Formally verified compilation of low-level C code

Pierre Wilke

ANNÉE 2016

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Pierre Wilke

préparée à l'unité de recherche 6074 - IRISA
Institut de recherche en informatique et systèmes aléatoires
UFR Informatique Électronique (ISTIC)

Formally Verified Compilation of Low-Level C code

Thèse soutenue à Rennes le 9 novembre 2016

devant le jury composé de :

Julia LAWALL
Directrice de recherche - Inria / rapporteur
Frank PIESSENS
Professeur - KU Leuven / rapporteur
Xavier LEROY
Directeur de recherche - Inria / examinateur
Isabelle PUAUT
Professeur - Université Rennes 1 / examinatrice
Boris YAKOBOWSKI
Ingénieur - CEA List / examinateur
Frédéric BESSON
Chargé de recherche - Inria / examinateur
Sandrine BLAZY
Professeur - Université Rennes 1 / directrice de thèse

Remerciements

Je tiens à remercier Julia Lawall et Frank Piessens d'avoir accepté de rapporter ma thèse et pour l'intérêt qu'ils ont porté à mes travaux. Je remercie également Xavier Leroy et Boris Yakobowksi d'avoir accepté d'être examinateurs à ma soutenance. Un merci plus particulier à Isabelle Puaut qui a accepté de présider mon jury de thèse.

Je n'aurais pas pu effectuer ces travaux de thèse sans ma directrice de thèse, Sandrine Blazy, avec qui j'ai pu effectuer de nombreux voyages à l'autre bout du monde: Singapour, Nanjing, Nancy... Cette thèse n'aurait pas vu le jour non plus sans mon encadrant de thèse, Frédéric Besson, à qui je dois beaucoup pour ses nombreux conseils techniques et ses conversations inspirantes. Merci à eux pour les remarques constructives qu'ils ont su apporté pour améliorer la qualité des travaux décrits dans cette thèse et la qualité des articles que nous avons écrits ensemble.

Une thèse n'est pas limitée aux travaux décrits dans un manuscrit, c'est aussi une expérience humaine enrichissante au sein d'une équipe; et l'équipe Celtique est formidable. Pour ces moments extraprofessionnels entre doctorants, je remercie Martin Bodin, Pauline Bolignano, Gurvan Cabon, Alexandre Dang, Yon Fernandez de Retana, Vincent Laporte, Petar Maksimović, André Maroneze, Florent Saudel, Alix Trieu et Yannick Zakowski. Je remercie également les autres membres de l'équipe pour leur convivialité et leur bienveillance: David Cachera, Delphine Demange, Thomas Genet, Laurent Guillo, Thomas Jensen, David Pichardie et Alan Schmitt. Merci aussi à Lydie Mabil pour son aide précieuse pour mes différentes démarches administratives.

Je terminerai par remercier mes parents et ma famille qui m'ont soutenu tout au long de cette thèse, et dont certains ont été présents le jour de ma soutenance. Merci enfin à Nadia, avec qui je partage ma vie depuis bientôt six ans et qui m'a soutenu tout au long de ce doctorat. Je lui dédie ce manuscrit.

Résumé étendu en français

De plus en plus, notre vie quotidienne est régie par des programmes informatiques. Que ce soit pour des applications de divertissement, des véhicules autonomes ou le système de contrôle de vol des avions, les logiciels sont omniprésents. Les logiciels comportent des erreurs (des *bogues* informatiques) dont les conséquences peuvent varier d'une simple nuisance bénigne – dans le cas de jeux vidéos par exemple – à de graves conséquences humaines, écologiques ou financières – dans le cas de systèmes plus *critiques*.

Pour ces systèmes critiques, l'utilisation de méthodes formelles est de plus en plus commune. Les méthodes formelles sont des techniques, reposant sur des fondations mathématiques, qui visent à vérifier qu'un programme vérifie sa spécification, *i.e.* il se comporte comme on s'y attend. La *vérification formelle* consiste en l'application mécanisée des méthodes formelles, c'est-à-dire que le raisonnement effectuée est vérifié par un programme, que l'on appelle un *assistant à la preuve*, ce qui permet d'atteindre un haut niveau de confiance dans le résultat obtenu.

Pour raisonner sur des programmes, les méthodes formelles se basent sur une *sémantique formelle* du langage de programmation considéré. La sémantique d'un langage décrit le comportement de n'importe quel programme écrit dans ce langage. La plupart du temps, cette sémantique est informelle et contient des ambigüités – inhérentes au langage naturel dans lequel la sémantique est spécifiée. En revanche, les méthodes formelles s'appuient sur des sémantiques formelles, c'est-à-dire des objets qui définissent avec rigueur et précision le comportement des programmes du langage considéré, sans ambigüité.

Les méthodes formelles sont traditionnellement appliquées au code source d'un programme (écrit en C par exemple). La garantie formelle est donc établie vis-à-vis de la sémantique du language source. Cependant, c'est une garantie concernant l'exécution du programme compilé (en assembleur ou en langage machine) qui nous intéresse en fin de compte. Plutôt que d'analyser directement le programme compilé (ce qui est compliqué, puisque beaucoup d'abstraction a été perdue), la solution que nous considérons est la compilation formellement vérifiée.

Un compilateur formellement vérifié produit, à partir d'un programme source, non seulement un programme compilé mais également une garantie formelle que les programmes source et compilé se comportent de manière identique. On appelle cette garantie formelle le *théorème de préservation sémantique*. Une manière d'interpréter ce théorème est la suivante: "Le compilateur n'introduit pas de bogues."

La notion de comportement est primordiale dans l'énoncé du théorème de préservation sémantique. Un comportement de programme est soit un *comportement défini* – qui peut lui même représenter la terminaison d'un programme avec une valeur v ou la divergence d'un programme qui rentre en boucle infinie – soit un *comportement indéfini* – pour les programmes qui comportent des instructions illégales, par exemple une division par zéro. On dit qu'un programme est *sûr* si tous ses comportements sont *définis*. Le théorème de préservation sémantique peut alors être énoncé plus précisément: si le programme source $\mathcal{S}$

est sûr et si le compilateur réussit à générer un programme compilé $\mathcal{C}$ alors $\mathcal{T}$ se comporte comme $\mathcal{S}$.

L'hypothèse selon laquelle le programme source est sûr est primordiale pour le théorème de préservation sémantique. En effet, étant donné un programme comportant des comportements indéfinis, il est permis qu'un compilateur réalise des optimisations et produise un programme ne comportant que des comportements définis. Sans l'hypothèse de sûreté du programme source, le compilateur ne serait pas autorisé à réaliser l'optimisation qui élimine les comportements indéfinis. Pour aboutir à un résultat formel complet, il est donc nécessaire de prouver séparément la sûreté du programme source. Pour ce faire, il est possible d'utiliser les méthodes formelles pour prouver qu'un programme est sûr: par exemple Astrée [Bla+03], Frama-C [Kir+15] ou Verasco [Jou+15] sont des analyseurs statiques (un type particulier de méthodes formelles) dont le but est de prouver que des programmes C sont sûrs.

Dans cette thèse, nous nous intéressons au langage C. Le langage C a été introduit en 1972 comme le langage de développement du système d'exploitation Unix [JR78]. Depuis, C est utilisé pour le développement de tous types d'applications et est toujours parmi les langages les plus populaires aujourd'hui. La diversification des usages de C et des architectures sur lesquelles on exécutait les programmes a entraîné la nécessité de standardiser le langage. Le standard C [ISO99] décrit, de manière informelle, le comportement des programmes C.

CompCert [Ler09b] est un compilateur formellement vérifié, utilisé dans l'industrie, du langage C vers les langages assembleurs des plateformes x86, PowerPC et ARM. Le compilateur est entièrement spécifié, implémenté et prouvé à l'aide de l'assistant à la preuve Coq. Cela signifie que des sémantiques formelles ont été écrites pour le langage C, pour chacun des langages assembleur des diverses architectures, ainsi que pour les 8 langages intermédiaires utilisés dans CompCert. On appelle chaque transformation de programme d'un langage vers le suivant une *passe de compilation*. Le compilateur est défini comme la composition de toutes les passes de compilation. De manière analogue, chaque passe de compilation est prouvée correcte indépendemment des autres, puis le théorème global de préservation sémantique est obtenue par la composition des théorèmes associés à chaque passe de compilation.

Le théorème de préservation sémnatique de CompCert est soumis à l'hypothèse de sûreté: les programmes source ne doivent pas entraîner de comportement indéfini. Le standard C, pour des raisons de performance et de portabilité, utilise plusieurs notions de *sous-spécification*, qui se divisent en trois catégories:

- les *comportements non-spécifiés* sont des comportements pour lesquels le standard propose un certain nombre d'alternatives parmi lesquelles une implémentation du langage est libre de choisir pour chaque occurence du comportement;
- les *comportements définis par l'implémentation* sont des comportements non-spécifiés pour lesquels l'implémentation doit documenter ses choix;
- les *comportements indéfinis* sont des comportements pour lesquels le standard C n'impose rien: un compilateur est alors libre de générer – ou pas – du code exécutable, d'ignorer l'instruction responsable du comportement indéfini, ou de générer n'importe quel code.

Les comportements indéfinis sont nombreux: plus de 200 cas sont recensés dans l'annexe J.2 du standard [ISO99]. Les opérations qui provoquent des comportements indéfinis incluent, sans surprise, les divisions par zéro ou les accès mémoire *via* un pointer nul.

En revanche, pour certains comportements indéfinis, il existe des sémantiques raisonnables auxquelles on peut penser: ce sont ces comportements qui vont nous intéresser pour la suite. Par exemple, le dépassement d'entier signé ou le décalage bit-à-bit d'une trop grande quantité sont des comportements indéfinis. On peut penser que des comportements définis par l'implémentation seraient plus adaptés pour ces cas.

Dans cette thèse, nous nous intéressons en particulier à deux comportements indéfinis que l'on retrouve dans des programmes importants, par exemple le noyau Linux ou l'implémentation de la librairie standard de FreeBSD. Ces comportements sont provoqués par des opérations sur la représentation binaire des pointeurs (*e.g.* opérateurs bit-à-bit, arithmétique arbitraire) et la manipulation de données non-initialisées.

Nous examinons ces programmes issus de projets *open source* et définissons la sémantique de ces opérations (arithmétique arbitraire et opérateurs bit-à-bit sur des pointeurs et manipulation de données non initialisées) en conséquence. Notre but est de formaliser le modèle informel que les programmeurs C ont en tête lorsqu'ils écrivent du code de bas-niveau. Cela peut se résumer en deux idées principales:

1. les pointeurs sont des entiers, qui satisfont un certain nombre de propriétés (notamment d'alignement), et les opérations sur les pointeurs ne sont rien d'autre que des opérations sur les entiers qui les représentent; et

2. les données non-initialisées peuvent être manipulées et la lecture de telles données résulte en une valeur arbitraire, mais stable: deux lectures successives donneront le même résultat.

Bien que ces hypothèses soient contraires aux opinions du comité du standard C, nous pensons qu'elles correspondent au modèle mental des utilisateurs de C. De plus, il est classique pour un compilateur de faire des choix sémantiques, c'est-à-dire de rendre des comportements indéfinis plus définis. Par exemple, le dépassement d'entier signé est un comportement indéfini en C, mais CompCert lui donne une sémantique définie telle que `INT_MAX + 1 == INT_MIN`.

Le but de cette thèse est d'adapter CompCert avec une sémantique de C plus définie, *i.e.* une sémantique qui permet des opérations arbitraires sur des pointeurs et des données non-initialisées. En conséquence le théorème de préservation sémantique *s'appliquera* plus souvent, puisque davantage de programmes auront une sémantique définie et seront *sûrs*. Nous modifions le moins possible le code du compilateur CompCert. En revanche, nous modifions le modèle mémoire sur lequel repose CompCert en profondeur, nous adaptons les sémantiques formelles utilisées par tous les langages intermédiaires de CompCert et, bien sûr, nous adaptons les preuves de préservation sémantique de chacune des passes de compilation.

Les contributions de cette thèse sont les suivantes.

Valeurs symboliques. Nous définissons un domaine de *valeurs symboliques* [BBW14], qui modélisent les résultats d'opérations (par exemple des opérations bit-à-bit sur des pointeurs) qui seraient indéfinis, que ce soit pour le standard C ou la sémantique existante de CompCert.

Modèle mémoire de bas-niveau. Nous définissons un modèle mémoire [BBW15] de bas-niveau, qui repose à la fois sur le modèle mémoire plus *abstrait* de CompCert ainsi que sur les valeurs symboliques. Le modèle mémoire est un composant sémantique, utilisé dans les sémantiques formelles de tous les langages de CompCert, du C jusqu'à l'assembleur.

Il définit comment la mémoire est organisée, les valeurs qui y sont stockées, ainsi que des opérations de base sur la mémoire (écriture, lecture, allocation et libération). Le modèle mémoire est également équipé de propriétés de bonne formation qui permettent de raisonner sur les états mémoire.

La particularité de notre modèle mémoire est sa capacité à capturer les constructions de bas-niveau que le modèle mémoire de CompCert ne peut pas modéliser.

Une autre différence entre notre modèle mémoire et celui de CompCert est que nous modélisons une mémoire finie, contrairement à CompCert, qui modélise une mémoire infinie, dans laquelle l'allocation d'une nouvelle région de mémoire ne peut jamais échouer.

Sémantiques formelles symboliques. En utilisant ce modèle mémoire de bas-niveau, nous adaptons les sémantiques formelles de tous les langages, depuis le C jusqu'à l'assembleur de l'architecture x86. Nous appelons ces sémantiques *symboliques* puisque les valeurs qu'elles manipulent sont des valeurs symboliques. Nous montrons que les sémantiques symboliques sont des raffinements des symboliques existantes dans CompCert, c'est-à-dire que tous les comportement capturés par les sémantiques de CompCert sont également capturées par nos sémantiques symboliques. Nos sémantiques associent par ailleurs des comportements définis à des programmes auxquels les sémantiques de CompCert associaient des comportements indéfinis.

Transformations de la mémoire. Afin de prouver les théorèmes de préservation sémantique de chacune des passes de CompCert, des notions génériques de transformations de la mémoire sont définies. En particulier, les *injections mémoire* sont des transformations qui modifient l'agencement de zones de mémoire. Nous réutilisons et généralisons ces notions à notre modèle mémoire de bas-niveau.

Passes de compilation et Théorème de préservation sémantique. Les passes de compilation de CompCert sont réutilisées telles quelles. En effet, ces passes sont des transformations de la syntaxe du programme et sont indépendantes de la sémantique des langages. Évidemment, lorsque les passes de compilation se basent sur des analyses de code (par exemple pour les optimisations), ces analyses sont dépendentes de la sémantique et doivent être adaptées à notre modèle. Néanmoins, les preuves de préservation sémantique de chacune des passes de compilation doivent, elles, être adaptées en utilisant nos généralisations des transformations de la mémoire. On appelle CompCertS le compilateur résultant de toutes ces modifications: un domaine de valeurs symboliques, un modèle mémoire de bas-niveau, des sémantiques symboliques plus *permissives* et des preuves de préservation sémantique adaptées.

Le théorème de préservation sémantique de CompCertS est plus fort que celui de CompCert pour deux raisons. Premièrement, parce qu'il s'applique à plus de programmes puisque plus de programmes sont *sûrs*. Ensuite, puisque notre modèle mémoire est fini, nous sommes en mesure de quantifier la consommation mémoire des programmes C. Nous apportons en particulier la garantie suivante: si un programme C est sûr et se compile sans erreurs vers un programme assembleur, alors non seulement les programmes C et assembleurs se comportent de manière identique, mais aussi le programme assembleur utilise moins de mémoire que le programme C.

Notes sur le développement Coq associé. Sauf explicitement indiqué dans ce document, l'intégralité des théorèmes ont été prouvés grâce à l'assistant à la preuve Coq. Le développement est accessible en ligne[1]. Des liens vers le développement sont indiqués à l'aide du logo Coq: .

Dans ces travaux, nous nous concentrons sur l'architecture x86. Toutefois, nous ne voyons pas d'obstacle à adapter nos travaux aux autres architectures cibles de CompCert, à savoir PowerPC et ARM. De plus, les parties de CompCert dépendentes de l'architecture sont relativement localisées.

[1]`http://www.irisa.fr/celtique/wilke/phd/index.html`

Contents

Chapter 1

Introduction

Software systems are pervasive. From benign uses such as entertainment and web browsing to more involved cases such as self-driving cars and airplane flight control systems, our daily lives are becoming more and more governed by software systems. Errors in such software are found regularly and may have dramatic consequences. Errors in benign systems such as video games generally lead to minor annoyances; however errors in *critical* software systems may lead to disastrous consequences, either humanly, ecologically or financially. For instance, in 2016, the Japanese satellite Hitomi was lost, after reacting to inaccurate sensor data: while trying to counterbalance a detected but inexisting rotation movement, the satellite lost communications. In 2008, Halperin *et al.* [Hal+08] show that pacemakers are vulnerable to denial-of-service attacks, *i.e.* one could prevent the device from functioning. From 2002 to 2009, Toyota vehicles have suffered from bugs that provoked unintended and uncontrollable acceleration of the vehicles, resulting in multiple fatal accidents [Sam14]. More recently, in April 2014, the Heartbleed bug [Dur+14] was discovered in the OpenSSL cryptography library. The bug is a buffer overflow which leaks cryptographic private keys, therefore annihilating the confidentiality of the communications.

Because the consequences of software errors can be dramatic, the need rises for the use of *formal methods*. Formal methods are a set of techniques, based on mathematical and rigorous foundations, whose aim is to verify that such *critical* programs are safe, *i.e.* that their execution never results in run-time errors. We call *formal verification* the *mechanised* application of formal methods, whereby the mathematical rigor required for the use of formal methods is verified by a computer program, called a *proof assistant*. This gives a high level of confidence, because one does not need to check the entirety of all the reasoning steps, but merely trust that the proof assistant is correct.

To reason about programs, formal methods need *formal semantics*. The semantics of a programming language answers the question of assigning meanings to programs. In other words, it describes the behaviour of every program written in that language. In most cases, the semantics of programming languages is given by informal specifications in natural language prose – with its load of imprecisions and ambiguities. Formal methods need formal semantics, *i.e.* an object that describes, with mathematical rigor and without ambiguities, the behaviour of every program written in that language.

Formal methods are usually applied to the source code of programs. They therefore give formal guarantees about the behaviour of programs according to the formal semantics of the source language. Static analysers, for instance, are formal methods tools whose aim is to prove that a given property is satisfied by every execution of the input program, as specified by the formal semantics of the source language. In general, it is undecidable whether a property holds for every execution of a program (see Rice's theorem [Ric53]).

Static analysers circumvent this issue by computing over-approximations of the possible behaviours of a program. It is always sound to compute over-approximations because every behaviour of the actual system is captured by the approximation. Hence, if the over-approximation does not contain undesirable behaviours, then neither does the set of behaviours of the actual program.

Successful examples of the application of formal methods to industrial contexts include the Astrée static analyser [Bla+03; DS07], which has been used in particular to prove the absence of run-time errors in the primary flight control software of the Airbus fly-by-wire systems since the A340 airliner. Frama-C [Kir+15] is another static analysis framework which provides various analyses for C programs. Verasco [Jou+15] is yet another static analyser, that aims at proving the absence of run-time errors in C programs, whose distinguishing feature is that it is entirely specified, implemented and proved in the Coq proof assistant.

All those examples yield a guarantee about C programs. However, what we really value is a guarantee about the behaviour of the program being actually run on our machine, *i.e.* after it has been compiled to machine code. Performing program analysis at this level is an option [BR10; KV08], sometimes even the only option when the source code of programs is not available. However, it is much harder than at the source level. Indeed, most abstraction (be it *code abstraction* with the high-level notions of functions and loops, or *data abstraction* such as types, variables or `struct` constructs) has been lost, making reasoning harder.

To achieve the formal guarantee over the assembly program, while keeping the analysis at the source level, the idea is to verify that the compilation preserves the properties proved at the source level. Formally verified compilers fill the gap between the result of a formal verification on a source program and the guarantee we expect on the running program. A verified compiler provides, in addition to a compiled program, a formal proof that the compiled program behaves as the source program. This is known as the *semantic preservation theorem*. An intuitive reading of this theorem is: "The compiler does not introduce bugs." It also follows that any safety property that was proved by static analysis for the source program still holds for the compiled program.

In order to introduce the topic of this thesis, we need to give more information about the semantic preservation theorem. The notion of program behaviour is central to the semantic preservation theorem. A program behaviour is one of the two following: either a *defined behaviour*, which can be *termination* (the program terminates with a return value v) or *divergence* (the program loops forever), or an *undefined behaviour* (for programs that perform illegal operations, *e.g.* division by zero). We say that a program is *safe* if all its behaviours are defined. The semantic preservation theorem can be stated as follows: if the source program $\mathcal{S}$ is *safe* and the compiler generates a target program $\mathcal{T}$, then $\mathcal{T}$ behaves as $\mathcal{S}$.

We need the hypothesis that the program is safe in the semantic preservation theorem, because compilers routinely optimise away pieces of code that may exhibit undefined behaviours. For example, consider a program that assigns to a variable `x` the result of dividing by zero, and then never uses `x` again. This program has undefined behaviour because of the illegal division by zero. However, since the assignment is never used, it may be removed by a dead code elimination optimisation, resulting in a program that does not have undefined behaviours. For that reason, semantic preservation theorems generally exclude programs that have undefined behaviours.

The important hypothesis that the source program must be safe can be discharged by running a static analyser (*e.g.* Astrée, Frama-C, Verasco) on the source program.

In this thesis, we will focus on programs written in C. The C language has been introduced in 1972 as the development language of the Unix operating system [JR78]. Since then, C has been used as a general purpose programming language, and is still widely used nowadays. It soon became important that the C language be portable so that C programs could be run on different platforms. This need for portability led to the development of specifications for the C language, starting from the K&R book [KR78], to more formal standardisations of C by ANSI [ANS89] (*American National Standards Institute*) or ISO [ISO99; ISO11] (*International Organization for Standardization*). Those documents explain, informally, the behaviour of every program written in C.

CompCert [Ler09b] is a formally-verified, industrial-strength compiler for a large subset of the C language down to the assembly languages for the x86, PowerPC and ARM architectures. The compiler is fully specified, implemented and proved using the Coq proof assistant. More precisely, the C language, the assembly languages for each target architecture and the 9 intermediate languages are defined and given formal semantics in Coq. Each program transformation from one language to another (lower-level) language is called a *compiler pass*, and the whole compiler is the composition of all those compiler passes. Similarly, each compiler pass is proved to be *semantics preserving* independently and the final semantic preservation theorem of the whole compiler is the composition of the semantic preservation of the individual compiler passes.

CompCert provides unprecedented confidence in a C compiler. As an illustration of this confidence, Yang *et al.* [Yan+11], the authors of the Csmith tool – that generates random C programs and tests compilers (11 of them, including GCC and LLVM) – report:

> The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

CompCert's semantic preservation theorem only holds for safe programs, *i.e.* those that do not exhibit undefined behaviour. The C language, for performance and portability reasons, uses several notions of *under-specification*, *i.e.* some behaviours are not, or only partially, given semantics. These behaviours fall in three different categories:

- *unspecified behaviours* are behaviours for which the standard gives a number of alternatives, from which the implementation is free to choose any for any occurrence;
- *implementation-defined behaviours* are unspecified behaviours for which the implementation must document how the choice between alternatives is made in every particular situation;
- *undefined behaviours* are behaviours for which the standard imposes **no requirements** on the implementation: it may or may not generate executable code; the offending situation may be silently ignored or compiled into *any* piece of code.

Undefined behaviour happens in a wide variety of cases: around 200 cases are listed in Appendix J.2 of the C standard [ISO99]. This list includes `NULL` pointer dereferences and out-of-bounds array accesses, for which no reasonable semantics comes to mind: it is therefore natural to make such behaviours undefined. However, the list also includes signed integer overflows or over-sized shifts for which reasonable semantics can be thought of, such as considering the shift amount modulo 32 or discarding the higher bits. One might argue that an *unspecified behaviour* or *implementation-defined behaviour* would have been more appropriate for this case. As John Regehr puts it,[1] one might suspect "that the C standard body simply got used to throwing behaviours into the "undefined" bucket and got a little carried away".

There are other kinds of behaviours that, although being undefined, are nevertheless used in production code. For example, bitwise operations are performed on pointers with the mental assumption that pointers can be treated as integers. Another misconception that programmers have is related to uninitialised data: reading the value of an uninitialised variable does not result in some *arbitrary* value – as one could believe – but is undefined behaviour. As we will see in this document, examples of such misunderstandings of the C standard happen in system code, mainly for performance reasons, and rely on assumptions that are not shared by compilers, and therefore the compiler does not necessarily preserve the semantics of these programs.

Contributions In this thesis, we want to reconcile the programmers' *informal* mental model of the memory in the C language with the formal semantics needed for both static analyses and verified compilers correctness. In particular, we formalise the assumption that pointers can be casted to and from integers and their binary representation can be manipulated as standard integers. We also formalise accesses to uninitialised data as reading an arbitrary but stable value, meaning that reading an uninitialised value twice results in the same arbitrary value. This is closely related to Defect Report #260 [ISO], which demands clarification to the C standard committee on the two following questions:

1. if an object holds an indeterminate value, can that value change other than by an explicit action of the program?

2. if two objects hold identical representations derived from different sources, can they be used exchangeably?

Regarding Question 1 about indeterminate values, the standard committee answers that the result of accessing indeterminate values may change even without a direct action of the program. Our answer to the same question is opposite to that of the standard committee, namely that the value may be arbitrary but must be stable. We advocate that no reasonable architecture would modify the bit-pattern representation of indeterminate values.

Question 2 can be slightly rephrased into a more specific question about pointers: *is a pointer anything more than its bit-pattern representation?* The answer of the standard committee states that two pointers may be treated differently based on their *origin*, or *provenance*, even if they have the same binary representation. Once again, we give a different answer to that question. Our motto is that pointers can be treated as integers, in a way that will be described later in this thesis. Hence, we consider two pointers with the same bit-pattern as equal and interchangeable.

The answers we give are opposite to those of the standard committee, however we believe that they capture the mental model that programmers have in mind when writing C code that can be found in real-life projects, as we will show in Chapter 3.

This work builds upon the CompCert compiler, which is the only formally-verified compiler for C. We aim at preserving as much as possible of the compiler passes provided by CompCert. We only change the formal semantics of the languages of CompCert, therefore making more programs have defined semantics. As a result, more programs are

[1]http://blog.regehr.org/archives/213

covered by the semantic preservation theorem. This necessitates to adapt the semantic preservation theorems of the individual passes in consequence. Our contributions can be stated as follows.

- We define a formalism of *symbolic values* [BBW14] that denote the result of operations (*e.g.* bitwise manipulation of pointers, computation on uninitialised data) that would otherwise be undefined, according to both the C standard and the formal semantics of CompCert. This formalism is the basis for reasoning about C expressions and programs that perform bit-level manipulation of data.

- We define a *low-level* memory model for C [BBW15], based on the abstract memory model of CompCert and our formalism of symbolic values. This memory model features in particular a finite memory space (contrasting with CompCert's unbounded memory). We also reprove the *good-variable properties* on our memory model – those are well-behavedness properties of the memory (*i.e.* reading just after writing at the same location results in the value that was just written).

- We adapt the formal semantics of all the languages of the CompCert development: the C language, the assembly language for the x86 architecture, and all the intermediate languages in between, that will be introduced in Section 2.5.1. We call the resulting formal semantics *symbolic semantics* because the values they operate on are symbolic values. We show that the symbolic semantics are a refinement of CompCert's semantics.

- The generic notions of memory transformations (extensions and injections), introduced in Section 2.5.3, formally describe how memory states are transformed by the CompCert compiler. We generalise these notions to our low-level memory model.

- The compiler passes of CompCert hardly need to be modified. However the correctness proofs of those passes need to be reworked. We adapt the proofs to our symbolic semantics. This results in CompCertS (S stands for Symbolic), our modified version of CompCert equipped with our low-level memory model, our symbolic semantics and proved correct with our generalisations of memory transformations. This compiler gives the formal guarantee that the compiled program behaves as the source program, but also that the compiled program uses no more memory than the source program.

In this work, we focus on the x86 back-end of CompCert to build CompCertS. The parts of CompCert's development that are specific to the target architecture are relatively small and we foresee no obstacle for adapting our contributions to PowerPC and ARM architectures. Unless explicitly stated otherwise, all the material described in this thesis has been formally specified, implemented and proved using the Coq proof assistant. The entire development is available online.[2] The electronic version of this document includes links to the development, showing the corresponding functions or theorems. Those are signaled by a Coq logo: .

Outline The remainder of this thesis can be split into three parts. The first part (Chapters 1 to 3) includes this introduction, general information about the context of this work and motivating examples. The second part (Chapters 4 to 6) defines the formalism we use

[2]`http://www.irisa.fr/celtique/wilke/phd/index.html`

to create our low-level memory model and symbolic semantics. The third part (Chapters 7 and 8) is dedicated to the proof of correctness of CompCertS. Chapter 9 concludes this thesis.

General information regarding the C standard and formally-verified compilation is provided in Chapter 2. A particular focus is made on CompCert, an industrial-strength formally verified compiler for C, upon which this work builds. We explain the memory model that CompCert uses, the overall architecture of the compilation chain of CompCert. We also give background regarding the proof techniques used for the correctness proof of the individual compiler passes. Finally, we give the statement of CompCert's semantic preservation theorem.

Next, Chapter 3 exhibits a number of motivating examples of C programs that come from major open source pieces of software and that CompCert's formal guarantees do not apply to, because those programs have *undefined behaviour*. We give an intuitive explanation of how the example programs *should behave* in the programmers' mental model. Our aim throughout this thesis will be to build a formal semantics for C that assigns a meaning to these widely-used low-level idioms.

The remaining chapters are the bulk of our work, and aim at defining the semantics of those programs and providing formal guarantees about their compilation. We follow a bottom-up approach: we build CompCertS from its heart – the domain of symbolic values – to the full theorem of compiler correctness.

Chapter 4 introduces the domain of symbolic values and the notion of *normalisation*. Symbolic values are used to represent the result of computations that would otherwise be undefined. The normalisation aims at simplifying symbolic values into values. This process is based on the definition of *concrete memories*, which are all the possible concrete layouts of the memory.

Then, Chapter 5 pushes the formalism of symbolic values and normalisations into CompCert's memory model, resulting in a low-level, more permissive albeit finite, memory model. We show that the *good-variable properties* – well-behavedness properties on the primitive operations of the memory model – still hold in our low-level memory model. We also show how we cope with a finite memory, *i.e.* how we decide whether there is enough available space.

Next, in Chapter 6, we show how this low-level memory model is used in the formal semantics of all the intermediate languages of CompCert. We call the resulting semantics *symbolic semantics*. We also show how to get an executable version of our symbolic semantics of the C language, enabling us to execute C programs with our symbolic semantics. The challenge resides in the implementation of the normalisation – we use an SMT solver for this purpose. We demonstrate the usefulness of our symbolic semantics by running our C semantics on a set of programs, including examples from Chapter 3.

The remaining chapters concentrate on the adaptation of the proof of correctness of the CompCert compiler to our symbolic setting, resulting in the CompCertS compiler.

Chapter 7 defines the generalisations of generic memory relations (extensions and injections) used as invariants in CompCert. We show how to reprove the existing theorems of CompCert and we introduce new theorems linking *e.g.* the normalisation to the memory relations. Those theorems are the building blocks of the semantic preservation proofs of the individual compiler passes, and therefore of the whole CompCertS compiler.

Finally, Chapter 8 builds on top of every definition and theorem we introduced in earlier chapters and builds the correctness proof of all the individual compiler passes. It specifically reports on four passes that require more work than the others to port CompCert to CompCertS. The difficulties we face are mainly due to the finiteness of our low-level memory model. The final result, CompCertS, is a formally verified compiler for the C language equipped with the semantics that we believe to be commonly-assumed by programmers. CompCertS comes with an end-to-end correctness theorem that states that not only the semantics, but also the amount of memory used by programs is preserved by compilation.

Chapter 9 concludes this thesis. It summarises the results we achieved and provides ideas for future work, improvements and applications of this low-level compiler, CompCertS, particularly to security issues.

Chapter 2

Background

2.1 The C Standard And Underspecified Behaviours

The C standard [ISO99; ISO11] is the official documentation of the C language. It describes the behaviour of every program by specifying an *abstract machine*. The statements and expressions of C are then defined in terms of interactions with the state of this machine. The first document used as a reference for the C language is the book by Kernighan and Ritchie [KR78], which explains informally the concepts of C. It is only in 1983, ten years after the introduction of C, that the first committee for a C standard was formed. The first version of the standard was published in 1989 by ANSI (*American National Standards Institute*) and is known as C89. ISO (*International Organization for Standardization*) then adopted the standard and reworked it several times, including new features such as the `long long` type, variable-length arrays and better support for floating-point numbers in C99 and support for concurrency in C11.

This document is important because it defines a contract between compiler writers and programmers. Compiler writers are *required* to generate executable code that *behaves* as prescribed by the source code that the programmer gave as input, under the condition that the input program is *well-defined according to the C standard*.

To understand precisely this condition, we must introduce a few notions used in the C standard about the *behaviour* of programs. The C standard does not associate a precise behaviour for every possible C program. Indeed, some programs are given underspecified behaviours. Those underspecified behaviours can be either *unspecified*, *implementation-defined* or *undefined* behaviours.

An *unspecified* behaviour is a behaviour for which each implementation (*i.e.* compiler or interpreter) may choose among a list of possibilities and is free to change its choice for every occurrence of the behaviour. For example, the order in which the arguments to a function are evaluated is unspecified behaviour (left-to-right or right-to-left).

An *implementation-defined* behaviour is a special case of unspecified behaviour for which the implementation is required to document how the choice between alternatives is made. For example, the representation of signed integers (using sign and magnitude, two's complement or one's complement) is implementation-defined. The `gcc` compiler and Microsoft Visual Studio document that they use only two's complement. The `clang` compiler does not seem to document its choices[1], however they seem to follow the same choices as `gcc`.

[1]The bug report `https://llvm.org/bugs/show_bug.cgi?id=11272` is still open after more than 5 years, at the time of writing. Recently, Richard Smith ironises that being an implementation, the implementation-defined behaviours of `clang` are defined by the implementation.

An *undefined* behaviour is a behaviour for which the standard imposes no requirements. The implementation may abort processing the input program, silently ignore the problem or generate arbitrary code. The C standard makes no distinction of severity between the following selected undefined behaviours:

- dereference a `NULL` or dangling pointer;
- division by zero;
- signed integer overflow;
- sequence point violations;
- bitwise pointer arithmetic;
- access to uninitialised data.

Indeed, C programmers will probably not expect their program to have well-defined semantics when it dereferences the `NULL` pointer, or when a division by zero occurs. It is commonly understood that these behaviours are undefined.

However, regarding signed integer overflow, most programmers will assume that it wraps around modulo 2^{32} (in a 32-bit architecture). Still, it is undefined behaviour and enables unexpected optimisations.[2] `gcc` actually provides an command-line option (`-fwrapv`) to force the wrap-around behaviour and disable this optimisation.

Similarly for sequence point violations, *e.g.* `(x = 1) + (x = 2)` is undefined because there is no sequence point between the two assignments to variable `x`. However, a programmer might expect the result of this expression to be 3, because the first assignment evaluates to 1 and the second assignment evaluates to 2, no matter in what order they are evaluated. This expression is actually transformed by `gcc` 4.9.2 (at all optimisation levels) into `x = 1; x = 2; x + x`, which doesn't match the programmer's expectations but is legal with respect to the C standard.

Pointers, on modern platforms, are merely integers that represent addresses. While the C standard forbids treating pointers as integers (*e.g.* bitwise operations are not allowed), one can be tempted to exploit this fact and perform arbitrary pointer comparisons or storing some information in spare bits of pointers (see Section 3.1).

Reading uninitialised data is sometimes thought of as a way of generating *randomness* (see examples in Section 3.2), although it is undefined behaviour. The compiler is free to generate any code, and not necessarily treat the uninitialised data as some arbitrary value, and as we will see in greater detail in Section 3.2, it uses this freedom and optimises away computations based on uninitialised data.

2.2 Formal Semantics

The C standard is a specification of the meaning of C programs in natural language, and is hence ambiguous and subject to interpretation. Formal verification relies on a formal, unambiguous specification of the meaning of programs, *i.e.* a mathematical object that reflects this informal specification. This formal specification is called a *formal semantics*.

Formal semantics come in various flavours, each of which has its own purpose. Denotational semantics [SS71] describes the meaning of programs using an abstract mathematical

[2]See `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=33498`. An overflow inside the body of a `for` loop makes the loop infinite.

model relying on partial orders, continuous functions and least fixpoints. Axiomatic semantics [Flo67] defines the meaning of programs by giving proof rules to reason about them. The canonical example of axiomatic semantics is Hoare logic [Hoa69], and it is used to prove, among other properties, that a program satisfies its specification. In this work, we focus on a third type of formal semantics, operational semantics [Plo81], where the meaning of a program is given by a transition system in an abstract machine. For every syntactic construct of the languages, transition rules dictate how the state of the abstract machine evolves when this construct is executed. Definition 2.2.1 formalises the notion of labelled transition system, which is the basis of operational semantics, as introduced in [Plo81].

Definition 2.2.1 (Labelled transition system (LTS))**.** *A labelled transition system is a tuple $(\Sigma, \mathcal{E}, I, F, \rightarrow)$ where Σ is a set of states (or configurations), $\mathcal{E}$ is a set of events including a silent event ϵ, $I \subseteq \Sigma$ is the set of initial states and $F \subseteq \Sigma$ is the set of final states. $\rightarrow \subseteq \Sigma \times \mathcal{E} \times \Sigma$ is the transition relation such that $(\sigma, e, \sigma') \in \rightarrow$ (written $\sigma \xrightarrow{e} \sigma'$) if and only if a transition can be fired from state σ to state σ', emitting event e.*

The events on the transitions enable to abstract from the low-level details of program states, which are dependent on the actual language we consider and its formal semantics. Events aim at being language-agnostic and capturing the behaviour of programs, at a higher level of abstraction. Definition 2.2.2 captures the notion of traces, which extract the sequences of events from all possible paths in a labelled transition system.

Definition 2.2.2 (Traces)**.** *A trace is a (possibly infinite) sequence of events. We write finite traces $e_0 e_1 \dots e_n$ and infinite traces $e_0 e_1 \dots$. The set of traces of a labelled transition system $S = (\Sigma, \mathcal{E}, I, F, \rightarrow)$ is written $Traces(S)$ and captures the sequences of events emitted by every possible derivation in S. Formally,*

$$\begin{array}{rcll} Traces(S) & = & \left\{ e_0 \dots e_{n-1} \mid \exists\, \sigma_0 \dots \sigma_n, \begin{pmatrix} & \sigma_0 \in I \\ \wedge & \forall i < n, \sigma_i \xrightarrow{e_i} \sigma_{i+1} \\ \wedge & \neg\exists \sigma,\ \sigma_n \rightarrow \sigma \end{pmatrix} \right\} & \textit{finite traces} \\ & \cup & \{ e_0 \dots \mid \exists\, \sigma_0 \dots\ \sigma_0 \in I \wedge \forall i, \sigma_i \xrightarrow{e_i} \sigma_{i+1} \} & \textit{infinite traces} \end{array}$$

Traces are either finite or infinite. Finite traces are either *terminating* traces (those that end with $\sigma_n \in F$) or *stuck* traces (those that end with $\sigma_n \notin F$). Note that those traces are maximal, *i.e.* they end with a state from which no step can be taken. Infinite traces model non-terminating executions and are said to be *diverging*. Definition 2.2.3 defines program behaviours on top of program traces.

Definition 2.2.3 (Program Behaviours)**.** *The set of behaviours of a program P, written $Beh(P)$, is the set of traces of the labelled transition system associated with P.*

Program behaviours can be split between stuck, going-wrong, behaviours and *normal* behaviours (either terminating or diverging).

Formal Semantics for C. Since the 1990's, a number of formal semantics have been given for C.

Gurevich and Huggins [GH92] describe the semantics of C using abstract state machines (ASMs). Their semantics is not executable and can therefore not be applied mechanically to C programs. This work has not been conducted inside a proof assistant.

Cook and Subramanian [CS94] define a semantics for the C language inside the Nqthm theorem prover. Their aim is to perform formal verification of functional correctness of C programs. Their semantics is written as an interpreter and is therefore executable. The semantics does not cover the entirety of C: it is restricted to a limited set of types and expression constructs.

In his thesis [Nor98], Norrish defines another formal semantics for C, with the purpose of verifying C programs. This semantics is written using the HOL theorem prover. His semantics is not executable but is aimed at proving properties of programs. A particular effort is made on capturing precisely all the possible evaluation orders *e.g.* for the arguments to function calls, which the previous semantics did not do accurately. Norrish also defines a Hoare logic for C programs in order to perform verification.

In the context of the CompCert compiler [Ler09b] (which will be introduced in greater detail in Section 2.5), Leroy *et al.* formalise a large subset of C inside the Coq proof assistant. While the goal of the previous formal semantics was to perform verification at the C level, the objective of CompCert is to verify that the compiler is correct, *i.e.* it preserves the behaviour of programs. Besides, they have developed an interpreter for C, that captures all the possible evaluation orders, and that can be applied to C programs to test their semantics. Unlike other semantics that intend to follow as closely as possible the C standard, CompCert takes the freedom (and is justified in doing so) of giving arbitrary (though reasonable) semantics to behaviours that are undefined or unspecified according to the standard, *e.g.* signed integer overflow.

Ellison and Roşu [ER12] define an executable semantics of C inside the $\mathbb{K}$ framework, based on rewriting systems. Their aim is to develop a practical tool, that can be used for finding bugs or observing sets of behaviours however it is not tailored for use inside a proof assistant. Hathhorn *et al.* [HER15] extend their work, and put emphasis on precisely distinguishing undefined and defined behaviours: they want to be able to identify precisely programs with undefined behaviours.

Kang *et al.* [Kan+15] propose a formal semantics in Coq of a C-like language that focuses on assigning semantics to pointer-to-integer casts. They do so by *realising* pointers only when needed, *i.e.* actually giving a concrete 32-bit address to a pointer only when it is cast to an integer. This model allows them to prove the soundness of several optimisations, which would not be valid if pointers were always realised, *i.e.* if they always had a concrete address.

Memarian *et al.* [Mem+16] present Cerberus, a *de facto* semantics for C, *i.e.* a semantics that captures not the ISO C standard, but the C as it is used in practice. To discover the *de facto* semantics of C, they have designed a set of questions regarding various unclear aspects of the C standard and received responses from hundreds of C programmers and members of the standard committee. Their formalisation consists of a translation from C to a core language. The core language is a typed call-by-value language of function definitions and expressions. It is parametric on the memory model to be used. Hence, one can plug in different memory models to obtain the various behaviours (possibly deviating from the standard) that are the *de facto* semantics of C.

In his thesis [Kre15], Krebbers formalises in Coq the C standard. His formalisation, CH_2O, consists of an operational semantics (used to reason about program transformations), an executable semantics (used to compute the set of behaviours of a given program) and an axiomatic semantics (used to reason about programs). This formalisation aims at being close the C11 standard [ISO11]. Like the work of Memarian *et al.*, CH_2O is based on a core language, into which C programs are translated prior to any reasoning.

2.3 Formally-Verified Compilation

A compiler translates a program from a source language $\mathcal{S}$ into a target language $\mathcal{T}$, where $\mathcal{T}$ is often a lower-level language than $\mathcal{S}$. For example, a C compiler typically generates assembly programs from C programs.

A formally-verified compiler is a compiler that provides formal guarantees about the code it generates. The purpose of this section is to investigate various compiler correctness properties, *i.e.* answer the question "What does it mean for a compiler to be correct?". Robert Dockins's thesis [Doc12] gives a detailed survey of several verified compilers and the correctness properties they claim. Informally, such a correctness property should be a trade-off between permissiveness (traditional program transformations should be allowed) and tightness (interesting program properties should be preserved).

A natural candidate for compiler correctness is bisimilarity, and can be stated as in Property 2.3.1.

Property 2.3.1 (Bisimilarity)**.** *Two programs $\mathcal{S}$ and $\mathcal{T}$ are bisimilar if both programs have the same set of behaviours,* i.e. *if $Beh(\mathcal{S}) = Beh(\mathcal{T})$.*

The bisimulation relation captures equivalent programs, *i.e.* bisimilar programs behave identically, and all properties of the behaviours of the programs are preserved. However it is too strong a property to be a criterion for compiler correctness.

A first reason why bisimulation is not appropriate in the case of compiler correctness is because not all behaviours of the source program need to be behaviours of the target program. To understand why behaviours of the source program should be allowed to be *forgotten*, consider a non-deterministic feature of the source language. For example, in C, the order of evaluation of the arguments to a function call is chosen non-deterministically. Consider the code snippet `f(g(),h())` where `g` and `h` are functions that may produce side-effects. The set of behaviours of this program will include different traces depending on whether `g` is executed before or after `h`. However, the compilation of this code might yield the code `int x = h(); int y = g(); x + y;` thus forcing the evaluation order and reducing the set of behaviours.

This results in Property 2.3.2, a relaxed property that is a candidate criterion for compiler correctness, and that is called *backward simulation*.

Property 2.3.2 (Backward simulation)**.** *All the behaviours of program $\mathcal{T}$ are included in the behaviours of program $\mathcal{S}$.*

$$Beh(\mathcal{T}) \subseteq Beh(\mathcal{S})$$

Only safety properties, *i.e.* the set of behaviours does not intersect with a predetermined set of undesirable behaviours, are preserved by this correctness property. However, it is better-suited to compilation than bisimilarity. Indeed, it allows program transformations that reduce non-determinism, *i.e.* that choose a strategy from a set of possibilities.

We are getting closer to the compiler correctness property. The backward simulation property is still slightly too strong. Recall the notion of undefined behaviour in C. It is stated in the C standard that when a C program exhibits undefined behaviour, the standard imposes no requirements on the implementation. That is to say, a C compiler can compile code with undefined behaviour into *any* program. More generally, going-wrong behaviours can be compiled into anything. The corresponding formal property is the backward simulation with behaviour improvement. We say that a behaviour $B_{\mathcal{T}}$ improves over a behaviour $B_{\mathcal{S}}$ (written $B_{\mathcal{S}} \sqsubseteq B_{\mathcal{T}}$) if either $B_{\mathcal{S}}$ is a going-wrong behaviour, or $B_{\mathcal{S}} = B_{\mathcal{T}}$. We write $P \Downarrow B$ to denote that program P exhibits behaviours B, *i.e.* $B \in Beh(P)$.

Property 2.3.3 (Backward simulation with behaviour improvement)**.** *Every behaviour of $\mathcal{T}$ is an improvement over a behaviour of $\mathcal{S}$.*

$$\forall B,\ \mathcal{T} \Downarrow B \Rightarrow \exists B',\ \mathcal{S} \Downarrow B' \ \wedge\ B' \sqsubseteq B$$

This property is suitable for verified compilation. As a matter of fact, this is the final theorem of the CompCert compiler (see `transf_c_program_preservation`). A direct corollary is that programs that do not exhibit undefined behaviour enjoy the simpler backward simulation property without behaviour improvements. We say that a program P is safe (written $\mathit{Safe}(P)$) when it has no going-wrong behaviours. The following property gives a high-level view of the meaning of CompCert's theorem, however keep in mind that it is slightly weaker than Property 2.3.3.

Property 2.3.4 (Backward simulation for safe programs)**.**

$$\mathit{Safe}(\mathcal{S}) \Rightarrow \mathit{Beh}(\mathcal{T}) \subseteq \mathit{Beh}(\mathcal{S})$$

It is an interesting property because we can deduce for example that the compilation does not introduce bugs in safe programs. Indeed, if $\mathcal{T}$ contains a bug (*i.e.* an undesirable behaviour), then this behaviour was already present in the source program $\mathcal{S}$: it is therefore not the responsibility of the compiler.

Another interesting corollary is that safety properties proved on the source program are still valid for the target program. A safety property $\mathcal{P}$ is a property of traces, *i.e.* $\mathcal{P} \subseteq \mathcal{E}^*$. A program P satisfies a safety property $\mathcal{P}$ if all the behaviours of P are in $\mathcal{P}$, *i.e.* $\mathit{Beh}(P) \subseteq \mathcal{P}$. Because the inclusion of behaviours is transitive, if there is a backward simulation between programs $\mathcal{S}$ and $\mathcal{T}$, and $\mathcal{S}$ satisfies some safety property $\mathcal{P}$, then so does $\mathcal{T}$.

Backward simulations are difficult to prove. Indeed, it involves reasoning by induction on the semantics of the target language, and somehow inverting the compilation function to figure out the possible shapes of the source program that match the target program's constructs. An alternative property, called *forward simulation*, can be stated as follows.

Property 2.3.5 (Forward simulation with behaviour improvements)**.**

$$\forall B,\ \mathcal{S} \Downarrow B \Rightarrow \exists B',\ \mathcal{T} \Downarrow B' \ \wedge\ B \sqsubseteq B'$$

The forward simulation argument states that for every behaviour B of the source program $\mathcal{S}$, there exists a behaviour of $\mathcal{T}$ that is an improvement over B. While it may seem counter-intuitive, under certain conditions of determinacy on the semantics of the source and target languages, one can transform a forward simulation proof into a backward simulation proof. The appeal of doing so resides in the fact that forward simulations are much simpler to prove than backward simulations. The standard proof technique consists in a structural induction on the semantics of the input program. Each construct has only one image in the target program, *i.e.* one does not have to *invert* the compilation function.

This proof technique is however subject to a few restrictions. First, it must be provable that the forward simulation argument holds. For example, consider a program transformation that reduces non-determinism. It is not provable that the compiled program has more behaviours than the source program, because the very purpose of the transformation is to choose one behaviour among many. Second, the semantics of the source and target languages must obey certain determinacy requirements, that we do not detail here (see [Ler09a] for details).

Verified compilers and their correctness properties. The first mechanically verified realistic compiler is due to Young [You89]. It is a compiler from Gypsy (a programming language for specifying, implementing and proving programs) to Piton (a generic high-level assembly-like language). The correctness property associated with this compiler states the equivalence between an interpreter of the source language and the execution of the compiled program.

The correctness property used by the CompCert compiler has been explained above. However, early versions of CompCert [BDL06] provided a less useful theorem that was only applicable to programs which terminate, and the property claimed by the final theorem was merely the equality of the return values of programs. With the notion of behaviours, we have a closer matching between $\mathcal{S}$ and $\mathcal{T}$, namely they must exhibit the same trace of events all along their executions.

The notion of behaviour improvement used by CompCert is slightly stronger than the one presented above. Indeed, the definition we gave stated that *any* behaviour is an improvement of a going-wrong behaviour. Actually, in CompCert, a going-wrong behaviour has a trace τ of observable events until the point where the execution gets stuck. An improvement over a going-wrong behaviour with trace τ is any behaviour whose trace is prefixed by τ. In other words, all the events emitted before the triggering of undefined behaviour are preserved by CompCert.

CompCertTSO [Sev+11] is a fork of CompCert whereby concurrency and weak memory are modelled. Because of the concurrency introduced, the determinacy properties required to turn forward simulations (easier to prove) into backward simulations (valuable result) do not hold. Nevertheless, the executions are threadwise-deterministic. Hence, they can prove threadwise-forward simulations, that they transform into threadwise-backward simulations. Finally they can transform several threadwise-backward simulations into a whole-system backward simulation, hence proving a similar backward simulation property as CompCert.

CakeML [Tan+16] is an ML compiler that targets multiple architectures. The correctness compiler they use resembles the backward simulation property presented above, with the additional property that compiled programs are allowed to fail because of out of memory errors. Indeed, since it is difficult to estimate the memory consumption of ML programs, compiling into memory exhausting programs is permitted. The final theorem is therefore: the set of behaviours of the compiled program is a subset of the union of out-of-memory behaviours and the set of behaviours of the source program.

2.4 Simulation Relations

This section aims at providing proof techniques for proving forward simulations. We introduce simulation relations [Mil89], that are used to relate program states throughout whole executions. Given two transition systems $S_1 = (\Sigma_1, I_1, F_1, \rightarrow_1)$ and $S_2 = (\Sigma_2, I_2, F_2, \rightarrow_2)$, a binary relation $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ is a simulation relation if and only if it relates initial and final states of S_1 and S_2, and the relation is preserved all along the execution of programs. Formally, $\mathcal{R}$ is a simulation relation if:

- every initial state of S_1 has a matching initial state in S_2:

$$\forall \sigma_1 \in I_1,\ \exists\ \sigma_2 \in I_2,\ \sigma_1\ \mathcal{R}\ \sigma_2$$

- every final state of S_1 is matched only by final states of S_2:

$$\forall \sigma_1 \in F_1,\ \sigma_2 \in \Sigma_2,\ \sigma_1\ \mathcal{R}\ \sigma_2 \Rightarrow \sigma_2 \in F_2$$

a.

$$\begin{tikzcd}
\sigma_1 \arrow[d, "e_1"'] \\
\sigma_2 \arrow[d, "e_2"'] \\
\sigma_3
\end{tikzcd}$$

b.

$$\begin{tikzcd}
\sigma_1 \arrow[d, "e_1"'] \arrow[r, dashed, no head, "\mathcal{R}"] & \sigma_1' \\
\sigma_2 \arrow[d, "e_2"'] & \\
\sigma_3 &
\end{tikzcd}$$

c.

$$\begin{tikzcd}
\sigma_1 \arrow[d, "e_1"'] \arrow[r, no head, "\mathcal{R}"] & \sigma_1' \arrow[d, dashed, "e_1"] \\
\sigma_2 \arrow[d, "e_2"'] \arrow[r, dashed, no head, "\mathcal{R}"] & \sigma_2' \\
\sigma_3 &
\end{tikzcd}$$

d.

$$\begin{tikzcd}
\sigma_1 \arrow[d, "e_1"'] \arrow[r, no head, "\mathcal{R}"] & \sigma_1' \arrow[d, "e_1"] \\
\sigma_2 \arrow[d, "e_2"'] \arrow[r, no head, "\mathcal{R}"] & \sigma_2' \arrow[d, dashed, "e_2"] \\
\sigma_3 \arrow[r, dashed, no head, "\mathcal{R}"] & \sigma_3'
\end{tikzcd}$$

Figure 2.1: Forward simulation diagrams

- starting from states σ_1 and σ_2 such that $\sigma_1 \mathcal{R} \sigma_2$, any step from σ_1 to some σ_1' can be simulated by steps from σ_2 to σ_2' such that $\sigma_1' \mathcal{R} \sigma_2'$

If those properties are satisfied, we say that S_2 simulates S_1, or that the system S_1 is simulated by S_2.

This last requirement is intentionally vague because the preservation of the matching relation comes in various flavours, depending on the number of steps allowed for the target system to simulate one step of the source system. The simplest simulation property is called *lock-step* simulation. It captures executions where both programs perform the same number of steps, and program states stay related by $\mathcal{R}$ at every step of the execution. This can be formally stated and visualised as follows. The picture represents hypotheses as plain lines and conclusions as dashed lines.

$$\begin{aligned} \forall \quad & \sigma_1 \in \Sigma_1,\ \sigma_2 \in \Sigma_2, \\ & \sigma_1 \mathcal{R} \sigma_2 \Rightarrow \sigma_1 \xrightarrow{e}_1 \sigma_1' \Rightarrow \\ & \exists \sigma_2',\ \sigma_2 \xrightarrow{e}_2 \sigma_2' \ \wedge\ \sigma_1' \mathcal{R} \sigma_2' \end{aligned}$$

$$\begin{tikzcd}
\sigma_1 \arrow[d, "e"'] \arrow[r, no head, "\mathcal{R}"] & \sigma_2 \arrow[d, dashed, "e"] \\
\sigma_1' \arrow[r, dashed, no head, "\mathcal{R}"'] & \sigma_2'
\end{tikzcd}$$

This property is sufficient to prove Property 2.3.5, *i.e.* the forward simulation property. Consider the following derivation $\sigma_1 \xrightarrow{e_1} \sigma_2 \xrightarrow{e_2} \sigma_3$, where $\sigma_1 \in I_1$ and σ_3 in F_1. This is the situation depicted in Figure 2.1a. Because σ_1 is an initial state and $\mathcal{R}$ is a simulation relation, we know that there exists a corresponding initial state σ_1' such that $\sigma_1 \mathcal{R} \sigma_1'$, as illustrated by Figure 2.1b. Then, starting from those matching states, the step from σ_1 to σ_2 can be simulated by a step starting from σ_1'. This implies the existence of a state σ_2' such that $\sigma_1' \xrightarrow{e_1} \sigma_2'$ and $\sigma_2 \mathcal{R} \sigma_2'$, as illustrated by Figure 2.1c. A similar step exists for state σ_3 (Figure 2.1d.). Finally, since σ_3 is a final state and is in relation with σ_3', then σ_3' is also a final state. From this reasoning, it follows that all the behaviours of the first program (on the left-hand-side) are also behaviours of the second program.

More sophisticated properties can be used in lieu of the *lock-step* property. Indeed, one step in the source program may correspond to zero steps (*e.g.* if an optimisation removes some useless code) or many steps (*e.g.* a high-level construct is broken into several lower-level instructions). This is the purpose of the *star* simulation, shown below, where the relation $\xrightarrow{e}^*$ is the reflexive transitive closure of the $\xrightarrow{e}$ relation.

$$\begin{aligned} \forall \quad & \sigma_1 \in \Sigma_1,\ \sigma_2 \in \Sigma_2, \\ & \sigma_1 \mathcal{R} \sigma_2 \Rightarrow \sigma_1 \xrightarrow{e}_1 \sigma_1' \Rightarrow \\ & \exists \sigma_2',\ \sigma_2 \xrightarrow{e}_2^* \sigma_2' \ \wedge\ \sigma_1' \mathcal{R} \sigma_2' \end{aligned}$$

$$\begin{tikzcd}
\sigma_1 \arrow[d, "e"'] \arrow[r, no head, "\mathcal{R}"] & \sigma_2 \arrow[d, dashed, "e^*"] \\
\sigma_1' \arrow[r, dashed, no head, "\mathcal{R}"'] & \sigma_2'
\end{tikzcd}$$

2.5 CompCert

CompCert is an industrial-strength C compiler [Bed+12]. It compiles C code into assembly language for three different architectures: x86, PowerPC and ARM. CompCert is a formally-verified compiler, in the sense defined in Section 2.3. It is written in the Coq language, which allows to prove formal properties. This mechanisation of the correctness proof of the compiler gives a high-level of confidence in CompCert.

This section first introduces the overall architecture of CompCert, *i.e.* it describes the intermediate languages used in CompCert and briefly explains what the transformations between those languages do. Then, we describe the *memory model* of CompCert, *i.e.* how the memory is modelled and what operations can be performed. Finally, we introduce memory transformations, *i.e.* formal ways to relate memory states. In particular, we will focus on memory injections and memory extensions, which are a crucial notion of memory transformation used by several transformations.

2.5.1 Overall architecture of the CompCert compiler

The CompCert compiler targets three different architectures: x86, Power PC and ARM. CompCert compiles C programs into assembly programs through 9 intermediate languages, split between a *front-end* which is architecture-independent, and a *back-end* which is architecture-dependent. Figure 2.2 shows the different languages of the front-end (on the left-hand side) and of the back-end (on the right-hand side).

2.5.1.1 CompCert's front-end

The input language of CompCert's front-end is a large subset of C, called CompCert C, which includes all of MISRA-C 2004 [Mot04] and almost all of ISO C99 [ISO99], with the exceptions of variable-length arrays and unstructured, non-MISRA `switch` statements (*e.g.* Duff's device). CompCert C is non-deterministic, *i.e.* multiple behaviours are acceptable for a given C program. In particular, the order in which the arguments to a function call are evaluated is non-deterministic.

CompCert ships with an interpreter for CompCert C. This is an executable version of the semantics of C, which allows to test whether a given C program has defined semantics, and therefore whether the semantics preservation theorem applies for this program. The other semantics in CompCert are not executable, but are inductively defined predicates that describe which steps are allowed. While the operational semantics style can be rather easily transformed into executable semantics, it is not in general needed for the purpose of the semantics preservation proofs that are performed in CompCert.

The second language of CompCert is called Cstrategy. Its syntax is the same as C, but its semantics is deterministic, *i.e.* only one evaluation order for arguments to function calls is allowed. The very first proof amount to showing that every behaviour of the program in the Cstrategy semantics is also a behaviour in the CompCert C semantics. Note that the proof of this first pass is necessarily performed as a backward simulation proof, since the forward simulation property does not hold: there are some behaviours in the CompCert C semantics that have no counterpart in the Cstrategy semantics.

Cstrategy programs are then translated into Clight. Clight is a subset of C (*i.e.* any valid Clight program is a valid C program), where side-effects have been pulled out of expressions and made explicit. Clight programs are transformed into simpler Clight programs by the compilation pass `SimplLocals`. The aim of this pass is to transform certain local variables out of memory, and replace them by *temporaries*, *i.e.* pseudo-registers.

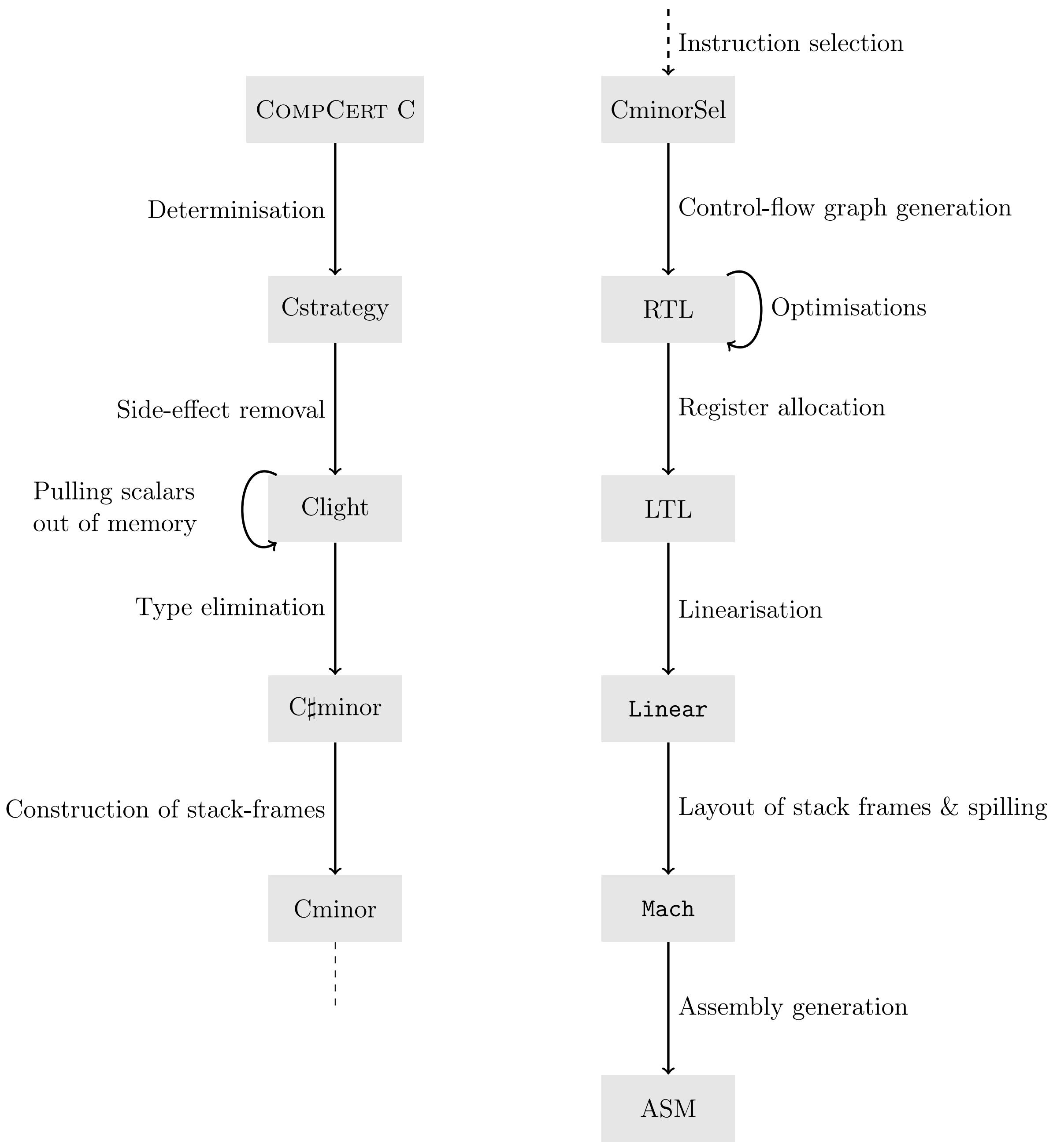


Figure 2.2: Architecture of CompCert

Clight is then transformed into C♯minor, where all type-information is erased and operations are transformed accordingly. For example, the Clight expression `p+2` where `p` is a pointer to `int` is transformed into the following C♯minor expression: `p+2*sizeof(int)`. The semantics of addition is then simpler in C♯minor because it does not need to reason about the type of its operands, but simply adds an offset to a pointer.

Finally, C♯minor programs are transformed into Cminor programs, where a stack frame is built for every function, and accesses to variables are translated into accesses in the stack frame. This transformation and its proof of correctness are more involved because the memory layout of the program is heavily modified. More details about this transformation are present in Section 8.2. This ends the front-end of CompCert, *i.e.* the architecture-independent part of the compiler.

2.5.1.2 CompCert's back-end

Subsequent passes are architecture-dependent and form the back-end of CompCert. Still, most of the back-end is common to all target architectures: the intermediate languages involved are the same; they are only parameterised by a different set of operators for expressions, for instance.

Cminor programs are transformed into CminorSel programs by an instruction selection pass. The goal of this transformation is to take advantage of the instructions available on the targeted architecture. For example, multiplication by a power of 2 can be turned into a logical left-shift during the selection pass.

CminorSel programs are then turned into RTL programs. RTL is a *register transfer language*, *i.e.* it is a 3-address code language. The code of functions is organised as control flow graphs, and instructions explicitly store their successors. RTL programs manipulate infinitely many pseudo-registers. Because the structure is simple, RTL is the host language for a number of static analyses and optimisations, such as inlining, constant propagation, common subexpression elimination, dead-code elimination and tail code recognition.

RTL code is then transformed into LTL code. This is the register allocation pass. The structure of programs is the same as in RTL. However, LTL programs manipulate only finitely many registers. Also, the nodes of the control flow graph no longer contain single instructions but basic blocks of instructions (*i.e.* purely sequential code with no jumps or calls).

LTL code is *linearised* to produce `Linear` code. The structure of programs is now linear, *i.e.* the code of a function is not a control flow graph anymore but a list of instructions including conditional jumps and labels.

`Linear` code is transformed into lower-level `Mach` code during the `Stacking` pass. The `Mach` language is like `Linear` except that accesses to the stack frames of functions are made more concrete. The machine-specific layout for stack frames is specified and accesses to the function's stack frames are modified accordingly. In particular, the layout specifies how callee-save registers and spilled local variables fit in the stack frame.

Finally, `Mach` code is transformed into Assembly code. This last pass is truly architecture-dependent, *i.e.* the assembly language is necessarily different for the three target architectures: x86, PowerPC and ARM. `Mach` instructions are mapped to the actual assembly instruction that will be executed.

Every single program transformation comes with its proof of correctness with respect to a unique memory model. The final theorem of CompCert is the composition of all the correctness proofs of individual passes. In Chapter 8, we adapt the proofs of all these passes for our memory model.

2.5.2 The Memory Model of CompCert

The memory model of CompCert defines the layout of the memory and the different memory operations. It is shared by all the languages of the CompCert compiler. CompCert uses an abstract block-based model where memory is an infinite collection of separated blocks [Ler+14]. Intuitively, a block is an array of bytes that represent values. At the C level, each block corresponds to an allocated variable (*e.g.* a 32-bit integer is stored in a 4-byte-wide block, an array of 10 characters is stored in a 10-byte-wide block). In lower-level languages, this correspondence between variables and memory blocks does not hold anymore: for example, after the Cminor language, the local variables of a function are merged together in one block that serves as the stack frame of the function, therefore losing the variable-block correspondence.

2.5.2.1 Locations and values

The values used in CompCert's memory model are given in Figure 2.3. *Locations* l are pairs (b, i) where b is a block identifier and i is an integer offset that indicates a position within this block. *Values* (of type *val*) used in the semantics of the CompCert languages (see [LB08]) are the disjoint union of 32-bit integers (written $\texttt{int}(i)$), 64-bit integers (written $\texttt{long}(l)$), 32-bit floating-point numbers (written $\texttt{float}(f)$), 64-bit floating-point numbers (written $\texttt{double}(d)$), pointers (written $\texttt{ptr}(l)$), and the special value `undef` representing the result of undefined operations or the value of uninitialised variables. Operations are strict in `undef` *i.e.* they yield `undef` as soon as one of the operands is `undef`.

Locations:	l	$::=$	(b, i)	(block, integer offset)
Values:	val	$::=$	$\texttt{int}(i) \mid \texttt{long}(l)$	
			$\mid \texttt{float}(f) \mid \texttt{double}(d)$	
			$\mid \texttt{ptr}(l) \mid \texttt{undef}$	

Figure 2.3: CompCert's values

2.5.2.2 Memory and Operations

Abstract bytes:	`memval`	::=	`Byte`(b)	
			| `Pointer`(b, i, n)	
			| `Undef`	
Memory chunks:	`memory_chunk`	::=	`Mint8signed`	8-bit integers
		|	`Mint8unsigned`	
		|	`Mint16signed`	16-bit integers
		|	`Mint16unsigned`	
		|	`Mint32`	32-bit integers or pointers
		|	`Mfloat32`	32-bit floats
		|	`Mint64`	64-bit integers
		|	`Mfloat64`	64-bit floats

`alloc` $m\ lo\ hi = (m', b)$	Allocates a fresh block with bounds $[lo, hi[$.
`free` $m\ b = \lfloor m' \rfloor$	Frees (invalidates) the block b
`load` $\kappa\ m\ b\ i = \lfloor v \rfloor$	Reads consecutive bytes (as determined by κ) at block b, offset i of memory state m. If successful, returns the contents of these bytes as value v.
`store` $\kappa\ m\ b\ i\ v = \lfloor m' \rfloor$	Stores the value v as one or several consecutive bytes (as determined by κ) at offset i of block b. If successful, returns an updated memory state m'.
`loadbytes` $m\ b\ i\ n = \lfloor mvl \rfloor$	Reads n consecutive bytes from memory state m, starting at location (b, i). If successful, returns a list of `memvals`.
`storebytes` $m\ b\ i\ mvl = \lfloor m' \rfloor$	Stores the bytes from mvl in memory state m, starting at location (b, i). If successful, returns an updated memory state m'.
`size_chunk` κ	Returns the size (number of bytes) that κ holds.
`bounds` $m\ b$	Returns the bounds $[lo, hi[$ of block b.
`nextblock` m	Returns the identifier of the next block to be allocated.
`contents` $m\ b$	Returns the contents of block b as a finite map from offsets to `memvals`.

Figure 2.4: Operations over memory states

The memory itself is not a direct mapping from locations to values; instead it is a mapping from locations to *abstract bytes* called `memvals` (see Figure 2.4). This allows to reason about byte-level accesses to the memory. A `memval` is a byte-sized quantity that can be one of the following: `Undef` represents uninitialised bytes, `Byte` (b) represents the concrete byte (8-bit integer) b and `Pointer` (b, i, n) represents the n-th byte of the binary representation of the pointer `ptr`(b, i).

The memory model defines four main memory operations: `load`, `store`, `free` and `alloc`. The `load` and `store` operations are parameterised by a memory chunk κ which concisely describes the number of bytes to be fetched or written, and the signedness of the value. An access at location (b, o) with chunk κ is aligned if `size_chunk` κ divides o[3]. For

[3]It is slightly too strong a condition: a 64-bit float variable only needs to be accessed at addresses that

instance, the size of the chunk `Mint32` is 4 bytes, hence an integer could be accessed with this chunk at offsets that are multiples of 4. These operations are partial, *i.e.* they may fail *e.g.* when the access is out of bounds, misaligned, or when the value and the chunk are inconsistent. This is modelled by the option type: we write $\emptyset$ for failure and $\lfloor x \rfloor$ for a successful return of value x.

The memory model also defines lower-level memory access operations, namely `loadbytes` and `storebytes`, which allow to access the memory at the byte level, *i.e.* they load and store lists of `memval`s from and to the memory.

The `free` operation frees a given block. It fails when the given block either has never been allocated or has already been freed. The `alloc` operation allocates a new block of the requested size. It never fails, thus modelling an infinite memory.

The `nextblock` property of memory states gives the identifier of the next block to be allocated. It usually serves as a threshold for identifying whether blocks are valid (*i.e.* have been allocated). The `contents` property gives access to the internal structure of memory states and returns the finite map corresponding to a given block, that associates to each offset a `memval`. The `bounds` function returns the bounds of a given block. Those accessors (`nextblock`,`contents` and `bounds`) are not intended to be used in the semantics of the intermediate languages. Rather, using them in formal specifications give strong connections that we will make use of later in this thesis.

2.5.2.3 Pointer Arithmetic

A location (b, i) is valid for a memory m (written `valid`(m, b, i)) if the offset i lies within the bounds of the block b. It is weakly valid (written `weakly_valid`(m, b, i)) if it is either valid or just one byte past the end of its block. This accounts for a subtlety of the C standard, stating that pointers *one-past-the-end* of an object deserve a particular treatment, namely that they can be compared to the other pointers to this object. This is intended to make looping over an entire array easier, allowing to compare the current pointer to the pointer just *one-past-the-end.*

Example 2.5.1 (Valid and weakly valid pointers)**.** *Consider a block b with bounds $[0;3[$. Then, pointers `ptr`$(b, 0)$ and `ptr`$(b, 3)$ are valid (and also weakly valid* a fortiori*). Pointer `ptr`$(b, 4)$ is not valid, however it is weakly valid. Pointer `ptr`$(b, 5)$ is neither valid nor weakly valid.*

Pointer arithmetic is defined in Figure 2.5. The only defined operations on pointers are the addition of an integer offset to a pointer, the subtraction of an integer offset from a pointer, and the subtraction of two pointers that point to the same object. Comparisons are also defined between pointers to the same object. All operations not described are undefined (they return `undef`). Note that, starting from pointer `ptr`(b, i) it is not possible to reach a pointer to a different block *via* pointer arithmetic, as blocks are separated by construction.

2.5.3 Memory Transformations

Each of the compilation passes of CompCert is proved correct independently. For most of the passes, this amounts to showing a forward simulation between the source program and the target program, as explained in Section 2.4. Proving a forward simulation requires to exhibit some relation $\mathcal{R}$ over program states, and then prove that $\mathcal{R}$ is a forward simulation

are multiple of 4, not 8.

$\texttt{ptr}(b,o) \pm \texttt{int}(i)$	$= \texttt{ptr}(b, o \pm i)$	
$\texttt{ptr}(b,o) - \texttt{ptr}(b,o')$	$= \texttt{int}(o - o')$	
$\texttt{ptr}(b,o) \star \texttt{ptr}(b,o')$	$= o \star o'$	when $\star \in \{<, \leq, ==, \geq, >, \texttt{!=}\}$ and both pointers are weakly valid
$\texttt{ptr}(b,o) == \texttt{ptr}(b',o')$	$= \texttt{false}$	when $b \neq b' \wedge \texttt{valid}(m,b,o) \wedge \texttt{valid}(m,b',o')$
$\texttt{ptr}(b,o)\texttt{!=}\texttt{ptr}(b',o')$	$= \texttt{true}$	when $b \neq b' \wedge \texttt{valid}(m,b,o) \wedge \texttt{valid}(m,b',o')$
$\texttt{ptr}(b,o) \star \texttt{ptr}(b',o')$	$= \texttt{undef}$	when $b \neq b'$ and $\star \in \{<, \leq, \geq, >\}$
$\texttt{ptr}(b,o)\texttt{!=}\texttt{int}(0)$	$= \texttt{true}$	when $\texttt{weakly\_valid}(m,b,o)$
$\texttt{ptr}(b,o) == \texttt{int}(0)$	$= \texttt{false}$	when $\texttt{weakly\_valid}(m,b,o)$

Figure 2.5: Pointer arithmetic in CompCert

relation. Once $\mathcal{R}$ is fixed, the proof of the simulation is performed by induction on the semantic derivation of the source program, which can be lengthy but relatively straightforward. The difficulty of the proofs therefore lies in finding a suitable relation $\mathcal{R}$ that is strong enough to give enough information about the target states, but general enough so that it is indeed invariant throughout the execution of both programs.

Since $\mathcal{R}$ is a relation over program states, and program states always include memory states, relations over memory states are needed to construct simulation relations. CompCert defines two such relations over memory states that capture different memory transformations. Memory injections capture memory transformations that merge blocks together. Memory extensions capture memory transformations that do not change the number of blocks, but their size may increase and their contents may be specialised.

2.5.3.1 Memory Injections in CompCert

Memory injections are the most complex memory transformations in CompCert. They capture memory transformations that merge blocks together. The canonical example of memory injection is the `Cminorgen` pass, which transforms C♯minor programs into Cminor. At the C♯minor level, every local variable of a given function is stored in its own block. At the Cminor level, all local variables of a given function are stored in a single stack block, representing its stack frame. Memory blocks from the C♯minor program are mapped to offsets in the memory block of the Cminor program. This is shown in Figure 2.6, where three blocks are merged into a single one. Also, the values contained in the blocks are *injected* in a sense that will be explained by the `val_inject` predicate.

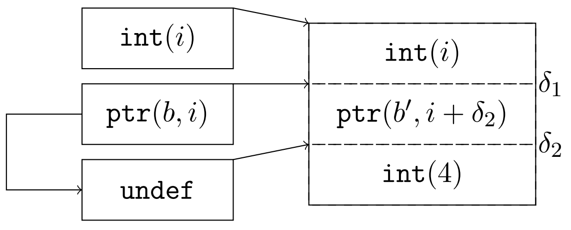


Figure 2.6: Injecting local variables into a stack block

Formally, a memory injection is a relation between two memory states m_1 and m_2 parameterised by an injection function f : $\texttt{block} \rightharpoonup \texttt{location}$[4] mapping blocks in m_1 to

[4]We use the notation $A \rightharpoonup B$ to denote partial function types. The actual type in the Coq implemen-

locations in m_2. The injection relation is defined over values (and called `val_inject`) and then lifted to memories (and called `mem_inject`).

The `val_inject` relation is defined inductively in Figure 2.7. Rule VINJ-PTR captures the intuitive semantics of injection that is depicted in Figure 2.6. It states that a pointer $\texttt{ptr}(b_1, i)$ is in injection with a pointer $\texttt{ptr}(b_2, i+\delta)$ if $f(b_1) = \lfloor (b_2, \delta) \rfloor$. Rule VINJ-VUNDEF states that `undef` is in injection with any value. Finally, Rule VINJ-NO-PTR states that for non-pointer values, the injection is reflexive.

$$\text{VINJ-PTR}\quad \frac{f(b_1) = \lfloor (b_2, \delta) \rfloor}{\texttt{val\_inject}\ f\ \texttt{ptr}(b_1, i)\ \texttt{ptr}(b_2, i+\delta)} \qquad \text{VINJ-VUNDEF}\quad \frac{}{\texttt{val\_inject}\ f\ \texttt{undef}\ v} \qquad \text{VINJ-NO-PTR}\quad \frac{v \neq \texttt{ptr}(b, i)}{\texttt{val\_inject}\ f\ v\ v}$$

Figure 2.7: `val_inject` in CompCert

The purpose of the injection of values is twofold: it establishes a relation between pointers using the function f but it can also specialise `undef` by any value. The latter can be understood intuitively as follows. Consider the situation of Figure 2.6. Consider the blocks on the left-hand side are named b_1, b_2 and b_3 from top to bottom. Now consider the pointer subtraction $\texttt{ptr}(b_3, 0) - \texttt{ptr}(b_2, 0)$. It evaluates to `undef` because of the pointer arithmetic rules defined in Figure 2.5. However, after the injection, the expression reads $\texttt{ptr}(b', \delta_2) - \texttt{ptr}(b', \delta_1)$, where b' is the name of the block on the right-hand side of Figure 2.6. This expression is well-defined and evaluates to $\texttt{int}(\delta_2 - \delta_1)$. Hence, we have transformed an `undef` result into a defined result by injection.

The relation `memval_inject` is built on the same principles as `val_inject` and relates `memval`s. It is defined as follows.

1. Concrete bytes are in injection with themselves only.

2. `Pointer` (b, i, n) is in injection with `Pointer` $(b', i+\delta, n)$ when $f(b) = \lfloor (b', \delta) \rfloor$.

3. `Undef` is in injection with any `memval`.

The `mem_inject` relation is built on top of `memval_inject`, but it also includes well-formedness properties. Consider a block b_1 of m_1 injected to a location (b_2, δ) of m_2; the following properties must hold to establish a memory injection between m_1 and m_2:

- for every valid offset o of b_1, $o + \delta$ must be a valid offset of b_2;

- δ must be properly aligned with respect to the size of b_1; and

- for every valid offset o of b_1, the `memval`s at locations (b_1, o) in m_1 and $(b_2, o+\delta)$ in m_2 must be related by `memval_inject`.

The alignment constraint ensures that all aligned accesses remain aligned after the injection, therefore that loads and stores are preserved by the injection. To build a valid memory injection, the injection f must also be an injective function, *i.e.* for every pair of disjoint blocks (b_1, b_2), the locations they are injected to do not overlap. The corresponding formal definition is the following:

tation uses option types and reads $A \to \texttt{option}\ B$. When the function is defined and returns a value v, we write $\lfloor v \rfloor$ (`Some v` in Coq). Otherwise, when it fails to produce a value, we write $\emptyset$ (`None` in Coq).

Definition 2.5.1 (`meminj_no_overlap`).

$$\begin{array}{l}\texttt{meminj\_no\_overlap}\ f\ m : \mathbb{P}\ :=\ \forall b_1\ b'_1\ \delta_1\ b_2\ b'_2\ \delta_2\ ofs_1\ ofs_2,\\ \quad b_1 \neq b_2 \Rightarrow f(b_1) = \lfloor(b'_1, \delta)\rfloor \Rightarrow f(b_2) = \lfloor(b'_2, \delta_2)\rfloor \Rightarrow\\ \quad \texttt{valid}(m, b_1, ofs_1) \Rightarrow \texttt{valid}(m, b_2, ofs_2) \Rightarrow (b'_1 \neq b'_2 \vee ofs_1 + \delta_1 \neq ofs_2 + \delta_2).\end{array}$$

The memory model provides theorems about the behaviour of memory operations with respect to injections. For example, Theorem 2.5.1 (`store_mapped_inject`) states that, starting from two memory states m_1 and m_2 in injection, if a store of a given value v_1 can be performed in m_1 at a location (b_1, o), resulting in a memory state m'_1, and if b_1 is injected into location b_2 at offset δ, then a store of a value v_2 (in injection with v_1) can be performed on m_2, resulting in a memory state m'_2 such that m'_1 and m'_2 are in injection.

Theorem 2.5.1 (`store_mapped_inject`).

$$\begin{array}{ll}\forall & f\ m_1\ m_2\ b_1\ b_2\ o\ \delta\ v_1\ v_2,\\ & \texttt{mem\_inject}\ f\ m_1\ m_2 \Rightarrow \texttt{store}\ \kappa\ m_1\ b_1\ o\ v_1 = \lfloor m'_1\rfloor \Rightarrow\\ & f(b_1) = \lfloor(b_2, \delta)\rfloor \Rightarrow \texttt{val\_inject}\ f\ v_1\ v_2 \Rightarrow\\ & \exists\ m'_2,\ \texttt{store}\ \kappa\ m_2\ b_2\ (o + \delta)\ v_2 = \lfloor m'_2\rfloor \wedge \texttt{mem\_inject}\ f\ m'_1\ m'_2.\end{array}$$

Similar theorems are proved for all the operations of the memory model (`load`, `store`, `alloc` and `free`, `loadbytes`, `storebytes`). Those theorems are the building blocks of the forward simulation theorems used in the correctness proofs of most compiler passes.

2.5.3.2 Memory Extensions

Not all compiler passes modify the memory layout as much as the `Cminorgen` pass. Most passes do not modify the structure of the memory, *i.e.* the number and size of blocks, but are allowed to specialise the values stored in the memory, *i.e.* transform `undef` values into any other value. Those transformations are captured by memory extensions.

First, since the contents of the memory can be specialised, we formalise in Definition 2.5.2 this specialisation of values by the *less-defined* relation over values.

Definition 2.5.2 (The *less-defined* relation). *A value v_1 is less defined than a value v_2 (written $v_1 \leq v_2$) either if v_1 is* `undef` *or if $v_1 = v_2$. This can be summarised by the two following rules:*

$$\textsc{lessdef-undef}\ \frac{}{\texttt{undef} \leq v} \qquad\qquad \textsc{lessdef-refl}\ \frac{}{v \leq v}$$

It is worth remarking that the less-defined relation is a special case of the `val_inject` relation, with $f(b) = \lfloor(b, 0)\rfloor$ for every block b. We call such an injection function the *identity injection* Hence, the extension relation over `memvals` is simply a special case of `memval_inject`.

Finally the memory extension relation also shares most properties with the memory injection relation with an identity injection function. Some properties needed for memory injections are not needed in the case of memory extensions, *e.g.* the fact that the injection function is injective does not need to be proved separately.

Because of the close relationship between extensions and injections, the theorems that hold for injections also hold for extensions and the proofs are factored. Most program transformations in CompCert are proved using the memory extension relation.

2.6 Notations

In the remainder of this thesis, we use some notations, defined in Appendix A.

Chapter 3

Motivation: Low-Level C Code In The Wild

The C standard leaves many behaviours underspecified. As explained in Section 2.1, underspecified behaviours are split between three categories: unspecified, implementation-defined or undefined behaviours [ISO99, §3.4].

Undefined behaviours have a dramatic impact on the human understanding of what a program is supposed to do. Consider the simple program in Figure 3.1. It performs a naive overflow check, assuming that signed overflow is defined in modular arithmetic, *i.e.* it wraps around modulo. Compiled with `gcc` (version 4.9.2) at optimisation levels `-O0` and `-O1`, it behaves as expected, *i.e.* the overflow check succeeds. However, at higher levels, the condition `i + 1 > i` is optimised and transformed into `true`. This optimisation is sound from the compiler's perspective because *a)* if the computation does not overflow, it is obvious that `i + 1 > i`, *b)* if it overflows, this is undefined behaviour and therefore the compiler is allowed to remove the else branch.

This counter-intuitive optimisation is not correct for CompCert, because its developers have made the choice to define signed overflow as a wrap-around behaviour, hence CompCert does not have the opportunity to optimise this.

Unsafe programming languages like C have undefined behaviours by nature because preventing them would require the introduction of runtime checks in the compiled programs, thus making the programs slower, while an important purpose of the C language is its speed of execution.

There is no way to give a meaningful semantics to an out-of-bound array access or a `null` pointer dereference. Yet, certain behaviours in C were made undefined on purpose to ease either the portability of the language across platforms or the development of efficient compilers. For example, the behaviour of signed overflow has been made undefined because at the time when the standard was written, several concurrent architectures used different

```
int main(){
  int i = INT_MAX;
  if (i + 1 > i) printf("Overflow check failed");
  else           printf("Overflow check succeeded");
  return 0;
}
```

Figure 3.1: A simple program triggering undefined behaviour

representations of signed integers (one's complement, two's complement or sign and magnitude representations) with different behaviours on overflow. Nowadays, most architectures use the two's complement representation in which the wrap-around behaviour is the one chosen on overflow: there is therefore little reason left to keep this behaviour undefined.

We believe that defining the semantics of real-life C idioms is the way to go to reconcile the programmer's intentions with the actual program's behaviour. CompCert went in that direction by defining the behaviour of signed overflow. We go further in that direction and aim at giving semantics to low-level idioms such as low-level pointer arithmetic and manipulation of uninitialised data, that are present in real-life code.

In the following, we give examples of low-level C programs that have no defined semantics in C (or in CompCert). These programs rely on the bit-representation of values. We use the `0x` prefix to denote hexadecimal constants and the `0b` prefix to denote binary constants. Sometimes, we will need to introduce variables for hexadecimal and binary digits that are unknown. We use upper case letters that cannot be mistaken for hexadecimal constant digits (*e.g.* `P`, `Q`, ...) to represent arbitrary hexadecimal digits and lower case letters to represent arbitrary binary digits.

First, Section 3.1 shows C programs that exploit the binary representation of pointers. Then, Section 3.2 shows programs that use uninitialised contents in computations. The programs we will show are excerpts (or derived from such excerpts) of real-life code that has been found in major open source software such as the Linux kernel, various standard C libraries and applications necessitating a low-level access to pointers.

3.1 Bitwise Pointer Arithmetic

The C standard does not specify the bit-width or the alignment of pointers: those are implementation-defined. In CompCert, pointers are 32-bit-wide. We consider, for the sake of the following examples, that the `malloc` function returns pointers that are 16-byte aligned (*i.e.* the 4 least significant bits of the returned address are zeros).

As we showed in Section 2.5.2, pointer arithmetic is very limited in C. In order to perform arbitrary operations over a pointer, it is possible to cast it to an unsigned integer of type `uintptr_t` for which the ISO C standard provides the following specification [ISO99, Section 7.18.1.4].

> [The type `uintptr_t`] designates an unsigned integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to pointer to `void`, and the result will compare equal to the original pointer.

We also know from [ISO99, Section 6.3.2.3] that any pointer can be converted to a pointer to `void`.

> A pointer to `void` may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to `void` and back again; the result shall compare equal to the original pointer.

Note that this specification is very weak and does not ensure anything if a pointer, cast to `uintptr_t`, is modified before being cast back.

In our model, a pointer fits into 32 bits and we implement `uintptr_t` as a 32-bit unsigned integer. More importantly, we ensure that casts between pointers and `uintptr_t` integers preserve the binary representation of both pointers and integers. In other words, casts between pointers and a `uintptr_t` integers are a no-op. In the following, we illustrate how existing low-level C idioms can exploit this specification.

3.1.1 Storing information in spare bits

With the previous specification of pointer casts, consider the code snippet of Figure 3.2. It is a made-up example inspired from an implementation of `malloc` in the standard library for Mac. The pointer `p` is a 16-byte aligned pointer to a heap-allocated integer obtained by a call to the `malloc` function. Therefore, the 4 trailing bits of the binary representation of `p` are zeros. We can think of the binary representation of `p` as `0xPQRSTUV0` where letters `P` to `V` are hexadecimal indeterminate values. The last digit of the representation of `p` is `0`, because of the 16-byte alignment constraint.

```
char hash(void *ptr);

int main(){
  int *p = (int *) malloc(sizeof(int));
  // p = 0xPQRSTUV0
  *p = 0;
  int *q = (int *) ((uintptr_t) p | (hash(p) & 0xF));
  // q = 0xPQRSTUVH
  int *r = (int *) (((uintptr_t) q >> 4) << 4);
  // r = 0xPQRSTUV0 = p
  return *r;
}
```

Figure 3.2: Unspecified behaviour: low-level pointer arithmetic

Next, pointer `q` is obtained from the pointer `p` by filling its 4 trailing bits with a hash of the pointer `p` (the hash is masked with `0xF` to ensure that it fits on 4 bits). We write `H` for the abstract digit corresponding to the hash of `p`. The representation of `q` is exactly that of `p` with the last digit changed to `H`. Then, pointer `r` is obtained by clearing (using left and right shifts) the 4 least significant bits of `q`, resulting in the binary representation of `r` being equal to that of `p`.

This pattern is commonly used as a hardening technique (*e.g.* in an implementation of `malloc`). [1] In this context, a list of free memory areas is maintained. The first bytes contained in those free areas indicate the size of the current chunk of memory and the address of the next. To ensure that the address of free chunks is not modified by a malicious user, a checksum is stored in the least significant bits of the pointer to the next free memory block.

Our model provides semantics to this program, which CompCert does not because of the undefined operations on pointers (`hash`, shifts, bitwise OR/AND).

3.1.2 System call return value

It is common for system calls (*e.g.* `mmap` or `sbrk`) to return either the pointer `(void *)-1` to indicate a failure, *e.g.* because no memory is available, or a pointer aligned on a page boundary. In two's complement arithmetic `-1` is encoded by the bit-pattern `0xFFFFFFFF`

[1] See "free list utilities" in `http://www.opensource.apple.com/source/Libc/Libc-594.1.4/gen/magazine_malloc.c`

and a page aligned pointer is of the form `0xPRSTU000`, assuming that the page size is 4kB. Consider the code of Figure 3.3 which calls `mmap` to allocate a single character. The call to `mmap` is rather complex. The important part here is the second argument: the size of the requested region. In our case, we request a 1-byte-wide region. The program then tests whether the allocation succeeded, and exits.

```
  int main(){
  char *p = (char*)mmap(NULL, 1,
                        PROT_READ|PROT_WRITE,
                        MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
  return (p == (void*) -1);
}
```

Figure 3.3: Undefined behaviour: `mmap` usage

In the semantics of C and CompCert, the comparison between a pointer and `-1` is undefined. The only allowed comparison between pointers and integers is when the integer is 0. However, we advocate that a defined meaning should be assigned to that program.

Suppose that the call to `mmap` fails and returns `-1`. In that case, the condition

```
(void *) -1 == (void *) -1
```

always holds and the program returns `1`. Otherwise, if `mmap` succeeds, the condition

```
0xPRSTU000 == 0xFFFFFFFF
```

does not hold, because `0` and `F` hexadecimal digits can not be equated, and the program returns 0. Again, because we model alignment constraints, we give a meaning to this program.

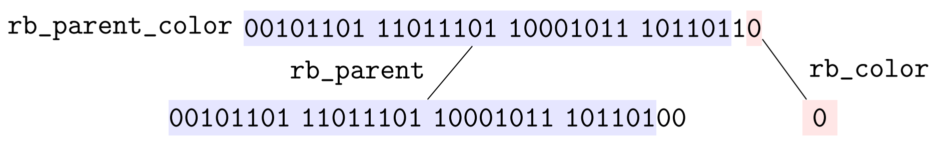
3.1.3 Red-Black Trees

The Linux kernel uses red-black trees as a data structure in schedulers to track various kinds of requests, in filesystems to store directory entries and in many other situations. Red-Black trees are defined in the `"include/rbtree.h"` header. The implementation of red-black trees aims at being very fast and memory efficient. To that end, the internal structure of red-black trees uses low-level bit-stealing, as shown in Figure 3.4.

The structure, shown in Figure 3.4a, contains three fields. The two pointers to other `rb_node`s are the pointers to the left and right children of this node. The first field, `rb_parent_color`, is the most intriguing. It stores, *in a single variable*, a pointer to the parent node and the color of the node (whether it is red or black). This is achieved *via* this simple reasoning. The pointer to the parent node is necessarily at least 4-byte aligned (because every field of a `rb_node` struct necessitates a 4-byte alignment); hence its 2 least significant bits are necessarily zeros. It is therefore possible to encode the color of the node using these two spare bits (actually, one bit suffices). The accessors `rb_parent` and `rb_color`, whose code is shown in Figure 3.4a, extract respectively the pointer to the parent node and the color of the node. Figure 3.4b illustrates the process of retrieving information from the `rb_parent_color` field. For example, the `rb_parent(r)` macro first accesses the `rb_parent_color` field of `r`, and discards the last two bits (this is the purpose of the `& ~3`). Then, this unsigned integer is casted into a pointer to a `rb_node`, resulting in the pointer to the parent node.

```
struct rb_node {
  uintptr_t rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};
#define rb_color(r) (((r)-> rb_parent_color) & 1)
#define rb_parent(r) \
  ((struct rb_node *) \
    ((r)-> rb_parent_color & ~3))
```

(a) Red-black tree C structure and accessors



(b) Extracting information from `rb_parent_color`

Figure 3.4: Red-black trees in Linux

3.1.4 XOR-linked lists

XOR-linked lists are a memory-efficient data structure that implements doubly-linked lists with only one pointer per node. Traditionally, a doubly-linked list is implemented through a structure similar to the following (`dll` stands for **d**oubly-**l**inked **l**ist):

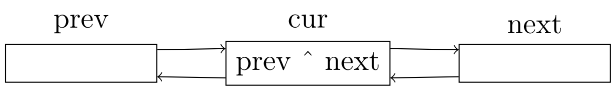
```
struct dll {
  int val;
  struct dll* prev;
  struct dll* next;
};
```

The structure is usually composed of a field for the actual value to be stored in the list (an integer in our example), and two fields that hold pointers to the previous and next elements of the list. XOR-linked lists are an improvement over doubly-linked lists because they only need one field for the pointers to previous and next elements. The idea is to store the result of XORing those pointers. The stucture of a XOR-linked list is shown in Figure 3.5a (`xll` stands for **X**OR-**l**inked **l**ist). The figure also shows the code to retrieve the pointer to the next element given a pointer to the previous element. The reasoning is as follows: the value stored in `prev_next` is the result of XORing the previous and the next pointers, *i.e.* `prev_next = prev ^ next`. XORing this with the previous pointer yields the following: `prev_next ^ prev == (prev ^ next) ^ prev` , which is equal to `next` by the rules of the XOR operator. Figure 3.5b illustrates the structure of a XOR-linked list.

Since retrieving the pointers or constructing the XOR of those pointers involves bitwise operations, this does not have defined semantics, neither in C nor in CompCert. We argue that this program should be given defined semantics, because thinking of pointers as integers is *de facto* a widely shared intuition among programmers.

```
struct xll {
  int val;
  uintptr_t prev_next;
};
typedef struct xll xll;
xll* next(xll* cur, xll* prev){
  return (xll*) ((cur -> prev_next) ^ ((uintptr_t) prev));
}
```

(a) Implementation



(b) A XOR linked list

Figure 3.5: XOR-linked lists: a memory-efficient doubly-linked list structure

3.1.5 Portable Software Fault Isolation

Software Fault Isolation (SFI) is a technique first introduced by Wahbe *et al.* [Wah+93], aiming at executing untrusted binary code in a sandbox. Roughly speaking, it consists of a program instrumentation that transforms every memory access into a memory access into a sandbox memory region, properly aligned so that the addresses of the whole safe memory region share a common prefix. Memory accesses are transformed by replacing the most significant bits of the address to be accessed by the prefix of the safe region. Consider for example that the safe region spans addresses `0xFDCB0000` to `0xFDCBFFFF`. The prefix of this region consists of the hexadecimal digits `FDCB`. Making an arbitrary address `addr` safe consists in first clearing the most significant bits from the addr and then replace them with the safe prefix, *i.e.* `(addr & 0x0000FFFF) | 0xFDCB0000`. This technique is inherently architecture-dependent because the instrumentation is made on assembly programs, that are tailored to a specific architecture.

Appel *et al.* [KSA14] proposed a *portable* version of SFI, where the instrumentation takes place in an architecture-independent language, that resembles C. This work is formalised in Coq, inside of CompCert. The SFI instrumentation is implemented in Coq, however the correctness proof of the program transformation has not been fully done. They have proved that the instrumented programs are *SFI-secure*, *i.e.* all the memory accesses are done within a pre-identified memory region. However, the masking function, which transforms any pointer into a pointer to a *sandbox* cannot be written in C, because it involves bit-level manipulation which have no defined semantics. This function is actually modelled as an external call whose semantics is axiomatised. We argue that we can give semantics to such a masking function with our low-level memory model. Doing so would provide a formal basis to reason about the security that these techniques add to the original program.

3.1.6 Variable Splitting Obfuscations

A program obfuscation is a program transformation that preserves the semantics of the original program while making it harder, for humans and tools, to understand. Collberg *et al.* [CTL97] introduces a number of obfuscating transformations, that can be applied to programs so as to increase their complexity. In particular, they introduce *variable splitting*.

This obfuscation splits each variable into several variables, thus losing some intuition of what the contents of variables is supposed to represent. A simple example of variable splitting consists in transforming every integer variable `x` into a pair of variables `x1` and `x2` such that `x1` holds the result of dividing `x` by 10 and `x2` holds the remainder of `x` by 10. It is always possible to recover the original value with the following expression: `x == x1 * 10 + x2`.

This can be done in C on integer variables, however not on pointers, because dividing or multiplying pointers is not permitted by the C standard, or CompCert's semantics.

In recent work, Blazy *et al.* [BT16] formalise in Coq and in CompCert another obfuscation: control flow graph flattening. This obfuscation aims at deconstructing loops and other control structures into lower-level `switch` constructs. Program analyses are therefore more subtle to perform, because a lot of abstraction has been lost. This obfuscation could be made even more aggressive if combined with the variable splitting obfuscation presented above.

Once again, we advocate that reasoning about the bit-pattern of pointers should be permitted, in a way that shall be described further in the remainder of this thesis.

3.1.7 Checking pointer alignment

Dynamic memory allocation operations allow to request memory regions during the execution of the programs. The simplest is probably the `malloc` function, which allocates a region of the requested size. The `memalign` function is slightly more complex, and allows to request a memory region aligned on some boundary, passed as a parameter. A call to `memalign(alignment,bytes)` is a request to allocate an `alignment`-byte aligned `bytes`-byte wide region. The implementation present in Doug Lea's allocator [Lea] first calls the `malloc` function and checks whether the pointer returned is correctly aligned. The following code checks if pointer `mem` is misaligned, where `mem` has type `void*`, and `alignment` is a `size_t`.

```
if ((((size_t)(mem)) & (alignment - 1)) != 0) /* misaligned */
```

To see why this code actually checks for misalignment, recall that a 2^n-byte aligned address has its n least significant bits set to 0. Consider that `mem` has the bit-pattern `ABCDEFGH` (on 8 bits, for the sake of simplicity), where each of the `A` ... `H` are binary digit variables. Consider also `alignment` to be 16, *i.e.* `0b00010000`.

A	B	C	D	E	F	G	H	`mem`
0	0	0	1	0	0	0	0	`alignment`
0	0	0	0	1	1	1	1	`alignment - 1`
0	0	0	0	E	F	G	H	`mem & (alignment - 1)`

The condition therefore holds if the last four bits of `mem` are 0, *i.e.* if `mem` is a multiple of 16. Once again, the code uses bitwise operations on pointers that are permitted neither by the C standard nor by CompCert. We argue that such programs should be given semantics.

3.2 Manipulation Of Uninitialised Data

Another axis of our work, except from bitwise pointer arithmetic, is the use of uninitialised data. The C standard states that any read access to uninitialised memory triggers undefined behaviour [ISO99, section 6.7.8, §10]: "If an object that has automatic storage duration is not initialised explicitly, its value is indeterminate." Here, indeterminate means that the value is either unspecified or a *trap representation*. In case the object may have a trap representation[2], reading a variable's value before it has been initialised is an undefined behaviour. In CompCert, reading uninitialised data returns the special `undef` value upon which no computation can be meaningfully performed. In this work, we aim at being more permissive. We want to model that uninitialised memory has an indeterminate arbitrary but stable value. To be more precise, we ensure that reading twice from the same uninitialised location returns the same result. We show below a number of idioms found in real world C code, that would benefit from this more defined semantics.

3.2.1 Flag setting in an integer variable

Consider the code snippet of Figure 3.6 that is representative of a C pattern found in an implementation of `malloc` (see Section 6.4.2.3).

```
unsigned int set(unsigned int p, unsigned int flag) {
  return p | (1 << flag);
}

int isset(unsigned int p, unsigned int flag) {
  return (p & (1 << flag)) != 0;
}

int main() {
  unsigned int status = set(status,0);
  return isset(status,0);
}
```

Figure 3.6: Undefined behaviour: reading the uninitialised variable `status`

The program declares a `status` variable of type `unsigned int` whose purpose is to store a number of bits. Function `set` is used to set some bit, addressed by its number, and function `isset` checks whether some bit is set. The `main` function first sets the least significant bit of `status`, then tests whether the least significant bit is set. The expected return value of the program is therefore obviously `1`.

According to the C standard, this program has undefined behaviour because the `set` function reads the value of the `status` variable before it is ever written.

However, we argue that this program should have a well-defined semantics and should always return the value `1`. The argument goes as follows: whatever the initial value of the variable `status`, the least significant bit of `status` is known to be 1 after the call `set(status,0)`. Moreover, the value of the other bits is irrelevant for the return value of the call `isset(status,0)`, which returns 1 if and only if the least significant bit of the variable `status` is 1. More formally, the program should return the value of the

[2]All types expect `unsigned char` may have trap representations.

expression `(status|(1 << 0))&(1 << 0) != 0` which simplifies to `(status|1)&1 != 0`, which evaluates to 1 no matter what the initial value of `status` is.

3.2.2 Bit-Fields in CompCert

Another motivation is illustrated by the translation of bit-fields in CompCert version 2.4: they are emulated in terms of bit-level operations by an elaboration pass preceding the formally verified front-end. Figure 3.7 gives an example of such a transformation.

```c
int main() {
  struct {
    unsigned int a0 : 1;
    unsigned int a1 : 1;
  } bf;
  bf.a1 = 1;
  return bf.a1;
}
```

(a) Bit-fields in C

```c
1  struct bfs {
2    unsigned char __bf1;
3  } bf;
4
5  int main(){
6    struct  { unsigned char __bf1;} bf;
7     bf.__bf1 = (bf.__bf1 & ~2U) | ((unsigned int) 1 << 1U & 2U);
8     return (int) ((unsigned int)(bf.__bf1 << 30) >> 31);
9  }
```

(b) Bit-fields in CompCert C

Figure 3.7: Emulation of bit-fields in CompCert

The program defines a structure with bit-fields `bf` with two fields `a0` and `a1`; both fields are 1-bit-wide. The `main` function sets the field `a1` of `bf` to 1 and then returns this value. The expected semantics is therefore that the program returns 1.

The transformed code (Figure 3.7b) is not very readable but the gist of it is that the bit-field structure is encoded by a standard structure with one integer field, and bit-field accesses are encoded using bitwise and shift operators, operating over the integer field of the transformed structure. After evaluation of compile time constants, Line 7 of the program in Figure 3.7b can be read as `bf.__bf1 = (bf.__bf1 & 0xFFFFFFFD) | 0x2`. The mask with `0xFFFFFFFD` clears the second least significant bit of `bf.__bf1` and keeps all the other bits unchanged. The bitwise OR with `0x2` sets the second least significant bit. In Line 8, the value of the field is extracted by first moving the field bit towards the most significant bit (`bf.__bf1 << 30`) and then moving this bit towards the least significant bit (`>> 31`). The transformation is correct and the target code generated by CompCert correctly returns 1. However, using the existing memory model, the semantics is undefined. Indeed, the program starts by reading the field `__bf1` of the uninitialised structure `bf`. This triggers undefined behaviour according to the C standard. Even though this case could be easily solved by modifying the pre-processing step, C programmers might themselves write such low-level code with reads of undefined memory and expect it to behave correctly. With our model of uninitialised memory, this program has a perfectly defined semantics.

3.2.3 Using uninitialised data as random seed

The following example is an excerpt from the FreeBSD standard C library.[3] The undefinedness of this program has been reported by Wang *et al.* [Wan+12]. It concerns the random number generator and in particular the generation of a random seed. The computation of the seed relies on some *junk*, *i.e.* arbitrary value read from an uninitialised variable that is believed to introduce randomness.

```
struct timeval tv;
unsigned long junk; // left uninitialised on purpose
gettimeofday(&tv, NULL);
srand((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

The computation of the seed relies on a bitwise combination of the current process id (`getpid()`), the current time (`tv.tv_sec` and `tv.tv_usec`) and some supposedly random uninitialised data (`junk`).

Since using uninitialised data in computations is undefined behaviour, compilers are free to remove all the seed computation and replace it with a constant seed value. This compiler "optimisation" actually happened with `gcc` and `clang` compilers. The FreeBSD code has since then fixed the seed computation by simply removing the `junk` variable.

While it is debatable whether it is a good idea to use uninitialised data to introduce randomness, it certainly shouldn't annihilate the security of software. We advocate for a more permissive semantics that considers that `junk` holds an arbitrary, unknown value, rather than a trap representation that allows compilers to transform arbitrarily the code of programmers.

3.3 Conclusion

We have introduced a number of C programs that exhibit undefined behaviour, either because of unallowed operations on the representation of pointers, *e.g.* bitwise operations, or because they use uninitialised data in computations. Those programs do not have well-defined semantics, neither in C nor in CompCert. However, all these programs are excerpts of code that have been found in real-life projects such as the Linux kernel or various versions of standard C libraries.

The fact that C programs found *in the wild* exhibit undefined behaviour raises an important problem: C programmers do not write C programs with the C standard as a mental model of which program constructs are allowed and which are forbidden, but with a more relaxed model that treats, in particular, pointers as mere integers and uninitialised data as arbitrary, non-blocking, values.

In a broad sense, the objective of this thesis is to reconcile the formal semantics of the C programming language, in particular the one used in the formally verified C compiler CompCert, with the relaxed mental model of C programmers.

A similar objective is shared by the proposal of a "Friendly C" by John Regehr *et al.*[4] This proposal is the fruit of the following discussion: undefined behaviour in C is

[3]See the exact SVN revision at: `http://svnweb.freebsd.org/base/head/lib/libc/stdlib/random.c?r1=241046&r2=241373`.

[4]See `http://blog.regehr.org/archives/1180`.

responsible for numerous unintuitive optimisations. Because they are unintuitive, `gcc` for example provides command-line switches to disable certain optimisations. For example the `-fwrapv` switch sets the behaviour for signed integer overflow to a wrap around behaviour. The *friendly C* dialect proposed by Regehr *et al.* can then be seen as a set of switches, whose effect is mainly to replace triggering of undefined behaviours with returning an unspecified value, thereby taming the power of optimisations. The work of this thesis is a step in that direction.

The following chapters introduce new formalisms into CompCert to make the semantics of C conform to that mental model. Eventually, we reprove the entire CompCert compiler with this relaxed model, resulting in CompCertS, a verified compiler for more real-life C programs.

Chapter 4

Symbolic Values and Normalisation

To give a semantics to the C idioms given in Chapter 3, a direct approach is to have a fully concrete memory model where a pointer is a mere integer and the memory is an array of bytes. In this model, bitwise operations on pointers are allowed because they are just bitwise operations on integers. Uninitialised data can be dealt with by introducing some kind of non-determinism. Initially, every location contains a non-deterministic byte. This model is indeed very expressive, however reasoning about it is cumbersome. For example, determining whether a pointer is valid necessitates to reconstruct abstractions. Another pitfall of the fully concrete memory model is that it prevents a number of optimisations. Consider the code snippet in Figure 4.1.

```
int x = 4;
f();
return x;
```

$$\xrightarrow{\text{Constant Propagation}}$$

```
int x = 4;
f();
return 4;
```

Figure 4.1: The constant propagation optimisation

This program allocates an integer `x` initialised to `4`. A constant propagation optimisation could transform the `return x` instruction into `return 4`. This is a valid optimisation in CompCert for any function `f` because the function is not given any way to access the local variable `x` and `f` can not forge a pointer to `x`. As a consequence, `f` cannot modify the value of `x`. However, this is not valid if pointers are mere integers. Indeed, in that case, function `f` may *guess* the address of `x` and modify its value, thus invalidating the optimisation. Consider for example that `x` is allocated at the concrete address `0x0000ABCD`. Function `f` may modify `x` with the following code: `*((int*)0x0000ABCD) = 7`. This makes the optimisation invalid, because instead of returning the new value of `x`, the *optimised* program returns its old value. Of course, it is improbable that `f` actually guesses the concrete address of `x`. However, it may iterate over concrete addresses, and modify the contents of all these addresses, possibly including that of `x`. This iteration is well-defined in a fully concrete memory model. On the other hand, it is not well-defined in CompCert to transform integers into pointers, thus making the optimisation valid.

In order to preserve the structure of the existing transformations and of the correctness proofs, and to keep the validity of the existing optimisations, we choose to keep the block-based memory that CompCert uses. This chapter first introduces *symbolic values*, that are used to denote the result of otherwise undefined constructs, while keeping an abstract block-based memory model. We show how to *evaluate* these symbolic values, and introduce the notions of *concrete memories* that give concrete addresses (32-bit integers) to abstract pointers and *indeterminate memories* that give arbitrary values to uninitialised data. Then, we define a *normalisation* process, whose aim is to recover a genuine value from a symbolic value. The notions defined in this chapter will be used in subsequent chapters to build a symbolic memory model and semantics for the intermediate languages of CompCert that enable reasoning about low-level C programs, and ultimately build CompCertS, a C compiler that provides guarantees for low-level C code that performs arbitrary pointer arithmetic and computations based on uninitialised data.

4.1 Symbolic Values

Our approach to improve the semantics coverage of CompCert consists in delaying the evaluation of expressions which result in undefined values in CompCert. To that end, we replace the semantic domain of CompCert values by *symbolic values*, defined in Figure 4.2.

$$
\begin{array}{llcll}
\text{Types:} & \mathit{typ} & ::= & \texttt{Tint} \mid \texttt{Tlong} \mid \texttt{Tsingle} \mid \texttt{Tfloat} & \\[1em]
\text{Comparisons:} & \mathit{cmp} & ::= & \texttt{Ceq} \mid \texttt{Cne} \mid \texttt{Clt} \mid \texttt{Cle} \mid \texttt{Cgt} \mid \texttt{Cge} & \\[1em]
\text{Operators:} & \texttt{op}_1 & ::= & \texttt{OpBoolval} \mid \texttt{OpNotbool} \mid \texttt{OpNeg} \mid \texttt{OpNot} \mid \texttt{OpAbs} & \\
 & & \mid & \texttt{OpZeroext} \mid \texttt{OpSignext} \mid \texttt{OpLoword} \mid \texttt{OpHiword} & \\
 & & \mid & \texttt{OpSingleofbits} \mid \texttt{OpDoubleofbits} & \\
 & & \mid & \texttt{OpBitsofsingle} \mid \texttt{OpBitsofdouble} & \\
 & & \mid & \texttt{OpConvert}(\mathit{tfrom}, \mathit{tto}) & \\
 & \texttt{op}_2 & ::= & \texttt{OpAnd} \mid \texttt{OpOr} \mid \texttt{OpXor} \mid \texttt{OpAdd} \mid \texttt{OpSub} \mid \texttt{OpMul} & \\
 & & \mid & \texttt{OpDiv} \mid \texttt{OpMod} \mid \texttt{OpShr} \mid \texttt{OpShl} \mid \texttt{OpCmp}(\mathit{cmp}) & \\
 & & \mid & \texttt{OpFloatofwords} \mid \texttt{OpLongofwords} & \\[1em]
\text{Symbolic values:} & \mathit{sv} & ::= & \mathit{val} & \text{value} \\
 & & \mid & \texttt{indet}(l) & \text{indeterminate content of location} \\
 & & \mid & \texttt{op}_1\ \mathit{sv} & \text{unary operation} \\
 & & \mid & \mathit{sv}_1\ \texttt{op}_2\ \mathit{sv}_2 & \text{binary operation}
\end{array}
$$

Figure 4.2: Semantics of symbolic values

The simplest case of symbolic values is a simple CompCert value *val*. It can therefore be a pointer $\texttt{ptr}(b, i)$, a 32-bit integer $\texttt{int}(i)$, a 64-bit integer $\texttt{long}(l)$, a 32-bit floating-point number $\texttt{float}(f)$ or a 64-bit floating-point number $\texttt{double}(d)$, or the special undefined value `undef`.

A symbolic value can also be an indeterminate value $\texttt{indet}(l)$ labelled by a location l. As we shall see in Section 5.3, indeterminate values will be used to model uninitialised memory. In particular, $\texttt{indet}(l)$ represents the arbitrary value that is stored at location l before any write is performed at this location.

Symbolic values can also denote operations with symbolic values as operands. The exhaustive list of unary operators ($\texttt{op}_1$) and binary operators ($\texttt{op}_2$) is given in Figure 4.2. These are all the operators that are defined on CompCert values and that are needed to evaluate C programs. Our semantics do not evaluate operators but instead construct symbolic values which represent delayed computations.

Operator `OpBoolval` transforms a value into a boolean (1 for `true` or 0 for `false`) value: any non-zero value is mapped to `true` and 0 is mapped to `false`. Operator `OpNotbool` is the boolean negation of `OpBoolval`. Operators `OpNeg` and `OpAbs` represent respectively the unary negation and the absolute value operators. Operator `OpNot` is the bitwise negation operator. Operators `OpZeroext`(n) and `OpSignext`(n) convert integers to n-bit wide integers, considering the operand respectively as unsigned and signed integers. Operators `OpLoword` and `OpHiword` retrieve respectively the least significant and the most significant 32-bit words from 64-bit integers. Operators `OpSingleofbits`, `OpDoubleofbits`, `Bitsofsingle` and `OpBitsofdouble` convert single (32-bit floating point) and double (64-bit floating point) values to their bit-pattern representation and back. Operator `OpConvert`$(\mathit{tfrom}, \mathit{tto})$ converts a symbolic value from type *tfrom* to type *tto*. Types are one of `Tint` (for 32-bit integers and pointers), `Tlong` (for 64-bit integers), `Tsingle` (for single-precision floating-point numbers) or `Tfloat` (for double-precision floating-point numbers).

Operators `OpAnd`, `OpOr`, `OpXor`, `OpShr` and `OpShl` perform the obvious bitwise operations. Operators `OpAdd`, `OpSub`, `OpMul`, `OpDiv`, `OpMod` perform the self-explanatory arithmetic operations. Operator `OpCmp`(cmp) performs the comparison *cmp*. Comparisons include equality, disequality and various inequality tests. Finally, `OpFloatofwords` and `OpLongofwords` construct respectively a 64-bit floating point and a 64-bit integer from the bit-patterns of two 32-bit words.

In this document, we use a concise and concrete C-like syntax for symbolic values and operators. For instance, we will write $(\texttt{ptr}(b, i) \mid \texttt{int}(3))\&\texttt{int}(3)$ instead of the less readable expression: $\texttt{OpAnd}(\texttt{OpOr}(\texttt{ptr}(b, i), \texttt{int}(3)), \texttt{int}(3))$.

4.2 Evaluation of Symbolic Values

Symbolic values were introduced with a promise that they would enable reasoning about the concrete encoding of pointers as integers and uninitialised data. So far, we have seen that we could build symbolic values instead of returning an undefined value or ending in a stuck semantic state. But we still have no clue about how to perform the *reasoning* we exposed in Chapter 3. A first step towards this end is to define an evaluation function, that maps symbolic values to values.

We need a more precise semantics for pointer operations. Indeed, the existing semantics of operations on pointers (defined in Figure 2.5) would result in a model almost identical to CompCert that does not enable reasoning about the bit-encoding of pointers. Note however, that a model with symbolic values and the existing semantics of pointers would give semantics to programs that contain undefined operations but do not use the result of such operations. For example, the program `x = y >> 33; return 3;` is undefined in CompCert because the shift amount cannot be greater than or equal to 32 for integer variables. However, with symbolic values, this program would have semantics because we would have simply stored a symbolic value in `x`, but never evaluated it.

We could enrich this restricted semantics of pointers to include special cases. For instance, we could state that the exclusive or (^) applied to two copies of the same operand yields 0 and that 0 is a neutral element for bitwise or (|):

$$\begin{array}{rcl} \texttt{ptr}(b, o)\,\hat{}\,\texttt{ptr}(b, o) & = & \texttt{int}(0) \\ \texttt{ptr}(b, o) \mid \texttt{int}(0) & = & \texttt{ptr}(b, o) \end{array}$$

This approach may help giving semantics to specific cases of pointer operations. However, giving a complete axiomatisation of low-level operations on pointers relying on simplification rules would be cumbersome, notably because of symmetric rules that make it difficult to implement an evaluation function. We rule out this approach to introduce a more principled way of evaluating symbolic values.

As we intend to reason about the bit-level encoding of pointers, we need to somehow model memory as an *array of bytes.* We introduce the notion of *concrete memory* for that purpose. A concrete memory cm is a mapping from block identifiers to concrete addresses as 32-bit integers. The evaluation of a pointer $\texttt{ptr}(b, i)$ in a concrete memory cm yields the concrete address of this pointer in cm, *i.e.* the integer $cm(b) + i$.

Indeterminate values $\texttt{indet}(l)$ model arbitrary values that represent the uninitialised contents of location l. We introduce the notion of *indeterminate mapping* for that purpose. An indeterminate mapping im is a mapping from locations to concrete byte values. The evaluation of an indeterminate value $\texttt{indet}(l)$ in an indeterminate mapping im yields $im(l)$.

The evaluation of other kinds of symbolic values is straightforward. Non-pointer values evaluate to themselves (regardless of cm and im). Unary and binary operations recursively evaluate their operands and apply CompCert's semantics for the corresponding operators (`eval_unop` and `eval_binop`). The complete set of rules for the evaluation function (written $[\![\cdot]\!]^{im}_{cm}$) is given in Figure 4.3.

Parameters:

$$\begin{array}{lll} cm & : \texttt{block} \to \texttt{int} & \text{concrete memory} \\ im & : \texttt{location} \to \texttt{byte} & \text{indeterminate mapping} \end{array}$$

Evaluation:

$$\begin{array}{ll} [\![\texttt{ptr}(b,i)]\!]^{im}_{cm} & = cm(b) + i \\ [\![v]\!]^{im}_{cm} & = v \\ [\![\texttt{indet}(l)]\!]^{im}_{cm} & = im(l) \\ [\![\texttt{op}_1\ sv]\!]^{im}_{cm} & = \texttt{eval\_unop}(\texttt{op}_1, [\![sv]\!]^{im}_{cm}) \\ [\![sv_1\ \texttt{op}_2\ sv_2]\!]^{im}_{cm} & = \texttt{eval\_binop}(\texttt{op}_2, [\![sv_1]\!]^{im}_{cm}, [\![sv_2]\!]^{im}_{cm}) \end{array}$$

Figure 4.3: The evaluation of symbolic values

The evaluation function is a total function: evaluation errors are mapped to the `undef` value (*e.g.* oversized shifts return `undef`).

In the rest of this document, we call cm a concrete memory and im an indeterminate mapping. Both cm and im bridge the gap between the high-level concepts of blocks and locations and a low-level memory model represented as an array of bytes.

Notice that symbolic expressions are side-effect free, therefore their evaluation is independent from the contents of the memory. We can also note that the result of the evaluation of any symbolic value sv is always a non-pointer value. The process of evaluation has brought us to a lower-level model where there is no distinction between pointers and integers, which was our original goal: treating pointers as integers.

4.3 Well-formedness Condition for Concrete Memories

As stated earlier, a concrete memory cm maps blocks to concrete addresses, representing the base address of this block. In this section, we show that not all concrete memories are of interest, because some concrete memories do not represent feasible low-level memory states, and we characterise the properties that one concrete memory must satisfy to be *valid.*

Intuitively, those properties should ensure that the semantics of operations on pointers that are defined in C (see Figure 2.5) are preserved in valid concrete memories.

This section proceeds by trial-and-error to discover the interesting properties that a valid concrete memory should satisfy. Starting from an abstract memory state, we explore the set of concrete memories, from naive models – which do not satisfy the assumptions we expect from C pointer arithmetic – to more elaborate models – that are consistent with the already defined C pointer arithmetic. This process intends to imitate our initial reasoning regarding concrete memories.

4.3.1 Towards a notion of validity for concrete memories

Throughout this section, we will consider an abstract (block-based) memory m with two distinct blocks b_1 and b_2, both 16-bytes wide. We will then propose several concrete memories and show why these concrete memories are valid or not.

Address space and location overlap. A first requirement for concrete memories to be valid is that they give the same semantics as CompCert to pointer comparisons. Example 4.3.1 illustrates a counter-example to that informal rule.

Example 4.3.1. *Consider a trivial concrete memory* $cm_0 = \lambda b.\ 0$*,* i.e. *the concrete memory that maps every block to the address* 0*.*

In the abstract model, $\boldsymbol{ptr}(b_1, 0) \neq \boldsymbol{ptr}(b_2, 0)$ *yields* `true` *because* b_1 *and* b_2 *are distinct blocks and* 0 *is a valid offset of both blocks (see Figure 2.5 for the definition of pointer comparisons in* CompCert*).*

However, the evaluation of the corresponding symbolic value in cm_0 *yields* `false`*:*

$$\begin{aligned}
[\![\boldsymbol{ptr}(b_1,0) \neq \boldsymbol{ptr}(b_2,0)]\!]^{im}_{cm_0} &= [\![\boldsymbol{ptr}(b_1,0)]\!]^{im}_{cm_0} \neq [\![\boldsymbol{ptr}(b_2,0)]\!]^{im}_{cm_0} \\
&= \boldsymbol{int}(cm_0(b_1)+0) \neq \boldsymbol{int}(cm_0(b_2)+0) \\
&= \boldsymbol{int}(0+0) \neq \boldsymbol{int}(0+0) \\
&= \boldsymbol{int}(0) \neq \boldsymbol{int}(0) = \mathtt{false}
\end{aligned}$$

This concrete memory cm_0 is not to be considered valid, because it gives different semantics to pointer comparisons, compared with CompCert. This leads us to the first validity condition for a concrete memory cm with respect to an abstract memory m.

Property 4.3.1 (No overlap)**.** *A concrete memory* cm *has the no-overlap property for a memory* m *if* cm *is an injective function for valid locations. In other words, for any two valid locations in different blocks,* cm *gives distinct concrete addresses. Formally,*

$$\begin{aligned}
\forall\ b_1\ i_1\ b_2\ i_2,\quad & b_1 \neq b_2\ \wedge\ \mathtt{valid}(m, b_1, i_1)\ \wedge\ \mathtt{valid}(m, b_2, i_2) \Rightarrow \\
& cm(b_1) + i_1 \neq cm(b_2) + i_2
\end{aligned}$$

Another defined operation on pointers is comparison of pointers to the same object. It states that for two valid pointers $\mathtt{ptr}(b, i_1)$ and $\mathtt{ptr}(b, i_2)$, the operation $\mathtt{ptr}(b, i_1) > \mathtt{ptr}(b, i_2)$ can be reduced to the comparison of the offsets of pointers, namely $i_1 > i_2$. Example 4.3.2 shows that not all concrete memories that satisfy Property 4.3.1 give the expected semantics to pointer comparison, and extra care needs to be taken.

Example 4.3.2. *Let us now consider* $cm_1 \triangleq cm_0[b_1 \mapsto 2^{32} - 8, b_2 \mapsto 16]$*. This concrete memory* cm_1 *satisfies Property 4.3.1,* i.e. *valid locations do not overlap. However, this concrete memory fails to transport the comparison of pointers to the same object to the comparison of their offsets.*

In CompCert*'s memory model,* $\mathbf{ptr}(b_1, 0) < \mathbf{ptr}(b_1, 15)$ *evaluates to* `true` *(see Figure 2.5 in Section 2.5.2).*

However, the evaluation of this symbolic value in cm_1 *yields a different result.*

$$\begin{aligned}
[\![\mathbf{ptr}(b_1,0) < \mathbf{ptr}(b_1,16)]\!]^{im}_{cm_1} &= [\![\mathbf{ptr}(b_1,0)]\!]^{im}_{cm_1} < [\![\mathbf{ptr}(b_1,16)]\!]^{im}_{cm_1}\\
&= \mathbf{int}(cm_1(b_1)+0) < \mathbf{int}(cm_1(b_1)+16)\\
&= \mathbf{int}(2^{32}-8) < \mathbf{int}(2^{32}+8)\\
&= \mathbf{int}(2^{32}-8) < \mathbf{int}(8) = \mathtt{false}
\end{aligned}$$

The reason of this unintuitive behaviour is that the addition on 32-bit integers may overflow, and therefore not behave as the addition of mathematical integers. A first attempt at avoiding this situation is to specify that valid locations must have concrete addresses in the range $]0; 2^{32}[$. This restriction only prevents valid concrete addresses to be 0; all the other 32-bit integers are valid. An immediate side-product of this restriction is that comparisons between valid pointers and the `NULL` pointer always evaluate to `false` in valid concrete memories, as expected in the CompCert semantics of pointers. It also prevents cm_1 from being a valid concrete memory, because pointer $\mathtt{ptr}(b_1, 8)$ is valid and has concrete address $\mathtt{int}(cm_1(b_1) + 8) = \mathtt{int}(2^{32} - 8 + 8) = \mathtt{int}(2^{32}) = \mathtt{int}(0)$. More generally, because the set of valid offsets for a given block b in memory m is convex (*i.e.* , if $\mathtt{valid}(m, b, i_1)$ and $\mathtt{valid}(m, b, i_2)$, then $\forall i,\ i_1 \leq i \leq i_2 \Rightarrow \mathtt{valid}(m, b, i)$), it is impossible to have a block begin at a concrete address at the end of the memory (*e.g.* $2^{32} - 4$) and end at the beginning of the memory (*e.g.* at concrete address 4), because that would violate this range property.

However, this is not strong enough a restriction because of pointers one-past-the-end, briefly discussed in Section 2.5.2. The C standard stipulates that, given an array of n elements, `a[n]`, the addresses of successive elements (including n) are strictly increasing. Formally, we have: `a+0` $<$ `a+1` $< \cdots <$ `a+(n-1)`$<$ `a+n`. Note that `a+n` is a pointer *one-past-the-end* of the array. Example 4.3.3 shows why the previous restriction to the $]0; 2^{32}[$ interval is too weak.

Example 4.3.3. *Consider the concrete memory* $cm_2 = cm_0[b_1 \mapsto 2^{32} - 16; b_2 \mapsto 8]$. *It satisfies Property 4.3.1 and the range restriction discussed above. Consider that* b_1 *is the block associated with an array* `a` *of* `char` *variables of size* 16.

In CompCert, *we have that* `a[0] < a[16]` *because both locations,* $(b_1, 0)$ *and* $(b_1, 16)$*, are* weakly valid*, and this reduces to the simpler test* `0 < 16`*, which is* `true`*.*

The evaluation of the symbolic value $\mathbf{ptr}(b_1, 0) < \mathbf{ptr}(b_1, 16)$ *in* cm_2 *yields* `false`*:*

$$\begin{aligned}
[\![\mathbf{ptr}(b_1,0) < \mathbf{ptr}(b_1,16)]\!]^{im}_{cm_2} &= [\![\mathbf{ptr}(b_1,0)]\!]^{im}_{cm_2} < [\![\mathbf{ptr}(b_1,16)]\!]^{im}_{cm_2}\\
&= cm_2(b_1) < cm_2(b_1)+16\\
&= \mathbf{int}(2^{32}-16) < \mathbf{int}(2^{32}-16+16)\\
&= \mathbf{int}(2^{32}-16) < \mathbf{int}(2^{32})\\
&= \mathbf{int}(2^{32}-16) < \mathbf{int}(0) = \mathtt{false}
\end{aligned}$$

To solve this problem, we need to exclude the concrete address $2^{32} - 1$ from the address space, therefore preventing a possible wrap-around of `a+n` that would invalidate the inequality expected by the C standard.

These requirements yield the following property needed to be satisfied by concrete memories.

Property 4.3.2 (Address space)**.** *A concrete memory* cm *has the address-space property for a memory* m *if valid locations are given concrete addresses in the range* $]0; 2^{32} - 1[$.[1]

[1]We use the notation $]x; y[$ to denote the interval of integers $\{z \mid x < z < y\}$. It is equivalent to the notation (x, y) that can be found in other documents.

Formally,

$$\forall\ b\ i,\ \texttt{valid}(m, b, i) \Rightarrow 0 < cm(b) + i < 2^{32} - 1$$

Alignment constraints To comply with the C standard and the Application Binary Interfaces (ABI) of various architectures, blocks cannot be allocated at arbitrary addresses but must satisfy alignment constraints. The C standard requires that fields of structures are aligned in an implementation-defined way (see [ISO11, Section 6.7.2.1, § 14]). The ABI for Power PC requires natural alignment for loads and stores, *i.e.* a 8-byte quantity can only be stored to or loaded from an 8-byte aligned address. The ABI for ELF x86-32 has similar requirements. In CompCert, these alignment constraints are modelled at two different levels. First, `loads` and `stores` only succeed when given a sufficiently aligned address, *i.e.* a pointer with a sufficiently aligned offset. Second, when building the stack frames of functions, local variables are mapped to offsets in a single stack block so that the offsets are sufficiently aligned. The required alignment of variables depends on the number of bytes of the data-structure and an upper bound for it is given by the function `alignment_of_size`, which returns the number of trailing bits that must be zero:

Definition 4.3.1. $\texttt{alignment\_of\_size}\ \mathit{size} \triangleq \begin{cases} 0 & \mathit{if\ size} \leq 1 \\ 1 & \mathit{if}\ 2 \leq \mathit{size} \leq 3 \\ 2 & \mathit{if}\ 4 \leq \mathit{size} \leq 7 \\ 3 & \mathit{if}\ 8 \leq \mathit{size} \end{cases}$

In particular, a variable of type `char` (1-byte wide[2]) has no alignment constraint; a `short` (16-bit) integer is 2^1-byte aligned; an `int` (32-bit) integer is 2^2-byte aligned and a `long` (64-bit) integer is 2^3-byte aligned. It follows that the alignment of a block is obtained by the function `min_alignment` which retrieves the size of a block and returns the number of trailing bits that are 0s in the concrete representation of the block.

Definition 4.3.2. The `size` and `min_alignment` functions are defined as:

```
size(m,b) := let (lo,hi) := bounds(m,b) in hi - lo.
min_alignment(m,b) := alignment_of_size(size(m,b)).
```

Notice that this is a minimal alignment constraint, *i.e.* it is the weakest acceptable alignment for a given data size. However, a stricter alignment may be requested. Think for example of the `malloc`-allocated block in Figure 3.2 which is assumed to be 16-byte aligned, or of the memory chunks returned by `mmap` which are page-aligned blocks, where page alignment is typically 2^{12} for 4Ko pages. To account for this stronger alignment constraints in our model, the memory model explicitly associates with each block b an alignment, accessed through the function `alignment`, such that `alignment(m,b)` is always greater than or equal to `min_alignment(m,b)`.

Property 4.3.3 (Alignment constraint)**.** *A concrete memory cm satisfies the alignment-constraint property for an abstract memory m if every block is given a suitably aligned address. Formally,*

$$\forall\ b,\ \ cm(b)\ \texttt{mod}\ 2^{\texttt{alignment}(m,b)} = 0$$

This property can be equivalently stated as a divisibiliy property or using bit-masks, *i.e.* $cm(b)\ \&\ (2^{\texttt{alignment}(m,b)} - 1) = 0$.

We now have all the necessary definitions to state the definition of a *valid concrete memory.*

[2]All this work assumes a 32-bit architecture where `sizeof(char) = 1`, `sizeof(short int) = 2`, `sizeof(int) = 4` and `sizeof(long long) = 8`.

Definition 4.3.3 (Valid concrete memory)**.** *A concrete memory cm is valid for a memory m (written $cm \vdash m$), if and only if the three following properties are satisfied.*

1. *Property 4.3.1: valid locations from distinct blocks do not overlap.*
2. *Property 4.3.2: valid locations lie in the range* $]0; 2^{32} - 1[$.
3. *Property 4.3.3: blocks are mapped to suitably aligned addresses.*

This axiomatisation is sufficient to ensure that the operations on pointers that are defined in CompCert's model are still defined with the same semantics in CompCertS, our symbolic model. We will show in Section 6.2, after we define semantics of intermediate languages with symbolic values, that our semantics, based on this notion of valid concrete memories, are a refinement of that of CompCert. This validates the fact that this set of validity properties accurately models the axiomatisation of pointers in CompCert.

4.3.2 Preservation of validity of concrete memories by memory operations

Valid concrete memories will be very important objects in the following, since they will be used as the basis of reasoning for the formal semantics of C and all the intermediate languages used in CompCert. It is therefore interesting to study the properties of valid concrete memories and in particular the preservation (or lack thereof) of the validity of concrete memories by the operations that construct new memory states. We will show the following relations.

$$\forall\ cm,\ cm \vdash \texttt{empty} \tag{4.1}$$

$$\forall\ cm\ \kappa\ m\ sv\ b\ o\ m',\ \texttt{store}\ \kappa\ m\ b\ o\ sv = \lfloor m' \rfloor \Rightarrow (cm \vdash m \Leftrightarrow cm \vdash m') \tag{4.2}$$

$$\forall\ cm\ m\ lo\ hi\ b\ m',\ \texttt{alloc}\ m\ lo\ hi = (m', b) \Rightarrow cm \vdash m' \Rightarrow cm \vdash m \tag{4.3}$$

$$\forall\ cm\ m\ b\ m',\ \texttt{free}\ m\ b = \lfloor m' \rfloor \Rightarrow cm \vdash m \Rightarrow cm \vdash m' \tag{4.4}$$

As a base case, consider the `empty` memory which consists of 0 blocks. Every concrete memory is valid for `empty` (4.1) because Property 4.3.1 and Property 4.3.2 are only concerned with valid locations and therefore are vacuously satisfied in `empty`. Property 4.3.3 ensures that blocks are suitably aligned. However, in the `empty` memory, blocks do not have alignment constraints and are therefore trivially suitably aligned.

Now, consider a memory m and a concrete memory cm valid for m ($cm \vdash m$). The following investigates whether cm is still valid in the memory states obtained after a `store`, `alloc` or `free`.

After a `store` operation resulting in a memory state m', a key observation is that the bounds and the alignment constraints of all blocks are untouched. As a result, any valid location of m is a valid location of m' and *vice versa*, hence the two first properties hold. Also, since the alignment constraints are the same in both m and m', the third property of valid concrete memories holds. Hence cm is still valid in m' (4.2).

After an `alloc` operation resulting in a memory state m' and a new allocated block b, the situation is more delicate. In cm, b can be mapped to any address whatsoever because it is an invalid block for m. However, for cm to be a valid concrete memory of m', the address of block b has to satisfy the three properties of Definition 4.3.3. Hence in general, cm is not a valid concrete memory for m'. However, any valid concrete memory cm' for m' is also a valid concrete memory of m, because m' is *more constrained* (4.3).

Symmetrically, after a `free` operation resulting in a memory state m', we know that cm is still a valid concrete memory for m' (4.4) because freeing some block b amounts to clearing all the constraints associated with block b. Hence if cm satisfied a more constrained set of properties, it must satisfy a more relaxed one.

These properties will be used in Section 5.4 when we prove that there exist concrete memories that satisfy certain constraints for every abstract memory. Besides, these properties are useful to give intuition about valid concrete memories.

4.4 Normalisation of Symbolic Values

We have introduced symbolic values to capture the meaning of otherwise undefined operations, and we know how to *evaluate* these symbolic values for a given environment given by a concrete memory cm and an intermediate mapping im. However, we want to keep our memory model abstract, *i.e.* the memory object that the memory primitives operate on is still an abstract collection of blocks, and not a concrete memory (that would result in a flat memory model, discussed at the beginning of this chapter).

4.4.1 Sound normalisation

We introduce the notion of *normalisation*, which can be seen as a lifting of evaluation from concrete memories to abstract memories. The intuition behind the normalisation of a symbolic value sv is that the result v of the normalisation should evaluate as sv in every valid environment. We formalise this intuition in the definition of a *sound normalisation.*

Definition 4.4.1 (Sound normalisation)**.** *A value v is a* sound normalisation *of sv in a memory m (written $sv \overset{m}{\twoheadrightarrow} v$) if v and sv evaluate identically in every concrete memory cm valid for m and in every indeterminate mapping im. Formally,*

$$sv \overset{m}{\twoheadrightarrow} v \triangleq \forall\ cm \vdash m,\ \forall\ im, [\![sv]\!]^{im}_{cm} = [\![v]\!]^{im}_{cm}$$

Note that not all symbolic values have a sound normalisation, as illustrated by Example 4.4.1.

Example 4.4.1. *Consider the symbolic value $sv = \texttt{indet}(b, o)$. There does not exist a value v such that $[\![sv]\!]^{im}_{cm} = [\![v]\!]^{im}_{cm}$ for every im. That would imply that $\forall\ im\ im',\ im(b, o) = im'(b, o)$, which is a contradiction (take $im = \lambda\ l.0$ and $im' = \lambda\ l.1$ for example).*

Likewise for $sv' = \texttt{ptr}(b, 0) - \texttt{ptr}(b', 0)$: it evaluates to $\texttt{int}(cm(b) - cm(b'))$ for every $cm \vdash m$. For different concrete memories, the evaluation of sv' returns different values, hence there is no sound normalisation for sv'.

Using Definition 4.4.1, we can reason about the programs we exposed in Chapter 3. Example 4.4.2 illustrates this reasoning.

Example 4.4.2. *Consider the following code, copied from Figure 3.7b.*

```
struct bfs {
  unsigned char __bf1;
} bf;

int main(){
  struct  { unsigned char __bf1;} bf;
7     bf.__bf1 = (bf.__bf1 & ~2U) | ((unsigned int) 1 << 1U & 2U);
8     return (int) ((unsigned int)(bf.__bf1 << 30) >> 31);
9  }
```

Unlike the existing semantics, operators are not strict in **undef** *but construct symbolic values. Hence, in Line 7, we store in* `bf.__bf1` *the symbolic value sv defined by*

$$(\texttt{indet}(l)\&{\sim}\texttt{0x2}) \mid (1 \ll 1\&\texttt{0x2})$$

The value returned in Line 8 is the symbolic value $sv' = (sv \ll 30) \gg 31$. *Let us show that* $\texttt{int}(1)$ *is a sound normalisation of* sv'*, as expected (see Section 3.2.2).*

We need to show that for any concrete memory cm and any indeterminate memory im, we have $\llbracket \texttt{int}(1) \rrbracket^{im}_{cm} = \llbracket sv' \rrbracket^{im}_{cm}$.

$$\begin{aligned}
\llbracket sv \rrbracket^{im}_{cm} &= \llbracket (\texttt{indet}(l)\&{\sim}\texttt{0x2}) \ \mid \ (1 \ll 1\&\texttt{0x2}) \rrbracket^{im}_{cm} \\
&= \llbracket \texttt{indet}(l)\&{\sim}\texttt{0x2} \rrbracket^{im}_{cm} \ \mid \ \llbracket 1 \ll 1\&\texttt{0x2} \rrbracket^{im}_{cm} \\
&= (\llbracket \texttt{indet}(l) \rrbracket^{im}_{cm}\&\texttt{0xFFFFFFFD}) \ \mid \ \texttt{int}(2) \\
&= (im(l)\&\texttt{0xFFFFFFFD}) \ \mid \ \texttt{0x00000002}
\end{aligned}$$

$$\begin{aligned}
\llbracket sv' \rrbracket^{im}_{cm} &= \llbracket (sv \ll 30) \gg 31 \rrbracket^{im}_{cm} \\
&= (\llbracket sv \rrbracket^{im}_{cm} \ll \texttt{int}(30)) \gg \texttt{int}(31) \\
&= (((im(l)\&\texttt{0xFFFFFFFD}) \ \mid \ \texttt{0x00000002}) \ll \texttt{int}(30)) \gg \texttt{int}(31) \\
&= \texttt{int}(1)
\end{aligned}$$

We now detail the last rewriting step, and we write `0xPQRSTUVW` *for the hexadecimal representation of* $im(l)$.

$$\begin{array}{rcll}
im(l) & = & \texttt{0xPQRSTUVW} & \textit{where } \texttt{W} = \texttt{0bxyzt} \\
im(l)\&\texttt{0xFFFFFFFD} & = & \texttt{0xPQRSTUVX} & \textit{where } \texttt{X} = \texttt{0bxy0t} \\
(im(l)\&\texttt{0xFFFFFFFD}) \mid 2 & = & \texttt{0xPQRSTUVY} & \textit{where } \texttt{Y} = \texttt{0bxy1t} \\
((im(l)\&\texttt{0xFFFFFFFD}) \mid 2) \ll 30 & = & \texttt{0xZ0000000} & \textit{where } \texttt{Z} = \texttt{0b1t00} \\
(((im(l)\&\texttt{0xFFFFFFFD}) \mid 2) \ll 30) \gg 31 & = & \texttt{0x00000001} &
\end{array}$$

Hence $\texttt{int}(1)$ *is a sound normalisation of sv.*

4.4.2 The normalisation is functional

An important property that we expect is that the sound normalisation relation is functional: a given symbolic value admits at most one sound normalisation. After showing that this is not the case in general, we introduce a slight restriction over abstract memory states that ensures that this desirable property is always satisfied. This enables us to assume the existence of a `normalise` function, that will be used in the semantics of the intermediate languages and in the memory model.

Let us first examine Example 4.4.3, where two different values are equally sound normalisations of a same symbolic value.

Example 4.4.3. *Suppose a memory m with a single block b of size* $2^{32} - 9$. *Because it is 8-byte aligned and the last address* ($2^{32} - 1$) *is not in the range of valid addresses, the unique valid concrete memory cm for m is such that* $cm(b) = 8$. *As a result, both the value* $\texttt{int}(8)$ *and* $\texttt{ptr}(b, 0)$ *are a sound normalisation for the degenerate symbolic value* $\texttt{ptr}(b, 0)$.

In our previous work [BBW14; BBW15], we designed a more complex specification of the normalisation, that avoided this discrepancy. In particular, the normalisation was parameterised by the *kind* of result that is expected: a pointer or a non-pointer value. Moreover, when the expected result was a pointer, it could only be a valid pointer. These alternate definitions are more intricate and have the counter-intuitive side-effect that values did not always normalise to themselves (*e.g.* an invalid pointer has no sound normalisation).

We have later identified that the real problem of having multiple sound normalisations comes from *near out-of-memory situations*, *i.e.* situations where the memory is so constrained that few concrete memories are valid, and the position of one block can be deduced from the positions of others. We avoid this situation by ensuring that every abstract memory satisfies Property 4.4.1.

Property 4.4.1 (Sliding Blocks)**.** *A memory m is such that for any block b, there exist at least two valid concrete memories cm and cm' that allocate b at different concrete addresses while allocating all the other blocks at the same address. Formally,*

$$\forall\ b,\quad \exists cm, cm', \bigwedge \begin{cases} cm \vdash m \\ cm' \vdash m \\ cm(b) \neq cm'(b) \\ \forall b' \neq b, cm(b') = cm'(b') \end{cases}$$

In Section 5.4, we prove that Property 4.4.1 holds for every abstract memory. This suffices to show that the sound normalisation relation is functional.

Theorem 4.4.1 (`sound_norm_functional`)**.** *Assuming Property 4.4.1, the sound normalisation relation is functional,* i.e. *for every symbolic value sv and memory m, for any values v and v' such that $sv \overset{m}{\twoheadrightarrow} v$ and $sv \overset{m}{\twoheadrightarrow} v'$, we have $v = v'$.*

Proof. By Definition 4.4.1 because v and v' are sound normalisations of sv, we get:

$$\forall im,\ \forall cm \vdash m, [\![v]\!]^{im}_{cm} = [\![sv]\!]^{im}_{cm} \wedge [\![v']\!]^{im}_{cm} = [\![sv]\!]^{im}_{cm} \tag{4.5}$$

By transitivity, we get Hypothesis 4.6:

$$\forall im,\ \forall cm \vdash m, [\![v]\!]^{im}_{cm} = [\![v']\!]^{im}_{cm} \tag{4.6}$$

The proof then goes by case analysis over v and v'.

- Case $v \neq \texttt{ptr}(b, o)$ and $v' \neq \texttt{ptr}(b', o')$. By Property 4.4.1, there exists $cm \vdash m$. Moreover, because v and v' are not pointers, their evaluation is independent from cm and we get from Hypothesis 4.6: $v = [\![v]\!]^{im}_{cm} = [\![v']\!]^{im}_{cm} = v'$. Hence, the property holds.

- Case $v = \texttt{ptr}(b, o)$. By Property 4.4.1, we exhibit $cm \vdash m$ and $cm' \vdash m$ such that Hypotheses 4.7 and 4.8 hold:

$$cm(b) \neq cm'(b) \tag{4.7}$$
$$\forall\ b' \neq b,\ \ cm(b') = cm'(b') \tag{4.8}$$

  – Case $v' = \texttt{int}(i)$. From Hypothesis 4.6 using cm and cm', we get:

$$cm(b) + o = [\![v]\!]^{im}_{cm} = [\![v']\!]^{im}_{cm} = i = [\![v']\!]^{im}_{cm'} = [\![v]\!]^{im}_{cm'} = cm'(b) + o$$

  As a result, $cm(b) = cm(b')$. This contradicts Hypothesis 4.7 and the property holds.

- Case $v' = \texttt{ptr}(b', o')$.
  - Case $b = b'$. By Hypothesis 4.6, we get:

    $$cm(b) + o = [\![v]\!]^{im}_{cm} = [\![v']\!]^{im}_{cm} = cm(b) + o'$$

    As a result, we deduce that $o = o'$ and the property holds.
  - Case $b \neq b'$. By Hypothesis 4.6, we get:

    $$\begin{array}{ccccccc} cm(b) + o & = & [\![v]\!]^{im}_{cm} & = & [\![v']\!]^{im}_{cm} & = & cm(b') + o' \\ cm'(b) + o & = & [\![v]\!]^{im}_{cm'} & = & [\![v']\!]^{im}_{cm'} & = & cm'(b') + o' \end{array}$$

    Because $cm(b') = cm'(b')$ (from Hypothesis 4.8), we get by transitivity that $cm(b) + o = cm'(b) + o$ and therefore $cm(b) = cm'(b)$. This contradicts Hypothesis 4.7 and the property holds.
- Case $v' = \texttt{long}(l)$ or $v' = \texttt{float}(f)$ or $v' = \texttt{double}(d)$. This is in contradiction with the facts that v and v' evaluate the same and v is a pointer, hence v evaluates to an integer. The property holds.

- Other cases are symmetric and therefore the property holds.

□

Existence of a normalisation function. We have defined a sound normalisation relation, and have proved that this relation is functional, *i.e.* for a given memory state m and a given symbolic value sv, there is only one value v such that $sv \overset{m}{\twoheadrightarrow} v$. For the remainder of this document, we will assume the existence of a function that we call `normalise` of type $mem \rightarrow sval \rightarrow val$ such that for every memory m and symbolic value sv, `normalise` $m\ sv$ returns a value v such that $sv \overset{m}{\twoheadrightarrow} v$ when such a value exists, and returns `undef` otherwise.

Assuming the axiom of choice and the law of excluded middle, we can prove the existence of this normalisation function. The axiom of choice can be stated as follows, for any types A and B and any binary relation $\mathcal{R} \subseteq A \times B$:

$$(\forall x \in A, \exists y \in B, x\mathcal{R}y) \Rightarrow \exists f \in (A \rightarrow B), \forall x \in A, x\mathcal{R}(f(x))$$

The axiom of choice allows to transform a left-total binary relation $\mathcal{R}$ into a function that associates with every x a y such that $x\mathcal{R}y$. A binary relation is left-total if for every x, there exists a y such that $x\mathcal{R}y$. The $\twoheadrightarrow$ relation is not exactly such a relation: some symbolic values do not have sound normalisations, as explained in Example 4.4.1. However, we can complete the $\twoheadrightarrow$ relation by associating the symbolic values that have no sound normalisation with the value `undef`. We define `sound_norm_comp` as an inductive predicate satisfying the two following rules:

$$\text{SOUND-NORM-EXISTS}\quad \frac{sv \overset{m}{\twoheadrightarrow} v}{\texttt{sound\_norm\_comp}\ m\ sv\ v} \qquad\qquad \text{SOUND-NORM-NOT-EXISTS}\quad \frac{\neg\exists v,\ sv \overset{m}{\twoheadrightarrow} v}{\texttt{sound\_norm\_comp}\ m\ sv\ \texttt{undef}}$$

The law of excluded middle ($\forall P, P \vee \neg P$) is needed to prove that the completed relation `sound_norm_comp` is left-total, as required by the axiom of choice. Specifically, it is used to decide whether a symbolic value has a sound normalisation or not, *i.e.* we use the following specialised property:

$$\forall\ m\ sv, (\exists\ v,\ sv \overset{m}{\twoheadrightarrow} v) \vee \neg(\exists\ v,\ sv \overset{m}{\twoheadrightarrow} v)$$

Therefore, depending on which branch is true, we apply one of the two rules SOUND-NORM-EXISTS or SOUND-NORM-NOT-EXISTS, and prove the left-totality of `sound_norm_comp`. The axiom of choice may then be used, resulting in the existence of the normalisation function.

Actually, those axioms are not needed to prove the existence of the normalisation function, because the set of values of interest is finite. Values are said to be *of interest* if they are not pointers in unallocated blocks, *i.e.* they are not pointers $\texttt{ptr}(b, o)$ where b has never been allocated. The set of these values is finite because integers and single-precision floating point numbers can only take one of 2^{32} different values, and the set of 64-bit integers and double-precision floating point numbers can only take one of 2^{64} different values. The set of pointers of interest $\texttt{ptr}(b, o)$ is also finite because b must belong to the finite set of allocated blocks and o is a 32-bit integer, hence its possible values can be finitely enumerated. Likewise, for a given memory m, the set of valid concrete memories valid for m is finite. Indeed, there are only finitely many blocks to allocate inside a finite address space. Using these finiteness arguments, a naive implementation of a sound normalisation can be constructed. Algorithm 1 shows such an implementation. First, Function `is_norm` is an

Algorithm 1: Deciding the existence of a sound normalisation

```
Function is_norm($m, sv, v$) $\triangleq$
    input : $m$: a memory state
            $sv$: a symbolic value
            $v$: a candidate normalisation
    output: a boolean true if and only if $sv \overset{m}{\twoheadrightarrow} v$
    foreach $cm \vdash m$ do
        if $[\![sv]\!]_{cm}^{im} \neq [\![v]\!]_{cm}^{im}$ then return false
    end
    return true

Function normalise($m, sv$) $\triangleq$
    input : $m$: a memory state
            $sv$: a symbolic value
    output: a value $v$ such that $sv \overset{m}{\twoheadrightarrow} v$ if one exists; undef otherwise
    foreach $v \in val$ do
        if is_norm($m, sv, v$) then return $v$
    end
    return undef
```

implementation of the sound normalisation $\twoheadrightarrow$ relation. It enumerates all valid concrete memories (which is possible because this set is finite), and returns `true` if and only if the symbolic value and the value evaluate identically in every valid concrete memory. Then, Function `normalise` is an implementation of the normalisation function. It enumerates all values of interest (again, this is possible because of the finiteness of these values) and looks for a value that satisfy the `is_norm` predicate.

In the following, we assume the existence of this `normalise` function, and we give a more tractable implementation than that of Algorithm 1 in Chapter 6.

4.4.3 Syntactic appearance and normalisation

Another interesting property of the `normalise` function is given by Lemma 4.4.1. It states that a pointer $\texttt{ptr}(b, i)$ can only be the result of the normalisation of a symbolic value sv if b appears syntactically in sv. This is also a consequence of Property 4.4.1. This property is used in the implementation of the normalisation (see Section 6.3) but also to relate memory injections and normalisations (see Theorem 7.2.2).

Let us first define the notion of syntactic appearance. A block b appears in a symbolic value sv if $sv = \texttt{ptr}(b, o)$ for some o, or if b appears in any of the operands of unary or binary operations.

Definition 4.4.2 (Syntactic appearance of blocks)**.**

```
block_appears sv b :=
  match sv with
  | ptr(b′, i)      => b = b′
  | op₁ sv₁         => block_appears sv₁ b
  | sv₁ op₂ sv₂     => block_appears sv₁ b ∨ block_appears sv₂ b
  | _               => ⊥
  end.
```

Lemma 4.4.1 (`norm_block_appears`)**.** *For any memory m, for any symbolic value sv, if* `normalise` $m\ sv = \texttt{ptr}(b, i)$*, then the block b appears syntactically in sv.*

Proof. The proof is by contradiction. Assume b does not appear in sv. Property 4.4.1 applied on block b provides two concrete memories cm and cm' such that

$$\begin{cases} cm \vdash m \\ cm' \vdash m \\ cm(b) \neq cm'(b) \\ \forall b' \neq b, cm(b') = cm'(b') \end{cases}$$

For any indeterminate memory im, we can derive the two following contradictory facts:

- Since b does not appear in sv, and cm and cm' agree on all blocks but b, we have $[\![sv]\!]^{im}_{cm} = [\![sv]\!]^{im}_{cm'}$.
- By Definition 4.4.1 and because $sv \xrightarrow{m} \texttt{ptr}(b, i)$, we have that $[\![sv]\!]^{im}_{cm} = cm(b) + i$ and $[\![sv]\!]^{im}_{cm'} = cm'(b) + i$. Since $cm(b) \neq cm'(b)$, we have that $[\![sv]\!]^{im}_{cm} \neq [\![sv]\!]^{im}_{cm'}$.

□

4.5 Conclusion

In this chapter, we have defined the core notion of this work, namely symbolic values, that capture the meaning of operations that would have otherwise been undefined. We introduced the concept of *concrete memories* and *indeterminate memories*, which bridge the gap between the high-level abstract view of memory states of CompCert and the low-level concrete view that is needed to reason about *e.g.* bit-level encoding of pointers. We identify the properties that make concrete memories *valid*, *i.e.* they conform to what programmers expect. We show how to evaluate symbolic values, in order to recover values from symbolic values. We define a *normalisation* function that lifts the evaluation to all concrete memories valid for the considered abstract memory, *i.e.* it transforms a symbolic value into a value that evaluates identically in every valid environment.

All these notions form the basis of this work. In the next chapters, we will introduce them first in the memory model of CompCert (Chapter 5), then in the semantics of all the intermediate languages of CompCert (Chapter 6). Later chapters will show how the proofs of semantic preservation have been updated.

Chapter 5

A Novel Memory Model Using Symbolic Values

This chapter builds upon the definitions introduced in the previous chapter, namely those of symbolic values and normalisation. We adapt the memory model of CompCert, introduced in Section 2.5.2, with symbolic values, *i.e.* we replace CompCert values with symbolic values. This leads to a number of necessary modifications; this chapter highlights the most fundamental of those. A summary of the resulting symbolic memory model can be found in Figure 5.1. Section 5.1 explains how loads and stores are performed in this memory model, in particular it focuses on the encoding and decoding of symbolic values. Section 5.2 shows how properties of the memory model have been adapted to this symbolic memory model. Section 5.3 gives a precise account of our handling of uninitialised values. Finally, Section 5.4 shows how to implement the allocation operation to ensure that Property 4.4.1 (Sliding Blocks), needed for the well-behavedness of the normalisation, holds for every abstract memory state. All along this chapter, we will give a contrast between properties that are true of CompCert's memory model but not of CompCertS', and *vice versa.*

Symbolic `memvals`:

`smemval`	::=	$\texttt{Symbolic}(sv, n)$	n-th byte of symbolic value sv

Memory operations:

$\texttt{palloc}\ m\ lo\ hi = \lfloor(m', b)\rfloor$	Allocate a fresh block with bounds $[lo, hi[$. Fails if no concrete memory can be constructed.
$\texttt{free}\ m\ b = \lfloor m' \rfloor$	Free (invalidate) the block b
$\texttt{load}\ \kappa\ m\ b\ i = \lfloor sv \rfloor$	Read consecutive bytes (as determined by κ) at block b, offset i of memory state m. If successful, return the contents of these bytes as symbolic value sv.
$\texttt{store}\ \kappa\ m\ b\ i\ sv = \lfloor m' \rfloor$	Store the symbolic value sv as one or several consecutive bytes (as determined by κ) at offset i of block b. If successful, return an updated memory state m'.

Figure 5.1: The symbolic memory model

5.1 Encoding And Decoding Of Symbolic Values In Memory

The memory content of CompCert's memories is modelled by `memvals`, ranging over `Undef` for undefined bytes, `Byte` (b) for concrete byte b or `Pointer` (b, i, n) for the n-th byte of the pointer `ptr`(b, i). In our symbolic memory model, the memory content is no longer represented by the `memvals` that we described in Section 2.5.2. Rather, we use a generalised form called `smemval` (see Figure 5.1) with a single constructor that subsumes all the existing ones and makes it possible to encode symbolic values. A `smemval` is merely a pair `Symbolic` sv n composed of a symbolic value sv and a natural number n denoting the n-th byte of the symbolic value sv, following the same principles as the `Pointer` constructor of `memval`.

Symbolic `smemvals` contain the bit-level representation of the contents of the memory. To facilitate the decoding function `decode`, the symbolic values found inside `smemvals` are converted to the binary 64-bit representation by the `to_bits` function, shown in Figure 5.2. The `to_bits` function takes a chunk and a symbolic value and returns the bit-representation of the symbolic value. For instance, starting from an integer chunk, the 64-bit representation is obtained by applying an integer-to-long conversion. The function `convert` is simply a wrapper around `OpConvert`, introduced in Figure 4.2. Functions `bits_of_single` and `bits_of_double` retrieve the binary encoding of floating-point symbolic values. The

```
Definition to_bits chunk sv :=
  match chunk with
  | Mint8signed  | Mint8unsigned
  | Mint16signed | Mint16unsigned
  | Mint32             => convert Tint Tlong sv
  | Mint64             => sv
  | Mfloat32           => convert Tint Tlong (bits_of_single sv)
  | Mfloat64           => bits_of_double sv
  end.

Definition from_bits chunk sv :=
  match chunk with
  | Mint8signed      => sign_ext 8 (loword sv)
  | Mint8unsigned    => zero_ext 8 (loword sv)
  | Mint16signed     => sign_ext 16 (loword sv)
  | Mint16unsigned   => zero_ext 16 (loword sv)
  | Mint32           => loword sv
  | Mint64           => sv
  | Mfloat32         => single_of_bits (loword sv)
  | Mfloat64         => double_of_bits sv
  end.
```

Figure 5.2: Converting values to their bit-pattern representation

`from_bits` function does the opposite: it interprets a 64-bit bit pattern as a typed value, as dictated by a chunk. The functions `sign_ext`, `zero_ext`, `loword`, `single_of_bits` and `double_of_bits` are simple wrappers around the corresponding operators introduced in Figure 4.2 and their definition is omitted in this document.

Encoding a symbolic value sv into a list of `smemvals` with respect to a chunk κ is straightforward. It consists in building a list of $n =$ `size_chunk` κ elements of the form

`Symbolic` (`to_bits` κ *sv*) i, $i \in 0, \ldots, n-1$. For example, encoding a symbolic value *sv* of integer type with chunk `Mint32` results in the following list[1]:

$$\begin{array}{ll} [& \texttt{Symbolic}\ (\texttt{convert Tint Tlong}\ sv)\ 3; \\ & \texttt{Symbolic}\ (\texttt{convert Tint Tlong}\ sv)\ 2; \\ & \texttt{Symbolic}\ (\texttt{convert Tint Tlong}\ sv)\ 1; \\ & \texttt{Symbolic}\ (\texttt{convert Tint Tlong}\ sv)\ 0 \quad] \end{array}$$

Decoding a list of `smemvals` into a symbolic value is somewhat more involved. Let's first show how to decode one `smemval`: `Symbolic` *sv n*. We define a function `extr` : $sval \to \mathbb{N} \to sval$ in Figure 5.3 to that end. Its purpose is to extract the n-th byte from a symbolic value. It is defined recursively: the 0-th byte is obtained by masking the higher bits; the $(n+1)$-th byte of *sv* is obtained by shifting *sv* 8 bits to the right, then taking the n-th byte of the resulting symbolic value. For instance, `extr` *sv* 2 results in the symbolic value:

$$\begin{aligned} \texttt{extr}\ sv\ 2 &= \texttt{extr}\ (sv \gg 8)\ 1 \\ &= \texttt{extr}\ ((sv \gg 8) \gg 8)\ 0 \\ &= ((sv \gg 8) \gg 8)\ \&\ \texttt{0xFF} \end{aligned}$$

Function `smv_to_sval` (see Figure 5.3) is a simple lifting of `extr` to `smemvals`.

```
Fixpoint extr (sv : sval) (n: nat) : sval :=
match n with
| O => sv & 0xFF
| S m => extr (sv >> 8) m
end.

Definition smv_to_sval (smv: smemval) : sval :=
match smv with Symbolic sv n => extr sv n
end.
```

Figure 5.3: Decoding a `smemval` into a symbolic value

Now, we need to decode lists of such `smemvals`. This is done by converting each `smemval` into a symbolic value using `smv_to_sval`, and then concatenating those symbolic values: the `concat` function (Figure 5.4) takes a list of `smemvals` in little-endian order[2] (least significant bytes first) and builds a symbolic value that represents the binary encoding of the value to be read. Finally, the `decode` function applies the `from_bits` function to the result of `concat` with the appropriate chunk, yielding the decoded symbolic value.

In the higher-level memory model, the `load` operation first retrieves a list of `smemvals` (the number depends on the chunk κ) and then decodes this list with the aforementioned `decode` function. Note that it results in a symbolic value, where the original CompCert `load` operation resulted in a value.

Symmetrically, the `store` operation first encodes the symbolic value to be stored into a list of `smemvals` (as opposed to a list of `memvals` in the original CompCert model), and puts these `smemvals` in the memory at the requested address.

Figure 5.1 shows the new type signatures of the memory operations, together with the type definition of `smemvals`. Note that the address given to `load` and `store` is really a

[1]Assuming a big-endian architecture. The endianness is a parameter in CompCert, instantiated differently depending on the target architecture.

[2]The list is reversed if needed, depending on the architecture.

```
Fixpoint concat (l : list smemval) : sval :=
match l with
| nil => 0
| a::r => (smv_to_sval a) + (concat r) << 8
end.

Definition decode (l: list smemval) (κ : memory_chunk) : sval :=
  from_bits κ (concat l).
```

Figure 5.4: Decoding `smemvals` into symbolic values

block identifier and an integer offset, as in CompCert, and not symbolic values. This is because the inner structure of CompCert memories is a map from block identifiers to arrays of bytes indexed by integer offsets. Consider the CompCert `loadv` and `storev` functions that take as input an address as a value v, and call `load` and `store` with the address (b, i) if $v = \texttt{ptr}(b, i)$ and fail otherwise. To adapt these functions so that they take symbolic values for the address, we need to include normalisations. The code of those functions is shown in Figure 5.5.

```
Definition loadv (κ: memory_chunk) (m: mem) (addr: sval) : option sval :=
  match normalise m addr with
  | Vptr b ofs => load κ m b ofs
  | _ => None
  end.

Definition storev (κ: memory_chunk) (m: mem) (addr: sval) (sv: sval)
                  : option mem :=
  match normalise m addr with
  | Vptr b ofs => store κ m b ofs sv
  | _ => None
  end.
```

Figure 5.5: The `loadv` and `storev` operations.

5.2 Good Variable Properties

CompCert's memory model exports an interface summarising all the properties of the memory operations necessary to prove the compiler passes. Those properties include so-called *good-variable properties* [Ler+14], and describe the behaviour of combinations of memory operations. For instance, the property `load_store_same` states that loading at an address that has just been written with some value v results in the same value v, converted to the appropriate chunk κ. The function `load_result` does this conversion. It consists of truncating integers to the expected size for chunks `Mint8xxx` and `Mint16xxx` and it is the identity function for other chunks. Formally, we have:

Theorem 5.2.1 (`load_store_same`).

$$\forall \kappa\ m\ b\ o\ v\ m', \texttt{store}\ \kappa\ m\ b\ o\ v = \lfloor m' \rfloor \Rightarrow \texttt{load}\ \kappa\ m\ b\ o = \lfloor \texttt{load\_result}\ \kappa\ v \rfloor.$$

Because we use symbolic values and delay their evaluation, this theorem does not hold anymore. This is illustrated by Example 5.2.1.

Example 5.2.1. *Consider* $\kappa =$ `Mint16unsigned`, $o = \mathit{int}(0)$ *and* $v = \mathit{int}(3735928559) = \mathit{int}(\texttt{0xDEADBEEF})$. *In* CompCert, *the* `store` *operation first encodes* v *into concrete bytes, keeping only the two least significant (because* $\kappa =$ `Mint16unsigned`*)* $b_1 = \texttt{0xBE}$ *and* $b_0 = \texttt{0xEF}$ *and stores them at addresses* $(b, 1)$ *and* $(b, 0)$ *(respectively). The* `load` *then decodes these two bytes and computes the resulting value* $v = \mathit{int}(b_1 \ll 8 + b_0) = \mathit{int}(\texttt{0xBEEF})$. *Applying* `load_result` *with* $\kappa =$ `Mint16unsigned` *to* v *results in the same integer* $\mathit{int}(\texttt{0xBEEF})$*, because it already fits in 2 bytes.*

In our model however, the behaviour is slightly different. The `store` *encodes each byte lazily,* i.e. *the addresses* $(b, 1)$ *and* $(b, 0)$ *do not contain concrete bytes but symbolic* ***smemvals*** *that denote them. Let* sv *be the symbolic value denoting the binary representation of value* v *for chunk* `Mint16unsigned`, i.e.

$$sv = \texttt{to\_bits Mint16unsigned}\ v = \texttt{convert Tint Tlong}\ v$$

For example, the location $(b, 1)$ *contains the* ***smemval*** (`Symbolic` sv 1) *that encodes byte number 1 of the binary representation of the original symbolic value* v*. The* `load` *first decodes* ***smemvals*** *into symbolic values, and then concatenates them to produce the final result. The* ***smemval*** (`Symbolic` $sv\ n$) *is decoded into* $(sv \gg (8 * n))\,\&\,\texttt{0xFF}$*. In our example we have* $sv_1 = (sv \gg 8)\,\&\,\texttt{0xFF}$ *and* $sv_2 = sv\,\&\,\texttt{0xFF}$*. The concatenation is again expressed as a symbolic value based on shifts. The result of the* `load` *is then equal to the concatenation of* sv_1 *and* sv_2*,* i.e. $L = (sv_1 \ll 8) + sv_2$*. On the other hand,* `load_result Mint16unsigned` v *amounts to zeroing the 2 highest bytes, resulting in the symbolic value* $v\,\&\,(2^{16} - 1)$*.*

The theorem `load_store_same` clearly does not hold for Example 5.2.1: the two sides of the equation are different symbolic values. However, they are *equivalent*, *i.e.* they always evaluate to the same value. This equivalence relation between symbolic values is noted $\equiv$ and is formally defined as follows:

Definition 5.2.1 (Equivalence of symbolic values).

$$sv_1 \equiv sv_2 \triangleq \forall\ cm\ im, [\![sv_1]\!]^{im}_{cm} = [\![sv_2]\!]^{im}_{cm}$$

We generalise `load_store_same` and every theorem of the memory model to use equivalence in lieu of syntactic equality when needed. We then state that there exists a symbolic value sv' that is the result of the `load` and this symbolic value is equivalent to the result we expect. The resulting theorems are of the form:

Theorem 5.2.2 (`load_store_same` with symbolic values).

$$\begin{aligned}\forall\ \ & \kappa\ m\ b\ o\ sv\ m', \texttt{store}\ \kappa\ m\ b\ o\ sv = \lfloor m' \rfloor \Rightarrow \\ & \exists sv', \texttt{load}\ \kappa\ m\ b\ o = \lfloor sv' \rfloor \land sv' \equiv \texttt{load\_result}\ \kappa\ sv.\end{aligned}$$

This generalisation is also needed for theorem `load_int64_split`, as shown below:

$$\begin{aligned}\forall\ \ & m\ b\ o\ sv,\ \texttt{load Mint64}\ m\ b\ o = \lfloor sv \rfloor \Rightarrow \\ & \exists\ sv_1\ sv_2,\ \texttt{load Mint32}\ m\ b\ o = \lfloor sv_1 \rfloor \land \\ & \qquad\qquad \texttt{load Mint32}\ m\ b\ (o + 4) = \lfloor sv_2 \rfloor \land \\ & \qquad\qquad sv \equiv \texttt{longofwords}(sv_1, sv_2)\end{aligned}$$

Theorem `load_int64_split` states that loading a value using the `Mint64` chunk (for `long`-typed values, on 8 bytes) can be simulated by loading two adjacent `Mint32` chunks (`int`-typed values, on 4 bytes) and concatenating the result of those reads with the `longofwords` operator. For the same reason as `load_store_same` (*i.e.* the decoding results in symbolic values), we had to generalise the theorem to use our equivalence relation instead of plain equality.

While the proof structure follows that of CompCert, the proof effort to port the whole memory model is non-negligible because we have to reason modulo equivalence of symbolic values.

5.3 Uninitialised Data As Indeterminate Values

We have included a constructor $\mathtt{indet}(l)$ in the constructors of symbolic values to represent uninitialised values. We can think of those indeterminate values as unknown variables. This construction has enabled us to reason about indeterminate values in Example 4.4.2. In this section, we explain how these values are used in the memory model, in particular how we make the connection between uninitialised data in C and indeterminate values in our model.

The idea is to initialise the contents of newly allocated blocks with $\mathtt{indet}(l)$ values. The location l is simply the address of the byte being initialised. In other words, when we allocate a new block b with bounds $[lo, hi[$, all the byte values stored at address (b, i), for all $i \in [lo, hi[$, are initialised with value $\mathtt{indet}(b, i)$. Using the location as an identifier of indeterminate value is a convenient way to assign each uninitialised value an independent variable (because block identifiers are never reused).

Note that giving names to indeterminate values models the assumption that two subsequent reads of the same uninitialised location result in the same arbitrary value, therefore enabling reasoning on such values, as illustrated by Example 5.3.1.

Example 5.3.1 (Evaluation of symbolic values with uninitialised values)**.** *Let b be a block corresponding to a freshly allocated variable `x` of type `char`. The contents of the cell at location $(b, 0)$ are initialised with $\mathtt{indet}(b, 0)$.*

In CompCert, the C expression `x - x` evaluates to $\mathtt{undef} - \mathtt{undef}$, which reduces to $\mathtt{undef}$. As a result, the semantics of the C program containing this expression is stuck.

By contrast, in CompCertS, the C expression `x - x` is first transformed into the symbolic value $\mathtt{indet}(b, 0) - \mathtt{indet}(b, 0)$. This symbolic value evaluates the same as the value $\mathtt{int}(0)$, because for any im:

$$[\![\mathtt{indet}(b,0) - \mathtt{indet}(b,0)]\!]^{im}_{cm} = im(b,0) - im(b,0) = 0$$

As a result, this symbolic value normalises into $\mathtt{int}(0)$ and the semantics of the C program does not get stuck on this expression.

5.4 Memory Allocation and Finite Memory

In CompCert, memory allocation always succeeds and returns a new block of the requested size. This makes the implicit assumption that the memory is infinite. This assumption is very convenient for the proof of correctness of CompCert because the memory consumption needs not be accounted for in the proof. However, the actual hardware has (obviously) a finite memory space, and a program that tries to use more memory than available will crash. This has the unsettling consequence that a program can be safely compiled by

CompCert into an assembly program that exhausts memory, and even though the source program has semantics and the semantic preservation applies, the compiled program may crash.

In CompCertS, because the semantics of the normalisation is based on concrete memories that map blocks to a finite 32-bit address space, we model a finite memory. As a result, our allocation function, `palloc`, is partial (hence the `p` in the name of the function) and may fail (see Figure 5.1) when no more memory is available. More precisely, `palloc` only succeeds when it can build a concrete memory for the resulting abstract memory. Besides, `palloc` is designed in such a way that we can prove that all abstract memories satisfy Property 4.4.1 (Sliding Blocks), needed for the well-behavedness of the normalisation (see Section 4.4).

In this section, we first describe the algorithm used to decide whether an allocation is possible, then we describe how we can derive from this partial allocation function that all abstract memories satisfy Property 4.4.1, thus making this memory model fit into the framework of the normalisation function exposed in Chapter 4.

5.4.1 Allocation Algorithm

The implementation of `palloc` is shown in Figure 5.6. Let us examine the code of the different functions.

```
Fixpoint alloc_blocks (bl : list (block * Z)) (next_available: Z)
                        (cur : block -> Z) : (Z * (block -> Z)) :=
  match bl with
  | nil => (next_available, cur)
  | (b,sz)::l => alloc_blocks l (align next_available 2^MA + sz)
                 cur[b ↦ align next_available 2^MA]
  end.

Definition size_mem (bl : list (block * Z)) : Z :=
 fst (alloc_blocks bl 2^MA (λb => 0))

Definition can_alloc (m: mem) (sz: Z) (al: Z) : bool :=
  let b := fresh_block m in
  let size := size_mem ((b,sz)::blocks_of m) in
  alignment_of_size sz <= al <= MA && size < Int.max_unsigned - 2^MA.

Definition palloc (m: mem) (sz: Z) (al: Z) : option (mem * block) :=
  if can_alloc m sz al
  then ⌊ set_alignment (alloc m 0 sz) al ⌋
  else ∅.
```

Figure 5.6: Definition of the new allocation operation

Compared to the existing `alloc` function, `palloc` takes an additional argument `al` which specifies the alignment of the block, *i.e.* the number of trailing bits guaranteed to be zeros. To decide whether it is possible to allocate the block, `palloc` is guarded by the predicate `can_alloc`. If the predicate holds, the allocation succeeds, calls the existing CompCert allocation operation `alloc` and records the alignment with the `set_alignment` function. Otherwise, the allocation fails.

The `can_alloc` predicate checks two properties, detailed in the following paragraphs.

Valid alignment constraint. The first property to be checked is that the requested alignment is at least as large as the minimal alignment computed by `alignment_of_size`, and not larger than a maximal alignment `MA`. The whole development is parametric in `MA`, with the constraint that `MA` should be greater than or equal to 3 (the maximal alignment requested by CompCert *e.g.* for `long`-typed variables). For programs which do not explicitly perform dynamic memory allocation, *i.e.* the CompCert alignment is the only one required, the value of `MA` can be set to 3. For programs using `malloc`- or `mmap`-allocated blocks, `MA` would typically be the alignment of a kernel page (*i.e.* 12 for pages of 4Ko).

Existence of a valid concrete memory. The second property to be checked is that there exists a concrete memory valid for the abstract memory obtained after performing the allocation of the new block. This property is checked in a constructive way, *i.e.* we run an algorithm that attempts to construct such a concrete memory. The `palloc` function will succeed when the algorithm succeeds, and fail otherwise. Using the `size_mem` function, function `can_alloc` computes the size of the memory composed of the blocks of m plus the new block to be allocated. The size of the memory is defined as the first fresh address in a concrete memory where all blocks are maximally ($2^{\texttt{MA}}$-byte) aligned. `size_mem` takes as input a list of pairs (b_i, sz_i) where sz_i is the size in bytes of block b_i. The resulting size `size` can be seen as an address such that all the blocks can be allocated below `size` at addresses that are $2^{\texttt{MA}}$-byte aligned. The predicate `can_alloc` holds only if there are still $2^{\texttt{MA}}$ reserved bytes above `size`. As we shall see, this reserved space will be necessary to ensure Property 4.4.1.

The size of the memory is recursively computed by `alloc_blocks`. It allocates each block at the next available ($2^{\texttt{MA}}$-byte aligned) address and returns the next available address and the constructed concrete memory. It takes as arguments two accumulators: `next_available` and `cur`. The accumulator `next_available` is the next available address and `cur` is the concrete memory currently being constructed. The initial values of these accumulators are given in the `size_mem` function: the first available aligned address is $2^{\texttt{MA}}$, and the initial concrete memory is $\lambda b.0$, *i.e.* it maps every block to the address 0.

This notion of memory size will be used in the rest of this thesis as a way to give some guarantees about the memory usage of programs. In particular, we force CompCertS to reduce the memory usage of programs, and the final theorem of the compiler accounts for this resource usage. It would be valid for CompCert to transform any program into one that first exhausts memory (by allocating many blocks) and then perform a faithful program compilation, because CompCert does not model out-of-memory behaviours. Our model forbids such abnormal compilations because the memory usage is forced to decrease with compilation. This is an improvement over the theorem of CompCert.

5.4.2 Allocation Properties

The specification of the normalisation is well-behaved only under the conditions of Property 4.4.1 (see Section 4.4). It states that for any memory m, it is possible to rearrange the blocks so that there always exist two concrete memories which only differ on a single block. In other words, any block b can be found at least two valid concrete addresses. We show in Theorem 5.4.1 that this is a property of the allocation algorithm presented above.

The *memory* type in the Coq development of CompCert is a dependent record, which holds on one hand data structures that model the memory and on the other hand proof of well-formedness properties on those data structure. In order to prove the Sliding Blocks property for every memory, we add the property to the dependent record that represents the memory state. This way, we have a proof that every constructed memory state satisfies the property, because it becomes a typing constraint of the *memory* type. As we shall see, this generates proof obligations for every function that returns a memory state: one must prove that the resulting memory actually satisfies Property 4.4.1.

Theorem 5.4.1 (Sliding Blocks). *Every memory m is such that for any block b, there exist at least two valid concrete memories cm and cm' that allocate b at different concrete addresses while allocating all the other blocks at the same addresses. Formally,*

$$\forall\ b, \quad \exists cm, cm', \bigwedge \begin{cases} cm \vdash m \wedge cm' \vdash m \\ cm(b) \neq cm'(b) \\ \forall b' \neq b, cm(b') = cm'(b') \end{cases}$$

Proof. As discussed above, the property is part of the memory type. Hence, once a memory m has been constructed, the proof of the claim of Theorem 5.4.1 is for free. The real proof is disseminated in the proof obligations of each operation on memory states, *i.e.* the proof that the initial memory state m_0 satisfies the property, and the proof that starting from a memory state m that satisfies the property, the memory obtained by `store`, `free` and `palloc` still satisfy the property.

- For the initial memory m_0, as there are no allocated blocks, all the concrete memories are valid. Given a block b, we can therefore construct cm and cm' such that $cm = (\lambda x.0)[b \mapsto 1]$ and $cm' = (\lambda x.0)[b \mapsto 2]$. Hence, the property holds for m_0.

- Suppose a memory m_2 obtained after performing a `store` in some memory m_1, for which the property holds. Since m_1 and m_2 have the same set of valid concrete memories, and the property doesn't depend on the contents of the memory blocks but only on their structure, the property holds.

- Suppose a memory m_2 obtained after performing a `free` in some memory m_1, for which the property holds. Since every valid concrete memory of m_1 is also a concrete memory of m_2, the property holds.

- Suppose that a memory m is obtained by the allocation function `palloc`. The algorithm in `palloc` checks that all the blocks fit in memory by running the function `size_mem` which constructs as witness a valid concrete memory cm and returns the first fresh address `addr`. A key insight of the proof is that the order of the blocks is not relevant for the success of `palloc`. The argument is illustrated by Figure 5.7 and goes as follows. If the `alloc_blocks` function followed a *first fit* allocation dis-

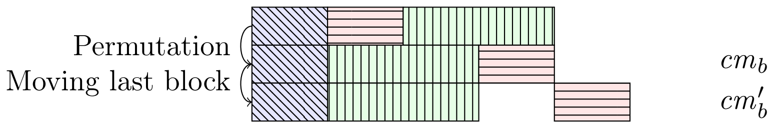


Figure 5.7: Construction of two concrete memories for Property 4.4.1

cipline, the alignment constraints could have an impact on the fragmentation of the witness concrete memory and therefore `palloc` could succeed or fail depending on the order the blocks are allocated. To prevent this, all the addresses computed by `alloc_blocks` are maximally aligned. Therefore, the success of the allocation is independent of the allocation order, we can therefore choose any permutation for the order of the blocks. Hence, without loss of generality, for every block b, we can construct $cm_b \vdash m$ such that block b is allocated last. In Figure 5.7, we consider b to be the light red block with horizontal lines.

Moreover, the test `addr < Int.max_unsigned - ` $2^{\texttt{MA}}$ ensures that the last block, say b, can also be allocated at $cm_b(b) + 2^{\texttt{MA}}$. This constructs a second concrete memory cm'_b, as depicted in the last line of Figure 5.7: block b simply needs to be shifted by $2^{\texttt{MA}}$ bytes from its position in cm_b.

Hence the property holds for any memory state m and any block b. □

5.5 Conclusion and Discussion

In this chapter, we have shown how to introduce the notion of symbolic values into the memory model of CompCert, allowing to express the result of low-level operations on pointers and on uninitialised data.

Loading from and storing to the memory. The `load` and `store` operations operations have been adapted to operate on symbolic values: symbolic values are read from and written to memory. The accessed address may also be given as a symbolic value, in which case it needs to be normalised into a genuine pointer before actually performing the access. This results in a more relaxed semantics than that of CompCert because the locations to be read from or written to may be computed with low-level operations, provided that the computation yields a unique location at the time of dereference.

It would be interesting to investigate the case of a *fully symbolic memory*, in which the inner structure is no longer a concrete map from block identifiers to contents but a symbolic map where keys need not be concrete values. A fully symbolic memory state would be a sequence of symbolic stores (*i.e.* a store at a symbolic address, that needs not evaluate to a unique location), and we would extend our domain of symbolic values to include symbolic loads in a given symbolic memory.

This would enable to give semantics to C programs that use hash of pointers as indices in arrays. See for example function `hash_ptr` from the source code of the Linux kernel[3], which computes the hash of a pointer using bitwise shift operators. Now consider we use this hash to index an array, as in `table[hash_ptr(ptr,n)]`. This hash depends on the concrete bit-representation of the pointer `ptr` and is likely not to evaluate the same in every valid concrete memory. Hence in our model with normalisations, the array access will fail because the address can not be normalised to a unique location. By contrast, with a fully symbolic memory, we would be able to perform the store symbolically and retrieve the value stored symbolically. However, this would not suffice to give semantics to accesses in hash maps where keys (*i.e.* hashes of pointers) need to be compared. Indeed, with a deterministic semantics, we would still need to normalise the guard of a conditional branch to a unique value to continue the execution of the program. Treating hash maps would require a non-deterministic semantics, as noted by Kang *et al.* [Kan+15]. In the remainder of this thesis, we do not investigate further this idea and keep our deterministic model with normalisations before memory accesses.

[3]See `https://github.com/torvalds/linux/blob/master/include/linux/hash.h#L71`.

Allocation in a finite memory. The `alloc` operation has been profoundly modified. First, it initialises the contents of every allocated block with `indet(`l`)` values, making it possible to reason about accesses to uninitialised data.

Second, it is now a partial function (hence the `p` in `palloc`). The `palloc` function only succeeds when it can build a valid concrete memory for the abstract memory after allocation. To do so, it runs an eager algorithm that allocates blocks at maximally aligned addresses. As a consequence, blocks may be allocated in any order, which allows to prove Theorem 5.4.1, which states that Property 4.4.1 (Sliding Blocks) holds for every abstract memory m. This is a significant result because this was stated as an hypothesis for the well-behavedness of the normalisation function in Section 4.4.

Third, it assigns alignment constraints to blocks. These alignment constraints are not arbitrary but must be contained within a lower bound that depends on the size of the considered block and a maximal alignment `MA`. This maximal alignment needs to be larger than 3, which is the maximal alignment already considered in CompCert, even though in CompCert the alignment is a property of an offset within a block, and in our model it becomes a property of a block in a concrete memory. For programs that do not perform dynamic allocation of memory, `MA` can be set to 3. For other programs that need to express stronger alignment constraint, we set `MA` to 12 which is the alignment required for pages of 4Ko (as returned by `mmap` for example). However, since all blocks have to be maximally aligned for the success of `palloc`, some programs with many small variables may fail to be given semantics because of a too large `MA` when a reasonable alternative would be possible: our algorithm would behave in a too conservative fashion. To mitigate this issue, a possibility would be to split the memory space into two distinct parts: the *stack* and the *heap*. The stack only requires 8-byte alignment because it only contains statically allocated blocks already present in CompCert. However, the heap may require 2^{12}-byte alignments. This solution is more relaxed than the existing one because only the blocks in the heap would require 2^{12}-byte alignment, therefore *wasting* less memory space than the existing solution.

As a result of this finite memory model, we are now able to account for memory consumption. In particular, the definition of the size of the memory, *i.e.* the first unallocated concrete address in a concrete memory in which all blocks are maximally aligned, will be used in the rest of this thesis to prove that compiled programs use *less* memory than source programs, in the sense of the `size_mem` function.

Chapter 6

More Defined Semantics For CompCert

In Chapter 5, we have shown how to adapt the memory model of CompCert using symbolic values. This chapter lifts these modifications from the memory model to the semantics of the different languages used in CompCert, including CompCert C, the assembly language for x86 and all the intermediate languages used during the compilation.

The semantics of these languages are of capital importance, especially those of CompCert C and assembly, because they are part of the *trusted computing base* of the whole compiler. The semantic preservation theorem is stated with respect to these semantics. In other words, a bug in the semantics may invalidate the correctness of the whole compiler because it does not accurately represent the real world C and assembly languages. We therefore put a lot of care into this adaptation of the semantics.

This chapter is organised as follows. First, we explain in Section 6.1 how we adapt the semantics of all the intermediate languages, and identify patterns that need to be adapted similarly in several languages. Then, in Section 6.2 we perform a cross-validation of CompCert's semantics and ours, whose aim is to strengthen our confidence both in our novel semantics and in CompCert's. This results in the discovery of bugs both in CompCert and in preliminary versions of our semantics. The C interpreter that ships with CompCert is an executable semantics of C, that needs an executable normalisation: Section 6.3 explains how the normalisation is implemented with the help of a SMT solver. Finally, Section 6.4 reports on the experimental evaluation of this executable semantics and details several low-level idioms that we are now able to give semantics to.

6.1 Updating The Semantics Of CompCert's Languages

This section presents the modifications that we apply to the semantics of all intermediate languages. We will see that the changes are relatively small and they are mostly the same for every language.

Symbolic values instead of values. The first obvious change is to use symbolic values (*sval*) everywhere values (*val*) are used in CompCert. For example, the semantics of the Clight language uses an environment for temporaries called `temp_env` and defined as `PTree.t val`, where `PTree.t A` is the type of maps indexed by `positive` numbers (the type `positive` in Coq represents bitvectors of arbitrary length) and whose content is of type `A`. We adapt this definition to use symbolic values instead of values, *i.e.* we redefine `temp_env` as `PTree.t sval`.

Evaluation of expressions into symbolic values. The evaluation of expressions in the front-end of CompCert is typically expressed by a relation between expressions and CompCert values. This predicate is also parametrised by a memory state and an environment whose type we note E here (it varies accross different languages). For example, in C♯minor, the evaluation of expressions (of type `expr`) is formalised by a predicate `eval_expr : mem -> E -> expr -> val -> Prop`. In our CompCertS semantics for C♯minor, the predicate associates symbolic values to expressions, *i.e.* we have `eval_expr : mem -> E -> expr -> sval -> Prop`. The evaluation of expressions is mostly a translation to symbolic values, and no computation happens. For example, the following shows the rules for the addition expression. Rule Eval-Add shows the rule as it is in CompCert, while Rule Eval-Add-Symb shows how we adapt the rule. The main difference, apart from the fact that symbolic values are used instead of values, is that the `Val.add` function actually computes on values, while the `OpAdd` symbolic operator merely constructs a symbolic value.

Eval-Add

$$\frac{\texttt{eval\_expr}\ m\ E\ e_1\ v_1 \qquad \texttt{eval\_expr}\ m\ E\ e_2\ v_2}{\texttt{eval\_expr}\ m\ E\ (e_1 + e_2)\ (\texttt{Val.add}\ v_1\ v_2)}$$

Eval-Add-Symb

$$\frac{\texttt{eval\_expr}\ m\ E\ e_1\ sv_1 \qquad \texttt{eval\_expr}\ m\ E\ e_2\ sv_2}{\texttt{eval\_expr}\ m\ E\ (e_1 + e_2)\ (sv_1\ \texttt{OpAdd}\ sv_2)}$$

C♯minor expressions also include `Eload` expressions, whose purpose is to fetch some content from the memory. We show below the rules for the evaluation of `Eload` expressions in CompCert (Rule Eval-Eload) and in CompCertS (Rule Eval-Eload-Symb). Here, there is no visible difference in the rules, however keep in mind that the `loadv` operation involves a normalisation in our setting (see Section 5.1).

Eval-Eload

$$\frac{\texttt{eval\_expr}\ m\ E\ e_{addr}\ v_{addr} \qquad \texttt{loadv}\ \kappa\ m\ v_{addr} = \lfloor v \rfloor}{\texttt{eval\_expr}\ m\ E\ (\texttt{Eload}\ \kappa\ e_{addr})\ v}$$

Eval-Eload-Symb

$$\frac{\texttt{eval\_expr}\ m\ E\ e_{addr}\ sv_{addr} \qquad \texttt{loadv}\ \kappa\ m\ sv_{addr} = \lfloor sv \rfloor}{\texttt{eval\_expr}\ m\ E\ (\texttt{Eload}\ \kappa\ e_{addr})\ sv}$$

Functions expecting pointers. Some functions or predicates expect locations as parameters, split between a block identifier and an offset. For instance, in the semantics of C and Clight, the `deref_loc` predicate has the following type signature:

```
deref_loc: type -> mem -> block -> int -> val -> Prop.
```

A derivation of `deref_loc ty m b o v` can be understood as follows: reading a value of type ty in memory m at location (b, o) results in value v. We adapt the type signature to symbolic values:

```
deref_loc: type -> mem -> sval -> sval -> Prop.
```

The various constructors of `deref_loc` are adapted to symbolic values but the logic of this predicate from CompCert is preserved. We recall the different cases here, and in Figure 6.1. Depending on the C type, the `access_mode` function dictates whether the access is to be performed *by value*, *by reference* or *by copy*. If the type is a scalar type, the access mode is *by value* and the value is directly fetched from the memory (see Rule Deref-Loc-Value). If the type is an array type or a function pointer type, the access mode is *by reference* and the `deref_loc` predicate simply relates the symbolic value representing the location with itself (Rule Deref-Loc-Copy). If the type is a structure or union type, the access mode is *by copy*, and the `deref_loc` predicate is similar to the case of *reference* accesses.

$$\textsc{Deref-Loc-Value}\quad \frac{\texttt{access\_mode}(ty) = \texttt{By\_value}\ \kappa \qquad \texttt{loadv}\ \kappa\ m\ sv_{ptr} = \lfloor sv \rfloor}{\texttt{deref\_loc}\ ty\ m\ sv_{ptr}\ sv}$$

$$\textsc{Deref-Loc-Copy}\quad \frac{\texttt{access\_mode}(ty) \neq \texttt{By\_value}\ \kappa}{\texttt{deref\_loc}\ ty\ m\ sv_{ptr}\ sv_{ptr}}$$

$$\textsc{Assign-Loc-Value}\quad \frac{\texttt{access\_mode}(ty) = \texttt{By\_value}\ \kappa \qquad \texttt{storev}\ \kappa\ m\ sv_{dst}\ sv = \lfloor m' \rfloor}{\texttt{assign\_loc}\ ty\ m\ sv_{dst}\ sv\ m'}$$

$$\textsc{Assign-Loc-Copy}\quad \frac{\begin{array}{c}\texttt{access\_mode}(ty) = \texttt{By\_copy} \\ \texttt{normalise}(m, sv_{src}) = \texttt{ptr}(b_{src}, i_{src}) \\ \texttt{loadbytes}\ m\ b_{src}\ i_{src}\ (\texttt{sizeof}(ty)) = \lfloor mvals \rfloor \\ \texttt{normalise}(m, sv_{dst}) = \texttt{ptr}(b_{dst}, i_{dst}) \\ \texttt{storebytes}\ m\ b_{dst}\ i_{dst}\ mvals = \lfloor m' \rfloor\end{array}}{\texttt{assign\_loc}\ ty\ m\ sv_{dst}\ sv_{src}\ m'}$$

Figure 6.1: Memory access predicates

The symmetric predicate, `assign_loc`, relates an input memory state and an output memory state where a store has taken place. The original predicate has the following type signature:

```
assign_loc: type -> mem -> block -> int -> val -> mem -> Prop.
```

A derivation of `assign_loc` $ty\ m\ b\ o\ v\ m'$ can be understood as follows: starting from a memory state m, performing a store of type ty at location (b, o) of value v yields the memory state m'. We adapt the type signature to symbolic values in a similar way as we did for `deref_loc`:

```
assign_loc: type -> mem -> sval -> sval -> mem -> Prop.
```

The predicate distinguishes two cases depending on `access_mode`(ty). The cases are shown in Figure 6.1. If the access is to be performed by value (*i.e.* the type ty is a scalar type), then the predicate simply models the effect of a `storev` operation (Rule Assign-Loc-Value). If the access is to be performed by copy, a byte-wise copy is performed, using the `loadbytes` and `storebytes` operations of the memory model (Rule Assign-Loc-Copy).

Memory accesses. Normalisations must be introduced before memory accesses, as stated in Section 5.1. However, most memory accesses are performed through the use of the `loadv` and `storev` operations, which already include the normalisations. Hence, we need not worry about those in the semantics of languages.

Conditional branches. The semantics of `if (..) { .. } else { .. }` statements consists of two rules in the semantics of all languages. The rules shown below are not tied to a specific language but are representative of the constructs present in most intermediate languages of CompCert. We assume a semantic state made only of the instruction to be executed and a memory state, written $\langle S, m\rangle$ for the state where S is the instruction to execute and m is the current memory state. Languages typically have a more complex semantic state, however only the parts shown here are common to all languages. Similarly, `eval_expr` evaluates expressions in the considered language and is different in every language. In CompCert's semantics, the guard is first evaluated into an integer. If the integer is 0, the semantics is that of the `else` block (rule If-false); otherwise it is that of the `then` block (rule If-true).

$$\textsc{If-true}\quad \frac{\texttt{eval\_expr}\ m\ E\ b\ =\ \texttt{int}(i) \qquad i \neq 0}{\langle \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2,\ m\rangle \to \langle s_1,\ m\rangle} \qquad \textsc{If-false}\quad \frac{\texttt{eval\_expr}\ m\ E\ b\ =\ \texttt{int}(0)}{\langle \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2,\ m\rangle \to \langle s_2,\ m\rangle}$$

In our model, because C expressions evaluate to symbolic values, the guards of conditional statements are evaluated to symbolic values, and a normalisation is needed to obtain an integer which dictates which branch of the conditionnal is executed (see rules If-true-symb and If-false-symb).

$$\textsc{If-true-symb}\quad \frac{\texttt{eval\_expr}\ m\ E\ b\ =\ sv \qquad \texttt{normalise}\ m\ sv = \texttt{int}(i) \qquad i \neq 0}{\langle \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2,\ m\rangle \to \langle s_1,\ m\rangle}$$

$$\textsc{If-false-symb}\quad \frac{\texttt{eval\_expr}\ m\ E\ b\ =\ sv \qquad \texttt{normalise}\ m\ sv =\ \texttt{int}(0)}{\langle \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2,\ m\rangle \to \langle s_2,\ m\rangle}$$

Lazy operators. In the semantics of CompCert C, the input language of CompCert, more constructs require normalisations in their semantics. For example, the sequential AND `&&` and sequential OR `||` operators and the ternary condition `a ? b : c` need normalisations to encode the lazy behaviour of these operators. While it may seem counterintuitive that more work is needed to encode laziness, the normalisation of the left-hand-side is needed so that the right-hand-side is evaluated (with its potential side effects) only if necessary – hence the laziness. In particular, the right-hand-side is evaluated only when the left-hand-side is `true` (for `&&`) or `false` (for `||`).

Return value of programs. Finally, in all languages, the state of a program at the end of its execution is represented as a so-called *return state* containing, among others, a return value (an integer) and a memory state. We change the type of return values into symbolic values. Still, to be compatible with all the formal results about formal semantics and simulation arguments, we require that the symbolic value used as return value of programs be normalisable into an integer (because the return value of a program is always an integer).

This sums up all the places where normalisations had to be introduced. An important thing we have realised through the process of transforming semantics is that the normalisations must be introduced at the same places in the different semantics so that the semantics stay consistent, thus the proofs can be easily adapted.

One way to think about these extended semantics is the following: we allow some kind of non-deterministic reasoning as long as it doesn't affect the control flow or memory accesses. At those points, we demand that the non-determinism is resolved, *i.e.* all paths converge, *i.e.* the result of evaluating a symbolic value is independent from the precise layout of the memory.

6.2 Our Semantics Is A Refinement Of CompCert's

The semantics of the CompCert C language is part of the *trusted computing base* of the compiler. Any modelling error can be responsible for a buggy, though formally verified, compiler. It is therefore crucial to ensure that the semantics is accurate. To detect a glitch in the semantics, a first approach consists in running tests and verifying that the CompCert C interpreter computes the expected value. With this respect, the CompCert C semantics successfully runs hundreds of random test programs generated by CSmith [Yan+11]. Another indirect but original approach consists in relating formally different semantics for the same language. For instance, when designing the Clight semantics, several equivalences between alternate semantics were proved to validate this semantics [Bla07]. Our new memory model with symbolic values is a new and interesting opportunity to apply this methodology. We will see that, since our model is built on a notion of concrete memory, which is lower-level, we are able to detect incorrect assumptions in CompCert's semantics. In the following, we first describe the cross-validation of the Clight semantics that we performed, then we explain the errors that we discovered during the process of doing the proof.

6.2.1 Forward simulation between CompCert Clight and CompCertS Clight

The cross-validation proof that we perform is a forward simulation between CompCert Clight (CClight) and CompCertS Clight (SClight). That is, whenever a program has defined semantics in CClight, it will have the same semantics in SClight. We prove a *lock-step simulation*, as illustrated by Figure 6.2. Assuming a relation $\mathcal{R}$, we must prove that starting from two matching states σ_1 and σ_2 (in the sense of $\mathcal{R}$), if a step is possible from σ_1 to σ_1' in CClight, then it is possible to take a step from σ_2 to σ_2' in SClight, such that σ_1' and σ_2' are matching states (in the sense of $\mathcal{R}$).

$$\begin{array}{ll}\forall & \sigma_1 \in \Sigma_1,\ \sigma_2 \in \Sigma_2, \\ & \sigma_1\ \mathcal{R}\ \sigma_2 \Rightarrow \sigma_1 \xrightarrow{\tau}_1 \sigma_1' \Rightarrow \\ & \exists \sigma_2',\ \sigma_2 \xrightarrow{\tau}_2 \sigma_2' \ \wedge\ \sigma_1'\ \mathcal{R}\ \sigma_2'\end{array}$$

$$\begin{tikzcd}
\text{CClight} & \sigma_1 \arrow[r, dash, "\mathcal{R}"] \arrow[d, "\tau"'] & \sigma_2 \arrow[d, dashed, "\tau"] & \text{SClight} \\
& \sigma_1' \arrow[r, dash, dashed, "\mathcal{R}"'] & \sigma_2' &
\end{tikzcd}$$

Figure 6.2: Simulation for the cross-validation of the semantics of Clight

Of course, since the memory in SClight is finite and that in CClight is infinite, this simulation will only hold when the CClight program does not exhaust the memory space. Thus, we perform the proof under the hypothesis that our allocation function never fails. This is a reasonable assumption: we *expect* our semantics to be less defined than that of CompCert in those *out-of-memory* situations.

To prove the simulation, we need to define an invariant `match_states` (represented as $\mathcal{R}$ in Figure 6.2) that relates CClight and SClight program states and that is preserved at every step of the semantics. This invariant is built on top of a relation `match_val` that relates CompCert values and symbolic values. We show in Example 6.2.1 a C program that we execute both with CClight and SClight semantics. We will then discuss our choice for the `match_val` relation.

Example 6.2.1. *Consider the following C program:* `int i; return (&i != 0)` *. It tests whether a valid pointer is different from* `NULL`*. We are interested in the return value of this program. We assume that variable* `i` *is allocated in block b. In CClight, the C expression is transformed into* $\mathtt{ptr}(b,0)! = \mathtt{int}(0)$*, which in turn evaluates to* `true`*,* i.e. $\mathtt{int}(1)$*. In SClight, we merely build the symbolic value* $\mathtt{ptr}(b,0)! = \mathtt{int}(0)$*.*

A natural candidate for `match_val v sv` is that v must be the normalisation of sv, *i.e.* $sv \xrightarrow{m} v$. However, this requires parameterizing `match_val` with a memory state and proving that all memory operations preserve `match_val`. As a matter of fact, the `free` operation does not preserve the normalisation. For example, consider m the memory state of the program before returning its result. The symbolic value $\mathtt{ptr}(b,0)! = \mathtt{int}(0)$ normalises to $\mathtt{int}(1)$ in m. However, if we call m' the memory state obtained after freeing block b from memory m, then the same symbolic value does not normalise in m' because $\mathtt{ptr}(b,0)$ is no longer valid. This is in accordance with the C standard[1] but a loss of completeness with respect to the existing CompCert semantics.

For the sake of the proof, we adapt the semantics of SClight to avoid this situation. The solution is to normalise symbolic values in a more eager manner *i.e.* before any write into memory or into a register, and only keep symbolic values when the normalisation fails. This is performed by the function `simplify`:

Definition 6.2.1. $\mathtt{simplify}\ m\ sv \ := \ \mathtt{if\ normalise}\ m\ sv = \mathit{undef}$ $\mathtt{then}\ sv\ \mathtt{else\ normalise}\ m\ sv.$

Back to our example, after introducing the simplifications, the `match_val` relation needs to relate $\mathtt{int}(1)$ and the simplification of $\mathtt{ptr}(b,i)! = \mathtt{int}(0)$, *i.e.* $\mathtt{int}(1)$. We define `match_val` as follows:

Definition 6.2.2. $\mathtt{match\_val}\ v\ sv \ := \ \forall\ cm\ im,\ [\![v]\!]_{cm}^{im} \leq [\![sv]\!]_{cm}^{im}.$

We use $\leq$ instead of equality to account for the fact that SClight gives semantics to more programs than CClight, *i.e.* `undef` in CClight can be matched with any symbolic value in SClight.

A large part of the simulation proof is the preservation of C operators. That is, in a memory m, for any operation *op* that produces a value v in CClight, the same operation will produce a symbolic value sv, such that `match_val` v (`normalise` m sv). Indeed, if CClight produced a value $v \neq$ `undef`, then we must normalise it into the same value. This is stated formally in Lemma 6.2.1. The `sem_binop` function gives the CompCert semantics of a binary operator *op* applied to values v_1 of type t_1 and v_2 of type t_2. It is parameterised by a function `valid` m that takes a location (b,i) and returns *true* if and only if the location (b,i) is valid in memory m. This is needed for example for the semantics of pointer comparisons (see Figure 2.5). The function `sem_binop_sval` mimics the signature of `sem_binop` except that symbolic values replace values and it does not need information about the validity of pointers when constructing the symbolic values.

[1][ISO11][§6.2.4.2]: The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

Lemma 6.2.1 (`expr_binop_preserved`).

$$\begin{aligned}\forall\ & op\ m\ v_1\ sv_1\ v_2\ sv_2\ t_1\ t_2\ v,\ \texttt{match\_val}\ v_1\ sv_1\ \Rightarrow\ \texttt{match\_val}\ v_2\ sv_2 \Rightarrow\\ & \texttt{sem\_binop}\ op\ v_1\ t_1\ v_2\ t_2\ (\texttt{valid}\ m) = \lfloor v \rfloor \Rightarrow\\ & \exists sv, \texttt{sem\_binop\_sval}\ op\ sv_1\ t_1\ sv_2\ t_2 = \lfloor sv \rfloor \wedge \texttt{match\_val}\ v\ (\texttt{simplify}\ m\ sv).\end{aligned}$$

This can be pictured as a diagram close to simulation diagrams, as in Figure 6.3. Plain lines represent hypotheses, and dashed lines represent conclusions. The `match_val` relation is depicted by $\mathcal{R}$. The proof of this lemma (and its sibling about unary operators) is a copious case analysis on the considered operator *op*. The existence of *sv* as the result of `sem_binop_sval` is only dependent on the types t_1 and t_2, and not on the actual symbolic values. Then, the semantics of every operator follow a similar structure: depending on the type of the operands, we perform different computations on the inputs (symbolic) values. For example, the semantics of the addition operator distinguishes the following cases: addition of an integer to a pointer, addition of a long to a pointer or addition of two scalars of the same type (*i.e.* two integers, two longs, two floats, ...). Our semantics follows the same structure, which makes it easy to compare the two semantics. Then, yet another case analysis on v_1 and v_2 is necessary to relate the result v in COMPCERT and the symbolic value sv that we construct.

$$\begin{tikzcd}
v_1, v_2 \arrow[rr, leftrightarrow, "\mathcal{R}"] \arrow[d, "\texttt{sem\_binop}"'] & & sv_1, sv_2 \arrow[d, dashed, "\texttt{sem\_binop}"] \\
v \arrow[rr, leftrightarrow, dashed, "\mathcal{R}"'] & & sv
\end{tikzcd}$$

Figure 6.3: `expr_binop_preserved` as a simulation diagram.

6.2.2 An opportunity to discover bugs

The high-level intuition of why Lemma 6.2.1 is true is that whenever COMPCERT succeeds in evaluating an expression, we should succeed as well. Situations where this is not the case are likely bugs. During the proof, we encountered several such issues.

First, we made some silly mistakes in the evaluation of symbolic values: a particular cast operator was mapped to the wrong syntactic constructor. While this is a benign error and easy to fix (just map the correct syntact constructor), this shows how useful this cross-validation is: it enables us to detect errors.

Pointers one-past-the-end. This is also during the proof that we have identified the issue of weakly valid pointers and therefore have excluded $2^{32} - 1$ from the address space (see Section 4.3). Indeed, in early versions of our development, we did not think of pointers one-past-the-end. As a consequence, such pointers could be assigned the concrete address 0, resulting in inconsistent behaviour with respect to COMPCERT. Consider for example Lemma 6.2.1 in the case where the operator *op* is the comparison operator $<$ (less than) and v_1 and v_2 are pointers. Consider the particular case where $v_1 = \texttt{ptr}(b, o_1)$ and $v_2 = \texttt{ptr}(b, o_2)$, *i.e.* both pointers are in the same block. This particular case of Lemma 6.2.1 is shown in Lemma 6.2.2.

Lemma 6.2.2 (`expr_binop_preserved` (particular case))**.**

$$\forall\ m\ b\ o_1\ o_2\ sv_1\ sv_2,\ \texttt{match\_val}\ (\mathtt{ptr}(b,o_1))\ sv_1 \Rightarrow \texttt{match\_val}\ (\mathtt{ptr}(b,o_2))\ sv_2 \Rightarrow \\ \texttt{weakly\_valid}(m,b,o_1) \Rightarrow \texttt{weakly\_valid}(m,b,o_2) \Rightarrow \\ \texttt{match\_val}\ (\texttt{of\_bool}\ (o_1 < o_2))\ (\texttt{simplify}\ m\ (sv_1\ (\texttt{OpCmp Lt})\ sv_2))$$

The `of_bool` function takes a boolean b and returns $\mathtt{int}(1)$ if b is `true` and $\mathtt{int}(0)$ if b is `false`. The $<$ symbol models the addition of 32-bit integers, *i.e.* it may overflow. The symbolic operator `OpCmp Lt` models the $<$ comparison on symbolic values. We write it with its full name instead of the $<$ symbol to avoid confusion with the comparison on integers. We must prove that $(sv_1\ (\texttt{OpCmp Lt})\ sv_2)$ normalises in m into $\texttt{of\_bool}\ (o_1 < o_2)$, which can be rewritten conveniently into a property about machine integer arithmetic:

$$\forall\ cm\ im,\ [\![\texttt{of\_bool}\ (o_1 < o_2)]\!]_{cm}^{im} = [\![(sv_1\ (\texttt{OpCmp Lt})\ sv_2)]\!]_{cm}^{im}$$
$$\Leftrightarrow$$
$$\forall\ cm\ im,\ \texttt{of\_bool}\ (o_1 < o_2) = \texttt{of\_bool}\ ([\![sv_1]\!]_{cm}^{im} < [\![sv_2]\!]_{cm}^{im})$$
$$\Leftrightarrow$$
$$\forall\ cm\ im,\ (o_1 < o_2) = (cm(b) + o_1 < cm(b) + o_2)$$

This last property holds if the computations $cm(b) + o_i, \forall\ i \in \{1,2\}$ do not overflow, *i.e.* $0 \le cm(b) + o_i \le 2^{32} - 1$. To prove the theorem, we need that every weakly valid location verifies this property, *i.e.* :

$$\forall\ b\ o,\ \texttt{weakly\_valid}(m,b,o) \Rightarrow 0 \le cm(b) + o \le 2^{32} - 1$$

Or equivalently, using only the notion of valid locations:

$$\forall\ b\ o,\ \texttt{valid}(m,b,o) \Rightarrow 0 \le cm(b) + o < 2^{32} - 1$$

Hence, Definition 4.3.3 (valid concrete memories) requires that valid locations be mapped at addresses strictly lower than $2^{32} - 1$.

Comparison with `NULL`. After these relatively easy fixes, we have found an interesting discrepancy with the semantics of CompCert C (version 2.4). The issue is related to the comparison of pointers with the `NULL` pointer. In CompCert, the `NULL` pointer is represented by the integer 0. The semantics therefore assumes that a location can never be equal to the `NULL` pointer. In CompCertS, a location (b,i) can evaluate to 0 if the computation $cm(b) + i$ overflows and wraps around. This is a glitch in the CompCert semantics that is illustrated by the code snippet of Figure 6.4. This program initialises a pointer `p` to the address of the variable `i`. In the loop, `p` is incremented until it equals `0` in which case the loop exits and the program returns `1`. The executable semantics of CompCert C returns `0` because `p==0` is always `false` whatever the value of `p`. However, when running the compiled program, the pointer is a mere integer, the integer eventually overflows; wraps around and becomes 0. Hence, the test holds and the program returns 1. One might wonder how the CompCert semantic preservation can hold in the presence of such a contradiction. Actually, the pointers are kept logical all the way through to the assembly level, and the comparison with the `NULL` pointer is treated identically during all the compilation process, thus even the assembly program in CompCert returns 0. The inconsistency only appears when the assembly program is compiled into binary and run on a physical machine.

The fix consists in defining the semantics of the comparison with the `NULL` pointer only if the pointer is *weakly valid*. This causes the program to have undefined semantics at

```
int main(){
  int i=0, *p = &i;
  for(i=0; i < INT_MAX; i++) {
    if (p++ == 0) {
      return 1;
    }
  }
  return 0;
}
```

Figure 6.4: A `NULL` pointer comparison glitch

the C level as soon as we increment the pointer beyond its bounds. The issue has been acknowledged and is fixed since CompCert 2.5.

After adjusting both memory models, we are able to prove that operators of CClight are refined by SClight operators. Using this result, and under the hypothesis that the program does not run out of memory, we prove a forward simulation between CClight and SClight, thus cross-validating our formal semantics with that of CompCert.

6.3 An Executable Semantics For C

As we mentioned in Section 2.5.1, CompCert ships with an executable interpreter for CompCert C. The interpreter is a valuable tool to test whether a given C program has defined semantics or not. In Section 6.1, we explained that the semantics of all the languages in CompCertS, including C, rely on normalisations in a number of semantic rules. Hence, in order to get an executable interpreter, we need to provide an executable implementation of the normalisation.

The problem is the following. Given a symbolic value sv and an abstract memory m, find a value v such that v and sv evaluate identically in all concrete memories that are valid for m.

Given a memory m, there are finitely many valid concrete memories cm. It is thus decidable to compute a sound and complete normalisation and the naive algorithm consists in enumerating over the valid concrete memories and checking that the symbolic values always evaluate to the same values. Yet, this is not tractable.

We show that the normalisation can be thought of as a decision problem in the logic of bitvectors. A bitvector of size n is the logic counterpart of a machine integer with n bits. This logic is therefore a perfect match for reasoning over machine integers. This decision problem will then be solved by a SMT (*Satisfiability Modulo Theory*) solver (*e.g.* Z3 [MB08], CVC4 [Bar+11]).

First, we briefly recall what an SMT solver is and what problems it solves. Then, we axiomatise the memory and the notion of valid concrete memory in the SMT language. Then, we show how to encode the normalisation problem into a SMT problem. We show how to interpret the response **sat** or **unsat** from the solver. Finally, we present an optimisation of this SMT encoding that makes this solution tractable.

6.3.1 SMT solvers

Satisfiability Modulo Theories (SMT) is a generalisation of the boolean Satisfiability (SAT) problem with domain-specific theories. While SAT formulas are propositional logic formulas, over boolean variables, SMT formulas are first order logic formulas, enriched with theories *i.e.* the variables are not only boolean but can be integers, arrays, bitvectors, *et cætera.*

The input of a SMT solver is a set of variables, *e.g.* bitvectors in our case, together with constraints or assertions about those variables, expressed as first-order logic formulas. The goal of the solver is to find a *model* for this problem. A model is a *valuation*, *i.e.* an assignment of actual values to variables, such that all the constraints are satisfied. The output of a SMT solver is either **unsat** (for *unsatisfiable*), meaning that there exists no valuation that satisfies the given problem; **sat**(M), meaning that M is a *model* of the input problem; or **unknown** when the SMT solver is unable to reach a conclusive answer.

The SMT-LIB [BFT15] initiative provides a unified input language for stating SMT problems and a library of benchmarks. Using a unified language enables to run multiple SMT solvers on one given problem without having to rewrite the problem in a different format for each SMT solver used as a backend.

In the following, we show how we encode the problem of finding a normalisation into an instance of SMT problem, using the theory of bitvectors. We will use Z3 as SMT solver, however using an alternative solver should not affect our results in any way.

6.3.2 Axiomatising the memory

To encode a memory m in our logical framework, we define one logical variable for each block in m. The variable associated with each block is both its identifier and its concrete address. This works because we constrain different blocks to be mapped to different logical variables. We then define a logical function *size* mapping each block to its size and a logical function *alignment* mapping each block to its alignment, *i.e.* the number of trailing bits that must be zero. Next, we axiomatise the valid concrete memory relation by directly translating Definition 4.3.3 into first-order logic formulae.

Example 6.3.1. Consider a memory m restricted to two blocks b_1 and b_2, with b_1 of bounds $[0,4[$ and alignment 2 bits and b_2 of bounds $[0,8[$ and alignment 3 bits. The axiomatisation of m is given by the following formulae.

$$\begin{array}{rl}
\text{Disjoint blocks:} & \texttt{distinct}(b_1, b_2) \\
\text{Block sizes:} & \mathit{size}(b) = \begin{cases} 4 & \text{if } b = b_1 \\ 8 & \text{if } b = b_2 \\ 0 & \text{otherwise} \end{cases} \\
\text{Block alignments:} & \mathit{alignment}(b) = \begin{cases} 2 & \text{if } b = b_1 \\ 3 & \text{if } b = b_2 \\ 0 & \text{otherwise} \end{cases} \\
\text{No overlap:} & \forall b, b', o, o'. \bigwedge \begin{cases} b \neq b' \\ o < \mathit{size}(b) \\ o' < \mathit{size}(b') \end{cases} \Rightarrow b + o \neq b' + o' \\
\text{Address space:} & \forall b, o. o < \mathit{size}(b) \Rightarrow 0 < b + o < \texttt{Int.max\_unsigned} - 1 \\
\text{Alignment :} & \forall b, b \,\texttt{mod}\, 2^{\mathit{alignment}(b)} = 0
\end{array}$$

6.3.3 Translating symbolic values into logical expressions

We process the symbolic value sv to be normalised into a logical symbolic value sv^*.

We can safely assume that the symbolic value sv does not contain the `undef` value at any depth, *i.e.* the `undef` value does not appear as an operand of any symbolic operator inside sv. If it did, the whole symbolic value sv would evaluate to `undef`, and `undef` would therefore be the normalisation of sv. In other words, we check that sv does not contain `undef` before even calling the SMT solver.

We replace pointers $\texttt{ptr}(b, i)$ by the bitvector addition of the variable associated with block b and the bitvector representing the integer i.

Indeterminate values $\texttt{indet}(l)$ are modelled by fresh variables; the same variable is used for every occurence of the same label, modelling the intuition of an arbitrary fixed value that we introduced in Chapter 4.

Other values (32-bit and 64-bit integers, 32-bit and 64-bit floating-point numbers) are mapped to their representation as bit-vectors. Unary and binary operations on symbolic values are mapped to their equivalent operations in terms of bitvectors.

6.3.4 Normalisation as SMT queries

We now show how a SMT solver can be used to compute normalisations. As we will see, the queries are quite different depending on whether we expect the normalisation to result in a pointer or an integer value. Indeed, like the specification of the normalisation in earlier versions of this development, the implementation of the normalisation function is parametrised by its expected return type. Since we have seen (see Theorem 4.4.1) that a symbolic value may normalise into either an integer or a pointer but never both, it is sound for the ultimate (typeless) implementation to *try* both types and keep the one that succeeds, if any.

The reasoning behind the algorithm of the normalisation is the following. Given a symbolic value sv and a memory m, our goal is to find a value v which evaluates the same as sv in every valid concrete memory for m. We first find a candidate value v_0 such that there is some $cm_0 \vdash m$ such that $[\![sv]\!]^{im}_{cm_0} = [\![v_0]\!]^{im}_{cm_0}$. Then, we check that this v_0 evaluates like sv in *every* valid concrete memory, or equivalently that there exists no valid concrete memory in which sv evaluates to a value $v \neq v_0$.

In the following we describe algorithms for the normalisation. These algorithms include calls to SMT solvers via the function SMT. This function returns one of **unsat** meaning the problem is unsatisfiable or $\textbf{sat}(M)$, meaning the problem is satisfiable with model M. A model is a valuation $\{k_1 \mapsto v_1; \ldots ; k_n \mapsto v_n\}$ that associates values v_i to variables k_i.

6.3.4.1 Normalising into an integer.

The algorithm to normalise sv^* into an integer is described in Algorithm 2. First, we generate the SMT query: $sv^* = i$, where i is a fresh logical variable. Suppose the formula is satisfiable for a value v for logical variable i. This means that there exists a valid concrete memory such that sv is evaluated into the value v. We now need to check that no valid concrete memory evaluates sv to a different value. We generate a second SMT query: $sv^* = i \wedge i \neq v$. This query is expected to be unsatisfiable. If it is indeed unsatisfiable, then we return v as the normalisation of sv, because it means that *every* valid concrete memory yields this value v. On the other hand, if it is satisfiable, then there exists a different result with a different valid concrete memory, meaning that the result depends non-deterministically on the concrete memory. In this case the normalised value is `undef`.

Example 6.3.2. *Consider the memory m introduced in Example 6.3.1. Consider the symbolic value $sv = \texttt{ptr}(b_2, 0)\&\texttt{0x00000007}$. This symbolic value clears all bits but the*

Algorithm 2: Normalisation of *sv* into an integer

```
if $SMT(sv^* = i) = \textbf{sat}(\{i \mapsto v\})$ then
  if $SMT(sv^* \neq v) = \textbf{unsat}$ then
    return $\texttt{int}(v)$
  end if
end if
return undef
```

three least significant from pointer $\texttt{ptr}(b_2, 0)$*. It is expected to normalise to* $\texttt{int}(0)$ *because alignment constraints ensure that the last three bits are 0.*

We generate the SMT query $var_{b_2}\&\texttt{0x00000007} = i$*. The SMT solver finds a model where* $i \mapsto 0$ *with a witness concrete memory* cm_0 *where* e.g. $cm_0(b_2) = 8$*. In that concrete memory, the symbolic value indeed evaluates to* $8\&7 = 0$*. We now check that there is no other integer solution by submitting the following query:* $var_{b_2}\&\texttt{0x00000007} = i \wedge i \neq 0$*. The SMT solver answers* **unsat***, indicating that no valid concrete memory yields an integer different from 0. Hence sv normalises into* $\texttt{int}(0)$*.*

Consider now the symbolic value $sv = \texttt{ptr}(b_1, 0) < \texttt{ptr}(b_2, 0)$*. We transform sv to get* $sv^* = var_{b_1} < var_{b_2}$*. The first SMT query* $sv^* = i$ *can be satisfied with* e.g. $i = 0$*, meaning that there is a valid concrete memory cm where* b_1 *is allocated after* b_2*,* e.g. $cm(b_1) = 16$ *and* $cm(b_2) = 8$*. We then submit the second SMT query:* $sv^* = i \wedge i \neq 0$*. It is satisfied with* $i = 1$ *by a concrete memory where* e.g. $cm(b_1) = 4$ *and* $cm(b_2) = 8$*. Hence, sv normalises into* $\texttt{undef}$*, since sv has no sound normalisation in m.*

6.3.4.2 Normalising into a pointer.

Getting the normalisation of a pointer value is more complicated because there are several ways of decomposing an integer into a location made of a base and an offset. Algorithm 3 explains how we proceed. Theorem 4.4.1 tells us that only one such decomposition will be valid for all concrete memories. Moreover, by Lemma 4.4.1 we know that a symbolic value sv can only have $\texttt{ptr}(b, o)$ as normalisation if b appears syntactically in sv. As a result, given fresh logical variables b and o, we encode that b must be a block that appears in sv by asserting the logical constraint $b \in B$, where B is initially the set of blocks that appear in sv. We generate the SMT query $sv^* = cm(b) + o$. Suppose we get a model such that $b \mapsto b'$ and $o \mapsto o'$. The following query checks whether there can be another pointer denoted by the same symbolic value in another valid concrete memory: $sv^* \neq cm(b') + o'$. If the query is unsatisfiable, then the normalisation returns $\texttt{ptr}(b', o')$. Otherwise, if the query is still satisfiable, we know that $\texttt{ptr}(b', o')$ is not a sound normalisation of sv. We can therefore discard block b' from the candidates for the normalisation of sv (*i.e.* we remove it from the set B) and we iterate the search. This process eventually terminates because there are finitely many blocks b that appears syntactically in sv.

Example 6.3.3. *Consider again the memory m of Example 6.3.1 and the symbolic value* $sv = \texttt{ptr}(b_1, 1) - \texttt{ptr}(b_2, 2) + \texttt{ptr}(b_2, 4) + \texttt{indet}(b_3, 4)\ \&\ \texttt{int}(0x0)$*. We process sv into a logical expression* sv^* *by replacing* $\texttt{indet}(b_3, 4)$ *by the fresh variable* $x_{3,4}$*:*

$$sv^* = cm(b_1) + 1 - cm(b_2) - 2 + cm(b_2) + 4 + x_{3,4}\ \&\ \texttt{0x0}$$

Notice that the two occurrences of $cm(b_2)$ *cancel each other out, and that we have* $\forall x, x\ \&\ \texttt{0x0} = 0$*. The expression* sv^* *is therefore equivalent to* $cm(b_1) + 3$*. This simplification is not actually made in the implementation and is merely present here for the sake of clarity.*

Algorithm 3: Normalisation of *sv* into a pointer

```
$B \leftarrow \{b \mid b \in sv\}$
while true do
  if $SMT(sv^* = b + o \land b \in B) = \mathbf{sat}\{b \mapsto b', o \mapsto o'\}$ then
    if $SMT(sv^* \neq b' + o') = \mathbf{unsat}$ then
      return $\mathtt{ptr}(b', o')$
    else
      $B \leftarrow B \backslash \{b'\}$
    end if
  else
    return undef
  end if
end while
return undef
```

The SMT query we need to solve is: $cm(b_1)+3 = cm(b)+o$*. Although it seems silly, the SMT solver may generate a valid concrete memory* cm *where* $cm(b_1) = 4$ *and* $cm(b_2) = 8$ *and propose the solution* $b^* = b_2$ *and* $o^* = -1$*, which satisfies the equation we gave as input. However, the query* $sv^* \neq cm(b^*) + o^*$ *is indeed satisfiable, for example with a concrete memory* cm' *identical to* cm *except that* $cm(b_2) = 16$*.*

We begin the whole process again, with the extra constraint that $b \neq b_2$*. A more natural solution is* $b^* = b_1$ *and* $o^* = 3$*. It turns out this is the only solution to this equation, as we can see by submitting this second query to the SMT solver,* $cm(b_1) + 3 \neq cm(b_1) + 3$*, which is obviously unsatisfiable. Therefore the symbolic value sv is normalised into the location* $\mathtt{ptr}(b_1, 3)$*.*

6.3.5 Relaxation and Optimisation of the SMT Encoding

The encoding of the memory that we presented is linear in the number of allocated blocks, as there is one definition for the *size* function and one for the *alignment* function for every block. Thus, as the memory gets bigger, the normalisation would get slower. In practice, we observe that the size of the memory has a dramatic (negative) impact on SMT solvers. To tackle the problem, we propose a relaxation of the SMT query that is independent of the number of allocated blocks and only depends on the size of the symbolic value to be normalised. A key observation is that a symbolic value can only be normalised if the corresponding SMT query has a unique solution. As a result, it is always sound to relax the SMT query and generate a weaker one (*i.e.* with potentially more solutions) provided the initial formula is satisfiable. Indeed, if there are more solutions, the normalisation will fail – this is always sound.

In our relaxation, we do not fully axiomatise the memory but only specify the bounds and alignments of the memory blocks B that appear syntactically in the symbolic value to be normalised. When normalising into a pointer, we also state explicitly in the SMT query that the normalisation, if it exists, should be a location (b, i) such that $b \in B$.

This relaxation is always sound, as we discussed before, for two reasons: 1) there always exists a valid concrete memory, thanks to our allocation algorithm; 2) we generate a weaker SMT query, with potentially more solutions. This relaxation is however not complete. It might miss a normalisation in pathological cases where blocks $b \in B$ are constrained not to appear at certain locations, because of other blocks $b' \notin B$. This is illustrated by

Example 6.3.4.

Example 6.3.4. *Consider a memory with 2 blocks b_1 of size 8 and b_2 of size 2^{31}. Figure 6.5 shows the possible addresses of block b_2. Because of size constraints, the concrete address 2^{31} will always be part of block b_2. Notice that block b_1 can be mapped in any of these concrete memories either before or after block b_2. However b_1 will never be at address 2^{31}. The symbolic value $sv = \texttt{ptr}(b_1, 0) == \texttt{int}(2^{31})$ therefore normalises into `false`.*

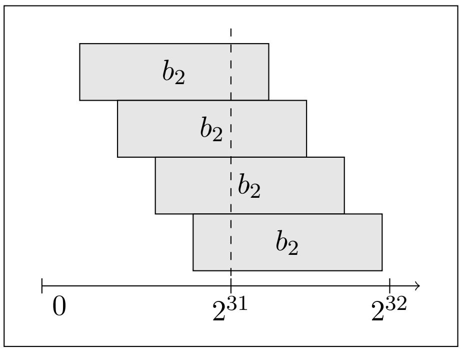


Figure 6.5: Large blocks prevent some addresses from being allocated to others.

Now if we relax the validity to only account for blocks that appear syntactically in sv, then some concrete memories will allocate b_1 at address 2^{31} and some others not. The symbolic value sv will therefore have different evaluations depending on the concrete memories, hence the normalisation will fail.

The normalisation of Example 6.3.4 requires a full axiomatisation of the memory and cannot be obtained using our relaxation. In the implementation, we also make the additional assumption that the normalisations result only in valid pointers *i.e.* their offsets are within the bounds of the blocks. This simplification limits the search space and is therefore sound but not complete: it will miss pointers out of their bounds. In our experience, since normalisations of pointers are performed just before memory accesses, the semantics will get stuck when trying to dereference an out-of-bounds pointer. In practice, we have never encountered such pathological cases where the relaxation fails when there exists a normalisation. In particular, this relaxation is *complete enough* to give a defined normalisation to all the examples we give in Chapter 3.

6.4 Experiments

After adapting the semantics of CompCert C and designing an implementation for the normalisation, we performed some experiments and executed C programs with our more permissive semantics. Because we target real-world, low-level programs, we needed to design stubs to model system calls such as `mmap`. This system call is mapped to the `alloc` operation of our memory model with appropriate parameters. Other system calls such as `open`, `read` or `write` that operate on files are mapped to their OCaml equivalent, again with appropriate parameters.

We have tested our C semantics with symbolic values on the benchmarks of CompCert. Their size ranges from a few hundreds to a few thousands lines of code. We checked the absence of regression: when the CompCert interpreter returns a defined value, our interpreter returns exactly the same value.

We have also run our interpreter over Doug Lea's memory allocator [Lea] and on parts of the NaCl cryptographic library [BLS12], which are challenging programs because they perform low-level pointer arithmetic; their size is about a few thousands lines of code. Our interpreter succeeds in giving semantics to memory management functions, such as `malloc`, `memalign` or `free`, built on top of `mmap`. As there is no other formal C semantics able to deal with low-level pointer arithmetic, we checked that the result of our interpreter was matching the output of `gcc`. Programs reading uninitialised variables have undefined semantics and `gcc` could exploit this to perform arbitrary computations. Yet, the output of `gcc` and our interpreter agree on examples similar to those presented in Chapter 3.

First, we explain how we implemented stubs for low-level system calls, and then we review a list of interesting patterns found while experimenting on the benchmarks.

6.4.1 Stubs in the interpreter

Because we target low-level C code, the programs we are interested in contain system calls, *i.e.* functions that perform low-level accesses to the system. The code of these functions is not available with the source code, since it is typically implemented in the operating system kernel. In order to interpret such programs, we need to design *stubs* for system calls, that give their semantics in the memory model.

The `mmap` system call is used in `malloc` C implementations to fetch a region of memory from the system. Its prototype is as follows.

```c
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

If `fd` is a valid file descriptor, then the contents of the file described by `fd` is mapped into memory, starting from offset `offset`. If `fd` is `-1`, a memory region is made available, but is not backed by any file. The address of the memory to be allocated in the virtual address space is specified by the `addr` parameter. This parameter is only taken as a hint as to where the mapping should be placed. If `addr` is `NULL`, the system chooses where to allocate the region. The allocated memory region spans `length` bytes. The `prot` parameter specifies whether pages can be read, written, executed or not. The `flags` parameters specifies whether this memory mapping is only visible in the current process or if it should be visible from other processes, amongst other properties.

This is a rather complex specification. Since our intended use is merely mapping readable and writable memory for `malloc` implementations, not backed by any file and limited to a single process, we restrict ourselves to this simple case. The stub for the `mmap` system call first checks the values of the arguments to ensure that we are in this simple case. If not, the execution of the whole program fails, as we do not model this behaviour. If it is the case, we fetch the value of the `length` argument and allocate a block of the requested size, with a 2^{12}-byte alignment (*i.e.* a page alignment).

Other use cases of system calls include file-managing operations such as `open`, `read` and `write`. These are used in the implementation of the C standard library functions `fopen`, `fread` and `fwrite`. However, the stubs for those objects are less linked to our memory model than that for `mmap`. For those calls, we simply map the system calls to the corresponding OCaml system call. For instance, we map the `open` system call to the OCaml `Unix.openfile`, performing the adequate conversion between flags given as an integer in the C code and flags given as a list of flags in OCaml. We also maintain a correspondence between C and OCaml file handles. In particular, we encode the fact that file handle 0 in C is mapped to the standard input, file handle 1 to standard output and file handle 2 to standard error output. We do the same kind of transformation for `read` and `write` system calls.

6.4.2 Patterns and Idioms of Low-Level C Code

Using the stubbed interpreter with an implementation for the normalisation, we are able to give semantics to real-life programs. This is an improvement over the existing CompCert interpreter, which fails in giving semantics (and thus interpreting) low-level programs that rely on the concrete bit-encoding of pointer and uninitialised data. The following reports on patterns we encountered in such code during our experiments.

6.4.2.1 Pointer Arithmetic Using Alignment and Bitwise Operations

The implementation of `malloc` by Doug Lea [Lea] uses the following `is_aligned` macro to check whether a pointer is aligned.

```
/* True if address A has acceptable alignment */
#define is_aligned(A) (((size_t)(A) & ALIGN_MASK) == 0)
```

For our experiments, pointers are allocated by `mmap` and are therefore known to be at least 2^{12}-byte aligned. In this example, we consider `ALIGN_MASK` to be equal to `0xF`, therefore the macro `is_aligned` checks whether a pointer is 2^4-byte aligned.

Consider a pointer `p` whose logical address is $\texttt{ptr}(b, 3)$. Since b is known to be 2^{12}-byte aligned, we have that the last 12 bits of `b` are zeros. The code `is_aligned(p)` expands to `(((size_t)(p) & 0xF) == 0)` and constructs the symbolic value $\texttt{ptr}(b, 3)\&\texttt{int}(\texttt{0xF})$. This symbolic value normalises into $\texttt{int}(0)$, since the last 4 bits of `p` are `0011`, *i.e.* 3 in decimal, hence different from 0.

In general, with these alignment constraints, we have that $\texttt{ptr}(b, o)\&\texttt{int}(\texttt{0xF})$ is equivalent to $o\&\texttt{0xF}$, *i.e.* it is equivalent to o for o less than 15.

A similar example is the function `memalign(al,nb)`, where `al` must be a power of two (*i.e.* `al`$= 2^n$). The function dynamically allocates a `nb`-byte region, and ensures that the address returned is 2^n-byte aligned, *i.e.* the n last bits are zeros. When called with `al` $= 32$, the function computes checks such as `p & 0x1F == 0` to check that the 5 last bits are zeros. The left-hand side of the comparison is evaluated in the same manner as the example above, and the comparison is computed trivially.

6.4.2.2 Comparison Between Pointers and `(void*)(-1)`

As discussed in Section 3.1, several system calls, such as `mmap` or `sbrk`, are expected to return pointers but return `(void*)(-1)` on error. Figure 3.3 shows an example of such a call to `mmap`. Our normalisation gives a defined semantics to these comparisons between pointers and -1 using the following reasoning.

We know that pointers returned by `mmap` are aligned on a page boundary (2^{12} in our implementation), *i.e.* the 12 last bits of the pointer are zeros. When the allocation succeeds, the pointer can therefore never be -1 (in binary `0xFFFFFFFF`) because it would violate alignment constraints. Hence this comparison `p == (void*)-1` normalises to `false`, as expected for a successful run of `mmap`.

6.4.2.3 Operations on Uninitialised Values

The example shown in Figure 3.6 (flag setting) is a simplified version of a C idiom that appears in real-life programs. For example, the `memalign` function described in Section 6.4.2.1

features this kind of operations on undefined values.

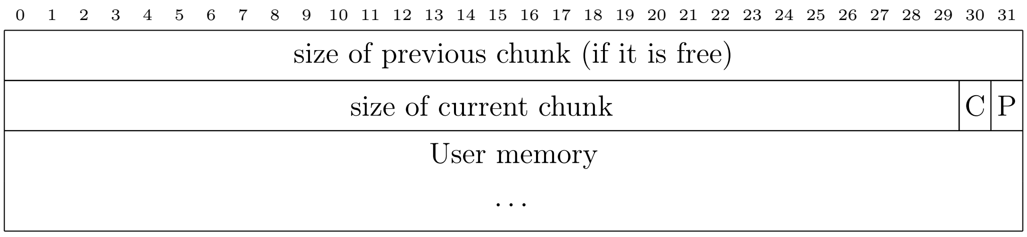


Figure 6.6: Structure of the memory managed by `malloc`

Let us examine the structure of the chunks of memory allocated by `malloc`, illustrated in Figure 6.6. As explained in the documentation of `dlmalloc` [Lea], every memory chunk is accompanied by two 32-bit words of meta-data. The first word of metadata contains the size of the previous chunk, if the previous chunk is free, *i.e.* not used, and is part of the previous chunk otherwise.

The second word of metadata contains the size of the current chunk, which must be 4-byte aligned, *i.e.* a multiple of 4, therefore the last two bits can be used to store the extra `C` and `P` bits which indicate respectively whether the current and previous chunks are in use (`1`) or free (`0`). Initialising the second word of meta-data can be done with the C assignment `*p = ( *p & 0b1)|size|0b10` (where the `0b` prefix applies to constants in binary format). The interested reader can find the definition of the `set_inuse` macro which expands to this code in the implementation of `dlmalloc` [Lea]. When the memory pointed by `p` is uninitialised, we construct the symbolic value (`indet`(l)&`0b1`) | `size` | `0b10`. Starting from `indet`(l) written with binary variables $A \dots H$, Figure 6.7 shows the bitwise construction of this symbolic value, for `size`$= 2^8 =$ `0x00000100`.

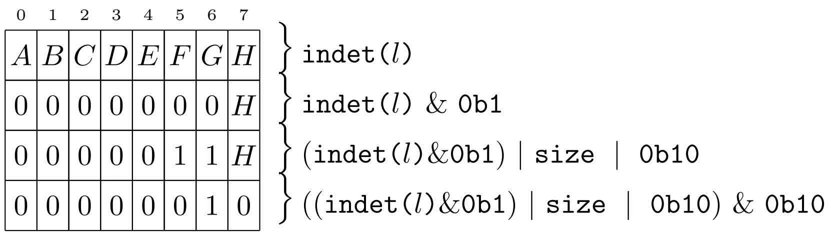


Figure 6.7: Bit-representation of symbolic values used in `dlmalloc`.

The last bit of this symbolic value is H, *i.e.* the last bit of the original `indet`(l), because `size` is a multiple of 4, hence the bitwise OR with `size` does not set the last bit. The symbolic value itself does not normalise, because its last bit H is indeterminate, however we are able to compute on this symbolic value, *e.g.* retrieve its second least significant bit with this symbolic value: ((`indet`(l)&`0b1`) | `size` | `0b10`) & `0b10`. As the last line of Figure 6.7 shows, this symbolic value evaluates to `int`(`0b10`), hence it normalises into `int`(`0b10`), as expected.

6.4.2.4 Copying Bytes between Memory Areas with `memmove`

Because we include normalisations before every branching instruction, our semantics requires the target of a jump instruction to be unique. This is a consequence of the fact that a symbolic value representing a condition should normalise to some unique boolean value. In other words, a program whose control-flow depends on the memory layout has an undefined behaviour. This dependance on the memory layout (*e.g.* on the memory allocator) is a portability bug that is detected by our semantics.

In our experiments, we have encountered this situation in an implementation of the `memmove` function (see Figure 6.8) which implements a memory copy even when the origin and destination memory regions do overlap. In this case, *e.g.* when the last byte of the origin is the same as the first byte of the destination, the naive `memcpy` function would overwrite the last byte of the source, therefore invalidating the copy.

The aim of the `memmove` function is to avoid this situation by first checking which of the source and destination address is the smallest, and performs the copy forwards or backwards. This involves the pointer comparison `dest <= src`, which is undefined in C when `dest` and `src` point to different objects. It is undefined in CompCert's memory model, and in ours, when the pointers are from distinct memory blocks, because the result of this comparison depends on the concrete memory layout.

```c
void *memmove( void *s1, const void *s2, size_t n ) {
  char *dest = (char *) s1;
  const char *src = (const char *) s2;
  if ( dest <= src )
     while ( n-- ) { *dest++ = *src++; }
  else {
     src += n; dest += n;
     while ( n-- ) { *--dest = *--src; }
  }
  return s1;
}
```

Figure 6.8: `memmove` with an undefined semantics

We have solved the issue by replacing the original condition `dest <= src` with the more involved condition `src <= dest & dest < src + n`. This condition explicitly tests whether the memory regions overlap using the integer `n` which is the number of bytes to be copied. Notice that we use the bitwise & operator on purpose instead of the lazy boolean && operator. The lazy && would force the evaluation of `src <= dest` which cannot be normalised. The new condition with a bitwise & operator constructs a symbolic value which is independent from the memory layout and has therefore always a defined normalisation. In particular, if the pointers are from distinct blocks, the condition is always false because locations from distinct blocks cannot overlap.

6.5 Conclusion and Discussion

In order to benefit from the more relaxed memory model introduced in Chapters 4 and 5, and therefore give semantics to more programs, we have extended the semantics of all the languages of CompCert with our formalism of symbolic values and normalisations.

We have shown that normalisations must be introduced before every memory access, as was already suggested in Section 5.1. Normalisations are also necessary for the semantics of conditionnally branching instructions. The latter are necessary because we want to keep CompCert's semantics deterministic, so that we can reuse the existing proof of CompCert as much as possible. Therefore the condition of an if-then-else structure must normalise into a unique value so that the execution continues on one branch or the other.

In Section 6.2, we have proved that the resulting symbolic semantics are refinements of those of CompCert, so that programs that have defined semantics in CompCert have the same defined semantics in CompCertS. In addition to this formal result, the process of doing this proof has helped discover discrepancies in CompCert, related to pointer comparison to `NULL`.

Note that to perform the proof, we had to use an alternate semantics of Clight, where *so-called* simplifications (normalisation attempts) were introduced eagerly. The reason why this is necessary is because the normalisation of a given symbolic value does not always get more defined when the memory evolves: it is not *monotonic*. For instance, consider a pointer validity test `&x != NULL`, where `x` is a stack-allocated integer mapped at location $\texttt{ptr}(b, 0)$. In a memory state where b is a valid, allocated block, the validity test normalises into the integer $\texttt{int}(1)$ because valid pointers are distinct from the `NULL` pointer. However, once b is freed (because its host function returns), the same expression evaluates to `undef`, because b may now be allocated anywhere, and in particular possibly at the address 0. We contemplate tightening the validity constraint on concrete memories to force b not to be allocated at invalid addresses even after the end of its lifetime. We would restate Property 4.3.2 so that it applies to all blocks that once were allocated. We would then need to record the lifetime of every block: when it was allocated and when it was freed. Using this information, we would restate Property 4.3.1 so that any pair of blocks whose lifetimes overlap may not be allocated at overlapping locations. These modifications would maintain the semantics of pointer comparisons `&x != NULL` and `&x != &y` defined even after `x` and `y` are freed. The proof would then be doable with the same Clight semantics that is used in the proof of correctness of the compiler.

Next, we have explained how to obtain an executable normalisation. To that end, we use a SMT solver, and an encoding of valid concrete memories and symbolic values into the logic of bitvectors. Currently, not all operations are axiomatised into bitvectors (*e.g.* operations on floating point numbers). Besides, the translation is made outside of the trusted computing base, and the various algorithms that compute a normalisation based on SMT queries are programmed in OCaml. While experimenting with real-life programs has not uncovered obvious bugs in our translation, we envisage performing a more principled usage of the SMT solver in later releases of this development. Assuming only the correctness of the underlying SMT solver, we could prove that our translation of symbolic values into bitvector expressions is faithful and that algorithms actually compute sound normalisations in Coq.

Finally, we have experimented on real-life C programs that had undefined semantics according to CompCert (and the C standard), but for which our semantics is defined and in accordance with our expectations and the output of mainstream compilers such as `gcc`. The C programs we have tested range from few-lines hand-written programs that exhibit low-level operations on pointers or uninitialised data to real-life code such as an implementation of `malloc` or the testing of a complete C standard library.

In the next chapters, we will show how we reprove the compiler passes with this new semantics, ultimately achieving CompCertS, a formally verified compiler for low-level C.

Chapter 7

Memory Relations

As seen in Section 2.4, CompCert's passes are proved correct using simulation arguments. We recall below the general shape of these transformations and their proofs, for a source language $\mathcal{S}$ and a target language $\mathcal{T}$ (possibly the same language in the case of optimisations). Let $\mathbb{P}_L$ be the set of programs in language L, for $L \in \{\mathcal{S}, \mathcal{T}\}$. A program is merely a mapping from function and global variable identifiers to functions and global variables. A compilation pass is a partial function $comp : \mathbb{P}_{\mathcal{S}} \rightharpoonup \mathbb{P}_{\mathcal{T}}$ which transforms the code of every function of the program. A simulation theorem is proved for every compilation pass in the CompCert compiler. For the simplest case of *lock-step* simulations (*i.e.* one step in the source is mapped to exactly one step in the target), the theorem is of the form shown in Figure 7.1, where $\xrightarrow{t}{}^{P}_{L}$ is the transition relation associated with language L with respect to program P, emitting event t. We write `state` L the domain of semantic states associated with programs in language L. The relation $\mathcal{R} \subseteq$ `state` $\mathcal{S} \times$ `state` $\mathcal{T}$ (written `match_states` in the Coq development) is referred to as a simulation relation and is an invariant that must hold throughout the execution of both programs. The diagram on the right-hand side of Figure 7.1 is a graphical representation of the lock-step simulation theorem, where hypotheses are drawn with plain lines and conclusions are drawn with dashed lines.

```
Theorem step_simulation:
  ∀ S1 t S2, S1 $\xrightarrow{t}{}^{P_S}_{\mathcal{S}}$ S2 ⇒
  ∀ S1', S1 $\mathcal{R}$ S1' ⇒
    ∃ S2', S1' $\xrightarrow{t}{}^{P_T}_{\mathcal{T}}$ S2' ∧ S2 $\mathcal{R}$ S2'.
```

$$\begin{tikzcd}
\mathcal{S} & S_1 \arrow[r, dash, "\mathcal{R}"] \arrow[d, "t"'] & S_2 \arrow[d, dashed, "t"] & \mathcal{T} \\
& S'_1 \arrow[r, dash, dashed, "\mathcal{R}"'] & S'_2 &
\end{tikzcd}$$

Figure 7.1: Lock-step simulation theorem

Simulation proofs are typically performed by induction on the execution relation of programs $\xrightarrow{t}{}^{P_S}_{\mathcal{S}}$. This reasoning is handled by the Coq proof assistant. The difficulty in the proof lies in the choice of the simulation relation $\mathcal{R}$, for which there is no general methodology to follow. The purpose of this chapter is to investigate the simulation relation $\mathcal{R}$. The choice of this matching relation is crucial for the simulation proof. It must be carefully chosen:

- $\mathcal{R}$ must be strong enough so that it gives enough information linking both states to prove that S_2 can take a step towards a matching S'_2. For example, a relation such that $\forall\ x\ y,\ x\mathcal{R}y$ gives no information about the relationship between the two states.

While this relation makes it trivial to prove that S_1' and S_2' match, it is impossible to prove that S_1' can step to S_2'.

- $\mathcal{R}$ must be an invariant: the relation must hold at every step throughout the execution of both programs, so that it is possible to prove that S_1' and S_2' match from the fact that S_1 and S_2 match.
- For every initial state S of the source program, there must exist an initial state S' of the target program such that $S\mathcal{R}S'$.
- For every final state S of the source program, every state S' such that $S\mathcal{R}S'$ must be a final state of the target program.

To satisfy these conditions, $\mathcal{R}$ relates the components of the semantic states of the different languages involved. From the C semantics down to the assembly semantics, all the states include, among other components, an abstract memory state m. Depending on the language, it may also include a map from variable identifiers to symbolic values, various environments for temporary variables, pseudo-registers or machine registers.

For example, consider a compiler pass from a language $\mathcal{S}$ to a language $\mathcal{T}$. Consider that those languages have semantic states that include only a memory state and an environment (mapping from identifiers to symbolic values), *i.e.* $\texttt{state}\ \mathcal{S} = \texttt{state}\ \mathcal{T} = \texttt{mem} \times \texttt{env}$. The simulation relation $\mathcal{R}$ needs to relate those states and will be of the form:

$$(m_1, e_1)\mathcal{R}(m_2, e_2) \triangleq \bigwedge \begin{cases} m_1 \mathcal{R}_m m_2 \\ e_1 \mathcal{R}_e e_2 \end{cases}$$

The memory is ubiquitous in the semantic states of all intermediate languages. We will study in this chapter how memory states are related, *i.e.* how the $\mathcal{R}_m$ relation can be instantiated.

We will first introduce a notion of structure-preserving memory relations, which relates two memory states that have the same *structure*, *i.e.* they have the same blocks with the same bounds, but the contents of these blocks are symbolic values that are not necessarily pairwise equal but only satisfy a given binary relation, *e.g.* the equivalence relation on symbolic values that has been introduced in Definition 5.2.1.

Then, we will focus on memory injections, a notion of memory transformation used in CompCert that we have presented in Section 2.5.3.1. Memory injections describe how different blocks may be merged together. Those are the most complex memory transformations used in CompCert. As we shall see, some special care must be taken to generalise memory injection to symbolic values, in particular because we model a finite memory and because of our treatment of uninitialised values.

For both structure-preserving memory relations and memory injections, we provide theorems that will be useful in Chapter 8, where we adapt the proofs of correctness of all the compiler passes. In particular, we provide theorems linking normalisation and memory operations to the various memory relations.

7.1 Structure-Preserving Memory Relations

The purpose of this section is to introduce memory relations that preserve the structure of the memory. The contents of the two memories satisfying those structure-preserving memory relations, however, are not required to be the same but to satisfy some binary relation on symbolic values, *e.g.* the $\equiv$ relation (see Definition 5.2.1). These relations will be used instead of the plain equality of memory states or memory extensions (introduced in Section 2.5.3.2. The differences between the relations used in CompCert and the ones we introduce in this section are twofold: first, we require that the structure is exactly the same for the two memories because our semantics (and in particular that of the normalisation) is sensitive to changes in the memory structure; second, the contents stored in memory states are no longer values but symbolic values, and the relations over those contents therefore need to be generalised.

7.1.1 Structural Equivalence

What we call the *structure* of a memory state m is essentially everything but the actual contents of the memory.

Definition 7.1.1 (Structural equivalence)**.** *Two memory states m_1 and m_2 are structurally equivalent (written $m_1 \cong m_2$) if and only if all their attributes except* `contents` *are equal. Formally,*

$$m_1 \cong m_2 \triangleq \begin{cases} \texttt{nextblock}(m_1) = \texttt{nextblock}(m_2) & \text{same block counter} \\ \forall\ b, \texttt{bounds}(m_1, b) = \texttt{bounds}(m_2, b) & \text{same bounds} \\ \forall\ b, \texttt{alignment}(m_1, b) = \texttt{alignment}(m_2, b) & \text{same alignment constraints} \end{cases}$$

Definition 7.1.1 does not constrain the contents of the memory states at all. However, the structural equivalence is a tight enough relation so that interesting facts can be derived from it, no matter what the contents of the memories are.

Lemma 7.1.1. *Structurally equivalent memory states admit the same set of valid concrete memories.*

$$\forall\ m_1\ m_2,\ m_1 \cong m_2 \Rightarrow \forall\ cm,\ (cm \vdash m_1 \Leftrightarrow cm \vdash m_2)$$

Proof. The proof is immediate from the fact that m_1 and m_2 give the same bounds and the same alignment constraints to blocks, hence the validity constraints are the same. □

From Lemma 7.1.1, we can deduce the useful Theorem 7.1.1 about the normalisation of symbolic values in structurally equivalent memory states.

Theorem 7.1.1. *For any memory states m_1 and m_2 that are structurally equivalent ($m_1 \cong m_2$), for any symbolic value sv, the normalisations of sv in m_1 and in m_2 are equal. Formally,*

$$\forall\ m_1\ m_2,\ m_1 \cong m_2 \Rightarrow \forall\ sv,\ \texttt{normalise}\ m_1\ sv = \texttt{normalise}\ m_2\ sv$$

Proof. Recall that `normalise` is a function that returns a sound normalisation when one exists, and `undef` otherwise. The proof distinguishes those two cases.

- In the first case, there exists a sound normalisation, *i.e.* there exists a value v such that $sv \xrightarrow{m_1} v$. Unfolding Definition 4.4.1 (of the sound normalisation relation) yields the following equation:

  $$\forall\ cm \vdash m_1,\ \forall\ im, [\![sv]\!]^{im}_{cm} = [\![v]\!]^{im}_{cm}$$

  Because m_1 and m_2 are structurally equivalent, we can use Lemma 7.1.1 to prove that v is also a sound normalisation of sv in m_2, *i.e.* $sv \xrightarrow{m_2} v$. Hence, `normalise` m_2 sv returns this value v.

- In the second case, there does not exist a sound normalisation of *sv* in m_1. Because m_1 and m_2 are structurally equivalent, we can use Lemma 7.1.1 to prove that there does not exists a sound normalisation of *sv* in m_2 either. Indeed, if there were one, it would also be a sound normalisation in m_1, thus contradicting the hypothesis. Hence, `normalise` m_1 *sv* and `normalise` m_2 *sv* both return `undef`, and the property holds.

□

Theorem 7.1.1 will be useful to prove that memory operations preserve structural equivalence in Section 7.1.3. Chapter 8 relies on this theorem and its implications to prove the correctness of all the passes of our symbolic compiler.

7.1.2 Symbolic Values Relations

In this section, we enrich the structural equivalence relation with additional properties on the contents of the two memories. We will first define an equivalence relation between memories using the equivalence of symbolic values that has been defined in Definition 5.2.1. Then we will consider the *less-defined* relation (see Definition 2.5.2) that is used in CompCert to capture the notion of *improvement* of values. We will show how we lift this relation to a relation $\leq_{sv}$ on symbolic values. Finally we will define an *improvement* relation $\leq_m$ on memory states.

7.1.2.1 Memory Equivalence

As demonstrated in Section 5.2, several different symbolic values $sv \in sval$ can denote equivalent sets of values. Because we do not want to distinguish between equivalent symbolic values, we introduce equivalence classes between symbolic values that always denote the same set of values. We recall here the definition of equivalent symbolic values, written $\equiv$:

$$sv_1 \equiv sv_2 \triangleq \forall\ cm\ im, [\![sv_1]\!]_{cm}^{im} = [\![sv_2]\!]_{cm}^{im}$$

The equivalence generalises the equality of values into an equality of symbolic values. This is needed to prove the `load_store_same` and `load_int64_split` theorems seen in Section 5.2 (*good-variable properties*), for example. However, those theorems only establish the equivalence of symbolic values, not of memory states.

We define an equivalence relation on memory states as a combination of structural equivalence and additional properties on the contents of the memories, using the function `smv_to_sval` (see Figure 5.3) to transform `smemval`s into the symbolic values they denote.

Definition 7.1.2 (Memory equivalence). *Two memory states m_1 and m_2 are equivalent (written $m_1 \equiv_m m_2$) if and only if they are structurally equivalent and the `smemvals` contained at each location are pairwise equivalent. Formally,*

$$m_1 \equiv_m m_2 \triangleq \begin{cases} m_1 \cong m_2 \\ \forall\ b\ o,\ \texttt{smv\_to\_sval}\ m_1[b][o] \equiv \texttt{smv\_to\_sval}\ m_2[b][o] \end{cases}$$

In the above definition, the notation $m[b][o]$ is used to fetch the `smemval` at location (b, o) in memory state m.

Definition 7.1.2 allows to adapt theorems from CompCert's memory model to our symbolic setting. Consider for instance the theorem `store_int64_split`, symmetric to `load_int64_split`:

$$\begin{array}{ll}\forall & m\ b\ o\ sv\ m',\ \texttt{store Mint64}\ m\ b\ o\ sv = \lfloor m' \rfloor \Rightarrow \\ & \exists\ m_1\ m_2,\ \texttt{store Mint32}\ m\ b\ o\ (\texttt{hiword}\ sv) = \lfloor m_1 \rfloor \wedge \\ & \qquad\qquad \texttt{store Mint32}\ m_1\ b\ (o+4)\ (\texttt{loword}\ sv) = \lfloor m_2 \rfloor \wedge \\ & \qquad\qquad m' \equiv_m m_2\end{array}$$

It states that storing a 64-bit symbolic value sv results in the same memory as the memory state obtained after storing first the 4 most significant bytes of sv ($\texttt{hiword}(sv)$) and then storing the 4 least significant bytes ($\texttt{loword}(sv)$). While in CompCert, the theorem establishes the equality of the resulting memory states (m' and m_2 in the theorem above), in CompCertS the theorem establishes the equivalence of the memory states. Example 7.1.1 illustrates why the memories are not equal, but equivalent.

Example 7.1.1. *Consider a memory m and a block b of bounds $[0, 8[$. Consider a 64-bit integer l. On one hand, storing $\mathtt{long}(l)$ with the* `Mint64` *memory chunk results in the contents drawn on the left-hand-side of the following picture,* i.e. *simply different bytes of the original value $\mathtt{long}(l)$. On the other hand, storing the 64-bit integer* via *two 32-bit accesses, using symbolic operators* `hiword` *and* `loword`*, results in the contents of the memory shown on the right-hand-side of the following picture.*

One `Mint64` store	Two `Mint32` stores	$\uparrow$ address growth
$\texttt{Symbolic}(\texttt{long}(l), 7)$	$\texttt{Symbolic}(\texttt{hiword}(\texttt{long}(l)), 3)$	
$\texttt{Symbolic}(\texttt{long}(l), 6)$	$\texttt{Symbolic}(\texttt{hiword}(\texttt{long}(l)), 2)$	
$\texttt{Symbolic}(\texttt{long}(l), 5)$	$\texttt{Symbolic}(\texttt{hiword}(\texttt{long}(l)), 1)$	
$\texttt{Symbolic}(\texttt{long}(l), 4)$	$\texttt{Symbolic}(\texttt{hiword}(\texttt{long}(l)), 0)$	
$\texttt{Symbolic}(\texttt{long}(l), 3)$	$\texttt{Symbolic}(\texttt{loword}(\texttt{long}(l)), 3)$	
$\texttt{Symbolic}(\texttt{long}(l), 2)$	$\texttt{Symbolic}(\texttt{loword}(\texttt{long}(l)), 2)$	
$\texttt{Symbolic}(\texttt{long}(l), 1)$	$\texttt{Symbolic}(\texttt{loword}(\texttt{long}(l)), 1)$	
$\texttt{Symbolic}(\texttt{long}(l), 0)$	$\texttt{Symbolic}(\texttt{loword}(\texttt{long}(l)), 0)$	

One `Mint64` store　　Two `Mint32` stores

However, the contents of the two memories are `smemval`*s that are pairwise equivalent,* i.e. *at a given location l, if the left-hand-side memory holds a* `smemval` *mv_1 and the right-hand-side memory holds mv_2, then the symbolic values that mv_1 and mv_2 represent are equivalent. For example,*

$$\texttt{smv\_to\_sval}\ (\texttt{Symbolic}(\mathtt{long}(l), 7)) \equiv \texttt{smv\_to\_sval}\ (\texttt{Symbolic}(\texttt{hiword}(\mathtt{long}(l)), 3)).$$

7.1.2.2 Memory Improvement

The notion of memory equivalence is useful to reprove the theorems of the memory model. However, for most compiler passes, a more relaxed relation is needed because these passes *improve* the programs, *i.e.* they generate programs having a *more defined* semantics. This relation between semantics relies on the *less defined* relation on values (see Definition 2.5.2). The *less-defined* relation on values can be lifted to symbolic values in a similar way that the equality of values has been lifted to the equivalence of symbolic values.

Definition 7.1.3 (The *less-defined* relation for symbolic values)**.** *Let sv_1 and sv_2 be symbolic values. We say that sv_1 is less defined than sv_2 (written $sv_1 \leq sv_2$, with the same* *symbol as the relations on values) if they evaluate to values v_1 and v_2 such that v_1 is less defined than v_2 in every environment. Formally,*

$$sv_1 \leq sv_2 \triangleq \forall\ cm\ im,\ [\![sv_1]\!]_{cm}^{im} \leq [\![sv_2]\!]_{cm}^{im}$$

We then lift this generalised $\leq$ relation to memory states, by simply using $\leq$ instead of $\equiv$ in the constraint about memory contents and still use the same structural constraints. The resulting relation about memories is written $\leq_m$.

Definition 7.1.4 (Memory improvement)**.** *Let m_1 and m_2 be two memory states. We say that m_2 improves m_1, or m_1 is less defined than m_2 (written $m_1 \leq_m m_2$), if m_1 and m_2 are structurally equivalent and their contents at every location are pairwise related by the* less-defined *relation. Formally,*

$$m_1 \leq_m m_2 \triangleq \begin{cases} m_1 \cong m_2 \\ \forall\ b\ o,\ \texttt{smv\_to\_sval}\ m_1[b][o] \leq \texttt{smv\_to\_sval}\ m_2[b][o] \end{cases}$$

The memory improvement relation is used in the proofs of most compiler passes as the memory invariant in the `match_states` predicates, where memory extensions were used in CompCert. While the contents of the memories are related by the *less-defined* relation in both cases (extensions and improvements), the memory improvement relation is tighter. In particular, memory extensions do not enforce the sizes of the blocks to be the same in both memories: blocks may be larger in the second memory. The equality of block sizes is necessary to ensure that the set of valid concrete memories is the same for both memory states and therefore that the normalisation behaves identically in both memory states.

7.1.3 Compatibility With Normalisation And Memory Operations

In the following, we show interesting theorems about structure-preserving memory relations that establish the preservation of the normalisation function and of the memory operations. These theorems are the building blocks of most simulation proofs of CompCertS.

7.1.3.1 Compatibility With Normalisation

We first show that, given a fixed memory state, the normalisation preserves equivalence and improvement relations on symbolic values. Theorem 7.1.2 explicits this result.

Theorem 7.1.2. *Given a memory state m and two symbolic values sv_1 and sv_2 that are equivalent (resp. in the* less-defined *relation), the normalisations of sv_1 and sv_2 in m are equal (resp. in the* less-defined *relation). Formally,*

$$\begin{array}{ll} \forall\ m\ sv_1\ sv_2, & sv_1 \equiv sv_2 \Rightarrow \texttt{normalise}\ m\ sv_1 = \texttt{normalise}\ m\ sv_2 \\ \forall\ m\ sv_1\ sv_2, & sv_1 \leq sv_2 \Rightarrow \texttt{normalise}\ m\ sv_1 \leq \texttt{normalise}\ m\ sv_2 \end{array}$$

Using Theorem 7.1.2 and structural equivalence of memories, Theorem 7.1.3 states that the normalisation function is compatible with structure-preserving memory relations and symbolic values relations.

Theorem 7.1.3. *For any memory states m_1 and m_2 that are structurally equivalent, for any symbolic values sv_1 and sv_2 that are equivalent (resp. in the* less-defined *relation), the normalisations of sv_1 in m_1 and of sv_2 in m_2 are equal (resp. in the* less-defined *relation). Formally,*

$$\begin{array}{ll} \forall\ m_1\ m_2\ sv_1\ sv_2, & m_1\ \cong\ m_2 \Rightarrow sv_1 \equiv sv_2 \Rightarrow \\ & \texttt{normalise}\ m_1\ sv_1 = \texttt{normalise}\ m_2\ sv_2 \end{array}$$

$$\begin{array}{ll} \forall\ m_1\ m_2\ sv_1\ sv_2,\ , & m_1\ \cong\ m_2 \Rightarrow sv_1 \leq sv_2 \Rightarrow \\ & \texttt{normalise}\ m_1\ sv_1 \leq \texttt{normalise}\ m_2\ sv_2 \end{array}$$

Proof. The proofs of both parts of the theorem follow directly by application of Theorem 7.1.1 and Theorem 7.1.2. □

The two results of Theorem 7.1.3 are instrumental for the simulation proofs of most compiler passes. Indeed, consider two matching states in a simulation proof. Since we took care of introducing normalisations in the corresponding rules of every semantics (see Section 6.1), a normalisation in the source program will match a normalisation in the target language in the simulation proof. Moreover, the symbolic values to be normalised in the two programs are in the chosen relation over symbolic values (most often the *less-defined* relation $\leq$). In this case, we are able to use Theorem 7.1.3 to relate the normalisations, and subsequently to maintain the $\mathcal{R}$ relation between program states.

7.1.3.2 Compatibility With Memory Operations

We now consider the four basic operations on memory states: `load`, `store`, `palloc` and `free`. We will show that starting from memory states related by a structure-preserving memory relation, the result of an operation in the first memory state can be simulated by a similar operation in the second memory state. For the following theorems, we assume that the underlying relation on symbolic values is the *less-defined* relation $\leq$, and that the memory relation is memory improvement $\leq_m$. In fact, this memory improvement relation is used by most transformations. However, the theorems also hold for the $\equiv$ and $\equiv_m$ relations.

Theorem 7.1.4. *For any memory states m_1 and m_2 such that m_2 is an improvement of m_1, we have the following:*

$$\begin{array}{ll}
 & \forall\ m_1\ m_2,\ m_1 \leq_m m_2 \Rightarrow \\
\textit{Preservation of}\ \texttt{load} & \quad \forall\ \kappa\ b\ o\ sv,\ \texttt{load}\ \kappa\ m_1\ b\ o = \lfloor sv \rfloor \Rightarrow \\
 & \qquad \exists\ sv',\ \texttt{load}\ \kappa\ m_2\ b\ o = \lfloor sv' \rfloor \wedge sv \leq sv' \\[1em]
\textit{Preservation of}\ \texttt{store} & \quad \forall\ \kappa\ b\ o\ sv_1\ m_1',\ \texttt{store}\ \kappa\ m_1\ b\ o\ sv_1 = \lfloor m_1' \rfloor \Rightarrow \\
 & \qquad \forall\ sv_2,\ sv_1 \leq sv_2 \Rightarrow \\
 & \qquad \exists\ m_2',\ \texttt{store}\ \kappa\ m_2\ b\ o\ sv_2 = \lfloor m_2' \rfloor \wedge m_1'\ \leq_m\ m_2' \\[1em]
\textit{Preservation of}\ \texttt{palloc} & \quad \forall\ sz\ al\ m_1'\ b,\ \texttt{palloc}\ m_1\ sz\ al = \lfloor (m_1', b) \rfloor \Rightarrow \\
 & \qquad \exists\ m_2',\ \texttt{palloc}\ m_2\ sz\ al = \lfloor (m_2', b) \rfloor \wedge m_1'\ \leq_m\ m_2' \\[1em]
\textit{Preservation of}\ \texttt{free} & \quad \forall\ m_1'\ b,\ \texttt{free}\ m_1\ b = \lfloor m_1' \rfloor \Rightarrow \\
 & \qquad \exists\ m_2',\ \texttt{free}\ m_2\ b = \lfloor m_2' \rfloor \wedge m_1'\ \leq_m\ m_2'
\end{array}$$

Proof. We provide a proof sketch for the preservation of each memory operation.

- **Preservation of `load`.** Because m_1 and m_2 are structurally equivalent, the success of the `load` in m_1 implies the success of the `load` in m_2. Then, because the contents of m_1 and m_2 satisfy the $\leq$ relation, we can prove that the decoding function (see Section 5.1) results in symbolic values that are in the $\leq$ relation as well, thus proving the property.

- **Preservation of `store`.** For the same reasons as for the `load` operation, the success of the `store` follows directly from the structural equivalence hypothesis. Then, the proof that the resulting memory states are in the $\leq_m$ relation is a consequence of the fact that 1. the original memory states are in the relation; and 2. the encoding (see Section 5.1) of symbolic values that are in the $\leq$ relation result in `smemvals` in the relation as well.
- **Preservation of `palloc` and `free`.** These are direct consequences of the structural equivalence of the initial memory states.

□

We now state the preservation of the `loadv` and `storev` functions, introduced in Figure 5.5, that are variants of the `load` and `store` operations for which the address to be accessed is represented by a symbolic value that must be normalised before the actual operation is performed.

Theorem 7.1.5. *For any memory states m_1 and m_2 such that m_2 is an improvement of m_1, for any symbolic values that represent addresses sv_{addr} and sv'_{addr} in the* less-defined *relation, we have the following results.*

$$\begin{array}{ll}
 & \forall\ m_1\ m_2\ sv_{addr}\ sv'_{addr},\ m_1 \leq_m m_2 \Rightarrow sv_{addr} \leq sv'_{addr} \Rightarrow \\
\textit{Preservation of } \texttt{loadv} & \forall\ \kappa\ sv,\ \texttt{loadv}\ \kappa\ m_1\ sv_{addr} = \lfloor sv \rfloor \Rightarrow \\
 & \quad \exists\ sv',\ \texttt{loadv}\ \kappa\ m_2\ sv'_{addr} = \lfloor sv' \rfloor \wedge sv \leq sv' \\
\textit{Preservation of } \texttt{storev} & \forall\ \kappa\ sv\ sv'\ m'_1,\ \texttt{storev}\ \kappa\ m_1\ sv_{addr}\ sv = \lfloor m'_1 \rfloor \Rightarrow sv \leq sv' \Rightarrow \\
 & \quad \exists\ m'_2,\ \texttt{storev}\ \kappa\ m_2\ sv'_{addr}\ sv' = \lfloor m'_2 \rfloor \wedge m'_1 \leq_m m'_2
\end{array}$$

Proof. We present the proof for the `loadv` case. From the success of the `loadv` in m_1, we know that, for some block b and offset o:

$$\texttt{normalise}\ m_1\ sv_{addr} = \texttt{ptr}(b, o) \tag{7.1}$$
$$\texttt{load}\ \kappa\ m_1\ b\ o = \lfloor sv \rfloor \tag{7.2}$$

Using Theorem 7.1.3 on Hypothesis 7.1 and Theorem 7.1.4 on Hypothesis 7.2, we get:

$$\texttt{normalise}\ m_2\ sv'_{addr} = \texttt{ptr}(b, o) \tag{7.3}$$
$$\exists\ sv',\ \texttt{load}\ \kappa\ m_2\ b\ o = \lfloor sv' \rfloor \wedge sv \leq sv' \tag{7.4}$$

Hence, the property holds.

The proof of the `storev` case is similar. □

The properties of Theorem 7.1.4 and Theorem 7.1.5 are generalisations of properties that exist in the CompCert development about memory extensions. Those properties are building blocks of the existing proofs of semantic preservation and are therefore crucial to generalise adequately. Chapter 8 shows that the generalisation is well-suited to be used in the compiler proofs and allows to reprove most passes.

7.2 Memory Injections

In contrast with Section 7.1, which deals with structure-preserving memory relations, this section deals with a structure-transforming memory relation used in CompCert: memory injections. As explained in Section 2.5.3.1, memory injections are an essential component of CompCert for the proof of correctness of several compiler passes. Intuitively, memory injections account for memory transformations that group several blocks into one. The typical example of injection is illustrated by the `Cminorgen` pass (see Chapter 8) that consists in grouping the blocks corresponding to the local variables of a function into a single block, that represents its stack frame.

In this section, we show how we adapt the definitions of memory injections to symbolic values. First, we explain how symbolic values are injected. Then, we show how we lift this symbolic value injection to memories, highlighting the differences with the existing CompCert injections. Finally, we give a crucial theorem linking normalisation and injections, and give its proof that requires a generalisation of injections to concrete memories and indeterminate memories.

7.2.1 Injection of Symbolic Values

The injection of values `val_inject` is lifted to symbolic values, yielding the relation `sval_inj`. The injection function f has the same type as in CompCert, *i.e.* $block \rightharpoonup block \times \mathtt{Z}$. It maps a block b either to $\emptyset$, *i.e.* the block b is not injected, or to $\lfloor(b', \delta)\rfloor$, *i.e.* an offset δ in another block b'.

$$\textsc{inj-vundef}\quad \frac{}{\mathtt{sval\_inj}\ f\ \mathtt{undef}\ sv} \qquad\qquad \textsc{inj-val}\quad \frac{\mathtt{val\_inject}\ f\ v_1\ v_2}{\mathtt{sval\_inj}\ f\ v_1\ v_2}$$

$$\textsc{inj-indet}\quad \frac{f(b) = \lfloor(b', \delta)\rfloor}{\mathtt{sval\_inj}\ f\ \mathtt{indet}(b, i)\ \mathtt{indet}(b', i+\delta)} \qquad \textsc{inj-unop}\quad \frac{\mathtt{sval\_inj}\ f\ sv_1\ sv_2}{\mathtt{sval\_inj}\ f\ (\mathtt{op}_1\ sv_1)\ (\mathtt{op}_1\ sv_2)}$$

$$\textsc{inj-binop}\quad \frac{\mathtt{sval\_inj}\ f\ sv_1\ sv_2 \qquad \mathtt{sval\_inj}\ f\ sv_3\ sv_4}{\mathtt{sval\_inj}\ f\ (sv_1\ \mathtt{op}_2\ sv_3)\ (sv_2\ \mathtt{op}_2\ sv_4)}$$

Figure 7.2: Injection `sval_inj` of symbolic values

Rules INJ-VAL, INJ-UNOP and INJ-BINOP directly lift the injection `val_inject` of values to symbolic values by induction over the structure of symbolic values. Rule INJ-VUNDEF states that `undef` can be injected into *any* symbolic value. This is a direct generalisation of Rule VINJ-VUNDEF, *i.e.* `undef` can be injected into any value. Finally, Rule INJ-INDET explains how to inject indeterminate values. This is a difference with the existing injection. It mimics Rule VINJ-PTR (see Section 2.5.3.1) that injects pointers: the locations (b, i) of indeterminate values are injected by the injection function f. An important property of using locations as labels for uninitialised data is that those locations are always fresh, *i.e.* every uninitialised location holds a different label. Injecting indeterminate values ensures that this freshness is preserved by injections.

Note that the definition of `sval_inj` is syntactic, *i.e.* only symbolic values that share the same structure can be related by `sval_inj`. For example, consider an injection function

f such that $f(b) = \lfloor(b', \delta)\rfloor$. We have `sval_inj` f ($\mathtt{ptr}(b, i) + 1$) ($\mathtt{ptr}(b', i + \delta) + 1$), by Rule INJ-BINOP, but not `sval_inj` f ($\mathtt{ptr}(b, i) + 1$) ($\mathtt{ptr}(b', i + \delta + 1)$). As this is too restrictive, we consider the relation `sval_inject` that is obtained by closing the relation `sval_inj` by the equivalence relation on symbolic values $\equiv$ (see Definition 5.2.1).

Definition 7.2.1 (`sval_inject`).

$$\mathtt{sval\_inject}\ f\ sv_1\ sv_2\ :=\ \exists\ sv_1'\ sv_2', sv_1 \equiv sv_1' \wedge \mathtt{sval\_inj}\ f\ sv_1'\ sv_2' \wedge sv_2' \equiv sv_2.$$

We lift this injection of symbolic values to `smemvals`, using the `smv_to_sval` function (see Figure 5.3).

Definition 7.2.2 (`memval_inject`). *Two `smemvals` mv_1 and mv_2 are in injection if the symbolic values they represent are in injection.*

$$\mathtt{memval\_inject}\ f\ mv_1\ mv_2 := \mathtt{sval\_inject}\ f\ (\mathtt{smv\_to\_sval}\ mv_1)\ (\mathtt{smv\_to\_sval}\ mv_2).$$

7.2.2 Injection of Memories

Given the previous generalisation of injections to symbolic values, the definition of memory injections `mem_inject` is very similar to the original definition of CompCert. Definition 7.2.3 shows an excerpt from the `mem_inject` specification, in particular it highlights the differences with CompCert.

Definition 7.2.3 (`mem_inject`).

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{mem\_inject}\ f\ m_1\ m_2\ :\ \mathbb{P}\ :=\ \{} \\
\quad \ldots & \\
\mathtt{mi\_align} : & \forall\ b\ b'\ \delta,\ f(b) = \lfloor(b', \delta)\rfloor \Rightarrow \\
 & \quad \mathtt{alignment}(m_1, b) \le \mathtt{alignment}(m_2, b') \wedge 2^{[\mathtt{alignment}(m_1, b)]} \mid \delta; \\
\mathtt{mi\_size\_mem} : & \mathtt{size\_mem}\ m_2 \le \mathtt{size\_mem}\ m_1 \\
\} &
\end{array}
$$

It features two distinctive properties, `mi_align` and `mi_size_mem`, that illustrate the main modifications due to symbolic values.

Absence of offset overflows. The existing specification of `mem_inject` has a property `mi_representable` which states that if $f(b) = \lfloor(b', \delta)\rfloor$, then for any valid offset o of b, the offset $o + \delta$ obtained after injection does not overflow, *i.e.* it is an integer that fits in 32 bits. With our memory model, this property can be derived from the other properties of the injection. Indeed, if o is a valid offset of b, then $o + \delta$ is a valid offset of b' (see the well-formedness properties of the injection in Section 2.5.3.1). Since $o + \delta$ is a valid offset of a block, then it is necessarily lower than the size of the whole memory, which is itself, as we have explained in Section 5.4, strictly less than 2^{32}, therefore $o + \delta$ fits in a 32-bit integer. This property is important because it ensures that no overflow happens, hence the semantics of comparisons is faithfully preserved by injections. Being able to derive this property is a good thing because it lightens the burden of proving injections.

Alignment constraints are modelled by the property `mi_align`. In CompCert, this is only a property of the offsets δ. As explained in Section 2.5.3.1, an access at location (b, o) with a chunk κ is valid only if the offset o is a multiple of `size_chunk` κ. The existing CompCert makes the implicit assumption that memory blocks are always sufficiently aligned to make the actual concrete address aligned as expected. In CompCertS, blocks are given an explicit alignment, and data alignment must be a property of concrete addresses. As a result, we can precisely state that an injection preserves alignment: this is the purpose of the `mi_align` property of Figure 7.2.3. We require that the target block is *at least as aligned as* the source block ($\texttt{alignment}(m_1, b) \leq \texttt{alignment}(m_2, b')$) and that the offset δ is sufficiently aligned ($2^{[\texttt{alignment}(m_1,b)]} \mid \delta$) so that aligned locations are injected into *at least as aligned* locations.

The size constraint is a property that is only present in our specification. It states that the memory after injection has to be *smaller*, in the sense of the `size_mem` function (see Figure 5.6), than the original memory. The `size_mem` function computes the least address that is not allocated to a block, *i.e.* all allocated blocks can be mapped to lower addresses. This constraint is needed to ensure that if a memory allocation succeeds for a source language, it also succeeds for the target language performing the allocation on an injected memory. This is illustrated by Theorem 7.2.1 given below, which can be seen as a forward simulation for the special case of allocating a block.

Theorem 7.2.1 (`palloc_parallel_inject`). *Provided two memory states m_1 and m_2 in injection, if we can allocate a block of size sz in m_1, then we can do the same in m_2 and the resulting memory states will be in injection. Formally,*

$$\begin{array}{l} \forall \;\; f\ m_1\ m_2\ sz\ al\ m_1'\ b_1, \\ \quad 0 \leq sz \Rightarrow \texttt{mem\_inject}\ f\ m_1\ m_2 \Rightarrow \texttt{palloc}\ m_1\ sz\ al = \lfloor (m_1', b_1) \rfloor \Rightarrow \\ \quad \exists\ m_2'\ b_2, \texttt{palloc}\ m_2\ sz\ al = \lfloor (m_2', b_2) \rfloor \wedge \texttt{mem\_inject}\ f[b_1 \mapsto \lfloor (b_2, 0) \rfloor]\ m_1'\ m_2'. \end{array}$$

Proof. The insight of the proof is that the allocation ($\texttt{palloc}\ m_1\ sz\ al$) succeeds for a memory m_1 that is larger than m_2. By definition of `palloc`, we have that

$$\texttt{size\_mem}\ m_1 + sz \leq \texttt{Int.max\_unsigned} - 2^{\texttt{MA}}$$

Moreover, by definition of the injection between m_1 and m_2, we also have that

$$\texttt{size\_mem}\ m_2 \leq \texttt{size\_mem}\ m_1$$

By arithmetic, it follows that $\texttt{size\_mem}\ m_2 + sz \leq \texttt{Int.max\_unsigned} - 2^{\texttt{MA}}$. As a result, the allocation ($\texttt{palloc}\ m_2\ sz\ al$) succeeds and returns a memory m_2' and a block b_2. It remains to prove that m_1' is in injection with m_2'. Though tedious, the proof of this part mimics the existing proof of CompCert, and is omitted here. □

7.2.3 Preservation of Normalisation by Injection

This section details the proof of the main result relating normalisation and injection. Theorem 7.2.2 is the main theorem about injections and normalisations: it is essentially a forward simulation proof applied to normalisation, when the matching relation is a memory injection. The theorem requires the injection function f to be *total*: this precondition roughly states that every allocated block must be injected. Definition 7.2.4 formalises this notion; it will be explained later why this is required.

Definition 7.2.4. $\texttt{total\_injection}\ f\ m : \mathbb{P} \ := \ \forall\ b,\ \texttt{size}\ m\ b > 0 \Rightarrow f(b) \neq \emptyset.$

Theorem 7.2.2 (`norm_inject`). *Given a total injection function f, given two memory states m and m' in injection by function f, given two symbolic values sv and sv' in injection by f, the normalisations of sv in m and sv' in m' are in injection by f.*

$$\begin{aligned}\forall\ \ & f\ m\ m'\ sv\ sv', \texttt{total\_injection}\ f\ m \Rightarrow\\ & \texttt{mem\_inject}\ f\ m\ m' \Rightarrow \texttt{sval\_inject}\ f\ sv\ sv' \Rightarrow\\ & \texttt{val\_inject}\ f\ (\texttt{normalise}\ m\ sv)\ (\texttt{normalise}\ m'\ sv').\end{aligned}$$

Informally, Theorem 7.2.2 states that the normalisation function preserves the injection of symbolic values. In particular, if the normalisation in m results in a pointer, then the normalisation in m' results in a pointer that is in injection. Also, if the normalisation in m is `undef`, then the normalisation in m' can be any value. The intuition behind that fact is that memory injections amount to merging blocks; as a result, pointer arithmetic gets more defined and therefore more symbolic values get a defined normalisation. The following picture represents Theorem 7.2.2 as a simulation diagram.

$$\begin{tikzcd}
m \arrow[r, dash, "\texttt{mem\_inject}\ f"] & m' \\
sv \arrow[r, dash, "\texttt{sval\_inject}\ f"] \arrow[d, "\texttt{normalise}"'] & sv' \arrow[d, dashed, "\texttt{normalise}"] \\
v \arrow[r, dash, dashed, "\texttt{val\_inject}\ f"] & v'
\end{tikzcd}$$

The rest of this section introduces useful lemmas and finally proves the theorem. First, we will show that the normalisation of sv in m can be injected by f, *i.e.* if sv normalises into a pointer $\texttt{ptr}(b, o)$ then $f(b) \neq \emptyset$. Then, we will introduce a notion of injection for concrete and indeterminate memories, and we will provide algorithms to construct valid concrete memories and indeterminate memories from their injection. Finally, using all these results we will give the proof of Theorem 7.2.2, which is a building block of the injection-based simulation proofs.

7.2.3.1 Existence of the injection of the normalisation.

Lemma 7.2.1 is an important step in the proof of the `norm_inject` theorem. It states that if a symbolic value sv can be injected by f, then its normalisation can also be injected. In other words, if sv normalises into a pointer $\texttt{ptr}(b, o)$, then b is necessarily injected by f, *i.e.* $f(b) \neq \emptyset$.

Lemma 7.2.1 (`sval_inject_val_inject`).

$$\begin{aligned}\forall\ \ & f\ m\ sv\ sv'\ v,\ \texttt{sval\_inject}\ f\ sv\ sv' \Rightarrow\\ & \texttt{normalise}\ m\ sv = v \Rightarrow \exists v', \texttt{val\_inject}\ f\ v\ v'.\end{aligned}$$

Proof. By definition of `sval_inject` we have for some sv_1 and sv_2

$$sv \equiv sv_1 \wedge \texttt{sval\_inj}\ f\ sv_1\ sv_2 \wedge sv_2 \equiv sv'.$$

Since the normalisation is invariant under $\equiv$ (by Theorem 7.1.2), we have $\texttt{normalise}\ m\ sv_1 = v$ and it remains to prove:

$$\texttt{sval\_inj}\ f\ sv_1\ sv_2 \Rightarrow \texttt{normalise}\ m\ sv_1 = v \Rightarrow \exists v', \texttt{val\_inject}\ f\ v\ v'.$$

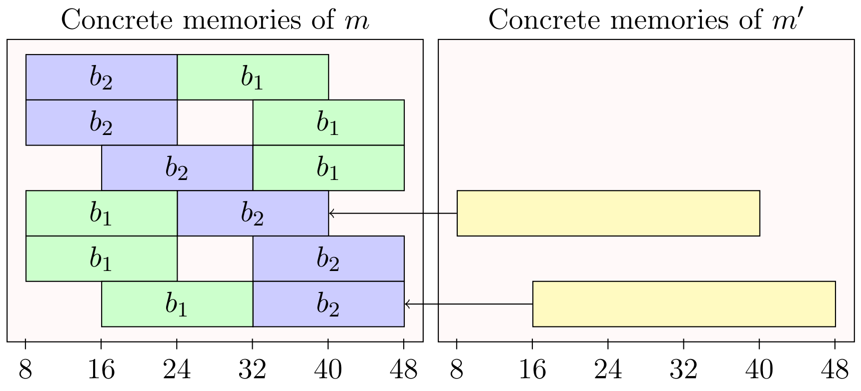


Figure 7.3: Injection of concrete memories

The proof is by case analysis over v, we exhibit a witness v' for each case such that `val_inject` $f\ v\ v'$.

- Case $v =$ `undef`. The witness we exhibit in this case is `undef`. By Rule VINJ-VUNDEF, the property holds.
- Case $v \neq$ `ptr`(b, i). The witness we exhibit in this case is v. By Rule VINJ-NO-PTR, the property holds.
- Case $v =$ `ptr`(b, i). From Lemma 4.4.1, we have that b appears syntactically in sv_1. By direct induction over `sval_inj` $sv_1\ sv_2$, it follows that $f(b) = \lfloor(b', \delta)\rfloor$ for some b' and δ. The witness we exhibit in this case is $v' =$ `ptr`$(b', i+\delta)$. By Rule VINJ-PTR, v' is in injection with v and the property holds.

□

7.2.3.2 Injection of concrete memories and indeterminate memories.

To go further in the proof of Theorem 7.2.2, we need to relate normalisations in m_1 and m_2 when m_1 and m_2 are in injection. Recall that the definition of normalisations involves a quantification over valid concrete memories and indeterminate memories. We therefore have to introduce new definitions regarding concrete memories and indeterminate memories and injections. To this end, we define the `cm_inject` and `im_inject` predicates. Definition 7.2.5 details those predicates: `cm_inject` relates concrete memories that associate the same concrete addresses to locations in injection and `im_inject` relates indeterminate memories that associate the same byte values to locations in injection.

Definition 7.2.5 (`cm_inject` and `im_inject`).

$$\begin{aligned}
&\texttt{cm\_inject}\ f\ cm\ cm' := \forall\ b\ b'\ \delta,\ f(b) = \lfloor(b', \delta)\rfloor \Rightarrow cm(b) = cm'(b') + \delta.\\
&\texttt{im\_inject}\ f\ im\ im' := \forall\ l\ l',\ \texttt{sval\_inj}\ f\ \textit{indet(l)}\ \textit{indet(l')} \Rightarrow im(l) = im'(l').
\end{aligned}$$

Figure 7.3 illustrates the injection of concrete memories. It shows on the left-hand side the set of valid concrete memories for some memory state m, and on the right-hand side the set of valid concrete memories for some memory state m' such that m and m' are in injection according to a function f. The effect of the injection is to group b_1 and b_2 together, in that order. One way to think about concrete memories injections is that it selects the valid concrete memories of m that have a corresponding valid concrete memory in m'.

Lemma 7.2.2 is a result about the evaluation of symbolic values in injection in concrete memories and indeterminate memories in injection.

Lemma 7.2.2 (`eval_sval_inject`). *For any injection f, and for any concrete memories cm and cm', for any indeterminate memories im and im', for any symbolic values sv and sv', if (cm, im, sv) and (cm', im', sv') are in injection by f component-wise, the evaluations $[\![sv]\!]_{cm}^{im}$ and $[\![sv']\!]_{cm'}^{im'}$ satisfy the* less-defined *relation. Formally,*

$$\begin{array}{l}\forall\ f\ cm\ cm'\ im\ im'\ sv\ sv',\\ \quad \texttt{cm\_inject}\ f\ cm\ cm' \Rightarrow\\ \quad \texttt{im\_inject}\ f\ im\ im' \Rightarrow\\ \quad \texttt{sval\_inject}\ f\ sv\ sv' \Rightarrow\\ \quad [\![sv]\!]_{cm}^{im} \leq [\![sv']\!]_{cm'}^{im'}.\end{array}$$

Proof. By definition of `sval_inject`, there exist sv_1 and sv_2 such that

$$sv \equiv sv_1 \tag{7.5}$$

$$sv' \equiv sv_2 \tag{7.6}$$

$$\texttt{sval\_inj}\ f\ sv_1\ sv_2 \tag{7.7}$$

After rewriting Hypotheses 7.5 and 7.6, it remains to show:

$$[\![sv_1]\!]_{cm}^{im} \leq [\![sv_2]\!]_{cm'}^{im'}.$$

The proof is by induction over the derivation of Hypothesis 7.7.

- Case $sv_1 = \texttt{undef}$. Then, $[\![sv_1]\!]_{cm}^{im} = \texttt{undef}$. By Rule LESSDEF-UNDEF, the property holds.

- Case $sv_1 = v$ and $sv_2 = v'$ where v and v' are values such that `val_inject` $f\ v\ v'$. We must prove that $[\![v]\!]_{cm}^{im} \leq [\![v']\!]_{cm'}^{im'}$. The proof is by case analysis over `val_inject` $f\ v\ v'$.

  – $v = \texttt{undef}$. Then $[\![v]\!]_{cm}^{im} = \texttt{undef}$ and by Rule LESSDEF-UNDEF, the property holds.

  – $v = \texttt{ptr}(b, i)$ and $v' = \texttt{ptr}(b', i + \delta)$ and $f(b) = \lfloor(b', \delta)\rfloor$. On one hand we have $[\![v]\!]_{cm}^{im} = cm(b) + i$, and on the other hand, we have $[\![v']\!]_{cm'}^{im'} = cm'(b') + (i + \delta)$. Because cm and cm' are in injection, we know that $cm(b) = cm'(b') + \delta$ and the property holds.

  – v and v' are neither `undef` nor pointers. In this case $v = v'$, and the evaluations of v and v' do not depend on the concrete memory and are therefore equal.

- Case $sv_1 = \texttt{indet}(b, i)$ and $sv_2 = \texttt{indet}(b', i + \delta)$ and $f(b) = \lfloor(b', \delta)\rfloor$. This case is similar to the case of pointers.

- Case $sv_1 = \texttt{op}_1\ sv_1'$ and $sv_2 = \texttt{op}_1\ sv_2'$ and `sval_inj` $f\ sv_1'\ sv_2'$. The induction hypothesis gives us:

  $$\forall cm \vdash m, \forall im, [\![sv_1']\!]_{cm}^{im} \leq [\![sv_2']\!]_{cm'}^{im'}$$

  We have on one hand $[\![sv_1]\!]_{cm}^{im} = [\![\texttt{op}_1\ sv_1']\!]_{cm}^{im} = \texttt{eval\_unop}(\texttt{op}_1, [\![sv_1']\!]_{cm}^{im})$ and on the other hand $[\![sv_2]\!]_{cm'}^{im'} = [\![\texttt{op}_1\ sv_2']\!]_{cm'}^{im'} = \texttt{eval\_unop}(\texttt{op}_1, [\![sv_2']\!]_{cm}^{im})$. The property holds because `eval_unop` is a morphism for $\leq$, *i.e.* for any symbolic values sv and sv' such that $sv \leq sv'$, we have $\texttt{eval\_unop}(\texttt{op}_1, sv) \leq \texttt{eval\_unop}(\texttt{op}_1, sv')$.

- Case $sv_1 = sv_3\ \texttt{op}_2\ sv_4$ and $sv_2 = sv_3'\ \texttt{op}_2\ sv_4'$. The property holds by application of the induction hypotheses using the same arguments as for the unary operators.

$\square$

7.2.3.3 Construction of concrete and indeterminate pre-memories.

We have seen that the `cm_inject` and `im_inject` predicates represent a selection of concrete memories and indeterminate memories. Namely, they capture those concrete and indeterminate memories that have an injection, that we call *pre-memories*. In this section we give algorithms to construct pre-memories. Let m and m' be two memories in injection by f. Given a concrete memory $cm' \vdash m'$ and an indeterminate memory im', the goal is to construct a concrete memory $cm \vdash m$ such that `cm_inject` $f\ cm\ cm'$ and an indeterminate memory im such that `im_inject` $f\ im\ im'$. Graphically, the function we seek is represented by the left arrows $\leftarrow$ in Figure 7.3.

The following algorithms satisfy these requirements.

$$cm(b) = \texttt{match}\ f(b)\ \texttt{with}\ \mid \lfloor(b',\delta)\rfloor \Rightarrow cm'(b') + \delta \mid \texttt{None} \Rightarrow 0\ \texttt{end}.$$
$$im(b,i) = \texttt{match}\ f(b)\ \texttt{with}\ \mid \lfloor(b',\delta)\rfloor \Rightarrow im'(b', i+\delta) \mid \texttt{None} \Rightarrow 0\ \texttt{end}.$$

Let us examine the construction of cm. To get the concrete address of block b, we first examine $f(b)$. If b is injected, *i.e.* there exist b' and δ such that $f(b) = \lfloor(b',\delta)\rfloor$, then the address of b is equal to the address of b plus the offset δ. If b is not injected, we have no constraint whatsoever about the concrete address of b for cm to be a pre-memory of cm', hence we give the default address 0. The construction of im is similar.

We will now prove properties about these constructions. In particular we will prove that the construction of cm yields a valid concrete memory for m. This requires as a precondition that the injection is total, *i.e.* that all non-empty blocks (*i.e.* those with a strictly positive size) are injected *i.e.* the injection function f is defined for all the allocated (non-empty) blocks.

Lemma 7.2.3. *Provided that cm' is a valid concrete memory for m', and provided that f is a total injection, the construction for cm yields a valid concrete memory for m. Formally,*

$$\begin{aligned}\forall\ f\ m\ m'\ cm\ cm',\ & \texttt{total\_injection}\ f\ m \Rightarrow \\ & \texttt{mem\_inject}\ f\ m\ m' \Rightarrow \\ & cm' \vdash m' \Rightarrow \\ & cm \vdash m\end{aligned}$$

Proof. We prove each of the three properties of $\vdash$ independently.

- **Address space.** The address space constraint unfolds into:

  $$\forall\ b\ o,\ \texttt{valid}(m,b,o) \Rightarrow cm(b) + o \in]0; 2^{32}-1[$$

  Because all non-empty blocks are injected and b is non-empty (because $\texttt{valid}(m,b,o)$), there exist a block b' and an offset δ such that $f(b) = \lfloor(b',\delta)\rfloor$. The goal becomes:

  $$cm'(b') + \delta + o \in]0; 2^{32}-1[$$

  Because m and m' are in injection, valid locations in m inject into valid locations in m', hence we have that $\texttt{valid}(m', b', o+\delta)$. As a consequence, the goal is solved by the address-space property from $cm' \vdash m'$.

- **No overlap.** The no-overlap constraint unfolds into:

  $$\begin{aligned}\forall\ b_1\ b_2\ o_1\ o_2,\ & b_1 \neq b_2 \Rightarrow \\ & \texttt{valid}(m,b_1,o_1) \Rightarrow \texttt{valid}(m,b_2,o_2) \Rightarrow \\ & cm(b_1) + o_1 \neq cm(b_2) + o_2\end{aligned}$$

By the same arguments as in the previous case, b_1 and b_2 are injected, and the valid locations inject into valid locations:

$$\begin{array}{cc} f(b_1) = \lfloor(b'_1, \delta_1)\rfloor & f(b_2) = \lfloor(b'_2, \delta_2)\rfloor \\ \texttt{valid}(m', b'_1, o_1 + \delta_1) & \texttt{valid}(m', b'_2, o_2 + \delta_2) \end{array}$$

The goal becomes:

$$cm'(b'_1) + \delta_1 + o_1 \neq cm'(b'_2) + \delta_2 + o_2$$

Because m and m' are in injection by f, we know that f is an injective function for valid locations, *i.e.* any two different valid locations are mapped to different valid locations. More precisely, we have that:

$$b'_1 \neq b'_2 \lor o_1 + \delta_1 \neq o_2 + \delta_2$$

The proof now goes by case analysis on the equality of blocks b'_1 and b'_2.

- Case $b'_1 = b'_2$. Then we know that $o_1 + \delta_1 \neq o_2 + \delta_2$, which solves our goal.
- Case $b'_1 \neq b'_2$. Our goal is solved by application of the non-overlap property from $cm' \vdash m'$.

• **Alignment constraints.**

The alignment constraint unfolds into:

$$cm(b) \mod 2^{\texttt{alignment}(m,b)} = 0$$

We proceed by case analysis on the result of $f(b)$.

- Case $f(b) = \emptyset$. In this case, $cm(b) = 0$, and the property holds.
- Case $f(b) = \lfloor(b', \delta)\rfloor$. The goal becomes:

$$(cm'(b') + \delta) \mod 2^{\texttt{alignment}(m,b)} = 0$$

We will prove on one hand that $cm'(b') \mod 2^{\texttt{alignment}(m,b)} = 0$ and on the other hand that $\delta \mod 2^{\texttt{alignment}(m,b)} = 0$.

  * Goal: $cm'(b') \mod 2^{\texttt{alignment}(m,b)} = 0$. From the fact that $cm' \vdash m'$, we have that: $cm'(b') \mod 2^{\texttt{alignment}(m',b')} = 0$. From the property `mi_align` of the injection of memories, we get that $\texttt{alignment}(m', b') \geq \texttt{alignment}(m, b)$. Since $2^{\texttt{alignment}(m,b)}$ divides $2^{\texttt{alignment}(m',b')}$, the property holds.
  * Goal: $\delta \mod 2^{\texttt{alignment}(m,b)} = 0$. From the fact that $cm' \vdash m'$, we have that: $2^{\texttt{alignment}(m,b)} \mid \delta$. Hence, the property holds.

□

Lemma 7.2.4. *The construction for cm yields a concrete pre-memory of cm'. Formally,*

$$\texttt{cm\_inject}\ f\ cm\ cm'.$$

Proof. The goal unfolds into: $\forall\ b\ b'\ \delta,\ f(b) = \lfloor(b', \delta)\rfloor \Rightarrow cm(b) = cm'(b') + \delta$.
This is a direct consequence of the definition of *cm*. □

Lemma 7.2.5. *The construction for im yields an indeterminate pre-memory of im'. Formally,*

$$\texttt{im\_inject}\ f\ im\ im'$$

Proof. The goal unfolds into: $\forall\ b\ b'\ \delta\ o,\ f(b) = \lfloor(b', \delta)\rfloor \Rightarrow im(b, o) = im'(b', o + \delta)$.
This is a direct consequence of the definition of *im*. □

7.2.3.4 Proof of the final theorem.

Lemmas 7.2.1, 7.2.2, 7.2.3, 7.2.4 and 7.2.5 play a major role in the proof of Theorem 7.2.2 whose statement is recalled below.

Theorem 7.2.2 (`norm_inject`). *Given a total injection function f, given two memory states m and m' in injection by function f, given two symbolic values sv and sv' in injection by f, the normalisations of sv in m and sv' in m' are in injection by f.*

$$\begin{array}{l} \forall \;\; f\ m\ m'\ sv\ sv', \texttt{total\_injection}\ f\ m \Rightarrow \\ \quad \texttt{mem\_inject}\ f\ m\ m' \Rightarrow \texttt{sval\_inject}\ f\ sv\ sv' \Rightarrow \\ \quad \texttt{val\_inject}\ f\ (\texttt{normalise}\ m\ sv)\ (\texttt{normalise}\ m'\ sv'). \end{array}$$

Proof. The proof is by case analysis over the result, say v, of the normalisation `normalise` $m\ sv$.

- Case $v = \texttt{undef}$. By Rule VINJ-VUNDEF, the property holds.
- Case $v \neq \texttt{undef}$. From Lemma 7.2.1, we can always construct a value v' such that

$$\texttt{val\_inject}\ f\ v\ v' \tag{7.8}$$

  To prove the property, it remains to show that v' is indeed the result of the normalisation of sv', *i.e.* that $sv' \overset{m'}{\twoheadrightarrow} v'$. We have to prove the following:

$$\forall\ cm' \vdash m', \forall im', [\![v']\!]_{cm'}^{im'} = [\![sv']\!]_{cm'}^{im'}.$$

  First, we use Lemmas 7.2.3, 7.2.4 and 7.2.5 to get:

$$cm \vdash m \tag{7.9}$$

$$\texttt{cm\_inject}\ f\ cm\ cm' \tag{7.10}$$

$$\texttt{im\_inject}\ f\ im\ im' \tag{7.11}$$

  We can now use Lemma 7.2.2 for sv and sv' on one hand and for v and v' on the other hand to get the following:

$$[\![sv]\!]_{cm}^{im} \leq [\![sv']\!]_{cm'}^{im'} \tag{7.12}$$

$$[\![v]\!]_{cm}^{im} \leq [\![v']\!]_{cm'}^{im'} \tag{7.13}$$

  By Definition 4.4.1 (sound normalisation), we get:

$$[\![v]\!]_{cm}^{im} = [\![sv]\!]_{cm}^{im} \tag{7.14}$$

  Because $v \neq \texttt{undef}$ and Hypotheses 7.12, 7.13 and 7.14, we have

$$[\![v']\!]_{cm'}^{im'} = [\![v]\!]_{cm}^{im} = [\![sv]\!]_{cm}^{im} = [\![sv']\!]_{cm'}^{im'}$$

  As a result, the property holds.

□

7.3 Conclusion and Discussion

The memory is a central component of the semantic states used by all intermediate languages in CompCert. It is thus very important to design convenient ways of reasoning about memory states. In particular, the proof of semantic preservation of CompCert is based on forward simulation arguments, and it is crucial to design matching relations about memory states that capture the transformations that happen in CompCert passes. In CompCert, those relations are memory extensions and memory injections.

We generalised the memory extension relation into a memory improvement relation, that is tighter than extensions in CompCert, because the structure of the memory is forced to be identical in both memories. This constrains the normalisation to be the same in both memories.

We generalised the memory injection to symbolic values. This demanded a generalisation of injection of values, in particular for indeterminate values. We also gave a central theorem linking injections and normalisations. The proof of this theorem required to introduce new notions of injections of concrete memories and indeterminate memories.

Note that Theorem 7.2.2 holds only for total injections, *i.e.* injection functions f that satisfy the `total_injection` predicate. Almost all transformations of CompCert that rely on memory injections feature only total injections. However, the `SimplLocals` pass uses inherentely partial injection functions, because the purpose of this pass is to remove some variables from the memory. We can therefore not use Theorem 7.2.2 for the proof of `SimplLocals`. However, we will see in Section 8.1 that the partial injections used in this pass are of a special kind that enables to prove a version of the theorem that is usable for the correctness proof.

Finding the right generalisation for the various memory relations has not been an instantaneous process. Before we came up with a nice and simple notion of structure-preserving memory relation parameterised by an underlying relation over symbolic values, we proceeded by trial and error in defining *ad hoc* memory invariants tailored for each compilation pass. One of the unfruitful tries consisted for example in relating symbolic values by the equality of their normalisations in their respective memories, as defined in the `match_val` predicate below:

$$\texttt{match\_val}\ m_1\ m_2\ sv_1\ sv_2 \triangleq \texttt{normalise}\ m_1\ sv_1 = \texttt{normalise}\ m_2\ sv_2$$

This relation is not well-suited for the simulation proofs we have to prove because it depends on the memory states m_1 and m_2, and therefore every time the memory states change, one needs to prove that the predicate still holds for the updated memory states. In addition to the added proof burden that this represents, the predicate is simply not preserved by the `free` operation. Indeed, we cannot relate the normalisations of a symbolic value in a memory m and in a memory m' obtained by freeing some blocks. The reason is that there are more concrete memories that are valid after the `free` operation has been performed, hence the normalisation (see Definition 4.4.1) needs to quantify over a larger set of concrete memories and we have therefore no insurance that the normalisations are preserved. This discussion is similar to what we discussed in Section 6.2, where we tried to introduce normalisations in memory invariants.

Similarly for memory injections, our first attempts at generalising injections were more limited than what we have achieved now. For example, we only had syntactic injections, *i.e.* only symbolic values with the same *structure* could be in injection, akin to the `sval_inj` predicate. Moreover, at first, we did not account for indeterminate values and it was unclear how to inject those. Proving theorems about our injections and how they relate to normalisations has uncovered a number of problems and has helped make our injections better.

Chapter 8

Semantic Preservation Of The Compiler Passes

The correctness of the CompCert compiler relies on the correctness of the individual compiler passes, most of which are proved as forward simulation theorems, as described in Section 2.4, except for the first determinisation pass which needs to be proved directly as a backward simulation. The individual correctness proofs rely extensively on the theorems of the memory model (*good-variable properties*), and their preservation by memory relations and injections. Because CompCert provides all the infrastructure required for the proof of correctness of the compiler passes, we aim at reusing the existing proofs as much as possible.

Most proofs can be straightforwardly adapted from the existing proofs in CompCert by generalising memory extensions into memory improvements and memory injections into our generalised memory injections. However, a number of compiler passes need more work to be adapted to CompCertS, with symbolic values and finite memory.

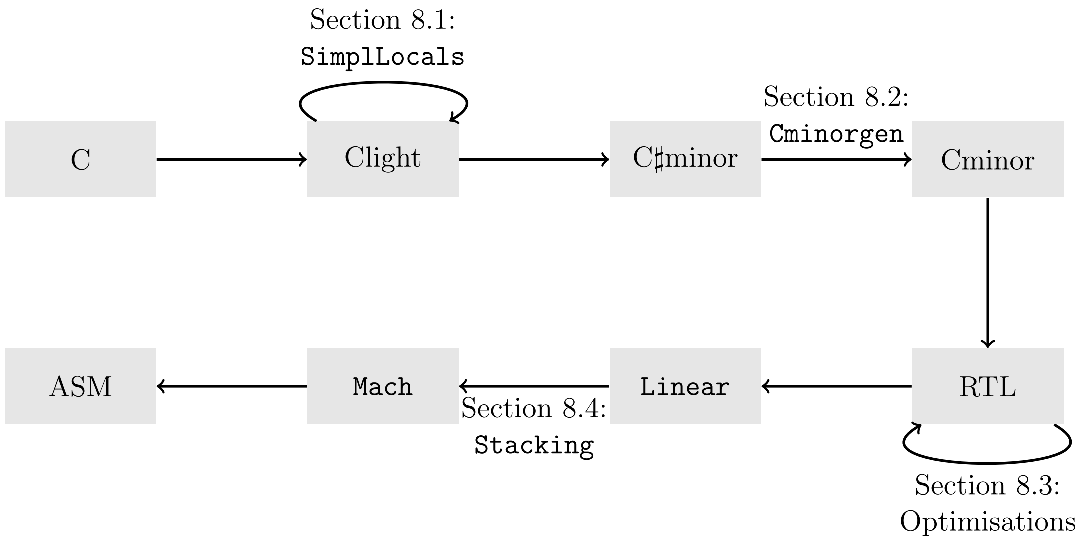


Figure 8.1: Compiler passes of CompCert that require substantial work.

This chapter reports on the compiler passes for which substantial work is required and describes the adjustments we make. We focus successively on four different passes, represented on Figure 8.1 (for clarity, not all languages are shown here). Section 8.1 reports on the `SimplLocals` pass, that pulls out of memory scalar variables whose address is not taken, and generates temporary variables instead. This is challenging because the existing proof uses partial injections, that do not fit in the framework imposed by Theorem 7.2.2. Then, we show in Section 8.2 how to adapt the `Cminorgen` pass that most notably builds stack frames for every function, using our generalised notion of memory injections. Then, Section 8.3 shows how the abstract domains used in the constant propagation and common subexpression elimination optimisations have to be updated to fit in our new framework. Finally, we show in Section 8.4 that the `Stacking` pass, which performs notably register spilling, requires major adjustments to be proved correct in our finite memory model, because of the decreasing memory usage constraint.

8.1 Generation Of Temporaries

The second compiler pass of CompCert, `SimplLocals`, is a transformation of Clight programs. It pulls out of memory scalar local variables whose address is not taken in the program. These variables are transformed into *so-called* temporary variables (or *temporaries*), that do not reside in memory. This transformation is crucial because subsequent optimisations at the RTL level operate on these temporaries. We first explain further the transformation performed by this pass; then we explain the arguments for the correctness proof of this pass in CompCert. Last, we explain why the memory injections we have generalised in Section 7.2 are not well-suited for the proof of this transformation, and we propose a solution to generalise further memory injections.

8.1.1 Description of the transformation

Figure 8.2 illustrates the `SimplLocals` pass. It shows a C function `f` with two local variables `x` and `y` on the left-hand side. The address of `x` is used as a parameter of a call to another function `g`. The transformed program on the right-hand side also has two variables, however `y` does not reside in memory anymore but in a *temporary variable* instead, as indicated by the keyword `var`. This transformation is allowed because the address of `y` is never taken in the original program, *i.e.* `&y` never occurs and therefore it can be pulled out of memory. This generation of temporary variables is important because the optimisations such as constant propagation operate on temporary variables and not on variables that reside in the memory. For instance, during the further constant propagation pass, the temporary variable `y` will be replaced by its value 11 (see Section 8.3).

```
int f() {                              int f() {
 int x = 7;        SimplLocals          int x; var y;
 int y = 11;     ─────────────→         x = 7; y  = 11;
 g(&x);                                 g(&x);
 return y;                              return y;
}                                      }
```

Figure 8.2: A Clight function (left) transformed by the `SimplLocals` pass (right)

This transformation is based on a syntactic analysis of the code of individual functions. The result of the analysis is the set of local variables whose address is taken in the code of the function (*i.e.* all local variables `x` such that `&x` appear in the code of the function). Then, accesses to local variables are transformed, if need be, into accesses to temporaries.

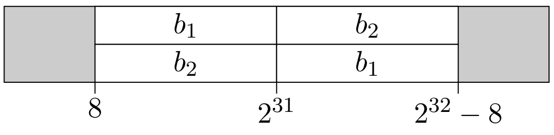
8.1.2 Correctness arguments

The correctness proof of the `SimplLocals` pass relies on the assumption that the modifications to a given variable `x` are either performed inside the considered function *via* a direct access to `x`, or indirectly through the address of this variable (`&x`). An immediate corollary is that local variables whose address is never taken are not leaked to other functions and we can reason locally about these variables; thus it is legitimate to pull those variables out of memory.

The fact that a pointer to `x` must be passed to the external function `g` so that `g` can access `x` holds because a program can not *forge* a pointer from nothing. This property is true in CompCert, and it is also true in CompCertS. In particular, we prove that the normalisation function does not forge pointers. This holds in particular thanks to Theorem 4.4.1, which states that the normalisation of a symbolic value sv may only result in a pointer $\texttt{ptr}(b, o)$ if block b appears syntactically in sv.

It is interesting to note that this property did not hold in early versions of our development, in which the normalisation function did not satisfy Theorem 4.4.1. Indeed, in very constrained memory states, a symbolic value sv could normalise into a pointer $\texttt{ptr}(b, o)$, even if b did not appear syntactically in sv. This was a discrepancy due to *near out-of-memory* situations, where very few concrete memories are valid, and one could deduce the address of a block from a pointer to a different block. Example 8.1.1 illustrates this situation.

Example 8.1.1. *Consider m with 2 blocks b_1 and b_2 (coloured in white) of size $sz = 2^{31} - 8$ that need to be 8-byte aligned (*i.e.* the 3 least significant bits of their address are zeros). As the range of valid addresses excludes 0 and $2^{31} - 1$, there are only 2 possible concrete memory configurations, depicted in the following figure: either b_1 is allocated at address 8 and b_2 is allocated at 2^{31} (first concrete memory) or b_2 is allocated at address 8 and b_1 is allocated at 2^{31} (second concrete memory). In this situation, the pointer $(b_2, 0)$ can be forged by the conditional expression $(b_1 == 8)?(b_1 + sz) : 8$.*[1]



Fortunately, this situation is solved in the current implementation, thanks to Property 4.4.1 (Sliding Blocks) which prevents such constrained, near *out-of-memory*, abstract memories.

8.1.3 Proof of `SimplLocals` in CompCertS

The existing proof of this pass in CompCert uses memory injections to relate memory states before and after the transformation. However, since the transformation pulls some variables out of memory, it is a *partial* injection where some blocks may not be injected, *i.e.* they may be *forgotten*. Such a situation is illustrated on the example program of Figure 8.2. The memory states relevant for function `f` are shown in Figure 8.3.

To perform the proof of correctness of this pass, we need a theorem akin to Theorem 7.2.2 that relates the normalisations of symbolic values in the memory states before and after injection. We wish to prove a theorem of the following form:

[1]Symbolic values do not actually include ternary conditions `a?b:c`, however it can be encoded using the fact that conditions evaluate to either 0 or 1 as `a*b + !a*c` when `b` and `c` are of integer type.

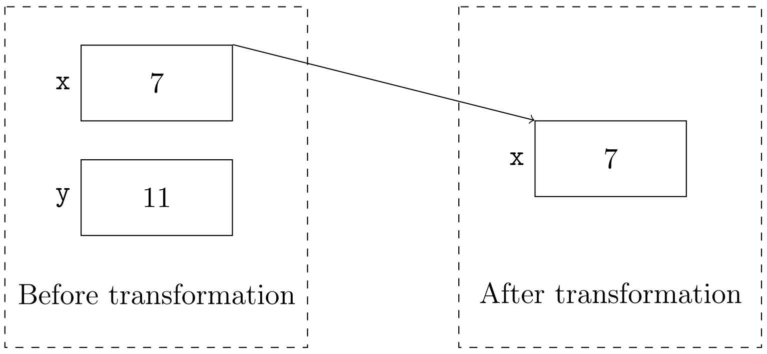
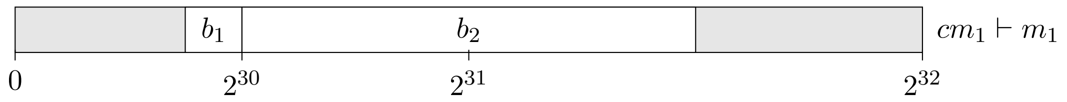


Figure 8.3: Partial memory injection

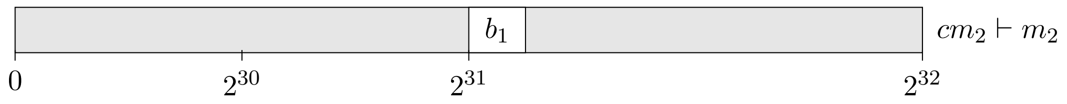
$$\forall\ f\ m_1\ m_2\ sv_1\ sv_2, \texttt{mem\_inject}\ f\ m_1\ m_2 \Rightarrow \texttt{sval\_inject}\ f\ sv_1\ sv_2 \Rightarrow \texttt{val\_inject}\ f\ (\texttt{normalise}\ m_1\ sv_1)\ (\texttt{normalise}\ m_2\ sv_2)$$

We have seen in Section 7.2 that Theorem 7.2.2 holds for injections functions that are total, *i.e.* all non-empty blocks are injected. We have already established that this is not the case of the injection we need for the `SimplLocals` pass. An intuitive way to understand why the theorem does not hold for arbitrary injection functions f is the following. If f *forgets* some blocks from m_1, *i.e.* if they are not injected into m_2, then there are fewer constraints for concrete memories cm_2 to be valid for m_2. As a result, there are *more* concrete memories valid for m_2 than there are for m_1. Because the specification of the normalisation is defined by the equality of the evaluations in all valid concrete memories, it is more likely to fail when there are more concrete memories. Therefore, the normalisation could be defined in m_1 and undefined in m_2, contradicting the theorem we wish to prove. We exhibit in Example 8.1.2 a situation in which the normalisation becomes *less defined* after injection.

Example 8.1.2. *Consider a memory state m_1 containing one unaligned block b_1 of size 1 and one 8-byte-aligned block b_2 of size 2^{31}. The following concrete memory cm_1, for example, is valid for m_1:*



In m_1, the block b_1 may never be assigned the concrete address 2^{31} because it is always part of block b_2. Now consider an injection function f such that $f(b_2) = \emptyset$. In m_2, the block b_1 may now be assigned any concrete address, in particular 2^{31}. Consider for example the following concrete memory cm_2, valid for m_2.



As a result, the symbolic value $sv = \boldsymbol{ptr}(b_1, 0)! = \boldsymbol{int}(2^{31})$ normalises to `true` in m_1 because b_1 is never *allocated at address 2^{31}, but does not normalise in m_2 because sv has different evaluations in different valid concrete memories for m_2.*

In the following, we first state a lower-level property, about concrete memories and memory injections, that is sufficient to prove the desired theorem about normalisations and injections. Then we show that the injection functions we need to consider for this correctness proof are not arbitrary, but conform to some well-formedness properties that rule out the undesired behaviour we just described. Finally, we give a sketch of the proof of the theorem.

Property 8.1.1 is a condition over an injection function f that is sufficient to prove the preservation of normalisations by injections, as was explained in Section 7.2.3.

Property 8.1.1. *For every memory states m_1 and m_2 in injection by f, for any concrete memory cm_2 valid for m_2, there exists a concrete memory cm_1 valid for m_1 such that cm_1 and cm_2 are in injection. Formally,*

$$\begin{aligned}&\forall\ m_1\ m_2,\ \texttt{mem\_inject}\ f\ m_1\ m_2 \rightarrow\\ &\quad \forall\ cm_2,\ \ cm_2 \vdash m_2 \rightarrow\\ &\quad \exists\ cm_1,\ \ cm_1 \vdash m_1 \wedge \texttt{cm\_inject}\ f\ cm_1\ cm_2\end{aligned}$$

The intuition behind Property 8.1.1 is that we should always be able to build a concrete pre-memory of any concrete memory valid for an injected memory. The construction we proposed for this in Section 7.2.3 does not work in our case, because it was restricted to total injections.

8.1.3.1 Restrictions over injection functions.

We restrict ourselves to injection functions f such that the blocks that are forgotten are less than 8-byte wide. In the particular case of the `SimplLocals` pass, the blocks that are forgotten correspond to scalar variables, hence their maximal size is indeed 8 bytes (for `long`-typed variables). We formalise this notion in Definition 8.1.1.

Definition 8.1.1. *Given a memory state m, an injection function f is 8-forgetful for m if the blocks forgotten,* i.e. *not injected, by f have a size not greater than 8 bytes. Formally,*

$$\texttt{forgetful}\ f\ m\ \triangleq\ \forall\ b,\ \ f(b) = \emptyset \Rightarrow \texttt{size}\ m\ b \leq 8$$

Another restriction we impose on the injection is that f is a *one-to-one* injection, *i.e.* blocks are not merged together but merely kept or forgotten. Definition 8.1.2 formalises this intuition.

Definition 8.1.2. *An injection function f is* one-to-one *if any block b is either injected into itself or is not injected at all. Formally,*

$$\texttt{one\_to\_one}\ f\ \triangleq\ \forall\ b, f(b) = \lfloor(b', \delta)\rfloor \Rightarrow \delta = 0$$

We can now restate Property 8.1.1 with the appropriate hypotheses in Theorem 8.1.1.

Theorem 8.1.1. *For every injection function f, for every memory states m_1 and m_2 in injection by f, if f is 8-forgetful for m_1 and f is one-to-one, then for any concrete memory cm_2 valid for m_2, there exists a concrete memory cm_1 valid for m_1 such that cm_1 and cm_2 are in injection. Formally,*

$$\begin{aligned}\forall\ f\ m_1\ m_2,\ &\texttt{forgetful}\ f\ m_1 \rightarrow\\ &\texttt{one\_to\_one}\ f\\ &\texttt{mem\_inject}\ f\ m_1\ m_2 \rightarrow\\ &\forall\ cm_2,\ \ cm_2 \vdash m_2 \rightarrow\\ &\exists\ cm_1,\ \ cm_1 \vdash m_1 \wedge \texttt{cm\_inject}\ f\ cm_1\ cm_2\end{aligned}$$

Using this theorem, it is straightforward to reuse the proof we presented in Section 7.2.3.

8.1.3.2 Proof sketch for Theorem 8.1.1 about forgetful injections.

This section aims at giving a proof sketch for Theorem 8.1.1.

Consider a one-to-one 8-forgetful injection f, two memory states m_1 and m_2 in injection by f and a concrete memory cm_2 valid for m_2. Consider three blocks b_1, b_2 and b_3 4-byte-wide and 4-byte-aligned and two blocks b_4 and b_5 that are 2-byte-wide and 2-byte-aligned in m_1. Then, consider that f does not inject blocks b_4 and b_5. This situation is depicted by Figure 8.4.

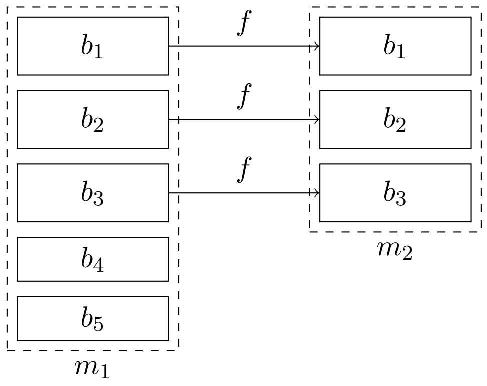


Figure 8.4: One-to-one forgetful injection.

Our goal is to prove that for every concrete memory cm_2 valid for m_2, it is possible to construct a concrete memory cm_1 valid for m_1 such that `cm_inject` f cm_1 cm_2.

We write `next_addr` m cm for the first maximally aligned address that follows the last (greatest) mapped address in the concrete memory cm (where cm is such that $cm \vdash m$). Recall that `size_mem` m computes the size of a maximally aligned concrete memory for m, as constructed by the allocation algorithm presented in Section 5.4. We write $cm \Vdash m$ to capture *maximally aligned* valid concrete memories. In the particular case where $cm \Vdash m$, we have that `next_addr` m cm = `size_mem` m.

Consider a concrete memory cm_2 valid for m_2. The construction of cm_1 such that $cm_1 \vdash m_1$ and `cm_inject` f cm_1 cm_2 will be different depending on which of `next_addr` m_2 cm_2 or `size_mem` m_2 is greater.

If `size_mem` $m_2 \geq$ `next_addr` m_2 cm_2, the situation is depicted by Figure 8.5.

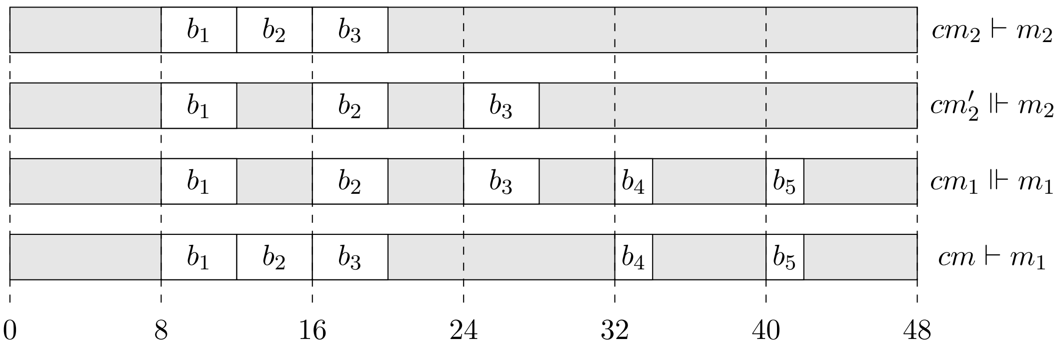


Figure 8.5: Constructing $cm \vdash m_1$ from cm_2: `size_mem` $m_2 \geq$ `next_addr` m_2 cm_2

The first line of Figure 8.5 shows the concrete memory cm_2 that we start from. We can see that `next_addr` m_2 cm_2 = 24. The second line of the figure shows a concrete memory $cm'_2 \Vdash m_2$ where the blocks are allocated in the same order however at maximally aligned addresses. This construction is the result of running the allocation algorithm (see Section 5.4). It exhibits that `size_mem` m_2 = `next_addr` m_2 $cm'_2 = 32$. Likewise, we can construct a concrete memory $cm_1 \Vdash m_1$ (shown on the third line of the figure) such that the blocks that are not forgotten by f are allocated first. Finally, the desired concrete memory cm can be constructed using the following algorithm:

$$cm(b) = \begin{cases} cm_2(b') & \text{if } f(b) = \lfloor (b', \delta) \rfloor \\ cm_1(b) & \text{otherwise} \end{cases}$$

It is straightforward that cm is such that `cm_inject` f cm cm_2 (from the definition of `cm_inject`). Proving that cm is valid for m_1 requires more reasoning. The alignment and address space constraints are easily inherited from the validity of cm_1 and cm_2. The proof of non-overlap follows from the validity of cm_1 and cm_2, but also from the fact that forgotten and not-forgotten blocks have been mapped to disjoint regions of the memory (before and after `size_mem` m_2).

If `next_addr` m_2 cm_2 > `size_mem` m_2, the construction is more delicate. The situation is depicted by Figure 8.6.

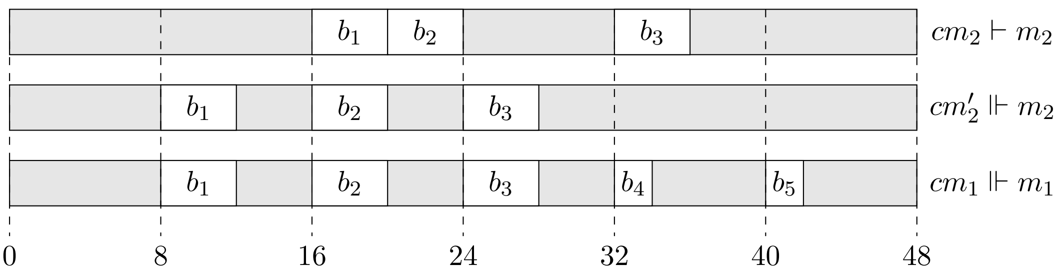


Figure 8.6: Inverting partial injections.

Like in the previous case, it is possible to construct from cm_2, a concrete memory $cm'_2 \Vdash m_2$ and a concrete memory $cm_1 \Vdash m_1$ that have the same properties as before. However, this time, we have that `next_addr` cm_2 $m_2 = 40$ and `size_mem` m_2 = `next_addr` m_2 $cm'_2 = 32$.

We call a *box* an 8-byte-wide 8-byte-aligned region of memory. Because the blocks that have been forgotten are smaller than 8 bytes and with alignment constraints smaller than 8 bytes, every such block fits in a box.

We write `num_free_boxes` m cm for the number of free (unmapped) boxes in concrete memory cm, up to address `next_addr` m cm. In our example, `num_free_boxes` m_2 cm_2 = 2 because there are two available boxes (from 8 to 16 and from 24 to 32). The box starting at address 0 is not taken into account because 0 is not a valid address. Theorem 8.1.2 gives an important result regarding the number of available boxes in concrete memories.

Theorem 8.1.2. *For any memory m and any concrete memory cm valid for m, if*

$$\texttt{next\_addr}\ m_2\ cm_2 > \texttt{size\_mem}\ m_2$$

then there are at least $N = \frac{\texttt{next\_addr}\ m\ cm - \texttt{size\_mem}\ m}{8}$ *available boxes in cm up to address* `next_addr` *m cm,* i.e. `num_free_boxes` *m $cm \geq N$.*

In our example, Theorem 8.1.2 states that there is at least $\frac{40-32}{8} = 1$ available box in cm_2 below address 40. An intuitive explanation of Theorem 8.1.2 is the following: if a valid concrete memory uses more space than a maximally aligned concrete memory, then it must have left a certain number of available *boxes* between allocated blocks.

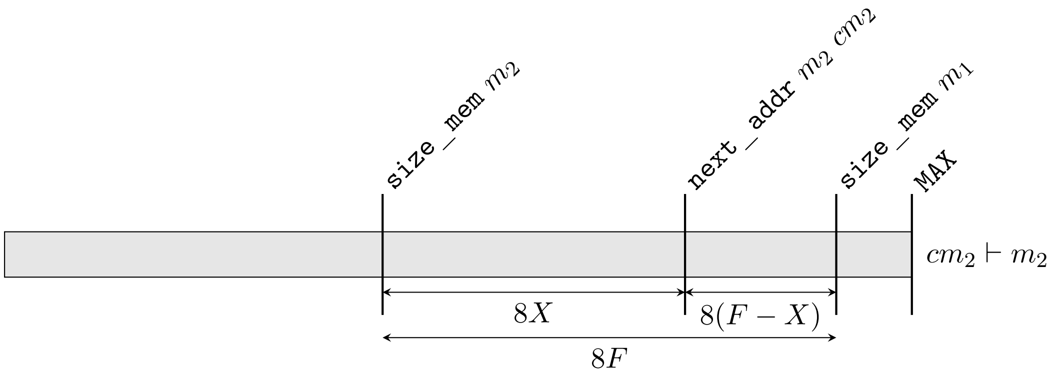


Figure 8.7: The concrete memory cm_2 ends after `size_mem` m_2.

Figure 8.7 illustrates the situation. Consider that F blocks have been forgotten from m_1, hence we have the following where `MAX` is the maximal concrete address $(2^{32}-1)$:

$$\texttt{size\_mem}\ m_2 + 8F = \texttt{size\_mem}\ m_1 < \texttt{MAX}$$

Besides, because of the inequality, and because `size_mem` and `next_addr` always return 8-byte-aligned addresses, we have that, for some natural number X:

$$\texttt{next\_addr}\ m_2\ cm_2 = \texttt{size\_mem}\ m_2 + 8X$$

We now split the F variables we forgot into two parts: one of X variables and the other $(F-X)$ variables. We will insert each part independently.

- **The X first variables.** Theorem 8.1.2 states that there are X available boxes in cm_2 before address `next_addr` m_2 cm_2 :

  $$\frac{\texttt{next\_addr}\ m_2\ cm_2 - \texttt{size\_mem}\ m_2}{8} = \frac{8X}{8} = X$$

- **The remaining $(F-X)$ variables.** As we can see on Figure 8.7, we have the following inequality:

  $$\texttt{next\_addr}\ m_2\ cm_2 + 8(F-X) < \texttt{MAX}$$

  It follows that $(F-X)$ boxes are available after `next_addr` m_2 cm_2.

As a result, all forgotten blocks can be reinjected into a concrete memory $cm \vdash m_1$ such that `cm_inject` f cm cm_2. This finishes the proof of Theorem 8.1.1, which we needed for the proof of our theorem relating normalisations and injections for partial injections.

8.2 Construction Of Stack Frames

From the proof point of view, the compiler pass `Cminorgen`, from C♯minor to Cminor, is particularly challenging for our model. The reason is that this particular pass is responsible for allocating the stack frame: therefore, it transforms significantly the memory layout and thus the memory accesses. After the transformation, the stack frame is a single block and local variables are accessed via offsets within this block. The proof introduces a memory injection stating how the blocks representing local variables in C♯minor are mapped into the single block representing the stack frame in Cminor. Chronologically, `Cminorgen` is the first pass relying on memory injections that we have adapted to CompCertS. It is an important milestone towards validating the adequacy of our generalisation of memory injections.

8.2.1 Description of the transformation

The `Cminorgen` pass is responsible for allocating the stack frame of functions. In C♯minor, each local variable is allocated in its own block. The goal of this transformation is to allocate a single block, that will be referred to as *the stack block*, and through which the individual variables will be accessed.

```
int main(){
  int x, y, *z;
  x = 2;
  z = &y;
  *z = x + 1;
  return y;
}
```

```
int main(){
  int locals[3];
  locals[0] = 2;
  locals[2] = &locals[1];
  *locals[2] = locals[0] + 1;
  return locals[1];
}
```

Figure 8.8: A C♯minor function (left) and its Cminor compilation

Figure 8.8 shows a C♯minor function on the left-hand-side that contains three local variables[2] x, y and z. On the right-hand-side, the compiled function no longer has three variables but an array, called `locals`, in which the variables are stored. Accesses to `x` for example are transformed into accesses to `locals[0]`. The memory states of both programs before the return instruction are depicted by Figure 8.9.

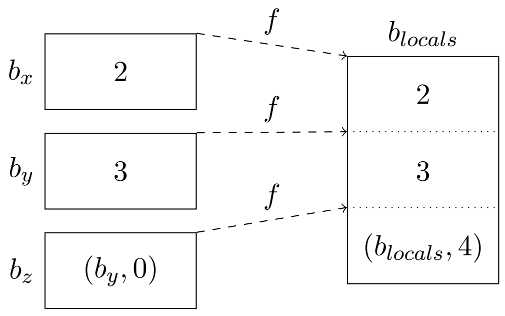


Figure 8.9: Memory injection for `Cminorgen`

8.2.2 Adaptation of the existing proof

The existing proof maintains a simulation relation based on memory injections between states of the C♯minor and Cminor programs at each step. Using the generalisation of

[2]This C♯minor program would not actually appear in the compilation of CompCert because x and z, whose addresses are not taken, should have been pulled out of memory by the `SimplLocals` pass. However, it is an illustrative example for this pass and it would suffice to artificially take the addresses of all the variables to make this program a valid output of the previous passes.

memory injections that we introduced in Section 7.2, it is possible to adapt the existing proof while keeping most of its structure. However, there are major differences between our proof and that of CompCert. Those differences are essentially due to our finite memory model. The differences appear in intermediate lemmas related to allocation and de-allocation, the proof of which cannot be reused in our symbolic model.

8.2.2.1 Preservation of injection by allocation

The first problem is the preservation of the memory injection when allocating the variables in C♯minor and the stack frame in Cminor. We first recall the corresponding lemma in CompCert and the structure of its proof. Then we highlight the changes that we need to make to this lemma and its proof in our finite memory model.

The existing lemma and its proof.
In CompCert, the lemma named `match_callstack_alloc_variables` aims at proving the preservation of injection by allocation. The function `stack_size` *vars* gives the size of the stack frame needed to store the variables *vars*. Here we show a simplified but representative version of the lemma that focuses only on the memory states.

$$\begin{array}{l}\forall\ f\ m_1\ m_2\ m_1'\ m_2'\ sp\ vars,\\ \quad \texttt{mem\_inject}\ f\ m_1\ m_2 \Rightarrow\\ \quad \texttt{alloc\_variables}\ m_1\ vars = \lfloor m_1' \rfloor \Rightarrow\\ \quad \texttt{alloc}\ m_2\ (\texttt{stack\_size}\ vars) = (m_2', sp) \Rightarrow\\ \quad \exists\ f',\ \texttt{mem\_inject}\ f'\ m_1'\ m_2'.\end{array}$$

The lemma states that given two memory states m_1 and m_2 in injection by f, allocating the local variables *vars* in m_1 and the stack frame of size `stack_size` *vars* in m_2 results in memory states m_1' and m_2' that are in injection by another injection function f'. This is one of the most important results for the correctness of the `Cminorgen` pass.

$$\begin{tikzcd}
m_1 \arrow[r, "\texttt{mem\_inject}\ f"] \arrow[dr, dashed, "\texttt{mem\_inject}\ f"'] & m_2 \arrow[d, "①"] \\
m_1 \arrow[d, "②"'] \arrow[dr, "\texttt{mem\_inject}\ f"] & m_2' \\
m_1' \arrow[r, dashed, "\exists f', \texttt{mem\_inject}\ f'"'] & m_2'
\end{tikzcd}$$

1. allocation of stack frame
2. allocation of local variables

Figure 8.10: Structure of `match_callstack_alloc_variables`'s proof in CompCert

The structure of the original proof is depicted in Figure 8.10 where plain arrows represent hypotheses and the dotted arrow the conclusion. The existing proof first establishes that the existing injection between the initial memories m_1 and m_2 still holds between m_1 and m_2'. In a second step, it constructs an injection between m_1' and m_2', thus concluding the proof.

Our modified version of the lemma. Because of our finite memory model, we need to modify the lemma so that it uses the `palloc` allocation function (that may fail) instead of CompCert's `alloc`. The theorem we need to prove becomes the following:

$$\begin{array}{l}\forall\ f\ m_1\ m_2\ m_1'\ m_2'\ sp\ vars,\\ \quad \mathtt{mem\_inject}\ f\ m_1\ m_2 \Rightarrow\\ \quad \mathtt{alloc\_variables}\ m_1\ vars = \lfloor m_1' \rfloor \Rightarrow\\ \quad \mathtt{palloc}\ m_2\ (\mathtt{stack\_size}\ vars) = \lfloor (m_2', sp) \rfloor \Rightarrow\\ \quad \exists\ f',\ \mathtt{mem\_inject}\ f'\ m_1'\ m_2'.\end{array}$$

Note that the success of `palloc` is stated as an hypothesis. It can be proved from the other hypotheses. Indeed, recall that the allocation succeeds if and only if the size of the resulting memory is less than some threshold `MAX`. We know that the allocation succeeds in m_1 and that the size of m_1 is greater than or equal to that of m_2 (because of the injection). Besides, since `stack_size` *vars* $\leq$ `sz_vars` *vars*, we conclude that the allocation succeeds in m_2 as well.

In order to prove that the injection is preserved by the allocations of the local variables on one hand and the stack frame on the other hand, we need to show that the memory state after injection uses fewer (or the same amount of) memory space. Because of this additional property of injections, we cannot use the original two-step proof, because the intermediate memory injection (`mem_inject` $f\ m_1\ m_2'$) does not hold in our memory model. We therefore perform the proof by direct induction on the number of allocated variables. The idea is the following: if we have already proved an injection f for some list of variables *vars*, we need to prove an injection f' for the list *var* :: *vars*. The injection f' is obtained by updating f to inject the block associated to *var* in the stack frame at the first available offset. Because the size of the stack frame depends on the list of variables, we can prove that the relative sizes of the memory states are preserved by each induction step.

8.2.2.2 Preservation of injection by deallocation

Symmetrically, at function exit, the variables and the stack frame are freed from memory. The lemma we wish to prove states that any injection that holds between memory states before the deallocations, still holds after. The corresponding lemma in CompCert is called `match_callstack_freelist` and is of the following form:

$$\begin{array}{l}\forall\ f\ m_1\ m_2\ vars\ sp\ m_1',\\ \quad \mathtt{mem\_inject}\ f\ m_1\ m_2 \Rightarrow\\ \quad \mathtt{free\_variables}\ m_1\ vars = \lfloor m_1' \rfloor \Rightarrow\\ \quad \exists\ m_2',\ \mathtt{free}\ m_2\ sp = \lfloor m_2' \rfloor \wedge \mathtt{mem\_inject}\ f\ m_1'\ m_2'.\end{array}$$

Similarly to the lemma about allocation and injection that we described previously, the original proof is a two-step proof using intermediate injections which do not hold in our model due to the size of the memories. The success of the `free` operation is guaranteed in the same way it was in the original proof in CompCert (namely, it follows from the fact that the `free_variables` operation succeeds in m_1 and that m_1 and m_2 are in injection).

The proof of the injection (`mem_inject` $f\ m_1'\ m_2'$) is more involved, in particular because of the additional property that the size of m_2' must be less than or equal to the size of m_1'. A key insight for proving this is the following: the memory states m_1' and m_2' (after the `free` operation) have the same sizes as the memory states before the allocation of the variables and stack frame, and those memory states were in injection and therefore satisfied the decreasing size constraint.

This intuition needs to be formalised and maintained as an invariant throughout the simulation proof. We use the existing notion of call stack, which is a list of function frames. A frame, as defined in the original proof of this pass in CompCert, is a proof artifact relating C♯minor and Cminor program states. For our concerns, we will model a frame as a record containing a field `sz_vars` and a field `sz_stk` that return respectively the size of the C♯minor variables and the size of the Cminor stack frame.

$$\textit{frame} ::= \{\texttt{sz\_vars} : \mathbb{Z}; \texttt{sz\_stk} : \mathbb{Z}\}$$

The predicate `size_history : list mem -> list mem -> callstack -> Prop` remembers a history of the sizes of the memory states and it captures the intuition that before allocating the local variables and the stack frame, the Cminor memory state was already smaller than or the same size as the C♯minor memory state. The predicate maintains two stacks of memory states (one for C♯minor memory states and the other for Cminor memory states). The third parameter is a call stack and is used to link a memory state to the next on the stack. The left-hand side of Figure 8.11 gives the definition of `size_history` as inference rules. The right-hand side represents the sizes of related memories. Stacked rectangles represent memory states, building on top of each other. The information that the `size_history` predicate captures is depicted by the dashed lines: it remembers the relative size of the memories for each frame in the call stack.

SH-NIL

$$\frac{\texttt{size\_mem}\ m_1' \leq \texttt{size\_mem}\ m_1}{\texttt{size\_history}\ [m_1]\ [m_1']\ []}$$

SH-CONS

$$\frac{\begin{array}{c}\texttt{size\_mem}\ m_1 = \texttt{size\_mem}\ m_2 - \texttt{sz\_vars}\ f \\ \texttt{size\_mem}\ m_1' = \texttt{size\_mem}\ m_2' - \texttt{sz\_stk}\ f \\ \texttt{size\_mem}\ m_1' \leq \texttt{size\_mem}\ m_1 \\ \texttt{size\_history}\ (m_1 :: lm)\ (m_1' :: lm')\ cs\end{array}}{\texttt{size\_history}\ (m_2 :: m_1 :: lm)\ (m_2' :: m_1' :: lm')\ (f :: cs)}$$

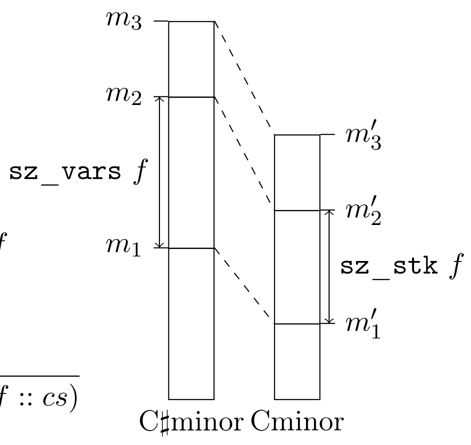


Figure 8.11: The `size_history` predicate

Rule SH-nil is the base case of this predicate: it relates memory states m_1 and m_1' provided that `size_mem` $m_1' \leq$ `size_mem` m_1 when no frame has been allocated yet. Rule SH-cons is the inductive constructor of `size_history`. In order to add memory states m_2 and m_2' and a frame f to the `size_history` predicate, the sizes of m_2 and m_2' must be consistent with the frame f with respect to the previously top memory state. In other words, if m_1 was the memory state previously at the top of the history and we wish to add a memory state m_2 on top of it, then the size of m_2 should be exactly the size of m_1 plus the size of the variables, as dictated by the frame f; and likewise for m_2'.

With `size_history` as an invariant of the memory states throughout the execution of C♯minor and Cminor programs, the memory injection, and in particular the size constraint, can now be proved through simple reasoning about the sizes of the memory states after allocation. At function entry, we push new memory states into this invariant, to remember the relative sizes of memory states. At function exit, we use the `size_history` hypothesis to prove the *decreasing size* property of injections.

8.3 Optimisations

Most of CompCert's optimisations are performed at the RTL level. These optimisations are based on the result of static analyses whose results indicate for example the abstract content of every register and memory cell. Depending on the abstract values at various locations, several optimisations can be performed. For example, if it is statically known that a given variable always holds the same value, constant propagation may be performed. This section focuses on two optimisations, constant propagation and common subexpression elimination, that share the same abstract domain for values and the same dataflow analysis.

First, we explain the principles of the value analysis of CompCert. Then, we introduce the notion of pointer tracking. This notion is absent in CompCert version 2.4. Later versions include such a notion, however no formal semantics can be given in CompCert to this pointer tracking. We show that our model allows us to define its semantics. We show that some transfer functions from CompCert are unsound in our model and explain how to fix them. Finally, we use the resulting abstract model to reprove the constant propagation and common subexpression elimination optimisations.

8.3.1 Value analysis of CompCert

Robert and Leroy [RL12] describe an early version of the value analysis of CompCert. It aims at propagating constant pointers, but also at tracking whether stack pointers are possibly passed as arguments of functions calls. This is essential to decide whether invariants about stack variables are preserved across function calls. Indeed, if a pointer to the current stack is passed to a function `f`, since `f` may modify arbitrarily the contents of the whole stack, the only sound approach consists in invalidating the stack invariant.

When undefined operations are performed, it might be unclear whether a pointer is passed to a function. The following code snippet illustrates such a situation.

```c
int foo() {
  int x = 7;
  f(uintptr_t(&x)>>1);
  return x;
}
void f(uintptr_t ptr) {
  *((int*)(ptr << 1)) = 0;
}
```

Function `foo` initialises a (stack-allocated) local variable `x` with the value 7, then calls a function `f` and finally returns `x`. The argument passed to `f` is the result of right-shifting the address of `x`. Since this operation is undefined in CompCert's semantics, it is sound for the dataflow analysis to assume that the argument of `f` can never be a pointer. Since the function call has no parameter that are pointers, it is mathematically sound for the compiler to trigger constant propagation and for the compiled program to return 7. CompCert 2.4 has this aggressive behaviour.

However, the function `f` may forge a pointer to `x` because the integer that it receives as an argument is derived from a pointer to `x`. In fact, with a concrete memory model such as ours, since pointers to `int` must be aligned, we can always reconstruct the original pointer by left-shifting the integer, as shown in the code snippet above. This is a delicate situation where the original program returns 0, because of the update done by function `f`, and the *optimised* program returns 7 because of the constant propagation.

Even if this behaviour looks wrong, it is important to realise that it is allowed in CompCert because the original code does not have well-defined semantics, hence the semantics of the program is stuck before the function call. The rest of the program may therefore be optimised in any way.

Still, in order to disable such unintuitive optimisations, later versions of CompCert take a more conservative approach and track whether a value may originate from a pointer, as in the right-shifting example we have seen above. In such cases, the dataflow analysis makes the conservative assumption that those pointers could be reconstructed and dereferenced. As this pointer tracking cannot be formally expressed with the existing CompCert semantics, the weakness of this approach is that one needs to inspect the code of the transfer functions to ensure that pointer variables never leak through integer variables.

Because our semantics defines arbitrary pointer arithmetic, our semantic preservation theorem is forced to preserve these semantics. Hence, we now have a formal guarantee that CompCertS cannot have the misbehaviour of CompCert 2.4 illustrated above, since it would violate the theorem.

8.3.2 Formal tracking of pointer provenance

A crucial step to adapt the existing alias analysis of CompCert to our semantics consists in formally defining what it means for a symbolic value to depend on a pointer. The dataflow analysis of CompCert [RL12] is formalised as an abstract interpretation [CC77]. We first define an abstract domain *aptr* for pointers, largely inspired from CompCert.

$$aptr ::= \bot \mid Cst \mid Gl\ id\ ofs \mid Glo\ id \mid Glob \mid Stk\ ofs \mid Stack \mid \neg Stack \mid \top$$

We give semantics to these abstract pointers by defining a concretisation function $\gamma_b : aptr \to \mathcal{P}(sval)$ that maps each abstract element to a set of symbolic values, where b is the block corresponding to the current stack frame. The $\bot$ element represents the empty set of symbolic values. *Cst* abstracts symbolic values that are constants, *i.e.* their evaluation is independent of the memory layout. *Stk ofs* is the set of symbolic values that evaluate to $\mathtt{ptr}(b, ofs)$ (*i.e.* offset *ofs* in the current stack frame). *Stack* represents all symbolic values that depend in any way on the block of the current stack frame. Note that it is not necessarily a pointer in the block b, but it might be a symbolic value like $\mathtt{ptr}(b, o) \gg 1$ as we have seen earlier. This notion of dependence is formalised by the *dep* predicate defined in Figure 8.12: $dep(sv, B)$ means that the symbolic value *sv* depends at most on the concrete address of blocks $b \in B$, *i.e.* for all concrete memories allocating blocks in B at the same addresses, the evaluation of the symbolic value *sv* is unchanged. Said otherwise, the values of the blocks not belonging to B have no impact on the evaluation of the symbolic value. Then, *Gl id ofs* captures the set of symbolic values that evaluate like $\mathtt{ptr}(b_{id}, ofs)$, where b_{id} is the block associated with the global identifier *id*. *Glo id* is the set of symbolic values that depend on the block b_{id}. *Glob* is the set of symbolic values that depend only on blocks that are associated with global identifiers. Finally, $\top$ is the set of all symbolic values. All these concretisation relations are summed up in Figure 8.12, where predicate `is_glob_block` *id b* holds if *b* is the block associated with global identifer *id*.

The difference with CompCert's *aptr* domain is that our domain represents not only pointers but also symbolic values that *depend* in any way on the concrete address of some set of blocks. In CompCert, the γ_b concretisation function associates only actual pointers $\mathtt{ptr}(b, o)$ to an abstract element *aptr*. Hence, CompCert can not model that a value *depends* on some block without itself being a pointer, like the symbolic value $\mathtt{ptr}(b, o) \gg 1$ for example. Another difference is that we include the abstract element `Cst` for symbolic values that do not depend on any block.

$$
\begin{array}{lcl}
\gamma_b(\bot) & = & \emptyset \\
\gamma_b(\mathit{Cst}) & = & \{sv \mid dep(sv, \emptyset)\} \\
\gamma_b(\mathit{Gl\ id\ ofs}) & = & \{sv \mid \forall cm\ im, [\![sv]\!]_{cm}^{im} = cm(b') + ofs \land \texttt{is\_glob\_block}\ id\ b'\} \\
\gamma_b(\mathit{Glo\ id}) & = & \{sv \mid dep(sv, \{b'\} \land \texttt{is\_glob\_block}\ id\ b'\} \\
\gamma_b(\mathit{Glob}) & = & \{sv \mid dep(sv, \{b' \mid \exists\ id,\ \texttt{is\_glob\_block}\ id\ b'\})\} \\
\gamma_b(\mathit{Stk}\ o) & = & \{sv \mid \forall cm\ im, [\![sv]\!]_{cm}^{im} = cm(b) + o\} \\
\gamma_b(\mathit{Stack}) & = & \{sv \mid dep(sv, \{b\})\} \\
\gamma_b(\neg \mathit{Stack}) & = & \{sv \mid dep(sv, block \setminus \{b\})\} \\
\gamma_b(\top) & = & sval
\end{array}
$$

where

$$
\begin{array}{lcl}
dep(sv, B) & = & \forall cm\ cm'\ im,\ cm \equiv_B cm' \Rightarrow [\![sv]\!]_{cm}^{im} = [\![sv]\!]_{cm'}^{im} \\
cm \equiv_B cm' & = & \forall b, b \in B \Rightarrow cm(b) = cm'(b)
\end{array}
$$

Figure 8.12: Concretisation of abstract pointers

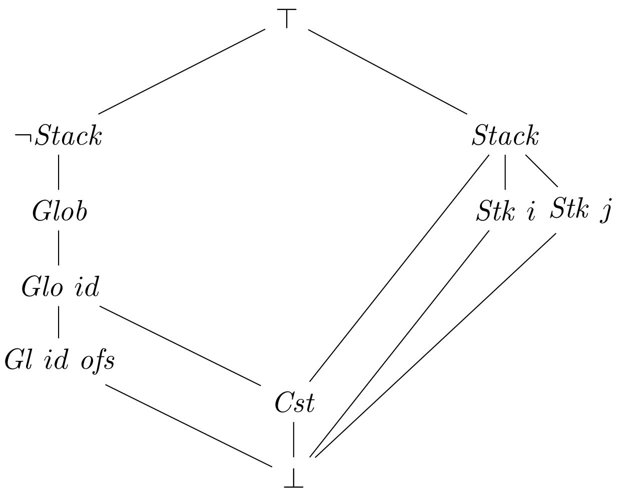


Figure 8.13: Ordering of abstract pointers

Note that the definition of the concretisation is robust and takes into account the semantics of symbolic values. For instance, the symbolic values 1, $1 + 1$ and $(b, 0) * 0 + 1$ are all in the concretisation of *Cst*, even if $(b, 0) * 0 + 1$ mentions the block b, because its evaluation is 1 in every concrete memory.

The domain of abstract pointers is partially ordered by $\sqsubseteq$, which is depicted in Figure 8.13. A least upper bound operator, written $\sqcup$, can be defined such that for every $(x, y) \in aptr^2$, $x \sqcup y$ is the least element such $x \sqsubseteq x \sqcup y$ and $y \sqsubseteq x \sqcup y$.

8.3.3 Improving the transfer functions

In order to establish the abstract semantics of a program, one needs to give semantics to every construction of the language on the abstract domain. In particular, we give the semantics of arithmetic operations on the domain of abstract values, defined as follows:

$$aval ::= I\ i \ \mid\ Ptr\ aptr \ \mid\ \ldots$$

An abstract value is either an integer $I\ i$ where i is a concrete 32-bit integer, or an abstract pointer *aptr*. The actual definition also includes constant floating-point numbers and 64-bit integers, which are omitted here for simplicity.

Let us now examine the transfer functions of a few operations. For example, consider the abstract operation $Stk\ i \gg I(1)$. In CompCert 2.4, it returns Cst[3], meaning that this is not a pointer but a constant value; this is what causes the unintuitive optimisation described in Section 8.3.1. In more recent versions of CompCert (2.5 and above) however, it returns *Stack* because the transfer function takes into account the *provenance* of pointers. However the *Stack* abstract element in CompCert does not have the same concretisation as ours. In CompCert, *Stack* abstracts every pointer to the current stack block. In our model, it abstracts any symbolic value that depends, in one way or another, on the current stack block. While this is a good thing that CompCert is more conservative than it could be for this case, this reasoning is not formal by any means, *i.e.* the quality of the transfer function and the absence of *bugs* inside them is not formally assessed. Our transfer function, on the other hand, has to be conservative because the underlying semantics of pointer operations is more defined. It returns *Stack* also, but it could not have returned *Cst*, because then we could not have proved that the transfer function is sound, namely in our case:

$$\forall\ cm\ cm'\ im,\ [\![\mathtt{ptr}(b,i) \gg \mathtt{int}(1)]\!]^{im}_{cm} = [\![\mathtt{ptr}(b,i) \gg \mathtt{int}(1)]\!]^{im}_{cm'}$$
$$\Leftrightarrow$$
$$\forall\ cm\ cm',\ (cm(b)+i) \gg 1 = (cm'(b)+i) \gg 1$$

This equation is clearly not true: consider for example $cm(b) = 16$, $cm'(b) = 32$ and $i = 0$. We have on one side $(cm(b)+i) \gg 1 = 16 \gg 1 = 8$ and on the other side $(cm'(b)+i) \gg 1 = 32 \gg 1 = 16$.

During the proof of the soundness of all transfer functions for all operators, we found several such transfer functions which are unsound for our semantics. Those *bugs* have been reported upstream and their impact is currently evaluated. In certain cases, the dependence really looks benign. Anyhow, the fix consists in weakening the transfer functions and therefore does not have any impact on the existing proof.

The constant propagation and common subexpression elimination passes exploit the dataflow analysis to transform programs. We adapt the correctness proofs of these optimisations to account for our domain of symbolic values. This does not require to modify the optimisation passes, but merely to use our memory improvement relation (defined in Section 7.1) instead of the memory extension relation, as described in Section 2.5.3.2.

We use the same methodology to adapt the neededness domain used by the dead-code elimination pass of CompCert. The neededness domain captures the set of bits of integer values that are live at each program point and relies on the memory extension relation. We lift this to a memory improvement relation and the whole transformation is straightforwardly reproved, with few adjustments.

8.4 Construction of `Mach` stack frames

CompCert is proved correct with respect to an unbounded memory model: memory allocation never fails. Therefore, the semantic preservation guarantees do not account for memory consumption. As a consequence, a compiled program may require more or less

[3]As we stated earlier, the abstract element *Cst* does not exist in CompCert 2.4, however the abstract element returned by the transfer function for the right-shift is $Ifptr\ \bot$ (see CompCert's development for further details), which has the same concretisation as our element *Cst*.

memory than the source program. If the compiled program requires more memory, it may exhaust the whole memory of the machine and crash. This is an unfortunate possibility that CompCert's theorem does not account for, because CompCert's memory space is unbounded. For example, the following program transformation can be proved sound with respect to CompCert's memory model:

```
int main(){
  return 0;
}
```

$\xrightarrow{\text{dubious transformation}}$

```
int main(){
  int dumb_array[0x80000000];
  return 0;
}
```

On the left-hand side, we have a C program that merely returns the integer `0`. On the right-hand side, the C program also returns 0 but has allocated a large array of integer elements, for a total size of `0x80000000 * sizeof(int)` bytes, *i.e.* $2^{30} \times 4 = 2^{32}$ bytes, therefore exhausting memory. This transformation can be proved sound in CompCert because the allocation of the large array succeeds and does not impact the rest of the program. However, when run, the programs will behave differently: one will safely return 0 while the other will crash at runtime because of an *out-of-memory* situation.

In CompCertS, we model a finite 32-bit addressed memory and the allocation operation may fail if it is unable to construct a concrete memory (as discussed in Section 5.4). Since allocations may fail, it is important for the semantic preservation property that whenever a source program succeeds in allocating a chunk of memory, the corresponding compiled program succeeds as well in allocating the same chunk of memory. In other words, compiled programs must use *less* memory than source programs. Said otherwise again, compilation must decrease the memory usage of programs.

As a result, we are able to guarantee that, for any source program that has defined semantics in C (in particular it does not exhaust the memory at the C level), then the corresponding compiled program uses less memory and therefore does not crash with an *out-of-memory* situation. This is an improvement over CompCert, because this rules out the example program transformation we discussed above.

The *decreasing memory space* property is already satisfied by most compiler passes. Only the `Stacking` pass does not have this behaviour. The purpose of this section is therefore to focus on this pass, namely `Stacking`, and explain how we cope with it.

First, we introduce the `Stacking` compiler pass, whose purpose is to allocate callee-save variables and metadata such as the return address in the stack frame of functions, in addition to the stack frame constructed in previous passes. Obviously, this is in contradiction with the principle of decreasing memory usage that we just described. Then, we show how we cope with this by provisioning memory in the semantics of all intermediate languages, so that the compilation indeed decreases the memory usage of programs.

8.4.1 The `Stacking` transformation

The `Stacking` pass is the penultimate pass of the CompCert compiler (see Figure 2.2). It transforms `Linear` programs into `Mach` programs. `Linear` is similar to `RTL` with the differences that only a finite number of registers is available and the code of a given function is linearised into a list of instructions instead of a control-flow graph. The `Mach` language is very close to the `Linear` language, but with a more concrete view of the stack frames of functions.

Figure 8.14 shows the structure of the `Mach` stack frames for the x86 architecture. In particular, it contains the `Linear` stack frame, shifted by a given offset from the start of the `Mach` stack frame. This will be modelled by an injection function in the proof of this transformation, and pointers to the stack frame will need to be shifted by this offset (*i.e.*

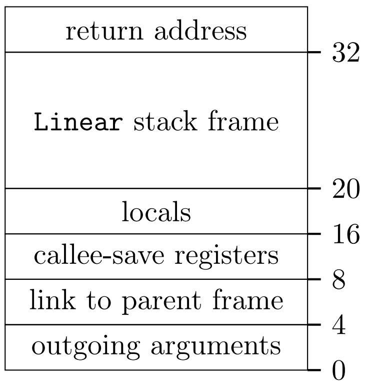


Figure 8.14: The `Mach` stack frame

20 in the case of Figure 8.14). The stack frame also contains the return address of the current function, slots for local variables, callee-save registers, a link to the stack frame of the parent (caller) function and stack space for outgoing arguments.

The return address is a pointer to the code of the caller function, indicating where the control should return once the current function has finished executing. This information did not appear in the stack frame of the previous languages because the return address was recorded in the semantic state of programs. However, the compiled program will run in an low-level environment that does not record the return address automatically. Therefore, it is the responsibility of the program itself to save its return address explicitly.

The local variables shown in Figure 8.14 are the RTL pseudo-registers that could not be allocated to hardware registers during the register allocation pass (from RTL to LTL, see Figure 2.2), and have been *spilled* into memory.

The callee-save registers are registers whose value should be preserved across function calls. The Application Binary Interface (ABI) of different architectures describes which calling conventions should be used and what registers should be *callee-save*. Such registers may still be used by a called function, provided that the function ensures that the values stored in those registers are restored at the end of its execution. It is the responsibility of the *callee* (the called function) to save their values. Those registers are saved within the stack frame of the function, as shown in Figure 8.14.

The link to the parent frame is a pointer to the caller's stack frame at offset 0. This pointer is used to access the parameters of the function call, stored in the caller's frame. This is required in CompCert because the stack frames are each allocated in their own block, separated from the other blocks. In particular, there is no relation between the concrete addresses of the block of the stack frame of a function and the block of its caller's stack frame. By contrast, in traditional compilation of functions into assembly, the calling conventions dictate where the arguments are to be fetched from, *e.g.* at a known offset from the stack register `esp`.

The outgoing arguments to a function call are also inserted into the stack frame of functions. This space is reserved so that every function call has enough space to push its arguments on the stack before transferring control to the callee.

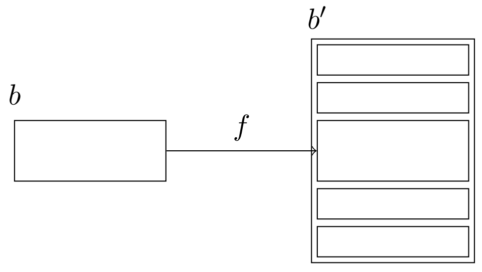


Figure 8.15: `Stacking`'s proof in CompCert: memory injection

All this metadata, that was not present in the stack frames or in the memory state in previous higher-level languages, is what we call the *hidden cost* of the C semantics. This memory is required for the execution of programs in lower-level languages and is not precisely accounted for in the semantics of C.

Hence, the `Mach` memory state is necessarily strictly larger than the corresponding `Linear` memory state since it includes strictly more data in the stack frames. This contradicts our hypothesis of decreasing memory usage, making it possible for a program to have well-defined semantics before the `Stacking` pass and undefined semantics after this pass – thus invalidating the current proof of correctness of the compiler.

8.4.2 Adapting the correctness proof with memory provisions

The existing proof of this pass in CompCert uses a memory injection of the form shown in Figure 8.15, where one block b (the `Linear` stack frame) is injected into a larger block b' (the `Mach` stack frame) at a given offset. The locations that are not injections of locations of b are used to store all the metadata described in the previous section.

When we introduced our generalisation of memory injections in Chapter 7, we explained that the size of the memory should be decreasing with injections. The injection used by CompCert for the proof of the `Stacking` pass and depicted in Figure 8.15 does not satisfy this condition, because the stack size grows and therefore we cannot formally ensure for example that the allocation of the stack frame will succeed.

In order for this pass to fit in the *decreasing memory usage* framework, the solution we choose is to make memory provisions in earlier languages, so that the memory used for the *hidden cost* of the semantics of C are already accounted for in the semantics of C. In other words, we modify the semantics of all languages from C to `Linear` so that at function entry, one does not only allocate the local variables or the stack frame in the memory, but also a certain amount of additional pieces of memory that serve as place holders for the metadata that is inserted in `Mach` stack frames.

Using this solution, the injection used in the proof of correctness of the `Stacking` pass resembles more the injection used in the `Cminorgen` pass (see Section 8.2). It is depicted in Figure 8.16. In this situation, every location in b' has a corresponding location either in b or in one of the additional blocks used as memory provisions, hence the size of the memory stays the same across the injection. Therefore we can prove that the two memory states are in injection, using techniques similar to those we used for the proof of the `Cminorgen` pass.

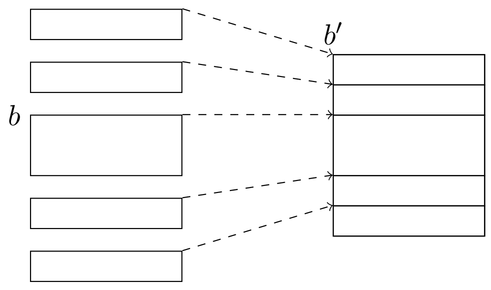


Figure 8.16: `Stacking`'s proof in our symbolic model: memory injection

8.4.3 Memory provisions in the intermediate languages

The solution of provisioning memory allows to prove the correctness of the `Stacking` pass. However, we need to explain how we provision memory for each function. Our solution uses an oracle that gives, for every function, the amount of memory that it is necessary to provision so that the stack frame can be allocated at the `Mach` level. First, we will show how, given such an oracle, the provisioned memory is preserved across compilation up to the `Stacking` pass. Second, we will show how to construct such an oracle from the result of the compilation.

8.4.3.1 Preservation of the memory provision.

To propagate the provision of memory, we instrument the semantics of all the languages from C to `Linear` in a similar fashion. We parameterise these semantics with a mapping `needed_stackspace : function -> nat` that associates with each function the number of additional bytes it requires for the `Mach` stack frame to be allocated, and therefore for the `Stacking` pass to decrease the memory usage. At function entry into function `f`, we allocate both the blocks needed for the local variables of `f` (or its stack frame, depending on the language), and extra blocks of a total size of `needed_stackspace f` bytes. At function exit, we free the blocks corresponding to the local variables and those additional blocks. All the compiler passes from C to `Linear` simply preserve these extra blocks and leave them untouched. Only the `Stacking` pass consumes these blocks to justify the use of extra stack space for the construction of the stack space.

Note that this makes the requirement for a program to have a defined C semantics slightly stronger. In CompCert, there is no requirement on the memory consumption of programs that implies undefined semantics. In CompCertS, a given C program has defined semantics only if all the allocations of the local variables succeed (which is itself governed by the allocation algorithm described in Section 5.4). With this provisioning, a C program has defined semantics only if the allocations of the local variables and the provisioning blocks succeed. This is a stricter requirement than before. This makes our C semantics stricter than that of CompCert in some sense, because we don't give semantics to C programs that exhaust the memory. However, as we have advocated in Section 6.2, our semantics is more defined than that of CompCert for all other aspects, *i.e.* we give semantics to more operations.

We show in Figure 8.17 how the stack space of functions is structured and how it evolves throughout the compilation. Before the `SimplLocals` pass, each variable owns a memory block (in white) and there are several provisioning memory blocks (in grey). After the `SimplLocals` pass, some variables go out of memory and are transformed into provisioning blocks. The `Cminorgen` pass leaves the provisioning blocks untouched and groups the

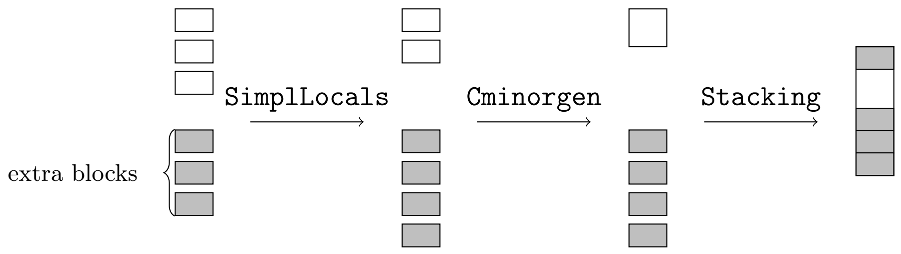


Figure 8.17: Structure of stack frames and memory provisions during compilation

variable blocks into one stack frame block. Finally, the `Stacking` pass consumes all the provisioning blocks to allocate a bigger stack frame, as described earlier.

8.4.3.2 Computing the oracle.

The last step in making this provisioning technique work is to construct the oracle used in the various semantics that gives the number of extra bytes that are necessary to ensure that the compilation happens in decreasing memory usage. The computation of this oracle will actually be performed by the compiler. The compiler can be seen as a function $comp : \texttt{prog}_C \rightharpoonup \texttt{prog}_{ASM}$. The assembly program that is output by the compiler contains not only the assembly code of functions but also metadata about the functions that we introduce during the compilation in order to remember some useful information. In particular, if the output of the compiler is the assembly program tp, for each function f, we remember inside tp: *a)* the number of additional bytes required for the allocation of the `Mach` stack frame, that we write $ns(tp, f)$ (for needed stack-space); and *b)* the number of bytes made available by the `SimplLocals` pass, that we write $sl(tp, f)$. Those pieces of information are available during the compilation and it is straightforward to remember them.

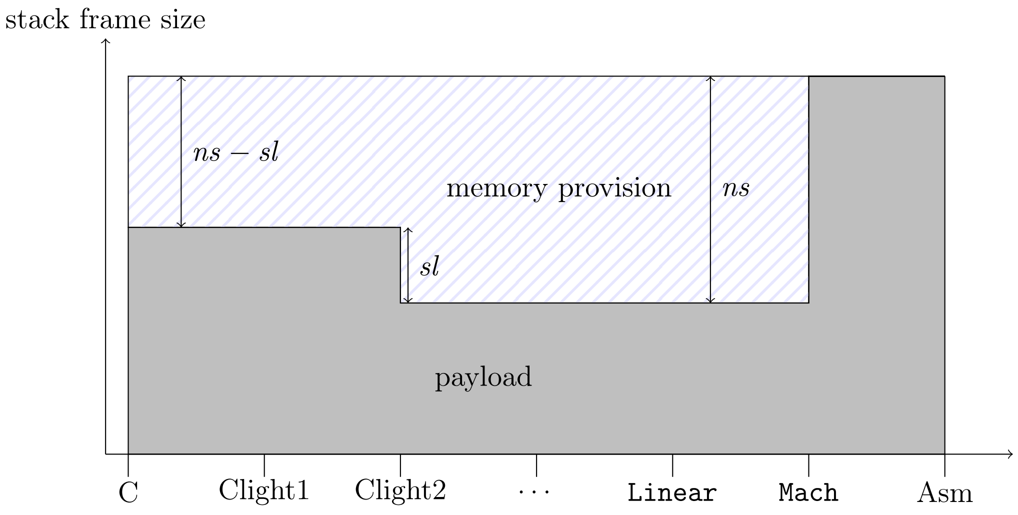


Figure 8.18: Evolution of the size of function stack frames during compilation

The evolution of the memory consumption of the instrumented semantics is given by Figure 8.18. It is another view of the information displayed by Figure 8.17 that allows to quantify the amount of memory that needs to be provisioned. The lower grey part of the diagram is the memory used by each function's stack frame (or local variables, depending on the language) in CompCert's semantics. At the Clight2 level, which represents the Clight language after the `SimplLocals` pass, the memory usage decreases because some blocks are pulled out of memory. The memory usage is preserved by all passes until the `Stacking` pass, which makes the memory usage increase again. Our solution of memory provisioning is pictured by the higher, hatched area of the graph: we allocate the hatched part of the memory (provision) on top of the grey part (the *payload* memory, *i.e.* the memory that corresponds to actual data in the semantics of higher-level languages). We can see that the overall memory usage stays constant all along the compilation.

Figure 8.18 shows that two oracles are actually needed: one for the languages from C to Clight1, that we call o_1, and the other for the languages from Clight2 to `Linear`, that we call o_2. Both associate with each function f the number of bytes that need to be allocated at function entry. Following Figure 8.18, we define o_1 and o_2 as follows:

$$o_1(tp) = \lambda f.\ ns(tp, f) - sl(tp, f)$$
$$o_2(tp) = \lambda f.\ ns(tp, f)$$

We can now use these oracles to parameterise the semantics of the languages from C to `Linear`, and the proof of correctness of the compiler. This construction may seem circular at first sight (using the compiler to produce oracles that are used in its own proof of correctness), however it is not. The compiler does not need any oracle as input, only the semantics of the languages and the proofs of correctness of the individual compiler passes do.

8.4.3.3 CompCertS' theorem.

Theorem 8.4.1 is the final theorem of correctness of CompCert (see Section 2.5). It relates behaviours of C programs and those of the compiled assembly programs. We recall its statement below.

Theorem 8.4.1 (`transf_c_program_preservation`). *For any C program p, if the compilation succeeds in generating an assembly program tp, then every behaviour of tp is an improvement over a behaviour of p. Formally,*

$$\forall\ p\ tp,\ comp(p) = \lfloor tp \rfloor \Rightarrow \forall\ B,\ tp \Downarrow_{ASM} B,\ \exists\ B',\ p \Downarrow_{C} B' \wedge B' \subseteq B.$$

In Theorem 8.4.1, the $\Downarrow_L$ relation is such that $p \Downarrow_L B$ means that the execution of p according to the semantics of language L exhibits behaviour B.

Since the semantics of the C language is parameterised by an oracle, compared to Theorem 8.4.1, the semantic preservation theorem of CompCertS (Theorem 8.4.2) mentions this oracle and the correctness is stated with respect to this oracle.

Theorem 8.4.2 (`transf_c_program_preservation`). *For any C program p, if the compilation succeeds in generating an assembly program tp, then every behaviour of tp is an improvement over a behaviour of p according to the C semantics parameterised by the oracle $o_1(tp)$. Formally,*

$$\forall\ p\ tp,\ comp(p) = \lfloor tp \rfloor \Rightarrow \forall\ B,\ tp \Downarrow_{ASM} B,\ \exists\ B',\ p \Downarrow_{C}^{o_1(tp)} B' \wedge B' \subseteq B.$$

Here, $p \Downarrow_{C}^{o_1(tp)} beh$ means that the execution of program p according to the semantics of C with oracle $o_1(tp)$ exhibits B. An interesting corollary of this theorem is the following: for any C program p that compiles into a program tp, if p has a defined semantics according to the oracle $o_1(tp)$, then executing tp does not produce *out-of-memory* errors. That is because the C program has defined semantics, hence it uses a given amount of memory that is less than the maximal amount of memory and we know that the assembly program uses less memory than the C program.

8.5 Conclusion and Discussion

In this chapter, we have covered the main changes we brought to the compiler passes of CompCert to adapt to our symbolic memory model, in particular to the proof of correctness of those passes. The final result is a complete compiler, that we call CompCertS, that compiles C programs into assembly programs, with the guarantee that C programs are compiled into safe assembly programs whose behaviours are improvements of the original programs, like CompCert. However, unlike CompCert, our compiler gives more guarantees about the generated assembly programs. First, the notion of behaviour improvement gives the compiler the freedom to replace *going-wrong*, or *stuck*, behaviours by any behaviour. We reduce this freedom by giving a defined semantics to more programs, hence fewer programs exhibit going-wrong behaviours that can be optimised in an unintuitive manner by the compiler. This is an improvement over CompCert because more programs can be compiled faithfully to the programmer's intentions. Second, because our memory model is a finite memory model, we account for the memory consumption of programs, and our final correctness theorem ensures that the compiled program does not go out of memory, provided that the source C program does not.

Similarly, Carbonneaux *et al.* [Car+14] propose a modified version of CompCert that they call Quantitative CompCert and which makes several contributions related to the question of memory consumption of C programs. First, they provide a Hoare-like logic for C programs that they use to prove bounds on the stack-space usage of C functions. Then, they prove that those bounds are preserved throughout the compilation. Finally, they modify the assembly language into an ASM_{sz} language, where a block of size sz bytes is allocated at the beginning of the program and serves as the stack. Their theorem states that: provided that the source program has defined semantics, and the bounds they infer on the source program are lower than sz, then the compiled assembly program does not stack overflow. Our work provides a similar conclusion, namely that the assembly program does not run out of memory. While their work obtains space bounds through a *quantitative Hoare logic*, our work uses the allocation algorithm described in Section 5.4 to decide whether an allocation is possible or not.

In the following, we list a few limitations that the development of this compiler still has, and explain how we hope to cope with those.

`SimplLocals` and indeterminate values. The `SimplLocals` pass is crucial because it generates temporary variables in the Clight language, that are subsequently transformed into RTL pseudo-registers. Those pseudo-registers are the main target of most of the optimisations performed at the RTL level, *e.g.* constant propagation, common subexpression elimination, dead-code elimination. Recall that the proof of correctness of the `SimplLocals` pass relies on a partial injection which, for each block b, either keeps it (injects it into itself) or forgets it (does not inject it). Blocks that are not injected are replaced by temporary variables, *i.e.* semantic objects that do not reside in memory.

In the existing proof of CompCert, the values inside the block being forgotten and the corresponding temporary must be in injection. Local variables are initialised with `indet`(b,i) values, whereas temporary variables are initialised with `undef`. Our injection does not hold in this case because we do not have `sval_inject` f (`indet`(b,i)) `undef`; indeterminate values only inject into other indeterminate values.

A solution we would therefore envision is to initialise temporary not with `undef` but with `indet`(b',i') values. Now the problem becomes that of finding such a block b'. We could allocate a block b' for every temporary variable we create and state that $f(b) = \lfloor(b',0)\rfloor$

to get the desired injection between indeterminate values, but that would imply that the contents of both memories in blocks b and b' should be in injection, which defeats the very purpose of this transformation: pulling variables out of memory.

Another solution would be to invent a new domain `temp_id` of temporary identifiers, and change the type of injection functions from $block \rightharpoonup (block \times Z)$ into the more complex $block \rightharpoonup ((block \times Z) + reg)$ (where the type $(A + B)$ is the sum type of A and B, *i.e.* an element of this type is either an element of A or an element of B). In other words, a block would be injected either in another block at a given offset (usual case) or into a register. This is a more demanding solution because memory injections are already a complex notion with many properties that would need to be reproved. However, this approach looks promising and deserves further investigation.

The workaround used in the current proof is simply not to introduce `indet`(b, i) indeterminate values in the allocated blocks, *i.e.* blocks are initialised with `undef` and the injection holds trivially because `undef` injects into any value. This however prevents reasoning about the uninitialised contents of memory, mimicking the semantics of CompCert in that regard.

Nevertheless, the proof we have done for all other passes works with indeterminate values with very few changes. This claim is backed by an alternate version of the development[4] where the `SimplLocals` pass does nothing, *i.e.* it does not remove any variable from the memory, and the blocks are initialised with `indet`(b, i). All the passes are then proved correct by success, however the optimisations are not as good because they operate mainly on temporaries.

Optimisations: inlining. CompCert also includes an inlining pass, which transforms certain functions calls into the code of the function. This is a useful transformation especially for short functions whose code may subsequently optimised by intra-procedural optimisations. The proof of this optimisation is based on a memory injection, however the transformation may increase the memory usage of programs. We expect that memory provisions similar to the technique we used for the `Stacking` pass will help us prove this transformation correct. However it seems to be more complex. Consider the following code snippet:

```
void f() {
  ...
  g()
  ...
}
```

If the function `g` is inlined, then the stack frame of `f` has to be large enough to contain that of `g`. Hence, the oracle for `f` must provision additional blocks for the stack frame of `g` to fit in, for the inlining pass. However, at the C level, `g` is not inlined yet and must allocate space for its local variables and provision blocks for its stack frame at the `Mach` level. This results in a sub-optimal oracle that provisions stack space twice for the same function. It is still to be investigated how to compute a better oracle with inlined functions.

To sum things up, we have adapted most passes of CompCert with the generalisations of various CompCert's concepts that we presented all along this thesis: symbolic values, concrete memories, normalisations, finite memory, memory relations and memory

[4]This alternate version is available online, see `http://www.irisa.fr/celtique/wilke/phd/`.

injections. We end up with a formally-verified compiler, CompCertS, for low-level C programs with respect to a finite memory model, which gives guarantees about the run-time memory consumption of programs.

Chapter 9

Conclusion

In this thesis, we have defined the first formally-verified compiler for C that accounts for bit-level manipulation of pointers and uninitialised data. All the existing formally-verified compilers and formal semantics for C give undefined semantics to such idioms. Only Kang *et al.* [Kan+15] give defined semantics to such low-level manipulations, with the aim of proving optimisations correct.

The semantics and memory model of C are complex objects. As an illustration, even starting from CompCert's development, we had to iterate several times before we found the formalisation we present in this thesis. In particular, the notions of valid concrete memories and of normalisations have had several less well-behaved specifications.

Making modifications inside the code of a large project like CompCert is somewhat frightening at first, but in the end most of the structure of the CompCert compiler can be reused and provide a strong basis to build upon. The main difficulty we encountered was to find the *right* generalisation of semantic properties used to prove the correctness theorems of each compiler pass. Although the generalisations we present here look simple (equivalence of symbolic values, injection of symbolic values), there has been a number of unsuccessful definitions for those notions before we got to the right ones, *i.e.* those that enable us to prove the correctness of the CompCertS compiler.

Section 9.1 summarises the results we presented throughout this thesis. Short-term improvements and extensions of the developement are presented in Section 9.2 and Section 9.3. Finally, Section 9.4 gives somes ideas of applications of CompCertS.

9.1 Summary

With our daily lives becoming more and more dependent on software systems, it becomes of paramount importance to gain confidence in the correctness and safety of those systems. Formal methods are becoming mature enough to be applied to large-scale verification endeavours, such as the static analysis of the primary flight control software of the Airbus A340 series with Astrée [Bla+03], the verification of an industrial-strength C compiler, CompCert [Ler09b], or even the functional correctness of an operating system kernel, CertiKOS [Gu+15]. Still, those tools are based on a *formal semantics* of the C language. The correctness of those approaches is stated with respect to this semantics.

In our work, we advocate that the existing C semantics do not capture the features that programmers use and push into real-world projects such as the Linux operating system kernel or implementations of the C standard library. On the contrary, low-level C idioms that are not valid according to the C standard are nevertheless used by developers, that seem to share a common mental model, distinct from the C standard [ISO99], of how the memory is managed in C.

This gap between the formal semantics and the *commonly-assumed* semantics is a source of bugs and unintuitive compilations. Indeed, a low-level C program may be compiled unfaithfully to what the programmer expected if the C program exhibits undefined behaviour. The aim of this thesis is to bridge the gap between the formal semantics with respect to which the correctness theorems are stated and the mental model of C programmers. In particular, pointers can be manipulated as integers and uninitialised data can be reasoned about with the additional property that indeterminate values are stable.

We propose a definition of *symbolic values*, which form the basis of our work and that we incorporate into CompCert. This domain is more expressive than CompCert's value domain. In particular it models operations on pointers that would otherwise be undefined. We also give a more concrete view of the memory than in CompCert, and establish the notion of *concrete memory* that specifies a concrete layout of the memory space. Based on those notions, we extend CompCert's memory model with symbolic values and we adapt the semantics of CompCert's intermediate languages to operate over these symbolic values.

Then, we show that the proof methods from CompCert need to be generalised to our symbolic setting. In particular, memory injections describe how the memory is reorganised by compiler passes. These memory injections need to be reworked to accomodate for our finite low-level memory model. Finally, we have shown how to adapt the correctness proofs of the individual compiler passes and stated a new end-to-end correctness theorem for our symbolic compiler, CompCertS. The distinguishing feature of CompCertS, compared to CompCert, is that low-level bitwise operations are given semantics and compiled programs use no more memory than the source programs, thereby strengthening the formal guarantee offered by the compiler.

9.2 Short-term improvements

We summarise in the following various limitations and places for improvements in the current version of CompCertS. Most of these have been reported in the conclusions of the individual chapters but are reported here as well for convenience.

9.2.1 External functions

In CompCert, the semantics of C programs performing calls to external functions, *i.e.* functions whose code is not available at compile-time (either because they come from different compilation units or library calls) is *axiomatised.* This means that no precise semantics is assigned to such function calls, however it is assumed that the external functions terminate in memory states that satisfy certain properties, as prescribed by the `extcall_properties` predicate.

During the development of the correctness proof of CompCertS, we ruled out external functions that allocate or free blocks, *i.e.* those that change the structure of the memory, because they would affect the behaviour of the normalisation in the resulting memory states. Unfortunately, we have built our development on intermediate lemmas that would not be easily generalisable to memory transforming external functions. For example, we state that the size of the memory state is invariant under external function calls. However, the size of the memory states may change due to an external call, *e.g.* if a dynamic allocation has been performed. Adapting the axiomatisation of external calls so that allocations are possible should be possible, and would give a better model of external calls.

9.2.2 Formalisation of the SMT encoding of normalisations

As discussed in Section 6.3, we provide an implementation of the normalisation function using an SMT solver. The translation of normalisation queries into SMT problems is entirely written in OCaml with no guarantee whatsoever that the encoding is correct. We merely have an assumption on the Coq side that the normalisation function is correct.

A more principled option would be to encode the normalisation into a formal model of SMT queries inside Coq, and then state the correctness of the encoding with respect to the correctness of the underlying SMT solver. In other words, instead of trusting our encoding and the SMT solver, we would restrict ourselves to just trusting the SMT solver, thus eliminating middle-end translation bugs.

To reduce further the trusted computing base (TCB) regarding the SMT solver, we could imagine validating the output of the SMT solver [BCP11; Arm+11], *i.e.* verify that the solution the SMT solver outputs is actually a valid solution.

9.2.3 Injection of Indeterminate Values

As discussed in Section 8.5, indeterminate values `indet(`b, i`)` can only be injected into other indeterminate values `indet(`b', i'`)`. This works fine for all compiler passes but `SimplLocals`, where some stack-allocated variables are transformed into register-allocated variables. The issue is that registers are not initialised with indeterminate values, but rather with the `undef` value.

Initialising registers with indeterminate values requires 1. to come up with unique identifiers for registers; 2. modifying the type of indeterminate values to be also indexed by register identifiers; and 3. modifying the type of injections to capture that an indeterminate value `indet(`b, i`)` injects into a register r_{id}.

This solution, however invasive, seems to be the one to follow: there is no fundamental reason why registers could not hold some kind of indeterminate values.

As noted in Section 8.5, the workaround we use for the moment is not to initialise the memory blocks with indeterminate values but with `undef` values, as was the case in the existing CompCert releases. However, the proofs for the rest of the compiler passes can be done with indeterminate values. This claim is backed by an alternate version of CompCertS, where `SimplLocals` has been turned into an identity pass, *i.e.* it does not transform any variable into temporaries, and for which all compiler transformations are proved correct.

9.3 Extensions

In the following, we explain a few ideas we have for extending our work. Those ideas range from tuning the fundamental notion of validity of concrete memories to handle special cases to really extending the compiler with more passes.

9.3.1 Validity of concrete memories with lifetimes

We have seen in Section 6.2 that the validity relation on concrete memories is not preserved by the primitive memory operations of our memory model. Said otherwise, given two memory states m and m' such that m' is obtained after performing combinations of `store`, `alloc` and `free` operations on m, it is not true that every concrete memory cm' valid for m' is also valid for m. The counterexample is with the `free` operation, after which *more* concrete memories are valid, because less blocks are constrained.

However, concrete memories are supposed to be concrete views of how the memory is possibly laid out. A key insight is that blocks are not re-allocated at different concrete addresses during the execution of a program. Said otherwise, if at some point, we have that blocks b and b' are valid and distinct, therefore pointers $\texttt{ptr}(b, 0)$ and $\texttt{ptr}(b', 0)$ compare unequal; then at any time later in the execution of the program, even if those blocks have been deallocated, those pointers must compare unequal.

Note that this is different from what the C standard [ISO99] states, in §6.2.4.2 :

> The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

As a consequence, comparing a pointer with a pointer to a freed object exhibits undefined behaviour. This is the current behaviour of our symbolic semantics, because the validity relation for concrete memories only considers currently allocated blocks and may allocate freed blocks at *any* concrete address, therefore modelling that the pointer value is *indeterminate*.

However, we wish to capture the fact that pointers that have been disjoint once, will always be disjoint. One can see it as follows: at the beginning of the program, every concrete memory is valid, *i.e.* any block identifier (allocated or not) can be mapped to *any* concrete address. All along the program execution, the set of valid concrete memories should become more and more restricted, pruning out concrete memories that allocate disjoint blocks to overlapping concrete addresses.

To achieve this, we envision to enrich the memory state with a function that remembers the lifetime of blocks, *i.e.* at what time they were allocated and at what time they were freed. In CompCert, the block identifiers are positive numbers that are assigned incrementally, hence the block identifier itself is the *allocation time* of the block. One must only remember the *deallocation time* (the block identifier about to be allocated, *i.e.* the `nextblock` field of the memory state).

Consider that $\texttt{lifetime}(m, b)$ returns the time interval between which block b was allocated, $\cap$ computes the intersection of time intervals and $\emptyset$ represents the empty interval. Consider also the predicate $\texttt{was\_valid}(m, b, i)$ which holds if and only if there has been a time where location (b, i) was valid. The no-overlap constraint of the validity property for concrete memories will then be expressed as in the following:

$$\begin{aligned} \forall\, b_1\; i_1\; b_2\; i_2,\quad & b_1 \neq b_2 \;\wedge\; \texttt{lifetime}(m, b_1) \cap \texttt{lifetime}(m, b_2) \neq \emptyset \;\wedge \\ & \texttt{was\_valid}(m, b_1, i_1) \;\wedge\; \texttt{was\_valid}(m, b_2, i_2) \Rightarrow \\ & cm(b_1) + i_1 \neq cm(b_2) + i_2 \end{aligned}$$

Note that this property does not prevent one block from being allocated at the same concrete address of another block, as long as those blocks do not share any live range.

This more relaxed notion of valid concrete memories would allow to perform the proof of semantic refinement presented in Section 6.2 over the same semantics of Clight as in the rest of the compiler, *i.e.* without introducing artificial simplifications in the semantics. The rest of the compiler correctness proofs should not be affected much: we expect that most fundamental properties of the normalisation will stay true in this relaxed model.

9.3.2 More Optimisations

As we noted in Section 8.5, we have left out two optimisations for the moment, that were in CompCert: dead-code elimination and inlining.

The former necessitates a specialised abstract domain that records the *liveness* of variables, *i.e.* at each program point, which variables will be used later before being overwritten. If it can be determined that the value of a variable will not be used before being written again, one might as well not write the value into the variable in the first place. We have not yet reimplemented the abstract domain for our symbolic values, but we foresee no obstacle in doing so.

The latter optimisation, inlining, necessitates more work. Inlining consists in replacing certain selected function calls with the body of the functions being called. This is particularly efficient for small functions and avoids to waste some time switching contexts. It is also very valuable from the point of view of optimisations because it makes the analysis of the resulting code easier. Indeed, in intraprocedural analyses, functions are analysed independently with few knowledge about their environment, *e.g.* the possible values of the arguments. When the functions are inlined, it may be determined by subsequent analyses that some condition always evaluates to `true` for example, and the code may therefore be optimised.

The problem with inlining is that it may make functions increase the amount of memory they need. Consider the following code snippet.

```
void g(){
  int a;
  // manipulate a
}
int f(){
  if ( condition ) {
    g();
  }
  return 0;
}
```

Function `f` tests whether `condition` is `true`. If so, function `g` is called. Afterwards, the function exits with value `0`. If `g` is inlined, the code of `f` may look like the following:

```
int f(){
  int a;
  if ( condition ) {
    // manipulate a
  }
  return 0;
}
```

The problem with this program transformation is that, while the original function `f` did not need any stack space for its local variables, the transformed function `f` now needs to allocate the local variables of `g`, no matter whether the condition `condition` is ever satisfied. This is a problem for our finite memory model, whereby the compilation must decrease the memory usage. We could use the same memory provisioning technique that we presented for the correctness of the `Stacking` pass, *i.e.* pre-allocate additional chunks of memory at the C level so that the memory usage effectively decreases. However, as we noted in Section 8.5, it is unclear how to pre-allocate the memory space of `g` only when necessary, *i.e.* not to pre-allocate the additional space both in `f` and `g` at the C level.

9.3.3 A More Concrete Assembly Language

There is still a gap between CompCert's assembly language and the x86 assembly code that is passed to the actual assembler, and then run. CompCert's assembly language still contains high-level pseudo-instructions `Pallocframe` and `Pfreeframe` that are responsible for allocating and deallocating functions' stack frames.

For instance, the semantics of the `Pallocframe(sz,ofs_ra,ofs_link)` (where `sz` is the size of the stack frame to allocate, `ofs_ra` is the offset in the stack frame where the return address should be written and `ofs_link` is the offset in the stack frame where the link to the caller's stack frame should be stored) is responsible for allocating the memory region for the stack frame, and storing the return address and the pointer to the caller's stack frame at the appropriate locations. Note that the actual values to store at those offsets are not given as parameters to the `Pallocframe` pseudo-instruction. The address of the caller's stack frame is actually stored in the register `ESP`, and the return address is stored in a pseudo-register named `RA`.

On the other hand, this instruction is pretty-printed into the following sequence of assembly instructions:

```
sub esp, sz
lea edx, [esp + sz + 4]
mov [esp + ofs_link], edx
```

The first instruction corresponds to the allocation of the stack frame. The second instruction stores in register `edx` the address of the caller's stack frame. The third instruction stores this address at the offset `ofs_link` from the current stack pointer. Here, the return address is completely ignored, because it has already been set by the preceding `Pcall` instruction.

This dissymmetry makes it difficult to convince oneself that the printing phase is correct. An interesting piece of further work would be to make the assembly language more concrete. In order to match closely the `sub esp, sz` and the allocation of a stack frame, one would need to pre-allocate a single block of fixed size for the stack. This is delicate in CompCert because the memory is unbounded and therefore we have no guarantee that all the stack frames will fit in this stack block. However, in CompCertS, we know that the total amount of memory is less than some threshold (see Section 5.4 for details). Hence, our memory model would enable us to create a more concrete assembly language, where this large block represents the stack.

Also, the return address was not stored in the callee's stack frame at the time of the `Pcall` instruction because the stack frame was not allocated yet. With a single pre-allocated stack block, this would no longer be an issue: the return address would be stored at the time of the `Pcall` instruction, as it should be.

9.4 Perspectives

In this thesis, we have presented a formally-verified compiler for low-level C code: CompCertS. Being able to model low-level operations, especially on pointers, makes it possible to perform several security-enhancing program transformations, that exploit the binary representation of pointers. In the following, we give a few ideas of applications of our symbolic semantics.

9.4.1 Portable Software Fault Isolation

Software Fault Isolation (SFI) was first introduced by Wahbe *et al.* [Wah+93] as a mechanism to execute untrusted code in sandboxed environments. The idea is to determine a memory region in which memory accesses are permitted, and to instrument the untrusted program so that all memory accesses are performed inside the pre-determined safe memory region. This is usually performed at the assembly level, introducing a bitwise mask of pointers before every memory access.

In 2014, Appel *et al.* [KSA14] proposed a method for *Portable Software Fault Isolation*. Their method is portable in the sense that it does not transform assembly programs but rather C programs. This work is based on the CompCert compiler and is formalised in Coq. They have proved in Coq that the instrumented programs are secure, *i.e.* all the memory accesses are effectively performed inside the safe memory region.

However, because the masking function (the function that transforms a pointer into a pointer inside the safe memory region) has to be expressed with bitwise operations, it has no well-defined semantics and it cannot be reasoned about precisely. Instead, the masking function is merely axiomatised as an external function in their development. An inconvenient side-effect of this is that calls to this function may not be optimised by subsequent passes of the compiler (since the code is unknown) and the instrumentation may therefore incur a high overhead.

In actual implementations of SFI (Rocksalt [Mor+12], NaCl [Yee+10]), the safe memory region is a chunk of memory of size 2^n that is 2^n-byte aligned, *i.e.* the $32 - n$ higher bits constitute what is called a *tag* that entirely identifies a block.

Using our memory model, we could give defined semantics to a program that retrieves this *tag* and computes the masking function using bitwise operations.

9.4.2 Obfuscations

Program obfuscations [CTL97] are semantic-preserving program transformations which increase the complexity of programs, *i.e.* programs become harder to understand and reverse engineer. The goal of these obfuscations is to reach some kind of security by obscurity, to preserve some secret inside the code of a program or to make reverse engineering harder, *e.g.* for intellectual property issues.

There has been recent work by Blazy and Trieu [BT16] that formalises a control-flow graph flattening obfuscation inside the CompCert compiler. This is an advanced obfuscation that could be improved by combining it with simple data obfuscations such as *variable splitting* [CTL97]. The idea is to split occurences of a given variable `x` into two variables `x1` and `x2`, such that `x` can be expressed as a combination of `x1` and `x2`, *i.e.* no information is lost. A standard way of splitting variables relies on Euclidean division: `x1 = x / 10` and `x2 = x % 10`. Then, `x` can be recomposed as: `x == x1 * 10 + x2`. The problem of formalising this obfuscation in the CompCert compiler is that it is not possible to decompose and recombine `x` when it is a pointer. Indeed, the multiplication is undefined on pointers in C. Blazy and Trieu would therefore benefit from our low-level memory model for their work on formally-verified obfuscations, because pointers are a primary target for obfuscations, especially function pointers.

9.4.3 A Lower-Level Static Analyser

We have discussed in the introduction a formally verified static analyser, Verasco [Jou+15], whose distinguishing feature is to be embedded within CompCert and to be proved correct in Coq. The aim of Verasco is to prove the absence of run-time errors at the C level. To do so, it formalises multiple abstract domains for program states and numerical domains, relational and non-relational.

However, their approach is inherently limited by the formal semantics they use. In particular, they cannot prove anything about programs that perform low-level manipulation of pointers or of uninitialised data. Though it would require a large amount of work (Verasco is a large Coq development – around 34 thousand lines of code), we believe that a low-level static analyser would be profitable to treat low-level code, and it would discharge the hypothesis of the final theorem of our compiler CompCertS.

Bibliography

[ANS89] Americal National Standards Institute (ANSI). *Programming Language C.* Technical Report. 1989.

[Arm+11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings.* Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 135–150. ISBN: 978-3-642-25378-2. DOI: `10.1007/978-3-642-25379-9_12`. URL: `http://dx.doi.org/10.1007/978-3-642-25379-9_12`.

[Bar+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. ISBN: 978-3-642-22109-5. DOI: `10.1007/978-3-642-22110-1_14`. URL: `http://dx.doi.org/10.1007/978-3-642-22110-1_14`.

[BBW14] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Precise and Abstract Memory Model for C Using Symbolic Values". In: *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings.* Ed. by Jacques Garrigue. Vol. 8858. Lecture Notes in Computer Science. Springer, 2014, pp. 449–468. ISBN: 978-3-319-12735-4. DOI: `10.1007/978-3-319-12736-1_24`. URL: `http://dx.doi.org/10.1007/978-3-319-12736-1_24`.

[BBW15] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Concrete Memory Model for CompCert". In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings.* Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 67–83. ISBN: 978-3-319-22101-4. DOI: `10.1007/978-3-319-22102-1_5`. URL: `http://dx.doi.org/10.1007/978-3-319-22102-1_5`.

[BCP11] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. "Modular SMT Proofs for Fast Reflexive Checking Inside Coq". In: *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings.* Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 151–166. ISBN: 978-3-642-25378-2. DOI: `10.1007/978-3-642-25379-9_13`. URL: `http://dx.doi.org/10.1007/978-3-642-25379-9_13`.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. "Formal Verification of a C Compiler Front-End". In: *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings.* Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, pp. 460–475. ISBN: 3-540-37215-6. DOI: `10.1007/11813040_31`. URL: `http://dx.doi.org/10.1007/11813040_31`.

[Bed+12] Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. "Formally verified optimizing compilation in ACG-based flight control software". In: *ERTS2 2012: Embedded Real Time Software and Systems.* AAAF, SEE. Toulouse, France, Feb. 2012. URL: `https://hal.inria.fr/hal-00653367`.

[BFT15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.5.* Technical Report. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2015.

[Bla+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A Static Analyzer for Large Safety-critical Software". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation.* PLDI '03. San Diego, California, USA: ACM, 2003, pp. 196–207. ISBN: 1-58113-662-5. DOI: `10.1145/781131.781153`. URL: `http://doi.acm.org/10.1145/781131.781153`.

[Bla07] Sandrine Blazy. "Experiments in validating formal semantics for C". In: *C/C++ Verification Workshop.* Raboud University Nijmegen report ICIS-R07015, 2007.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. "The Security Impact of a New Cryptographic Library". In: *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings.* Ed. by Alejandro Hevia and Gregory Neven. Vol. 7533. Lecture Notes in Computer Science. Springer, 2012, pp. 159–176. ISBN: 978-3-642-33480-1. DOI: `10.1007/978-3-642-33481-8_9`. URL: `http://dx.doi.org/10.1007/978-3-642-33481-8_9`.

[BR10] Gogul Balakrishnan and Thomas Reps. "WYSINWYX: What You See is Not What You eXecute". In: *ACM Transactions on Programming Languages and Systems* 32.6 (Aug. 2010), 23:1–23:84. ISSN: 0164-0925. DOI: `10.1145/1749608.1749612`. URL: `http://doi.acm.org/10.1145/1749608.1749612`.

[BT16] Sandrine Blazy and Alix Trieu. "Formal Verification of Control-flow Graph Flattening". In: *CPP'16.* To appear. ACM, 2016.

[Car+14] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. "End-to-end verification of stack-space bounds for C programs". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014.* Ed. by Michael F. P. O'Boyle and Keshav Pingali. ACM, 2014, p. 30. ISBN: 978-1-4503-2784-8. DOI: `10.1145/2594291.2594301`. URL: `http://doi.acm.org/10.1145/2594291.2594301`.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977.* Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973.

[CS94] J. V. Cook and S. Subramanian. *A formal semantics for C in Nqthm.* Technical Report. Trusted Information Systems, 1994.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations.* Technical Report. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[Doc12] Robert Dockins. "Operational Refinement for Compiler Correctness". PhD thesis. Princeton University, 2012.

[DS07] David Delmas and Jean Souyris. "Astrée: From Research to Industry". In: *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings.* Ed. by Hanne Riis Nielson and Gilberto Filé. Vol. 4634. Lecture Notes in Computer Science. Springer, 2007, pp. 437–451. ISBN: 978-3-540-74060-5. DOI: 10.1007/978-3-540-74061-2_27. URL: http://dx.doi.org/10.1007/978-3-540-74061-2_27.

[Dur+14] Zakir Durumeric et al. "The Matter of Heartbleed". In: *Proceedings of the 2014 Conference on Internet Measurement Conference.* IMC '14. Vancouver, BC, Canada: ACM, 2014, pp. 475–488. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663755. URL: http://doi.acm.org/10.1145/2663716.2663755.

[ER12] Chucky Ellison and Grigore Rou. "An Executable Formal Semantics of C with Applications". In: *SIGPLAN Not.* 47.1 (Jan. 2012), pp. 533–544. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/2103621.2103719.

[Flo67] Robert W Floyd. "Assigning meanings to programs". In: *Mathematical aspects of computer science* 19.19-32 (1967), p. 1.

[GH92] Yuri Gurevich and James K. Huggins. "The Semantics of the C Programming Language". In: *Computer Science Logic, 6th Workshop, CSL '92, San Miniato, Italy, September 28 - October 2, 1992, Selected Papers.* Ed. by Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini, and Michael M. Richter. Vol. 702. Lecture Notes in Computer Science. Springer, 1992, pp. 274–308. ISBN: 3-540-56992-8. DOI: 10.1007/3-540-56992-8_17. URL: http://dx.doi.org/10.1007/3-540-56992-8_17.

[Gu+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. "Deep Specifications and Certified Abstraction Layers". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 595–608. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676975. URL: http://doi.acm.org/10.1145/2676726.2676975.

[Hal+08] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. "Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses". In: *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society, 2008, pp. 129–142. ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.31. URL: http://dx.doi.org/10.1109/SP.2008.31.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Rou. "Defining the undefinedness of C". In: *PLDI'15*. ACM, 2015, pp. 336–345.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[ISO] ISO. *WG14 Defect Report #260*. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

[ISO11] ISO. *C Standard 2011*. Technical Report. ISO, 2011. URL: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf.

[ISO99] ISO. *C Standard 1999*. Technical Report. ISO, 1999. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[Jou+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. "A formally-verified C static analyzer". In: *POPL 2015: 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Mumbai, India: ACM, Jan. 2015, pp. 247–259. DOI: 10.1145/2676726.2676966. URL: https://hal.inria.fr/hal-01078386.

[JR78] Stephen C. Johnson and Dennis M. Ritchie. "UNIX Time-Sharing System: Portability of C Programs and the UNIX System". In: *Bell System Technical Journal* 57.6 (1978), pp. 2021–2048. ISSN: 1538-7305. DOI: 10.1002/j.1538-7305.1978.tb02141.x. URL: http://dx.doi.org/10.1002/j.1538-7305.1978.tb02141.x.

[JS11] Jean-Pierre Jouannaud and Zhong Shao, eds. *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011. ISBN: 978-3-642-25378-2. DOI: 10.1007/978-3-642-25379-9. URL: http://dx.doi.org/10.1007/978-3-642-25379-9.

[Kan+15] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. "A formal C memory model supporting integer-pointer casts". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 326–335. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2738005. URL: http://doi.acm.org/10.1145/2737924.2738005.

[Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Aspects of Computing* 27.3 (2015), pp. 573–609. ISSN: 1433-299X. DOI: 10.1007/s00165-014-0326-7. URL: http://dx.doi.org/10.1007/s00165-014-0326-7.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978. ISBN: 0-13-110163-3.

[Kre15] Robbert Krebbers. "The C standard formalized in Coq". PhD thesis. Radboud Universiteit Nijmegen, 2015.

[KSA14] Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel. "Portable Software Fault Isolation". In: *CFS 2014*. IEEE, 2014, pp. 18–32. URL: `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6954678`.

[KV08] Johannes Kinder and Helmut Veith. "Jakstab: A Static Analysis Platform for Binaries". In: *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings*. Ed. by Aarti Gupta and Sharad Malik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 423–427. ISBN: 978-3-540-70545-1. DOI: `10.1007/978-3-540-70545-1_40`. URL: `http://dx.doi.org/10.1007/978-3-540-70545-1_40`.

[LB08] Xavier Leroy and Sandrine Blazy. "Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations". In: *Journal of Automated Reasoning* 41.1 (2008), pp. 1–31. DOI: `10.1007/s10817-008-9099-0`. URL: `http://dx.doi.org/10.1007/s10817-008-9099-0`.

[Lea] Doug Lea. *A Memory Allocator*. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[Ler+14] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. "The CompCert memory model". In: *Program Logics for Certified Compilers*. Cambridge University Press, 2014. ISBN: 9781107048010. URL: `http://hal.inria.fr/hal-00905435`.

[Ler09a] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom. Reasoning* 43.4 (2009), pp. 363–446. DOI: `10.1007/s10817-009-9155-4`. URL: `http://dx.doi.org/10.1007/s10817-009-9155-4`.

[Ler09b] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115. DOI: `10.1145/1538788.1538814`. URL: `http://doi.acm.org/10.1145/1538788.1538814`.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: `10.1007/978-3-540-78800-3_24`. URL: `http://dx.doi.org/10.1007/978-3-540-78800-3_24`.

[Mem+16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. "Into the depths of C: elaborating the de facto standards". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery Berger. ACM, 2016, pp. 1–15. ISBN: 978-1-4503-4261-2. DOI: `10.1145/2908080.2908081`. URL: `http://doi.acm.org/10.1145/2908080.2908081`.

[Mil89] Robin Milner. *Communication and concurrency.* PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2.

[Mor+12] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. "RockSalt: better, faster, stronger SFI for the x86". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012.* Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 395–404. ISBN: 978-1-4503-1205-9. DOI: `10.1145/2254064.2254111`. URL: `http://doi.acm.org/10.1145/2254064.2254111`.

[Mot04] Motor Industry Software Reliability Association. *MISRA-C: 2004 – Guidelines for the use of the C language in critical systems.* 2004.

[Nor98] Michael Norrish. "C formalised in HOL". PhD thesis. University of Cambridge, 1998.

[Plo81] Gordon D. Plotkin. *A Structural Approach to Operational Semantics.* Lecture Notes. University of Aarhus, 1981. URL: `http://citeseer.ist.psu.edu/plotkin81structural.html`.

[Ric53] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Trans. Amer. Math. Soc.* 74 (1953), pp. 358–366.

[RL12] Valentin Robert and Xavier Leroy. "A Formally-Verified Alias Analysis". In: *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings.* Ed. by Chris Hawblitzel and Dale Miller. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 11–26. ISBN: 978-3-642-35307-9. DOI: `10.1007/978-3-642-35308-6_5`. URL: `http://dx.doi.org/10.1007/978-3-642-35308-6_5`.

[Sam14] Miro Samek. *Are we shooting ourselves in the foot with Stack Overflow?* `http://embeddedgurus.com/state-space/2014/02/are-we-shooting-ourselves-in-the-foot-with-stack-overflow/`. Blog post. 2014.

[Sev+11] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "Relaxed-memory concurrency and verified compilation". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 43–54. ISBN: 978-1-4503-0490-0. DOI: `10.1145/1926385.1926393`. URL: `http://doi.acm.org/10.1145/1926385.1926393`.

[SS71] Dana S Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages.* Vol. 1. Oxford University Computing Laboratory, Programming Research Group, 1971.

[Tan+16] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. "A New Verified Compiler Backend for CakeML". In: *ICFP'16.* ACM, 2016.

[Wah+93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation". In: *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993.* Ed. by Andrew P. Black and Barbara Liskov. ACM, 1993, pp. 203–216.

ISBN: 0-89791-632-8. DOI: `10.1145/168619.168635`. URL: `http://doi.acm.org/10.1145/168619.168635`.

[Wan+12] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. "Undefined behavior: what happened to my code?" In: *Asia-Pacific Workshop on Systems, APSys '12, Seoul, Republic of Korea, July 23-24, 2012*. ACM, 2012, p. 9. ISBN: 978-1-4503-1669-9. DOI: `10.1145/2349896.2349905`. URL: `http://doi.acm.org/10.1145/2349896.2349905`.

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 283–294. ISBN: 978-1-4503-0663-8. DOI: `10.1145/1993498.1993532`. URL: `http://doi.acm.org/10.1145/1993498.1993532`.

[Yee+10] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: a sandbox for portable, untrusted x86 native code". In: *Commun. ACM* 53.1 (2010), pp. 91–99. DOI: `10.1145/1629175.1629203`. URL: `http://doi.acm.org/10.1145/1629175.1629203`.

[You89] William D. Young. "A Mechanically Verified Code Generator". In: *J. Autom. Reason.* 5.4 (Nov. 1989), pp. 493–518. ISSN: 0168-7433. URL: `http://dl.acm.org/citation.cfm?id=83471.83479`.

Appendices

Appendix A

Notations

Partial functions. We use the notation $A \rightharpoonup B$ to denote partial function types. The actual type in the Coq implementation uses option types and reads $A \to \texttt{option}\, B$. When the function is defined and returns a value v, we write $\lfloor v \rfloor$ (`Some v` in Coq). Otherwise, when it fails to produce a value, we write $\emptyset$ (`None` in Coq).

Function update. The notation $f[x \mapsto y]$ represents a function which behaves as f except for input x, for which the function returns y.

Array notation. We use the array notation for accesses to finite maps in CompCert. Finite maps are used to model the contents of the memory, for example. We write $m[b][o]$ for the access to the offset o in block b in the memory state m.

Boolean values. When talking about C programs that manipulate *boolean values*, we write `true` for $\texttt{int}(1)$ and `false` for $\texttt{int}(0)$, as is standard.

VU:	VU:
Le Directeur de Thèse	**Le Responsable de l'École Doctorale**
(Nom et Prénom)	

VU pour autorisation de soutenance
Rennes, le

Le Président de l'Université de Rennes 1

David ALIS

VU après soutenance pour autorisation de publication

Le Président de Jury
(Nom et Prénom)