int, ab_mc ] (* one value per program point; unbound values are ⊥ *);
  result_hlt: d+⊥ (* final value *) }.

Definition analysis_init P : analysis_state :=
  {| worklist := Int.zero :: nil ; result_fs := ([])[ Int.zero <- P ] ; result_hlt := Bot |}.
```

Figure 4.10: Internal state of the Goto⋆ analyzer

4.3.3 Fixpoint Computation

Finally, a full program analysis is performed applying this abstract semantics iteratively. The analysis follows a worklist algorithm as the one found in [BR10, § 3.4]. It maintains a state holding three pieces of data (see Figure 4.10):

1. the worklist, a list of program points left to explore; initially a singleton;
2. the current solution, mapping to each program point an abstract machine configuration; initially empty, but at program point zero, where it holds an abstraction of the program;
3. an abstraction of the final value, initially ⊥.

A single step of analysis is performed by the function analysis_step shown in Figure 4.11. It picks a node n in the worklist — unless it is empty, meaning that the analysis is over — and retrieves the abstract configuration ab_mc associated with this program point in the current state (function bot_get finds a value bound to a key in a map and returns Bot if the key is not bound). The abstract semantics is then applied to this configuration; it yields a list next of outcomes (see Figure 4.9) that are then propagated to the analysis state (function propagate). If the outcome is GiveUp, then the whole analysis aborts. Otherwise, if it is Run n′ ab — meaning that ab describes reachable configurations at program point n′ —, this abstract configuration is joined with the one previously associated with that program point (function bot_set updates a binding in a map and ensures that no key is ever bound to the Bot value). In case that something new is learned, the program point n′ is pushed on the worklist. If it is Hlt res, then the abstraction of the final value is updated similarly.

Since there may be infinite ascending chains, so as to ensure termination, we need to apply widening operators instead of regular joins frequently enough during the search. Therefore the analysis is parameterized by a widening strategy that decides along which edges of the control-flow graph widening should be applied instead of a plain join. The implementation allows to easily try different strategies. The one we implemented mandates a widening on every edge from a program point to a smaller one.

The analysis repeatedly applies the analysis step until the worklist is empty (see bottom of Figure 4.11). So as to ensure that the analysis indeed terminates, we rely on a counter (known as fuel) that obviously decreases at each iteration; when it reaches zero, the analyzer must give up.

```
Definition analysis_step (E:analysis_state) : analysis_state+⊤ :=
 match E.(worklist) with
 | nil => Just E (* fixpoint reached *)
 | n :: w =>
  match bot_get E.(result_fs) n with
  | NotBot ab_mc =>
   let next := ab_post_many n ab_mc in
   List.fold_left
   (λ acc res, do E′ <- acc; propagate (widen_oracle n res) E′ res)
   next
   (Just {| worklist := w
         ; result_fs := E.(result_fs)
         ; result_hlt:= E.(result_hlt) |})
  | Bot => All end end.

Definition propagate (widenp: bool) (E: analysis_state) (n: ab_post_res)
 : analysis_state+⊤ :=
 match n with
 | GiveUp => All
 | Run n′ ab =>
  let old := bot_get E.(result_fs) n′ in
  let new := (if widenp then widen else join) old (NotBot ab) in
  if new ⊑ old
  then Just E
  else Just {| worklist := push n′ E.(worklist)
            ; result_fs := bot_set E.(result_fs) n′ new
            ; result_hlt := E.(result_hlt) |}
 | Hlt res => (* similar case, not shown. *) end.

Fixpoint analysis_loop (fuel: nat) (E: analysis_state) : analysis_state+⊤ :=
 match fuel with
 | O => Just E
 | S fuel′ => do E′ <- analysis_step E;
              if is_final E′ then Just E′ else analysis_loop fuel′ E′ end.

Definition analysis (P: memory) (dom: list int) fuel : analysis_state+⊤ :=
 analysis_loop fuel (analysis_init (T.(init) P dom)).
```

Figure 4.11: Main loop of the Goto⋆ analyzer

This iteration strategy is different from the one used in the previous chapter (§ 3.2), which relies on the knowledge of the control-flow graph to compute, at the beginning of the analysis, the iteration order and the widening points. Here, the control-flow graph is discovered during the analysis; so the iteration order has to be adapted as the edges of the graph are discovered. Similarly, loops are not a priori known, and an optimal set of widening points cannot be pre-computed.

4.3.4 Soundness of the Abstract Interpreter

We now describe the formal verification of our analyzer. The soundness property we ensure is that the result of the analysis of a program P over-approximates its semantics $[\![P]\!]$. This involves on one hand a proof that the analysis result is indeed a fixpoint of the abstract semantics and on the other hand a proof that the abstract semantics is correct with respect to the concrete one.

The soundness of the abstract semantics is expressed by the following lemma, which reads: given an abstract state ab and a concrete one m in the concretization of ab, for each concrete small-step $m \Rightarrow m'$, there exists a result ab′ in the list ab_post_many m.(pc) ab that over-approximates m′. Our use of Coq type classes enables us to extensively overload the γ notation and write this statement in a concise way as follows.

```
Lemma ab_post_many_correct : ∀ (m: machine_config) (m′: machine_state) (ab: ab_mc),
  m ∈ γ(ab) → m ⇒ m′ → m′ ∈ γ(ab_post_many m.(pc) ab).
```

The proof of this lemma follows from the soundness of the various abstract transformers (as load_sound in Figure 4.7) and of the decoder:

```
Lemma abstract_decode_at_sound : ∀ (m: machine_config) (ab: ab_mc) (pp: addr),
  m ∈ γ(ab) → decode_from m.(mc_mem) pp ∈ γ(abstract_decode_at pp ab).
```

The proof that the analyzer produces a fixpoint is not done directly. Instead, we rely on *a posteriori* verification: we do not trust the fixpoint computation and instead program and prove a checker called validate_fixpoint. Its specification, proved thanks to the previous lemma, reads as follows.

```
Lemma validate_correct : ∀ (P: memory) (dom: list addr) (E: AbEnv),
  validate_fixpoint P dom E → [[P]] ⊆ γ(E).
```

Going through this additional programming effort has various benefits: on the one hand, a direct proof of the fixpoint iterator would be very hard; on the other hand, we can adapt the iteration strategy, optimize the algorithm and so on with no additional proof effort.

This validation checks two properties of the result E: that the result over-approximates the initial state; and that the result is a post-fixpoint of the abstract semantics, i.e., for each abstract state in the result, performing one abstract step leads to abstract states that are already included in the result. These properties, combined to the soundness of the abstract semantics, ensure the conclusion of this lemma.

Finally we pack together the iterator and the checker with another operation performed on sound results that checks for their safety. The resulting analysis enjoys the following property: if, given a program P, it outputs some result, then that program is safe.

```
Theorem analysis_sound : ∀ (P: memory) (dom: list addr) (fuel: nat) (int♯: num_dom_index),
  analysis int♯ P dom fuel ≠ None ⟶ P ∈ safe.
```

The arguments of the analysis program are the program to analyze, the list of addresses in memory to track, the counter that enforces termination and the name of the numeric domain to use. We provide two numeric domains: intervals with congruence information (also known as *strided intervals* [Bal07]) and finite sets.

To enhance the precision, we have introduced three more techniques: a dedicated domain to abstract the flag state, a partitioning of the state space, and a use of abstract instructions. They will be described in the next section.

4.4 Case Studies and Analysis Extensions

The extraction mechanism of Coq enables us to generate an OCaml program from our development and to link it with a front-end. Hence we can automatically analyze programs and prove them safe. This section shows the behavior of our analyzer on chosen examples, most of them taken from [CSV07] (they have been rewritten to fit our custom syntax). All examples are written in an assembly-like syntax with some syntactic sugar: labels refer to byte offsets in the encoded program, the enc(i) notation denotes the encoding of the instruction i. The study of some examples highlights the limits of the basic technique presented before and suggests to refine the analyzer as we describe below. These extensions have been integrated to our formalization and proved correct. The source code of all the examples that are mentioned thereafter is available on the companion web site [Web].

4.4.1 Basic Example

The multilevel run-time code generation program of Figure 4.12 is a program that, when executed, writes some code to the addresses starting at line gen and runs it; this generated program, in turn, writes some more code at line ggen and runs it. Finally execution starts again from the beginning. Moreover, at each iteration, register R6 is incremented.

The analysis of such a program follows its concrete execution and exactly computes the content of each register at each program point. It thus correctly tracks what values are written and where, so as to be able to analyze the program as it is generated.

However, when the execution reaches program point loop again, both states that may lead to that program point are merged. And the analysis of the loop body starts again. After the first iteration, the program text is exactly known, but each iteration yields more information about the dynamic content of register R6. Therefore we apply widening steps to ensure the termination of the analysis: the widening operator (of the memory domain) is used instead of the join operator on every edge from a program point to a smaller program point (i.e., in this example, from program point ggen to program point loop). Finally, the set of reachable program points is exactly computed and for each of them, we know what instruction will be executed from there.

Many self-modifying programs are successfully analyzed in a similar way: opcode modification, code obfuscation, and code checking [Web].

```
    cst 0 -> R6
    cst 1 -> R5
loop: add R5 -> R6
    cst gen -> R0
    cst enc(store R1 -> *R2) -> R1
    store R1 -> *R0
    cst enc(goto R2) -> R1
    cst gen + 1 -> R0
    store R1 -> *R0
    cst ggen -> R2
    cst loop -> R0
    cst enc(goto R0) -> R1
    goto gen
gen:  skip
    skip
ggen: skip
```

Figure 4.12: Multilevel run-time code generation

```
    cst -128 -> R6
    add R6 -> R1
    cmp R6, R1
    gotoLT ko
    cst -96 -> R7
    cmp R1, R7
    gotoLE ko
    store R0 -> *R1
ko:halt R0
```

Figure 4.13: Array bounds check

4.4.2 A First Extension: Dealing with Flags

The example program in Figure 4.13 illustrates how conditional branching relies on implicit flags. This program stores the content of R0 in an array (stored in memory from address -128 to address -96) at the offset given in register R1. Before that store, checks are performed to ensure that the provided offset lies inside the bounds of the array. The destination address is compared against the lowest and highest addresses of the array; if any of the comparisons fails, then the store is bypassed.

To properly analyze this program, we need to understand that the store does not alter the code. When analyzing a conditional branch instruction, the abstract state is refined differently at its two targets, to take into account that a particular branch has been taken and not the other. However, the only information we have is about one flag, whereas the comparison that set this flag operated on the content of registers. We therefore need to keep the link between the flags and the registers.

To this end, we extend our ab_machine_config record[5] with a field containing an optional pair of registers ab_reg: (reg × reg)+⊤. It enables the analyzer to remember which registers were involved in the last comparison (the All value is used when this information is unknown). With such information available, even though the conditional jump is not directly linked to the comparison operation, we can gain some precision in the various branches. More precisely, the compare operator can now be implemented as follows.

```
Definition compare ab rs rd := lift (λ ab', {| ab_flg := ⌊(rs, rd)⌋
                                ; ab_reg := ab'.(ab_reg) ; ab_mem := ab'.(ab_mem) |}) ab.
```

Back to the example of Figure 4.13, when we assume that the first conditional branch is not taken, the flag state is abstracted by the pair ⌊(R6,R1)⌋, so we refine our knowledge

[5]This record has been introduced in Section 4.3.1.

```
Definition assume (x: ab_machine_config+⊤) (f: flag) (b: bool)
 : ab_machine_config+⊤+⊥ :=
 match x with
 | ⌊x'⌋ =>
  match x'.(ab_flg) with
  | ⌊(Ru, Rv)⌋ =>
   let u := find_def x'.(ab_reg) Ru in let v := find_def x'.(ab_reg) Rv in
   let op := match f with FLE => Cle | FLT => Clt | FEQ => Ceq end in
   let v'u' := backward_int_binop (OpCmp op) v u (const_int (of_bool b)) in
   match v'u' with
   | (NotBot v', NotBot u') =>
    NotBot (⌊{| ab_reg := (x'.(ab_reg)) [ Ru <- u' ] [ Rv <- v' ]
            ; ab_flg := x'.(ab_flg); ab_mem := x'.(ab_mem) |}⌋)
   | _ => Bot
   end
  | _ => NotBot x
  end
 | _ => NotBot x
 end.
```

Figure 4.14: Implementation of the assume transfer function

about register R1: its content is not less than the content of register R6, namely −128. Similarly, when we assume that the second conditional branch is not taken, the abstract flag state is ⌊(R1,R7)⌋, so we can finally infer that the content of register R1 is in the bounds.

The actual implementation of such a precise assume relies on a *backward* transfer function of the numeric domain, discussed in Section 3.3.2. Given such a backward transfer function, assume can be implemented as shown in Figure 4.14: if the registers Ru and Rv involved in the last comparison are known, then the abstract values u and v associated to them can be refined using the backward operator for the given comparison. In case any of these refined values is ⊥, this information is propagated to the whole abstract state: the branch is unreachable and should not be analyzed any further.

Special care has to be taken in the assign transfer function. If a register that is part of the abstract flag is updated, then no information about its new content can be inferred from the outcome of the comparison. Therefore, in such cases, the abstract flag is simply forgotten, i.e., set to All.

This extension of the abstract domain has little impact on the formalization, but greatly increases the precision of the analyzer on programs with conditional branches. Indeed, without this extension, the analyzer cannot deduce anything from the guards of conditional branches as it ignores all comparison instructions.

4.4.3 A Second Extension: Trace Partitioning

During the execution of a self-modifying program, a given part of the memory may contain completely unrelated code fragments. When these fragments are analyzed, since they are stored at the same addresses, flow sensitivity is not enough to distinguish them.

```
 0: cst 40 -> R7
 2: cst 21 -> R0
 4: cst 13 -> R4
 6: load *R4 -> R1
 7: cst 14 -> R4
 9: load *R4 -> R2
10: cst 13 -> R4
12: store R2 -> *R4
13: goto R7
14: add R0 -> R3
15: cst 14 -> R4
17: store R1 -> *R4
18: cst 11 -> R4
20: load *R4 -> R2
21: cst 16 -> R4
23: load *R4 -> R1
24: store R2 -> *R4
25: cst 11 -> R4
27: store R1 -> *R4
28: cst 12 -> R4
30: load *R4 -> R2
31: cst 17 -> R4
33: load *R4 -> R1
34: store R2 -> *R4
35: cst 12 -> R4
37: store R1 -> *R4
38: goto 4
40:
```

Figure 4.15: Polymorphic program

If these fragments are merged in the abstract state, then the two programs get mixed and it is no longer possible to predict the code that is executed with sufficient precision. To prevent such a precision loss, we use a specific form of *trace partitioning* [Kin12] that makes an analysis sensitive to the value of a particular memory location.

Consider as an example the *polymorphic* program of Figure 4.15. Polymorphism here refers to a technique used by for instance viruses that change their code while preserving their behavior, so as to hide their presence. The main loop of this program (bytes 4 to 39) repeatedly adds forty-two to register R3 (two add instructions at bytes 13 and 14). However, it is obfuscated in two ways. First, the source code initially contains a jump to some random address (byte 13). But this instruction will be overwritten (bytes 7 to 12) before it is executed. Second, this bad instruction is written back (bytes 4 to 6 and 15 to 17), but at a different address (byte 14 is overwritten). The remainder of the loop swaps the contents of memory at addresses 11 and 16, and at addresses 12 and 17 (execution from byte 18 to byte 27, and from byte 28 to byte 37, respectively). So when the execution reaches the beginning of the loop, the program stored in memory is one of two different versions, both featuring the unsafe jump. In other words, this program features two variants that are functionally equivalent and look equally unsafe. And running any version changes the program into the other version.

When analyzing this program, the abstract state computed at the beginning of the loop must over-approximate the two program versions. Unfortunately it is not possible to analyze the mere superposition of both versions, in which the unsafe jump may occur. The two versions can be distinguished through, for instance, the value at address 12. We therefore prevent the merging of any two states that disagree on the value stored at this address. Two different abstract states are then computed at each program point in the loop, as if the loop were unrolled once.

More generally, the analysis is parameterized by a partitioning criterion δ: ab_mc $\to$ K that maps abstract states to keys (of some type K). No abstract states whose keys differ according to this criterion are merged. Taking a constant criterion amounts to disabling this partitioning. The abstract interpreter now computes for each program point a map from keys to abstract states (rather than only one abstract state).

```
Definition vpAbEnv : Type := (Map [ addr, Map [ K, ab_mc ] ] * int♯+⊥).
```

Such an environment E represents the following set of machine configurations (ignoring

the halted configurations represented by the second component):

$$\gamma(E) = \{c \in \mathsf{machine\_config} \mid \exists k,\ c \in \gamma\left((\mathsf{fst}\ E)[c.(\mathsf{pc})][k]\right)\}$$

This means that the actual key under which an abstract state is stored has no influence on the concrete states it represents. It can only improve the precision: if two abstract states x and y are mapped to different keys hence not merged, they can represent the concrete set $\gamma(x) \cup \gamma(y)$ which may be smaller than $\gamma(x \sqcup y)$.

For instance, the criterion used to analyze the polymorphic program of Figure 4.15 maps an abstract state m to the value stored at address 12 in all concrete states represented by m; or to an arbitrary constant if there may be many values at this address.

To implement this technique, we do not need to modify the abstract domain, but only the iterator and fixpoint checker. The worklist holds pairs (program point, criterion value) rather than simple program points. The iterator and fixpoint checker (along with its proof) are straightforwardly adapted. The safety checker does not need to be updated since we can forget the partitioning before applying the original safety check.

Thanks to this technique, we can selectively enhance the precision of the analysis and correctly handle challenging self-modifying programs: control-flow modification, mutual modification, and code encryption [Web]. However, the analyst must manually pick a suitable criterion for each program to analyze; the analyzer itself is not able to figure out what criterion to use. In practice, we have used the contents of some particular register or memory location.

When using this extension, the termination of the analysis may not be guaranteed any longer as the type K may have infinitely many values (or too many for the analysis to enumerate them all). To ensure termination, Kinder [Kin12] proposes a widening operator that merges keys at a particular program point when the number of different keys encountered at this program point exceeds some threshold. We did not implement such a widening operator and require the analyst to be careful when the partitioning criterion is designed.

4.4.4 A Third Extension: Abstract Decoding

The program in Figure 4.16 computes the n^{th} Fibonacci number in register R2, where n is an input value read from address -1 and held in register R0. There is a for-loop in which register R1 goes from 1 to n and some constant value is added to register R2. The trick is that the actual constant (which is encoded as part of an instruction and is stored at the address held in R6) is overwritten at each iteration by the previous value of R2.

When analyzing this program, we cannot infer much information about the content of the patched cell. Therefore, we cannot enumerate all instructions that may be stored at the patched point. So we introduce abstract instructions: instructions that are not exactly known, but of which some part is abstracted by a suitable abstract domain. Here we only need to abstract values using a numeric domain: the resulting instruction set is shown in Figure 4.17. This abstraction of the instructions could be pushed further to capture other self-modification patterns. For instance a program might modify only the encoding of a register; in such a case, the "register" part of the instructions could be abstracted by a finite set of registers.

```
    cst -1 -> R7
    load *R7 -> R0
    cst key+1 -> R6
    cst 1 -> R1
    cst 1 -> R2
loop: cmp R1, R0
    gotoLE last
    cst 1 -> R7
    add R7 -> R1
    cst 0 -> R3
    add R2 -> R3
key:  cst 0 -> R4
    add R4 -> R2
    store R3 -> *R6
    goto loop
last: halt R2
```

Figure 4.16: Self-modifying Goto⋆ program computing Fibonacci numbers

```
Inductive ab_instruction (int♯: Type) : Type :=
| AICst (v:int♯) (dst:reg) | AICmp (src dst: reg)
| AIBinop (op: int_binary_operation) (src dst: reg)
| AILoad  (src dst: reg) | AIStore (src dst: reg)
| AIGoto (tgt: int♯) | AIGotoInd (r: reg)
| AIGotoCond (f: flag) (tgt: int♯) | AISkip | AIHalt (r: reg).
```

Figure 4.17: Abstract instructions

With such a tool, we can decode *in the abstract*: the analyzer does not recover the exact instructions of the program, but only the information that some (unknown) value is loaded into register R4, which is harmless (no stores and no jumps depend on it).

This self-modifying code pattern, in which only part of an instruction is overwritten occurs also in the vector dot product example [Web] where specialized multiplication instructions are emitted depending on an input vector.

For this technique to be effective, the numeric abstract domain has to support it: mapping abstract values to abstract instructions (i.e., abstract decoding) should be more efficient than just enumerating all concrete values.

The abstract semantics (Figure 4.9) has to be slightly modified to deal with this new instruction set. In particular, all jumps behave like indirect jumps: their targets are only known as abstract values. For the analysis to follow such a jump, all concrete

```
  cst 0 -> R5
  cst j+1 -> R0
  cmp R5, R6
  gotoLE h
  store R7 -> *R0
j: gotoLE 0
h: halt R5
```

Figure 4.18: Not-a-branch

```
(* ... slice of ab_post_single ... *)
| (AIGotoCond f tgt, sz) =>
bot_cons (Run (pp + sz)) (T.(assume) m f false)
match T.(assume) m f true with
| NotBot m' =>
  match concretize tgt with
  | Just tgt => IntSet.fold (λ acc tgt, Run tgt m' :: acc) tgt nil
  | All => GiveUp :: nil
  end
| Bot => nil
end
```

Figure 4.19: Abstract conditional jump

Program	Result	Comment
Opcode modification	✓	
Multi-level run-time code gen.	✓	
Bootloader	✘	needs a model of system calls and interrupts
Control-flow modification	✓	partitioning on the jump target (address 15)
Vector dot product	✓	partitioning on loop counter (register R0); and abstract decoding
Run-time code checking	✓	
Fibonacci	✓	abstract decoding
Self-replication	✘	code segment is "infinite"
Mutual modification	✓	partitioning on the instruction to write (held in register R0)
Polymorphic code	✓	partitioning according to the different versions of the program (e.g., address 12)
Code obfuscation	✓	
Code encryption	✓	partition on the loop counter (register R0)

Figure 4.20: Summary of self-modifying examples

destinations need to be enumerated. However, consider the example shown Figure 4.18. The conditional jump on line j is unsafe: its target is overwritten on the line before with some input value (the contents of register R7). But the whole program is actually safe, because the branching condition is always false[6]. Therefore, the abstract transformer for conditional jumps (Figure 4.19) tries to prove that the branch cannot be taken (with the call T.(assume) m f true) before it enumerates its possible targets; and the program of Figure 4.18 can be proved safe by our analyzer.

The techniques presented here enable us to automatically prove the safety of various self-modifying programs including almost all the examples of Cai *et al.* [CSV07], as summarized in Figure 4.20. Out of twelve, only two cannot be dealt with. The *comment* column of the table lists the extensions that are needed to handle each example (if any), or the limitation of our analyzer. The boot loader example does not fit in the considered machine model, as it calls BIOS interrupts and reads files. The self-replicating example is a program that fills the memory with copies of itself: the code, being infinite, cannot be represented with our abstract domain. Our Coq development features all the extensions along with their correctness proofs, and several Goto* examples including the implementation of the programs listed in Figure 4.20.

4.5 Related Work

Most of the previous works on mechanized verification of static analyses focused on standard data-flow frameworks [KN06; CD04; BGL06; Cac+05] or abstract interpretation for small imperative structured languages [Ber09; CP10; Nip12]. Klein and Nipkow instantiate

[6]Such spurious branching instructions are known as "opaque predicates" and are mainly used for obfuscation [CTL98].

such a framework for inference of Java bytecode types [KN06]; Coupet-Grimal and Delobel [CD04] and Bertot *et al.* [BGL06] for compiler optimizations, and Cachera *et al.* [Cac+05] for control flow analysis.

Our current work formalizes advanced abstract interpretation techniques, complementary to the ones described in the previous chapter, but targeting self-modifying low-level code; it is based on several recent non-verified static analyses. A large amount of work was done by Balakrishnan *et al.* in this area [BR10]. Control-flow graph reconstruction was specially studied by Kinder *et al.* [KV08] and Bardin *et al.* [BHV11]. Still, these works are unsound with respect to self-modifying code.

The first formal semantics for self-modifying programs are defined by Debray *et al.* [DCT08] and Bonfante *et al.* [BMR09]. These are paper-and-pencil semantics that operate over small low-level languages as ours; they are used in a very different way from our semantics. In the first one, a denotational semantics based on traces is used to identify the different phases of successive program modifications during a program execution. An operational semantics is defined in the second one, where a rewriting process from self-modifying programs to non-modifying programs is defined.

Our current work tackles a core subset of a self-modifying low-level programming language. More realistic formalizations of x86 semantics were proposed [Myr10; Mor+12; Ken+13] but none of them handles the problem of disassembling self-modifying programs. Our work complements other verification efforts of low-level programs [Chl11; CSV07; JBK13] based on program logics. While we provide automatic inference of loop invariants, they are able to handle more expressive correctness properties.

The RockSalt project [Mor+12] features a formally verified checker of the safety policy of Native Client, a facility for running untrusted binary programs within a web browser. The safety policy is draconian (and in particular implies that the code is not self-modifying), so that checking it is relatively simple; therefore, the verification of the checker is in some aspect simpler than the verification of our value analysis. However, RockSalt tackles a large part of the x86 syntax including the many addressing modes, the quirky encoding (prefixes, w bit) etc. Their formalization is particularly interesting in that it features a core intermediate representation, which is then used to define the semantics of all supported instructions of x86.

So as to precisely analyze programs that are obfuscated by *virtualization*, Kinder [Kin12] proposes an abstract domain that is similar to our ad-hoc trace partitioning: a particular location is identified as holding the virtual program counter and the abstract domain is partitioned on the value at this location. A more general trace partitioning domain is proposed by Rival and Mauborgne [RM07]. The formal verification in Coq of such techniques is beyond the scope of this work.

4.6 Conclusion and Perspectives

This work provides the first verified static analysis for self-modifying programs. In order to tackle this challenge, we formalized original techniques such as control-flow graph reconstruction and partitioning. We formalized these techniques on a small core language but we managed to verify ten out of twelve of the challenges proposed by Shao *et al.* [CSV07].

An important further work is to scale these technique on more realistic Coq language

models [Mor+12; Ken+13]. Developing directly an analyzer on these representations may be a huge development task because of the number of instructions to handle. One strategy could be to relate on a good intermediate representation such as the one proposed by Rocksalt [Mor+12]. It could be very interesting to integrate our analyzer architecture within their language formalization; it would in particular enable the handling of compiled programs whereas our analyzer is currently limited to tediously hand-written Goto* code.

Our language lacks some realistic features, as for instance floating-point arithmetic or SIMD instructions. More importantly, there is no mechanism as interrupts or system calls to interact with the environment and perform run-time I/O. Also, there are no explicit instructions for function call and return, though they could be emulated with computed jumps. Our current work does not consider the specific challenge of call stack reconstruction [BR10; Fle+10] that may require some form of verified alias analysis [RL12]. This is an important place for further work.

More generally, our memory domain signature prevents any symbolic reasoning on memory addresses: all accessed addresses have to be exactly known at analysis time. Also, the lack of expressions in the interface prevents the computation of relational invariants, i.e., properties that may relate the contents of different registers and memory cells.

Next chapter is dedicated to a more general and more powerful memory domain. It is targeted at the analysis of the CFG language in which the control-flow is statically known and memory is a little bit structured; it implements the memory domain signature of previous chapter (Figure 3.11).

Chapter 5

Precise abstraction of C-like memories

The value analyses that have been presented in the two previous chapters feature a *memory domain*. This component is in charge of representing the data manipulated by the analyzed program, and of modeling basic instructions that affect this data (as opposed to the control that is handled by the iterator). The implementations of such domains in previous chapters have some limitations. The memory domain of chapter 3 computes precise information about the contents of the registers only and ignores the contents of the memory. The domain of chapter 4 is a simple map from addresses to abstract numerical values: it cannot represent relational invariants about the contents of different locations and needs, when modeling a memory access, to know the exact (concrete) address being accessed. It also operates on a simple memory in which all accesses target only one byte, whereas common languages (as C) usually provide means to access several bytes at once.

A basic and important component of a memory domain is usually a (relational) numerical domain. An extensive literature is dedicated to them [Min04; CC76; Gra89; Kar76; Gra91; CH78], various efficient libraries are available and they usually agree on their interface, similar to the one given in a previous chapter (§ 3.3.1). These domains operate on variables that need to be explicitly named: they do not permit referencing variables through pointers, and no wonder about pointer arithmetic.

Analyses of programs with pointers require dedicated treatments, known as *points-to* analysis. Indeed, the analysis needs to predict pointer targets in order to infer something about a value accessed through said pointer. However points-to analysis in presence of pointer arithmetics requires a numerical analysis: when the offset of a pointer is expressed as an arbitrary arithmetic expression, predicting the values of this offset amounts to analyzing this expression.

Such an analysis that integrates points-to and numerical analyses has been described by Miné [Min06]. We propose in this chapter to implement a memory domain that follows that work and integrates into the value analyzer for the CFG intermediate representation of the CompCert compiler presented in chapter 3. This implementation is also mechanically verified in Coq.

This contribution is part of a larger work that has been recently published [Jou+15]. This chapter goes into the details of the memory domain, that was only briefly sketched in that publication.

5.1 Background

The value analysis discussed in this chapter builds up on the one of chapter 3. The main difference with that work is the memory domain at the heart of this chapter. We briefly

```
Class pre_mem_dom (t: Type) : Type := {
 (* abstract domain structure *)
 mem_wl : weak_lattice t;
 (* consult the range of a local variable *)
 range: t → ident → signedness → itv+⊥;
 (* assignment to a local variable *)
 assign: ident → expr → t → t+⊥;
 (* assignment to a memory cell *)
 store: memory_chunk → expr → expr → t → t+⊥;
 (* assume an expression evaluates to non-zero value *)
 assume: expr → t → t+⊥;
 (* non-deterministic assignment to a local variable *)
 forget: ident → t → t+⊥
}.
```

Figure 5.1: Interface of the memory domain

recall the overall design of the analyzer and its two main components: an abstract domain and a fixpoint solver.

The fixpoint solver takes as input the code of a function and an abstract semantics of the CFG programming language. It then computes a flow-sensitive invariant that is sound w.r.t. the given semantics. It uses Bourdoncle's algorithm [Bou93] programmed in OCaml; the output of this solver is therefore not trusted and validated *a posteriori*.

The abstract domain is a data-type whose values represent sets of concrete memory states (local variables and global memory). Its signature is given on Figure 5.1. The type of the abstract values is `t`. The domain is equipped with a lattice structure (`mem_wl`) and a query operator (`range ab x`) that returns an over-approximation of the values of local variable `x` in any state represented by `ab`, as a pair of intervals (one if the value is interpreted as a *signed* integer, one if it is interpreted as an *unsigned* integer; this prevents a premature loss of precision due to an early choice of the signedness interpretation). It also features operators that model CFG statements: `assign`, `store` and `assume`. The last operator, `forget`, performs a non-deterministic assignment and can be used to model inputs, for instance. The `t+⊥` notation refers to the type `t` extended with an extra `Bot` element whose concretization is empty: it means that a contradiction has been found and that no concrete state can satisfy the given constraints. For instance, when analyzing a conditional branch, the analyzer may prove that the condition never holds, i.e., that the branch cannot be taken, hence returns `Bot`. There is no operator in this interface that enables to model function calls. Therefore, the analysis will be restricted to programs without functions calls. This is a limitation that will be relaxed in a following chapter (§ 6.2) where inter-procedural analysis is discussed. Meanwhile, we consider programs without function calls, or, equivalently, programs without recursion nor function pointers in which all function calls can be inlined before the analysis.

A key building block of such an abstract domain is a (relational) numerical domain. It has similar operators (`range`, `assume` and `assign`) that rely on numerical expressions (no loads, no pointers) of type `nexpr`. The precise definition of a numerical domain is given in

a previous chapter (§ 3.3).

The remainder of this chapter is organized as follows. Section 5.2 gives a high-level overview of the implementation of such a memory domain, and Section 5.3 a more detailed description of this domain. Section 5.4 reviews its soundness proof. Finally, Section 5.5 describes some extensions to the domain that enhance the precision of the analysis.

5.2 Overview of the Memory Domain

Relational numerical domains are well suited to represent numerical environments, i.e., concrete sets of functions from named variables to numbers. The operators of such domains always refer to the variables through their names (as in "assign $x + 2 \cdot y$ to z"). Such a domain cannot be directly applied to the analysis of languages like C in which: values are not only numerical but also comprise pointers; and variables are referred to by expressions which may involve arithmetic (as in "assign $*(x + 2 \cdot y)$ to $*(z+4)$", where $*p$ denotes the dereference of pointer p). Such an assignment may modify several memory locations: the exact variable targeted by a pointer expression may not be known statically. The analyzer must account for this uncertainty and consider that any of the possibly targeted location has been updated.

Programs to analyze manipulate data that is stored in various locations: registers, local and global variables, dynamically allocated memory. The numerical content of these locations is represented as a point in a (relational) numerical domain. The "numerical content" refers to the actual value of integers and the offset part of pointers.

One of the main tasks of the memory domain is to translate the queries that it receives in terms of CFG expressions (with pointer arithmetic and memory loads) into queries for the numerical domain, in terms of purely numerical expressions.

Let's consider an example program to be analyzed. The analyzer operates on the CFG intermediate representation but the program is written here in concrete C syntax for the sake of readability. Note that scaling (i.e., multiplication of the offset by the size of target type) is explicit in CFG expressions (but hidden here, when incrementing pointers).

```
1 int S[2], T[2];
2 int main(void) {
3   int b1 = any_bool(), b2 = any_bool(), *x = S, *p = T;
4   S[0] = T[1] = 0; S[1] = T[0] = 1;
5   if (b1) p = S; if (b2) ++p;
6   x = x + *p;
7   return *x;
8 }
```

In this program, there are two global variables (arrays `S` and `T`), that are initialized at the beginning of the `main` function. This program builds a pointer `p` to some element of the arrays `S` and `T`, depending on the values of `b1` and `b2` that can be seen as input in this example (as suggested by the assignment to `any_bool()`; we will come back on the modeling of user input in next chapter, § 6.5). It then uses the value of this element as an offset in array `S` (pointer `x`) and returns the value referenced by `x`.

The following pictures schematically shows a possible concrete state when reaching line 6. Variables `S` and `T` are allocated respectively to blocks 100 and 101. Registers `b1` and `b2` contain respectively integers 0 and 1. Therefore `p` points to offset 4 in block 101.

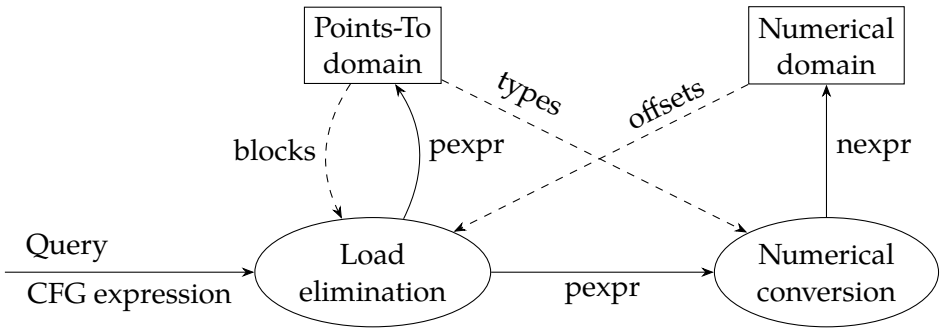


Figure 5.2: Sketch of the memory domain

```
Memory:                          Registers:

     +----------+----------+        b1: Vint 0    p: Vptr 101 4
100: |  Vint 0  |  Vint 1  |        b2: Vint 1    x: Vptr 100 0
     +----------+----------+

     +----------+----------+
101: |  Vint 1  |  Vint 0  |
     +----------+----------+
     0          4          8
```

We will focus on line 6. When analyzing this line, the memory domain is asked to model the assignment to variable x of the value resulting from the evaluation of expression x + *p. This query is expressed using CFG expressions, which cannot be directly given to the numerical domain. One difficulty in answering this query lies in the load; the memory domain needs to predict what are the locations that may be targeted by pointer p.

To do so, the memory domain embeds a (generic) numerical domain (as shown on the right of Figure 5.2) and a points-to domain (at the top). Queries addressed to the memory domain flow (plain arrows) to the points-to and numerical domains. On the way, they are converted to load-free queries for the points-to domain (using load-free expressions of type pexpr) and numerical queries for the numerical domain (using numerical expressions of type nexpr). These conversions use (dashed arrows) the information provided by these domains: the numerical conversion uses the points-to information to distinguish integers from pointers and the load-elimination uses the both points-to and numerical information to resolve loaded addresses.

We now roughly describe each component of the memory domain.

5.2.1 Memory Cells

In the CFG language, when a program accesses an array element or a structure field (as in "S[1] = 1"), the variable (here S) is not fully involved but only a chunk of it. Therefore, we introduce a notion of abstract cell (and the corresponding data-type acell) to represent locations that are accessed by the program. Such an abstract cell is either a chunk of a global variable or a register. (The discussion of stack-allocated local variables is delayed until Section 5.5.1.)

```
Inductive acell : Type :=
| ACglobal (s: ident) (κ: memory_chunk) (ofs: Z)
| ACreg (x: ident).
```

The memory chunk in the global-variable location describes how this cell is accessed (in particular, it gives the size of the cell and how to interpret its contents). There is no such meta-data for the register location, since the contents of registers are accessed directly and as-is (i.e., without transformation nor reinterpretation of the data).

For instance, the (abstract) memory cells involved in the analysis of our example program are: ACglobal S Mint32 0 and ACglobal S Mint32 4, that represent the two elements of array S; ACglobal T Mint32 0 and ACglobal T Mint32 4, that represent the two elements of array T; ACreg b1, ACreg b2, ACreg p, and ACreg x, that represent the four register variables of this program.

These abstract cells are used as variables of the numerical and points-to domains: they operate as if the program were manipulating cells rather than anything else. The role of the memory domain is to make this illusion correct, by converting queries about CFG expressions into queries about cells.

Notice that cells may overlap: two cells may refer to the same address with different chunks, or to overlapping chunks of the memory. For instance, if the program wrote a 64 bits integer at the address of array T, the cell ACglobal T Mint64 0 would be used to describe this access; and this cell overlaps with the two other cells about T. Most often, the abstract domains will not compute invariants about overlapping cells. Would this happen, the conjunction of these invariants would apply to the concrete part of memory they represent. Therefore, we need to take care of overlapping cells on stores (all possibly written cells have to be updated) rather than on loads (reading only one of the read cells always return a sound invariant).

5.2.2 Points-to Domain

In order to precisely understand expressions involving loads and pointers, the memory domain needs to: distinguish pointers from integers; and predict the set of blocks a pointer may target. We thus attach to each cell a type (integer (Int), pointer (Ptr) or unknown (All)) and, if it is a pointer, a finite set of blocks.

In the example program, when reaching line 6, the points-to domain state is as depicted in the following table.

Cell	ACglobal S Mint32 0	ACglobal S Mint32 4		ACglobal T Mint32 0	
Points-to	Int	Int		Int	
Cell	ACglobal T Mint32 4	ACreg b1	ACreg b2	ACreg p	ACreg x
Points-to	Int	Int	Int	Ptr({S; T})	Ptr({S})

5.2.3 Underlying (Relational) Numerical Domain

In addition to the points-to domain, the memory domain embeds a relational numerical domain whose abstract values (of type num) represent sets of functions from (abstract) cells to numerical values (machine integers, of type int). The signature of such domain

is given in Section 3.3.1. It is used to represent the numerical part of the contents of the abstract cells: the values of integers and the offsets of pointers.

5.2.4 Conversion

To analyze an instruction as the one on line 6 above, the numerical domain needs to translate the CFG expression `x + *p` into numerical expressions to hand them over to the numerical domain. This translation is decomposed into two steps: elimination of the loads; and elimination of the pointers. Theses two steps are now described.

Load elimination In order to eliminate a sub-expression of the form `*e`, the memory domain first computes an over-approximation of the set of cells that may be designated by expression `e`. This results in a set of expressions, replacing in the original expression the load `*e` by every possible cell it may read from.

To compute this set of cells, the loads from sub-expression `e` are recursively eliminated; this yields a set of expressions without loads. Then each expression is given to: a points-to domain that computes a set of blocks (i.e., names of global variables); and the numerical domain (after pointer-elimination) that computes a *concrete* set of offsets.

In our example, the load to eliminate is the dereference of `p`. The points-to evaluation may result in the set { `S`; `T` } since this pointer points inside one of these two arrays. The numerical evaluation may result in the set { `0`; `4`}. Finally, we get a set of four load-free expressions: `ACreg x + 4 × (ACglobal S Mint32 0)`, `ACreg x + 4 × (ACglobal S Mint32 4)`, `ACreg x + 4 × (ACglobal T Mint32 0)`, and `ACreg x + 4 × (ACglobal T Mint32 4)`.

Pointer elimination The expressions resulting from the load-elimination may still contain pointers: pointer constants or pointer arithmetic. To translate such expressions into purely numerical ones, constants are replaced by their offsets and operators by their numerical counterparts. This translation is mostly unsurprising: addition is mapped to addition and so on. The (only) non-trivial point is about boolean operations: all (non-null) pointers are *true*.

Therefore, to correctly translate a boolean operation (e.g., the C expression `!q`), the analyzer needs to predict whether `q` is always a non-null pointer (in which case this expression is always false) or it is always an integer (in which case this expression is already purely numerical) or may be any of them (in which case this expression may return any boolean). This information is available thanks to the points-to domain.

In the next section, we look more precisely at each component of this domain.

5.3 Technical Details

One central operation of this memory domain is conversion. It relies on an intermediate expression language that we present first. Then we dig into the details of the two phases of the conversion. Finally we describe how the abstract transformers are implemented.

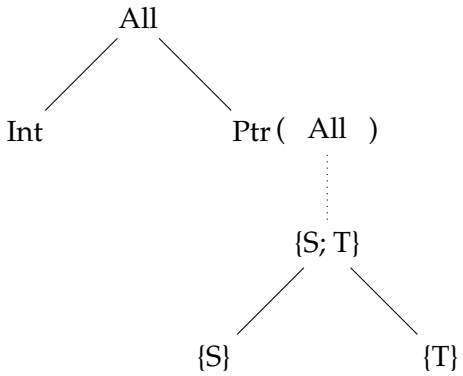


Figure 5.3: Lattice of types and points-to abstract values

5.3.1 Pointer Expressions

The pointer-expressions, of type pexpr, are CFG expressions without loads, or equivalently numerical expressions with pointers. They are parameterized by the type of the variables they may contain.

```
Inductive pexpr (var: Type) : Type :=
| PEvar (v: var)
| PEconst (cst: constant)
| PEunop (op: unary_operation) (e: pexpr var)
| PEbinop (op: binary_operation) (e₁ e₂: pexpr var)
| PEcond (b e₁ e₂: pexpr var).
```

They use the same constants (type constant) and the same operators (types unary_operation and binary_operation) as plain CFG expressions.

The concrete semantics of these expressions is very similar to the one of CFG expressions, except that it does not depend on a memory (for loads) but only on the permissions (for pointer comparisons).

5.3.2 Points-to Domain

The purpose of this domain is two-fold:

1. distinguish cells holding integers from cells holding pointers; and
2. predict the set of blocks a pointer may target.

This domain is non-relational and independent from the numerical domain. It attaches an abstract value to each (abstract) cell. The abstract value combines a type information (integer, pointer, or don't-know) and, in case of pointers, a points-to set.

This set is either a subset of the finite sets of all global variables, or the special All value that denotes any block and is used, for instance, to abstract pointers to stack-allocated variables. We finally get a semi-lattice (of type pointsto+⊤) that can be pictured as Figure 5.3.

This domain features in particular a forward evaluation function,

```
eval_ptr: Map [ acell, pointsto ] ⟶ pexpr acell ⟶ pointsto+⊤
```

that, in a context of an abstract state that maps each cell to its points-to approximation, given an expression, computes the points-to information corresponding to the result of the evaluation of this expression.

The abstract memory type is thus a pair made of a points-to information, and a point in the numerical domain:

```
Definition t : Type := ( Map [ acell, pointsto ] * num ).
```

5.3.3 Numerical Conversion

The second phase[1] of conversion, implemented by the function nconvert, produces a numerical expression (of type nexpr) from a pointer expression (of type pexpr). Its code is shown at the top of Figure 5.4.

To convert a pointer-expression, it is recursively traversed and its structure is mostly kept; e.g., a negation of an integer expression is mapped to the negation of the converted sub-expression. When converting constants, integers are kept, floats are forgotten, and pointers are replaced by their offsets. Non-trivial cases occur when converting boolean expressions. For instance, the Oboolval unary operator casts its argument into a boolean (i.e., integer zero or one). To correctly convert such an expression, we need to distinguish whether the argument is an integer or a pointer:

- if it is an integer, the conversion goes on as usual;
- if it is a pointer, the whole expression is mapped to the constant expression that evaluates to integer one; indeed, all (non-null) pointers are true;
- if its type is not known at analysis time, the result is soundly over-approximated by the expression that may evaluate to integers zero and one.

5.3.4 Load Elimination

The first phase of the conversion, implemented by the function convert, is shown at the bottom of Figure 5.4. It produces a set of pointer expressions out of one CFG expression (or gives up and returns All). It traverses an expression, converts sub-expressions, and combines all possible results (as in a regular set monad). When encountering an expression of the form (Eload κ e′), the sub-expression e′ is recursively converted. This produces a set lp of pointer-expressions. This set is dereferenced, i.e., the set of cells that may be targeted by these expressions is computed: each expression pe in the set lp is evaluated by the points-to domain; this yields a set bs of blocks; the expression pe is also converted to a numerical expression (as described in Section 5.3.3) that, thanks to the concretize operator of the numerical domain, yields a *concrete* set of offsets (or gives up, if the result is not known to be small enough). The Cartesian product of these two sets produces a concrete set of cells. This logic is implemented by the deref_pexpr function, whose code follows.

```
Definition deref_pexpr (ab: t) κ (lp: set (pexpr acell)) : set acell+⊤ :=
 let (pt, nm) := ab in
```

[1]The first phase depends on the second one and is described in the next section.

```
Definition nconvert_constant (cst: constant) : nconstant :=
 match cst with
 | Ointconst n => NOintconst n
 | Ofloatconst n => NOintunknown
 | Oaddrsymbol _ ofs | Oaddrstack ofs  => NOintconst ofs end.

Fixpoint nconvert (pe: pexpr acell) : nexpr acell :=
 match pe with
 | PEvar x => NEvar x
 | PEconst cst => NEconst (nconvert_constant cst)
 | PEunop op pe' => match op with
  | Onegint | Onotint => NEunop op (nconvert pe')
  | Oboolval => match eval_type pe' with
              | Just PtInt => NEunop op (nconvert pe')
              | Just (PtPtr _) => ne_true
              | All => any_bool end end
 | PEbinop op pe₁ pe₂ => match op with
  | Oadd | ...  => NEbinop op (nconvert pe₁) (nconvert pe₂)
  | Ocmpu c =>
    match eval_type pe₁, eval_type pe₂ with
    | All, _ | _, All => any_bool
    | Just ty₁, Just ty₂ =>
     match ty₁, ty₂ with
     | PtInt, PtInt => NEbinop op (nconvert pe₁) (nconvert pe₂)
     | PtInt, PtPtr _
     | PtPtr _, PtInt => match c with Cne => ne_true | _ => ne_false end
     | PtPtr bs₁, PtPtr bs₂ => ...
     end end end
 | ... end.

Fixpoint convert (e: expr) (ab: t) : (set (pexpr acell))+⊤ :=
 match e with
 | Evar s => Just (singleton (PEvar (ACreg s)))
 | Econst cst => Just (singleton (PEconst cst))
 | Eunop op e' => do_top pe' <- convert e' ab;
              Just (SetConstructs.map (PEunop op) pe')
 | Eload κ e' => do_top lp <- convert e' ab;
             do_top cells <- deref_pexpr ab κ lp;
             Just (SetConstructs.map PEvar cells)
 | ... end.
```

Figure 5.4: Conversion (excerpt)

```
SetInterface.fold
(λ (pe: pexpr acell) (s: set acell+⊤),
  do_top s' <- s;
  do_top ptr <- eval_ptr pt pe;
  match ptr with | PointsTo.PtPtr bs =>
    do_top bs <- bs;
    let ne := nconvert pt pe in
    do_top ofs <- concretize ne nm;
    Just (cell_product bs κ ofs ++ s')
  | _ => s end)
lp (Just {}).
```

In the example from section 5.2, to analyze the line 6, one CFG expression to convert is (in a more readable syntax): EVar x + 4 × ELoad Mint32 (EVar p). To do so, we first dereference the sub-expression (EVar p):

1. This sub-expression is converted to a set of pointer expressions, namely to the singleton {PEVar (ACreg p)}.

2. All expressions in this set (there is only one) are evaluated (in parallel) in the points-to domain and in the numerical one. In the points-to domain, cell ACreg p is mapped to the value Ptr({ S; T}) which represents all pointers to some cell of blocks to which global variables S or T are allocated. Evaluation in the numerical domain may result in the precise set { 0; 4}. [2]

3. The Cartesian product of these sets yields a set of four abstract cells:

   { ACglobal S Mint32 0; ACglobal S Mint32 4; ACglobal T Mint32 0; ACglobal T Mint32 4 }.

Then conversion proceeds and builds an expression PEVar x + 4 × c for each cell c from the above set.

5.3.5 Abstract Transformers

Conversion enables the implementation of the operators that model CFG instructions: forget, assume, assign, and store. We describe their design in this section.

Forget

The forget operator over-approximates any instruction that may overwrite a temporary in an unknown way, e.g., an input instruction. During the analysis, we need a more general operator that enables to forget the content of any cell. We therefore provide a forget_cells function that forgets anything about a set of cells. It removes them from the points-to map and calls the corresponding forget operator of the numerical domain. This operation involves no conversion.

[2] Such precision is achieved using a product of interval and congruence domains.

```
Definition assume (e: expr) (ab: t) : t+⊥ :=
 match convert (Eunop Oboolval e) ab with
 | All => NotBot ab
 | Just pes =>
  let (pt, nm) := ab in
  set_map_reduce (λ pe,
   do_bot pt' <- pt_assume pe pt;
   do_bot nm' <- assume (nconvert pt' pe) nm;
   NotBot (pt', nm')
  ) pes
 end.
```

Figure 5.5: Implementation of assume

```
1  int x, y, z;
2
3  int main(void)
4  {
5    int *p = any_bool() ? & x : & y;
6    if ( x < z && y < z) {
7      if ( 0 < *p ) {
8        assert ( 1 < z );
9      }
10   }
11   return 0;
12 }
```

Figure 5.6: Test case for assume

Assume

The purpose of the assume e ab transformer is to refine an abstract state ab to take into account the fact that expression e evaluated to a *true* value. Its implementation is given in Figure 5.5.

The expression is first prefixed by the operator Oboolval to ensure that the result is actually cast to a boolean. The resulting expression is then converted to a pexpr in order to eliminate the loads in the expression. This conversion may fail, in which case nothing new can be learned. Otherwise, the conversion returns a set pes of pointer expressions.

All these expressions are then given to the pt_assume transformer of the points-to domain and to the assume transformer of the underlying numerical domain (recall that nconvert casts a pointer-expression into the corresponding numerical expression).

The combinator set_map_reduce iterates over the set pes, runs the function it gets as argument in parallel for each element pe in the set, then joins all the results.

The pt_assume operator is in charge of refining the points-to information. It cannot do anything useful yet. It will be discussed in more details later (§ 5.5.3).

Example Consider the analysis of the program shown on Figure 5.6, using a relational numerical domain (e.g., polyhedra) that is able to infer, after the first if, that z is larger than both x and y. When the analysis reaches line 8, it knows that the guard 0 < *p is true. However, pointer p targets either x or y, so conversion of the guard yields two numerical expressions: one about x, the other about y. Since in both cases the numerical domain is able to infer that z is larger than one, in spite of uncertainty about the target of p, the analysis is able to prove the assertion on line 8.

This implementation is not the most precise that can be thought of. To illustrate the loss of precision, consider the following C snippet.

```
1  int x = 0, y = 1; int *p = any_bool() ? & x : & y;
2  if ( *p ) { /* p points to y */ } else { /* p points to x */ }
```

The conversion of the expression *p yields two expressions: one corresponding to variable x, one corresponding to variable y. The numerical domain is then able to prove that assuming that x is true leads to a contradiction. However, this information is not propagated

```
Definition assign (x: ident) (e: Cminor.expr) (ab: t) : t+⊥ :=
  assign_cells None (singleton (ACreg x)) e ab.

Definition store (κ: memory_chunk) (ℓ r: Cminor.expr) (ab: t) : t+⊥ :=
  match (do_top ptr <- convert ab ℓ; deref_pexpr ab κ ptr) with
  | All => NotBot( ⊤ ) (* give up: the whole heap may be modified *)
  | Just cells => assign_cells (Some κ) cells r ab
  end.

Definition assign_cells (κ: option memory_chunk) (dst: set acell) (e: expr) (ab: t) : t+⊥ :=
  match convert ab e with
  | All => forget_cells (overlapping_cells dst ++ dst) ab
  | Just pes =>
    let (pt, nm) := ab in
    do_bot ab′ <- set_map_reduce (λ c,
      do_bot res <- set_map_reduce (λ pe,
        let ty := eval_ptr pt pe in
        if chunk_type κ ty then
          let ne := nconvert pt (ensure_cast_for_chunk κ pe) in
          do_bot nm′ <- assign c ne nm;
          NotBot (ty, nm′)
        else NotBot (⊤))
      ) pes;
      let (ty, nm′) := res in NotBot (map_assign pt c ty, nm′)
    ) dst;
    forget_cells (overlapping_cells dst) ab′ end.
```

Figure 5.7: Updating cells: assign and store

to the points-to domain. Indeed no information about the choices that are made during the conversion is kept. To address this limitation, the conversion could not only produce a set of expressions, but a set of pairs made of an expression and an abstract-environment, where the abstract environment is the original one refined with the choices done during conversion. This would also address the precision loss in expressions like *x + *x. Such an improvement is left as future work.

5.3.6 Assign & Store

Both operators assign and store, whose implementation is given in Figure 5.7, share the same logic: evaluate an expression, and store its result at some location. The difference between the two is that the destination is definitely known in the case of assign whereas it is denoted by an expression in the case of store. Therefore, this second operator needs first to compute an over-approximation of the set of cells that may be designated by this expression (thanks to deref_pexpr, also used in the load elimination phase). Then, both operators rely on the more general assign_cells that updates a set of cells.

Abstract cells may overlap. Therefore, an explicit update to one cell may hide implicit

updates to overlapping cells. To soundly model this property, we conservatively forget anything that is known about any overlapping cell. To do so we rely on an auxiliary function overlapping_cells that computes a set of all cells that are distinct from all cells in a given set but overlap with some of them.

To assign to a set of cells the result of a given expression e, this expression is first converted to a set of pointer-expressions. In case of failure of the conversion, the target cells (and the overlapping ones) are erased. Otherwise, for each destination cell c and each pointer expression pe resulting from the conversion, the assignment of this expression to that cell is performed, in parallel, in the points-to domain and in the numerical one. All these parallel assignments are then joined together. This results in a strong update if the destination set is a singleton; a weak update otherwise. Finally, all overlapping cells are erased.

Abstract cells have a memory chunk as meta-data. It simplifies the view of the memory. Otherwise when reading a cell, we would have to decode its contents. In contrast, we chose to perform the trans-coding on stores. That's why there is first a check of the type of the value to store. And the expression is changed to add an explicit cast, so that the numerical domain knows that the value has to be trans-coded. For instance, when adding a cast to 8-bit unsigned integer, the numerical domain truncates the abstract value to the interval $[0; 255]$.

5.4 Soundness

We have seen so far the implementation of the memory domain. We now move on its soundness proof. We first introduce an intermediate specification of the concrete memory in Section 5.4.1, then discuss the concretization relation of the points-to domain in Section 5.4.2 and of the whole memory domain in Section 5.4.3. Finally we present the central lemmas of the soundness proof in Section 5.4.4.

5.4.1 Functional Memory

The (concrete) memory model of CompCert has been defined to enable the specification of various programming language semantics, and the verification of the compiler. Unfortunately, it is not very convenient for the verification of the abstract domain. Therefore, we introduce an intermediate specification of the concrete memory which is structurally close to the abstract memory. This concrete memory is made of *cells*. Each such cell represents a location from which some data can be fetched. A memory is then simply a *total* function from cells to values. The value Vundef represents uninitialized data and thus enables to see the memory as a total function.

There are two kinds of cells: cells that are in memory (global variables and local variables whose address is taken); and temporary variables (a.k.a. registers). The similarity with the abstract memory is intentional.

```
Inductive ccell : Type :=
| CCmem (b: block) (κ: memory_chunk) (ofs: Z)
| CClocal (s: ident).
```

To be able to correctly define the semantics of the CFG expressions in this model, we need to keep track of a little bit more information. Indeed, pointer comparison in

CompCert behaves differently whether its arguments are *valid* pointers or not. The validity of pointers is defined in term of permissions attached to each memory address.

Therefore, the memory is defined as the following record type.

```
Record fmem : Type := {
  f_of_fmem:> ccell → val;
  perm: block → Z → option permission
}.
```

The first field, namely f_of_fmem, is a coercion (as denoted by :>); this means that a value of type fmem can be used as a function of type cell → val. Therefore, reading the content of a cell c in a memory f amounts to the function application written f(c).

Writing to the memory is a little bit more intricate. Indeed, memory cells can *overlap*. We introduce a notion of disjointness (i.e., non-overlap) with the following predicates. Overlapping cells are either the same temporary or overlapping chunks of the same memory block.

```
Definition disjoint_ranges (r r': Z * memory_chunk) : Prop :=
  let (i, κ) := r in let (i', κ') := r' in
    i' + size_chunk κ' <= i ∨ i + size_chunk κ <= i'.

Definition disjoint_cells (c c': ccell) : Prop :=
  match c, c' with
  | CCmem b κ ofs, CCmem b' κ' ofs' => b' ≠ b ∨ disjoint_ranges (ofs, κ) (ofs', κ')
  | CClocal i, CClocal i' => i ≠ i'
  | _, _ => True end.
```

Then a store can be specified by the following relation between memories. It says that the memory resulting from the store, f', holds the new value v at the target cell c, and agrees with the original memory f for all cells disjoint from c[3].

```
Definition fmem_update (c: ccell) (v: val) (f f': fmem) : Prop :=
  f' c = v ∧ (∀ c', c' ∈ disjoint_cells c → f' c' = f c').
```

The abstract transformers of the memory domain can then be specified against this concrete view of the memory, as shown on Figure 5.8. The forget_sound property reads as follows. The concretization of the result of (forget x ab) contains (at least) all concrete memories obtained after updating an initial memory f (in the concretization of ab) at the target cell with *any* value v.

The specification for the assign transformer is very similar, except that the value v is taken among the possible results of the evaluation of the expression e.

Again, the specification of the store transformer is similar, except that the updated cell (CCmem b κ (Int.unsigned i)) is built from any result (Vptr b i) of the evaluation of the address expression ℓ, and that the stored value v is transformed according to the chunk κ specifying the memory access (as defined by CompCert's Val.load_result function). In

[3]This does not specify the value of overlapping cells, and is therefore not sufficient for a precise analysis of programs performing type-punning; see § 5.5.2 for a discussion.

```
Class fmem_dom (ge: Genv.t) (t: Type) (D:pre_mem_dom t) (G:gamma_op t fmem) : Prop := {
 range_sound: ∀ (ab: t) (f: fmem) (x: ident),
 f ∈ γ ab ⟶
 match f (CClocal x) with
 | Vint i | Vptr _ i => i ∈ ints_in_range (range ab x)
 | _ => True end;
 forget_sound : ∀ (x: ident) (ab: t) (f: fmem) (v: val),
  f ∈ γ ab ⟶
  fmem_update (CClocal x) v f ⊆ γ(forget x ab);
 assign_sound : ∀ (x: ident) (e: expr) (ab: t) (f: fmem) (v: val),
  f ∈ γ(ab) ⟶
  v ∈ eval_expr ge f e ⟶
  fmem_update (CClocal x) v f ⊆ γ(assign x e ab);
 store_sound : ∀ (κ: memory_chunk) (ℓ r: expr) (ab: t) (f: fmem) (b: block) (i: int) (v: val),
  let c := CCmem b κ (Int.unsigned i) in
  f ∈ γ(ab) ⟶
  Vptr b i ∈ eval_expr ge f ℓ ⟶
  v ∈ eval_expr ge f r ⟶
  c ∈ writable_cell f ⟶
  fmem_update c (Val.load_result κ v) f ⊆ γ(store κ ℓ r ab);
 assume_sound : ∀ (e: expr) (ab: t) (f: fmem),
  f ∈ γ(ab) ⟶
  true ∈ Union (eval_expr ge f e) Val.bool_of_val ⟶
  f ∈ γ(assume e ab) }.
```

Figure 5.8: Abstract memory specification

addition, this transformer may use the fact that the destination cell is *writable*, i.e., that the offset i is correctly aligned and that there are sufficient permissions to write there.

The whole specification is parameterized by a global environment ge, corresponding to the program being analyzed.

5.4.2 Points-to Domain

The soundness of the points-to domain is established against a concretization relation defined as follows. The Int abstract value represents all machine integers. The abstract values of the form Ptr(bs) represent all pointers whose block is in the set bs (the offset is not constrained). The trivial abstract type All is related to any value. In particular, it is the only abstract value that represents the bogus value Vundef.

This invariant enables to prove that some value cannot be Vundef. This is particularly useful for proving progress (see next chapter § 6.3) and to precisely analyze programs with type-punning (see § 5.5.2).

5.4.3 Concretization Relation

The relational domain comes with a concretization to functions from (abstract) cells to numerical values (machine integers). Using the ncompat relation (see below) it can be given a concretization to functions from cells to values (including pointers). The type domain, that maps abstract cells to an abstraction of their types, also concretizes to a function from abstract cells to values. We can then define the following concretization relation for the abstract memory domain; here concrete values are functions from abstract cells to values.

```
Definition ncompat (v: val) (j: int) : Prop :=
  match v with
  | Vint i | Vptr _ i => i = j
  | Vfloat _ | Vundef => True
  end.
Instance pre_gamma : gamma_op t (acell → val) :=
  λ ab f,
  let (pt, nm) := ab in
  f ∈ γ(pt) ∧ ∃ (v: acell → int), v ∈ γ(nm) ∧ ∀ c, ncompat (f c) (v c).
```

The set pre_gamma(pt, nm) is the intersection of the concretization γ(pt) of the points-to map and of the set of all memories related (by the point-wise lifting of the ncompat relation) to some function v in the concretization γ(nm) of the numerical abstraction.

In order to relate an abstract memory to a concrete one, we still have to relate abstract cells to concrete cells. This is done through an *allocation* partial function, that maps abstract cells to concrete cells. One such allocation function is given below and called δ_0. Since the set of abstract cells is a bit limited, this function is not so far from the identity function (modulo the correspondence between blocks identifiers and variable names). It will be refined when the type acell will be extended (such as with local variables).

```
Definition allocation : Type := acell → option ccell.

Definition δ₀ : allocation :=
  λ ac,
  match ac with
  | ACglobal g κ o => do_opt b <- Genv.find_symbol ge g; Some(CCmem b κ o)
  | ACreg i => Some(CClocal i)
  end.
```

Given such an allocation function, we can relate memories as follows: related cells are mapped to equal values (and cells that are related to nothing have unconstrained value).

```
Definition mem_rel (δ: allocation) (m: fmem) (ρ: acell → val) : Prop :=
  ∀ c a, δ(a) = Some c → m(c) = ρ(a).
```

Notice that, given an allocation function δ and a memory m, the set mem_rel δ m is not empty: there is in it at least the function $m \circ \delta'$, where δ' allocates every unallocated cell to a dummy one (e.g., δ′ a := match δ a with Some c => c | None => CClocal 1 end).

Then we can define the concretization relation for the memory abstract domain to concrete memories. A memory m is in the concretization of an abstract memory ab whenever all functions in the pre_gamma concretization of ab are related to m.

```
Instance gamma : gamma_op t fmem := λ ab m, mem_rel δ₀ m ⊆ pre_gamma(ab).
```

By using a *for-all* quantification (i.e., set inclusion) when a *there-exists* (i.e., non-emptiness of the intersection) would be plausible as well, we insist on the fact that we do not care about the value of the non-allocated cells, and that the memory domain should not compute anything about these cells.

5.4.4 Key Lemmas

So as to depict how the soundness proof of the memory domain is carried on, this section highlights the most important lemmas; their proofs are done in Coq and not described here.

Conversion

There are two phases during conversion: load elimination; and pointer elimination. The first phase produces a set of expressions such that, collectively, the resulting expressions may evaluate to all possible values of the original expression.

```
Lemma convert_correct (m: fmem) (ρ: acell → val) (ab: t) (e: expr) pes :
  mem_rel δ₀ m ρ →
  m ∈ γ(ab) →
  convert e ab = Just pes →
  eval_expr m e ⊆ Union pes (eval_pexpr ρ).
```

The second phase produces, for each pointer-expression, a purely numerical expression. Numerical expressions evaluate to numerical values (machine integer) whereas pointer-expressions may also evaluate to pointers. Therefore, the soundness property of this second phase states that for each possible value v of the original expression, the resulting expression evaluates to some number n that is compatible with value v. The compatibility relation has been defined in Section 5.2.3. It relates in particular numbers to themselves and pointers to their offsets.

```
Lemma nconvert_correct pe :
  ∀ v, v ∈ eval_pexpr pe →
  ∃ n, n ∈ (eval_nexpr (nconvert pe) ∩ ncompat v).
```

Assign

The soundness lemma for the (assign_cells dst e ab) operation reads as follows. For every concrete memory f and value v the expression e may evaluate to in f, for every destination cell a (related to the concrete cell c through δ_0), the concrete memories obtained by updating f at c with value (Val.load_result κ v) are in the concretization of the resulting abstract state. The stored value is trans-coded according to chunk κ.

```
union {
  int i;
  unsigned char b[4];
} u;
u.i = 0x12345678;
switch (u.b[0]) {
case 0x12:
  return BIG_ENDIAN;
case 0x78:
  return LITTLE_ENDIAN;
}
```

```
void*
memcpy(void *dest, const void *src, size_t n)
{
  uint8_t *d = dest;
  const uint8_t *s = src;
  int i;
  for ( i = 0 ; i < n ; ++i ) {
    d[i] = s[i];
  }
  return dest;
}
```

Figure 5.9: Type punning examples

```
Lemma assign_cells_sound (κ: option memory_chunk) (dst: set acell) (e: expr) (ab: t) :
  ∀ m, m ∈ γ ab ⟶
  ∀ v, v ∈ eval_expr m e ⟶
  ∀ a, a ∈ dst ⟶
  ∀ c, δ₀ a = Some c ⟶
  fmem_update c (Val.load_result κ v) m ⊆ γ(assign_cells κ dst e ab).
```

5.5 Extensions

The memory domain presented so far has various limitations; in this section we sketch how to remove some of them.

5.5.1 Local Variables

In the CFG language, (non-temporary) variables that are local to a function are allocated to one single memory block, corresponding to the stack frame of this function. Since we consider programs without function calls, the stack pointer (i.e., the pointer to the beginning of the stack frame) is constant and can be statically predicted.

The type of abstract blocks is therefore enriched with a name for the stack block. Similarly, the type of abstract cells is enriched with a constructor for stack-allocated local variables; it is parameterized, as for global variables, by a memory chunk and an offset. The allocation function is also defined for these cells.

The points-to domain can then be refined to precisely abstract constant stack pointers.

Modeling of local variables would be more involved in the presence of function calls. We defer the description of the analysis of programs with functions until next chapter (§ 6.2).

5.5.2 Realization & Type Punning

Some low-level programs perform so-called *type-punning*. This means accessing some data of a given type (say a 32-bits integer) as if it were of a different type (say an array of four 8-bit integers). This is usually used to access the bit-level representation of some data.

For instance, the program on the left of Figure 5.9 is a C snippet that discovers at run-time the endianness of the architecture. Such *puns* are only loosely specified by the C standard. Most of them are however precisely defined in the CompCert semantics.

There are several kinds of *puns*. They all involve reading some data with a different chunk than the one that had been used to write it. We will focus on the following two cases.

- Reading at the same address with a chunk of the same size. This covers changes in signedness and manipulations of the bit-level representation of floats[4]. The first are similar to casts (with the Ocast8unsigned operator, for instance) whereas the last are no standard C operations and are not handled by the numerical abstract domains.
- Reading one byte of a multi-byte data, as in the implementation of the memcpy function given on the right of Figure 5.9.

In case of type-punning, a cell whose content has to be read is not bound in the abstract domain. Indeed, when a cell is written, the (abstract) contents of all overlapping cells are forgotten. To precisely approximate the read value, the cell needs to be *realized* [Min06], that is, bound to some non-trivial abstract value derived from the values of overlapping cells.

We expect the following property from the realize c ab function, meant to realize the cell c in the abstract state ab: all memories in the concretization of some abstract value ab before realization, are in the concretization after realization.

```
Lemma realize_sound (c: acell) (ab: t) : γ(ab) ⊆ γ(realize c ab).
```

However, this is hardly provable, since in the functional view of the concrete memory, the values of overlapping cells are not constrained. One way to address this issue is to further constrain the fmem type. In a first step, we can add the following property:

```
Definition pun_similar (f: cell ⟶ val) := ∀ b κ κ′ ofs,
 size_chunk κ = size_chunk κ′ ⟶
 align_chunk κ = align_chunk κ′ ⟶
 ∃ bytes,
 f (CCmem b κ ofs) = decode_val κ bytes ∧
 f (CCmem b κ′ ofs) = decode_val κ′ bytes.
```

This is enough to handle some kinds of puns: when some data is written with some chunk and read back with a different one, but at the exact same offset with similar alignment constraints. This covers reinterpretation of the signedness of a small int and manipulating the bit-level representation of floats.

An other helpful property, which covers the endianness check and memcpy function of Figure 5.9 is as follows.

```
Definition pun_u8 (f: cell ⟶ val) := ∀ b κ ofs,
 (align_chunk κ | ofs) ⟶
 ∃ bytes,
```

[4]Fast computations of approximations of inverse square roots are a notable example of floating-point operations performed on the bit-level representation of floats.

```
int x;

int main(void)
{
  int * p = 0;
  x = 0;
  if ( any_bool() ) p = & x;
  if ( p != 0 ) *p = 1;
  int z = x;
  return z;
}
```

```
int x[2];

int main(void)
{
  int * p = 0;
  x[0] = 0; x[1] = 1;
  if ( any_bool() ) p = & x[1];
  if ( p != 0 ) *p = 1;
  int z = x[0];
  return z;
}
```

Figure 5.10: C programs checking for null pointers

```
f (CCmem b κ ofs) = decode_val κ bytes ∧
length bytes = size_chunk κ ∧
∀ i (LT: i < length bytes),
  f (CCmem b Mint8unsigned (ofs + i)) = decode_val Mint8unsigned (get i bytes LT :: nil).
```

It states that for every value read through some chunk κ at a properly aligned address, there is a sequence of bytes from which this value is decoded, such that reading in memory at the same address the i^{th} Mint8unsigned chunk yields the decoding of the i^{th} byte in the sequence[5]. Notice that this would not hold without the alignment requirement: mis-aligned reads of multi-byte chunks always yield Vundef, even though the byte-wise readings return well-defined values.

These properties of the memory are added as invariants of the type fmem. This enables to prove the soundness of realization functions that bind new cells in the memory domain (i.e., in both points-to and numerical domains) from information about overlapping cells. Such functions are called optimistically when expressions dereference unbound cells, so as to try to bind them to some non-trivial value.

Realization needs to take place during conversion. Here is an example: there is a pun on variable x, that is written as a 32-bit integer, and read as an 8-bit unsigned integer.

```
int t[2] = { 1, 2 };
union { int i; uint8_t[4] c; } x;
x.i = 1;
int v = t[x.c[0]];
```

To understand what cells are targeted by the array access, we need to numerically evaluate the subscript expression. This requires the 8-bit cell to be realized during conversion. The conversion function is therefore adapted to manipulate the abstract state as in the state monad, rather than as an input value.

5.5.3 The Case of Null Pointers

[5] The get function takes as argument a proof LT that there are at least i elements in the list, to be sure to be able to return a meaningful value.

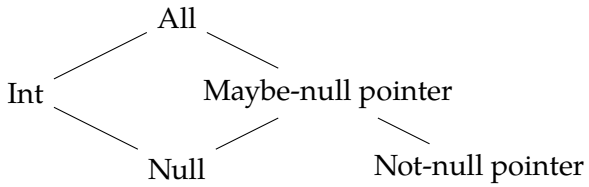


Figure 5.11: Lattice of abstract types to handle null pointers

The analyzer described so far is not able to prove that, after a test ensuring that a pointer is not null, said pointer is actually not null. Consider for instance the program on the left of Figure 5.10. There, pointer p is either null or points to variable x, depending on unknown input. The assignment on line 8 is guarded so that it is safe and it updates variable x.

The problem comes from the fact that zero and null pointers are the same value (in CompCert as in standard C99 [C99, § 6.3.2.3]). Indeed, in the above example, variable p may have two distinct types: integer or pointer. In other circumstances, the memory domain soundly ignores variables whose type cannot be inferred: a safe program is not expected to store values of different types in a same variable. The null pointer is an exception to this typing rule: a pointer can be used as an integer without explicit cast, provided it is null (and symmetrically).

We therefore refine two components of the memory domain, as we describe below.

Richer types Currently, the points-to domain loses precision when joining the abstraction for a null value (an integer) and a non-null pointer. This domain is therefore refined to still precisely handle maybe-null pointers (see Figure 5.11). Value with pointer-type are also associated to a points-to set, as before (not shown on the picture).

Assume When analyzing a guard as (p != 0), if p is known to be a maybe-null pointer, its abstract type has to be refined to the not-null pointer with the same points-to set.

More generally, we introduce a backward evaluation function in the points-to domain, similar to the one presented in the cases of numerical environments in Section 3.3.2. This backward evaluation function corresponds to the implementation of the pt_assume introduced in Section 5.3.5.

This enhancement enables the analysis of programs that manipulate pointers that may be null. However, precision could still be improved, as illustrated by the program on the right of Figure 5.10. This program is very similar to its neighbor, but variable x is now an array of two integers and p, when not null, targets the second cell (i.e., its offset is four). Thus, when the analysis reaches line 8, the points-to domain knows that p, when not null, points to the block of x, and the numerical domain knows that the offset of p is zero or four. This comes from the join, in both domains, of the information about the initial null value of p and the information about the non-null pointer to the second cell of x. Thus the analyzer considers that the store on line 8 may target both cells of x, resulting in a precision loss. This situation could be improved by a better communication between the numerical and points-to domains, left as future work.

5.6 Conclusion

In this chapter, we have described an abstract domain that represents the state of the global memory and the local environment of a CFG program. This domain is vastly inspired by Antoine Miné's [Min06]. This state is abstracted as a collection of cells. This domain is fully parameterized by a numerical domain, and, if that domain is relational, the memory domain leverages this feature and computes relational invariants about the contents of the cells. For instance, it is able to prove that an array resulting from a bubble sort is indeed sorted (provided that the size of the array is known, that the loops are unrolled, and that the numerical domain, e.g., the domain of polyhedra, can represent inequalities between variables).

A more precise analysis of the points-to relations, known as the *shape* of the memory, could be achieved by dedicated shape analyses [CR13]. Such analyses are beyond the scope of this memory domain, as they usually focus on dynamically allocated linked data structures, whereas we target embedded programs without dynamic memory allocation.

The cells of the memory domain may overlap, and care is taken so that overlapping cells are associated with consistent information. Cells can be opportunistically realized when needed, in particular in case of type punning.

This domain is fully verified in Coq against the CompCert semantics of the CFG language. The proof introduces a functional view of the (concrete) memory that is much more convenient to relate to its abstract counterpart.

Future improvements of this memory domain include the ability to hold so called *summary* cells, i.e., cells in the abstract domain that represent several cells at once in the concrete state. This is required to efficiently represent large arrays, and model dynamic memory allocation and recursion. This has been overlooked as we target safety-critical, embedded, programs in which the use of such features hardly occurs. Such an enhancement would require to generalize the allocation function and adapt the conversion function, for instance by replacing weak cells by a conservative interval.

The next chapter discusses how the various methods presented in this chapter and the previous ones are applied and refined to construct a verified static analyzer for safety-critical C programs.

Chapter 6

Practical static analyzer for safety-critical C

In the previous chapters, we have seen how to implement executable verified value analyzers. Still, so as to analyze actual programs and have stronger guarantees about the analysis results, the following issues need to be addressed.

Source vs. intermediate representation The analyzers of chapters 3 and 5 operate on the CFG intermediate representation of CompCert. When they are used to analyze C programs, the analysis takes place after a fair amount of program transformations: most notably, local variables have been allocated to stack slots or registers, and the control flow is expressed in CFG as a graph rather than as a sequence of nested statements. In addition, the compiler may remove behaviors and, if the source program is unsafe, even introduce new ones. Therefore, it is unlikely that the results that hold for the CFG programs directly apply to the source C program.

Programs with functions The case of function calls has been put aside in the previous chapters: an easy work-around was to inline all function calls. However, this is not always possible (e.g., when they are recursive calls or function pointers) nor desirable (e.g., because of the increase in code size).

Result validity The analyzers assume that the analyzed programs are free of undefined behaviors, but there is no guarantee that this assumptions hold. However, a value analysis could be used to prove such facts (no division by zero, array accesses are in-bounds etc.). To tie the knot, we will implement in a value analyzer the required checks so as to prove, at once, that 1. the analyzed program is free of undefined behaviors; and 2. the analysis results hold for all executions of the program.

Rudimentary numerical abstract domains All analyzers are parameterized by an underlying numerical domain. In practice, only intervals, or at best intervals with congruence information, have been implemented. It would be preferable to be able to reuse existing abstract domain libraries without completely re-implementing them in Coq.

Partial programs Actual programs to be analyzed do call library functions and perform I/O. In chapter 3, the issues was eluded since the memory was not analyzed, and library functions cannot tamper with the contents of local registers. In chapter 4, input was allowed before the program is loaded, and no more. A practical analyzer should provide means to precisely handle programs that perform I/O at any time and call external functions whose code is not available for analysis.

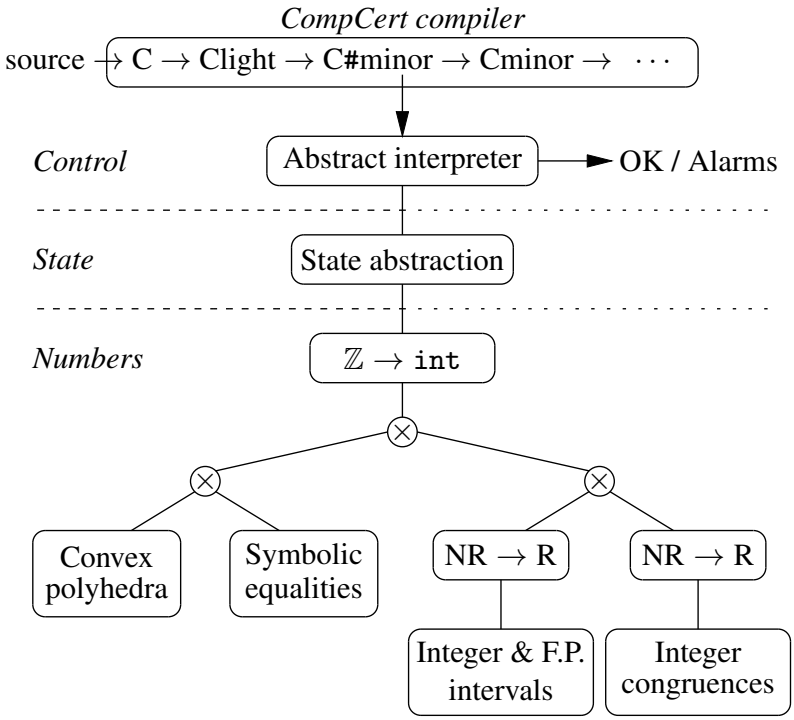


Figure 6.1: Global architecture of the Verasco analyzer

In this chapter we present an analyzer for the C♯minor intermediate representation of CompCert, named Verasco. It is structured as sketched on Figure 6.1: the interpreter reuses CompCert front-end from source code to C♯minor, it is parameterized by a state abstraction that has been described in chapter 5, and it uses a rich composition of numerical domains; non-relational domains are adapted to the relational interface by a generic functor ("NR → R" on the figure); numerical domains are designed to operate on ideal arithmetic and adapted to machine arithmetic by a functor ("ℤ → int") described in Section 6.4.1. This analyzer aims at proving that analyzed programs are free of undefined behavior, i.e., that their execution cannot be stuck. These undefined behaviors include the ones produced by failing assertions. Standing on CompCert, this analyzer targets the analysis of safety-critical programs and therefore requires that said programs do not use features as recursion or dynamic memory allocation.

This chapter does not describe how floating point arithmetic is handled. It must be noted that the Verasco analyzer features a non-relational abstract domain of float intervals that is able to infer invariants about the range of floating point values.

Part of the contents of this chapter has been published at the Symposium on Principles of Programming Languages [Jou+15].

6.1 Abstract Interpretation of Structured Programs with Gotos

All analyzers presented in previous chapters operate on a graph-based representation of programs. The C♯minor intermediate representation, that has been presented in Section 2.2, relies on a structured abstract syntax that prevents a direct application of the

$[\![\text{x := e}]\!]^\sharp(A_i, A_\ell)$ = (assign x e A_i, $\bot$, $\bot$, $\bot$)
$[\![\text{b; c}]\!]^\sharp(A_i, A_\ell)$ = $(C_o, B_r \sqcup C_r, B_e \sqcup C_e, B_g \sqcup C_g)$
where $(B_o, B_r, B_e, B_g) = [\![\text{b}]\!]^\sharp(A_i, A_\ell)$ and $(C_o, C_r, C_e, C_g) = [\![\text{c}]\!]^\sharp(B_o, A_\ell)$
$[\![\text{if e then b else c}]\!]^\sharp(A_i, A_\ell) = [\![\text{b}]\!]^\sharp(\text{assume e true } A_i, A_\ell) \sqcup [\![\text{c}]\!]^\sharp(\text{assume e false } A_i, A_\ell)$
$[\![\text{loop s}]\!]^\sharp(A_i, A_\ell)$ = $(\bot, A_r, A_e, A_g)$
where $(A_o, A_r, A_e, A_g) = \text{pfp } (\lambda\ (X_o, X_r, X_e, X_g), [\![\text{s}]\!]^\sharp(A_i \sqcup X_o, A_\ell))\ \bot\ N_{\text{widen}}$
$[\![\text{exit n}]\!]^\sharp(A_i, A_\ell)$ = $(\bot, \bot, (\lambda\ \text{m, m} = \text{n ? } A_i : \bot), \bot)$
$[\![\text{block s}]\!]^\sharp(A_i, A_\ell)$ = $(A_o \sqcup A_e(0), A_r, (\lambda\ \text{n}, A_e(\text{n+1})), A_g)$ where $(A_o, A_r, A_e, A_g) = [\![\text{b}]\!]^\sharp(A_i, A_\ell)$
$[\![\text{goto L}]\!]^\sharp(A_i, A_\ell)$ = $(\bot, \bot, \bot, (\lambda\ \text{X, X} = \text{L ? } A_i : \bot))$
$[\![\text{L: s}]\!]^\sharp(A_i, A_\ell)$ = $[\![\text{s}]\!]^\sharp(A_i \sqcup A_\ell(\text{L}), A_\ell)$

Figure 6.2: Representative cases of the C♯minor abstract interpreter

iteration techniques discussed in previous chapters (§ 3.2, § 4.3.3).

It would be possible to make explicit the graph structure of C♯minor programs by introducing program points. However, this task is not trivial, and the analyzer can instead take advantage of the structured form of programs (only goto statements are unstructured). For instance, structural abstract interpreters use less memory than graph-based ones, maintaining only a few different abstract states at any time, instead of one per program point [Cou+09]. The transfer function, noted $[\![\cdot]\!]^\sharp$, is defined over basic as well as compound statements: given the abstract state A "before" the execution of statement s, it returns $[\![\text{s}]\!]^\sharp(\text{A})$, the abstract state "after" the execution of s. For sequences, we have $[\![\text{s}_1\text{; s}_2]\!]^\sharp = [\![\text{s}_2]\!]^\sharp \circ [\![\text{s}_1]\!]^\sharp$. For loops, the transfer function takes a local fixpoint of the transfer function for the loop body.

However, the transfer function for our abstract interpreter is more involved than usual, because control can enter and leave a C♯minor statement in several ways. A statement s can be entered normally at the beginning, or via a goto that branches to one of the labels defined in s. Likewise, s can terminate either normally by running to the end, or prematurely by executing a return, exit, or goto statement. Consequently, the transfer function is of the form $[\![\text{s}]\!]^\sharp(A_i, A_\ell) = (A_o, A_r, A_e, A_g)$ where A_i (input) is the abstract state at the beginning of s, A_o (output) is the abstract state after s terminates normally, A_r (return) is the state if it returns, and A_e (exits) maps exit numbers to the corresponding abstract states. The goto statements are handled by two maps from labels to abstract states: A_ℓ (labels) and A_g (gotos), the first representing the states that can flow to a label defined in s, the second representing the states at goto statements executed by s. Figure 6.2 excerpts from the definition of the transfer function and shows all these components in action.

The loop case computes a post-fixpoint with widening and narrowing, starting at $\bot$ and iterating at most N_{widen} times. The pfp iterator is, classically, defined as

$$\text{pfp } F\ A\ N = \begin{cases} \top & \text{if } N = 0 \\ \text{narrow } F\ A\ N_{\text{narrow}} & \text{if } A \sqsupseteq F\ A \\ \text{pfp } F\ (A \nabla F\ A)\ (N-1) & \text{otherwise} \end{cases}$$

with the narrowing phase defined as

$$\mathsf{narrow}\ F\ A\ N = \begin{cases} A & \text{if } N = 0 \\ \mathsf{narrow}\ F\ (F\ A)\ (N-1) & \text{if } A \sqsupseteq F\ A \\ A & \text{otherwise} \end{cases}$$

Each iteration of pfp uses the widening operator ∇ provided by the abstract domain to speed up convergence. Once a post-fixpoint is found, F is iterated up to N_{narrow} times in the hope of finding a smaller post-fixpoint. In both widening and narrowing iterations, we use "fuel" N to convince Coq that the recursions above are terminating, and to limit analysis time.

Optionally, the abstract interpreter can unroll loops on the fly: the N first iterations of the loop are analyzed independently in sequence; the remaining iterations are analyzed with a pfp fixpoint. This delays widening and gains precision. The unrolling factor N is currently given by an annotation in the source code.

In addition to the analysis of loop statements, a post-fixpoint is computed for every C♯minor function to analyze goto statements. This function-global iteration ensures that the abstract states at goto statements are consistent with those assumed at the corresponding labeled statements. In other words, if s is the body of a function and $[\![s]\!]^\sharp(A_i, A_\ell) = (A_o, A_r, A_e, A_g)$ is its analysis, the analysis iterates until $A_g(L) \sqsubseteq A_\ell(L)$ for every label L. When this condition holds, the abstraction of the function maps entry state A_i to exit state $A_o \sqcup A_r$ corresponding to the two ways a C♯minor function can return (explicitly or by reaching the end of the function body).

Concerning functions, the abstract interpreter reanalyzes the body of a function at every call site, effectively unrolling the function definition on demand. We use fuel again to limit the depth of function unrolling. More details on inter-procedural analysis are given in next Section.

The soundness proof for the abstract interpreter is massive, owing to the complexity of the C♯minor language. To keep the proof manageable, we break it in two parts: 1. the definition and soundness proof of a suitable Hoare logic for C♯minor; and 2. a proof that the abstract interpreter infers Hoare "triples" that are valid in this logic.

C♯minor statements can terminate in multiple ways: normally, or prematurely on an exit, return or goto statement. They can also be entered in two ways: at the beginning of the statement, or via a goto to a label defined within. Consequently, our program logic for C♯minor manipulates Hoare "heptuples" of the form $\{ P, P_\ell \}\ s\ \{ Q, Q_r, Q_e, Q_g \}$ where P is the precondition if s is entered normally, $P_\ell(L)$ the precondition if s is entered by a goto L, Q the postcondition if s terminates normally, $Q_r(v)$ the postcondition if s terminates by a return of value v, $Q_e(i)$ the postcondition if s terminates by exit(i), and $Q_g(L)$ the postcondition if s terminates by goto L. The rules of this logic are similar to those of the program logics for Cminor and Clight by Appel and Blazy [AB07; App14] (without the separation logic aspects).

6.2 Inter-Procedural Analysis

Inlining all function calls before analyzing a program is a convenient work-around to make up for the lack of inter-procedural analysis in an analyzer. Unfortunately, this is not

```
Class mem_dom (t: Type) : Type := { (* ... *)
 deref_fun: (* dereference a function pointer *)
 expr ⟶ t ⟶ list (ident * fundef);

 push_frame: (* push a new stack frame *)
  (* function to call *) ident ⟶ function ⟶
  (* parameter values *) list expr ⟶
  t ⟶ t+⊥;

 pop_frame: (* pop a stack frame *)
  (* return value *) option (expr) ⟶
  (* temporary to write to *) option(ident) ⟶
  t ⟶ t+⊥
}.

Definition concrete_state : Type := list (ident * (temp_env * env)) * mem.
```

Figure 6.3: Operators of the memory domain for inter-procedural analysis

always possible, in particular when functions are called through (non-trivial) pointers or recursively; it may dramatically increase the code size (when a large function is called many times); and it prevents clever inter-procedural analyses (approximations for faster/smaller results).

Here we present an extension of our analyzer to inter-procedural analysis that is very similar to inlining (without any increase in the code size) but supports function pointers: each function is analyzed in every calling context, by a recursive call of the analyzer. We do not attempt to summarize or merge different calling contexts.

Extending the iterator The memory domain signature (presented on Figure 5.1) is completed (see Figure 6.3) with two operators that model function calls and returns, and an operator (deref_fun) that, given an expression and an abstract state, returns a super-set of all functions that may be designated by that expression in this state.

The concrete state against which the memory domain is specified is now a pair made of a call-stack and a concrete memory. The call-stack holds the name of the called functions and, for each called function, an allocation map of the local variables (mapping variable names to memory blocks) and an environment (mapping each temporary variable to its value).

To analyze a function call such as x := f(args), where f is an expression, the iterator first resolves the function by calling deref_fun. Then the abstract state at function entry is prepared by calling push_frame. Finally the function body is (recursively) analyzed from this state. There, the analysis of return statements will compute the abstract state after the call and assignment to x thanks to the function pop_state.

As an illustration, consider the following program. The external any_int64 function is interpreted as returning a non-deterministic value.

```
1  # include <assert.h>
2
3  extern long long any_int64(void);
4
5  int z = 0;
6
7  int f(int n) { return 1 / n; }
8
```

```
9  int main(void) {
10   while ( any_int64() ) {
11     z += f(-1);
12     z += f(+1);
13   }
14   assert ( z == 0 );
15   return 0;
16 }
```

The body of the function f will be analyzed twice each time the loop body is analyzed: once with the local variable n bound to (an abstraction of) -1, once bound to +1. The two possible abstract states at the beginning of f are never merged: this simple approach to inter-procedural analysis is also the most precise. Merging the numerical abstractions of variable n using a convex domain (e.g., intervals or polyhedra) could indeed lead to a false alarm about a division by zero in f.

Extending the memory domain The main issue is how to handle stack-allocated local variables. In C♯minor, each stack variable is allocated in its own block, and the particular name of this block cannot be (in general) predicted at analysis time. Since we only consider programs without recursive calls, stack variables can be unambiguously represented by a pair made of a function name and the variable name.

To implement the interface for inter-procedural analysis in the memory domain, the abstract domain is extended with a stack that keeps track of the names of the called functions. In addition, to be able to distinguish local from global variables in expressions, the set of variables local to each function is remembered in this abstract stack.

Function resolution uses points-to information to compute a set of functions that expression f may point to: forward evaluation of expression f in the points-to domain yields a set of blocks which can be mapped to a set of functions by looking at the program.

The push_frame operation of the state abstract domain performs the assignments corresponding to argument passing: arguments are evaluated in the context of the caller and then assigned to local variables of the callee.

Symmetrically, the pop_frame operation is used when analyzing return *e* statements. The expression *e* is analyzed in the callee context and assigned to a temporary in the caller context. Then, pop_frame simulates the freeing of local variables on function exit: this consists in invalidating the information associated to them, as well as invalidating pointers that may point to them. This is a costly operation: the whole points-to state has to be scanned for pointers to the top stack-frame. The following program illustrates the need for this freeing.

```
1  int *
2  f(int b, int *pz)
3  {
4    int i = 0;
5    if (b)
6      return &i + *pz;
7    return &i;
8  }
```

```
9  int
10 main(void)
11 {
12   int *p = f(0, 0);
13   int *q = f(1, p);
14   return 0;
15 }
```

This program calls a function f twice. The first time, this function returns a pointer

```
Definition block_mon (A:Type) : Type := (A * list string).

Inductive block_t (B: Type) : Type :=
| Blocked
| NotBlocked (b: B).

Instance block_mon_gamma A B (G:gamma_op A B): gamma_op (block_mon A) (block_t B) :=
 λ t: block_mon A,
 let (a, alarms) := t in
 match alarms with
 | nil => λ b, match b with NotBlocked b' => b' ∈ γ(a) | Blocked => False end
 | _ => λ _, True
 end.
```

Figure 6.4: The non-block monad

to its local variable i. The second time, it gets this pointer back in its argument pz and dereferences it, which is an error, even though it is a pointer to the local variable of this function! Therefore, pointers to stack-allocated variables have to be invalidated on function returns, to prevent any further use, even in contexts that may look safe.

6.3 Progress Verification

At the same time we transform abstract states, we perform verifications to prove that every C♯minor statement or expression evaluates safely (without blocking) in its evaluation context.

These checks are scattered all over the analyzer: in the numerical domains (e.g., no division by zero), in the memory domain (e.g., load and store are performed within bounds and with the correct alignment) and in the iterator (e.g., function must be called with the right number of arguments). So the interfaces of the analyzer are recast in term of a so-called non-block monad that enables the proof of progress.

6.3.1 The Non-Block Monad

Every operator of the memory domain now wraps its result in a logging monad: alongside the result, a list of alarms (unspecified strings) is returned. The block_mon type constructor is defined in Figure 6.4. For instance, the assign operator is now given the following type, in which the result type is wrapped in block_mon:

```
assign: ident ⟶ expr ⟶ t ⟶ block_mon (t+⊥).
```

The soundness of the result is only guaranteed when there are no alarms: a sound implementation is allowed to return any result provided there is at least an alarm. This enables the analyzer to do unsound assumptions when an alarm is raised. For instance, when the target of a store cannot be precisely determined, an alarm is raised (unsafe store) but the analysis continues (e.g., ignoring the possible effects of said store).

The specification of this block_mon data-type, i.e., its concretization, is given in Figure 6.4. It combines two aspects of this data-type. First, the list of alarms needs to be empty for the value to actually represent a non-trivial concrete set. Second, if the list of alarm is empty, then evaluation is not stuck.

This last fact is not directly stated in the definition of the concretization relation, since there we do not know what computation it is about. However, concrete values are wrapped in the NotBlocked constructor so that the Blocked value can appear in the concretization only when the list of alarms is not empty. We use this property in the specifications of the abstract transformers.

Consider for instance the case of (assign x q ab), that models the assignment of the result of evaluating expression q to variable x in any concrete state represented by ab. Recall the definition given in Section 3.4, on page 40, of the strongest post-condition of an assignment. The following definition refines that one to additionally express that the expression is not stuck when there are no alarms. This specification relies on the fact that eval_expr is deterministic: there cannot be two (different) values v and v' in the set eval_expr ge e t m q.

```
Definition Assign (x: ident) (q: expr) (E: ℘ concrete_state) : ℘ (block_t concrete_state) :=
λ cs: block_t concrete_state,
∃ f t e s m,
  ((f, (t, e)) :: s, m) ∈ E
∧ ∀ v, v ∈ eval_expr ge e t m q →
  cs = NotBlocked ((f, (PTree.set x v t, e)) :: s, m).

Definition assign_sound := ∀ x q ab, Assign x q (γ ab) ⊆ γ (assign x q ab).
```

The idea is that the set (Assign x q (γ ab)) contains Blocked only if there is a concrete state in (γ ab) from which the evaluation of expression q is stuck. Indeed, in such a case, the hypothesis (v ∈ eval_expr ... q) is a contradiction from which (Blocked = NotBlocked (...)) can be proved.

If the list of alarms returned by assign x q ab is empty, then Blocked is not in its concretization, and the assign_sound property states that the evaluation of expression q cannot be stuck.

6.3.2 Progress Verification in the Memory Domain

Proving that the execution cannot be stuck, as opposed to taking that fact as an hypothesis, requires to prove various additional properties of the concrete execution states. Some of these properties hold for all executions of all programs: they follow from the semantics of the programming language. However, they are not easily available: states do not come with any side well-formedness condition. We thus prove them for each analyzed program, i.e., check them at analysis time. To this purpose, we maintain as an invariant of the abstract memory domain that all concrete states (in the concretization of the current abstract memory) satisfy said properties (an excerpt of this invariant is shown on Figure 6.5).

An example of such property is about the allocation of global variables (globalValid): each global variable x of the analyzed program (i.e., that is bound in the global environment to some memory block b) is in a valid block. This is a useful invariant since the properties of valid blocks (permissions, contents…) are preserved when allocating new blocks, i.e.,

```
Record invariant (cs: concrete_state) : Prop := {
 globalValid: (* Globals are allocated *)
 ∀ x b, Genv.find_symbol ge x = Some b → b ∈ Mem.valid_block (snd cs);

 localNotGlobal: (* Globals and locals are in different blocks *)
 ∀ f t e, In (f, (t, e)) (fst cs) →
 ∀ x b y b' z,
  Genv.find_symbol ge x = Some b →
  e ! y = Some (b', z) →
  b ≠ b';

 noRec: (* All functions are different (i.e., no recursion) *)
 list_norepet (map fst (fst cs))
 (* … *)
}.
```

Figure 6.5: Invariant of the reachable concrete states

```
Definition permissions : Type := Map [ ablock, (Z * permission) ].
Definition permissions_gamma stk : gamma_op permissions fmem :=
 λ ab m,
 ∀ b sz p b',
  get_perm ab b = Just (sz, p) →
  b' ∈ ablock_gamma stk b →
  ∀ i, 0 <= i < sz → m.(perm) b' i = Some p.
```

Figure 6.6: Permission domain

when calling functions, whereas nothing is guaranteed about invalid blocks: this justifies that abstract properties known about global variables are not invalidated by function calls. An other important property that is stated in the invariant is that global and local variables never alias (localNotGlobal): no global variable can be allocated in the same block as a (stack) variable. This is indeed needed to justify the fact that updating some information about one cell preserves what is known about other cells.

Using this kind of invariant in the concretization relation enables to prove properties that do not hold for all programs but only for some. For instance, we state in the invariant that no function appears twice in the call-stack (noRec); and this is only true because the analyzer aborts when it encounters recursive calls.

One of the main roles of the memory domain w.r.t. progress verification is to ensure that pointers are properly used. This is enforced in the concretization relation of the types domain: whenever a cell is given a non-trivial (abstract) type, then accessing to this cell cannot result in an undefined behavior. This is in particular established on stores: for every cell that may be assigned during a store, we check that it is *in bounds*, i.e., that its offset lies within the bounds that are known for its block.

This requires an additional component in the memory domain: a permission domain

(shown on Figure 6.6). For each (abstract) block, we keep track of its size and the permission associated to the cells in this block. The specification of this domain states that an abstract permission map ab represents all memories m such that, for each (abstract) block b bound to size sz and permission p, any offset i between 0 and sz in the concrete block b' represented by b has permission p.

6.3.3 Progress Verification in Numerical Domains

Wrapping all numerical domains in the block_mon monad would require to rewrite them all. We chose a more light-weight modification of the interface of these numerical domains: one function is added to the interface:

nonblock: nexpr var → t → bool

This function is used by the memory domain at the end of numerical conversion, after calling nconvert (see § 5.3.3), to ensure that every numerical expression considered during the analysis is not stuck.

6.3.4 Summary of progress checks

All progress checks that are performed during the analysis are summarized in Figure 6.7. They are grouped in four categories.

Numerical checks correspond to the ones done in the nonblock function of the numerical domains; they amount to verifying for each operator, that the types and ranges of their arguments are correct.

Memory checks are performed in the memory domain: they relate to permissions, variable initialization, and pointer validity. Pointer subtraction is only allowed between pointers of the same block; pointer equality (== and !=) is only defined for valid pointers; pointer inequality (<, <=, > and >=) is only defined for weakly valid pointers of the same block. Checking that a pointer (expression) is valid involves the points-to domain, which predicts a set of targeted blocks, the permission domain, which knows the sizes and permissions associated to these blocks, and the numerical domain, which is able to prove that the offset is between zero and the smallest size of these blocks.

Well-formedness checks are mostly syntactic and not really interesting. Programs failing to pass them may nonetheless reveal bugs in their generation process: for instance, a development version of CompCert would give the same name to two built-in functions.

The last category, named “analyzer limitations”, corresponds to checks that are not required by the semantics. Some features of the language are not handled by the analyzer (recursion, external function calls, inline assembly etc.) and failing on programs that do use them is mandatory for the analyzer to be sound. Finally, since the memory domain enumerates all possible targets of pointers when they are dereferenced, the analysis raises an alarm, for efficiency reasons, if there are too many such targets to explore when performing an abstract store.

6.4 More Numerical Domains

The Verasco analyzer, is by design parameterized by the underlying numerical domain. In addition, the interface of this domain is general enough so that any domain from the

Numerical

- Expressions are well-typed
- Divisions and modulo: divisor is not null and, in the signed case, we don't divide `min_signed` by `-1`
- Shifts: the shift amount, as an unsigned integer, is smaller that the bit-width

Memory

- Alignment of memory accesses
- No dereference of a non-pointer value
- No read from uninitialized register or memory location
- Targets of stores have sufficient permissions (in-bound access in a writable block)
- Function pointers have a null offset, and target a function
- The main function returns an integer value
- Pointer comparisons and subtractions

Program well-formedness

- There is a non-extern `main` function
- Signature on call-site agrees with the declaration of the called function
- Good number of arguments
- No two local variables (including function parameters) have the same name
- Goto targets are existing labels
- All used symbols (e.g., variables in expressions) are declared
- Chunks `ManyXX` are not used for memory access
- No two functions have the same name

Analyzer limitations

- No recursive calls
- No unknown external function
- No annotations, built-ins, inline assembly fragments etc.
- Pointers used in `store` instructions should be known to have less targets than the value of the `max_concretize` run-time parameter

Figure 6.7: Summary of progress checks in the Verasco analyzer

literature can implement it. A major issue is that numerical domains usually represent unbounded integers rather than machine integers.

Therefore, Verasco provides a generic functor that makes any relational numerical domain cognizant of the wrapping behavior of integers on overflows. This functor has been applied to non-relational domains as intervals and congruences, and also to the domain of polyhedra. Several domains can also be combined so as to communicate.

6.4.1 Handling Machine Integers

Numerical domains such as intervals and polyhedra are well understood as abstractions of unbounded mathematical integers. Subtleties of machine-level integer arithmetic complicate static analysis. For specific abstract domains such as intervals and congruences, ad hoc approaches are known, such as strided intervals [RBL06], wrapped intervals [Nav+12], or reduced product of two intervals of $\mathbf{Z}$, tracking signed and unsigned interpretations respectively, as we have seen in a previous chapter (§ 3.3.3). These approaches are difficult to extend to other domains, especially relational domains (notice however the polyhedral domain with wrapping of Simon and King [SK07]). In Verasco, we use a more generic construction that transforms any relational domain over mathematical integers ($\mathbf{Z}$) into a relational domain over N-bits machine integers with modulo-2^N arithmetic, provided this domain satisfies the interface of "ideal" relational domains (similar to the interface given in Figure 3.9, using ideal expressions rather than CFG expressions; ideal expressions operate on ideal numbers rather than machine integers). Consider such a domain, with abstract states A, that represents, through its concretization relation γ, ideal environments ρ: var $\to$ Z. We can build an abstract domain, as we did in the memory domain (§ 5.4.3), with the same data-type A. The concretization is weakened by point-wise lifting of the following compatibility relation:

```
Definition compat (z: Z) (i: int) : Prop := (i = Int.repr z).
```

The abstract transformers, which are defined over ideal expressions in the original domain, are composed with a conversion function that translates numerical expressions (of type nexpr, see § 3.3.1) into ideal expressions.

The conversion of numerical expressions is non-trivial only in the case of operators that are not compatible with the compat relation, i.e., the ones that come in two flavors, depending on the signedness interpretation of their operands (division, comparison…). These operators behave consistently with the compat relation only when their arguments are in a specific range: [0; Int.max_unsigned] for *unsigned* operators, and [Int.min_signed; Int.max_signed] for *signed* operators. The conversion tries to shift the arguments of these operators, by adding to them a constant multiple of 2^{32}; if this fails, an interval covering the whole range is returned. The correctness of the conversion of expression is expressed by a theorem similar to nconvert_correct, stated in Section 5.4.4.

In addition, the conversion performs the progress checks discussed in Section 6.3. Indeed, the nonblock operator is specific to our analyzer and usually not provided in standard numerical domains.

6.4.2 Polyhedral Domain

Thanks to this generic adaptation of (relational) numerical domains, various existing domains can be used. The most notable example is the domain from the Verasco Polyhedra

Library [FB14]. This implementation uses the methodology of a posteriori validation: it is programmed in OCaml and each operator returns certificates along with the results. Said certificates enable a checker, written in Coq, to verify that the result is valid. Only the checker is proved correct, which is dramatically simpler that verifying the full optimized implementation of the domain. Experiments assess that the run-time cost of the additional validation is acceptably low.

6.4.3 Communication Between Domains

Numerical domains are complementary: they are able to infer different kinds of properties that may all be needed to achieve good precision. Moreover, domains may be able to use the properties inferred by other domains to discover more precise invariants. The following example program illustrates that the domain of interval needs the information discovered by the domain of congruences to prove the safety of the analyzed program.

```
1  int t[N];
2
3  int main(void)
4  {
5    int * p = t;
6    while ( p < t + N )
7      ++p;
8    return 0;
9  }
```

In this example, pointer p visits in sequence all N cells of array t (where N is a compile-time constant). Since this is an array of 32 bit integers, the size of t is 4×N octets and the offset of p will successively take the values 0, 4…4×N. The critical point for the analysis is the comparison between p and the end of array t, at line 6. The value t + N is *one past* the address of the last byte of the array. This is known as a *weakly* valid pointer: comparison of such pointers is allowed, but dereferencing them is forbidden. In order to prove that this comparison is safe, the analyzer needs to establish that pointer p is always weakly valid, i.e., that its offset is in range [0; 4×N].

An interval domain can infer that, at the beginning of the loop body, before the incrementation of p, the offset of this pointer is in the range [0; 4×N − 1]. Thus, after the incrementation, this offset is known to be in the range [4; 4×N + 3]. Therefore, when the guard at line 6 is evaluated, the interval domain knows that the offset of p is in the range [0; 4×N + 3]. This is not enough to prove that the comparison is safe, since offsets 4×N + 1, 4×N + 2 and 4×N + 3 are not weakly valid.

Fortunately, the congruence domain is able to infer that the offset of p is a multiple of four. This fact alone is not enough to prove that the pointer is weakly valid: both domains need to interact to discharge this proof obligation.

The most precise approach for combining domains is to build their *reduced product* [CC79]. However, this approach has some known drawbacks: a new domain has to be designed, implemented and proved for each possible combination of domains; and the reduced product may not exist or be efficiently implemented. In Verasco, we rely on a more flexible scheme, inspired from Astrēe, mostly contributed by J.-H. Jourdan [Jou+15].

Domains are composed in sequence, and communication between them is directed from the beginning of the sequence towards its end. The interface of numerical domains

comprises a finite set of *channels* in which a domain may read information coming from another domain that comes before it, or send information to the domains that come after it. The composition operator is thus generic (i.e., it does not depend on the particular abstract domains it operates on) and plugs the output port of each domain into the input port of the next one. Notice that channels are lazy: a domain will write to it only when an other one attempts to read from this channel.

For instance, the congruence domain implements the get_congr channel, and the interval domain queries it when performing backward evaluation, so as to tighten the bounds of an interval, as in the example above: from the interval $[0; 4\times N + 3]$ and the congruence information "multiple of 4", the more precise interval $[0; 4\times N]$ can be deduced.

6.5 On Using the Analyzer

We conducted preliminary experiments with the executable C♯minor static analyzer obtained after extraction. We ran the analyzer on a number of small test C programs (up to a few hundred lines). The purpose was to verify the absence of run-time errors in these programs, including violations of user-defined assertions (invariants that are expressible as C expressions).

6.5.1 Common Pitfalls

We now describe some issues encountered when running the Verasco analyzer on these programs.

Wrong main signature The Verasco analyzer rejects programs with the common signature int main(int argc, char** argv); since, as stated in the CompCert manual [Ler15b], programs do not have access to command line arguments.

The analyzed program could be (automatically) wrapped into one with a suitable main function that calls the original one with null arguments, as it is done in the reference interpreter. This might not be desirable, as highlighted by the following example.

```
1 int
2 main(int argc, char **argv)
3 {
4   if ( argc < 2 )
5     exit ( EXIT_FAILURE );
6   // ...
7 }
```

If this program is analyzed only with null arguments, then most of the code (not shown but suggested by the ellipsis) will be considered dead and silently ignored.

External calls Some programs rely on library functions, whose code is not available or trusted. Unfortunately, since external calls may completely obliterate the memory, the Verasco analyzer fails on such calls: it raises an alarm and the current branch is cut (as if that function never returned). (Another possibility would be to assume that the external call behaves as a no-op, still raising an alarm, but continuing the analysis.)

So as to be able to analyze programs with such library calls, the analyzer provides three primitives: any_int64(), any_double(), and verasco_assume(b). The two first functions nondeterministically return a value of the requested type. The last one constrains the set of possible executions: the boolean expression b is necessarily true when the call verasco_assume(b) is reached.

All other library functions need to be defined, as exemplified below, either with a (possibly naive) deterministic implementation, or with calls to the aforementioned primitives to specify the function. The resulting specification may be partial; its precision may be refined depending on the analyzed program.

```
1  double
2  fabs(double d)
3  {
4    if (d <= 0.)
5      return - d;
6    return d;
7  }
```

```
8  double
9  cos(double d)
10 {
11   double res = any_double();
12   verasco_assume(-1. <= res && res <= 1.);
13   return res;
14 }
```

These primitives can also be used to model user inputs, or more generally any interaction between the analyzed programs and their environments.

Large arrays Some programs operate on large arrays to perform I/O or to hold static data. When reading from a large array, a summary of all cell contents that may be fetched is computed. This is done by enumerating all these cells. The command-line parameter -max-concretize enables to set the limit.

Therefore, analyzing programs with such arrays can be very costly, in both time and memory footprint. When precisely knowing the contents of these arrays is not required for the analysis, they can be abstracted as described in a following paragraph (about abstract data-types).

Array initialization Proving that an array is properly initialized is known to be a difficult task that requires dedicated analyses or domains [GRS05; NS13; HP08]. Indeed, consider the following example program that initializes an array a with zeros, in which N is a strictly positive constant.

```
1  int a[N];
2  for ( int i = 0; i < N; ++i )
3  {
4    a[i] = 0;
5  }
```

When the analysis first reaches the loop header, nothing is known about the array contents and i is known to be zero. At the end of the loop body, the first cell is known to be initialized (with value zero) and i is known to be 1. The analysis thus considers again the loop header with an over-approximation of both states, loosing the information about the contents of the first cell.

To overcome this difficulty, array initialization loops can be unrolled. The iterator will recognize calls to a function named verasco_unroll as a hint for unrolling, at analysis time, the enclosing loop.

Relational loop invariants A frequent enough pattern is to use a pointer as loop index, preferring, in the examples below, the second form to the first one.

```
for ( i = 0 ; i < len ; ++i)
  /* ... */ p[i] /* ... */

for ( ; 0 < len ; --len, ++p )
  /* ... */ *p /* ... */
```

In order to prove that the array access is valid, the required loop invariant is, in the first case $0 <= i <= len$, and in the second case $0 <= p \wedge 0 <= len \wedge p + len = len_0$, where len_0 is the initial value of the variable len. This last invariant is relational, hence it can only be represented in a relational domain, such as polyhedra. Unfortunately, this domain is very expensive: applying weakly relational domains [Min04] or techniques like *packing* [Bla+03] could be helpful. These improvements are left as future work; for now the analyst is kindly required to rewrite the loops.

6.5.2 Sample Results

We now present various interesting sample programs that we have analyzed. They are distributed with the source code of Verasco [Web].

Function integration The example integr.c is a small program adapted from a CompCert benchmark. Most of its code is given below.

```
typedef double (*fun)(double);
fun functions[N] = { id, square, fabs, sqrt };

double
integr(fun f, double low, double high, int n) {
    double h, x, s; int i;
    h = (high - low) / n; s = 0;
    for (i = n, x = low; i > 0; i--, x += h)
        s += f(x);
    return s * h;
}

int main(void) {
    for (int i = 0; i < any_int(); ++i) {
        double m = any_double();
        verasco_assume( 1. <= m );
        int n = any_int();
        verasco_assume (0 < n);
        integr(functions[i % N], 0., m, n);
  }
  return 0;
}
```

This program repeatedly computes an approximation of the integral of a positive function between zero and some number greater than one. The function in question is picked from a constant array. It stresses various aspects of the analyzer such as function pointers, arrays, floating point and machine arithmetic.

Numerical simulations Two programs of a few hundred lines taken from the CompCert benchmark, nbody.c and almabench.c, feature heavy numerical (floating point) computations and array manipulation.

Cryptographic routines The smult.c example performs scalar multiplication. It is taken from the cryptography library NaCl. Scalars and group elements are stored in arrays of bytes or unsigned integers. Many of these arrays are initialized within a loop: to be able to prove that they are indeed initialized, we annotated the program to request full unrolling of these loops during analysis.

Abstract data-type specification The example spectral.c comes from the shootout benchmark, is only 81 lines long, and performs many numerical (floating point) computations in arrays. It computes the so-called *spectral norm* of the infinite matrix A such that:

$$A_{i,j} = \frac{1}{\frac{(i+j)\times(i+j+1)}{2} + i + 1}$$

This program takes as input a size N; a typical value for this input is 5500. It then builds a unit vector of size N and multiplies it twenty times by matrix A and its transposed A^T (both truncated to the square matrix of size N^2). It uses three arrays of size N, that are dynamically allocated (since their sizes are a run-time parameter).

This is an issue for the Verasco analyzer which is currently not able to deal with such data structures (dynamically allocated arrays). However, we can treat this data structure as an abstract data type whose implementation is trusted, give it a specification and analyze the program nonetheless. The specification needs not be complete; in particular no information is needed about the contents of the array (apart from being well-defined floating-point values). The signature of the array abstract data-type follows.

```
1 /** Arrays as an abstract data type. */
2 struct Array;
3 typedef struct Array array;
4
5 void array_init(array*, int);
6 double array_get(array*, int);
7 void array_set(array*, int, double);
8 void array_free(array*);
```

This abstract data-type is given two implementations (see Figure 6.8): one for program execution (on the left) and one for program analysis (on the right). The concrete implementation is a naked array that is directly accessed. The weak specification only checks that accesses are in-bounds; it does not specifies what happens to the array contents.

Analyzing a specification rather than the actual implementation yields weaker guarantees on the actual program. However, it has various benefits.

- The program, seen as a client of the trusted array library, can be proved free of undefined behaviors, provided that the library specification characterizes all sources of undefined behaviors.
- Analysis time and memory consumption do not depend on the size of the arrays.

```
/* Concrete array implementation. */
struct Array {
  double *data;
};
double array_get(array *a, int i)
{
  return a->data[i];
}
void
array_set(array *a, int i, double d)
{
  a->data[i] = d;
}
void array_init(array *a, int size)
{
  a->data = calloc(size,
      sizeof(*a->data));
}
void array_free(array *a)
{ free(a->data); }
```

```
/* Weak array specification. */
struct Array {
  int size;
};
double array_get(array *a, int i)
{
  int size = a->size;
  assert(0 <= i && i < size);
  return any_double();
}
void
array_set(array *a, int i, double d)
{
  int size = a->size;
  assert(0 <= i && i < size);
}
void array_init(array *a, int size)
{ a->size = size; }
void array_free(array *a)
{}
```

Figure 6.8: Two "implementations" of arrays

- Using relational domains (e.g., polyhedra), the size of the arrays can be left unknown.

Notice finally that this transformation cannot be performed automatically during the analysis because of the issue of array initialization. Moreover, such transformation is not beneficial if the contents of the arrays needs to be precisely tracked.

Preliminary results On the examples described above, Verasco was able to prove the absence of run-time errors. This is encouraging, since these examples exercise many delicate aspects of the C language: arrays, pointer arithmetic, function pointers, and floating-point arithmetic. The table below gather some of the examples and gives the order of magnitude of the analysis time (on a laptop with a 3 GHz CPU).

Program	Size (lines)	Time (s)	Comment
integr.c	42	$1 \cdot 10^{-1}$	
smult.c	330	$4 \cdot 10^{1}$	
nbody.c	179	$1 \cdot 10^{1}$	
almabench.c	352	$8 \cdot 10^{1}$	
arc4.c	157	$2 \cdot 10^{2}$	
bubble.c	65	$1 \cdot 10^{1}$	array of size 10, with polyhedra
spectral.c	213	1	array of size 5500
spectral.c	213	$1 \cdot 10^{2}$	array of unknown size, with polyhedra

The analysis times are quite high, in particular when the analyzed program manipulate large arrays. A pre-analysis program transformation (triggered by the -funload command-line argument) dramatically shortens the analysis time on some examples: it pulls loads

out of expressions and prevents combinatorial explosion of the conversion function of the memory domain.

6.6 Conclusion

This chapter presents the Verasco static analyzer: it operates on an intermediate representation of the CompCert compiler (C♯minor) to automatically prove that the analyzed program is free of undefined behaviors, and its soundness proof is machine-checked. It is built following the modular architecture discussed in Chapter 3, it features the precise memory domain detailed in Chapter 5 along with several numerical domains.

Simple experiments assess that this analyzer is able to prove the safety of several non-trivial programs featuring most of the C language constructs. Still, this analyzer could be improved in various directions. In particular, when the analyzer fails to prove that a program is safe and raises an alarm, understanding the meaning and the origin of this alarm can be very difficult. The Astrēe analyzer features a sophisticated framework for a better understanding of the analysis results [Riv05a; Riv05b]. Integrating similar facilities in the Verasco analyzer could tremendously improve its usability.

The execution times of the analyzer are in some cases higher that what could be expected for such an analyzer. The Astrēe analyzer, which is a major source of inspiration for this work, is both precise and efficient. Our primary focus has been on precision, in particular by using suitable interfaces which enable the implementation of precise relational numerical domains.

The analyzer operates on the C♯minor intermediate language, thus some analysis results may not apply to intermediate representations that come earlier in the compilation chain. For instance, some C programs are unsafe but become safe when compiled to C♯minor, since some type annotations are erased in the process. The C♯minor analyzer could be complemented so as to perform additional safety checks to ensure that the original Clight program is actually safe.

Chapter 7

Conclusion

7.1 Summary

Static analyzers are a key tool to increase reliability of software and our trust in it. The soundness of an analyzer is even more critical that the safety of the analyzed program, since a buggy analyzer could increase our confidence in a wrong program. In this work, we have shown how to build sound static analyzers with very strong guarantees: in order to trust an analysis result, there is no need to trust the implementation of the analyzer and even its proof; only the definition of the semantics of the analyzed program, the soundness statement of the analyzer, and the Coq kernel.

Throughout this document, we have described and applied a methodology to design and prove sound static analyzers in Coq, and shown central interfaces of core components. The three analyzers described in this work all share the same architecture and some modules, even though they operate on different languages.

This work focuses on *low-level* languages, i.e., languages which feature little abstractions (e.g., objects, functions, variables, types) or in which abstractions are broken on purpose by the programmers. These languages are pervasive as they are closer to the concrete machines that actually run the programs: every program eventually gets transformed into a low-level one so as to be executed; or some programs are directly written in such languages because they need this freedom of breaking the abstractions, for instance to implement operating systems, drivers, or standard library of higher-level languages.

In order to soundly analyze programs written in such languages, it is not possible to assume that the analyzed programs does not break the abstractions. At least, if the analysis makes such assumptions, it must verify, as part of the analysis, that the assumption holds.

On one the most extreme examples of abstractions that can be broken in low-level languages is the notion of *program text*: an immutable document that the machine executes. In self-modifying programs, this document is an ordinary piece of data that can be read and even modified and extended at execution time. In chapter 4 we studied the how to design and verify sound static analyses of such programs. The resulting analyzer is able to verify safety properties of several self-modifying programs.

Low-level languages provide a low-level view of the memory. Notably, fields of structures are referred to by offset (i.e., a number) rather than by name, making pointer arithmetic pervasive; a given piece of data can be accessed piecewise rather than at once; and a given bit pattern can be interpreted as data of different types. All these properties are soundly taken into account in the abstract memory domain presented in chapter 5. This domain is plugged into a value analysis of the CFG intermediate representation of the CompCert compiler.

Finally, chapter 6 reports on the Verasco static analyzer, a result of the combined effort

```
1  void *
2  f(int x, int* p)
3  {
4    if (x <= 0)
5      return (void*)(-1);
6    return p;
7  }
```

```
8  int
9  main(void)
10 {
11   int i;
12   int * p = f(0, &i);
13   return p + 1 < p; /* Error! */
14 }
```

Figure 7.1: Incorrect C program which is proved safe by Verasco

of the Verasco team. This analyzer has been built following the methodology and using abstract domains described in this document; it operates on the C♯minor intermediate representation of CompCert and automatically proves the safety of the analyzed programs. It soundly handles *all* features of the C programming language but is extremely imprecise when the analyzed program resorts to recursive calls or dynamic memory allocation, which is unexpected form safety-critical embedded softwares.

7.2 Perspectives

7.2.1 Improving the usability of the Verasco analyzer

The Verasco analyzer could be improved in various ways. We describe here some shallow improvements, mostly related to the usability of the analyzer.

Analysis closer to the source The analysis could take place closer to the source code, for instance at the Clight level. For programs in which all expressions are side-effect free (which can be syntactically checked), the compilation from CompCert C to Clight is the identity. So analyzing Clight would indeed achieve analysis of the source code. Rewriting the analyzer to operate at the Clight level would be tedious and difficult. A more efficient approach would be to complement the analysis at C♯minor level with a simple analysis at the Clight level which would prove that the compilation from Clight to C♯minor does not remove errors. This amounts to proving that the evaluation of expressions in Clight cannot be stuck because of typing errors. The analysis at the C♯minor level provides enough information to type-check the program at Clight level.

Consider the C program shown on Figure 7.1. There, the function f takes an argument x and checks that it satisfies some condition; if it does, then the execution proceeds normally and the function returns a valid pointer. Otherwise, if the condition does not hold, an error is returned. Such a protocol is found in several POSIX functions (e.g., mmap, shmat).

So as to prove that a program calling a function similar to f is safe, the analyzer must prove that for every call, the precondition holds. Unfortunately, this example program is unsafe, as it uses the erroneous result of f as if it were a pointer. But this error is not caught by Verasco since it operates on the C♯minor intermediate representation: at this level, the addition does not verify the type of its arguments as at the Clight level.

In order to verify that such a Clight program is well-typed, the type-checker needs to verify that all calls to f satisfy its precondition, which involves reasoning about arithmetic.

Such a type-checker can therefore benefit from the results of a prior analysis at the C♯minor level.

Understanding alarms When the analyzer fails, it is very difficult to understand the alarms. When the error is raised, the error message is very local: the only available information is the current abstract transformer being executed and the current abstract state. The location of the offending instruction is not available, and no context is given. One way to recover some context would be to log every step of the analysis. This does not work in practice as traces are huge and unreadable: too much data is as useless as no data, without proper tooling to mine it. X. Rival [Riv05a; Riv05b] proposes some inspiring work on this topic applied to Astrēe.

Annotation language The analyzer can only be used to prove, in addition to safety, properties that are expressible as C expressions. However, as it stands, the analyzer infers more involved properties. To broaden the application range of the analyzer, one could design an assertion language richer that the C expression language, and ask to check properties like: a given expression has a given type; a pointer is valid; two pointers are comparable (i.e., point to the same block); two pointers are incomparable (so as to check `restrict` annotations).

Moreover, such a communication mechanism between the source code and the analyzer could include debugging primitives that would be a first step towards a better understanding of the raised alarms.

Use analysis results for compilation The analyzer could be better integrated to the CompCert compiler: the analysis results could be used by the compiler, e.g., for optimizations (replace constant expressions by they result, remove useless tests and dead code). One major issue is that an analysis result may no longer be sound after any program transformation (optimization or compilation pass). In particular, when a program transformation moves code around, changes the scopes of variables, it is unlikely that the analysis result holds as is. It would be nice to be able to transform the result along with the program rather than re-analyzing the transformed program. Kunz [Kun09] describes some techniques of *certificate translation* to transform the description of semantic properties across several compilation passes.

7.2.2 Weak cells and summarization

In the current state of the memory abstract domain (discussed mainly in chapter 5), each abstract cell represents exactly one concrete cell (or none). All allocation functions (which map abstract cells to the concrete cells they may represent) agree on all (valid) cells.

Relaxing this invariant and allowing a single abstract cell to represent several concrete cells at one has various uses but raises new challenges; in particular, no strong update can be performed on such *summary cells*: the properties inferred about the contents of these cells can only loose precision during the analysis, they never get more precise.

Array summarization A single abstract cell can be used to represent a summary of all elements in an array. This prevents the analyzer to compute a precise approximation

for each element, but in some cases the analyzer cannot achieve great precision (e.g., when the array holds input data, or when the contents of the array is produced by a program beyond the capacity of the analyzer, say a cryptographic hash function). Using one cell for the whole array instead of one cell per element should improve the efficiency of the analyzer: the memory footprint is lower since only one cell is bound in the abstract states rather than many; the conversion of an expression involving an access to some cell produces only one expression rather than one expression per cell.

Since only weak updates can be performed on weak cells, it is harder to prove that such cells are initialized. A possibility would be to summarize an array after it has been initialized. As in the case of unrolling of initialization loops, the user could be asked to write down annotations in the program to suggest which arrays should be summarized and when. For instance, in the case of the following initializing loop:

```
1       for (int i = 0; i < 4096; ++i) { t[i] = 0; }
```

the current state of the memory model requires this loop to be fully unrolled and the array not summarized. It is unfortunate to have to go through four thousands iterations to prove that this loop indeed initializes the array, and relying on dedicated abstract domains could improve this situation.

Recursive calls Programs with recursive function calls are challenging first for the termination of the iterator, since a systematic (recursive) analysis of all recursive calls may not terminate. Therefore, different calling contexts must be merged so as to guarantee that each function is analyzed in finitely many such contexts. In particular, this may involve merging abstract states with different call stacks.

Also, each concrete cell which represents a local variable cannot be exactly represented by one abstract cell: when analyzing a recursive function f with a local variable x, there cannot be as many abstract cells as live variables x (one per running or suspended function f). Indeed, so as to guarantee the termination of the analysis, at least one abstract cell should represent many variables x.

Therefore, being able to reason about summary cells in the abstract memory domain is a prerequisite for analyzing programs with (statically unbounded) recursive calls. But as we have seen, it is far from enough.

Dynamic memory allocation As for variables that are local to recursive functions, a static analyzer may not be able to bound the number of dynamically allocated cells (for instance, if such allocation occurs inside loops). Therefore a single abstract cell should represent several dynamically allocated concrete cells.

A possibility is to introduce one abstract block to represent the dynamically allocated heap and have malloc return the pointer to the offset zero in this block. This implies that all dynamically allocated regions are merged. This is far from ideal and completely useless since there will be no way to prove that such memory is at some point initialized. Indeed, only weak updates can be performed on summarized cells; so if malloc returns a pointer to an uninitialized summarized cell, no information about the contents of such cell can ever be computed. Balakrishnan and Reps [BR06] present an abstract domain targeting this issue.

Having several abstract blocks for dynamically allocated memory could improve the precision. A common practice is to have one abstract block per static call to malloc (known

as the allocation-site abstraction). Due to the structure of the iterator, it is unclear how to produce identifiers.

Notice that safe memory reclamation of summarized cells is out of scope. The analysis is unlikely to be able to prove safe any use of maybe-reclaimed memory[1].

So as to properly handle summary cells in relational numerical domains, particular care is required. Suppose that a guard like if (t[i] < 0) is analyzed, and that there is one summary cell s representing the contents of the array t. This could result in the query assume (s < 0) addressed to a numerical domain. If this domain is unaware of summary cells, it would infer that *all* elements of the array are negative in the *true* branch, and that all of them are positive in the *else* branch; which is not valid (unless the array has only one element). Gopan *et al.* [Gop+04] propose to extend the interface of relational numerical domains with additional primitives and implement support for summary cells with these primitives. A less intrusive and simpler approach would be to systematically replace, in queries directed to numerical domains, summary cells by a conservative interval over-approximating their contents.

7.2.3 Verified static analysis of concurrent programs

All programs and programming languages that have been considered in this work are sequential, or single-threaded. However, the current way to improve the performances of software is to execute several tasks at the same time: most processors are *multi-cores*, they feature distinct processing units that operate simultaneously on the same memory. Programming such multiprocessors introduces new kinds of errors, including in particular *data-races*: several concurrent accesses to a same memory location, one of them being a write.

Static analyses can help in proving that concurrent programs are free of data-races. To build and verify a sound static analyzer for languages featuring concurrency, a precise formal semantics of these languages is first needed. The *interleaving* semantics composes per-thread semantics with a non-deterministic scheduler which runs one step in one thread at a time. Taking into account all possible interleavings during an analysis seems to be too costly. A. Miné proposes a scalable approach [Min11; Min13], featuring rely-guarantee reasoning [Jon83]: each thread is analyzed independently (no interleaving at all) under an assumption, named *interference*, about the changes that the shared memory may undergo as a result of the execution of the concurrent threads. These interferences are deduced from the results of the analysis of each thread. The dependency between the computation of the interferences and the per-thread analysis is solved as an other fix-point problem.

Unfortunately, interleaving semantics do not accurately model multiprocessors or languages featuring concurrency: actual behaviors, referred to as *weak memory models*, are not captured by interleavings. For instance, x86 processors feature *store-buffers*: when a thread writes a value to some location, this value hits the shared memory only after it exits the buffer; meanwhile, other threads may read *past* values at that location. This introduces behaviors that cannot be explained by a simple interleaving of the reads and writes of the threads.

[1] Moreover, the semantics of free —the function that releases dynamically allocated memory— in CompCert is not yet satisfactory [Ler15a].

So as to precisely take into account these *weak* behaviors in an analysis, Meshman *et al.* [Mes+14] propose a program transformation that makes explicit the store-buffers in the source code and analyze it under an interleaving semantics. This completely isolates the verification problems due to the weak memory model. However, since the analysis is unaware of the memory model, it cannot be designed in order to precisely take into account the buffers.

The set of behaviors introduced by weak memory models is, in some cases (including the memory model of x86 processors), characterized by a small family of reorderings. If the abstract semantics is preserved by these reorderings, then an analysis which only considers the interleavings is sound in the weaker memory model. The interference semantics of Miné is indeed preserved by a set of "reasonable" reorderings.

In case of weaker semantics, as for instance the one of POWER multiprocessors, or of the Java programming language, it is unclear how to encode the memory model as a program transformation or as reorderings. Designing sound and precise analyses aware of such weak memory models would benefit from a better understanding of these models.

Bibliography

[AB07] Andrew W. Appel and Sandrine Blazy. "Separation Logic for Small-Step Cminor". In: *Proc. of TPHOLs*. Vol. 4732. LNCS. Springer, 2007, pp. 5–21 (cit. on p. 98).

[Akr+08] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. "Preventing memory error exploits with WIT". In: *Symposium on Security and Privacy* (*SP*). IEEE, 2008, pp. 263–277 (cit. on pp. viii, 1).

[Ale96] Aleph One. "Smashing The Stack For Fun And Profit". In: *Phrack* 7.49 (Nov. 1996) (cit. on pp. vii, 1).

[App14] Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014 (cit. on p. 98).

[Bal+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. "Thorough static analysis of device drivers". In: *ACM SIGOPS Operating Systems Review*. Vol. 40. ACM, 2006, pp. 73–85 (cit. on pp. viii, 2).

[Bal07] Gogul Balakrishnan. "WYSINWYX: What You See Is Not What You eXecute". PhD thesis. Madison: University of Wisconsin, 2007 (cit. on p. 64).

[Ber09] Yves Bertot. "Structural abstract interpretation, A formal study in Coq". In: *Language Engineering and Rigorous Software Development, ,nternational LerNet ALFA Summer School 2008, revised tutorial lectures*. Vol. 5520. LNCS. Springer, 2009, pp. 153–194 (cit. on p. 70).

[Bes+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. "A few billion lines of code later: using static analysis to find bugs in the real world". In: *Communications of the ACM* 53.2 (2010), pp. 66–75 (cit. on pp. viii, 2).

[BGL06] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. "A structured approach to proving compiler optimizations based on dataflow analysis". In: *TYPES*. Vol. 3839. LNCS. Springer, 2006, pp. 66–81 (cit. on pp. xi, 4, 46, 70, 71).

[BHV11] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. "Refinement-Based CFG Reconstruction from Unstructured Programs". In: *Verification, Model Checking and Abstract Interpretation* (*VMCAI*). Vol. 6538. LNCS. Springer, 2011, pp. 54–69 (cit. on p. 71).

[BL09] Sandrine Blazy and Xavier Leroy. "Mechanized semantics for the Clight subset of the C language". In: *Journal of Automated Reasoning* 43.3 (2009), pp. 263–288 (cit. on p. 52).

[Bla+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "A static analyzer for large safety-critical software". In: *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'03*). Vol. 38. San Diego, California, USA: ACM, June 2003, pp. 196–207 (cit. on pp. 38, 110).

[Bla+13] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. "Formal Verification of a C Value Analysis Based on Abstract Interpretation". In: *Static Analysis Symposium* (*SAS*). Vol. 7935. LNCS. Springer, 2013, pp. 324–344 (cit. on p. 26).

[BLP14] Sandrine Blazy, Vincent Laporte, and David Pichardie. "Verified Abstract Interpretation Techniques for Disassembling Low-level Self-modifying Code". In: *Proc. of the 5th Conf. on Interactive Theorem Proving* (*ITP*). LNCS. Springer, 2014 (cit. on p. 48).

[BMR09] Guillaume Bonfante, Jean-Yves Marion, and D. Reynaud-Plantey. "A Computability Perspective on Self-Modifying Programs". In: *SEFM*. 2009, pp. 231–239 (cit. on p. 71).

[Bou93] F. Bourdoncle. "Efficient Chaotic Iteration Strategies With Widenings". In: *Proc. of FMPA 1993*. Vol. 735. LNCS. Springer, 1993, pp. 128–141 (cit. on pp. 17, 31, 74).

[BR06] Gogul Balakrishnan and Thomas Reps. "Recency-abstraction for heap-allo-cated storage". In: *Static analysis* (*SAS*). Springer, 2006, pp. 221–239 (cit. on p. 118).

[BR08] Gogul Balakrishnan and Thomas Reps. "Analyzing stripped device-driver executables". In: *Tools and Algorithms for the Construction and Analysis of Systems* (2008), pp. 124–140 (cit. on pp. ix, 3).

[BR10] Gogul Balakrishnan and Thomas Reps. "WYSINWYX: What You See Is Not What You eXecute". In: *ACM Transactions on Programming Languages and Systems* (*TOPLAS*) 32.6 (2010), p. 23 (cit. on pp. 47, 58, 61, 71, 72).

[C99] ISO. *The ANSI C standard* (*C99*). Tech. rep. WG14 N1124. ISO/IEC, 1999. URL: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf (cit. on pp. 19, 22, 93).

[Cac+05] David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. "Extracting a data flow analyser in constructive logic". In: *Theoretical Computer Science* 342.1 (2005), pp. 56–78 (cit. on pp. xi, 4, 25, 46, 70, 71).

[Cal+15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification". In: *NASA Formal Methods - 7th International Symposium*. 2015, pp. 3–11 (cit. on pp. viii, 2).

[CC76] P. Cousot and R. Cousot. "Static determination of dynamic properties of programs". In: *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130 (cit. on pp. 28, 73).

[CC77] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Symp. on Principles of Programm. Lang.* (*POPL*). Los Angeles, California: ACM, 1977, pp. 238–252 (cit. on pp. ix, 2, 14, 15, 50).

[CC79] Patrick Cousot and Radhia Cousot. "Systematic design of program analysis frameworks". In: *POPL*. ACM, 1979, pp. 269–282 (cit. on p. 107).

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. "Securing Software by Enforcing Data-flow Integrity". In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. OSDI'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 11–11 (cit. on pp. vii, 1).

[CD04] Solange Coupet-Grimal and William Delobel. "A Uniform and Certified Approach for Two Static Analyses". In: *TYPES*. Vol. 3839. LNCS. 2004, pp. 115–137 (cit. on pp. xi, 4, 46, 70, 71).

[CH78] P. Cousot and N. Halbwachs. "Automatic discovery of linear restraints among variables of a program". In: *5th Symp. on Principles of Programm. Lang.* (*POPL*). Tucson, Arizona: ACM, 1978, pp. 84–97 (cit. on pp. 34, 73).

[Chl11] Adam Chlipala. "Mostly-automated verification of low-level programs in computational separation logic". In: *Conf. on Programming Language Design and Implementation* (*PLDI*). ACM, 2011 (cit. on p. 71).

[Coq15] The Coq development team. *The Coq Proof Assistant*. 1989–2015. URL: https://coq.inria.fr/ (cit. on pp. xi, 4).

[Cou+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The Astrēe analyzer". In: *Proceedings of the European Symposium on Programming* (*ESOP'05*). Springer, 2005, pp. 21–30 (cit. on pp. viii, 2, 33, 45).

[Cou+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. "Why does Astrēe scale up?" In: *Formal Methods in System Design* 35.3 (Dec. 2009), pp. 229–264 (cit. on p. 97).

[Cow+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In: *Usenix Security*. Vol. 98. 1998, pp. 63–78 (cit. on pp. vii, 1).

[CP10] David Cachera and David Pichardie. "A Certified Denotational Abstract Interpreter". In: *Interactive Theorem Proving* (*ITP*). Vol. 6172. LNCS. Springer, 2010, pp. 9–24 (cit. on pp. 25, 46, 70).

[CR13] Bor-Yuh Evan Chang and Xavier Rival. "Modular Construction of Shape-Numeric Analyzers". In: *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013.* 2013, pp. 161–185 (cit. on p. 94).

[CSV07] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. "Certified Self-Modifying Code". In: *Conf. on Programming Language Design and Implementation* (*PLDI*). ACM, 2007, pp. 66–77 (cit. on pp. 48, 64, 70, 71).

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing cheap, resilient, and stealthy opaque constructs". In: *25th Symp. on Principles of Programm. Lang.* (*POPL*). ACM, 1998, pp. 184–196 (cit. on p. 70).

[Cuo+12] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C: A Software Analysis Perspective". In: *Proc. of SEFM 2012*. Vol. 7504. LNCS. Springer, 2012, pp. 233–247 (cit. on pp. 44, 46).

[CVE13] *CVE-2014-0160.* Available from MITRE, CVE-ID CVE-2014-0160. Dec. 2013. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 (cit. on pp. viii, 2).

[DCT08] Saumya K. Debray, Kevin P. Coogan, and Greg M. Townsend. "On the semantics of self-unpacking malware code". Draft. 2008 (cit. on p. 71).

[FB14] Alexis Fouilhé and Sylvain Boulmé. "A certifying frontend for (sub)polyhedral abstract domains". In: *VSTTE*. Vol. 8471. LNCS. Springer, 2014, pp. 200–215 (cit. on pp. xii, 6, 34, 107).

[Fle+10] Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. "Interprocedural control flow reconstruction". In: *Programming Languages and Systems*. Springer, 2010, pp. 188–203 (cit. on p. 72).

[FMP13] Alexis Fouilhé, David Monniaux, and Michaël Périn. "Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra". In: *Static analysis* (*SAS*). 2013 (cit. on p. 34).

[Gon07] Georges Gonthier. "The Four Colour Theorem: Engineering of a Formal Proof". In: *ASCM 2007*. Vol. 5081. LNCS. Springer, 2007, p. 333 (cit. on pp. xi, 4).

[Gon13] Georges Gonthier. "Engineering mathematics: the odd order theorem proof". In: *Proc. of POPL'13*. ACM, 2013, pp. 1–2 (cit. on pp. xi, 4).

[Gop+04] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. "Numeric domains with summarized dimensions". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 512–529 (cit. on p. 119).

[Gra89] Philippe Granger. "Static analysis of arithmetical congruences". In: *International Journal of Computer Mathematics* 30.3-4 (1989), pp. 165–190 (cit. on p. 73).

[Gra91] Philippe Granger. "Static analysis of linear congruence equalities among variables of a program". In: *TAPSOFT'91*. Springer, 1991, pp. 169–192 (cit. on p. 73).

[Gro+05] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. "Cyclone: A type-safe dialect of C". In: *C/C++ Users Journal* 23.1 (2005), pp. 112–139 (cit. on pp. viii, 2).

[GRS05] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. "A framework for numeric analysis of array operations". In: *Proc. of the 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*. 2005, pp. 338–350 (cit. on p. 109).

[Hal+15] Thomas C. Hales et al. “A formal proof of the Kepler conjecture”. In: *CoRR* abs/1501.02155 (2015) (cit. on pp. xi, 4).

[HKS10] M. Hofmann, A. Karbyshev, and H. Seidl. “Verifying a local generic solver in Coq”. In: *Proc. of SAS'10*. Springer, 2010, pp. 340–355 (cit. on p. 46).

[HP08] Nicolas Halbwachs and Mathias Péron. “Discovering Properties About Arrays in Simple Programs”. In: *PLDI*. Tucson, AZ, USA: ACM, 2008, pp. 339–348 (cit. on p. 109).

[JBK13] Jonas Jensen, Nick Benton, and Andrew Kennedy. “High-Level Separation Logic for Low-Level Code”. In: *Symp. on Principles of Programm. Lang.* (*POPL*). ACM, 2013 (cit. on p. 71).

[JHM11] Richard E. Jones, Antony L. Hosking, and J. Eliot B. Moss. *The Garbage Collection Handbook: The art of automatic memory management*. CRC Press, 2011. ISBN: 978-1420082791. URL: http://gchandbook.org/ (cit. on pp. vii, 1).

[Jim+02] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. “Cyclone: A Safe Dialect of C.” In: *USENIX Annual Technical Conference, General Track*. 2002, pp. 275–288 (cit. on pp. viii, 2).

[Jon83] Cliff B. Jones. “Tentative Steps Toward a Development Method for Interfering Programs”. In: *ACM Trans. Program. Lang. Syst.* 5.4 (1983), pp. 596–619 (cit. on p. 119).

[Jou+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. “A Formally-Verified C Static Analyzer”. In: *Proc. of the 42th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*. ACM, 2015, pp. 247–259 (cit. on pp. xii, 6, 38, 73, 96, 107).

[Kan+14] Jeehoon Kang, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. “Towards Scalable Translation Validation of Static Analyzers”. In: (2014). URL: http://rosaec.snu.ac.kr/publish/2014/techmemo/ROSAEC-2014-003.pdf (cit. on pp. x, 3).

[Kar76] Michael Karr. “Affine relationships among variables of a program”. In: *Acta Informatica* 6.2 (1976), pp. 133–151 (cit. on p. 73).

[Ken+13] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. “Coq: The world's best macro assembler?” In: *Symp. on Principles and Practice of Declarative Programming* (*PPDP*). ACM, 2013, pp. 13–24 (cit. on pp. 71, 72).

[Kil73] Gary A. Kildall. “A unified approach to global program optimization”. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1973, pp. 194–206 (cit. on p. 31).

[Kin12] Johanes Kinder. “Towards static analysis of virtualization-obfuscated binaries”. In: *Working Conference on Reverse Engineering*. 2012, pp. 61–70 (cit. on pp. 67, 68, 71).

[Kle+10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: formal verification of an operating-system kernel". In: *Comm. of the ACM* 53.6 (June 2010), pp. 107–115. ISSN: 0001-0782 (cit. on pp. xi, 4).

[KN06] Gerwin Klein and Tobias Nipkow. "A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler". In: *ACM Trans. On Programm. Lang. And Syst.* 28.4 (2006), pp. 619–695 (cit. on pp. xi, 4, 25, 46, 70, 71).

[Kun09] César Kunz. "Program Compilation and Proof Transformation". PhD thesis. INRIA Sophia Antipolis-Méditerranée., Feb. 3, 2009. 167 pp. (cit. on p. 117).

[KV08] Johannes Kinder and Helmut Veith. "Jakstab: A Static Analysis Platform for Binaries". In: *Computer Aided Verification*. LNCS. Springer, 2008, pp. 423–427 (cit. on p. 71).

[LB08] X. Leroy and S. Blazy. "Formal verification of a C-like memory model and its uses for verifying program transformations". In: *J. Automated Reasoning* 41.1 (2008) (cit. on p. 25).

[LD03] Cullen Linn and Saumya K. Debray. "Obfuscation of executable code to improve resistance to static disassembly". In: *ACM Conference on Computer and Communications Security*. 2003, pp. 290–299 (cit. on p. 47).

[Ler06] Xavier Leroy. "Formal certification of a compiler back-end or : Programming a compiler with a proof assistant". In: *POPL* (2006), pp. 42–54 (cit. on p. 19).

[Ler09] Xavier Leroy. "A formally verified compiler back-end". In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446 (cit. on p. 19).

[Ler15a] Xavier Leroy. Personal communication. June 1, 2015 (cit. on p. 119).

[Ler15b] Xavier Leroy. *The CompCert C verified compiler documentation and user's manual*. June 11, 2015. URL: http://compcert.inria.fr/man/ (cit. on p. 108).

[Mar14] André Oliveira Maroneze. "Verified Compilation and Worst-Case Execution Time Estimation". PhD thesis. Université Rennes 1, June 17, 2014. 145 pp. (cit. on pp. xii, 5, 44).

[Mau04] Laurent Mauborgne. "ASTRĒE: Verification of Absence of Runtime Error". In: *Building the Information Society*. Springer, 2004, pp. 385–392 (cit. on pp. viii, 2).

[Mes+14] Yuri Meshman, Andrei Marian Dan, Martin T. Vechev, and Eran Yahav. "Synthesis of Memory Fences via Refinement Propagation". In: *Static Analysis (SAS)*. 2014, pp. 237–252 (cit. on p. 120).

[Min04] Antoine Miné. "Weakly relational numerical abstract domains". PhD thesis. Ēcole Polytechnique, Dec. 2004. 322 pp. (cit. on pp. 34, 73, 110).

[Min06] Antoine Miné. "Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics". In: *Proc. of LCTES'06*. Ottawa, Ontario, Canada: ACM, June 2006, pp. 54–63 (cit. on pp. 45, 46, 73, 91, 94).

[Min11] Antoine Miné. "Static analysis of run-time errors in embedded critical parallel C programs". In: *Programming Languages and Systems*. Springer, 2011, pp. 398–418 (cit. on p. 119).

[Min13] Antoine Miné. "Static analysis by abstract interpretation of concurrent programs". HDR. ĒNS, May 28, 2013 (cit. on p. 119).

[Mor+12] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. "RockSalt: better, faster, stronger SFI for the x86". In: *Conf. on Programming Language Design and Implementation* (*PLDI*). 2012, pp. 395–404 (cit. on pp. 71, 72).

[Myr10] Magnus O. Myreen. "Verified just-in-time compiler on x86". In: *Symp. on Principles of Programm. Lang.* (*POPL*). ACM, 2010, pp. 107–118 (cit. on p. 71).

[Nav+12] Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. "Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code". In: *APLAS*. Vol. 7705. LNCS. Springer, 2012, pp. 115–130 (cit. on pp. 44, 46, 106).

[Nec+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. "CCured: type-safe retrofitting of legacy software". In: *ACM Trans. Program. Lang. Syst.* 27.3 (2005), pp. 477–526. DOI: 10.1145/1065887.1065892 (cit. on pp. viii, 1).

[Nec00] George Necula. "Translation validation for an optimizing compiler". In: *SIGPLAN Not.* 35.5 (2000), pp. 83–94 (cit. on pp. x, 3).

[Nip12] Tobias Nipkow. "Abstract Interpretation of Annotated Commands". In: *Interactive Theorem Proving* (*ITP*). Vol. 7406. LNCS. Springer, 2012, pp. 116–132 (cit. on pp. 25, 46, 70).

[NS13] Đurica Nikolić and Fausto Spoto. "Inferring complete initialization of arrays". In: *Theoretical Computer Science* 484 (May 2013), pp. 16–40 (cit. on p. 109).

[Pic05] David Pichardie. "Interprētation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiēs". In french. PhD thesis. Universitē Rennes 1, 2005 (cit. on pp. xi, 4, 23).

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. "Translation Validation". In: *TACAS '98*. Vol. 1384. LNCS. 1998, pp. 151–166 (cit. on pp. x, 3).

[RBL06] Thomas W. Reps, Gogul Balakrishnan, and Junghee Lim. "Intermediate representation recovery from low-level code". In: *PEPM*. ACM, 2006, pp. 100–111 (cit. on pp. ix, 3, 106).

[Riv05a] Xavier Rival. "Abstract dependences for alarm diagnosis". In: *Programming Languages and Systems*. Springer, 2005, pp. 347–363 (cit. on pp. 113, 117).

[Riv05b] Xavier Rival. "Understanding the origin of alarms in Astrée". In: *Static Analysis*. Springer, 2005, pp. 303–319 (cit. on pp. 113, 117).

[RL12] Valentin Robert and Xavier Leroy. "A formally-verified alias analysis". In: *Certified Proofs and Programs* (*CPP*). Vol. 7679. LNCS. Springer, 2012, pp. 11–26 (cit. on pp. 25, 46, 72).

[RM07] Xavier Rival and Laurent Mauborgne. "The trace partitioning abstract domain". In: *ACM Trans. Program. Lang. Syst.* 29.5 (2007) (cit. on p. 71).

[Rust] *The Rust Programming Language*. URL: https://www.rust-lang.org/ (cit. on pp. viii, 2).

[SK07] A. Simon and A. King. "Taming the Wrapping of Integer Arithmetic". In: *Proc. of SAS 2007*. Vol. 4634. LNCS. Springer, 2007, pp. 121–136 (cit. on pp. 45, 106).

[SO08] Matthieu Sozeau and Nicolas Oury. "First-class type classes". In: *Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 278–293. URL: http://link.springer.com/chapter/10.1007/978-3-540-71067-7_23 (cit. on p. 11).

[SW11] Bas Spitters and Eelis van der Weegen. "Type classes for mathematics in type theory". In: *Mathematical Structures in Computer Science* 21.4 (2011), pp. 795–825 (cit. on p. 11).

[Szo05] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005. ISBN: 0321304543 (cit. on p. 47).

[VN11] V. Vafeiadis and F. Zappa Nardelli. "Verifying fence elimination optimisations". In: *Proc. of SAS'11*. Springer, 2011, pp. 146–162 (cit. on pp. xi, 4, 25, 46).

[Web] *Companion website*. URL: http://people.irisa.fr/Vincent.Laporte/phd.html. Subject to change (cit. on pp. xii, 5, 7, 64, 68, 69, 110).

[WWM07] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. "Array bounds check elimination for the Java HotSpot™ client compiler". In: *PPPJ*. ACM Press, 2007, p. 125 (cit. on pp. viii, 1).