Formal Validation of Intra-Procedural Transformations by Defensive Symbolic Simulation

Léo Gourdin

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : VERIMAG

Validation Formelle de Transformations Intra-Procédurales par Simulation Symbolique Défensive

Formal Validation of Intra-Procedural Transformations by Defensive Symbolic Simulation

Présentée par :

Léo GOURDIN

Direction de thèse :

Sylvain BOULMÉ MAITRE DE CONFERENCES, Université Grenoble Alpes	Directeur de thèse
Frédéric PÉTROT PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Co-directeur de thèse

Rapporteurs :

Jean-Christophe FILLIÂTRE
DIRECTEUR DE RECHERCHE, CNRS Délégation Ile-de-France

Jens KNOOP
PROFESSEUR, Institute of Information Systems Engineering - TU Wien

Thèse soutenue publiquement le **12 décembre 2023**, devant le jury composé de :

Sylvain BOULMÉ MAITRE DE CONFERENCES, Université Grenoble Alpes	Directeur de thèse
Delphine DEMANGE MAITRESSE DE CONFERENCES, Université de Rennes	Examinatrice
Jean-Christophe FILLIÂTRE DIRECTEUR DE RECHERCHE, CNRS Délégation Ile-de-France	Rapporteur
Jens KNOOP PROFESSEUR, Institute of Information Systems Engineering - TU Wien	Rapporteur
Frédéric PÉTROT PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Co-directeur de thèse
Marc POUZET PROFESSEUR DES UNIVERSITES, École Normale Supérieure	Examinateur
Gwen SALAÜN PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Président

À René...

FOREWORD

HOW TO READ THIS DOCUMENT?

I recommend reading a digital, or at least colored, version of this document; but a printed version should be ok as well. Chapters, sections, or references marked with a dagger † present a personal contribution of my PhD work. The exhaustive list of my contributions is given in §1.3. Paragraphs, examples, or figures marked with an asterisk ∗ are partially or totally reused from another document (paper, article, draft) to illustrate a specific concept. Important notions are emphasized in *italic*, and very important ones in violet. Most folded acronyms used in the document are clickable links pointing to their definition.

The CompCert fork from Verimag, on which I worked during my PhD, is named Chamois-CompCert (see its brief description in §3.6). The whole Coq and OCaml code described in this document is available online in a frozen version of the Chamois-CompCert fork dedicated to this manuscript. See the git repository at `https://framagit.org/yukit/compcert-chamois-gl-thesis`. Some types, definitions, lemmas, or theorems are given in a mathematical notation rather than in Coq, mainly to simplify reading. Their actual implementation or documentation, when relevant to read, is linked with symbol [⋄] (in the digital version).

Throughout this document, I use the term "mainline CompCert" to refer to the public version of CompCert (3.12) distributed by AbsInt, and available on GitHub at `https://github.com/AbsInt/CompCert`.

Since all the mathematical proofs presented throughout this document are formalized in the Coq proof assistant, with their code publicly available online (URL above), I will not detail every lemma or theorem, but I will rather give an *intuition* of proof decomposition.

ABSTRACT

Compilers are highly complex software systems and may, therefore, contain bugs. These bugs can result in errors during the compilation process, or, much more annoyingly, in the generation of incorrect code. Bugs that subtly alter the semantics of generated programs are often very insidious and challenging to trace. In certain applications, particularly in embedded, safety-critical systems subject to stringent regulations and requirements (e.g. avionics, trains, etc.), eliminating these bugs is of paramount importance.

Although most of these bugs are typically found in optimization passes, disabling optimizations is not a viable option in many applications. In fact, simply turning off optimizations is insufficient to guarantee bug-free code. Regulatory standards often necessitate the use of simple, predictable processors, heavily reliant on the compiler for performance.

An alternative solution is to employ a certified compiler, mechanically proven correct in a proof assistant. Such a compiler ensures that the generated assembly code faithfully preserves the source code's semantics. CompCert belongs to this category, and stands as the first formally verified C compiler widely used in the industry. However, proving the correctness of intricate optimizations remains a challenge. This is why certified compilers, including CompCert, produce code that is significantly less performant compared to mainstream compilers like GCC or LLVM.

Translation validation offers a technique where only the result of an optimization is verified, rather than proving the correctness of its implementation. The optimization algorithm, referred to as an oracle, remains untrusted. Nevertheless, its results are always subjected to validation by a proven validator designed to reject any errors.

In this thesis, we delve into the concept of guided translation validation. The principle is to allow oracles to guide the validator by providing hints that reduce the search space, thereby minimizing the complexity of the validation process. Specifically, we propose a formally verified symbolic interpreter capable of validating an entire class of transformations. Our tool requests program invariants from oracles as hints to drive the symbolic simulation of both the original and optimized code. The proven simulation test defensively validates the applied optimizations, ensuring consistency with the unoptimized code.

We have successfully validated several new transformations using this approach, including some that had never been formally verified before, thanks to the communication between oracles and their validator. Notably, we verify a strength-reduction optimization targeting 64-bit RISC-V architectures, which show promise in the context of safety-critical embedded systems. In addition to strength-reduction, our symbolic simulation framework also supports partial redundancy elimination, dead code elimination, code motion, scheduling, and weak software pipelining with renaming.

We have integrated our validation mechanism into a fork of CompCert through the development of a new intermediate language called Block Transfer Language, BTL. Translations to and from BTL are also defensively validated, accomplished with a separate, formally verified checker capable of validating code duplication and factorization as control-flow graph morphisms. To rigorously assess the impact of our optimizations and the overhead introduced by their validation, we conducted multiple experimental measurements of both compilation time and runtime performance. Platform specific optimizations were tested on both AArch64 and RISC-V architectures. Results show a significant improvement of the runtime performance while maintaining a reasonable compilation time.

In the future, this same method could potentially be applied to validate other transformations, such as the automatic insertion of security countermeasures. Our designs appear to be applicable beyond CompCert.

Keywords: Formal verification, Translation validation, Symbolic execution, Compiler optimizations, RISC-V, the Coq proof assistant, the CompCert compiler.

RÉSUMÉ

Les compilateurs sont des systèmes logiciels très complexes et peuvent donc contenir des bogues. Ces bogues peuvent se traduire par des erreurs au cours du processus de compilation ou, plus ennuyeusement encore, par la génération d'un code incorrect. Les bogues qui altèrent subtilement la sémantique des programmes générés sont souvent très insidieux et difficiles à retracer. Dans certaines applications, en particulier dans les systèmes embarqués critiques pour la sécurité et sujets à des exigences et régulations strictes (par exemple, avionique, trains, etc.), l'élimination de ces bogues est d'une importance capitale.

Bien que la plupart de ces bogues soient typiquement situés dans les passes d'optimisation, la désactivation des optimisations n'est pas une solution viable dans de nombreuses applications. En fait, la simple désactivation des optimisations ne suffit pas à garantir un code exempt de bogues. Les normes réglementaires imposent en général l'utilisation de processeurs simples et prédictibles, dont la performance dépend largement du compilateur.

Une solution alternative est d'employer un compilateur certifié, mécaniquement prouvé correct dans un assistant de preuve. Un tel compilateur assure que le code assembleur généré préserve fidèlement la sémantique du code source. CompCert appartient à cette catégorie, et est le premier compilateur C formellement vérifié et largement utilisé dans l'industrie. Cependant, prouver la correction d'optimisations complexes reste un défi. C'est pourquoi les compilateurs certifiés, y compris CompCert, produisent un code significativement moins performant que les compilateurs classiques tels que GCC ou LLVM.

La validation de traduction est une technique où seul le résultat d'une optimisation est vérifié, plutôt que de prouver la correction de son implémentation. L'algorithme d'optimisation, appelé oracle, reste considéré comme non fiable. Néanmoins, ses résultats sont toujours soumis à la validation par un validateur prouvé et conçu pour rejeter toute erreur.

Dans cette thèse, nous approfondissons le concept de validation de traduction guidée. Le principe est de permettre aux oracles de guider le validateur en lui fournissant des indices qui réduisent l'espace de recherche, minimisant ainsi la complexité du processus de validation. Plus précisément, nous proposons un interpréteur symbolique formellement vérifié capable de valider toute une classe de transformations. Notre outil demande aux oracles des invariants de programme en tant qu'indices pour guider la simulation symbolique du code original et du code optimisé. Le test de simulation prouvé valide défensivement les optimisations appliquées, en garantissant leur cohérence vis-à-vis du code non optimisé.

Nous avons validé avec succès plusieurs nouvelles transformations en utilisant cette approche, dont certaines n'avaient jamais été formellement vérifiées jusqu'alors, grâce à la communication entre les oracles et leur validateur. Notamment, nous avons vérifié une optimisation de "strength-reduction" (littéralement, "réduction de force") ciblant les architectures RISC-V 64 bits, qui sont prometteuses dans le contexte des systèmes embarqués critiques pour la sécurité. En plus de la "strength-reduction", notre outil de simulation symbolique supporte l'élimination de redondances partielles, l'élimination du code mort, le déplacement de code, l'ordonnancement, et une forme faible de pipeline logiciel avec renommage.

Nous avons intégré notre mécanisme de validation dans notre version de développement (fork) de CompCert, en développant une nouvelle représentation intermédiaire nommée "Block Transfer Language", BTL (littéralement "Langage de Transfert en Blocs"). Les traductions de et vers BTL sont également validées de manière défensive, à l'aide d'un vérificateur dédié, formellement vérifié, et capable de valider de la duplication et factorisation de code en tant que morphismes des graphes de flux de contrôle. Pour évaluer rigoureusement l'impact de nos optimisations et le temps de compilation supplémentaire induit par leur validation, nous avons effectué de multiples mesures expérimentales du temps de compilation et des performances à l'exécution. Les optimisations spécifiques à une architecture cible ont été testées sur des plateformes AArch64 et RISC-V. Les résultats montrent une amélioration significative des performances à l'exécution tout en maintenant un temps de compilation raisonnable.

À l'avenir, cette même méthode pourrait potentiellement être appliquée pour valider d'autres transformations, comme l'insertion automatique de contre-mesures de sécurité. Nos conceptions semblent être applicables au-delà de CompCert.

Mots clefs : Vérification formelle, Validation de traduction, Exécution symbolique, Optimisations de compilateur, RISC-V, l'assistant de preuve Coq, le compilateur certifié CompCert.

ACKNOWLEDGMENTS

I'd like to start by expressing my deepest thanks to Sylvain. Thank you for all the help you gave me in designing and completing this thesis, for your advice, the time you spent discussing and reflecting with me, the many ideas and comments you contributed, and without which I wouldn't have been able to complete this work. Thank you for believing in me and listening to me during these three years. Thank you for everything. I'd also like to extend the same profound thanks to Frédéric; it hasn't always been easy to find the time to work together at TIMA, but you've always been available and willing to listen, to help, and support me. I'm truly grateful. My sincere thanks also to David. Although you were not officially my thesis supervisor, you've been a great help to me during these three years, with your many ideas, your enthusiasm, and your patience in the rigorous proofreading of this manuscript. I'll never have enough words to express how grateful I am to all three of you.

I'd like to thank all the members of my thesis jury. Thank you, Jean-Christophe Filliâtre and Jens Knoop for agreeing to read, review, and report on this manuscript, and for providing helpful and thoughtful feedback. It's a great honor. Most notably, Jean-Christophe Filliâtre, thank you for your extensive notes on the manuscript; Jens Knoop, thank you for the very interesting scientific discussion we had before the reports. Thank you also to Marc Pouzet and Gwen Salaün for accepting to be part of my thesis jury and for your interest in my thesis. Thank you, Delphine Demange, not only for being part of this jury but also for agreeing to follow my thesis as an external expert over the past three years. Our exchanges were important and meaningful for me, and your outside view helped me to have confidence in my work.

I also thank all my colleagues in the Verimag laboratory, who have given me the chance to work in a pleasant and caring environment. I'd especially like to thank Karine for following my progress over the last six years, and for listening to me and advising me on my decision to do a thesis three years ago. I'm also incredibly grateful to Olivier, with whom it's always been a pleasure to work. Thank you for your constant good mood, the moments spent laughing, and our endless discussions. You're the person I knew I needed to talk to to cheer me up! I'm thankful as well to Marie-Laure, who allows me to continue my research at Verimag and to meet the security research community.

I'd also like to extend my warmest thanks to all my friends from the academic world, for all our moments spent together, our coffee breaks, our discussions, our odd days at Eve, and for your essential good humor. Aina, Alban, Alexandre H. from Inria, and Alexandre B. from Verimag, Ana, Baptiste, Basile my new office colleague, Cyril, Hadi, Lucas, Marco, Doctor Thomas M., not-yet-doctor Thomas V. Thank you for being there. I'm really happy to have been able to meet such good friends during this thesis, and I wish you the best. Thank you also to Abderrahmane, Benjamin, Etienne, Oussama, and all the people I had good times with.

I'd like to give a huge thank you to my past roommates Léa and Olivier, and to my present roommates Adrien, Tom, and Vincent. And Solenne, who is almost a roommate too. Thank you for putting up with my changing moods, and thank you for your support, which really meant a lot to me during these three years.

I'd like to send a special thank you to Soline. Thank you so much for being there; you're certainly one of the nicest people I've met, and my best friend.

I'd like to thank my family as well. Mom, Dad, Eric, thank you from the bottom of my heart for all your help and love. Thank you for supporting me, listening, and thank you for allowing me to spend these eight years at university peacefully. Christine, Stéphane, Lola, thank you for being there in Grenoble and always available to spend time together. More generally, thank you to my whole family for their unconditional love and kindness.

Last but not least, I'd like to send a special thank you to Leila. I can't even express how much you helped me. Thanks for everything, really.

REMERCIEMENTS

Je voudrais commencer par exprimer mes plus sincères remerciements à Sylvain. Merci pour toute l'aide que tu m'as apportée dans la conception et la réalisation de cette thèse, pour tes conseils, le temps que tu as consacré à discuter et à réfléchir avec moi, les nombreuses idées et commentaires que tu as apportés, et sans lesquels je n'aurais pas pu mener à bien ce travail. Merci d'avoir cru en moi et de m'avoir écouté pendant ces trois ans. Merci pour tout. J'aimerais également étendre ces remerciements sincères à Frédéric ; ce n'a pas toujours été facile de trouver le temps pour travailler ensemble à TIMA, mais tu as toujours été disponible et prêt à écouter, à aider, et à me soutenir. J'en suis vraiment reconnaissant. Mes remerciements les plus sincères également à David. Bien que tu n'aies pas été officiellement mon directeur de thèse, tu m'as énormément aidé au cours de ces trois ans, avec tes nombreuses idées, ton enthousiasme, et ta patience dans la relecture rigoureuse de ce manuscrit. Je n'aurai jamais assez de mots pour exprimer ma gratitude envers vous trois.

Je tiens à remercier tous les membres de mon jury de thèse. Merci, Jean-Christophe Filliâtre et Jens Knoop, d'avoir accepté de lire, de passer en revue, et de rapporter sur ce manuscrit, et pour vos commentaires utiles et réfléchis. C'est un grand honneur. Plus particulièrement, Jean-Christophe Filliâtre, merci pour vos nombreuses annotations sur le manuscrit ; Jens Knoop, merci pour la discussion scientifique très intéressante que nous avons eue avant les rapports. Merci aussi à Marc Pouzet et Gwen Salaün d'avoir accepté de faire partie de mon jury et pour votre intérêt pour ma thèse. Merci, Delphine Demange, non seulement de faire partie de ce jury, mais aussi d'avoir accepté de suivre ma thèse en tant qu'experte extérieure durant ces trois ans. Nos échanges ont été importants et significatifs pour moi, et ton regard extérieur m'a aidé à avoir confiance en mon travail.

Je remercie aussi tous mes collègues du laboratoire Verimag, qui m'ont donné l'opportunité de travailler dans un environnement agréable et bienveillant. J'aimerais particulièrement te remercier, Karine, pour avoir suivi ma progression au cours des six dernières années, et de m'avoir écouté et conseillé lorsque j'ai pris la décision de faire une thèse il y a trois ans. Je suis également incroyablement reconnaissant envers Olivier, avec qui il a toujours été un plaisir de travailler. Merci pour ta bonne humeur constante, les moments passés à rire, et nos discussions interminables. Tu es la personne avec qui je savais qu'il fallait parler pour me remonter le moral ! Je suis également reconnaissant envers Marie-Laure, qui me permet de poursuivre ma recherche à Verimag et de rencontrer la communauté de recherche en sécurité.

J'aimerais également étendre mes remerciements les plus chaleureux à tous mes amis du monde académique, pour tous nos moments passés ensemble, nos pauses café, nos discussions, nos jours impairs à Eve, et pour votre bonne humeur essentielle. Aina, Alban, Alexandre H. de l'Inria, et Alexandre B. de Verimag, Ana, Baptiste, Basile, mon nouveau collègue de bureau, Cyril, Hadi, Lucas, Marco, docteur Thomas M., futur docteur Thomas V. Merci d'être là. Je suis vraiment heureux d'avoir pu rencontrer de si bons amis au cours de cette thèse, et je vous souhaite tout le meilleur. Merci également à Abderrahmane, Benjamin, Etienne, Oussama, et à toutes les personnes avec lesquelles j'ai passé de bons moments.

J'aimerais donner un immense merci à mes anciens colocataires, Léa et Olivier, et à mes colocataires actuels, Adrien, Tom, et Vincent. Et Solenne, qui est presque une colocataire aussi. Merci de supporter mes humeurs changeantes, et merci pour votre soutien, qui a signifié vraiment beaucoup pour moi au cours de ces trois ans.

J'aimerais envoyer un remerciement spécial à Soline. Merci beaucoup d'être là ; tu es certainement l'une des personnes les plus gentilles que j'ai rencontrées, et ma meilleure amie.

J'aimerais également remercier ma famille. Maman, Papa, Éric, merci du fond du cœur pour toute votre aide et votre amour. Merci de me soutenir, de m'écouter, et merci de m'avoir permis de passer ces huit ans à l'université en toute tranquillité. Christine, Stéphane, Lola, merci d'être là à Grenoble et toujours disponibles pour passer du temps ensemble. Plus généralement, merci à toute ma famille pour son amour inconditionnel et sa gentillesse.

Enfin, mais non le moindre, j'aimerais envoyer un remerciement spécial à Leila. Je ne peux même pas exprimer à quel point tu m'as aidé. Merci pour tout, vraiment.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ABI application binary interface
ALU arithmetic-logic unit
BTL block transfer language
CFG control-flow graph
CIC calculus of inductive constructions
CM code motion
CPS continuation passing style
CSE common subexpression elimination
CSASV compact sequence of assignments of symbolic values
DAG directed acyclic graph
DCE dead code elimination
DFA deterministic finite automaton
DSL domain specific language
FFI foreign function interface

FPASV finite parallel assignment of symbolic values
FRE full redundancy elimination
FVDP formally verified defensive programming
GI gluing invariant
GVN global value numbering
HI history invariant
ILP instruction level parallelism
IR intermediate representation
IR-points insertion and replacement points
ISA instruction set architecture
LCT lazy code transformations
LCM lazy code motion
LHS left hand-side
LFTR linear-function test replacement
LICM loop-invariant code motion
LTL location transfer language
LSR lazy strength-reduction
OoO out-of-order
PRE partial redundancy elimination
RHS right hand-side
RSD relative standard deviation
RTL register transfer language
SCS safety-critical systems
SE symbolic execution
SMT satisfiability modulo theories
SR strength-reduction
SSA static single assignment
TCB trusted computing base
VLIW very long instruction word
WCET worst-case execution time
WLP weakest liberal precondition

1

INTRODUCTION

The following chapter introduces the context and the motivations behind this work. Section 1.1 defines the notion of certified compilation and explains why it is necessary; I motivate the main topic of my thesis in Section 1.2; and a list of contributions is provided in Section 1.3. Finally, Section 1.4 gives a quick outline of this document.

1.1 CERTIFIED COMPILERS

A certified compiler comes with a mechanical proof of correctness, written using a proof assistant such as Coq[1]. Essentially, it means that the target program's semantics is unchanged: assuming deterministic source and target languages, the *observable behavior* of the source must imply the target's one (a more formal explanation is provided in §3.2.1). Therefore, if the source program does not crash, neither does the target.

A language is termed deterministic if a program's behavior in that language is solely determined by its inputs, and not by any internal choices.

1.1.1 *Security and Safety of Programs and Languages*

Programming languages are *standardized* in documents describing their semantics, and how the compiler should translate them into machine code. Often, these standards are mostly written in English, and they might be very difficult to understand completely (700 pages for the C11 ISO/IEC 9899:2011 standard). Depending on the source language, a program might be subject or not to various security and safety issues.

SAFETY Some languages directly integrate safety measures through the use of techniques such as *type checking*: the process of verifying types' consistency and enforcing constraints. Type checking can be static (i.e. based on an analysis of the source code, at compile time) or dynamic (at runtime). The aim of verifying types is to detect and avoid runtime errors, by ensuring some set of safety properties on all possible inputs of the program. Modern languages such as Rust include a lot of safety checks to ensure, for instance, that an array access is never out of bounds. Oppositely, low-level languages like C are far more permissive, and hence writing safe code is the *responsibility of the programmer*. C still features some weak, static type checking, but the latter is limited (it does not cover every language constructs, e.g. variadic functions) and it might be easily, or even inadvertently circumvented.

Since many language constructs are difficult to check statically, some languages combine both static and dynamic type checking.

The C standard lists undefined (or unpredictable) behaviors: cases where the compiler is authorized to do anything, since the semantics is not well-defined. In such situations, the program may either fail to compile, or execute incorrectly, and it is up to the compiler programmer to decide how to compile undefined behaviors. For example, dividing a value by zero, accessing an array out of bounds, or overflowing signed integers are all considered as undefined behaviors.

More generally, safety relates to the *predictability* of the program behavior, and its *conformance* w.r.t. the standard's semantics.

SECURITY Unlike safety, which is concerned with avoiding unexpected results due to the program itself, security is concerned with the program's isolation from the external environment. In a world where almost every machine is connected to a network and susceptible to external attacks, it is essential to assert the security properties of programs. Security issues might be direct *consequences of safety* issues: for instance, the "heartbleed" flaw discovered in OpenSSL in 2012 was in fact a safety exploit of the source language, C. The vulnerability arose from a malicious use of the heartbeat protocol: normally, this mechanism allows the client to send a message and its length to the server,

[1] https://coq.inria.fr/

so that the server returns the same message (with the same length) and maintains the secure connection open. However, due to a lack of safety checks (e.g. bounds checking), an attacker could request a message longer than the one actually sent. The server would then naively return the message, along with the adjacent content of its memory until reaching the requested length.

There are various methods to prevent security attacks, and one of them consists in adding *countermeasures* inside the source code in order to ensure that some path has been taken, or some computation executed. While adopting this preventive approach is interesting, it may prove to be more challenging than anticipated. Indeed, since the principle is to perform additional checks within the program, while keeping the original semantics, a clever compiler will try to *simplify the protected code* to make it faster (by propagating constants, or eliminating dead-code). For instance, many countermeasures consist in adding dead code, which is likely to be removed by optimizations. This introduces a new problem: how can we insert countermeasures without the compiler deleting them?

This discussion highlights the inherent conflict between safety and security measures, and the pursuit of code optimization. For the optimization aspect, a compiler is expected to *eliminate* code redundancies without changing the overall semantics. On the other hand, we may also need to *preserve* some redundancies preventing anomalous executions for security reasons. Another typical example is about the difference between the "official" semantics (i.e. as defined in the standard) description of a construct, and its expected semantics from programmers. Sometimes, mainstream compilers may even choose to favor the programmers' expectations rather than the "official" semantics, thereby assuming that "the use makes the norm" might be the safest strategy.

1.1.2 *Safety-Critical Systems (SCS) & Compilers Bugs*

The goal of a verified compiler is to minimize the risk of introducing errors in the program during the compilation process. The compiler thus focuses on code generation and optimization, by making strong hypotheses on the source program (e.g. no undefined behaviors). To gain confidence in these assumptions, one can first run a (preferably verified) static analysis using a specific tool. In the case of the C language, there exists formally proven static analyzers (e.g. Astrée [40]) enforcing many critical runtime properties.

Bugs in compilers are relatively rare, but are also quite insidious: detecting a bug in the compilation process that only appears at runtime is sometimes very difficult. In particular, when starting from a correct source code and experiencing an incorrect assembly code, changing the source to fix or isolate the bug might make it disappear.

In mainstream compilers, most bugs are found in optimization passes [136, 150]. When working on safety-critical systems (SCS), the use of a verified compiler is a crucial consideration to avoid these bugs. For instance, aircraft flight control software are subject to rigorous safety norms [60], which impose traceability from the source to object code [16]. Without certified compilers, this requirement often excludes the use of optimization passes, resulting in a suboptimal use of the target processor. This is all the more damaging since in this context, using modern, fast cores featuring complex mechanisms such as dynamic reordering, speculation, etc. or multicore processors, with concurrent programming issues, are not necessarily acceptable options. Indeed, these mechanisms can significantly complicate the worst-case execution time (WCET) analysis, inducing timing anomalies and making it much more pessimistic due to the unpredictability of execution paths [41]. Consequently, the hypothesis of *timing compositionality* [103, §6.1.5] is of paramount importance in SCS; roughly, it assumes that we can obtain an upper bound on the overall worst-case timing behavior. Timing compositionality can be achieved by hardware design, notably by using strictly in-order pipelines [71, §5]. Yet, optimizing the code might help in decreasing the WCET, which is great for safety concerns (i.e. for predictability and timing guarantees). As a demonstration, Kästner et al. [88] reported a gain of nearly thirty percent on the WCET of their programs by using the CompCert certified compiler in comparison to an untrusted, non-optimizing compiler.

Recently, researchers have begun to examine the predictability and timing compositionality of modern optimizing cores [69]. Gruin et al. [68] even proposed an open-source RISC-V core capable

of speculative execution, yet devoid of timing anomalies. Nevertheless, sophisticated cores carry a higher risk of bugs and greater power consumption, which does not always make them a desirable choice.

1.1.3 *Main Types of Intermediate Representations*

In this document, we only investigate the first technique: code partitioning into blocks.

To make compilers simpler and more general, we often use specific intermediate representations. For instance, I present in the two sections below two ways of structuring the code that facilitate some transformations or analyses.

1.1.3.1 *Decomposing the Code Into Blocks*

For many optimizations, a prerequisite is to group instructions into blocks of different sizes. For instance, a data-flow algorithm—as the one of Chapter 10—will be faster if it can solve the data-flow equations for entire blocks rather than individual instructions[2]. Moreover, blocks define a frame for the optimization to work with: it is usually a *code fragment* with a single entry point, so we can *reorganize* its content while controlling the effects that it entails on the rest of the program. In the case of a scheduler, the block corresponds to the window inside which the algorithm reorders instructions.

Generally, we distinguish four different types of blocks:

For definitions of extended basic blocks, the reader may refer to (i) the Wikipedia page[3], and (ii) the definition from Knoop, Rüthing, and Steffen [85, Footnote 15].

BASIC BLOCKS: the simplest form of block. Only one entry point, and one exit point (hence, a conditional branch, for example, would terminate the block).

EXTENDED (BASIC) BLOCKS: they are built as a *collection* of basic blocks, where only the first node is allowed to have multiple predecessors, while the other blocks can only have a unique predecessor, itself inside the collection. Their use is mainly for trace scheduling [142], and common subexpression elimination (CSE). Unlike loop-free blocks, extended blocks are a maximal tree of nodes *without* internal joins.

SUPERBLOCKS: are a particular case of extended basic blocks; they allow to have multiple exits, but with some constraints. Each non-final conditional branch must have one branch that goes directly out of the block, and another that continues inside. In other words, superblocks generalize basic blocks, such that each instruction of a given block has still at most one successor in this block, but may also branch to another superblock [72]. Their main application is (also) for scheduling [90]. Graphically, the control-flow graph (CFG) of instructions for superblocks looks like a comb.

LOOP-FREE BLOCKS: those are the largest form supported by the intermediate representation (IR), block transfer language (see Part ii), introduced in this thesis, and represent a *directed acyclic graph* (DAG) (e.g. a *tree* of instructions allowing internal joins).

1.1.3.2 *Static Single Assignment*

Another solution that helps in optimizing the code, and in particular by greatly facilitating data-flow analysis and the implementation of optimizations, is the static single assignment (SSA) form. This representation is widely used in popular compilers such as GCC & LLVM. Its principle is to assign each pseudo-register exactly once, by renaming and using a fresh pseudo-register for each assignment [43].

It is therefore much easier to build a structure of use-definition chain from an SSA graph, thanks to this single assignment invariant. The name "static" expresses that SSA does not take into account the runtime behavior of the program: an assignment within a loop is considered as single if its destination is not written elsewhere, no matter if the loop is executed several times.

SSA is close to lambda calculus [7, 79], and as pointed out by Chakravarty, Keller, and Zadarnowski [30], using this functional representation can be a solution for formalizing and reasoning about SSA

[2] E.g. in the case of the lazy code transformations algorithm proposed in this thesis, we use basic blocks.
[3] https://en.wikipedia.org/wiki/Extended_basic_block

forms. As of today, the most advanced work of an SSA formalization (within the CompCert compiler) is from Demange [45]. I briefly describe it in §2.5.3.

1.2 PURPOSE OF THIS WORK

I suggest below reasons and examples that motivate this work. Notably, I sketch a high-level view of the main contribution of this work: the idea of verifying global transformations by using a block-based simulation modulo invariants.

1.2.1 *Motivations*

The problem of program equivalence is undecidable in general (Rice's theorem). Yet, the need for certified compilation, especially concerning critical systems, is well-established (see §1.1.2). Offering a certified, but unoptimized compiler would not suffice to satisfy this need because typical use-cases are often working with a restricted amount of computational power or under specific conditions (limited memory, power consumption, etc.) As of today, CompCert [21, 92, 93] is the only formally verified compiler used at an industrial scale [88][4]. However, compared to the most popular toolchains as GCC and LLVM, it is only *moderately* optimizing.

For a certified compiler to be used in industrial or other applications, as opposed to being a research prototype or a teaching tool, its optimization capabilities are an important criterion. Furthermore, SCS often rely on old processor architectures (such as PowerPC) that feature a simple design. Newest architectures based on open hardware like RISC-V are promising in this domain [6, 52, 102]: the open design is attractive for academics, and the RISC-V instruction set architecture (ISA) is also very modular. Anyone can write a RISC-V extension, opening the possibility of specializing the ISA for each application. On the other hand, the backend was only added recently in CompCert (in 2017), and is not as optimized as the older ones like PowerPC.

Hence, both theoretical and practical research on verified optimizations, security measures, and formal guarantees are essential. Innovations in this domain can help other researchers and industrials in democratizing safe and secure solutions.

My thesis is part of this objective, focusing on the formal verification of optimizations; more precisely, we aim to obtain a framework capable of automatically verifying the correctness of a whole class of transformations (which do not necessarily aim at performance; e.g. the insertion of countermeasures). This desire for a generic validation framework is motivated by the difficulties encountered when trying to directly prove complex algorithms. Considering these difficulties, it would surely be beneficial to reduce the proof effort. The framework I propose in this work is not dependent on the target architecture, although I have mainly targeted the RISC-V backend with rewrites that depend finely on the ISA.

Symbolic execution is part of the translation validation techniques; see §2.2.

The idea is to start from a well-known generic method, symbolic execution (SE), that consists of virtually simulating the execution of programs to compare them. Usually, this method is only useful for transformations operating within a very restrained scope (e.g. with sequential portions of the whole code). In fact, SE might cause complexity issues if run over a program with many execution branches[5] (see §2.2.3). Similarly, how would a simulation work (and scale-up) if there are whole loops in the code? Moreover, some semantic constraints force us to split the code into blocks (e.g. when there is a call or a goto instruction). *But what about several small simulations connected to each other?* It is precisely the point I focused on this thesis: composing *local simulations* to prove the correctness of *global transformations*.

This list is not exhaustive, but suggests four of the most important and missing transformations.

Optimizations missing in CompCert w.r.t. to mainstream toolchains include:

- Strength-reduction (SR), which has the potential to greatly improve the runtime performance[6] of the generated code on some architectures, and in particular on *embedded processors* using a less sophisticated ISA (e.g. RISC-V). The principle is to replace a costly computation with a "cheaper" one (e.g. a right shift of n bits is less costly than a division by 2^n).

[4] The CakeML compiler for the ML functional language is the only other example of a large scale formally verified compiler, but is not used industrially, to the best of our knowledge.

[5] It is useful to visualize the program as a directed graph here.

[6] In terms of execution time or number of cycles.

- Loop-invariant code motion (LICM), which involves pre-computing instructions that would be redundant in a loop (for instance, a costly load instruction in a loop can be anticipated if its target memory cell does not change during the loop's iterations).
- Code motion (CM) in general to optimize the placement of instructions in the code.
- Finely dependent rewrites, to allow the replacement of complex instructions with simpler sequences available in the target's ISA.

Several of the above optimizations can be combined to achieve an even more efficient transformation, and are often performed by the same algorithms.

1.2.2 *A Simplified Example of Global Simulation*

In this section, I present a very simplified version of the simulation principle; in practice, there are a lot more constraints to consider, see §2.2.3, §4.4.3, and §5.1. Let us take the below source pseudocode (left-hand side) and its optimized version (right-hand side):

Example 1.2.1 (A motivating example: verifying loop-invariant code motion)**.**

```
int main(int y, int z) {
  Bhead:
    goto Bbody;
  Bbody:
    if (z > 1000)
      return z;
    x = y * y;
    z = z + x;
    goto Bbody;
}
```

```
int main(int y, int z) {
  Bhead:
    h = y * y;
    goto Bbody;
  Bbody:
    if (z > 1000)
      return z;
    z = z + h;
    goto Bbody;
}
```

We simply applied a LICM pass to take out the "`y * y`" computation, since its value never changes through iterations, by introducing a new variable `h` (in red). Although using a fresh variable name is not essential in this example, it helps to avoid being blocked by dependencies in more general cases (e.g. if `x` was modified just before the "`goto Bbody`" instruction, keeping the same variable would be incorrect). Both codes are separated into two "blocks": the `Bhead` block, which only contains a goto at the beginning, and the `Bbody` block. Our simple simulation test for this example works as follows:

- For each instruction, it updates a state representing the modified variables;
- The simulation aims to virtually execute a source and a target (i.e. transformed) code fragments, and to compare *the resulting states* to determine if they are equivalent;
- To simplify this introduction, we assume that data in memory *can only be accessed via variables* (and not via memory addresses);
- When the code splits in multiple branches, we obtain a state for each branch;
- Variables that are not used in the successor blocks (i.e. *dead* variables) can be ignored (and *removed* from the source's final state), since their value does not impact the semantics. Hence, considering that the optimized program should be able to introduce new, auxiliary variables, the source's final state *devoid of dead variables* must be included in the target's final state.

If we apply those principles for the `Bbody` block of the source code, we will obtain two possible states: either the condition "`z > 1000`" was true, so that `z` keeps its initial value and none of the variables are modified; or it was false, and so the end of the block updates both `x` and `z`. The goal of the simulation here is not to iterate on the loop a thousand times; we stop before looping again (at the last goto), and we simply remember that at this point, we have "`z ≔ z + y * y`" (here, we substituted `x` inside `z`). If the "`h ≔ y * y`" instruction was inserted just above the condition (thus staying in the `Bbody` block), a simulation of the target block would produce the same result as in the source (but the computation would still be inside the loop...)

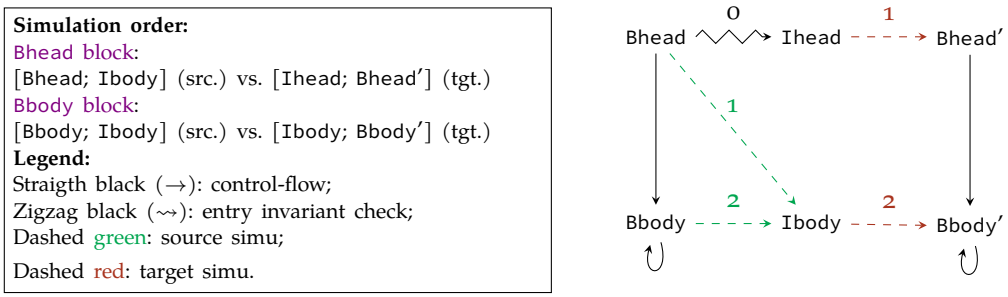


Figure 1.1: Sketch of the Simulation Process of Example 1.2.1.

For a transformation scoped on a single block, this method proves to be quite practical: we can reorder or rewrite instructions, as long as the *simulation result remains equivalent*. Unfortunately, our example here contains a loop, so we cannot simulate the whole program without cutting it into blocks. Taking the `Bbody` block alone is not a problem, since the loop entry point is also the block entry point, and since both branches end with a "final" instruction (return or goto). If the loop entry point was in the middle of the block, we would have to cut it again (i.e. the only possible entry point for a block must be its first instruction). Hence, the simulation is possible only *if loop entries are also block entries*; this is why I formatted it in two blocks, but now we must propagate information about the computation lifted in the `Bhead` block.

The term "invariant" here refers to the simulation invariant needed to preserve the semantics of each block (more details in §9.2).

We do this by adding a notion of invariants at each block entry: these invariants define the relationship between the target block's variables and terms over the source's variables. Invariants are common to both the source and the target blocks. The simulation test executes them in the following way:

- On the source, it executes the block first, and the invariant of the next block after;
- On the target, it is the opposite: the invariant of the block first, then the block.

The invariant for the entry block of a function must always be "trivial": we verify that it includes only *identity relations* of the form `[x ≔ x]`. Let us take the following invariants: `Ihead = [y ≔ y; z ≔ z]` for the `Bhead` block (i.e. the two input variables), and `Ibody = [h ≔ y * y; z ≔ z]` for `Bbody`[7]. Notice the trivial assignments in these invariants: they mean that variables `y` and `z` have the same value in both blocks (i.e. `y` in the target is equal to `y` in the source, and the same applies for `z`).

The execution rules defined above indicate how to verify the global simulation. The process for Example 1.2.1 is sketched in Figure 1.1, where `Bhead`/`Bbody` are the source's blocks, and `Bhead'`/`Bbody'` the target's blocks. After executing this simulation, we obtain (the correspondence with arrows of Figure 1.1 is indicated in parentheses):

1. **For the `Bhead` block:**
   - Entry (0: `Bhead` ⇝ `Ihead`): we ensure that every initial state in `Bhead` satisfies the trivial invariant from `Ihead`, since it is the function's entry point;
   - Source (1: `Bhead+Ibody`): nothing from `Bhead` (since the source's block is just a goto), then our invariant: we end up with "`h ≔ y * y ‖ z ≔ z`";
   - Target (1: `Ihead+Bhead'`): we have `[y ≔ y; z ≔ z]` from the `Bhead` invariant, then "`h ≔ y * y`" by executing the `Bhead'` block: we obtain the same result as in the previous point, with an additional identity assignment for `y` (because it is used in the `Bhead'` block). The result of the source (which does not contain dead variables) is included in the target's one, as expected.
2. **For the `Bbody` block,** when the condition "`z > 1000`" is true, both blocks exit without changing any variables; we ignore this trivial case in the following:
   - Source (2: `Bbody+Ibody`): we already computed `Bbody` above, which was "`z ≔ z + y * y`" (we omit the value of `x` which is not used—i.e. dead—after); and we apply the invariant

[7] In this section, invariants are noted as instruction sequences surrounded by brackets.

of the successor (which is `Bbody` again, so the invariant is `Ibody`), to finally obtain: "`z ≔ z + y * y ‖ h ≔ y * y`";

- Target (2: `Ibody+Bbody'`): the invariant is executed first, so we end up with "`z ≔ z + y * y ‖ h ≔ y * y`". Again, the source's result (without the dead variable `x`) is included in the target's one.

The " ‖ " notation is for parallel assignment: after simulating the block, we represent values as if they were assigned simultaneously (e.g. "`a ≔ b ‖ b ≔ a`" means that `a` is swapped with `b`). Thanks to this technique, we are able to obtain the same result for both the source and the target executions! Indeed, executing well-chosen invariants placed at the entry of each block allows us to propagate information across block boundaries. These invariants—we call them gluing invariants (GIs)—symbolize a *relation* between the target block's variables and the source's ones: for example, the `[h ≔ y * y; z ≔ z] (Ibody)` invariant indicates that variable `h` in the target block is equal to the expression "`y * y`" of the source block, and that `z` does not change. The LICM transformation of Example 1.2.1 thus preserves the program's semantics.

In the rest of this document, I explain how we came up with this method and how we formally proved it over a dedicated intermediate language, which enables us to validate several new optimizations (some of which had never been certified before) in the CHAMOIS-COMPCERT verified compiler.

1.3 CONTRIBUTIONS

1.3.1 *Exhaustive List With Links to Relevant Sections*

Contributions, approximately listed by decreasing significance:

- Introduction of a new IR, block transfer language (BTL), which is a variant of the existing register transfer language (RTL) IR of COMPCERT. BTL partitions the CFG into blocks, and was designed as a basis for the application and defensive verification of transformations. I present BTL in Chapter 5.

- Formalization and implementation of a translation validation framework, proven correct in Coq. This tool defensively validates transformations performed on BTL programs by symbolic execution modulo invariants (provided by untrusted oracles) annotations at block entries. Invariants relate the pseudo-registers of a "source" CFG to those of a "transformed" CFG, enabling the validation of global transformations of the CFG, and typically loop optimizations. The symbolic execution engine is parametrized by transformation-dependent rewriting rules. The framework's theory is formalized in Chapter 6, and its implementation is detailed in Chapter 7.

- Design of an untrusted oracle implementing a combined, enhanced variant of the lazy code motion (LCM) & lazy strength-reduction (LSR) algorithms over BTL. This oracle, named lazy code transformations (LCT), is untrusted by the formal proof of correctness, but provides the invariants necessary for the symbolic validation of its transformations. I detail my contributions to these algorithms and the LCT implementation in Chapter 10.

- Integration of BTL between RTL passes of COMPCERT thanks to another translation validator capable of validating CFG morphisms. Not only this morphism checker validates translations to and from BTL, it also supports the validation of code duplication and factorization. See Chapter 8.

- Implementation of a low-level expansion engine for pseudo-instructions on RISC-V, operating over BTL. This new optimization pass performs a kind of instruction selection, and is also translation validated by the symbolic execution I implemented (using rewriting rules). These expansions increase the opportunities of other optimizations such as the prepass scheduling and the LCT algorithm. The expansion mechanism is first detailed in Section 4.5.2 as a preliminary contribution to another IR, and its port to BTL is explained in Section 7.6.1.

- A port to AArch64 of the postpass scheduling initially realized for KVX platforms by Six, Boulmé, and Monniaux [134]. I also extended it with a peephole optimization to compact loads and stores pairs. This is detailed in Section 4.3
- Development of a testing framework for CompCert. Its design was driven by two primary objectives: firstly, to ensure the trusted computing base (TCB) was rigorously scrutinized through functional tests, helping in the identification of potential bugs in the compiler's trusted components; and secondly, to gauge the performance impact of optimizations across various target architectures. Detailed insights into this framework are provided in Chapter 12.
- Extension of the superblock prepass scheduling of Six et al. [135]†. In particular, the BTL simulation engine enables validating scheduling modulo register renaming and modulo the insertion of compensation code in intermediate exits. The oracles that achieve these extensions have been implemented by interns I co-supervised. BTL extensions, including this one, are covered in Chapter 11.

1.3.2 *Publications*

Below is an exhaustive, chronological list of my publications:

- Gourdin [63]† (June, 2021): A short paper accepted in the PhD student session of AFADL (in French, "Approches Formelles dans l'Assistance au Développement de Logiciels"—meaning "Formal Approaches to Software Development Assistance"). The paper (written in English) presents my port of the Six, Boulmé, and Monniaux [134]'s postpass scheduler and the design of a new peephole optimization on AArch64.
- Gourdin and Boulmé [67]† (July, 2021): An abstract accepted in the Coq-Workshop, accompanied by a presentation (in remote). Slides: https://coq-workshop.gitlab.io/2021/slides/Coq2021-04-01-slides-certifying-optimizations-hash-consing.pdf; Video record: https://www.youtube.com/watch?v=T8vRRguYG4A. This work is also about the AArch64 scheduler and peephole, but is more focused on the Coq implementation using symbolic execution and hash-consing.
- Six et al. [135]† (January, 2022): A CPP ("Certified Programs and Proofs") paper entitled "Formally Verified Superblock Scheduling". It is mainly about the RTLpath IR and related optimizations presented in Chapter 4.
- Monniaux, Boulmé, and Gourdin [107]† (June, 2023): A poster presented at the RISC-V summit Europe, in Barcelona. Abstract: https://riscv-europe.org/summit/2023/posters#lo-gourdin--formally-verified-advanced-optimizations-for-risc-v; Poster: https://riscv-europe.org/summit/2023/media/proceedings/posters/2023-06-08-L%C3%A9o-GOURDIN-poster.pdf.
- Gourdin [64]† (July, 2023): A ICOOOLPS ("International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems") paper about the co-design of an improved and combined version of the Lazy Code Motion and Lazy Strength-Reduction algorithms.
- Monniaux et al. [108]† (July, 2023): A TAP ("Tests And Proofs" conference) paper about our development method combining tests and formal proofs to extend the CompCert formally verified compiler.
- Gourdin et al. [65]† (October, 2023): A OOPSLA ("Object-Oriented Programming, Systems, Languages & Applications") paper published in the Proceedings of the ACM on Programming Languages (PACMPL) journal. This paper concerns the main topic of my thesis: a defensive simulation framework over the BTL IR.

1.4 CONTENTS OF THIS DOCUMENT

The document is divided into four parts. Part i sets the context of this work, and presents some preliminary contributions, the discussion of which will enable us to motivate the design choices behind Part ii. Subsequently, Part ii focuses on the formal verification aspect by presenting our new IR and validation framework. The transformation aspect, concerning optimization oracles, is covered in Part iii. Finally, Part iv reviews our experimental results and concludes.

Throughout this document, I have aimed to provide a comprehensive overview of the state of the art in translation validation and loop optimizations. I structured this overview across several chapters and sections, as detailed below:

- Chapters 2 & 3: a state of the art on translation validation, formally verified compilation, and in particular CompCert.
- Chapter 4: a state of the art on formally verified instruction scheduling within CompCert.
- Section 9.4 (in Part ii): complementary related work on CompCert and formal verification of loop optimizations.
- Sections 10.1 & 10.5.3 (in Part iii): complementary related work on redundancy elimination and strength-reduction optimizations.

Part I

SETTING & PRELIMINARY CONTRIBUTIONS

This thesis being part of a wide research context, the following part is necessary to not only position the contribution presented here, but also to make the whole document self-contained. The background is decomposed in three chapters:

- Chapter 2 on the background in formal validation and related work;
- Chapter 3 about the CompCert C verified compiler;
- Chapter 4 positioning this thesis w.r.t. previous works on scheduling and symbolic execution in CompCert.

2

FORMALLY VERIFIED DEFENSIVE PROGRAMMING (FVDP)

2.1 THE COQ PROOF ASSISTANT

Theorem provers are software that help in writing and verifying proofs within a formal logic. They are generally used both for purely mathematical purposes and for developing highly reliable software. Coq[1] is an interactive proof assistant supporting higher-order logic, dependent types, and proof automation; its formal logic is the *calculus of inductive constructions* (CIC) [37, 38, 118]. This formal logic is a pure functional language: functions are deterministic, without side effects (i.e. no mutable data), and there are no imperative loops (i.e. iteration is performed via recursion instead). CIC represents both programs and proofs as functions (using the Curry-Howard isomorphism). Coq developments are written in the Gallina specification language, which enables interactive constructions of CIC terms. In particular, Gallina features a *tactic* language, named Ltac, to let the user interactively produce proof terms. Gallina comes with many common tactics, but the tactic language can be extended as needed. A *kernel* then ensures that the generated proof terms are correct, that is well-typed terms of CIC. CIC acts as a voluntarily small part of the core language into which every high-level concept is translated in (to minimize the risk of error). This is an example of defensive design, as we will discuss in §2.3.

In the following, we abusively say "Coq code" to mean "Gallina code".

An important feature of the Coq proof assistant is extraction [97]: programs specified in the language can be extracted automatically to OCaml (or Haskell, JSON, Scheme), to then being compiled (and optimized) as verified libraries or whole programs. During this process, all the proof related information (theorems, constraints in dependent types) is removed, and only the computational part of the code is kept.

Several significant successes have been realized using Coq (which received the ACM Software System Award in 2013), both in mathematics and informatics. Notably, the proofs of the *four color theorem* (map coloring, in 2005) and the *Feit–Thompson theorem* (groups' classification, in 2012) were both verified in Coq. The *Univalent foundations* (a type-theoretic alternative to the set-theoretic foundations of mathematics), and the related *Homotopy Type Theory* (initiated by the Field medalist Vladimir Voevodsky) were also largely formalized within proof assistants; including Coq, but also Agda. The *CompCert* verified compiler—which also received the ACM Software System Award in 2021—is entirely proved in Coq; and the collection of verified software from the *DeepSpec*[2] project—which represents a major advance in formal verification—relies heavily on Coq.

2.2 TRANSLATION VALIDATION

In this dissertation, I study how translation validation—and in particular symbolic execution (SE)—is prone to help us when writing a certified compiler relying on non-trivial optimization algorithms. Translation validation [119, 129], and more generally *a posteriori verification*, is the idea of delegating a computation to an untrusted OCaml code—that we call "oracle"—and to only formally verify its *result* afterwards. By combining direct style proofs and a posteriori dynamic checks, it is possible to *a priori* prove the compiler correctness, as we do in CompCert. This section gives an overview of SE as a translation validation technique.

2.2.1 *A Classical Example*

The register allocator of Rideau and Leroy [123] was one of the first a posteriori validated algorithms in CompCert (the other one being the node enumeration of LTL nodes [92, §10.2]). Directly proving

[1] https://coq.inria.fr/
[2] https://deepspec.org/main

correct the heuristic would in fact be difficult, as register allocation reduces to a graph coloring problem, known to be NP-complete.

Fortunately, validating a posteriori the result is *much* easier: it involves checking some properties on an interference graph (the candidate coloring) returned by the untrusted oracle. If the result proposed by the oracle proves to be correct, the compiler will call the translation algorithm of the next pass; otherwise, the proven validator will detect the issue and stop the compilation.

The Coq code being extracted as OCaml, one simply has to declare the oracle in Coq as an *axiom* along with an extraction directive to specify the oracle's OCaml function. For instance, the declaration of the register allocator is defined as:

```
(* Declaring the oracle as an axiom in the result monad *)
Parameter regalloc: RTL.function → res LTL.function
(* Setting an extraction directive to specify the oracle's function *)
Extract Constant Allocation.regalloc ⇒ "Regalloc.regalloc"
```

Such a declaration works to call untrusted code from Coq, but it might allow some unsound reasoning over an OCaml external function that may behave non-deterministically (e.g. because of a hidden side effect). We discuss this risk in §2.4, and see how it can be mitigated by a proper monadic encapsulation.

2.2.2 *Using "Shadow" Fields to Combine Extracted and Handwritten OCaml Code*

One of the advantage of the extraction mechanism is that we can integrate in the Coq part (to be extracted) some information that will only be used by the untrusted part of the code (i.e. by the handwritten OCaml). For instance, let us consider a language whose instructions are of a Coq type "`inst`", and assume that several oracles need to store and read information from an analysis on each instruction. A solution is to wrap the type "`inst`" into a record, and to declare a second type, extracted directly to the *internal type* used in oracles as a parameter:

```
Parameter analysis_info: Set
Extract Constant analysis_info ⇒ "OraclesModule.oracles_type"
Record inst_wrap ≔ mk_iwrap {
  coq_inst: inst;
  oracles_stuff: analysis_info (* Shadow field *) }
```

With this structure, Coq does not know anything about the `analysis_info` type: we declare it as a parameter (i.e. an axiom) but we *cannot use it* for the reasoning. This technique enables embedding information that can only be used by the untrusted part of the code, but that can be transmitted along the Coq parts. If we have a Coq algorithm that transforms the code between two translation-validation steps using two different oracles, the information can be preserved seamlessly and without any risk. We name those types that directly target an oracle's internal type as "shadow" fields. Of course, they are not necessarily defined in records: we can also place them in inductive types, sum types, etc. Note that shadow fields here are the *dual* of "ghost" fields used in WhyML[3], Java Modeling Language[4] or ANSI C Specification Language[5]: instead of being only visible by the specification, shadow fields are only visible by the program (but not by the specification).

2.2.3 *Symbolic Execution*

Among the translation validation methods, *symbolic execution* [81, 130] is a generic approach to prove the correctness of a transformation. The principle is to write and *prove correct* a simulation test, asserting that the behaviors and semantics of the source program are preserved.

2.2.3.1 *An Introductory Example on Basic Blocks*

Symbolic execution works on program fragments, typically loop-free blocks of code, by first simulating the *source* fragment and the *target* (i.e. transformed) one, and second comparing the resulting

[3] See https://why3.lri.fr/doc/syntaxref.html#ghost-expressions.
[4] See https://www.openjml.org/tutorial/Ghost.
[5] See https://frama-c.com/download/acsl.pdf, Section 2.12, p72.

symbolic states. These states include a symbolic representation of memory and registers, being incrementally modified while *evaluating* instructions in the block. Furthermore, as informally stated in §1.1, the symbolic execution must ensure that the target code does not contain any additional potential trap w.r.t. to the source: inserting a new potentially trapping instruction in the code would be incorrect. This leads to a notion of precondition, where the precondition of the target block must be weaker (i.e. implied by) the source's one.

In the case of basic blocks (cf. §1.1.3.1), the symbolic state resulting from the SE of each block is a unique big symbolic term representing a *parallel assignment* of registers, as shown in Example 2.2.1 (I explain how SE behaves for larger blocks in §2.2.3.3).

Example 2.2.1 (Basic blocks simulation*)**.** Consider two basic blocks B_1 and B_2:
(B_1) $r_1 \coloneqq r_1 + r_2;\ r_3 \coloneqq \texttt{load}[m, r_1];\ r_3 \coloneqq r_1;\ r_1 \coloneqq r_1 + r_3$
(B_2) $r_3 \coloneqq r_1 + r_2;\ r_1 \coloneqq r_3 + r_3$
Both B_1 and B_2 lead to the same parallel assignment:

$$r_1 \coloneqq (r_1 + r_2) + (r_1 + r_2) \parallel r_3 \coloneqq r_1 + r_2$$

But, B_1 is preconditioned by "$\texttt{load}[m, r_1 + r_2]$ has not trapped", whereas the precondition of B_2 is trivially true. Hence, B_2 simulates B_1, but the converse is false. Above, the (empty) set of potentially trapping terms of the target is included in the set $\{\texttt{load}[m, r_1 + r_2]\}$ of such terms in the source.

Six, Boulmé, and Monniaux [134] encode such a precondition as a list of potentially trapping terms, hence relaxing the implication of preconditions as a list inclusion.

The above example is sufficiently expressive for reordering basic blocks. I explain along this dissertation how it can be extended to not only support larger blocks, but also to verify complex and even inter-block optimizations.

2.2.3.2 *Normalized Rewriting*

Necula [116] extended the technique with *normalized rewriting*, but without a formal proof of the validator (unlike the mechanically proved SE of Tristan and Leroy [142], see §4.1.3). Rewriting and normalizing expressions during the execution allows deducing equivalences between terms that were built differently. It reduces comparison (of symbolic values) modulo a set of equations to structural equalities [82].

Example 2.2.2 (Rewriting symbolic values)**.** Let us take two equivalent blocks B_1 and B_2:
(B_1) $r_1 \coloneqq 16;\ r_2 \coloneqq r_1 \times 4$ and (B_2) $r_1 \coloneqq 16;\ r_2 \coloneqq r_1 \ll 2$
Now, assume we have a rewriting rule stating that "$v \ll 2 \equiv v \times 4$"[6]. We choose to always apply this rule in the same direction (e.g. by replacing the left part with the right one), on the target block (the other way would also be possible). Then, values in the final states become syntactically equals: $r_2 \coloneqq 16 \times 4 \equiv r_2 \coloneqq 64$. The replacement of the multiplication by four with an equivalent left shift is thus correct.

Rewriting rules are only used to compute symbolic states in a canonical way—enabling syntactical equality—but they do not change the transformed code. With the above rule, the transformed code keeps the left shift instead of the multiplication; while in symbolic states, only the multiplication is present on both sides thanks to the rule. Proving correct the whole validator would thus require proving correct every rewriting rule beforehand.

2.2.3.3 *An Efficient Implementation*

There are two main sources of complexity with symbolic simulation techniques:

- **In the execution:** the size of code fragments (see §1.1.3.1), and notably the "width" of the CFG. Indeed, a naive trace partitioning [126] execution becomes exponential over the number of *internal joins* of the input block. If we consider basic blocks, superblocks, or extended blocks, it should not be a major problem (as experimented in this thesis), since they have, by definition, no internal joins.

*This introductory example is reused from [134, Example 4.3] and [135, Example 2.1]†.
[6] A left shift being often less costly than a multiplication by a power of two.

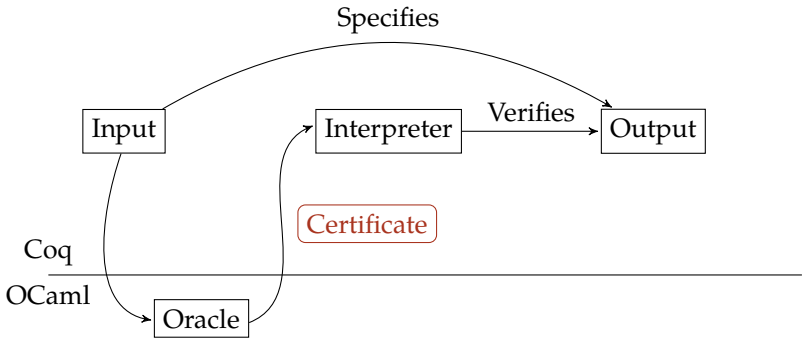


Figure 2.1: The "Certificate" Defensive Programming Style.

- **When comparing symbolic states:** the execution causes term duplication, as it was the case for "$r_1 + r_2$" in Example 2.2.1. More generally, the *structural comparison* of the nested symbolic values in states is exponential.

The second point above is challenging, because without an efficient comparison method, the verifier will not scale on large programs (for instance, the case of §4.1.3). Six, Boulmé, and Monniaux [134] solve this issue thanks to first, pointer equalities instead of structural equalities and second, verified hash-consing (see §2.4.2) that binds the two symbolic executions to the same pointers (i.e. term memoization). The hash-consing compares already memoized subterms using the OCaml pointer equality, so that the comparison remains *constant-time*. The formally verified hash-consing factory is implemented with the IMPURE monad library, described in §2.4.

2.3 THE PRINCIPLE OF DEFENSIVE PROGRAMMING

In short, FVDP provides a design style to reason about and decompose large, formally verified developments.

The name *formally verified defensive programming* (*FVDP*) was introduced by Boulmé [23] to suggest a high-level, yet precise view of the principle that consists in testing defensively the results of an untrusted and hidden computation to certify it. FVDP defines a set of *strategies* to embed oracles within a proof assistant such as Coq. Translation validation could be viewed as one of these strategies. With this method, errors in oracles that result in incorrect code are systematically turned into compilation failures (i.e. the aim of a certified compiler is to avoid generating a *wrong* output, and to abort the compilation in such a situation).
According to Boulmé [23], we distinguish three forms of certification:

1. The "autarkic" style, where there is no oracle, and with everything specified, proved, and computed on the Coq side;
2. A "certificate" approach (Figure 2.1), where the oracle may provide (or not, depending on the complexity of the defensive test) a *certificate* (a.k.a. *witness* or *hint*) to the Coq checker to *drive* the verification and make it less expensive algorithmically;
3. And a lightweight style for representing witnesses, inspired by the old LCF[7] prover, constraining oracles to build correct-by-construction results.

LCF style becomes interesting when the studied application must provide complex witnesses, as in the Verified Polyhedron Library (VPL) of Boulmé [23, §1.4.3].

Styles 2. and 3. above both introduce a representation for witnesses used in oracles. In this document, we will mainly focus on the second one: oracles yielding or not certificates to help the verification process. We choose to rely on this method because our witnesses are reasonably small, so there is no point in implementing an LCF style here.

The scheduling presented in the case-study of Chapter 4 performs the verification without any hint from the untrusted algorithm. In contrast, the BTL symbolic execution verifier proposed in this thesis leverages the use of certificates expressing invariants (recall §1.2.2) to aggregate local simulations into a global simulation.

Furthermore, the defensive approach is of great help to find out where oracles' bugs are located, and thus facilitates their development. Only two types of errors can lead to a verification failure:

[7] Stands for "Logic for Computable Functions"

either the oracle did something *wrong* (e.g. by yielding an incorrect certificate, or with an incorrect code transformation); or the verifier is too *weak* to assert the correctness of the transformation (e.g. the result is out of the supported class of transformations). The FVDP design also allows for simpler, more modular proofs, since many optimizations can be verified; even if an oracle completely changes internally, it will be transparent for the Coq side as long as it stays in the supported class. Of course, any translation validation method could follow this principle. In particular, a technique very similar to style 2. above was suggested by Glesner [61] where certificates help in reducing the search space for a validator which replays some analyses already performed by the untrusted optimization.

An extended discussion on certificate-based techniques is proposed in §2.5.2.

To conclude, the FVDP methodology emphasizes that *formally verified testing (with, e.g. SE) helps formally verified programming* [108]†.

2.4 IMPURE: A SAFE FOREIGN FUNCTION INTERFACE (FFI)

We saw in §2.2 different methods of translation validation. However, these methods should be used with caution, as a malicious use could lead to potential flaws in the verification process.

2.4.1 *The Risk of "Impurity"*

Abstracting OCaml functions in Coq is a challenging problem. The standard method, used notably for register allocation (cf. §2.2.1), is to declare the existence of a function of a certain type as an axiom, so that the latter will be replaced at extraction.

Let us define a pathologic example; we declare an oracle and prove a wrong lemma about it's *purity*. The Coq code would be:

```
Axiom oracle: nat → bool. Extract Constant oracle ⇒ "foo"
Lemma oracle_pure: ∀ n, oracle n = oracle n
 (* By congruence. *)
Qed
```

The implementation of `foo` being:

```
let foo =
  let b = ref false in
  fun (_:nat) -> (b:=not !b; !b)
```

Indeed, we see in the above example that we are hampered by the Coq constraint on purity of functions. The Coq *congruence* tells us that two calls to the same oracle, using the same parameters, are equals, but this is not the case in practice. In the `foo` function above, two successive calls will return two different values. Hence, hidden side effects may make OCaml functions appear as non-deterministic.

For a deeper discussion about the weaknesses of CompCert, see [106].

Thankfully, this is not a major issue as long as the purity of oracles is never assumed in the formal proof; for example, by deliberately calling the same oracle twice to enter an absurd case (this, of course, is never done in CompCert). Nevertheless, it is still interesting to *safely formalize* this approach. Boulmé [23]'s habilitation thesis proposes, in this objective, a Coq library to forbid formal reasoning about the *effect* of such functions, leaving only the possibility to reason about *non-deterministic results*. This library is based on a monad, also deeply detailed in Boulmé's habilitation thesis. In the latter document, several pitfalls of the "naive" axiom declaration are reported [23, §2.2.2, §2.5]. For example, one may accidentally rely on *implicit axioms*: if the axiom to be extracted has a return type $A : \{x : \mathbb{Z} \mid x < 5\}$, it will be extracted to a larger type $\mathbb{Z}$, that does not ensure the implicit requirement on A (i.e. x is lower than 5). As a result, it would be possible to extract the axiom as an OCaml function returning 7, thereby creating an absurd case. This is due to the OCaml typechecker, which is less expressive than the Coq typechecker. It is hence very important to check the extracted type w.r.t. the Coq one.

In practice, we continue to use the standard Coq foreign function interface (FFI) for optimization oracles, as long as they are invoked only once during the compilation process. We resort to the monadic method outlined above—and exemplified below—exclusively for sensitive external functions like the hash-consing factory or pointer equality tests.

2.4.2 *Motivation: FVDP of a Lightweight Hash-Consing Factory*

FVDP of hash-consing was first proposed by Boulmé and Vandendorpe [24], and reused in the postpass scheduling of Six, Boulmé, and Monniaux [134, §C.4.2].

We saw in §2.2.3 that SE requires comparing states (i.e. term trees) that may contain duplications, as in Example 2.2.1. Hash-consing is a technique to reduce the complexity of comparisons using *pointer equality* instead of *structural equality*. In practice, the constructors of inductive data types are memoized to share common subtrees, so that equal terms will be allocated to the same object in the memory, allowing pointer equality comparisons. The memoization is performed by replacing the original constructors of the type by *smart constructors* with an automatically generated memoization function from a hash-consing factory. This technique is inspired from Filliatre and Conchon [56], but our implementation features a simple formal property of correctness, written in Coq. The hash-consing factory is an untrusted OCaml oracle, whose results are *dynamically* checked during the formally verified symbolic execution. Actually, it is sufficient for our simulation test proof to ensure that the oracle's memoizing functions behave observationally like the identity.

Compared with the solutions of Braibant, Jourdan, and Monniaux [26], where hash-consing is implemented either directly in Coq or in the extracted OCaml code, our FVDP approach is weaker. Specifically, our pointer equality *implies* semantic (i.e. structural) equality, but is not equivalent to it. This weaker property is sufficient for our implementation, and allows for a lighter design.

Below, I only focus on pointer equality, because it is difficult to handle it correctly and conveniently in Coq. During my PhD, the IMPURE monad has been slightly extended in order to handle such equality more conveniently (see §7.2.4 for technical details about the instantiation of hash-consing in our SE engine).

2.4.3 *A Coq Model of OCaml Pointer Equality?*

Comparing hashed terms using pointer equality requires the use of an axiom (which will be extracted as the OCaml "==") for calling the operator. Exactly as in §2.4.1, if we (naively) declare the axiom below (e.g. aiming to compare Peano natural numbers; elements of the Coq type `nat`), we open the possibility of proving false properties.

```
Parameter phys_eq: nat → nat → bool (* only [nat] for simplicity .*)
Extract Constant phys_eq ⇒ "(==)"
```

First, let us observe that in OCaml, pointer equality is true only if applied to the same *physical* object: for instance, "`let n = (S 0) in n == n`" returns `true` while "`(S 0) == (S 0)`" returns `false`. One would like to pose the hypothesis:

```
Hypothesis phys_eq_axiom: ∀ x, phys_eq x x = true
```

However, `phys_eq_axiom` is too strong because each `x` above may correspond to a distinct pointer, allowing us to prove the wrong lemma below simply by substituting `y` by `x`.

```
Lemma wrong_property: ∀ x y, x = y → phys_eq x y <> false
 (* By congruence. *)
Qed
```

The counter-example "`(S 0) == (S 0) ≡ false`" illustrates why the lemma is wrong: structural equality does not imply pointer equality in OCaml. Conversely, substitution of variables in Coq propositions does not preserve memory allocation.

Even using *seemingly* weaker alternatives of `phys_eq_axiom`, for example:

```
Hypothesis phys_eq_axiom': ∀ x y, phys_eq x y = true → x=y
```

does not solve this issue, again because of this substitution power:

```
Theorem wrong_substitution x y: x=y → phys_eq x x = phys_eq x y
 (* By congruence. *)
Qed
```

As we have seen, Coq propositions cannot speak about pointers of OCaml executions, because this is not compatible with Coq congruence. Thus, we need to find another way to express that pointer equality has the power to establish some structural equalities.

2.4.4 *The May-Return Monad* [◇]

Fouilhe and Boulmé [59] proposed a general model of OCaml functions in Coq—a FFI—to solve reasoning issues of Sections 2.4.1 and 2.4.3. Their suggestion, embodied in the IMPURE library [23], is to use a may-return monad to forbid any reasoning on the effects of OCaml functions. This approach follows a weakest liberal precondition (WLP) style of reasoning, programmed with a special Coq tactic [23, §2.2.1]. Formally, each OCaml function of type "$A \to B$" is represented as a "relation" "$A \to ??B$" where "$??A$" abstracts $(A \to \mathrm{Prop})$ through "$\rightsquigarrow_A : ??A \to (A \to \mathrm{Prop})$" such that "$k \rightsquigarrow a$" means "$(k\ a)$" ("computation k *may return* a").

In other words, "$??A$" is the type of impure computations of type A. Moreover, since $??A$ is extracted like A, the monad does not cause a runtime overhead. The "$??$." monad only restricts Coq congruence w.r.t. usual functions, and allows modelling in Coq non-deterministic functions that may not terminate normally as "$A \to ??B$": the reasoning on the may-return operator "$\rightsquigarrow$" is limited to partial correctness.
Type "$??$." is bounded in a monad with the usual operators (and their axioms w.r.t. "$\rightsquigarrow$"):

BIND OPERATOR: extracted to "$\triangleright$" (i.e. the OCaml bind, sometimes also noted ">>=") with type "$??A \to (A \to ??B) \to ??B$" (to compose calls). Its correctness axiom states that "$\forall (A\ B : \mathrm{Type})\ (k_1 : ??A)\ (k_2 : A \to ??B)\ (b : B), (k_1 \triangleright k_2) \rightsquigarrow b \to \exists (a : A), k_1 \rightsquigarrow a \wedge (k_2\ a) \rightsquigarrow b$".

UNIT OPERATOR: "$\mathrm{RET} : A \to ??A$" (to enter the monad—i.e. lift a pure computation to an impure one). It must satisfy "$\forall (A : \mathrm{Type})\ (a\ b : A), (\mathrm{RET}\ a) \rightsquigarrow b \to a = b$".

ANNOTATION OPERATOR: "$\forall (A : \mathrm{Type})\ (k : ??A), ??\{a : A \mid k \rightsquigarrow a\}$" to annotate the returned type with the may-return relation. It allows embedding the "$\rightsquigarrow$" operator into dependent types.

EXIT OPERATOR: noted "$\mathtt{has\_returned}$" of type "$??A \to \mathtt{bool}$" to exit the monad on termination. This exit operator was added to IMPURE during my thesis. It enables exiting the monad to alleviate the Coq development, as exemplified in §7.2.4.1. It transforms a pure expression into an impure defensive test that either validates a property by terminating normally or fails by raising an exception. From the Coq point of view, the expression appears to return a Boolean, where a "true" value *guarantees the normal termination* of the impure test, but offers *no guarantee* for a "false" value. Hence, the "$\mathtt{has\_returned}$" operator only applies for computations whose result is Boolean. This operator must satisfy "$\forall (A : \mathrm{Type})\ (k : ??A), \mathtt{has\_returned}\ k = \mathtt{true} \to \exists r, k \rightsquigarrow r$". It is extracted in OCaml as "`(fun k -> k; true)`". Hence, "$\mathtt{has\_returned}$" reduces non-determinism to normal termination of the calculus, its result being always deterministic (it is overridden by "true"). We cannot write a Coq lemma falsified by this form of non-determinism: for all k, when the computation terminates, "$\mathtt{has\_returned}\ k$" always yields the same value.

A SAFE POINTER EQUALITY [◇] Now, we are able to model pointer equality in Coq without the pitfalls presented earlier, by encapsulating the declaration of the axiom in the monad. The below was proposed for the first time in [134].

```
Axiom phys_eq: ∀ {A}, A → A → ?? bool
Extract Inlined Constant phys_eq ⇒ "(==)"
Axiom phys_eq_correct: ∀ A (x y: A), phys_eq x y ⇝ true →  x=y
```

Here, "`phys_eq x y ⇝ b`" could be read as "if `b` is `true`, then it has existed an allocated object `o` such that `x=o=y`", since we can still prove the lemma below in Coq:

```
Lemma ok: ∀ x y, x = y →  phys_eq x x ⇝ true →  phys_eq x y ⇝ true
  (* By using the Coq substitution. *)
Qed
```

However, thanks to the monad, the wrong property below *cannot be proved* (except on empty types):

```
Lemma unprovable_wrong:
∀ x y, x = y → phys_eq x x ⇝ true → phys_eq x y ⇝ false → False
```

APPLYING THE SAME PRINCIPLE TO HASH-CONSING An OCaml oracle provides the untrusted factory with type "T → ??T". This factory is then a posteriori certified by enforcing that each returned term is structurally equal to its inputs: this is done by a *constant-time* check: if we have in input the inductive term $(c\ t_1\ ...\ t_n)$ (where c is the constructor and t_i the subterms) and an output term $(c'\ t'_1\ ...\ t'_n)$, we compare $c = c'$ (structurally) and $\forall i, t_i == t'_i$ (using pointer equality). This latter check works in practice because of the (non-formalized) invariant stating that all t_i are *already hash-consed* terms.

2.5 RELATED WORK IN TRANSLATION VALIDATION AND VERIFIED COMPILATION*

Since Pnueli, Siegel, and Singerman [119], translation validation has become a quite intensive research area.

The PhD dissertation of Clément [32, §8.1], which inspired me to fill this section, proposes a good and detailed overview of translation validation techniques up to 2022.

An important point when applying translation validation is to distinguish *structure-preserving* transformations from *structure-altering* ones [151]. Structure-preserving transformations can be proved from structure-directed simulations, whereas this is not necessarily the case for structure-altering ones. The former group includes optimizations such as CSE, dead code elimination (DCE), LICM, and even loop unrollings, since these still allow to define a CFG mapping between the source and the transformed program. Oppositely, the latter group of structure-altering transformations contains loop nest reorganizations [39] such as loop fusion, tiling, and interchange.

In this document, we focus on structure-preserving transformations. We apply symbolic execution (see Chapter 6) for optimizations that produce an identity CFG mapping (i.e. that only change the block content). Transformations which fold or unfold[8] the CFG by adding, removing, or duplicating nodes are validated using separated validators (see Chapter 8).

2.5.1 *Symbolic Execution*

When applied to semantic preservation, the goal of symbolic execution engines is to deduce a simulation relation between program fragments that match on synchronization points (e.g. blocks).

The primary purpose of symbolic execution in the seminal papers of King [81] and Samet [130] was for *testing*. Yet, for several years now, SE has also been used for verifying safety properties, especially for critical systems [35] (2001).

In CompCert, the first formally verified symbolic simulation test experienced was the one of Tristan and Leroy [142] for trace scheduling (more details in §4.1.3). Their framework was not using hash-consing, and was designed specifically for scheduling. A few years after, they also proposed a symbolic evaluation based framework for software pipelining [144]. However, these two validators were never integrated in mainline CompCert[9]. Then, Six, Boulmé, and Monniaux [134] proposed a similar but lighter verified validator for instruction scheduling.

Besides CompCert, several other uses of SE exist in the literature. Combined with model checking, SE was used to validate correctness of multithreaded Java programs and to generate input tests [80]. More recently, frameworks such as KLEE [28] have been proposed to not only generate tests, but also improve their coverage and find bugs.

Tristan, Govereau, and Morrisett [141] also suggested a (non-proven) symbolic evaluation translation validator working on value graphs. Their solution, applied directly on LLVM's assembly files, generates gated-SSA representations and uses symbolic evaluation with hash-consing and normalization. Value graphs are then compared syntactically (i.e. using pointers from hash tables).

2.5.2 *Other Translation Validation Approaches*

SPECIALIZED VALIDATORS One of the most common technique of a posteriori verification, well studied in CompCert, it to use a very specific validator that targets a single transformation. For instance, this method was used for the CompCert register allocation of Rideau and Leroy [123], for lazy code motion (a data-flow partial redundancy elimination optimization [84]) in the PhD of

*Some text of this section is reused from [65]†.

[8] "Fold or unfold" means that it is still possible to produce a structure preserving node mapping between both CFGs.

[9] Source code for their extensions at https://github.com/jtristan/CompCert-Extensions.

Tristan [140, 143], and for CSE & LICM in the work of Monniaux and Six [109]. These validators use formally verified analyses to replay some part of the optimization algorithm. As of today, only the register allocation was integrated into the mainline CompCert version.

PROOF PRODUCING FRAMEWORKS With their *credible compiler*, Rinard and Marino [124] propose an alternative solution to a priori proven tools like CompCert. Instead of having a total correctness proof stating that the compiler always preserves the source program semantics, their compiler "*produces a proof that it has operated correctly on the current program*." The authors describe their technique as orthogonal to proof-carrying code [115], in the sense that it proves a correspondence between two programs rather than a property on a single program. An explicit proof written in an ad-hoc logic is produced in output of their credible compiler, containing two types of invariants that relate both programs. These invariants are closely related to those I present in Chapter 6, and we consider this simulation approach as a foundation for our work. This credible compilation method is able to validate multiple optimizations, notably loop unrolling and dead code elimination. Note that dynamic (a posteriori) validation by SE is a form of proof generation: the proven SE engine yields symbolic states which, when equivalent, are evidence of semantic preservation.

Extending the idea of credible compilation, Kang et al. [77] have proposed a variant of formally verified translation validation, called "*Verified Credible Compilation*" for LLVM (a.k.a. CreLLVM). They validate the results of two existing optimizations of LLVM—register promotion and global value numbering (GVN)—with a dedicated oracle that generates proofs in a *relational Hoare logic* (inspired by Benton [17]), itself formalized in Coq. Their tool helped to find several new miscompilation bugs in these optimizations. However, it remains unclear what guarantees are provided to final users of the whole compiler.

INVARIANT TRANSLATION Another interesting approach is suggested in [125], in which the author uses symbolic transfer functions to match the semantic of C and assembly programs (in order to validate the entire compilation). The result is a translation validator based on invariant translation (i.e. instead of proving semantic preservation, only some invariants are proved to be preserved). The method is applied to both translation and optimization passes. Although Rival [125]'s solution is capable of proving safety properties on the program, and is well formalized, the checker itself is not mechanically proven. In addition, since the author relies on a first-order solver fed with abstract interpretation, it might (fortunately, it seems to rarely happen) produce false alarms. This designates a situation where the validator rejects a semantically correct transformation. False alarms often arise when the applied transformation is too complex to be discharged by the checker (see §9.4).

PRIMITIVES WITH SOUNDNESS CONDITIONS Kanade, Sanyal, and Khedker [76] implemented a trusted simulation test (in PVS[10]) for validating optimizations. By viewing transformations as sequences of predefined transformation primitives, they reduce the semantic preservation problem to soundness conditions to check on each primitive. Their framework validates multiple structure-preserving optimizations among which we find CSE, LCM, LICM, and DCE. It works on a three address code CFG. Primitives are used to decompose and apply the optimization (by mimicking some of its analyses) on the program, to validate soundness conditions. These conditions must themselves be proved sufficient to ensure semantic preservation (but, of course, we expect their—small step—proofs to be simpler than a direct proof of the optimization algorithm).

CROSS-PRODUCT As noticed by Zaks and Pnueli [148], one can validate transformations by computing the cross-product of the source and target programs, so that the equivalence check is reduced to analysis of a single, cross program. The paper presents a framework "CoVaC", applied to LLVM. It is implemented without a mechanical proof of correctness, in C++, and supports intra-procedural structure-preserving optimizations (e.g. constant folding, CSE, DCE, etc.) Inferring a bisimulation between both programs with this approach still requires finding a good alignment between them: a good product program should help in proving equivalence [31].

[10] The Prototype Verification System (PVS) is an automatic theorem prover, see https://pvs.csl.sri.com/.

PARAMETRIZED EQUIVALENCE CHECKING A generalization of translation validation that bridges the gap between a posteriori and a priori proven transformations is the parametrized equivalence checking (PEC) tool of Kundu, Tatlock, and Lerner [87]. They made the observation that "*we can reason about* [*symbolic, resulting from an interpreter*] *state equality even if we don't know what the program fragments are*". This means we can, by using parametrized programs (a.k.a. partially specified programs, a sort of template featuring symbolic properties and meta-variables) prove the correctness of classes of optimizations before running them. Optimizations are represented as rewrite rules in a domain specific language (DSL), and are proven by building and ensuring a correlation between two parametrized CFGs. The latter validation is partially automated by calls to the Simplify[48] automatic theorem prover. Thanks to this framework, they succeed in validating software pipelining, and multiple loop transformations. In their following paper, Tatlock and Lerner [138] designed an extensible CompCert—named "XCert"—applying this method, but their validator was not formally verified, hence significantly augmenting the trusted computing base.

GENERAL-PURPOSE PREDICTABLE VALIDATORS Tate et al. [137] generalized the notion of *symbolic value* with *e-graphs* (or expression graphs): such an e-graph represents the contents of a single variable after any arbitrary computations, even including loops. This enables reasoning on loop transformations only by rewriting these e-graphs (e.g. without explicit invariant inference). Moreover, in order to "*simultaneously explore all possible sequences of optimizations*", they applied a saturation technique over their e-graphs. Noticing that saturation does not scale well on large programs, Tristan, Govereau, and Morrisett [141] experimented with normalized rewriting instead, arguing it is sufficient for translation validation. Indeed, they succeeded to validate many existing LLVM optimizations, without instrumentation nor hints from these transformations. However, they acknowledged that their translation validator is algorithmically complex (and thus probably difficult to formally verify). Moreover, they did not attempt to be sound w.r.t. undefined or diverging behaviors, whereas these cases are often complex to handle in CompCert correctness proofs.

WITNESSING COMPILATION The idea of using certificates/witnesses originally comes from proof-carrying code [115], a technique where an untrusted algorithm was expected to supply a safety proof attesting the correctness of its result w.r.t. some policy. Later, Glesner [61] applied this idea to translation validation, and Barthe and Kunz [15] showed how abstract interpretation was helpful to study certificate generation and translation. Then, Namjoshi and Zuck [114] extended the approach by representing witnesses as invariants linking the source and target program fragments. They notably show how to define invariants for several optimizations as sets of assertions to be checked with a satisfiability modulo theories (SMT) solver. These certificates are relatively close to our gluing invariants (introduced in §1.2.2), and are designed to be propagated through a stuttering simulation (see §3.2.2) witness.

This approach might work if we had a formally verified SMT solver, or at least a formally proven verifier of certificates generated by an SMT solver (such as SMTCoq [11]). Nonetheless, that would be a hugely laborious task and additionally, there lies the issue of the complexity of SMT solvers, which sometimes struggle[12].

LANGUAGE-INDEPENDENT VALIDATION Recently, Kasampalis et al. [78] proposed a generalization of the Pnueli, Siegel, and Singerman [119]'s approach using a validator parametric to the input and output languages semantics. In addition, their solution is transformation-independent thanks to *verification conditions* (i.e. synchronization points) that are provided by an external component integrated to the compiler (here LLVM). Their implementation successfully validates the LLVM's instruction selection applied while translating code from LLVM intermediate representation to x86 Asm. The formalization of their validator relates programs with "cut"-bisimulations. Intuitively, the principle is to generalize the notion of bisimulation by only focusing on a set of "cut" states that are sufficient to witness all relevant transformations and prove equivalence. Their checker relies on a SMT solver to validate transformations, and on verification conditions as hints that help constructing the simulation relation.

BOUNDED AND AUTOMATED VALIDATION The "Alive2" framework, developed by Lopes et al. [100], aims at automatically validating intra-procedural transformations in LLVM. It tries to find a refinement relation between two LLVM IR functions by the mean of a SMT solver. The tool was able to discover several bugs in LLVM. As stated by the authors, one of the primary objective was to avoid false alarms due to solver failures. Hence, they use a *bounded* translation validation, that consists in only unrolling loops up to a certain threshold and to limit, for instance, the memory consumption or the execution time of the solver. Consequently, their tool will miss refinement failures that goes beyond the specified bound. This automated technique appears to be a nice solution to discover optimization bugs, but may not be suitable for a general-purpose certified compiler.

CONCLUSION: A SPECTRUM OF TRANSLATION VALIDATION APPROACHES The register allocator in [123] is undoubtedly one of the major success of specialized translation validation. In contrast, the—also successful—work of Sewell, Myreen, and Klein [131] attempts to automatically match the C source and object code of the seL4 kernel (itself proved correct w.r.t. a high-level specification); the resulting verification conditions are discharged by a SMT solver. These two projects had very different constraints.

The seL4 validation team had to work with an existing compiler, which was not to be modified; but they could write the software to be compiled in a certain way that helped with the "matching", and they could tune per-module optimization options if needed. Their scheme is unlikely to work with other programs, or even with other compiler versions, unless these programs or the matching scheme are manually modified[11]. In contrast, CompCert was (informally) expected to compile arbitrary source programs without failure[12]; but code transformations and validators were designed together. In such a context, it is possible to have the code transformation leave hints to the validator. The validator is then likely to be more robust (it need not guess how source and target match), simpler, and to perform fewer computations.

However, according to Leroy [94], special-purpose translation validation is not a "*silver bullet*" either. Indeed, developing specific validators is tedious and expensive: they should be formally proved yet reasonably efficient, characteristics that may be contradictory. Moreover, between ultra-specialized validators and fully general ones, there is a continuum that remains to be systematically explored.

2.5.3 *Verified Compilers*

The CompCert project (described in Chapter 3) is not the only formally verified compiler, but, as stated in introduction, is certainly the most advanced one. It is a C compiler, available both freely for research purposes[13] and as a commercial product[14].

In practice, CompCert is often combined with other certified software such as the Astrée certified static analyzer [40]; and is integrated in many larger projects such as the Verified Software Toolchain[15]. A lot of researchers have taken CompCert as their object of study, which explains the large number of existing forks and extensions.

For example, CompCert-SSA [13, 45, 46] is an SSA formalization with some optimizations. Among them, we find *copy* (*moves*) *propagation*, and *global value numbering*[16]. This interesting work reveals how difficult it is to mechanically formalize the SSA translation algorithms (returning from SSA without introducing too many redundancies is, according to Demange and Retana [47, §1], "*a notoriously difficult problem*"). Their framework makes use of *a posteriori* verification for the "de-SSA" pass [47, §5], and this seems to allow a less complex proof than with a direct verification.

Furthermore, it is common for safety-critical systems and especially control software to be specified using block-diagrams or state machine design tools, such as Lustre [29]. The latter is a synchronous

[11] According to [131, §4.2], the translation validation of seL4 is very unstable w.r.t. the version of GCC.

[12] This is another argument against general-purpose translation validation based on SMT-solving for compilation of many—different—and evolving code bases: SMT solvers tend to be brittle, changes in solver version or minor changes in the source program may result in the solver timing out on validation problems that it could previously discharge.

[13] https://github.com/AbsInt/CompCert

[14] https://www.absint.com/compcert/

[15] https://vst.cs.princeton.edu/

[16] An SSA-based transformation similar (but not equivalent) to CSE for redundancy elimination.

data-flow programming language which served as the core language of the SCADE[17] industrial tool, used to design critical real-time reactive systems. SCADE (and Lustre) aims at specifying programs that are then compiled into executable imperative or object-oriented languages (e.g. C or Ada). Given their role in critical applications, it was of primary importance to obtain a formally verified compilation toolchain for such languages. The Vélus[18] [25, 27] compiler addressed this need by providing a mechanically proven translation of Lustre into Clight, one of the first IR of the CompCert's pipeline (see Figure 3.1). Therefore, by combining Vélus with CompCert (or with Chamois-CompCert), we have a complete compilation chain for the development of real-time synchronous embedded code. In this way, our Chamois-CompCert optimizations can be connected to the Vélus output.
See also Section 3.3.3 for related works about improvements of the CompCert memory model.

Besides CompCert, Vellvm [149] and CakeML [86] are two other compilers, formally verified with an interactive proof assistant (respectively Coq and HOL4). To our knowledge, none of them attempts to leverage translation validation as we do. They do not integrate the kind of formally verified optimizations that we support.

Another—albeit much more specific—formally verified framework is Jasmin [5]. The tool, made of both the Jasmin programming language and its compiler, aims at facilitating the development of performant and certified cryptographic software. The Jasmin compiler currently only targets x86/64 platforms. It is verified in Coq, and relies on the EasyCrypt [14] framework to prove security properties about programs. The Jasmin register allocation passes are translation validated with a Coq checker parametric to the allowed renaming class. This generic approach enables to validate not only register allocation, but also post unrolling renaming, stack sharing, and a part of register-array expansion (i.e. replacing array accesses with variable accesses). Albeit their compiler can perform DCE, tunneling, and constant propagation, it does not support intricate intra-procedural optimizations such as code motion or strength-reduction.

[17] Stands for "Safety Critical Application Development Environment"; see https://www.ansys.com/products/embedded-software/ansys-scade-suite and https://www.systerel.fr/expertises/methodes-formelles/scade/.

[18] https://velus.inria.fr/

3

THE COMPCERT VERIFIED COMPILER

3.1 ARCHITECTURE OF COMPCERT

Compared to "mainstream" compilers (e.g. GCC & LLVM), CompCert features many intermediate representations (IRs) (8 in the mainline version). Each representation is designed to facilitate the proof of a specific transformation applied to the source program. An overview of this architecture is given in Figure 3.1. The "CompCert C" IR handles most of the C99 [74] and C11 [73] standards of the C language with only a few exceptions[1], and supports multiple target backends. This subset of C is obtained through parsing (with an LR(1) formally verified parser generated with the "Menhir" tool) and annotating the source[2].

Six [133] added a CompCert backend for the Kalray KVX core.

The software is mainly written in the Coq proof assistant (cf. §2.1), except for untrusted oracles used for translation validation, and for some expansion mechanisms that remain in the trusted computing base (see §3.5). Coq parts are then *extracted* into executable OCaml code. The first pass, translating to Clight, simplifies CompCert C expressions by removing side effects and choosing an evaluation order. From Clight to C#minor, control structures are simplified, and type dependent computations eliminated. The next step, leading to Cminor, allocates some (local) variables in the stack, and simplifies switch/case patterns. The pass from Cminor to CminorSel performs instruction selection, and hence enters the middle-end part of the compiler. Most optimizations are applied at the register transfer language (RTL) level (see §3.4), on a single-instruction CFG. Register allocation is then applied, and the code is translated in location transfer language (LTL), with a basic blocks (cf. §1.1.3.1) structure; the only transformation at this level is branch tunneling (branch optimization). The LTL to Linear translation lays out the CFG nodes in a linear order by inserting explicit control-flow instructions. Stack frames are built in the Linear to Mach pass, before emitting the final assembly code in Asm. Note that CompCert is not a complete toolchain: it relies on an external C preprocessor, and the final IR, Asm, is compiled and linked into binary code using GNU-C tools.

I detailed the above information about the general architecture to highlight the special nature of CompCert w.r.t. usual compilers such as LLVM. Indeed, in the LLVM terminology, the backend starts with the instruction selection pass, and includes register allocation. The middle-end, in contrast, is mostly independent of the architecture and uses a low-level representation (a CFG in static single assignment form); while the frontend is close to the source language, and does not apply advanced

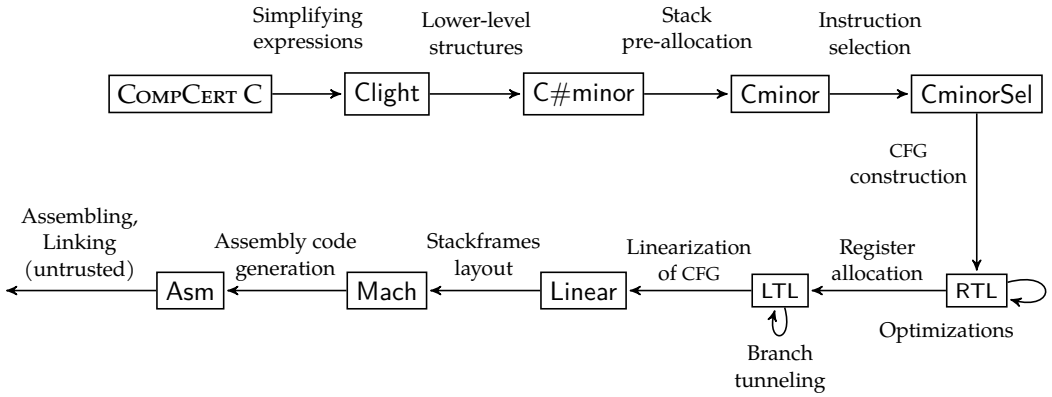


Figure 3.1: The Architecture of CompCert.

[1] Notably, variable-length arrays and `longjmp`/`setjmp` are not supported in the formal semantics (they can still be used, but without any formal guarantee).

[2] For more information about the structure of CompCert, please refer to https://compcert.org/man/manual001.html#sec4.

optimizations. Hence, there is no real middle-end in CompCert, since instruction selection ends the frontend part by using architecture specific patterns. On the other hand, CompCert comes with a large parametrized part, between CminorSel and Mach, which contains most of its optimizations. The latter can therefore be viewed as a kind of middle-end. After the Mach stage, the code representation becomes truly specific, and is considered as the backend. Therefore, our terminology here differs from the standard definition of frontend, middle-end, and backend, used in common compilers.

The decision to place CompCert's instruction selection at the Cminor level (i.e. early), is due to the necessity for a structured code. Broadly speaking, this representation facilitates the reasoning on selection rules that replace one pattern of structured code with another. Later (see Sections 7.6.1 and 10.2.4.2), we will discuss how our validation approach enables delaying it at the RTL level.

My contributions make use of the RTL (as a basis for our new IR), Mach, and Asm (for postpass scheduling, peephole, and code expansions) languages.

3.2 CORRECTION AND SIMULATION PROOFS

The goal of a correction theorem for a certified compiler is to prove, roughly speaking, that given a source program S and a compiled program T (for target), the compilation process preserves the semantics of S through all passes.

3.2.1 *Formalism of Program Behaviors*

The proven property is based on the behaviors of programs, which are classified into three categories: convergence and divergence (whether the program terminates or not), and *undefined* behaviors. The behavior—which emits an observable trace (i.e. from system calls or input/output operations)—is abstracted as B. Here, termination means that we obtain a compiled program *and* a finite trace; while a divergent program may have an infinite trace. Programs with undefined behaviors in the sense of the C standard (crashing operation, array access out of bounds, etc.) are considered as failing (Xavier Leroy describes them as "going wrong" in [93]), but their trace remains finite.

CompCert C (i.e. the input language, cf. Figure 3.1) has a non-deterministic semantics[3] inspired by the C standard. In the compiler chain, it is *made deterministic* just afterwards for having simpler proofs. Hence, all intermediate languages considered in this thesis are deterministic. For the sake of simplicity, we restrict ourselves to present CompCert correctness only for *deterministic languages*.

We denote $S \Downarrow B$ to state that we can observe behavior B when executing S (reusing the notations of [93, §2]). Assuming deterministic source and target languages, and a deterministic execution environment, program S (and the same applies for T) has only a single behavior B such that $S \Downarrow B$. Under these conditions, the semantic preservation property of CompCert tells us that if the observable behavior B of the source program is not undefined, then the compiled program T exhibits the same behavior. This corresponds to the safe forward simulation property: $\forall B \notin \mathit{Wrong}, S \Downarrow B \implies T \Downarrow B$.

Leroy [93] also mentions that since memory and type safety are preserved, the absence of undefined behaviors in the source implies the absence of undefined behaviors in the compiled program.

If we formalize the specification of S according to its observable behavior B as predicate $S \models \mathit{Spec}(B)$ (reads "S satisfies specification with behavior B"), we have another correctness property: $S \models \mathit{Spec}(B) \implies T \models \mathit{Spec}(B)$ (see [93, §2.1]).

Finally, the formal proof of correctness of the compiler can be thought as (for a single monolithic program) [93, §2.4]:

$$\forall S, T, B \notin \mathit{Wrong}, \mathrm{Comp}(S) = \mathrm{OK}(T) \wedge S \Downarrow B \implies T \Downarrow B$$

The above theorem can be thought of as: "If programs S and T, and behavior B are devoid of "going wrong" behaviors, and if the compilation of program S led to program T, then the unique behavior B of S is also the behavior of T.

For more information about correctness properties, the reader can refer to the documentation at [◇] (main proof) and [◇] (complements).

[3] Compiling such non-deterministic languages requires a more general notion of simulation than the one presented below, known as backward simulation [92, §2.1].

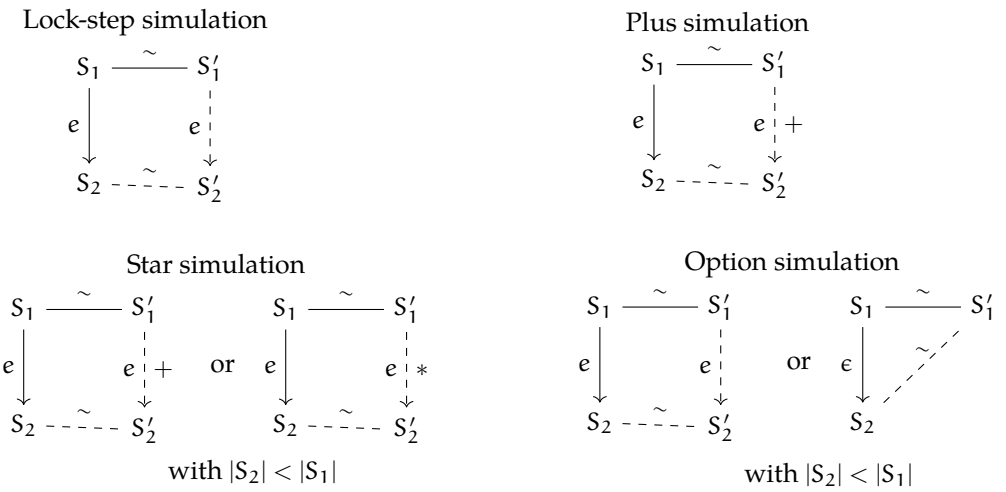


Figure 3.2: Simulation Diagrams Used in CompCert*.
Transitions are annotated with e when an observable event is emitted, and with ϵ otherwise.

3.2.2 *Simulation Schemes*

From the notion of semantic preservation, one can define a compiler *Comp* as a total function returning either the compiled code if everything is fine, or an error otherwise (i.e. using the error monad [93, §2.2.1]). Thanks to the *transitivity* of the semantic preservation properties, and by monadic composition, one can easily compose compilation passes, so that each pass can be proved independently.

Considering two languages and two programs P_1 and P_2 in these languages, the CompCert approach is to define a "match" relation—noted "$\sim$"—between states of both languages (I instantiate such a relation with my new intermediate representation in §6.1.1). Since the target assembly languages are deterministic, the correctness of CompCert's backend passes reduces to a *forward simulation*, that is obtained in three steps:

1. Relating initial states: S_1 from P_1 must simulate S'_1 from P_2 (i.e. $S_1 \sim S'_1$);
2. Relating final states in the same fashion as for initial states;
3. Extending the relation to transitions between states of P_1 and of P_2.

We talk about "stuttering" in a simulation when one program is not moving while the other is.

There are multiple ways of instantiating the third step above, known as the various simulation schemes of CompCert, and schematized in Figure 3.2. Letter e on the figure's arrows represents an optional observable event[4] emitted by the transition. When this event is absent, we note ϵ instead. The "lock-step" diagram (top-left) is the most trivial one: it stipulates that for each step in the source program, the target program, which emits exactly the same possible event (i.e. either the same observational event or the same absence of event) also takes a step forward. When P_2 contains more steps than P_1, the solution is to use the "plus" simulation, where the target can take several steps while the source takes a single step. The "star" simulation (a weaker version of "plus") allows programs to stutter, as long as all sub-sequences of successive stuttering steps are *finite*, and that the simulated steps do not emit observable events. The same condition applies to the "option"; the latter is a stronger form of "star", where a step is either synchronized (as in "lock-step") or optional (bottom-right of Figure 3.2). To ensure this finiteness condition, the "star" and "option" simulations require a well-founded ordering "$<$" over a measure between states of the source language. In Figure 3.2 bottom, for both schemes, the measure must decrease for each source step from S_1 to S_2. This constraint enforces the absence of any infinite stuttering sequence on the target side.

*Those diagrams are reused from [92, Figure 4].

[4]The CompCert's Coq code generally denotes events with a variable t (for *trace*) instead of e, but it uses letter t for both traces of events and single events. In this document, e always denotes a trace of *at most one* event.

3.3 COMPCERT INTERNALS

In this section, I give an overview of the internal representations used in the compiler concerning the memory, types, and values.

3.3.1 *Values and Operations*

The CompCert compiler defines a common type of values used in all intermediate representations; a value can be either: (i) a machine integer (32- or 64-bit); (ii) a floating-point number (also in both word sizes); (iii) a pointer, as a pair of a memory address (pointing to a memory block) and an offset inside this block; or (iv) the `Vundef` value. The latter denotes an arbitrary, *not observable* value: observing a `Vundef` (e.g. in an uninitialized variable) causes an undefined behavior. Albeit the fact that most languages in CompCert are deterministic, `Vundef` expresses *a form* of non-determinism: it may be simulated by any value (e.g. with a "less defined" relation where `Vundef` is poisoning). Values are simply implemented with the below sum-type:

E.g. if `x` is uninitialized, then `y = x + 1` does not cause an undefined behavior, but `printf("%d",x)` does.

```
Inductive val: Type :=
  | Vundef: val
  | Vint: int → val
  | Vlong: int64 → val
  | Vfloat: float → val
  | Vsingle: float32 → val
  | Vptr: block → ptrofs → val
```

The above type is equipped with the usual arithmetic operations, and a decidable equality.

3.3.2 *Register Sets*

In CompCert, "vectors" of registers—a.k.a. register *states*—are represented by a pair. Their type is noted "$\texttt{regset} \triangleq (v * \sigma)$", where σ is a finite map from registers to values and v a default initial value[5] (in other words, this representation internalizes that a register state has only a finite set of registers with distinct values). This representation is neither extensional (i.e. Leibniz equality distinguishes states that are extensionally equivalent) nor canonical. Indeed, associating v to register r in σ is extensionally *equivalent* to remove r from σ, but *produces two distinct states*. Hence, the identity over register states is often not expressive enough and extensional equality is needed instead.

3.3.3 *Memory*

The CompCert memory model [95, 96] is relatively basic: a collection of *disjoint blocks* in which accesses are performed with an *offset*, so that a pointer value is defined as a pair "block + offset". Each address defined as such is then tied to a permission policy. Transformations of a memory state are made through four operations: load (reading a chunk), store (writing a chunk), alloc (allocating a new memory block), and free (invalidating a memory block). Read and write operations operate on a specified quantity, known as the *memory chunk*, that contains information about the type and size of data being addressed. Five types of permissions are supported: (i) Freeable: for exclusive access, allowing all operations; (ii) Writable: allows for loads, stores, and pointer comparisons, but disallows freeing; (iii) Readable: only allows for loads and comparisons; (iv) Non-empty: a valid state only allowing comparisons; and (v) Empty: invalid (not allocated or freed), no operation allowed.

As it is, the CompCert memory model only uses bytes as its unit of size. This means we cannot define a single bit inside a byte (the entire byte's value remains at `Vundef`), thereby limiting the reasoning on bit fields[6]. Furthermore, because pointers are seen as abstract values "block + offset", we cannot completely manipulate pointers as if they were integers. Models of pointers as integers have already been explored in the literature [18–20].

[5] In practice, the default value of register sets is `Vundef`.
[6] See https://github.com/AbsInt/CompCert/issues/418.

i	$::=$	$\texttt{Inop}(pc_{\text{succ}})$	no-operation
	$\mid$	$\texttt{Iop}(op, \overrightarrow{reg_{\text{arg}}}, reg_{\text{dst}}, pc_{\text{succ}})$	normal operation
	$\mid$	$\texttt{Iload}(trap, chk, addr, \overrightarrow{reg_{\text{arg}}}, reg_{\text{dst}}, pc_{\text{succ}})$	memory load (trapping or not)
	$\mid$	$\texttt{Istore}(chk, addr, \overrightarrow{reg_{\text{arg}}}, reg_{\text{src}}, pc_{\text{succ}})$	memory store
	$\mid$	$\texttt{Icall}(sig, (reg \mid id), \overrightarrow{reg_{\text{arg}}}, reg_{\text{dst}}, pc_{\text{succ}})$	function call
	$\mid$	$\texttt{Itailcall}(sig, (reg \mid id), \overrightarrow{reg_{\text{arg}}})$	function call (not returning)
	$\mid$	$\texttt{Ibuiltin}(ef, \overrightarrow{reg_{\text{bargs}}}, reg_{\text{bdst}}, pc_{\text{succ}})$	compiler built-in function call
	$\mid$	$\texttt{Icond}(cond, \overrightarrow{reg_{\text{arg}}}, pc_{\text{ifso}}, pc_{\text{ifnot}})$	conditional branch
	$\mid$	$\texttt{Ijumptable}(reg_{\text{arg}}, \overrightarrow{pc_{\text{succ}}})$	"switch" jump (multiple successors)
	$\mid$	$\texttt{Ireturn}(\epsilon \mid reg)$	function return
cfg	$=$	$(pc \mapsto i)$	RTL graph as a map
f_{rtl}	$=$	$(sig, \overrightarrow{reg_{\text{arg}}}, ssize, cfg, pc_{\text{entry}})$	RTL function

Figure 3.3: Syntax of the RTL IR.

In accordance with the C standard [73, §6.5.8p5], a comparison of pointers allocated in two different blocks fails in CompCert.

3.4 THE REGISTER TRANSFER LANGUAGE INTERMEDIATE REPRESENTATION

The RTL IR is the last before register allocation, and features an unbounded number of available registers, represented by positive integers. This makes RTL convenient for "middle-end" optimizations, because they can easily introduce fresh (pseudo-)registers for storing intermediate results.

The code is cut into functions, where each function has its own set of pseudo-registers, and is organized as a control-flow graph (i.e. here a finite map): a node of the graph points to a specific instruction, and instructions syntactically contain the list of their successors nodes. The syntax of RTL is detailed in Figure 3.3; one of its important property is its modularity: the high-level syntax of the language is independent of the target architecture, but some constructors (e.g. the operation op in `Iop`, the condition type $cond$, addressing modes $addr$, etc.) are parametrized by the type of *operations* of the target. In the new IR that I present in Part ii, most of the various parameters for loads, stores, calls, and other instructions are reused; I thus skip their decomposition in the current section. A RTL function groups the signature, its parameters (a vector of registers), the stack size $ssize$, the CFG, and the entry point node number.

3.4.1 *Semantics*

As many languages in CompCert, RTL features a state semantics: there are different types of states that represent a specific status and point in the program execution. The semantics is thus defined as an inductive predicate specifying—under a global environment of symbols—the state transitions from an initial state to a final one, for each type of instruction. Those transitions may emit an execution trace (for built-ins and external calls). The semantics is relatively simple (less than five hundred lines of Coq specifications) [◇].

The three types of RTL states are represented in Figure 3.4. Each of them contains an abstract call stack as a list of frames, themselves encoded in a "Stackframe" type for still active function calls. A "Stackframe" contains the result register, the calling function along with its stack pointer, program counter, and register set. In addition to the call stack, states also encode the following information:

- The state "State" encodes the current function, stack pointer, program counter, register set, and memory state. As sketched in Figure 3.4, this kind of state is for basic transitions that describe a step within an internal function;

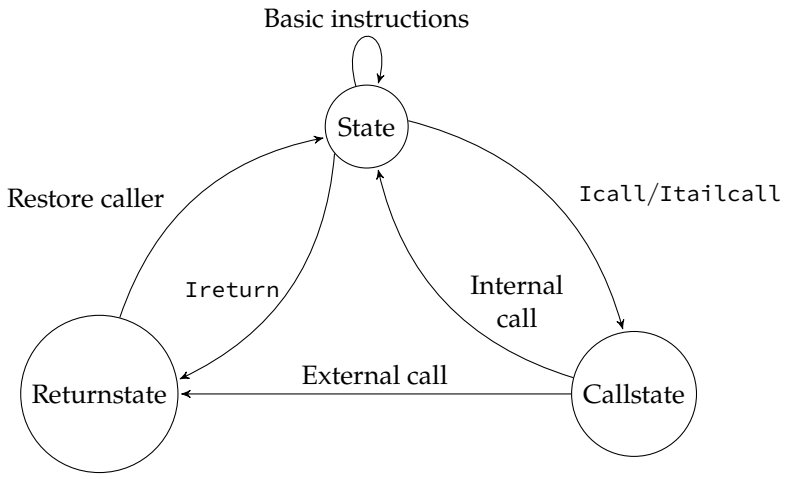


Figure 3.4: State Semantics of RTL.

- When a function is called, the program transitions from the normal "State" to an intermediate "Callstate"; the latter includes the function to call, its arguments (as a list of values), a memory state and returns the new program counter. The next step is then either a "State" changing the context to enter the internal function called, or a "Returnstate" in the case of an external function;
- "Returnstate" is an intermediate step before the restoration of the caller's state. It is reached either from an external function call, or when executing an `Ireturn` instruction of an internal function.

A RTL program is then defined using a small-step semantics from an initial state to a final one. The initial state is always a "Callstate" (the main function invocation), while the final state a "Returnstate" (the end point of the program).

3.4.2 *Limitations*

RTL registers are all seen as equivalent, i.e. there are no specific registers for flags, stack pointer, program counter, etc. Note that this model is quite limiting. First, it prevents from defining a semantic on special registers. A clever propagation (up to the Asm level) of the information via the operation constructors can avoid the problem in some cases, but this trick also undergoes some constraints: we are not able, for instance, to model the stack pointer in the semantic of a given RTL operation, because the layout of stack frames is not yet defined at this level (as briefly described in §3.4.1, stack accesses are only handled in a very abstract way). Second, the model is designed to assign one (and only one) destination register per operation: this is not really a problem for some backends (like in RISC-V, where each implemented instruction is only assigning a single register); but this becomes restrictive for some others, such as AArch64, where we cannot model, for instance, "arithmetic-comparison" instructions[7]. Still on AArch64, we found a third limitation while trying to implement sophisticated addressing modes: their semantics does not allow side effects on registers, yet some of them, such as the *post-indexed* addressing, are especially designed to update a register after the address computation. Those can thus not be supported without further modifications of the intermediate representation.

Globally, we see that several ad-hoc instructions of the AArch64 ISA cannot be represented at the RTL level, and must wait the Asm level to be modeled. Unfortunately, it is often a little late to take them into account in optimizations (e.g. in instruction scheduling).

[7] The AArch64 ISA [9] provides instructions capable of computing an arithmetic result *and* updating the condition flags based on this result—e.g. `ANDS`, `BICS`, etc.

3.5 ERRORS AND BUGS IN COMPCERT*

Although CompCert is formally proven, it is not totally infallible. This section briefly describes the main sources of bugs in CompCert, and discusses how to detect and avoid such bugs. Please see Monniaux and Boulmé [106] for an extensive discussion about CompCert's TCB.

First, as stated in Monniaux et al. [108]†, an important point is to clearly differentiate compile-time errors (when the compiler aborts and no code is produced) from miscompilations (when incorrect code is generated). Only the second category reveals flaws in the trusted computing base, on which the formal proof relies.
Here are the types of compile-time errors (points 1. to 3.) and miscompilations (points 4. to 6.) we experienced:

1. **Unexpected compilation failure (compiler internal error):** the formal proof assumes that compilation succeeds; always failing would trivially satisfy this criterion. Failures in CompCert can arise by some phase signaling an issue through the "error monad", or by an exception raised in handwritten OCaml code. Failing when incorrect behavior occurs protects against miscompilation, as no code is produced.

2. **Compilation timeout:** compilation may loop forever[8] or take prohibitively long.

3. **Error during assembling or linking:** in some cases, the code produced could not be assembled and linked; reasons for this range from details in the syntax comment of certain assemblers to the use of short branch instructions[9].

4. **Source semantics mismatch:** the C language is surprisingly complicated, and its semantics as formally defined in CompCert may diverge from the informal one defined in the standard, or in CompCert's manual.

5. **Assembly semantics mismatch:** the semantics of assembly language, plus platform-dependent peculiarities (e.g. how to access global symbols), may contain unexpected pitfalls (such as out-of-range operands resulting in a wrap-around behavior). Furthermore, some instructions present in CompCert's "assembly" languages are actually macros expanded by trusted (unverified) OCaml code. Some of these macros were inexactly specified, for instance by forgetting a clobbered register (for instance, bugs mentioned in §4.3.3.1)—this went unnoticed as long as the compiler did not take advantage of the value in that register being preserved.

6. **Assembly language mis-expansion or misprinting:** we also found rare miscompilations in the expansion or printing of macro-instructions. For example, I found an alignment error in the expansion of the memory copy built-in function for AArch64[10]. Fortunately, since this bug caused the GCC assembler to reject the code, it was not possible to produce an incorrect binary. Similarly, a harmless misprinting was present on the RISC-V backend[11], where the trusted OCaml printer was expanding goto and tail call macros to assembly instructions unable to benefit from linker relaxation. This resulted in a linking error for rare, heavy programs (once again, this could not produce incorrect code).

The key information to remember from this section is that the main source of miscompilation bugs is the sixth point above: the expansion of assembly macro-instructions [106]. These issues are possible because the last formally verified part of CompCert is only an abstraction of assembly code, whose concrete representation is generated by unverified code. The interested reader may refer to [108, §5]† for examples of such abstractions.

*Some text of this section is reused from [108]†.

[8] A compiler pass written in Coq cannot exhibit an infinite loop behavior (since Gallina programs always terminate), but this phenomenon may arise within OCaml code (e.g. oracles).

[9] A short branch and its target must be close, which may be false on large functions.

[10] Reported at https://github.com/AbsInt/CompCert/issues/410.

[11] Reported at https://github.com/AbsInt/CompCert/issues/436.

3.6 THE CHAMOIS-COMPCERT FORK

All transformations presented in this manuscript were implemented in the Verimag's fork of CompCert, named Chamois-CompCert. This fork is regularly merged with the upstream, mainline version to maintain its compatibility, and contains a bunch of transformations that do not exist in the mainline version. The most important ones, excluding my contributions, are: CSE3 [109], move-forwarding, superblock scheduling, postpass scheduling with peephole (on KVX), code duplication (various loop unrollings), code factorization, tail recursion optimization, aggressive inlining, branch profiling, and RTL-level tunneling (i.e. branch simplification). This list is not exhaustive. My contributions rely on some of them, notably the code duplications, the move forwarding pass, and the CSE3.

A comparison between CSE3 and LCM is proposed in §10.5.3.

4

SYMBOLIC EXECUTION: A CASE STUDY ON INSTRUCTION SCHEDULING VERIFICATION

This chapter aims to contextualize my work, present some preliminary contributions, and make the transition with the core topic developed in the next parts. The first verification experiment by symbolic execution in CHAMOIS-COMPCERT was from the work of Six [133]. It resulted in a successful verification of an instruction reordering optimization, applied directly on assembly code for Kalray's KVX cores. This transformation is named postpass scheduling [134], as it happens *after* the register allocation phase. A generalization of this approach *before* register allocation—a prepass scheduling—was proposed in [135]†.

I explain the notions of instruction level parallelism (ILP), and why scheduling is interesting for *in-order* cores in Section 4.1. During my PhD, I ported the KVX postpass to AArch64 [63]†, and I made a few improvements to the prepass [135]†. My contribution to the postpass reuses some generic modules and IRs from the KVX implementation, that are described in Section 4.2. Additions specific to the AArch64 port are presented in Section 4.3†. This chapter provides the basis for understanding symbolic execution, and highlights the differences between the specialized framework used for postpass, and its generalization for prepass scheduling in Section 4.4. I present some preliminary contributions I made on top of the prepass scheduling in Section 4.5†, as an introduction to the main contribution of my thesis outlined in Part ii. Lastly, Section 4.6 concludes.

4.1 INSTRUCTION SCHEDULING OPTIMIZATION

Most modern processors are *out-of-order* (OoO): they dynamically reorder instructions during the execution. To do this efficiently, they need to be capable of speculative execution, by predicting the control-flow path. In contrast, *in-order* cores execute the code sequentially: their internal control logic is simpler, and they consume less energy to achieve a given computation. Compile-time scheduling, as introduced by Feautrier [55], Fisher [57], and Rau, Glaeser, and Picard [121], and revisited by our research team in [135]†, aims to reorder instructions in code fragments to compensate the lack of dynamic optimization.

More surprisingly, we also notice a performance improvement on some OoO cores; this might be due to a smaller scheduling buffer inside the chip than the one of the scheduler.

4.1.1 *Interest: In-Order, VLIW, and Critical Systems*

A typical use of in-order cores is for safety-critical systems, because regulations often impose constraints on the worst-case execution time [60]. The *predictability* of the processor is thus an important concern in such applications, although this leaves the responsibility of generating efficient assembly code to the compiler, by fine-tuning the synthesis w.r.t. the core's specification. Hence, scheduling (whether it is before or after register allocation) can greatly reduce the execution time of the produced code. The principle is to exploit the processor *pipeline*, when the latter contains several execution units—e.g. arithmetic-logic units (ALUs)—allowing it to parallelize computations (a.k.a. superscalar pipeline, or horizontal parallelism [121]). This is complementary to *pipelining* (a.k.a. vertical parallelism), where the idea is to optimize the order according to the pipeline's stages. The compile-time scheduler uses the core description to place instructions cleverly, taking into account both axes of parallelism.

The Kalray KVX core is of the very long instruction word (VLIW) family [58]; each word is a *bundle*, or an encoding, of multiple atomic instructions, packed by the compiler. For instance, the Kalray KV3 core has 8-stages, 6 issue, interlocked pipeline: it can dispatch 6 instructions at a time, and, observationally, the assembly semantics of bundle composition is *sequential* (the hardware is able to dynamically stall when a dependency is not ready on time). The bundling is defined syntactically on assembly code using a string delimiter. For a bundle to be correct, it must not rely on more execution units than available in the pipeline. Since bundles are low-level constructs, they are built

during the postpass scheduling, over a code structured in basic blocks (i.e. with a single entry point and a single exit point, cf. §1.1.3.1). Scheduling optimizations for VLIW cores are well-documented in the literature, with software pipelining [89] or specialized toolchains [101].

Scheduling is not only interesting on VLIW cores: any in-order, superscalar processor might benefit from instruction reordering. For example, the AArch64 postpass scheduler port of §4.3 targets Cortex-A53 cores. These are dual-issue in-order superscalar cores very common in mobile systems on chip (e.g. in Raspberry Pi 3 and Nintendo Switch).

4.1.2 *Tiny Example of Instruction Scheduling**

Let us consider a small three-address code, as the one below:

$$I_1 \to r_2 \coloneqq r_1 + r_2; \quad I_2 \to r_3 \coloneqq \texttt{load}(r_1); \quad I_3 \to r_1 \coloneqq r_1 + r_3$$

If we have a 3-stages, single-issue pipeline, with one ALU, and assuming that arithmetic operations (e.g. addition and subtraction) have a latency of one cycle, while memory loads take two cycles, the above ordering is suboptimal. Indeed, as shown in the left table below, the core will stall before executing I_3 because the value of r_3 is not yet calculated.

bad scheduling $I_1; I_2; I_3$

	ISSUE	EXEC1	EXEC2
running time ↓	I_1		
	I_2	I_1	
	I_3	I_2	
	I_3	stall	I_2
		I_3	

good scheduling $I_2; I_1; I_3$

	ISSUE	EXEC1	EXEC2
running time ↓	I_2		
	I_1	I_2	
	I_3	I_1	I_2
		I_3	

One cycle is won!

With $t : \{I_1, I_2, I_3, \$\} \to \mathbb{N}$ the time at each instruction, the scheduler tries to minimize $t(\$)$, the total execution time for the sequence. The proposed schedule must of course satisfy the resource constraints, which correspond to $\forall i \in \mathbb{N}, |\{x / t(x) = i\}| \leqslant 1$ for a single-issue pipeline; and the latency constraints, defined as: $\{t(I_3) - t(I_1) \geqslant 1;\ t(I_3) - t(I_2) \geqslant 2;\ t(\$) - t(I_3) \geqslant 1\}$. On the above example, the optimal solution is the schedule "$I_2; I_1; I_3$", as it avoids the stall and executes with one less cycle.

4.1.3 *Previous Attempt at Verifying Postpass Scheduling in CompCert*

Integrating a scheduling pass after register allocation (i.e. in *postpass*) has already been attempted by Tristan and Leroy [142] (before the work of Six [133]), also relying on symbolic execution. Their scheduler was targeting the Mach IR (see Figure 3.1), and performs either list scheduling (on basic blocks) or trace scheduling [57] (on extended basic blocks). Operating at the Mach level is easier from an implementation point of view, but is also less efficient: in Mach, some instructions are in fact macros expanded into several assembly instructions (in §4.5.2, we show how to lift these expansions at the RTL level for the prepass scheduling). Hence, a scheduler at this level would have less precise information to complete its task than at the Asm level. Unfortunately, their work was not merged in a mainline CompCert release, due to complexity issues of the verifier on large programs [142, §7][140, §6.7.1].

Recall the paragraph on SE efficiency of §2.2.3.3.

4.1.4 *Prepass, Postpass, and Superblock Scheduling*

The postpass scheduler of Six, Boulmé, and Monniaux [134] operates on *basic blocks*, at the Asm level. This approach is simpler than prepass scheduling and aligns well with the bundle construction. In contrast, the prepass scheduler is implemented directly for *superblocks*.

*This example is partially reused from [23, §3.1].

4.1.4.1 *Scheduling and Register Allocation*

The main advantage of a postpass scheduler (w.r.t. a prepass scheduler) is the precise knowledge it has about real instructions: I said in §4.1.3 that scheduling at the Mach level would be less efficient; but doing it earlier would be even worse in terms of precision (i.e. there would be more macro-instructions). Nevertheless, an early scheduling is still interesting: when applied before register allocation, the optimization has more freedom (e.g. it can rename registers to remove dependencies), and can become quasi-independent of the target architecture[1].

A touchy point here is that when placed before allocation, the algorithm must be aware of the register pressure: in fact, since the number of registers is virtually unbounded, the algorithm will tend to increase the live range of registers. The live range here corresponds to the portion in the control-flow graph (CFG) where a register is *live*. If the live range of registers increases, the number of live registers at certain points in the code is also likely to increase; and when, at a point, this number is greater than the physical number of available registers (in the hardware of the target processor), then the allocator will insert "spills". Spilling is the technique used to free some registers by storing the content of these registers in the memory. It implies inserting loads and stores instructions, known to be very slow on many architectures. Put differently, a prepass scheduling is facing a conflict: it has to reduce what we call the makespan—the time at which the last value computed by the block is available—but it must also limit the register pressure [132]. Conversely, a postpass scheduling can reorder spills and insert other instructions between them.

After dead code elimination, we say that a variable is live at program point p if it is read on a path starting from p.

The prepass scheduler we propose in [135]† avoids the spilling issue by keeping a measure of register pressure. When only a few registers are available, it will try to choose an instruction that will free temporary registers to decrease the pressure (or at least one that does not increase pressure). This algorithm was implemented by Nicolas Nardino [117].

4.1.4.2 *Choosing Larger Blocks*

A scheduling heuristic is a *makespan minimization problem*, applied on a linear sequence of code; superblocks, as defined in §1.1.3.1, correspond in a sense to a linear sequence, since non-terminal branches (often called side-exits, or early exits) have only a single successor in the block. The slight difference is about liveness: to safely move an instruction below a side-exit, the result of the instruction must not be live at the side-exit. Intuitively, the live variables at a side-exit branching to block B are simply those at the entry of B. Superblocks provide a larger optimization window for the algorithm, and offer more scheduling opportunities than basic blocks. However, and in contrast with basic blocks[2], there are multiple ways of selecting superblocks; and this selection process is crucial to fully leverage the interest of scheduling. Actually, a bad selection could be counterproductive if the selected superblock corresponds to a rarely used path: a block optimized by the scheduler will be more efficient, but this is at the detriment of the other blocks. The goal is thus to select the paths that will be used most frequently, based on a prediction heuristic.

For more information about implementing an efficient selection heuristic with branch prediction, the reader can refer to [90], and [133, §4.1].

Similarly to the trace scheduling of Fisher [57] (and to the one of Tristan and Leroy [142]), each superblock is a possible path which might be taken by the program at runtime.

To increase the number of opportunities (and so the scheduling performance), a well-known method is to perform code duplications, such as *loop rotation*, *unrolling*, and *tail duplication*. Figure 4.1 illustrates some of these transformations. At first glance, this approach may seem counterintuitive as it appears to increase redundancy, which is generally undesirable in program optimization. However, as exemplified in §10.2.2, these duplications can actually facilitate the process of redundancy elimination, especially in the case of *trapping calculations*. This is due to the fact that duplication can prevent the need for proving an *anticipability property* on the trapping instruction, thereby simplifying the verification process.

4.1.5 *Untrusted Scheduler Oracle*

The prepass and postpass scheduling oracles actually leverage the same instruction scheduler backend. This backend abstracts the scheduling problem by viewing instructions as a combination

[1] It only needs to be configured by the latencies and constraints, without having to port it.
[2] Basic blocks are always selected in a way that maximizes the number of instructions per block.

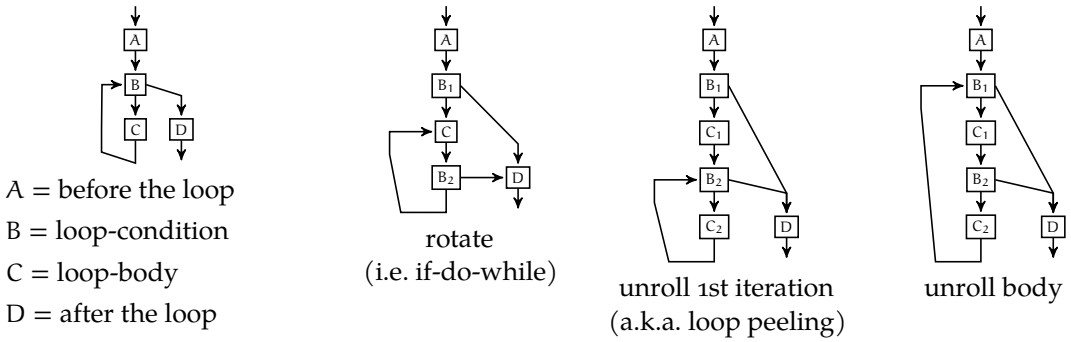


Figure 4.1: Three Loop-Unrollings of a "while-do" Loop.

of positions, latency, resource constraints, and two sets containing the input/output registers (read or written by the instruction). It aims to minimize the overall makespan using one of several heuristics: list, reverse list, greedy, integer linear programming, zigzag (alternating between list and reverse list), and register pressure sensitive scheduling. Naturally, dedicated frontends for prepass, postpass, and each backend are required to generate the abstract representation used by the generic backend. These distinct frontends and the unified backend are OCaml oracles invoked via the standard Coq foreign function interface (FFI) (recall the last paragraph of §2.4.1).

To configure the scheduler with the target core's latency and resource constraints, we define functions for each frontend that map every concrete instruction to its latency and resources. Latencies are simply encoded as the number of execution cycles for the instruction, while resources are tuples with each element signifying a specific processor unit. Consequently, core models are equipped with a reference resources tuple indicating the maximum simultaneous usage of each unit.

For example, the resource tuple for the Cortex-A53 core is $(\text{issue} : 2, \text{ALU} : 2, \text{MAC} : 1, \text{LSU} : 1)$. This means that the A53 is dual-issue, features two arithmetic-logic units, a single multiply-accumulate (MAC) unit, and a single load-store unit (LSU).

This oracle is a contribution of Cyril Six, David Monniaux and Nicolas Nardino. More technical details are provided in [134, §6], [135, §6]†, [133, §9], and [117].

4.2 FVDP OF A POSTPASS OPTIMIZER (*ARCHITECTURE-INDEPENDENT PARTS*)

The four next paragraphs sketch the four steps of Figure 4.2 to give an overview of the postpass framework. Only step (1) and a part of step (3) are architecture-independent. The block construction of step (1), summarized below, is not described further in this document, since I did not contribute to it, and it is not necessary to understand the rest. This section thus focuses on the language and simulation mechanism of step (3). Steps (2) and (4), along with the translation to the language of step (3), are covered in §4.3 for the AArch64 backend (to which I ported the postpass).

For a more complete description of this first step, please refer to Six, Boulmé, and Monniaux [134, §7].

To integrate postpass scheduling, the code is first cut into basic blocks, which is challenging in CompCert because the basic block structure is not directly recoverable at the Asm level. The issue comes from the state semantics of Asm: in that language, jumps or indirect calls instructions allow jumping *anywhere in the code* (i.e. there are no specific states for internal calls, nor return address protection in the semantics); so one cannot prove that the execution will not jump in the middle of a block. In contrast, Mach provides a well-separated state semantics (close to the RTL one presented in §3.4.1, with specific cases for returns and calls), making it suitable for constructing basic blocks. The Machblock IR (Figure 4.2), derived from Mach, is used for basic blocks construction and is shared between the AArch64 and KVX backends.

The second step involves translating Machblock to a block-based Asm called Asmblock on which we want to apply our optimization.

In the third step, the Asmblock code is scheduled and verified before its final translation to either Asm for AArch64, or AsmVLIW for KVX. From the outside, the scheduling is seen as a pass from Asmblock to itself; but the pass actually includes a translation to a DSL, AbstractBasicBlock, dedicated to the symbolic simulation. Proving the simulation from Asmblock to itself is straightforward, while

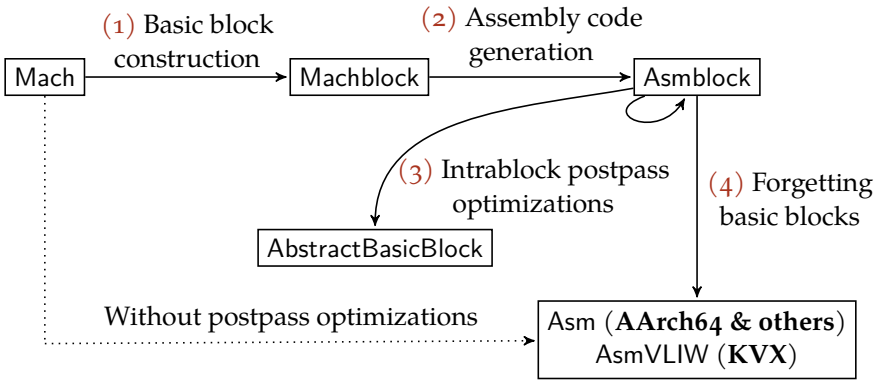


Figure 4.2: Architecture of the Postpass Optimizer Solution*.

proving the *correctness* of the symbolic simulation is more complex, and relies on refinement (refer to Boulmé [23, §1.3.2, §3.3] for an overview and example of refinement).

Finally, and fourthly, the scheduled Asmblock code is translated back to Asm (and to AsmVLIW for KVX), forgetting the basic block structure.

The AbstractBasicBlock DSL syntax is briefly introduced in §4.2.1 (without considering refinement), and its SE mechanism is presented in §4.2.2. Then, §4.2.3 shows how rewriting allows for a simple peephole optimization on the KVX backend, and §4.2.4 defines the refined symbolic states. The integration of our peephole & scheduling solution as an "Asmblock → Asmblock" CHAMOIS-COMPCERT pass is sketched in §4.2.5.

4.2.1 *AbstractBasicBlock: A Domain Specific Language (DSL) for Symbolic Execution*

To facilitate the development of the symbolic execution framework, Six, Boulmé, and Monniaux [134, §4.1] define a *deeply-embedded* DSL, named AbstractBasicBlock, in a generic module *parametrized* according to the target architecture (see its Coq interface [◇]). In their representation, an instruction is decomposed into *a list of tuples* $\texttt{inst} \triangleq \overrightarrow{(reg * exp)}$ where the first element is the destination register and *exp* the type of *expressions*. We say that type `inst` represents an atomic sequence of assignments. This representation allows for assembly instructions that modify several registers at once, which is not possible in RTL or Mach[3]. Hence, an *abstract basic block* is defined as a list of inst, $\overrightarrow{\texttt{inst}}$.

Expressions are *trees*, whose leaves are pseudo-registers (SE denotes each register with an identifier) and whose nodes are either operations (parametrized with a concrete operator), or "$\texttt{Old}(exp)$", with *exp* another expression. The "`Old`" constructor of an expression is used to encode the input value of a register in the surrounding AbstractBasicBlock instruction (see §4.2.3 & [134, C.1, Footnote 29]).

Definition 4.2.1 (Abstract type of expressions [◇])**.** The mutual variant of expressions allows to represent them as lists:

$$\begin{aligned} exp &::= \texttt{PReg}(reg) \mid \texttt{Op}(op, \texttt{list\_exp}) \mid \texttt{Old}(\texttt{exp}) \\ \textit{with } \texttt{list\_exp} &::= \texttt{Enil} \mid \texttt{Econs}(\texttt{exp}, \texttt{list\_exp}) \mid \texttt{LOld}(\texttt{list\_exp}) \end{aligned}$$

For instance, an instruction $r_1 := r_2 + r_3$ is noted:

$$[(r_1, \texttt{Op}(+, \texttt{PReg}(r_2) \mathbin{@} \texttt{PReg}(r_3) \mathbin{@} \texttt{Enil}))] \text{ with "\_@\_" the notation for } \texttt{Econs}(\_,\_).$$

Notice that source and destination registers are distinguished syntactically.

The domain specific language is parametrized by an evaluation function "$\texttt{eval\_op} : op \to \overrightarrow{val} \to val\ \texttt{option}$", with *val* the domain of values (cf. §3.3.1). The *memory* is represented as a single specific register "$r_m : reg$" that will remember memory writes as a stack of stores. Therefore, a symbolic state is a nothing more than a function "$\texttt{sstate} : reg \to val$".

*The figure is adapted from [134, Figure 3].

[3]This choice of representation takes advantage the fact that AbstractBasicBlock is only used to verify transformations, and not to generate code (contrary to RTL and Mach).

In this simplistic memory model, where the whole memory is represented as a single register, any instruction that writes to the memory is considered to overwrite it entirely. Consequently, the verifier is unable to validate a schedule interleaving loads or stores.

Setting a value "$v : val$" to a register "$r : reg$" in a symbolic state ss is done with a function "$\mathtt{assign}\ ss\ r\ v \triangleq \lambda r', (r = r')?v : ss\ r'$" [$\diamond$].

4.2.2 *Unidirectional Translation & Simulation Proof*

A simulation diagram of this block validation process is proposed in Figure 4.3.

The untrusted scheduler that performs the optimization works directly on the Asmblock representation, and returns a reordered version of the source program. Each block of both the source and transformed programs is translated to AbstractBasicBlock for verification. The goal is then to prove a *bisimulation* between the Asmblock programs and the AbstractBasicBlock ones, so that we do not need a reverse translation. Indeed, if the SE validates the proposed schedule on AbstractBasicBlock, then, thanks to the bisimulation theorem, we know that the schedule is also valid on the concrete representation.

4.2.2.1 *Note on the Asm Semantics*

Note that the Asmblock IR has the same semantics, but defined in a big-step fashion.

The Asm IRs of CompCert have executable semantics, based on very simple small-step state function. An Asm (or Asmblock) state is composed of a register set rs and a memory state m, and the semantics is a function "$(rs, m) \to (rs', m')\ \mathtt{option}$", where rs' and m' are the new register set and memory after executing the instruction at[4] $rs\#PC$ (the program counter). The function is in the option monad, so that it returns `None` when it is stuck, and the next state if it succeeds. For the KVX implementation, AsmVLIW also defines a state semantics for parallel execution of bundles. To the end of this section, I denote concrete Asm (and Asmblock) states with type "$\mathtt{state}$", and symbolic ones, as above, with type "$\mathtt{sstate}$".

4.2.2.2 *Bisimulation Between AbstractBasicBlock & Asmblock*

To show the bisimulation property, one need to compare symbolic states of AbstractBasicBlock with concrete ones of Asmblock. This leads to the following definition:

Definition 4.2.2 (Matching symbolic and concrete Asmblock states)**.** Let s and δ be the concrete and symbolic states, respectively; and let r_m the specific register representing memory in symbolic states. Recall (from §3.3.2) that CompCert's register sets are *not* extensional. We note "$\delta\ r$" the value of register r in symbolic state δ, and $s.(rs)$ to access the register state contained in s.

$$\mathtt{match\_states}\ s\ \delta \triangleq s.(m) = \delta\ r_m\ \wedge \forall r, s.(rs)\#r = \delta\ r$$

Now, we define two functions "$E : \mathtt{state} \to \mathtt{bblock} \to \mathtt{state\ option}$" and "$\xi : \mathtt{sstate} \to \mathtt{a\_bblock} \to \mathtt{sstate\ option}$". The former computes the result of the concrete execution given an initial state and an Asmblock basic block; and the latter returns the (symbolic) state obtained after symbolic execution, also given an initial (symbolic) state, and this time an *abstract* basic block (in the language of AbstractBasicBlock). Also, we note "$\tau_{\mathrm{ABB}} : \mathtt{bblock} \to \mathtt{a\_bblock}$" the *translation function* from Asmblock to AbstractBasicBlock. To compare the output of both execution functions, we define an auxiliary predicate:

Definition 4.2.3 (Matching concrete and symbolic executions' outcomes)**.** Let "$o_s : \mathtt{state\ option}$" and "$o_\delta : \mathtt{sstate\ option}$" two states in the option monad.

$$\begin{aligned}\mathtt{match\_outcome}\ o_s\ o_\delta \triangleq\ & \mathbf{match}\ o_s\ \mathbf{with}\\ & \mid \mathtt{None} \to o_\delta = \mathtt{None}\\ & \mid \mathtt{Some}\ s \to \exists \delta', o_\delta = \mathtt{Some}\ \delta' \wedge \mathtt{match\_states}\ s\ \delta'\end{aligned}$$

In other words, when one state is stuck, the other must be stuck too; and when both are ending on final states, then those states must match.

[4] We note $rs\#r$ the content of register r in regset rs.

The two previous definitions will now help us to define the bisimulation property. They are identical for both the AArch64 and KVX backends, and accessible here [⋄].

Theorem 4.2.1 (Bisimulation AbstractBasicBlock ↔ Asmblock[5] (AArch64 version [⋄])). *We assume two initial states, s the concrete one, δ the symbolic one, and a match hypothesis between them. For a basic block* bb *in the Asmblock language, we have:*

$$\texttt{match\_states}\ s\ \delta \implies \texttt{match\_outcome}\ (E(s, bb))\ (\xi(\delta, \tau_{ABB}(bb)))$$

The above property expresses that given matching initial states, then the states obtained after concrete execution on one side, and translation plus symbolic execution on the other side, match as well. The complete proof is a decomposition for basic (internal) instructions and exit (final) ones. The original version of this theorem was proved for the KVX backend by Six [133], and I adapted it for AArch64 by writing a translation from the AArch64's Asmblock to AbstractBasicBlock along with equivalent concrete and symbolic execution functions (see §4.3).

As mentioned before, this theorem is very convenient since it avoids translating back to Asmblock after SE.

4.2.2.3 *Correctness of the Postpass SE*

Before applying Theorem 4.2.1 to prove the simulation correctness, we need to define the simulation predicates for both languages. The first one below applies to AbstractBasicBlock, and is identical on AArch64 and KVX, while the second one, for Asmblock, depends on a concrete semantics.

As we are considering whole blocks here, both semantics are big-step. For clarity, we call such big-step semantics "blockstep" in the remainder of this section.

Definition 4.2.4 (Simulation predicate of AbstractBasicBlock (for both AArch64 & KVX [⋄])). The simulation predicate takes two abstract blocks: the source one and the transformed (scheduled) one, abb_1 and abb_2, respectively (they can be obtained by using τ_{ABB}). This definition is generic under a given implementation of ξ.

$$\begin{aligned}&a\_bblock\_simu\ abb_1\ abb_2 \triangleq \forall \delta : sstate, \xi(\delta, abb_1) \neq \texttt{None} \implies\\ &\quad \textbf{match}\ \xi(\delta, abb_1)\ \textbf{with}\\ &\quad \mid \texttt{None} \to \xi(\delta, abb_2) = \texttt{None}\\ &\quad \mid \texttt{Some}\ \delta_1 \to \exists \delta_2, \xi(\delta, abb_2) = \texttt{Some}\ \delta_2 \wedge \forall r, \delta_1\ r = \delta_2\ r\end{aligned}$$

Definition 4.2.5 (Simulation predicate of Asmblock (AArch64 version [⋄])).
The bblock_simu simulation predicate for Asmblock is very close to its abstract analogous, with the sstate type being replaced with concrete state, and with ξ replaced with E.

The correctness theorem of the whole SE is then simply an implication between the two simulation predicates. Its structure is sketched in Figure 4.3.

Theorem 4.2.2 (Symbolic execution correctness (AArch64 version [⋄])). *Let* bb_1 *and* bb_2 *the source and transformed blocks, respectively.*

$$a\_bblock\_simu\ (\tau_{ABB}(bb_1))\ (\tau_{ABB}(bb_2)) \implies bblock\_simu\ bb_1\ bb_2$$

Proof. We use two times Theorem 4.2.1 to show that both blocks transition to equivalent states. □

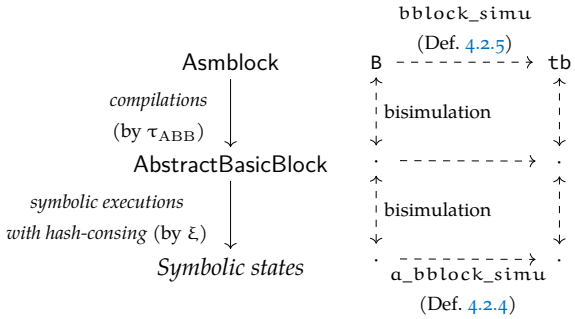


Figure 4.3: Correctness Diagram* for Theorem 4.2.2.

[5] This version of the theorem does not take into account the global environment of the program and some other details, for the sake of simplicity.

*This diagram is adapted from [134, Figure 7].

4.2.3 *Extending the KVX Postpass With a Simple Peephole*

Peephole optimizations [105] aim to replace some costly sequences of instructions by cheaper, semantically equivalent ones[6]. Such a technique was already implemented at the assembly level for x86 processors in a fork of CompCert, thanks to the dedicated framework of Mullen et al. [112][7].

The major advantage of the rewriting approach is that both peephole *and* scheduling could be applied sequentially (in that order, to leave more scheduling opportunities after peephole) by two distinct oracles, while being verified by a single run of the SE. Six [133] exploits this mechanism to replace pairs (respectively quadruplets) of loads and stores from aligned memory locations (with *the same base register*) by single quadruple-word (respectively octuple-word) instructions. Due to the constraints of the KVX architecture, its optimization can only replace instructions with aligned—destination (for loads) or source (for stores)—registers indices. Moreover, it can only perform the replacement when loads (or stores) are consecutive in the assembly code. Below is an example of a rewriting rule for quadruple loads, that illustrates the use of the "`Old`" AbstractBasicBlock operator:

The Coq implementation of this peephole complicates the support for non-consecutive instructions; an OCaml implementation would be more efficient thanks to the validation by translation.

Example 4.2.1 (Pairing two double loads into a quadruple load on KVX*)**.** Suppose the peephole oracle replaced two double loads of 64-bit words with a single quadruple load of a 128-bit word. The rewriting rule performs the reverse: it replaces the 128-bit quadruple load "`qload`" with a pair of 64-bit double loads "`dload`". The special register representing the memory is still (cf. §4.2.1) noted $\mathtt{r_m}$, and is, by definition, distinct from other registers. We name $\mathtt{r_0}$ and $\mathtt{r_1}$ the two adequately aligned destination registers. These two destinations are distinct from each other by construction. However, the base register $\mathtt{r_b}$ is not necessarily distinct from $\mathtt{r_0}$ or $\mathtt{r_1}$. Hence, in case of $\mathtt{r_b} = \mathtt{r_0}$, we specify the base register of the second load using the "`Old`" operator to refer to its initial value.

$$\begin{aligned}\mathtt{r_{dst}} \coloneqq \mathtt{qload(ofs)} \rightsquigarrow [&(\mathtt{r_0}, \mathtt{Op(dload(ofs), PReg(r_b) @ PReg(r_m) @ Enil})),\\ &(\mathtt{r_1}, \mathtt{Op(dload(ofs + 8), Old(PReg(r_b)) @ PReg(r_m) @ Enil}))]\end{aligned}$$

The above rule is applied during the translation from Asmblock to AbstractBasicBlock; the process is thus transparent w.r.t. SE. In §4.5.2, I present another manner of integrating such rules, by directly applying them during SE. In the AArch64 port presented in §4.3, I propose a more complex peephole optimization, also targeting memory related instructions, and capable of replacing non-consecutive sequences of loads and stores.

4.2.4 *Refining the AbstractBasicBlock Theory*

In contrast to the abstract states of §4.2.1, refined states do not model registers as an abstract function, but with a concrete data structure (a dictionary of hash-consed symbolic values "`hpost`", indexed by pseudo-registers):

```
(* Refined symbolic states *)
Record hsmem := { hpre: list term; hpost:> Dict.t term }
```

To ensure the absence of any additional trapping instruction in the target, refined (symbolic) states encode a block's precondition as a list of "ok" symbolic values "`hpre`", so that when the SE encounters a trapping operation, it is added to the list of the current state. The transformed block list (precondition) must thus be included in the source's one for the simulation to be correct. The *refinement relation* between the "`hsmem`" states above and their abstract version in the theory is then proved correct once and for all in the AbstractBasicBlock generic module (independently of the backend, see here [◇]).

[6] The "peephole" name indicates that the optimization operates over a window of instructions.

[7] Their peephole is based on the integer representation of pointers. Such low-level optimizations are out-of-scope of our work. In contrast, they do not support instruction reordering, nor loop optimizations. Moreover, they introduced a peephole execution engine with formally verified rewriting rules, but in a direct style, without translation validation.

*This example is adapted from [134, §C.1].

4.2.5 *Formally Verified Integration of an Assembly Optimizer in CHAMOIS-COMPCERT*

Postpass optimizations are integrated as an internal pass on the Asmblock IR. This pass is thus specific to each target backend. However, its structure remains similar. Initially, only KVX introduced such a postpass, and I later ported it to AArch64.

For each block `B` in the source Asmblock program, a function [◇] *sequentially* calls the postpass oracle (e.g. combining peephole and scheduling) to obtain an optimized block `B'`. This function is then proved to imply Definition 4.2.5 [◇] when it succeeds. Both `B` and `B'` are then translated to AbstractBasicBlock for validation. If the symbolic simulation fails, an error is raised and compilation is aborted. Otherwise, we repeat the process with the next source block. Ultimately, if no errors were raised, we obtain a transformed Asmblock program that preserves the semantics of the original one.

A figure illustrating this pass is available in [134, *Figure 6*].

The proof of this pass uses a simple "lock-step" simulation (top-left of Figure 3.2) ensuring that each individual step in the source block corresponds to *exactly one step* in the resulting block.

4.3 PORTING THE POSTPASS OPTIMIZER TO AARCH64†

To adapt the postpass optimizer, I had to proceed in five steps. These are listed below with the plan of this section and their corresponding step in Figure 4.2.

1. Defining Asmblock (i.e. a blockstep semantics) for the AArch64 Asm (§4.3.1, (2)).
2. Writing and proving the Machblock → Asmblock translation (§4.3.2, (2)).
3. Implementing the postpass itself. This consists in parametrizing the generic scheduler oracle (of §4.1.5) and conceiving a peephole oracle that pairs AArch64's loads and stores (§4.3.3, (3)). Also, it requires porting the surrounding Coq pass that combines the optional peephole and the scheduler oracles with their validator, as explained at §4.2.5.
4. Writing and proving the Asmblock → AbstractBasicBlock translation (§4.3.4, (3)).
5. Writing and proving the Asmblock → Asm translation (§4.3.5, (4)).

The table below indicates the lengths of Coq definitions and proofs in terms of *significant lines of code* (noted "sloc", excluding blank lines and comments with `coqwc`) for each part of this port.

Column "Defs" in this table comprises both Coq definitions and types, i.e. all extracted and executable code plus its specifications.

Name	Defs	Proof
Asmblock IR	743	26
Machblock → Asmblock translation	1072	15
Machblock → Asmblock translation proof	1255	2832
AbstractBasicBlock instantiation[8]	1800	725
Asmblock → Asm translation	392	0
Asmblock → Asm translation proof	602	1510
Coq Pass	212	197
Total	6076	5305

OCaml oracles are not counted in the table. In terms of sloc of OCaml (measured with `ocamlwc`) we have: 662 sloc for the postpass scheduler's frontend (including 111 sloc of fine-tuning); 495 sloc for the peephole oracle; and 1480 sloc for the scheduler's backend of David Monniaux and Nicolas Nardino. Recall that this backend is used for both prepass and postpass scheduling.

4.3.1 *A Blockstep Assembly Semantics for AArch64*

We present a blockstep assembly semantics for AArch64, which involves hierarchizing instructions based on their semantics and defining well-formed basic blocks.

[8] Including definitions, translation, evaluation functions specific to the DSL, and proofs.

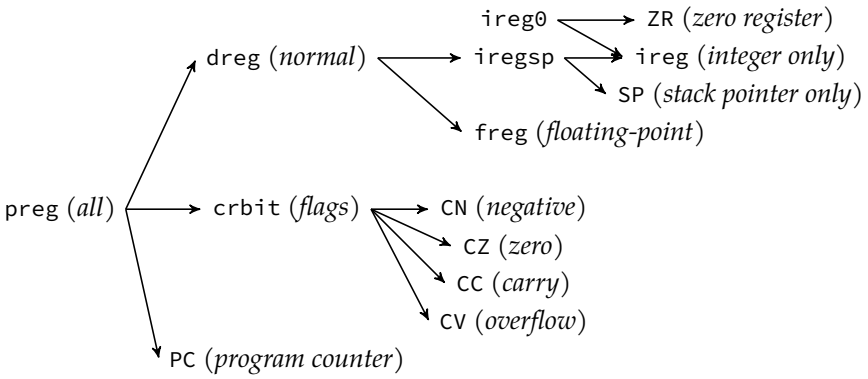


Figure 4.4: Existing Register Hierarchy of the CompCert AArch64 Backend.

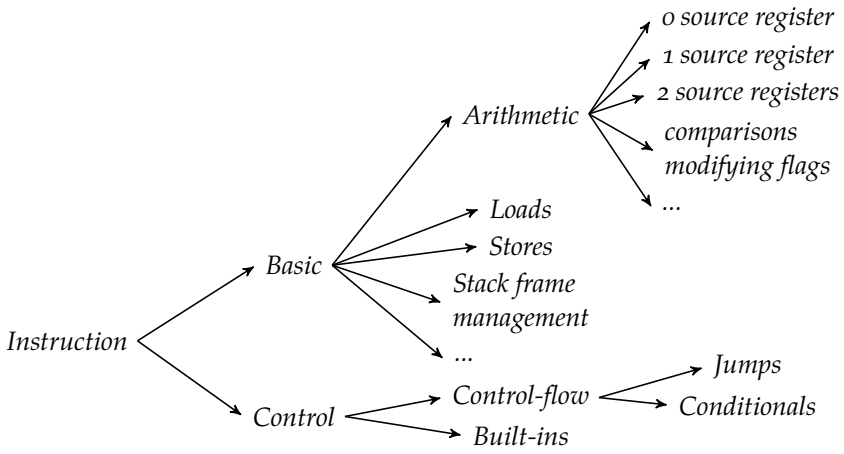


Figure 4.5: Extract of the Asmblock Instruction Hierarchy.

4.3.1.1 *Hierarchizing Instructions*

In order to automate Coq proofs of the translation to AbstractBasicBlock as much as possible, it is beneficial to *regroup* similar instructions—in terms of semantics—by defining a type hierarchy. In fact, we achieve a more precise regrouping by considering the existing register hierarchy [⋄] of the original AArch64 Asm. The latter defines multiple types of registers, so that we restrict instructions' arguments to a certain class of registers (see examples in §4.3.4). These classes of registers are depicted in Figure 4.4: the program counter is isolated, flags registers are grouped together, and integer registers are distinguished from floating-point ones. A special root group, `ireg0`, symbolizes the set of integer registers plus the special, read-only `ZR` register, whose value is always zero.

For a complete overview of this decomposition, please see the Asmblock Coq code here [⋄].

An extract of the Asmblock's instruction hierarchy is provided in Figure 4.5. The first level splits instructions into two main groups: *basic instructions*, which can only be part of the body of a basic block; and *control instructions*, which can only terminate a basic block. At the second level, control instructions are split to distinguish built-ins and control-flow instructions (and we split the latter group at level three to distinguish jumps from conditional branches). A built-in is considered as a control-flow instruction because it could emit a trace[9]. Similarly, basic instructions are divided into several groups, which may themselves contain other subgroups; for example, arithmetic instructions are a subgroup of basic ones, and they contain several subgroups depending on the type and number of argument registers.

The operational semantics of Asmblock adheres to the aforementioned hierarchy. Consequently, we obtain a concrete evaluation function *for each level* of Figure 4.5. These functions return a state in

[9]But, contrarily to the KVX implementation, we do not need to have built-ins alone in basic blocks (this requirement stemmed from VLIW bundles).

the option monad, and are subroutines of the top-level blockstep concrete execution function E (see its AArch64 implementation here [◇]).

4.3.1.2 *Well-Formed Basic Blocks and Operational Semantics*

We formalize basic blocks as follows:

Definition 4.3.1 (Basic block [◇])**.**

$$\begin{aligned} \mathit{bblock} \triangleq \{\, &\texttt{header} : \vec{L};\ \texttt{body} : \overrightarrow{\mathit{basic}};\ \texttt{exit} : \mathit{control}\ \texttt{option};\\ &\texttt{correct} : \texttt{body} \neq [] \vee \texttt{exit} \neq \texttt{None} \,\} \end{aligned}$$

Where L is the type of assembly labels tied to the block, and with "[]" denoting the empty list. A basic block is well-formed when either its body (which is a list of basic instructions) is not empty, or if it has an exit (i.e. a control-flow or built-in) instruction.

The execution of a basic block's body is performed through a recursive definition denoted as "$\texttt{exec\_body} : \texttt{state} \to \overrightarrow{\texttt{basic}} \to \texttt{state option}$" over the list of basic instructions. In the event that an instruction within the body fails, the execution halts and returns None. Otherwise, the resulting state and the remaining portion of the list are recursively passed to exec_body to ensure the execution continues.

As the semantics is blockstep, the program counter (PC) at rs#PC must be incremented *before* executing the exit instruction of the block (if it exists). The evaluation of the exit instruction modifies the state (and may emit a trace in case of a built-in). In any case, PC is incremented by the size of the basic block, which is the sum of the lengths of the label and instruction lists in the block, plus one if the exit instruction exists (zero otherwise). The function "$\texttt{exec\_exit} : f_{\text{asmblk}} \to \texttt{state} \to \texttt{control option} \to \mathbb{N}^* \to \texttt{state option}$" handles the PC increment and the evaluation of the exit instruction. The current Asmblock function (the first argument), is necessary to recover the position of a basic block while branching or jumping. The second parameter is the current state; the third is an option since the control instruction may not exist; and the fourth one contains the block's size, needed to increment the PC.

In practice, the concrete semantics of final instructions is written in two different forms. First, as a property that easily integrates with the CompCert's transition semantics:

Definition 4.3.2 (Semantic property of Asmblock [◇])**.** The property formalizes the execution of an entire block as follows: for an initial state $(\texttt{rs}, \texttt{m})$ and an *expected* final state $(\texttt{rs}', \texttt{m}')$, a block execution is valid if the body's execution successfully leads to an intermediate state $(\texttt{rs}_1, \texttt{m}_1)$ from which there is a valid exit step leading to the final state.

$$\begin{aligned} &\texttt{exec\_bblock}\ (\texttt{f} : f_{\text{asmblk}})\ (\texttt{bb} : \texttt{bblock})\ ((\texttt{rs}, \texttt{m})\ (\texttt{rs}', \texttt{m}') : \texttt{state}) : \texttt{Prop} \triangleq \exists \texttt{rs}_1\ \texttt{m}_1,\\ &\quad \texttt{exec\_body}\ (\texttt{rs}, \texttt{m})\ \texttt{bb.(body)} = \texttt{Some}(\texttt{rs}_1, \texttt{m}_1) \wedge\\ &\quad \texttt{exec\_exit}\ \texttt{f}\ (\texttt{rs}_1, \texttt{m}_1)\ \texttt{bb.(exit)}\ \texttt{size(bb)}) = \texttt{Some}(\texttt{rs}', \texttt{m}') \end{aligned}$$

Second, as an operational, executable version to instantiate function E of §4.2.2.2.
Note that an alternative of simulation Property 4.2.5, which was built on the executable blockstep semantics (i.e. function E), can be posed as the implication:

$$\begin{aligned} &\texttt{bblock\_simu}'\ (\texttt{f} : f_{\text{asmblk}})\ (\texttt{bb}\ \texttt{bb}' : \texttt{bblock}) : \texttt{Prop} \triangleq \forall \texttt{rs}\ \texttt{m}\ \texttt{rs}'\ \texttt{m}',\\ &\quad \texttt{exec\_bblock f bb}\ (\texttt{rs}, \texttt{m})\ (\texttt{rs}', \texttt{m}') \implies \texttt{exec\_bblock f bb}'\ (\texttt{rs}, \texttt{m})\ (\texttt{rs}', \texttt{m}') \end{aligned}$$

4.3.2 *Asmblock Generation From Machblock*

The whole translation module is available here [◇], and its proof here [◇].

The translation is very similar to the original Mach to Asm translation. Nonetheless, proving it correct is a difficult task.

4.3.2.1 *Translation and Macro-Instructions*

The Machblock IR defines blocks in the same fashion as Asmblock: they feature a header of labels, a body, and an optional control instruction at the end. Both Machblock and Asmblock functions store the code as a list of basic blocks. The translation from Machblock to Asmblock is done iteratively over this list. After translating the entire list of blocks, the function prologue is added as a separate block at the beginning of the list. For AArch64, the prologue includes a macro-instruction that *allocates the stack frame* (expanded in the untrusted OCaml printer, cf. §3.5); and a store that *saves the return address* in the stack.

A subtle issue arises regarding the application binary interface (ABI) of the target architecture. In AArch64 (and in most systems) there is a convention that specifies the roles of certain registers. For example, the AArch64 ABI [1] states that subroutine parameters should be passed through registers r0 to r7; but when there are more than eight arguments, they need to be stored in the activation record. Normally, the callee accesses these additional parameters by applying an offset on the stack pointer. However, this is not possible using the CompCert memory representation (i.e. the list of "block + offset" pairs, as mentioned in §3.3.3). Therefore, the prologue—more precisely, the frame allocation macro—saves the old stack pointer in a temporary register (here r15) when entering a subroutine. To avoid unnecessary reloading, the Asmblock generation pass keeps track of whether r15 was clobbered or not. This optimization reduces the number of memory accesses, and is further explained in [133, §3.3.2].

When the block to translate contains an exit instruction that *terminates* the current subroutine (e.g. a return or a tail call), a function epilogue is generated. The control-flow instruction corresponding to the return or tail call becomes the exit instruction of the generated basic block, and the epilogue is *appended to the block's body* (without being isolated like the prologue). Symmetrically to the prologue, the epilogue restores the return address from the stack and executes a macro-instruction to free the current stack frame.

Furthermore, *conditional branches* in Machblock are expanded into a sequence of assembly instructions during translation. In AArch64 assembly, most branches are encoded with an arithmetic comparison instruction that sets flag registers, followed by the conditional jump itself. In certain cases, such as immediate conditional branches, it may be necessary to load the immediate value beforehand. These preliminary instructions are inserted *into the body of the block*, so that the basic block's exit only contains the actual conditional jump for control-flow.

4.3.2.2 *Proof of Correctness*

Considering that our translation expanses macros, and handles *low-level constructs* such as the prologue, which involves creating an isolated basic block, the correctness proof needs to follow a "star" simulation (bottom left of Figure 3.2). In other words, each original basic block is translated into *one or more* Asmblock blocks. Additionally, since conditional branches are decomposed across the body and the exit during translation, we need to divide the proof into smaller simulations for the header, the body, and the exit parts.

We have a "star" simulation due to a single stuttering case on the Machblock step restoring the caller state (as in the existing Mach to Asm proof, see [134, A.1.2]).

I will not delve into the details of this proof in this document, as the overall proof pattern is very similar to the one depicted by Six, Boulmé, and Monniaux [134, §7.3, Figure 14]. The proof can be seen as a combination of the original Mach to Asm translation proof of Xavier Leroy and the corresponding proof implementation for the KVX backend.

4.3.3 *OCaml Oracles for Peephole & Scheduling*

4.3.3.1 *Parametrizing the Scheduler for AArch64*

On the Coq side, we declare the postpass scheduler frontend as "`Axiom schedule: bblock → (list basic) * option control`". Notice that it does not directly return an instance of the `bblock` type. This is because, as explained in §2.4.1, the OCaml typechecker is less expressive than the

Coq one. Thus, it would be *unsound to assume that the oracle can ensure the basic block proof field*[10] of Definition 4.3.1.

Writing the oracle's frontend to translate the AArch64 Asmblock to the backend's abstract representation was relatively simple. Our main obstacle was to obtain the most precise information possible about instruction latencies, to *accurately "tune"* the oracle. This is mainly because accurately measuring the number of execution cycles for each instruction is challenging, and the manufacturer in the case of the Cortex-A53 (i.e. ARM) does not publicly provide such information. However, the AArch64 LLVM backend uses a similar postpass scheduling optimization, and its source code contains some (though incomplete) latency information. Another source we utilized is a paper by Wiggers [146], where some latencies are manually measured. The reader can refer to §12.3.3 for an overview of the efficiency with our current settings.

Regarding the sets of read and written registers for each instruction, these can be deduced from the Asmblock semantics. However, an interesting point here is that during the implementation of our solution, we discovered critical bugs in the Asm semantics: indeed, certain instructions such as `Pfmovimms/Pfmovimmd` (floating-point moves for 32- and 64-bit registers) and `Pbtbl` (jump tables) were *incorrectly described*. The two moves overwrite a scratch register before writing the result in the destination register, and this behavior was not modeled in their operational semantics. The opposite issue arose for the jump table, which, contrary to its described semantics, *preserves* a scratch register. This issue does not result in incorrect code, but the specification was too strong and induced *inefficient code*. Those three instructions are macros expanded later in the unverified printer of CompCert (cf. §3.5), into real Asm instructions. Because their behavior was incorrectly formalized in Coq, the two moves could lead to incorrect code by interleaving them with other macros that use the same scratch register, and which are expanded in Coq, when translating to Asmblock (before our scheduling). Given that instructions had never been interleaved at the Asm level before, this bug remained undetected[11]. Our verifier, combined with postpass scheduling, helped identify these errors in CompCert's trusted computing base.

4.3.3.2 *Peephole Pairing Loads and Stores Instructions*

We also declare the peephole as a Coq axiom, but, unlike the scheduling frontend, the peephole oracle only returns the list of instructions (i.e. the block's body), since it never changes the block's exit instruction. The AArch64 ISA includes specific instructions to transfer two registers simultaneously to (store) and from (load) the memory [10, §C.3.2.3]. Mainline CompCert does not assign any semantics to those instructions. Therefore, we began by incorporating double load and store instructions into the Asm and Asmblock IRs. This addition results in *eight new instructions*: four double load instructions for 32/64-bit integers/floats, and four corresponding double store instructions.

REPLACEMENT CONSTRAINTS The four types of double loads (and the same applies for double stores) share the same operational semantics:

Definition 4.3.3 (Coq semantics of all double loads)**.**

```
Definition exec_load_double (chk1 chk2: memory_chunk) (a: addressing)
(rd1 rd2: preg) (rs: regset) (m: mem): option state :=
  if is_pair_addressing_mode_correct a then
    let addr := (eval_addressing a rs) in
    let ofs := match chk1 with Mint32 | Mfloat32 | Many32 ⇒ 4 | _ ⇒ 8 end in
    let addr' := (eval_addressing (get_offset_addr a ofs) rs) in
        match Mem.loadv chk1 m addr with
        | None ⇒ None
        | Some v1 ⇒
            match Mem.loadv chk2 m addr' with
            | None ⇒ None
            | Some v2 ⇒
```

[10] In other words, according to the terminology of Boulmé [23, §2.2.2], type `bblock` is not *permissive*. It should be avoided as an output type of oracles.

[11] Those three bugs were quickly fixed by Xavier Leroy in https://github.com/AbsInt/CompCert/commit/0df99dc46209a9fe5026b83227ef73280f0dab70.

```
            Some (nextinstr ((rs#rd1 ← v1)#rd2 ← v2)) m
          end
        end
  else None
```

The definition deduces the offset from the requested memory chunk; it equals four for 32-bit words and eight otherwise. Replacement only occurs for pairs of loads (or stores) with consecutive offsets, a constraint imposed by the AArch64 ISA specification. For two loads $\texttt{ld}_1$ and $\texttt{ld}_2$ with offsets $\texttt{ofs}_1$ and $\texttt{ofs}_2$ (respectively), we require $|\texttt{ofs}_1 - \texttt{ofs}_2|$ to equal four (32-bit case) or eight (64-bit case). For stores, we support only pairs where the second offset is greater than the first. This limitation on stores comes from our validation mechanism: inverting stores is not supported by our naive memory model (cf. §4.2.1). The execution semantics can merge loads of differing chunk types, as long as their sizes match.

The operational semantics for double instructions executes the second memory operation only if the first one succeeds, avoiding additional traps.

Double loads, unlike regular loads, lack support for embedded conversions or shifts, necessitating a preliminary check for valid addressing mode (`is_pair_addressing_mode_correct`). Indeed, the AArch64 ISA offers numerous sophisticated addressing modes [10, §C.1.3.3], not all of which CompCert implements. In fact, double loads/stores work with the classical "Base + Offset" addressing mode and pre/post indexed modes, but the two latter are currently not supported in CompCert. Therefore, our optimization is designed only for the case of an addition between the base register and an immediate offset without any additional modification. This corresponds to the `ADimm` constructor in the evaluation function below, which comes from the original CompCert Asm for AArch64:

```
(** Evaluating an addressing mode *)
Definition eval_addressing (a: addressing) (rs: regset): val :=
  match a with
  | ADimm base n ⇒ rs#base + (Vlong n)
  | ADreg base r ⇒ rs#base + rs#r
  | ADlsl base r n ⇒ rs#base + (rs#r << (Vint n))
  | ADsxt base r n ⇒ rs#base + ((Val.longofint rs#r) << (Vint n))
  | ADuxt base r n ⇒ rs#base + ((Val.longofintu rs#r) << (Vint n))
  | ADadr base id ofs ⇒ rs#base + (symbol_low ge id ofs)
  | ADpostincr base n ⇒ Vundef
  end
```

When we actually replace a load or a store, we take the smallest of the two source offsets as the offset for our newly generated pair instruction. Moreover, the selected offset immediate must satisfy certain properties for the load (or store) to be a valid potential candidate for substitution. The ISA reference manual imposes that any selected offset z satisfies: $-256 \leqslant z \leqslant 252 \wedge z \mod 4 = 0$ for a 32-bit word and $-512 \leqslant z \leqslant 504 \wedge z \mod 8 = 0$ for 64-bit. In practice, the peephole oracle disregards instructions that do not meet these constraints. However, they are *not enforced* in the Coq formal semantics. Omitting them is not a problem here, because the final assembler will raise an error in case they are not satisfied.

Finally, we must check that the base registers for both source instructions are the same, and, for loads, that their destination registers are different.

ORACLE AND EXAMPLE The oracle is capable of replacing both *consecutive* and *non-consecutive* loads and stores, as long as they respect the constraints enumerated above. The algorithm traverses the instruction list in *both directions*. As it iterates over the list, it keeps track of all encountered compatible loads and stores as potential candidates for peephole, and removes them if another instruction breaks a dependency in-between. The first pass, forward, attempts to replace the last encountered load or store with a double instruction, and substitutes the first one with a no-op (no-operation) instruction. Conversely, the second pass, backward, does the opposite operation. The following are examples supported by our oracle:

Example 4.3.1 (Four examples of pairing loads and stores*)**.** In CompCert AArch64, 32-bit registers are prefixed with letter "w" and 64-bit ones with letter "x". Immediates are always preceded by a pound "#".

```
movz x6, #0, lsl #0
ldr w4, [x6, #0]
sxtw x3, w4
ldr w1, [x6, #4]
ldr w5, [x3, #0]
ldr w7, [x3, #4]
add w2, w4, w1
adrp x16, a
```

```
movz x6, #0, lsl #0
ldp w4, w1, [x6, #0]
sxtw x3, w4
ldp w5, w7, [x3, #0]
add w2, w4, w1
adrp x16, a
```

```
mov x0, x19
ldr x19, [sp, #16]
ldr x30, [sp, #8]
movz x1, #0, lsl #0
str w2, [x1, #0]
movz w0, #0, lsl #0
str w2, [x1, #4]
sub w0, w0, w2
```

```
mov x0, x19
ldp x30, x19, [sp, #8]
movz x1, #0, lsl #0
movz w0, #0, lsl #0
stp w2, w2, [x1, #0]
sub w0, w0, w2
```

On listings 1 & 2:

1. In orange color: backward load pairing, with increasing offset (the offset of the second load is *greater* than that of the first one). Note that here, forward pairing would be *incorrect* as `w4` is read between the two source loads;
2. In lime color: consecutive load pairing, with increasing offset.

On listings 3 & 4:

1. In blue color: consecutive load pairing, with decreasing offset (the offset of the second load is *lower* than that of the first one);
2. In pink color: forward store pairing, with increasing offset.

Example 4.3.3 illustrates the reverse rewriting of a double load into two normal loads performed during the translation to the AbstractBasicBlock DSL.

4.3.4 *Instantiating the SE for AArch64*

The AbstractBasicBlock main translation function is defined here [◇]. Corresponding proofs are included in the same file.

4.3.4.1 *Translation to AbstractBasicBlock*

The translation to our DSL benefits from the previously defined hierarchy, avoiding the tedious task of describing each instruction individually. Instead, we define at least one operation for each class of instructions in the Asmblock IR.

Example 4.3.2 (Translation of a simple arithmetic instruction)**.** As an example, an Asmblock's arithmetic instruction `i` with one destination register r_d of type `dreg` and one source register r_s of the same type is noted as "`PArithPP i` r_d r_s". Various Asm instructions belong to the `PArithPP` class, such as moves, conversions, or immediate arithmetic operations. AbstractBasicBlock sticks to this hierarchy, making the translation very straightforward:

$$\texttt{PArithPP i rd rs} \rightsquigarrow [(r_d, \texttt{Op(Arith(OArithPP i))(PReg}(r_s)\texttt{@Enil)})]$$

The hierarchy is directly reflected in the syntax: instruction `i` belongs to class `OArithPP` (the AbstractBasicBlock's equivalent for `PArithPPP`), which is itself a subclass of `Arith`.

MORE COMPLEX TRANSLATIONS Nevertheless, recall that the AbstractBasicBlock DSL syntax represents an instruction as a sequence of atomic assignments. Each element in this sequence assigns a single register with the result of an expression containing the AbstractBasicBlock operation. In other words, an assembly instruction that has n distinct destination registers should be translated as a sequence of n distinct AbstractBasicBlock operations, each returning the value of the expected register. Moreover, since input registers are explicitly listed in the syntax, each AbstractBasicBlock operation must have a fixed number of inputs that correspond to a symbolic value in the state. This is an issue for special read-only registers like the always zero register `ZR` (of class `ireg0`), because they do not correspond to a symbolic state's value.

*This example is reused from my short publication at AFADL 2021 [63, Figure 3]†.

Let us clarify this with a more complex example, that illustrates both above subtleties. We focus on Asmblock instructions of the form "`PArithComparisonR0R` i r_{s1} r_{s2}". This class contains arithmetic comparisons with two arguments that may write the four flags registers (class `crbit`). The second argument is always an integer register (class `ireg`), but the first one is of class `ireg0`. Such comparisons are translated as a list of four atomic assignments writing the four possible destination flags (so we need four AbstractBasicBlock operations). Each of these four operations must be defined in two forms to separate the case where one argument is `ZR`. Hence, we end up with eight AbstractBasicBlock operations to model a single Asmblock instruction.

Note that the implementation of Six, Boulmé, and Monniaux [134] did not face such a problem, because the KVX architecture features a much less complex register hierarchy (KVX has 64 general-purpose registers usable for both integer and floating-point values, and no flag registers).

REVERSE REWRITING THE PEEPHOLE'S TRANSFORMATIONS Exactly as in §4.2.3, our translation to AbstractBasicBlock reverts the peephole rewrites by replacing double loads/stores with pairs of their normal equivalents.

Example 4.3.3 (Translation and rewrite of the orange double load of Example 4.3.1)**.**

$$\begin{aligned} r_{dst} \coloneqq \texttt{ldp}(0) \rightsquigarrow [&(w_4, \texttt{Op}(\texttt{ldr}(0), \texttt{PReg}(x_6) \mathbin{@} \texttt{PReg}(r_m) \mathbin{@} \texttt{Enil})),\\ &(w_1, \texttt{Op}(\texttt{ldr}(4), \texttt{Old}(\texttt{PReg}(x_6)) \mathbin{@} \texttt{PReg}(r_m) \mathbin{@} \texttt{Enil}))] \end{aligned}$$

The `Old` operator is always generated by our translation, even if here, $w_4 \neq x_6$.

4.3.4.2 *Semantics and Proof of Correctness*

Most evaluation functions of AbstractBasicBlock are directly reusing the concrete evaluation functions from the Asmblock operational semantics. For complex decompositions like the eight operations model of the above example, we sometimes had to redefine evaluation functions more adapted to the AbstractBasicBlock syntax.

We then prove our instance of bisimulation Theorem 4.2.1 between AbstractBasicBlock and Asmblock for AArch64 by case analysis on each kind of instruction. As claimed in §4.2.2.3, we apply two times the latter bisimulation theorem to prove the overall correctness of the symbolic execution framework, formalized by Theorem 4.2.2. See the code online for details.

4.3.5 *Coming Back to Asm*

4.3.5.1 *Flattening Basic Blocks and Translating to Asm*

The process of converting basic blocks to sequential assembly is quite simple. We just need to transform the basic block structure into a list of instructions in sequential order. To achieve this, we define an unfolding function that performs the task while checking the length of the header dynamically. This check relies on the tunneling optimization pass previously applied at the location transfer language (LTL) level of CompCert.

The main unfolding function is available here [◇].

The tunneling optimization ensures that consecutive labels cannot be found, meaning that the label list in a basic block header never contains more than one element. To emphasize the significance of this check, let us define the function used in the Asmblock IR to locate the position of a label in the code:

```
Fixpoint label_pos (lbl: label) (pos: Z) (lb: bblocks) : option Z :=
  match lb with
  | nil ⇒ None
  | b :: lb' ⇒ if is_label lbl b then Some pos
               else label_pos lbl (pos + (size b)) lb'
  end
```

This function will provide an address pointing to the first instruction of the basic block. By using the dynamic check, we can demonstrate that the `label_pos` function in Asm (which simply returns the position of the label in the sequential code) will retrieve the exact same address. Maintaining

this property allows us to use a straightforward state equality in the proof. Without this header check, we would have to handle the case where a basic block header contains multiple labels. In such a scenario, while `Asmblock.label_pos` would point to the same location at the beginning of the basic block for all labels, `Asm.label_pos` could return a position pointing into the original basic block.

Apart from this header check, the translation process is managed by only two functions: one for basic instructions and another for control-flow instructions.

4.3.5.2 *Forward Simulation Proof*

Thanks to the previous dynamic check on labels, Asmblock and Asm share the same state definition, comprising a register set and a memory (as briefly explained in the beginning of §4.2.2.1). This commonality is convenient as it enables us to define the matching relation between an Asmblock state and an Asm state using a *simple structural equality*.

The whole Coq proof of this translation is available here [◇].

Our main simulation theorem thus follows the "plus" scheme (top-right of Figure 3.2), indicating that one step of the source Asmblock program, producing an observable event trace e, can be *simulated* by multiple steps of the translated Asm program, also producing the same trace e. The theorem splits the simulation in two cases: the internal step and the external step, to account for external calls such as built-in functions. The second case simply expresses that symbols and arguments are preserved during such calls.

The internal case is somewhat tedious due to the change in the representation of instructions in Asmblock. Assuming an internal function `f`, and a program counter `pc` pointing to a basic block `bb` within `f`, we consider a successful execution "`exec_block f bb (rs, m) (rs', m')`" (Property 4.3.2) that emits a trace e. Our proof demonstrates that it implies the existence of an equivalent transition on the Asm side, occurring in at least one step: $(\mathtt{rs}, \mathtt{m}) \xrightarrow{e}_{+} (\mathtt{rs}', \mathtt{m}')$. Here, "$\xrightarrow{e}_{+}$" indicates a "plus" transition that emits the trace e.

We also encounter two subcases here, which arise from our well-formed property. Specifically, for a basic block to be considered valid, either the body or the exit must not be empty. Depending on which part is non-empty, we have the following simulation strategies. (i) If the basic block's body is not empty, we perform a "plus" simulation for the body and a "star" simulation for the exit. This means that we simulate the execution of the body in at least one step, while allowing zero or more steps for the exit. (ii) Conversely, if the exit is not empty, the simulation strategy is reversed. In this case, we need a "star" simulation for the body and a "plus" simulation for the exit.

4.4 GENERALIZING TO PREPASS SCHEDULING

After working on the postpass scheduling of the previous section, Six [133] reproduced the approach to obtain a verification framework at the RTL (cf. §3.4) level, for prepass scheduling over superblocks. In this section, I present this work and its limitations, as a transition to my preliminary contributions. More specifically, we will see how the idea of a modulo invariant simulation mechanism has become clearer. I start with a quick review of the Six's representation in §4.4.1, and of the related liveness verification in §4.4.2. A high-level example of superblock SE is given in §4.4.3; and the verification process, along with its integration within Chamois-CompCert are covered in §4.4.4. Finally, the limitations are discussed in §4.4.5.

4.4.1 *Decorating RTL With Path Maps: RTLpath*

The original RTL IR is a CFG of single instructions. In the work of Six et al. [135, §7.1]†, it is extended with a path map structure: each of these paths represents a superblock. Formally, paths are seen as the *traces* of trace scheduling, and are not required to be disjoint (the set of path is not necessarily a partition of the CFG)[12]. I mentioned in §4.1.4.2 that superblock selection was relying on a branch prediction heuristic; paths encode this information with a notion of default successor. Basic instructions (e.g. no-ops, loads, stores, operations) always have a default successor (the original one); while the default successor of a branching instruction is defined only if the condition was predicted.

[12] Notice that a superblock decomposition of the CFG exactly corresponds to the case where each node is in at most one path.

Non-predicted branches and final instructions (e.g. calls, gotos, etc.) have no default successor, and are thus always ending a path. That is, a path's execution emits at most a single observable event. RTLpath encapsulates the notion of RTL function (cf. §3.3) as follows:

```
(* Additional information about paths *)
Record path_info := {
  psize: nat; (* Number minus 1 of instructions in the path *)
  (** Live registers at the path's entry *)
  input_regs: regset;
  (** Live registers at the entry of the next path: *)
  (** - In pre-output: without considering the result of a call or built-in *)
  pre_output_regs: regset;
  (** - In complete output: always considering the last instruction
  NB: the output_regs field below is only used in oracles. *)
  output_regs: regset
}

Definition path_map: Type := pc ↦ path_info

Definition path_entry (pm: path_map) (n: pc): Prop := pm!n <> None

Inductive wellformed_path (c:code) (pm: path_map):
nat → pc → Prop :=
  | wf_last_node i pc:
      c!pc = Some i →
      (∀ n, List.In n (successors_instr i) → path_entry pm n) →
      wellformed_path c pm 0 pc
  | wf_internal_node path i pc pc':
      c!pc = Some i →
      default_succ i = Some pc' →
      (∀ n, early_exit i = Some n → path_entry pm n) →
      wellformed_path c pm path pc' →
      wellformed_path c pm (S path) pc

(* all paths defined from the path_map are well-formed *)
Definition wellformed_path_map (c:code) (pm: path_map): Prop :=
   ∀ n path, pm!n = Some path →
   wellformed_path c pm path.(psize) n

(* There is a trivial "forgetful functor" from RTLpath programs to RTL ones. *)
Record function : Type := {
  fn_RTL:> RTL.function;
  fn_path: path_map;
  fn_entry_point_wf: path_entry fn_path fn_RTL.(fn_entrypoint);
  fn_path_wf: wellformed_path_map fn_RTL.(fn_code) fn_path
}
```

Hence, a RTLpath function *contains* the original RTL function, two well-formedness properties, and the path map (i.e. which is simply a map from positive integers[13] to `path_map` option records). In our Coq code, we use the "`m!x`" notation to access element `x` in map `m`. Observe that the above definitions do not impose that paths have a single entry point; but this is required for the verifier (otherwise the simulation test would probably fail). A *well-formed* path must respect the below conditions (the last two are encoded in the `fn_path_wf` field):

- The entry point of the code must be the entry point of a path (`fn_entry_point_wf`);
- The successor of its *last* node (when size is `0`) must be the entry of another path;
- Each early exit's (a node `n` is an early exit if it has a default successor, and if it is a conditional branch; i.e. when `early_exit` is defined) successor is also a path entry.

[13] Implemented using the Coq `PTree` library [8].

In the `path_info` map, if a node `n` is defined in the map (i.e. if it is not `None`), then `n` is a path entry. Actual (from `RTL`) successors are given by the "`successors_instr`" function, and default ones by the "`default_succ`" function. Comparing with the work of Six [133], I added the `pre_output_regs` field in the `path_info` record. The latter, `input_regs`, and `output_regs` are used for storing and verifying information about liveness: they are introduced in §4.5.1. RTLpath offers two main bisimulation proofs: one with the original `RTL` semantics; and another for "path-step" execution (a big-step semantics that corresponds, in a sense, to the blockstep simulation of AbstractBasicBlock). I will not detail the simulation proof between `RTL` and RTLpath here, nor the executable semantics of RTLpath, since it is not relevant for understanding and already well explained by Six [133, §5.2.2]. A major advantage of this decoration approach is that all analyses available for `RTL` programs should be appliable "for free" on RTLpath programs too.

4.4.2 *Why Check the Liveness?*

When moving an instruction above or below a side-exit, the verifier needs to ensure that it does not change the result of the exit part of the branch. This requires a liveness analysis: the live registers at each exit must not be modified by the applied optimizations.

During the RTLpath generation from `RTL`, the live registers at the entry of each path are computed by the oracle, and a certified (*dedicated*, so apart from `SE`) validator dynamically check their correctness[14]. A predicate encoding the result is then propagated to the scheduling pass, to remember the liveness property. The symbolic execution in RTLpath is thus modulo liveness of source's live registers, but only on side-exits[15]. In other words, it means the simulation test (on side-exits), which compares symbolic states, is only performed on registers that were live in the source program.

Doing so is sufficient for superblock scheduling (see §4.4.3), but it *prevents the renaming* and the introduction of auxiliary *fresh registers*. This is because at the end of the path, these registers would be considered modified on the target side but not on the source side, and the verifier of Six [133] compares every register at the final exit, including those dead registers.

4.4.3 *An Example of Superblock SE*

In the case of blocks having multiple exits, the result of `SE` is not a single state anymore, but a binary decision tree, with symbolic states on the leaves (one per exit, whether early or not). The simulation test compares these trees to conclude about semantic preservation.

Lifting a trapping instruction above an early exit would add a potential trap if the exit is always taken. Moving an instruction below an early exit is possible only if its result is not live at the early exit output.

The source liveness based simulation, in addition to allowing the movement of (non-trapping) instructions above or below side-exits, alleviates the simulation test by reducing the number of variables to compare.

Example 4.4.1 (Superblocks simulation)**.** Consider two superblocks B_1 and B_2:
We assume the $\otimes$ operator to be potentially trapping.
(B_1) $r_1 \coloneqq r_1 \otimes r_2$; `if` $(r_2 \geqslant 0)$ `goto` L;
(B_2) `if` $(r_2 \geqslant 0)$ `goto` L; $r_1 \coloneqq r_1 \otimes r_2$;
B_2 simulates B_1 if r_1 not live at L; B_1 simulates B_2 if r_1 not live at L and "OK$(r_1 \otimes r_2)$".

This example shows how the *modulo liveness* simulation can validate the reordering of side-exits. I clarify in §4.4.5 why it would be even better to support the introduction of fresh registers (by making the whole simulation modulo liveness, not just side-exits).

4.4.4 *Overview of the RTLpath SE Verifier*

Although these concepts are not essential to understand the rest of this document, they establish the framework for this study.

Just like the AbstractBasicBlock symbolic execution, the RTLpath framework is defined with first a simplified theory, which is then refined a first time to comply with concrete specifications, and a second time to include implementation related tools such as hash-consing specific constructors (introduced in §2.4.2).

[14] It has no impact on the semantics, but it aborts the compilation if it fails.
[15] On the final exit of the superblock, the simulation is strict; see §4.4.5.

$$\texttt{sistate\_local} \triangleq \{\ \texttt{si\_pre}: \ regset \to mem \to Prop;\ \texttt{si\_sreg}: \ reg \to sval;\ \texttt{si\_smem}: \ smem\}$$

$$\texttt{sistate\_exit} \triangleq \{\ \texttt{si\_cond}: \ condition;\ \texttt{si\_condargs}: \ list\_sval;\ \texttt{si\_elocal}: \ sistate\_local;\ \texttt{si\_ifso}: \ pc\}$$

$$\texttt{sistate} \triangleq \{\texttt{si\_pc}: \ pc;\ \texttt{si\_exits}: \ \overrightarrow{sistate\_exit};\ \texttt{si\_local}: \ sistate\_local\}$$

Figure 4.6: Internal States in the RTLpath Theory: Local State (top-left), Exit State (top-right), and Full Internal States (bottom).

4.4.4.1 *The RTLpath Theory*

Verifying superblocks requires to execute them one by one, and then to compare the resulting symbolic states (as it was done in §4.2.1 for basic blocks). Nevertheless, since RTL (and RTLpath, and more generally CompCert) proves the simulation over a program decomposed into functions, the verifier correctness must consider a whole function (as explained in the end of this section) and its surrounding stack frame[16].

Definition 4.4.1 (Mutually inductive symbolic values)**.** The theory module first inductively defines a type for symbolic values as follows:

$$\begin{array}{rcl} sval & ::= & \texttt{Sinput}(reg) \mid \texttt{Sop}(op, \texttt{list\_sval}, \texttt{smem}) \\ & \mid & \texttt{Sload}(\texttt{smem}, trap, chk, addr, \texttt{list\_sval}) \\ with\ \texttt{list\_sval} & ::= & \texttt{Snil} \mid \texttt{Scons}(\texttt{sval}, \texttt{list\_sval}) \\ with\ \texttt{smem} & ::= & \texttt{Sinit} \mid \texttt{Sstore}(\texttt{smem}, chk, addr, \texttt{list\_sval}, \texttt{sval}) \end{array}$$

This type is mutually inductive, to also model lists of symbolic values (e.g. for arguments of operations) and memory (which is nothing more than a sequence of affectations—"Sstore"—over an initial state "Sinit"). Thus, a symbolic store contains the previous symbolic memory, the type of chunk and addressing mode (as in RTL), a list (of type list_sval) of symbolic arguments to compute the address, and the symbolic value (of type sval) to be stored. A symbolic value is either an input "Sinput" (i.e. the initial value of a register r in the path); an operation "Sop" with *op* the type of RTL operations (depending on the backend), a list of symbolic values (arguments), and a symbolic memory (of type smem); or a load "Sload", again containing symbolic memory and arguments, along with other information (trapping mode, chunk, and address) from RTL. Observe that the above type does not define final values: these are handled by a specific type, sfval (stands for "symbolic final value") which abstracts each type of final instruction (e.g. call, return, etc.)

A (mutually) inductive function of evaluation σ, with type "σ : (sval | list_sval | smem) → regset → mem → val option"[17], is then defined: any symbolic value can be evaluated, but the evaluation is not guaranteed (i.e. it can fail and return None). This function takes a symbolic value, and updates initial registers and memory states, before eventually returning a value if the evaluation succeeded.

SYMBOLIC STATES Internal states, as represented in Figure 4.6, contain the identifier of the current node (i.e. a positive natural number), a list of exit states, and the current local internal state. Such a complex decomposition is needed because the SE framework is built around the superblock structure, and expect side-exits to be in a specific format[18]. Furthermore, a symbolic state must represent the concrete output state w.r.t. to the concrete input state; and to simplify the proofs, it is preferable for this function to be deterministic. Hence the need of preserving the order of side-exits, and having a quite specialized data structure. At the end of SE, exit states of the source and target blocks are

[16] The stack frame simulation proof on BTL illustrates how delicate such a proof can be, see §6.1.

[17] Here, I voluntarily omit the global environment and stack pointer for the sake of simplicity.

[18] They must always be oriented the same (the "ifso" branch exits) and there must be at most one exiting condition per sistate (since the execution of the current block stops when the exit is true).

compared, and conditions (with their arguments) must match syntactically. Moreover, the verifier ensures that the "ifso" branch points to a valid superblock in the CFG.

This choice of modelling finely the superblock structure was certainly made to simplify the proof work.

The full notion of symbolic state is obtained by combining an internal state with a final symbolic value: $\mathtt{sstate} \triangleq \{\,\mathtt{internal}:\ \mathit{sistate};\ \mathtt{final}:\ \mathit{sfval}\,\}$.

Each type of state comes with its own semantics properties: either it is correct, and a specific property holds, or something went wrong, and an "abort" property holds, indicating that we must stop the compilation[19]. I give below a formal definition for the local internal states correctness property, while the others are outlined informally:

- **Local internal states:** for an initial pair $(\mathtt{rs}_0, \mathtt{m}_0)$ and state "$\mathtt{s}:\ \mathtt{sistate\_local}$", the pair $(\mathtt{rs}, \mathtt{m})$ is a possible valid final state if and only if: "$\mathtt{ssem\_local} \triangleq \mathtt{s}.(\mathtt{si\_pre}) \wedge \sigma(\mathtt{s}.(\mathtt{si\_smem}), \mathtt{rs}_0, \mathtt{m}_0) = \mathtt{Some\ m} \wedge \forall(\mathtt{r}:\mathtt{reg}), \sigma(\mathtt{s}.(\mathtt{si\_sreg})\ \mathtt{r}, \mathtt{rs}_0, \mathtt{m}_0) = \mathtt{Some\ rs\#r}$". Intuitively, it means that a state transition is valid if the precondition still holds, and if both memory and registers evaluate correctly.
  If the precondition is false, or if either the memory or the registers' evaluation fails (the "forall" becomes a "there exists" in the abort property), then the semantics property does not hold, but the abort property does.

- **Exit (internal) states:** first, let us assume that side-exits of superblocks are always on the "ifso" branch (it is the case in RTLpath, so that "reversed" conditions have been swapped during block selection). The semantics of an exit internal state is defined only for exits whose condition evaluates to "`Some true`"; in that case, for an exit state $\mathtt{s}_{\mathtt{exit}}$ branching to pc, "$\mathtt{ssem\_local}\ \mathtt{s}_{\mathtt{exit}}\ \mathtt{rs}_0\ \mathtt{m}_0\ \mathtt{rs}\ \mathtt{m}$" must hold, and $\mathtt{s}_{\mathtt{exit}}.(\mathtt{si\_ifso})$ must be equal to pc.
  When the condition fails to evaluate, or when the abort property of the internal local state $\mathtt{s}_{\mathtt{exit}}.(\mathtt{si\_elocal})$ holds, the abort property of the exit state holds.

- **Internal states:** if no early exit was taken, then all exits must evaluate to "`Some false`" (i.e. execution fall through), and the local internal state semantics property must hold. Otherwise, only the exit up to the taken one must be in "fall through", and the taken early exit semantics property must hold.
  There are thus two possible failure cases: either no early exit was met, but the execution aborted on a local state, or the evaluation of one early exit aborted.

Final symbolic values also have a semantics definition (which is linked to the concrete semantics of RTL). Global symbolic states sstate still follow the superblock structure in their semantics split in two cases:

- **An early exit semantics:** if the code took a side-exit, then the sstate semantics is reduced to the internal state semantics;

- **A final state semantics:** combining both the internal state semantics and the one of the final value sfval.

The theory provides two proofs about the SE semantics: a correctness proof, stating that *each concrete execution can be executed on the symbolic state—i.e., the SE is a correct over-approximation*; and an exactness proof, stating that *each execution of a symbolic state represents a concrete execution—i.e., the SE is exact*.

MAIN SIMULATION PROPERTY The main objective in such a framework is to prove that when the SE succeeds and returns without error, then a property linking the source and target RTLpath programs is valid. In the case of RTLpath, this property is defined using the SE theory. In contrast, the final correctness theorem must be defined after the whole implementation, since it assumes (as a hypothesis) that the execution returned without errors.

Definition 4.4.2 (Simulation property of RTLpath)**.** Let $\mathtt{f}_{\mathtt{src}}$ be the CFG of the source function, and let $\mathtt{pc}_{\mathtt{src}}$ and $\mathtt{pc}_{\mathtt{tgt}}$ the entry points of the source and target functions, respectively. The symbolic

[19] The correctness and the abort properties are mutually exclusive.

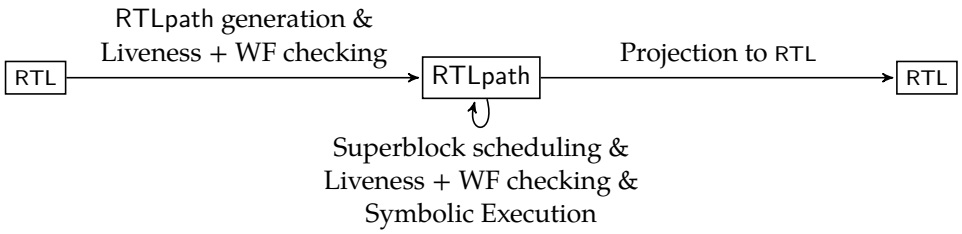


Figure 4.7: Architecture of the RTLpath Framework.

execution function is noted "$\xi : \mathtt{function} \to \mathtt{pc} \to \mathtt{sstate\ option}$", and symbolic states with letter δ. The simulation correctness property states that:

$$\forall \delta_{src}, \xi(f_{src}, pc_{src}) = \mathtt{Some}\ \delta_{src} \implies$$
$$\exists \delta_{tgt}, \xi(f_{tgt}, pc_{tgt}) = \mathtt{Some}\ \delta_{tgt} \wedge \forall ctx, \delta_{tgt} \simeq_{ctx} \delta src$$

Where the "$\simeq$" operator indicates that symbolic states are equivalent modulo liveness of the source on side-exits, and have the same (syntactically) final instruction.

This simulation property is not symmetric (as we saw in the example of §4.4.3), and must be valid under any *context* of execution (the notion of context is further detailed in §6.3.1).

4.4.4.2 *Formally Verified Integration Within Chamois-CompCert*

The liveness and WF properties are in fact verified together (by the same Coq checker).

Figure 4.7 schematizes how RTLpath was integrated into the Chamois-CompCert compiler: the RTL program is first *decorated* into a RTLpath program, pass during which an external oracle selects superblocks, and analyzes live variables. The liveness information and well-formedness (WF) are verified as soon as the block selection oracle returns. Once in RTLpath, the code is passed through another oracle for scheduling; at this point, the oracle potentially changes the path map and the entry point of the RTLpath function[20]. Because the path map has been modified, the WF check must be *run again*[21]; if it is still well-formed, the semantic equivalence is verified using SE. Finally, if the verifier did not abort, the code is projected back to RTL.

My goal here is to give an overview of the global proof architecture, this is why I mixed both simulations and SE in Figure 4.8.

Each RTLpath related pass has its own simulation proof, and everything is linked together in the main compiler proof. A (simplified) overall diagram about RTLpath proofs is provided in Figure 4.8. "Translations"[22] passes are at the extremities of the figure, and correspond to a standard *simulation diagram* (Figure 3.2). First, the RTL program P_1 is decorated into a RTLpath program P_2, following a star simulation on the left side starting from initial states S_1 and T_1. The RTL program moves forward while the RTLpath one *stutters*, and this until the *last step* where the RTLpath program makes a single big-step to reach its final state T_2 equivalent to the RTL one T_1. Likewise, when coming back to RTL, the RTLpath program P_3 only makes a single step from S_3 to S_4, whereas the RTL program P_4 advances of several steps from T_3 to T_4; so we have a plus simulation on the right side. Indeed, as RTLpath works with a big-step semantics, a single RTLpath step corresponds to many RTL steps; so the simulation needs to stutter on the RTLpath side (the internal RTLpath state is in fact progressing, but the visible RTLpath big-step needs to wait for the RTL steps to terminate). Note that the SE bisimulates (by combining correctness and exactness theorems) the RTLpath semantics. For the sake of simplicity, I represented the SE $\leftrightarrow$ RTLpath transitions with red edges. Those labeled "concretizes" represent the concretization of symbolic states δ_i to RTLpath states, and the symbolic simulation itself is performed on edges labeled with the ξ function (for both the source program P_2 and the scheduled program P_3). In other words, program P_2 (obtained by decorating P_1) is given as an input to the scheduling oracle, yielding P_3. The optimized program P_3 is compared by symbolic

[20] The oracle builds a *reverse mapping* from new path entries to old ones, to communicate its changes to the Coq side (a Coq function checks that the pairs of this mapping point to paths of their CFG).

[21] Here, only the WF check is needed, but since liveness and WF properties are verified together, the liveness is (uselessly) re-checked.

[22] I put in quotes the "translations" name because the RTLpath generation is, as shown, closer to a *decoration* (or encapsulation) of RTL than to an actual translation. Similarly, "translating" back to RTL is simply the process of *forgetting* superblocks.

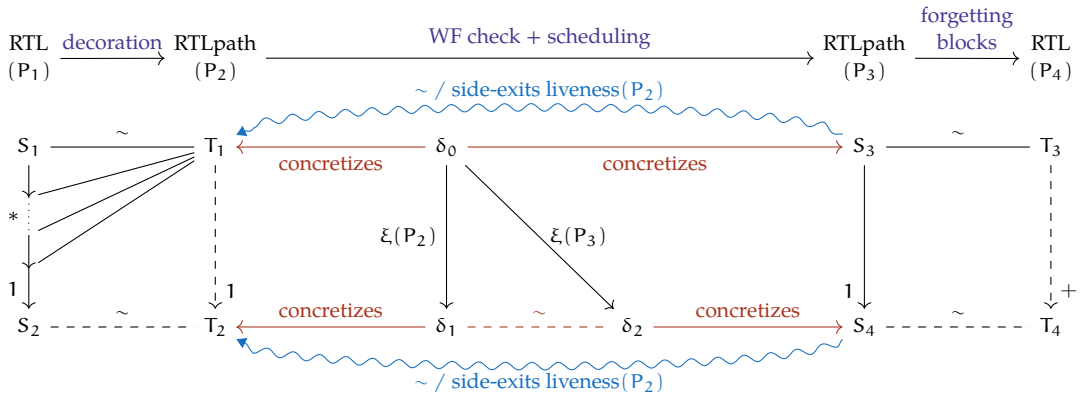


Figure 4.8: Coarse Overview of the RTLpath Proof Diagram.

execution to the source one. Since the SE only considers live registers from the source on side-exits (and considers all registers on the final exit), the concrete states T_1 and T_2 are respectively simulated by states S_3 and S_4 over the live variables of P_2's side-exits (and over all variables on the final exit). Figure 4.8 expresses this—incomplete on the final exit—modulo liveness relation by the wavy lines in light blue.

4.4.5 *Limitations of the Original RTLpath*

The RTLpath representation outlined until here corresponds to the original version implemented by Six [133]. In this section, I list the limitations of this approach as a transition to the preliminary contributions of §4.5, and to motivate the work of Part ii.

Firstly, as quickly mentioned in §4.4.2, the original RTLpath liveness handling poses several problems:

1. The simulation is modulo liveness only for side-exit states (the idea was to support the reordering of side-exits) but is *strict* over final states. This prevents us to introduce fresh variables. One of my objective was to leverage normalized rewrites to perform instruction selection (or, more generally, strength-reduction), but doing so often requires new variables. For instance, replacing a single instruction by a sequence of three operations will be easier if we can store intermediate results in new variables. Furthermore, optimizations such as common subexpression elimination often need such auxiliary variables to remember a previous computation. A real modulo liveness simulation is both more efficient, since it reduces the number of variables to compare; and more expressive, as assignments of non-live registers are ignored. *Generalizing the modulo liveness comparison* was thus one of my first preliminary contribution (see §4.5.1).

2. The liveness information considered, even if we extend the modulo liveness comparison to final exits, is still the one of the source program. However, the target liveness is *more expressive*: it validates "for free" a DCE. In fact, if a variable of the source program is dead, the oracle (which provides the liveness information to be verified) just have to omit the dead variables, so that if the validation succeeds, dead code removal is validated for free. The new IR I present in Part ii takes advantage of this opportunity.

3. Liveness is verified with a separated algorithm, while merging it with symbolic execution would be *more efficient* in terms of execution time. Nonetheless, this "preprocessing" verifier algorithm (see [133, §5.3.2]) is not dedicated to liveness, as it also ensures a well-foundedness condition over paths. When it returns without errors, we have a proof that the `wellformed_path_map` property of §4.4.1 holds. My preliminary contribution to solve problem 1. above is notably based on a modification of this algorithm. In Part ii, I go further, and I completely *integrate* liveness verification within symbolic execution.

Those issues about liveness are mostly restrictive on the class of supported transformations (only the third one above is also performance related), but they are not the only obstacles to circumvent in order to widen this class. I suggest below a list of some other (but at least as important) limitations:

1. **Cannot perform if-lifting:** the idea of creating a decorated IR based on RTL, resulting in a "forgetful functor" seems very smart, and, considering the difficulty of such an implementation when starting from scratch, it was probably necessary. Nevertheless, the *path structure* of the IR, and, more importantly, the SE verifier's level of specialization forces us to always keep the same structure, which sometimes proves to be a hindrance. A typical example, described in greater details in §11.3.1, is what we call the "if-lifting": the idea is to allow the superblock scheduler to *lift side-exits* in superblocks, to optimize their overall makespan. Except that moving an instruction below a condition (either in the "ifso" or in the else branch) might not be possible without inserting *compensation code* in the other branch. Indeed, if the variable written by the instruction is live on the other branch, we must also compute it there. The subtlety here comes from the structure of blocks: if the branch needing compensation is a side-exit, then inserting the actual code inside would break the superblock structure (the side-exit would no longer be just an exit)! This is, of course, not supported by the RTLpath symbolic verifier because of its dependence on the (super)block structure.

   Justus Fasse [75] proposed an experimental solution based on an intricate sequence of passes, combining ad-hoc validators. The more general representation proposed in Part ii is able to support a much simpler implementation, realized by Alexandre Bérard [4].

2. **Inflexible block structure in general:** this point is closely related to the previous one, but with other use-cases. Indeed, every optimization on RTLpath must be compatible with the (either basic or super) block structure. In particular, a scheduler working on larger blocks would be unsupported, and all optimizations are limited to a single superblock. This is limiting not only when a transformation changes the structure, but also for any extended block based algorithm.

3. **Even more generally, no global optimizations:** as oracles are only authorized to optimize locally inside blocks, the framework is unable to validate global transformations (i.e. over the whole function). Yet, common subexpression elimination and code motion algorithms are important optimizations for an efficient compilation, and operate globally. Of course, symbolically executing the entire function (containing calls, loops, etc.) would be extremely difficult (surely impossible as it is, and certainly too complex algorithmically), and this especially in the CompCert semantics. I demonstrate with block transfer language (see Part ii) that "synchronizing" information between blocks is possible, and helpful to reach this goal.

4. **A not very practical rewriting system:** the rewriting mechanism implemented in RTLpath is sufficient for a few simplification rules, but becomes difficult to use when there are many and complex rules used for a specific optimization oracle. This drawback is not a major concern, but it would be better to create a more modular rewriting engine. Precisely, I propose in the BTL IR to use configurable sets of normalization rules for specific oracles (see §7.6).

4.5 IMPROVING RTLPATH†

This section aims to serve as a transition to the next part, by introducing the first (preliminary) contributions I made over RTLpath:

- **Liveness improvement (§4.5.1):** I generalized the modulo liveness simulation to whole superblocks and not only side-exits;
- **Normalization rules for RISC-V (§4.5.2):** I lifted some macro-expansions that were done at the Asm level to RTLpath in order to increase scheduling opportunities (on RISC-V only).

In Part ii, we replace RTLpath with a new IR, and the above listed contributions thus become obsolete. A frozen version of RTLpath *with my contributions* is available on a dedicated branch [◇].

4.5.1 *A Full "Modulo Liveness" Comparison*

The decoration from RTL to RTLpath is defined as a specific Coq pass, which calls an untrusted oracle. The resulting RTLpath code and its attached liveness information are a posteriori verified by a dedicated checker introduced in §4.5.1.1. Originally, the oracle was only returning the liveness information at the entry of each block, and the verification was limited to this information. Section 4.5.1.2 details why this information is not enough to obtain a modulo liveness SE, and §4.5.1.3 how I modified the framework to extend this liveness checking at superblocks' end points.

4.5.1.1 *Preliminary Presentation of the "Input" Liveness Check*

Recall the definition of a RTLpath function from §4.4.1: the only field which must be generated is the path map, since the well-formedness properties are obtained by the verification, and the only other field is the RTL function (which is not modified by the RTLpath generation). The `path_info` record does not contain any RTL instructions, since they are already accessible in the `fn_RTL` field of a function. The generation is thus simply the process of building a RTLpath function record, with a new path map, and ensuring the well-formedness conditions. Therefore, the oracle has the following signature:

```
Axiom build_path_map: RTL.function → path_map
```

For each selected superblock, the oracle fills the path map with an information record about the path. The `path_info.output_regs` field is only used by the untrusted scheduler, but has no semantics in the Coq part (i.e. it does not need to be formally verified). The `pre_output_regs` field is what I use to generalize the liveness (see §4.5.1.3); so we are left with the path size (used in the well-formedness condition) and the `input_regs` set. The latter set contains, for each path, the list of live registers (a.k.a. input registers) at the beginning of the path.

Let `f` be the original RTL function and `pm` the returned path map. The first step is trivial: ensuring the `path_entry` property, by checking if "`pm!(f.(fn_entrypoint)) <> None`". The second part of the verification applies on every pair `(pc, pi)` in `pm`, and operates in two steps:

1. A decreasing loop starts from the path size (i.e. `psize`) to zero (and since `psize` is filled with the number of instructions in the path minus one, the loop stops just before the final instruction), and maintains an "alive" regset, whose initial value is the `input_regs` field. For each instruction (except the final one), the liveness is verified w.r.t. the current "alive" set (i.e. the instruction arguments *must* be live), and the "alive" set is updated accordingly (i.e. the instruction destination *becomes* live). If the instruction is a side-exit to a new superblock, the function checks if the target superblock is well-defined in the path map, and if its `input_regs` field is a subset of the current "alive" set. If everything is fine, it returns a pair with the current "alive" set, and the map address (in the original RTL code) of the last instruction in the path. Otherwise, the compilation is aborted.

2. The last instruction (whose address in the CFG is returned by step 1.) is retrieved. If it is a terminal instruction[23], the function simply checks that the arguments are alive in the set (also returned by step 1.); else, the same check as for side-exits is performed: verifying if the `input_regs` field of the next superblock is included in the "alive" set updated with the last instruction.

There is finally a theorem proving that if the two checks above were executed successfully, then the well-formedness conditions hold, *and* the liveness information is correct.

A predicate "`liveness_ok_function`" remembering the validity of the path checker for each path in the map is also saved for later, in order to prove the SE correctness (which assumes, of course, a correct liveness information).

4.5.1.2 *Why Does Not This Achieve Our Goal?*

From the details provided in the previous paragraph, it could be presumed that the necessary liveness information is already available to generalize the simulation at the final exits of superblocks.

[23]Here, "terminal" means without any successor (e.g. returns or tail calls).

While the actual verification would enable the validation of the "output_regs" field, it is not exactly what we want.

Indeed, with the "alive" set computed earlier and the inclusion test with the input set of the next block (if it exists), it may seem feasible to enhance the simulation test. Actually, there is a subtlety here: the "alive" set obtained after updating with the last instruction (step 2. of §4.5.1.1), is the *complete* set of *output* variables; in other words, this set is equivalent to the union of the input sets of each superblock in the successors of the block's last instruction[24]. The problem is that taking this output set for our modulo liveness simulation would be inaccurate when the last instruction is *a call or a built-in*. These external functions do not share any registers with the caller, and renaming their result register is prohibited. Keeping their result registers in the output set would work, but would induce *unnecessary comparisons* of symbolic values. By the way, the actual verifier already checks, independently of the liveness information, that result registers remain unchanged.

Therefore, the idea of the output set above goes in the right direction, but should be slightly adapted to fit with the real semantics. Precisely, the information we need is the one of this output set, except for calls and built-ins, where we should remove the result register from the output set. Put another way, the set we need is exactly the output set when the final instruction is neither a call nor a built-in, and the output set minus the result register of the final instruction otherwise.

Fortunately, computing this set from the oracle is trivial. The liveness analysis for input sets is computed by a data-flow analysis (i.e. fixpoint), and the output sets by the union of the successors' inputs. To obtain the right output set, we simply have to apply one more time the data-flow transfer function on the last instruction, if the latter is neither a basic instruction nor a conditional branch. In our formalism, we name this set "pre-output regs", since it is "almost" the complete output set.

4.5.1.3 *Generalizing the Liveness Check*

To summarize, two changes need to be realized in order to have a modulo liveness analysis: (i) *a posteriori verifying* this pre-output set; and (ii) *adapting* the simulation test and theory to compare final states modulo this new liveness information.

MODIFYING THE RTL → RTLPATH ORACLE Rather than computing and returning the output set (which is only used by the scheduler but not on the Coq side) as below:

```
let outputs = List.fold_left Regset.union Regset.empty list_input_regs in outputs
```

we apply again the transfer function [⋄].

```
let outputs = List.fold_left Regset.union Regset.empty list_input_regs in
let por = match last_instruction with
  | Icall (_, _, _, res, _) -> Regset.remove res outputs
  | Ibuiltin (_, _, res, _) ->  Liveness.reg_list_dead (AST.params_of_builtin_res res)
      outputs
  | Itailcall (_, _, _) | Ireturn _ ->
     assert (outputs = Regset.empty); (* defensive check for performance *)
     outputs
  | _ -> outputs
in (por, outputs)
```

Observe that while the verifier adds the result register of instructions to the "alive" set (by moving forward through the code), the oracle's analysis does the opposite. Above, the result register of calls and built-ins is *removed* from the output set (and the liveness analysis fixpoint is run backward).

ADAPTING THE COQ PART IN CONSEQUENCE Concerning the Coq part, the modifications are a bit more complex. For the liveness verifier, we only modify the *second step* of the algorithm outlined in §4.5.1.1. For non-terminal instructions, instead of performing the update and checking the inclusion with the "alive" set, it is done using the *pre-output* set. Additionally, we verify that the pre-output set is a subset of the final "alive" set, no matter the type of the superblock's final instruction.

We need to prove that our pre-output register set is correct; intuitively, this corresponds to the proof that concrete states obtained after the SE of the last instruction (starting from the same initial

[24] In practice, the only terminal instructions with multiple successors are conditionals and jump tables.

regset and memory) are equivalent no matter the final (output) regset (but keeping the same final memory), *as long as those regsets are equivalent modulo liveness*. Technically, the link between symbolic and concrete states is made by an inductive predicate. Under the program global environment (since we need to define a semantics for calls), and given a stack and a stack pointer, the program counter, the RTLpath function, and the initial regset and memory states, the predicate defines the new regset and memory obtained by executing the last instruction. It also gives the observable trace and the concrete RTLpath state resulting from this final step, as formalized below:

Definition 4.5.1 (Semantics of final symbolic values on concrete states [◇])**.** Let G the global environment, and sp the stack pointer. We have:
ssem_final G sp pc stack f rs0 m0 : sfval → regset → mem → trace → state → Prop

We now reuse the above predicate to prove our set of live variables correct:

Theorem 4.5.1 (Pre-output register set is correct [◇])**.** *Our theorem assumes a correct liveness check (i.e. that the* liveness_ok_function *predicate holds), an existing path* p *located at* pc *in the CFG of a function* f*, and a state* δ *resulting from the SE of* p *(here, SE is assumed to succeed, not to be correct). For all global environment* G*, stack pointer* sp*, stack* stk*, initial and final regsets* rs0 *and* rs1*, initial and final memories* m0 *and* m1*, trace* e *(recall the formalism of §3.2.1), and concrete state* s*, we have:*

$$
\begin{aligned}
&\text{liveness\_ok\_function f} \implies \\
&\text{f.(fn\_path)!pc} = \text{Some p} \implies \\
&\xi(\text{f}, \text{pc}) = \text{Some } \delta \implies \\
&\text{ssem\_final G sp } \delta.(\text{internal.(si\_pc)}) \text{ stk f rs0 m0 } \delta.(\text{final}) \text{ rs1 m1 e s} \implies \\
&\text{rs1} \equiv_{\text{por}} \text{rs1}' \implies \\
&\exists \text{s}',\ \text{ssem\_final G sp } \delta.(\text{internal.(si\_pc)}) \text{ stk f rs0 m0 } \delta.(\text{final}) \text{ rs1}' \text{ m1 e s}' \wedge \\
&\qquad \text{s} \equiv_{\text{input\_regs}} \text{s}'
\end{aligned}
$$

Where the notation $\equiv_{\text{E}}$ *means equivalent over the domain of variables of* E *(i.e. it is our modulo liveness comparison)*[25]*. In the above, we state that if the* ssem_final *predicate holds for a final regset* rs1 *and with concrete state* s*, and if there is another regset* rs1′ *equivalent to* rs1 *modulo liveness, then there exists a concrete state* s′ *such that the predicate still holds for* rs1′ *with state* s′*, and that concrete states* s *and* s′ *are equivalent modulo liveness.*

Be aware that the register sets equivalence is checked over our computed pre-output set, while the concrete state equivalence is over the input registers of the path.

I proved Theorem 4.5.1 in Coq by case analysis on final symbolic values, along with additional intermediate lemmas that I will not detail here. Comparing to the Coq implementation, I voluntarily omitted some details to facilitate understanding: for instance, we have a supplementary hypothesis on stack frames since the equivalence modulo liveness on concrete states is based on the RTL state's semantics of §3.4.1.

Finally, I modified the SE strict comparison for final exits into a modulo liveness one in both the theory and the refined (implementation) parts. This is only a slight change, since the modulo liveness comparison on hashed symbolic register sets was already defined and proved (for side-exits).

While implementing this complete liveness verifier, I took the opportunity to add a separated well-formedness check, so that the verification performed after the scheduling only focuses on WF and avoids the redundant liveness check mentioned (in footnote) in §4.4.4.2.

4.5.2 *RISC-V Macro-Expansions at the RTL Level**

On the CompCert RISC-V backend, some RTL level instructions are macros to be *expanded* at the Asm level (during the Mach to Asm translation, see Figure 3.1). A macro is an abstract instruction, that may be turned into a *sequence* of real instructions, depending on the arguments. On RISC-V, these macros are always related to immediate constants, as the immediate value might facilitate (or

[25] The interested reader can find its definition here [◇] for register sets and here [◇] for concrete states.

*The example (4.5.1) and some sentences of this section are reused from our CPP publication [135]†.

complicate) the code generation. For example, an addition between a register and an immediate might be translated as either:

- An immediate add (the normal case) translated as the RISC-V "`addi`" instruction;
- An immediate load of the high part of the source immediate (with the "`lui`" instruction), whose destination is added with the low part (as an immediate, again with "`addi`"). The whole result is then added with the source register, so this last addition is performed between two registers (instruction "`add`");
- An immediate load of the (whole) source immediate in a register (using "`ld`" from a literal label), itself added with the source register ("`add`" again).

I ported the expansion mechanism to BTL. The associated limitations and implementation challenges in both RTLpath and BTL are discussed in §7.6.1.

Applying these expansions earlier—typically, at the RTL level—is better in terms of performance, for at least two reasons: (i) the scheduling will be more efficient, as it will precisely know the latency of these sequences (rather than having an estimation depending on how the macro will be expanded); and (ii) the expanded code will be optimized by redundancy elimination, allowing to eliminate some duplicated instructions from expansions. A typical example of redundancies is the loading of constants: many macro-expansions first load an immediate constant in a register, and a simple common subexpression elimination pass would suffice to simplify the generated code.

Moreover, CompCert also has similar macros for conditional branches: for instance, immediate long comparisons require loading the immediate beforehand, but this step can (and should) be skipped if the immediate value is zero. Indeed, the RISC-V architecture features a specific register "x_0" whose value is always zero, but it does *not* feature immediate branching instructions. Hence, if the immediate to compare with is zero, the generated Asm should directly compare with x_0; otherwise, the immediate is loaded in a scratch (i.e. temporary) register, by adding it with the x_0 register.

4.5.2.1 *A Rewriting Preprocessor*

To illustrate the rewriting mechanism, we simply have to take an example involving *immediates* (i.e. "hard-coded" constants). From the source code on the left column below, I compare the code compiled with the mainline version of CompCert (3.12) (in the middle) and our version with *normalized rewrites* (on the right); both in RISC-V 64-bit assembly. We omit the prologue and epilogue for the sake of simplicity.

Example 4.5.1 (Expanding instructions in the scheduler's preprocessor)**.**

```
if (x + *t < 7)
  if (y < 7)
    return 421;
```

```
lw x7,0(x12)
; x7 MAY STALL
addw x6,x10,x7
addiw x31,x0,7
bge x6,x31,.L10
addiw x31,x0,7
bge x11,x31,.L10
```

```
lw x7, 0(x12)
addiw x12, x0, 7
addw x6, x10, x7
bge x6, x12, .L10
bge x11, x12, .L10
```

Both assembly codes represent a single superblock. Registers `x10`, `x11`, and `x12` respectively correspond to variables `x`, `y`, and `t` of the input program. Mainline CompCert does not feature scheduling, and the pipeline is not reordered. Thus, the load that dereferences variable `t` in `x7` may induce a pipeline stall, because `x7` may not be ready when executing the second instruction (the first add, on the left code). Loading a value from the memory often takes several cycles, and if the addition reading `x7` is *issued* directly after the load, the processor will have to wait. Moreover, conditional branch expansions (which—in mainline CompCert—happen during the Asm translation) load the value to compare by adding it with `x0`. The translation is naive, and *does not attempt to eliminate redundancies*, so when the second branch is translated, the immediate is unnecessarily loaded again in the scratch register `x31`.
The rewriting engine I proposed works as follows:

1. An untrusted rewriting algorithm is called in the preprocessing of the prepass scheduler, and expanses RTL instructions into sequences of special operations (specific to RISC-V, since RTL is

a parametrized IR). These new operations do not need to be expanded when translating to Asm since they already represent "real" operations of the target instruction set architecture. The oracle performs an expansion of comparisons with an immediate (branching or not), and of some other instructions (arithmetic operations on immediates, casts, loads of constants, and length conversions). Intermediate values generated by expansions are stored into *fresh pseudo-registers*, and the untrusted preprocessor uses a dynamic value numbering system to avoid redundant instructions. The latter is a CSE limited to the superblock's scope.

2. The prepass scheduler is called normally: thanks to the previous expansions, the scheduling has more opportunities.

3. The transformed (expanded & scheduled) code is returned to the SE verifier to ensure correctness.

Thanks to this light preprocessing pass of the scheduler, CHAMOIS-COMPCERT produces the code on the right side above. There are two improvements in this result: first, the expansion oracle allocates the load of immediate `7` inside a fresh variable `x12` (rather than the scratch register `x31`), so that `x12` is reused for both conditional instructions; second, the scheduler *interleaves* the immediate load with the addition into `x6` to avoid the stall we observed on the unoptimized code. Each of these changes potentially saves one cycle, so the resulting code has a potential gain of two cycles (we saved one instruction with the former, and avoided a stall with the latter). In the general case, because the scope of our preprocessing is limited to a superblock, this memoization of immediates should only have a limited impact on register pressure.

Furthermore, our rewriting oracle adds only a *light overhead* since (exactly like in §4.2.3) it does not require an additional SE run: both the preprocessor and the scheduler are applied sequentially, and verified in *a single run of SE*.

4.5.2.2 *Rewriting Symbolic Values On-The-Fly*

We validate expansions of Example 4.5.1 during SE using a single normalization rule:

$$\text{“}(\texttt{LTs}\ \texttt{i})[v] \rightsquigarrow \texttt{LT}[v; \texttt{Sop}(\texttt{0addiwx0}\ \texttt{i}, [\,], \texttt{m})]\text{”}$$

This implies that conditionals in the form $v < \texttt{i}$, where $\texttt{i}$ an immediate and v a symbolic variable, get normalized into $v < (x_0 + \texttt{i})$, with x_0 being the always zero register. In the above rule, "LTs" stands for less than (immediate), with suffix s to indicate that it is not yet expanded. The constructor directly includes the immediate, and (between brackets) the list—here of one element—of symbolic values to compare with. The rewrite turns condition "LTs" into "LT", a different operator on which the rule no longer applies. The symbolic memory $\texttt{m}$ of the newly generated symbolic operation (`Sop`) represents the memory state as it is when the execution reaches the conditional branch. Variable v can be instantiated with any symbolic value. Then, every time the SE reaches a condition of this type, it will try to apply the rule.

Let us apply this principle to our example. Without the normalization rule, the SE outcome of the code on the left side of Example 4.5.1, would be (in pseudocode with symbolic values)[26]:

$$\begin{array}{l}\texttt{if}\ (\texttt{LTs}\ 7)[\texttt{Sop}(\texttt{0add}, [\texttt{Sinput}(x_{10});\ \texttt{Sload}(m_0, ..., [\texttt{Sinput}(x_{12})])], m_0)] \\ \quad \texttt{if}\ (\texttt{LTs}\ 7)[\texttt{Sinput}(x_{11})] \\ \quad\quad \texttt{Sreturn}(...)\end{array}$$

The first condition's single argument contains the symbolic value associated with x_6, that is the result of the addition between x_{10} (variable `x` in the source) and the load dereferencing x_{12} (variable `t` in the source). The second condition compares the same immediate value, but this time with variable x_{11} (variable `y` in the source), which has remained unmodified since the block's entry (`Sinput`).

[26] Normally (cf. footnote of §4.4.4.1), early exits are reoriented in RTLpath. For the sake of simplicity, we write them in the same direction as in the source here (rewrites apply symmetrically).

With the implementation of our normalization rule, the SE outcome would be:

$$\begin{array}{l}\texttt{if LT}[\texttt{Sop}(\texttt{0add}, [\texttt{Sinput}(x_{10});\ \texttt{Sload}(m_0, ..., [\texttt{Sinput}(x_{12})])], m_0);\\ \quad \texttt{Sop}(\texttt{0addiwx0}\ 7, [], m_0)]\\ \quad \texttt{if LT}[\texttt{Sinput}(x_{11}); \texttt{Sop}(\texttt{0addiwx0}\ 7, [], m_0)]\\ \quad\quad \texttt{Sreturn}(...)\end{array}$$

As expected, our rule on conditions transforms the SE result of the source, such that it aligns precisely with the outcome we would obtain by symbolically executing the rewritten code (i.e. right side of Example 4.5.1). The symbolic trees of both the (normalized) source and target programs are syntactically identical, indicating that we have successfully validated our rewriting example through the simulation test.

Please note that we prefer to refer to these rules as normalization rules rather than rewriting rules, as they are specifically designed to directly transform symbolic values into their normal forms (i.e. that are never rewritten again in the subsequent SE).

In contrast to the peephole of §4.2.3, where normalization is applied on the transformed code (i.e. reverting the oracle's changes) during the translation to AbstractBasicBlock; the RTLpath rewriting engine operates on the source, and this during symbolic execution. Thus, RTLpath normalizations reproduce the oracle's rewrites.

Verifying that the untrusted preprocessor's rewrites correctly handle fresh registers is just a *particular case* of the simulation test modulo liveness: the expansion on the right-hand side of Example 4.5.1 is correct because `x12` is not live at label `.L10`. Embedding the normalization rules within SE allows the proof of these rules to ignore liveness issues. In contrast, expressed in a preprocessing of the simulation test as in the postpass, normalization rules would need to satisfy a *bisimulation modulo register liveness*.

Another interest of applying rules directly on symbolic values is to alleviate their proofs of correctness. Indeed, the inductive structure of symbolic values allows expressing an entire sequence of operations in a single symbolic term. Hence, proving a normalization rule simply reduces to showing that both sides evaluate to the same value for all inputs.

4.6 CONTRIBUTIONS & CONCLUSION

The work of Six [133] on the KVX backend, and its generalization to prepass over a decorated RTL representation was an important experiment to deeply understand the technique, and in particular its limitations. I list below the preliminary contributions presented in this chapter, along with a key-points summary.

Contributions:

1. A port of the formally verified postpass validator from KVX to AArch64. This SE framework at the Asm level validates a postpass scheduling and an improved peephole optimization, targeting in particular the Cortex-A53 in-order core.

   See §4.3 for information on development size.

2. A complete modulo-liveness symbolic simulation: I extended the liveness checker originally used in RTLpath to support the modulo liveness comparison on final states (at superblocks' end points). This contribution enabled the verification of rewriting optimizations, notably through the possibility of introducing fresh variables with oracles.

3. Directly using contribution 2., I added a rewriting engine to RTLpath, able to process conditions and operations. I use this engine to lift some late macro-expansions on the RISC-V backend, in order to increase the scheduler's performance and produce a less redundant assembly code.

Points 2. and 3. are smaller contributions compared to the postpass. The longest part is the rewriting engine, whose code is reused in our new IR, BTL. See §9.1 for an idea of its size directly in BTL.

Key points to remember from this chapter:

1. Optimizing at the Asm level is often more accurate, but requires an important amount of work to be adapted on several architectures;
2. Symbolic execution is an efficient method to validate scheduling, peephole, and expansion optimizations;
3. The scaling problem can be solved by using hash-consing;
4. The decoration approach raised several research questions:
   - Could we extend this approach to larger forms of blocks?
   - Can we generalize the rewriting engine used for expansions to other applications?
   - Is there a way of using SE to verify inter-block (i.e. global) optimizations?
   - Can we integrate the liveness check within the SE, and adapt it to verify the target program's liveness instead of the source's one?
5. Starting with a specific approach like RTLpath was certainly a necessary step before engaging in further work.

Part II

BLOCK TRANSFER LANGUAGE

This part delves into the primary contribution of my PhD thesis: our new BTL intermediate representation and its symbolic simulation test. This contribution was presented in [65]†. Here, I give much more detail on the Coq development. This part is intended to be a comprehensive reference documentation for my Coq code. It aims to help readers interested in building upon or extending this work. My goal was not to merely paraphrase the code, which is available online, but rather to explain its detailed design and the choices we made.

Adopting a "backward style" of presentation, I start with the main ideas, followed by technical details. Those unfamiliar with Coq or CompCert might find it helpful to initially consult the [65]† article to avoid being overwhelmed by the Coq formalism.

The aforementioned paper also addresses extensions of BTL and its validator, contributed by Benjamin Bonneau. These are not in my online code or in this part to clearly delineate my contributions. Bonneau's extensions get a brief introduction in Chapter 11 and can be found in the Verimag Chamois-CompCert repository (refer to the start of Chapter 11). They are also incorporated in the experimental evaluation of Chapter 12.

The content is divided in five chapters:

- Chapter 5 presenting block transfer language;
- Chapter 6 on the formalization of our symbolic execution theory in Coq;
- Chapter 7 refining the aforementioned theory as a concrete and efficient implementation;
- Chapter 8 about the integration of BTL in the existing CompCert pipeline, and in particular on the translation from and back to RTL;
- Chapter 9 concludes on the BTL defensive validation framework.

5

A BLOCK-BASED INTERMEDIATE REPRESENTION†

Block transfer language (BTL) is a new intermediate representation close to RTL, and dedicated to defensive certification of middle-end optimizations (before register allocation). Its main feature is a *syntactical representation* of blocks: informally, a block is a fragment of *loop-free* code, with a single entry point. A BTL program partitions a RTL program into such blocks (in opposition to RTLpath whose paths were potentially overlapping). Each block is run in one (big-)step, emitting at most *a single observational event*. The local optimizations (i.e. preserving locally the semantics of such blocks) are checked by symbolic execution with normalization rules. Global optimizations (i.e. preserving globally the semantics but not always locally) require invariants annotations at each block entry. BTL is specifically designed for symbolic execution modulo invariants, and gives a structured view of RTL code.

In Part ii, we always refer to such big-step semantics for blocks as "blockstep".

In this chapter, I introduce more formally the notion of invariants with an example in Section 5.1, and I detail the BTL abstract syntax and semantics in Sections 5.2 and 5.3.

5.1 A GLOBAL SIMULATION EXAMPLE*

We saw in Chapter 4 how symbolic simulation was effective in validating liveness based *intra-block* optimizations. Actually, liveness is *global* information on the CFG. Thus, dead code elimination and superblock scheduling are *not* pure intra-block optimizations, but global ones. In order to validate more *inter-block* transformations, we generalize the live sets (associated to each block entry point) into invariants relating *source* registers to *target* ones. I progressively explain this idea, and formalize the reasoning given in the example of §1.2.2, with the help of the more complex Example 5.1.1, still providing a transformation on a C pseudocode.

Note that for now, we only consider arithmetic computations on `long`. This choice is clarified in §10.5.2.

Example 5.1.1 (Simulation modulo invariants to verify a simple strength-reduction)**.**

```
long main(long i, long n) {
  Bhead:
    long s = 0;
    long a = 7;
    goto Bbody;
  Bbody:
    if (i > n)
      return s;
    s += i * a;
    i += 3;
    goto Bhead;
}
```

```
long main(long i, long n) {
  Bhead:
    // 𝒢 : i ≔ i ‖ n ≔ n
    long s = 0;
    long i_a = i * 7;
    goto Bbody;
  Bbody:
    // ℋ : a ≔ 7
    // 𝒢 : i_a ≔ i × a ‖ s ≔ s ‖
    //     i ≔ i ‖ n ≔ n
    if (i > n)
      return s;
    s += i_a;
    i_a += 21;
    i += 3;
    goto Bbody;
}
```

Both the source and the target code are CFGs of two extended blocks, labeled by Bhead for the entry part and Bbody for the loop part. The target is obtained after a combination of *constant propagation* (from `a = 7`) and strength-reduction (SR)[1]: the multiplication originally within the loop is moved to Bhead and reduced in Bbody to an addition on a *fresh* register `i_a` (in red), and register `a` is eliminated. Substitutions and compensation code are colored green.

*This example is adapted from our OOPSLA'23 paper [65]†.

[1]The complete transformation of Example 5.1.1 is handled by the LCT algorithm of Chapter 10.

Our "gluing invariant" have similarities with the "simulation invariants" of Rinard and Marino [124], and our "history invariant" with their "standard invariants".

In order to prove the simulation block-by-block, one need two types of invariants:

A GLUING INVARIANT (GI): equalities between some target registers and terms over source registers[2] (e.g. an invariant `[x ≔ y * y]` means that the `x` variable of the target block has the (symbolic) value `y * y` interpreted in the source block). They are used to anticipate non-trapping computations, to remember already computed trapping calculus, and to eliminate dead code (i.e. because they encode a liveness information). The term "gluing invariant (GI)" is inspired from [3].

A HISTORY INVARIANT (HI): equalities between some source registers (that are eliminated in the target, such as "a" in Example 5.1.1) and terms over source registers. Indeed, using only GIs would not suffice to remember that a variable in the source has some value, while this is important for certain transformations like CSE. History invariants (HIs) thus aim to share a common execution past. In the case of a CSE pass, they would propagate information about an already computed value to replace.

On Example 5.1.1, invariants generated by our oracle are attached to each block of the target code (preceded by an orange double-slash). Instead of the sequential representation of Example 1.2.1 (e.g. with `[I1 ; I2 ; ... ]`), GI and HI are denoted as parallel assignments $\mathcal{G}$ and $\mathcal{H}$, respectively[3]. In our SE validator, we efficiently check these invariants by restricting them to be *preconditioned parallel assignments of symbolic expressions*, similar to those resulting of the symbolic executions of basic blocks in Example 2.2.1 (except that in Example 5.1.1, all preconditions are trivially true).

As in Example 1.2.1, the GI gives the target input registers of its associated block: they are those syntactically assigned by the invariant. In fact, the trivial (i.e. identity) assignments visible in these examples encode a *liveness* information: even if the variable is not modified, the identity assignment indicates that it is live. Semantically, such a pair of invariants represents a relation of the form "$\mathrm{H}(\mathrm{r}_s, \mathrm{m}) \wedge \mathrm{r}_t \equiv_t \mathrm{G}(\mathrm{r}_s, \mathrm{m})$" from a source state $(\mathrm{r}_s, \mathrm{m})$ to a target state $(\mathrm{r}_t, \mathrm{m})$ which keeps memory state m unchanged, but turns register state r_s into register state r_t. Here, "$\equiv_t$" is equality of registers sets only for target live registers (this explains the inclusion of the source register state devoid of dead variables into the target one, as claimed in §1.2.2). More generally, "$\equiv_E$" is equality of register sets over frame E. See Definition 6.4.9. In Example 5.1.1, the live (target) registers at label Bhead are the function parameters `i` and `n`, and the live registers at label Bbody are `i_a`, `s`, `i`, and `n`.

The symbolic validator performs the simulation modulo invariants similarly as in Example 1.2.1, since the CFG structures are identical, except that this time, we also consider HIs. Their verification is performed over the source state, by first (symbolically) executing the input history invariant, then the source block, and finally the output history invariant. Note that since HIs *replay* a summary of the past execution of the source, the input $\mathcal{H}$ must also be applied before the verification of both the source and the target states. Roughly speaking, it reduces to the comparisons (with invariants in brackets):

1. Bhead/Bbody $\mathcal{H}$: "`s = 0; a = 7; [a ≔ 7]`" (source Bhead + Bbody $\mathcal{H}$) simulates "`s = 0; a = 7`" (source Bhead alone) on register frame `{a}`;
2. Bhead + Bbody $\mathcal{G}$: "`[i ≔ i ‖ n ≔ n]; s = 0; i_a = i * 7`" (Bhead $\mathcal{G}$ + target Bhead) simulates "`s = 0; a = 7; [i_a ≔ i * a ‖ s ≔ s ‖ i ≔ i ‖ n ≔ n]`" (source Bhead + Bbody $\mathcal{G}$) on frame `{i_a,s,i,n}`;
3. Body/Body $\mathcal{H}$: "`[a ≔ 7]; s += i * a; i += 3; [a ≔ 7]`" (Bbody $\mathcal{H}$ + source Bbody + Bbody $\mathcal{H}$ again) simulates "`[a ≔ 7]; s += i * a; i += 3`" (Bbody $\mathcal{H}$ + source Bbody) on frame `{a}`;
4. Body/Bbody $\mathcal{G}$: "`[a ≔ 7]; [i_a ≔ i * a ‖ s ≔ s ‖ i ≔ i ‖ n ≔ n]; s += i_a; i_a += 21; i += 3`" (Bbody $\mathcal{H}$ + Bbody $\mathcal{G}$ + target Body) simulates "`[a ≔ 7]; s += i * a; i += 3; [i_a ≔ i * a ‖ s ≔ s ‖ i ≔ i ‖ n ≔ n]`" (Bbody $\mathcal{H}$ + source Bbody + Bbody $\mathcal{G}$) on frame `{i_a,s,i,n}`;
5. Bbody return: "`[a ≔ 7]; [i_a ≔ i * a ‖ s ≔ s ‖ i ≔ i ‖ n ≔ n]; return s`" (Bbody $\mathcal{H}$ + Bbody $\mathcal{G}$ + target Body exit) simulates "`[a ≔ 7]; return s`" (Bbody $\mathcal{H}$ + source Bbody exit).

[2] Those are the invariants presented in Example 1.2.1.

[3] Both representations are equivalent, see §6.2.2 for a more detailed comparison between them.

final	::=	`Bgoto`(pc_{succ})	≈ branch to $[pc_{\text{succ}}]$
	|	`Breturn`$(\epsilon \mid reg)$	function return♠
	|	`Bcall`$(sig, (reg \mid id), \overrightarrow{reg_{\text{arg}}}, reg_{\text{dst}}, pc_{\text{succ}})$	function call♠
	|	`Btailcall`$(sig, (reg \mid id), \overrightarrow{reg_{\text{arg}}})$	function call (not returning)♠
	|	`Bbuiltin`$(ef, \overrightarrow{reg_{\text{bargs}}}, reg_{\text{bdst}}, pc_{\text{succ}})$	compiler built-in♠
	|	`Bjumptable`$(reg_{\text{arg}}, \overrightarrow{pc_{\text{succ}}})$	"switch" jump♠
iblock	::=	`BF`$(final, iinfo)$	final instruction (of the block)
	|	`Bnop`$(\epsilon \mid iinfo)$	no-operation
	|	`Bop`$(op, \overrightarrow{reg_{\text{arg}}}, reg_{\text{dst}}, iinfo)$	normal operation
	|	`Bload`$(trap, chk, addr, \overrightarrow{reg_{\text{arg}}}, reg_{\text{dst}}, iinfo)$	memory load (trapping or not)
	|	`Bstore`$(chk, addr, \overrightarrow{reg_{\text{arg}}}, reg_{\text{src}}, iinfo)$	memory store
	|	`Bseq`$(iblock_1, iblock_2)$	inductive sequence of *iblock*
	|	`Bcond`$(cond, \overrightarrow{reg_{\text{arg}}}, iblock_{\text{ifso}}, iblock_{\text{ifnot}}, iinfo)$	inductive conditional branch
iblock_info	=	$\{$`entry` : *iblock*; `binfo` : *binfo*$\}$	record encapsulating blocks
cfg	=	$(pc \mapsto iblock\_info)$	BTL graph as a map
f_{btl}	=	$\{$`fn_sig` : *sig*; `fn_params` : $\overrightarrow{reg_{\text{arg}}}$;	BTL function
		`fn_stacksize` : *ssize*; `fn_code` : *cfg*;	
		`fn_entrypoint` : pc_{entry};	
		`fn_gm` : *gm*; `fn_info` : *finfo*$\}$	

Figure 5.1: Syntax of the BTL IR (*gm*, *finfo*, *binfo* and *iinfo* will be defined later on).

To prove the preservation of the loop (Bbody) GI on register `i_a`, the SE uses a rewriting mechanism that normalizes affine expressions (see §7.6.2). Hence, "$(\mathtt{i} \times 7) + 21$" and "$(\mathtt{i} + 3) \times 7$" are normalized into "$21 + (7 \times \mathtt{i})$". The rewriting engine within SE is similar to the one presented in [135]†, with a more modular implementation allowing each transformation to define its own normalization rules, and a specific feature for symbolic affine forms. The complete syntax of both types of invariants is introduced along with the SE theory, in Chapter 6.

5.2 ABSTRACT SYNTAX [◇]

BTL instructions are defined inductively, so that a block *is a kind of compound instruction*. In practice, we only use BTL blocks such that each branch ends on a final instruction. A BTL function is a control-flow graph where each node is a block, represented by structured code.

5.2.1 *Syntactical Block Structure*

The abstract syntax is defined in Figure 5.1; instructions whose description is marked with a "♠" symbol are syntactically identical to their RTL equivalent (all *final* instructions except the `Bgoto`). In this syntax, we group final instructions in a specific subtype (with the `BF` constructor) of "classical" (i.e. non-final) instructions: *iblock* defines every classical instructions, along with the inductive sequences and conditionals; and *final* encodes instructions that terminate the block (or the function) execution[4]. To be compatible with the CompCert forward simulation proofs, calls are required to be final instructions.

This representation facilitates the traversal of blocks: one just have to run through execution paths starting from the (unique) entry point; the block separation is clear, and embedded syntactically

[4]If no final instruction is present, the block cannot step (see Definition 5.3.6).

```
if (x >= y) goto L;
x = z << 2;
return x;
```

```
Bseq(Bcond(_>=_, [x;y], BF(Bgoto(L)), Bnop),
     Bseq(Bop(_<<2, [z], x),
          BF(Breturn(x))))
```

Figure 5.2: A Superblock in C Syntax and its BTL Representation (without shadow fields).

```
if (i == 0) x = a;
else x = b;
return x;
```

```
Bseq(Bcond(_=0, [i], Bop(Omove, [a], x),
                     Bop(Omove, [b], x)),
     BF(Breturn(x)))
```

Figure 5.3: A Block With an Internal Join in C syntax and its BTL Representation (without shadow fields). The Omove operation simply copies its (unique) argument to its destination.

without the need for an additional structure (like the RTLpath "path maps"). Moreover, instructions do not need to encode their successor(s) directly in their constructors (as in RTL) anymore, since the successorship relation is structural (constructed using Bseq or Bcond, that compose two sub-blocks). Comparing to RTL (and RTLpath), the BTL syntax adds two new types of fields:

1. The *finfo*, *binfo*, and *iinfo* shadow fields: those are records implemented directly in OCaml, which have *no semantics* on the Coq side (cf. §2.2.2). They are used as a means of transmitting information about functions, blocks, and instructions (respectively) between oracles. For instance, *finfo* contains a Boolean to indicate if the BTL function is partitioned in basic blocks. Blocks' *binfo* fields record various data including the block number, a "visited" Boolean enabling to tag blocks, or even lists of successors and predecessors blocks. For instructions, the *iinfo* field's signification changes according to the type of instruction. For example, an "option bool" field is used to remember prediction information on branches, and this same field remembers the initial trapping mode on loads. Instructions' shadow fields also contain an ID, and a tag Boolean.

2. The *gm* field of BTL functions (stands for gluemap) maps program points *pc* to records containing the gluing and history invariants; these do not impact the semantics of BTL blocks, but only their symbolic execution to "synchronize" information (see §6.2.2.2). The gluemap is only filled by oracles: in other words, it is a *certificate* used to *drive* the formally verified defensive programming mechanism (recall §2.3).

A RTL "Inop(pc_{succ})" ending a block's branch is encoded by "Bseq(Bnop, BF(Bgoto(pc_{succ})))"; when it is in the middle of a branch, it is simply encoded by "Bnop"[5]. The same trick appears for all basic instructions, and for conditions. As visible in Figure 5.1, BTL *iblock* are not directly stored in the code tree, but rather encapsulated in a record (where we remember the first *iblock* instruction as the entry) containing the *binfo* shadow field. Figures 5.2 & 5.3 show examples of blocks of different sizes encoded in BTL.

5.2.2 *Detailed Breakdown of Instructions*

We saw in §3.4 that RTL is a parametrized representation; the same applies for BTL. Exhaustively, the architecture dependent constructors are:

- *op* (used in Bop instructions): the type of arithmetic operations;
- *cond* (in Bcond): the type of comparison instructions;
- *addr* (in Bload and Bstore): the type of addressing modes.

All the other instruction parameters are common to every target. Below is a brief description of their syntax as well as their role:

[5]The *iinfo* shadow field is hidden here to make the code easier to read.

- *sig* (in `Bcall`, `Btailcall`, and f_{btl}): the *signature* of functions, defined as a record containing the list of arguments' types, the return type, and a model of the calling convention;
- *id* (in `Bcall` and `Btailcall`): the function invoked by a call is either determined by a pointer found in a register, or by a *name* (i.e. an identifier *id*). This explains the $(reg|id)$ parameter of those instructions;
- *ef* (in `Bbuiltin`): built-ins call an *external function ef* having a predefined signature (e.g. for volatile loads and stores, mallocs, memcpys, annotations, etc.);
- $\overrightarrow{reg_{\text{bargs}}}$ and reg_{bdst}: arguments and destination registers for built-ins are encoded as specific inductive types, that can embed values or registers.

5.3 OPERATIONAL SEMANTICS

The dynamic semantics of `BTL` is very similar to `RTL` (recall §3.4.1), except that the step of one instruction is generalized into the run of one *iblock*. From here, we note v the type of CompCert values (defined in §3.3.1). The global environment associated with the program is still noted $\mathtt{G}$. `BTL` stack frames and states are defined analogously to the `RTL` ones[6]:

Definition 5.3.1 (`BTL` stack frames and states [⋄]).
A stack frame is defined as $\Sigma \triangleq (\mathtt{r}, \mathtt{f}, \mathtt{sp}, \mathtt{pc}_{\mathtt{succ}}, \mathtt{rs})$, where, from left to right, we have: the result register $\mathtt{r}$, the calling function $\mathtt{f}$ (the caller), its stack pointer $\mathtt{sp}$, the program counter of the calling function $\mathtt{pc}_{\mathtt{succ}}$ (i.e. the returning address after the call), and its regset $\mathtt{rs}$ (which saves the state of the caller).
From the above, we define the three types of states as follows:

$$\begin{array}{rcll} \mathtt{S} & ::= & \mathcal{S}(\vec{\Sigma}, \mathtt{f}, \mathtt{sp}, \mathtt{pc}, \mathtt{rs}, \mathtt{m}) & \text{State} \\ & | & \mathcal{C}(\vec{\Sigma}, \mathtt{f}_{\text{def}}, \overrightarrow{\mathtt{r}_{\text{arg}}}, \mathtt{m}) & \text{Callstate} \\ & | & \mathcal{R}(\vec{\Sigma}, v, \mathtt{m}) & \text{Returnstate} \end{array}$$

I give the below definitions directly in Coq to facilitate understanding, using the same notations for states and stack frames.

The internal step execution of an *iblock* is first defined as a relation transition between a pair $(\mathtt{rs}, \mathtt{m})$ of an initial register set and memory state to the resulting pair $(\mathtt{rs}', \mathtt{m}')$, under the global environment: $\mathtt{G} \vdash (\mathtt{rs}, \mathtt{m}) \to (\mathtt{rs}', \mathtt{m}')$. In coherence with the `rs#r` notation to access register `r` in regset `rs`, we note `rs##args` to access the *list* of registers `args` in `rs`.

Definition 5.3.2 (Relational semantics of internal `BTL` instructions)**.** In the Coq code below, the evaluation functions (whose name is starting with "`eval_`") and the `has_loaded` predicate (which defines the result of loads according to their trapping mode) are from the `RTL` module, and can be reused transparently. Those are hypotheses about the current instruction, and they must hold for the corresponding semantics rule to be defined.

```
Inductive iblock_istep G sp:
regset → mem → iblock → regset → mem → option final → Prop :=
  | exec_final rs m fin iinfo:
     iblock_istep G sp rs m (BF fin iinfo) rs m (Some fin)
  | exec_nop rs m oiinfo:
     iblock_istep G sp rs m (Bnop oiinfo) rs m None
  | exec_op rs m op args res v iinfo
     (EVAL: eval_operation G sp op rs##args m = Some v)
     : iblock_istep G sp rs m (Bop op args res iinfo) (rs#res ←v) m None
  | exec_load rs m trap chunk addr args dst v iinfo
      (LOAD: has_loaded G sp rs m chunk addr args v trap)
     : iblock_istep G sp rs m (Bload trap chunk addr args dst iinfo) (rs#dst ←v) m None
  | exec_store rs m chunk addr args src a m' iinfo
     (EVAL: eval_addressing G sp addr rs##args = Some a)
     (STORE: Mem.storev chunk m a rs#src = Some m')
     : iblock_istep G sp rs m (Bstore chunk addr args src iinfo) rs m' None
```

[6] In particular, `BTL` states also obey to the diagram of Figure 3.4.

```
| exec_seq_stop rs m b1 b2 rs' m' fin
   (EXEC: iblock_istep G sp rs m b1 rs' m' (Some fin))
   : iblock_istep G sp rs m (Bseq b1 b2) rs' m' (Some fin)
| exec_seq_continue rs m b1 b2 rs1 m1 rs' m' ofin
   (EXEC1: iblock_istep G sp rs m b1 rs1 m1 None)
   (EXEC2: iblock_istep G sp rs1 m1 b2 rs' m' ofin)
   : iblock_istep G sp rs m (Bseq b1 b2) rs' m' ofin
| exec_cond rs m cond args ifso ifnot b rs' m' ofin iinfo
   (EVAL: eval_condition cond rs##args m = Some b)
   (EXEC: iblock_istep G sp rs m (if b then ifso else ifnot) rs' m' ofin)
   : iblock_istep G sp rs m (Bcond cond args ifso ifnot iinfo) rs' m' ofin
```

The transition relation takes a stack pointer `sp` and a `BTL` instruction; its last parameter, which is of the "*final* option" type, is `None` when the instruction does not directly end the execution. Otherwise, for the `exec_final` and `exec_stop` rules, the option type embeds the terminating instruction. Observe that `exec_stop` applies when the first block (i.e. `b1`) of a `Bseq` is final: in such a situation, the second block `b2` is thus dead-code (i.e. it will never been reached). Oppositely, the `exec_seq_continue` rule imposes that the first block does not finish the execution. When the instruction is a conditional branch, the presence of a final instruction does not matter (i.e. rule `exec_cond`).

The relation of Definition 5.3.2 is a partial function; when writing proofs, having a pure functional variant is sometimes very useful to facilitate the reasoning. To implement such a variant, we start by defining the *outcome* of a block execution:

```
Record outcome := out { _rs: regset; _m: mem; _fin: option final }
```

In the following, we use a Coq notation to simplify the binding with the option monad:

Definition 5.3.3 (Option monad binder)**.**

```
Notation "'SOME' X ← A 'IN' B" :=
  (match A with Some X ⇒ B | None ⇒ None end)
  (at level 200, X name, A at level 100, B at level 200) : option_monad_scope
```

We then redefine the relation as a Coq function using this notation:

Definition 5.3.4 (Functional semantics of internal `BTL` instructions)**.** The functional variant is implemented as a fixpoint (having the same signature as in Definition 5.3.2):

```
Fixpoint iblock_istep_run G sp ib rs m: option outcome :=
  match ib with
  | BF fin _ ⇒
      Some {| _rs := rs; _m := m; _fin := Some fin |}
  (* basic instructions *)
  | Bnop _ ⇒
      Some {| _rs := rs; _m := m; _fin := None |}
  | Bop op args res _ ⇒
      SOME v ← eval_operation G sp op rs##args m IN
      Some {| _rs := rs#res ← v; _m := m; _fin := None |}
  | Bload TRAP chunk addr args dst _ ⇒
      SOME a ← eval_addressing G sp addr rs##args IN
      SOME v ← Mem.loadv chunk m a IN
      Some {| _rs := rs#dst ← v; _m := m; _fin := None |}
  | Bload NOTRAP chunk addr args dst _ ⇒
      match eval_addressing G sp addr rs##args with
      | Some a ⇒
          match Mem.loadv chunk m a with
          | Some v ⇒ Some {| _rs := rs#dst ← v; _m := m; _fin := None |}
          | None ⇒
              Some {| _rs := rs#dst ← Vundef; _m := m; _fin := None |}
          end
      | None ⇒
          Some {| _rs := rs#dst ← Vundef; _m := m; _fin := None |}
```

```
    end
  | Bstore chunk addr args src _ ⇒
    SOME a ← eval_addressing G sp addr rs##args IN
    SOME m' ← Mem.storev chunk m a rs#src IN
    Some {| _rs := rs; _m := m'; _fin := None |}
 (* composed instructions *)
  | Bseq b1 b2 ⇒
    SOME out1 ← iblock_istep_run G sp b1 rs m IN
    match out1.(_fin) with
    | None ⇒ iblock_istep_run G sp b2 out1.(_rs) out1.(_m)
    (* stop execution on the 1st final instruction *)
    | _ ⇒ Some out1
    end
  | Bcond cond args ifso ifnot _ ⇒
    SOME b ← eval_condition cond rs##args m IN
    iblock_istep_run G sp (if b then ifso else ifnot) rs m
  end
```

The functional Definition 5.3.4 returns `None` to represent every case where Relation 5.3.2 does not hold. This is proved by Lemma 5.3.1 (`iblock_istep_run_equiv`).

Lemma 5.3.1 (Proof that Relation 5.3.2 is a partial function [◇])**.**

```
Lemma iblock_istep_run_equiv G sp rs m ib rs' m' ofin:
  iblock_istep G sp rs m ib rs' m' ofin ↔
  iblock_istep_run G sp ib rs m = Some {| _rs :=rs'; _m :=m'; _fin :=ofin |}
```

Proof. Trivial induction on `ib`*, in both directions.* □

The semantics for final instructions is also defined as a transition relation, but in a slightly more complex way since this time, we go from an initial $(\mathrm{rs}, \mathrm{m})$ pair to a final *state*: $\mathrm{G} \vdash (\mathrm{rs}, \mathrm{m}) \xrightarrow{e} \mathrm{S}$. Letter e over the transition relation denotes the emitted *trace* (i.e. observable event) from some final instructions (e.g. built-ins). When an instruction does not produce any observable event, we note it with an ϵ.

Definition 5.3.5 (Relational semantics of final `BTL` instructions)**.** Since final instructions include calls to other functions, the relation must take into account the call stack (i.e. the list of stack frames $\vec{\Sigma}$), and the current function `f`.

```
Inductive final_step G Σ⃗ f sp rs m: final → trace → state → Prop :=
  | exec_Bgoto pc:
      final_step G Σ⃗ f sp rs m (Bgoto pc) ϵ (𝒮 Σ⃗ f sp pc rs m)
  | exec_Breturn or stk m':
      sp = (Vptr stk Ptrofs.zero) →
      Mem.free m stk 0 f.(fn_stacksize) = Some m' →
      final_step G Σ⃗ f sp rs m (Breturn or) ϵ (ℛ Σ⃗ (regmap_optget or Vundef rs) m')
  | exec_Bcall sig ros args res pc' fd:
      find_function ros rs = Some fd →
      funsig fd = sig →
      final_step G Σ⃗ f sp rs m (Bcall sig ros args res pc')
        ϵ (𝒞 (Σ(res, f, sp, pc', rs) :: Σ⃗) fd rs##args m)
  | exec_Btailcall sig ros args stk m' fd:
      find_function ros rs = Some fd →
      funsig fd = sig →
      sp = (Vptr stk Ptrofs.zero) →
      Mem.free m stk 0 f.(fn_stacksize) = Some m' →
      final_step G Σ⃗ f sp rs m (Btailcall sig ros args) ϵ (𝒞 Σ⃗ fd rs##args m')
  | exec_Bbuiltin ef args res pc' vargs e vres m':
      eval_builtin_args G (λ r ⇒rs#r) sp m args vargs →
      external_call ef G vargs m e vres m' →
      final_step G Σ⃗ f sp rs m (Bbuiltin ef args res pc')
```

```
      e (S Σ⃗ f sp pc' (regmap_setres res vres rs) m')
  | exec_Bjumptable arg tbl n pc':
      rs#arg = Vint n →
      list_nth_z tbl (Int.unsigned n) = Some pc' →
      final_step G Σ⃗ f sp rs m (Bjumptable arg tbl) e (S Σ⃗ f sp pc' rs m)
```

Here also, the relation is only partial, and some final instructions have a defined semantics only when their hypotheses are satisfied. In the case of a return instruction (leading to a return state), the stack pointer must be a pointer with offset zero, and the callee's stack freed. Calls and tail calls are defined only if their target function exists, and lead to a call state. For calls, a new stack frame is allocated in the call stack; and for tail calls, the hypotheses are similar as those of the return case. The semantics for built-ins is the only one emitting an observable trace. There are also two hypotheses about the arguments' evaluation (they must evaluate correctly to values), and the external call itself (which instantiates the relation between the input and output memories). The execution of built-ins leads to a normal state. Finally, both gotos and jump tables end in a normal state, but while the former's rule makes no assumptions, the latter's rule requires a valid argument and an existing destination address in the table.

The whole blockstep execution is then defined as a conjunction between Relations 5.3.2 and 5.3.5, where the former must end with a final instruction (handled by the latter):

Definition 5.3.6 (Blockstep execution of one *iblock*)**.** Given an initial global environment G, call stack $\vec{\Sigma}$, BTL function f, stack pointer sp, register set and memory state rs and m (respectively), an *iblock* ib, a trace e, and a final (output) state s, we have:

```
Definition iblock_step G Σ⃗ f sp rs m ib e s: Prop :=
  ∃ rs' m' (fin: final), iblock_istep G Σ⃗ rs m ib rs' m' (Some fin) ∧
                         final_step G Σ⃗ f sp rs' m' fin e s
```

By construction, such a blockstep ends just after the RTL style small-step of a final instruction, due to the `final_step` requirement.

The execution property of Definition 5.3.6 is parametrized by the output state, because we need this information to declare the transitions between BTL states. Those are presented as an inductive predicate split in four cases:

Definition 5.3.7 (Step transition predicate to integrate the BTL semantics in CHAMOIS-COMPCERT)**.** Lastly, we obtain the complete state relation $G \vdash S \xrightarrow{e} S'$ between an initial state S to a final one S' with a possible trace e.

```
Inductive step G: state → trace → state → Prop :=
  | exec_iblock Σ⃗ ib f sp pc rs m e s
      (PC: (fn_code f)!pc = Some ib)
      (STEP: iblock_step G Σ⃗ f sp rs m ib.(entry) e s)
      :step G (S Σ⃗ f sp pc rs m) e s
  | exec_function_internal Σ⃗ f args m m' stk
      (ALLOC: Mem.alloc m 0 f.(fn_stacksize) = (m', stk))
      :step G (C Σ⃗ (Internal f) args m)
         e (S Σ⃗ f (Vptr stk Ptrofs.zero) f.(fn_entrypoint)
                  (init_regs args f.(fn_params)) m')
  | exec_function_external Σ⃗ ef args res e m m'
      (EXTCALL: external_call ef G args m e res m')
      :step G (C Σ⃗ (External ef) args m) e (R Σ⃗ res m')
  | exec_return Σ⃗ res f sp pc rs vres m
      :step G (R (Σ(res, f, sp, pc, rs) :: Σ⃗) vres m)
         e (S Σ⃗ f sp pc (rs#res ←vres) m)
```

The above code still uses the "exclamation point" notation for map accesses.

The execution of a BTL *iblock* requires a valid location in the code through the "PC" hypothesis, and starts from a normal state. Depending on the final instruction, the transition will end up in an output

state given by Property 5.3.6. The `exec_iblock` rule being only applicable from normal states, the step predicate of Definition 5.3.7 also defines three more transitions: (i) `exec_function_internal` goes from a call state to a normal state without emitting a trace, but assuming a correct stack allocation; (ii) `exec_function_external` handles terminating calls, that go from a call state to a return state while emitting an observable trace; and (iii) `exec_return` simply comes back to a normal state from a return state, by popping off the callee and setting the result.

The execution of a whole program is described as *sequences of transitions* from an initial state $(\mathtt{P}, \mathtt{C})$ to a final state $(\mathtt{R}, \mathtt{r})$. An initial state is a call state `C` corresponding to the invocation of the "main" function of the program `P` without arguments and with an empty call stack. A final state is a return state `R` with an empty call stack and a return value `r` of type `int`. Each transition is thus a blockstep (i.e. big-step) following Definition 5.3.7.

6

SYMBOLIC SIMULATION THEORY†

I formalize in this chapter the theory of the symbolic simulation framework modulo invariants. In order to facilitate the proof decomposition, the theory reasons on a high-level representation of symbolic states, and the SE is defined without hash-consing. I explain in Chapter 7 how we refine this theory in a concrete, efficient implementation.

Our main objective here is to construct a simulation predicate between (the source and target) BTL blocks implied by the implementation, and implying the semantic simulation between BTL programs (actually, the forward simulation of the source program by the transformed one). I start by presenting this goal in Section 6.1 as a blockstep simulation diagram *encompassing* the SE predicate of simulation; it gives an overview of the proof scheme, and sketches a specification of that simulation predicate. The remaining parts of the chapter gradually introduce the necessary notions to build the latter predicate: Section 6.2 defines the syntax for *symbolic values* and *invariants*. Subsequently, their semantics in terms of *concrete* values is formalized in Section 6.3. The symbolic *semantics of execution* for blocks over symbolic *states* is presented in Section 6.4; and we show how to apply invariants to establish the final *simulation predicate* in Section 6.5. Lastly, Section 6.6 gives additional theorems that complete the initial proof diagram of Section 6.1.

6.1 A BLOCKSTEP FORWARD SIMULATION PASS

Our symbolic simulation test over BTL programs enables formally proving a generic pass parametrized by an oracle. The oracle is declared as an OCaml function expecting as argument a source BTL function f_{btl} (as defined in Figure 5.1), and returning a triplet $(cfg, finfo, gm)$ where cfg is the target CFG, $finfo$ is the shadow field for functions, and gm is the gluemap—see Definition 6.2.5—filled with invariants. When the checker validates the oracle's results on *all the functions* of a source BTL program, the pass returns the target BTL program. Otherwise, the pass fails. Moreover, our generic pass is also configured with a type indicating the set of normalization rules—type R below—that will be available during SE (see §7.6). In practice, the oracle and the set of rules are encapsulated in a Coq module type:

```
Module Type BTL_BlockSimulationConfig
  (** External oracle *)
  Parameter btl_optim_oracle: f_btl → cfg * finfo * gm
  (** Set of normalization rules for SE *)
  Parameter btl_rrules: unit → R
End BTL_BlockSimulationConfig
```

The blockstep simulation pass is thus a functor that we prove correct for any instance of the above type:
`Module BTL_BlockSimulation (B: BTL_BlockSimulationConfig).` It is formally proven to perform a lock-step forward simulation, as pictured in Figure 6.1[1]: for any blockstep on source concrete states $S_1 \to^e S_2$ (emitting a possible observational event e), for any target state S'_1 related to S_1 by the gluemap, relation written $S_1 \sim_{gm} S'_1$, there exists a blockstep on target concrete states $S'_1 \to^e S'_2$ such that $S_2 \sim_{gm} S'_2$.

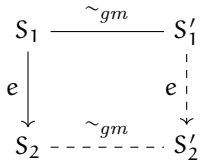


Figure 6.1: Lock-Step Simu.

Defining the "$\sim_{gm}$" relation is not completely straightforward because we need to express that the source call stack is simulated by the target call stack through the gluemap of each caller function. This is necessary even if the analysis and the transformation are performed separately for each function: a simulation invariant on stack frames is needed to establish that the

[1]This diagram follows the usual convention: solid lines are hypotheses of the theorem, dashed lines are conclusions. It resembles to the lock-step diagram on the top left of Figure 3.2, except that horizontal matching lines are specified to be "modulo gluemap".

invariant is still true when returning from a function call. The complete relation is formalized in Definition 6.1.3 below.

6.1.1 *Simulation of Concrete BTL States Induced by Symbolic Simulation*

First, we define a `match_function` relation between a source `BTL` function `f` and a target one `f'` expressing that `f'` is a *symbolic simulation* of `f`.

Definition 6.1.1 (Specification to verify between the source and target `BTL` functions)**.**

```
Record match_function (f f': f_btl) : Prop := {
  preserv_fnsig: fn_sig f = fn_sig f';
  preserv_fnparams: fn_params f = fn_params f';
  preserv_fnstacksize: fn_stacksize f = fn_stacksize f';
  preserv_entrypoint: fn_entrypoint f = fn_entrypoint f';
  trivial_histinv_entrypoint:
    only_liveness (history (f'.(fn_gm) (fn_entrypoint f)));
  trivial_glueinv_entrypoint:
    only_liveness (glue (f'.(fn_gm) (fn_entrypoint f)));
  match_sexec_ok: ∀ pc ib, (fn_code f)!pc = Some ib →
                   ∃ ib', (fn_code f')!pc = Some ib' ∧
                   instantiate_context match_sexec_si
                     f'.(fn_gm) (entry ib) (entry ib') pc;
}
```

In other words, the signatures, the functions' parameters, the stack sizes, and the CFG entry points are identical (from the four first conditions). The gluing and history invariants at the entry point only contain liveness assignments: I explain why, and formalize the corresponding `only_liveness` property in §6.3.3.4. And, as expressed by condition `match_sexec_ok`, for any source block `ib` at label `pc`, there is a target block `ib'` at label `pc` such that the simulation predicate—introduced in §6.5.3—is satisfied. Notice that the latter is itself given to an `instantiate_context` function (Definition 6.4.4) that universally quantifies the simulation context.

Second, the `match_stackframes` predicate relates a source and a target stack frame under a global environment `G` (recall Definition 5.3.1 of `BTL` stack frames).

Definition 6.1.2 (Specification of the matching relation between stack frames)**.**

```
Inductive match_stackframes G: Σ → Σ → Prop :=
  | match_stackframe_intro sp res f pc rs rs' f'
      (TRANSF: match_function f f')
      (MATCHI: ∀ v m, match_invs (mk_iblock_ctx(G, sp, (rs#res ←v), m))
                                 (f'.(fn_gm) pc) (rs'#res ← v))
      : match_stackframes G (Σ res f sp pc rs) (Σ res f' sp pc rs')
```

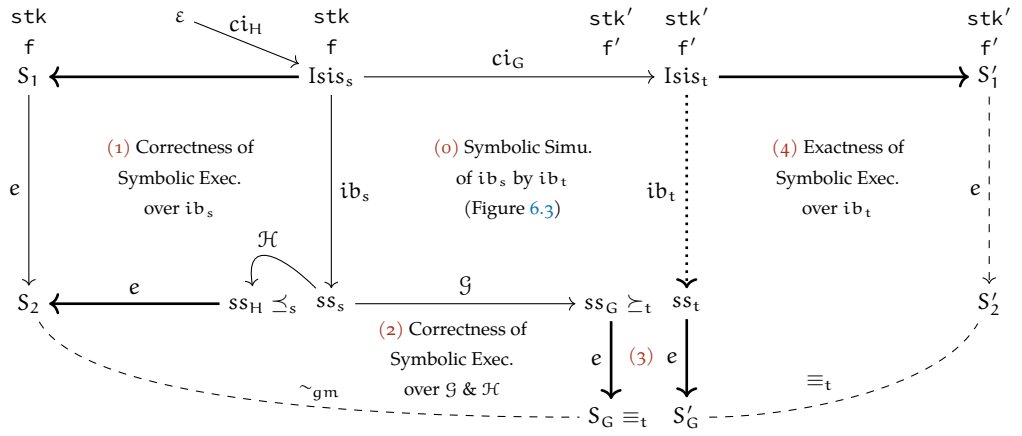
Hence, `match_stackframes` describes how the source stack frame is simulated by a target stack frame: the target caller is *symbolic* simulation of the source one (condition `TRANSF`); and, condition `MATCHI`, for any returned value `v` assigned to register `res`, then the source register state "`rs#res ← v`" is mapped to the target register state "`rs'#res ← v`" through the invariants at `pc`.

Above, the `match_invs` predicate characterizes—under the initial execution context—the semantics of our invariants (see Definition 6.3.11). The different notions of context, and in particular the *block level context* built by function $\mathrm{mk\_}iblock_{ctx}$ are explained in §6.3.1; the semantics of invariants under such a context is specified in §6.3.3.3.

Third, the relation between a source and a target concrete state is given by the `match_states` predicate below, for each type of state of Definition 5.3.1.

Definition 6.1.3 (Specification of the matching relation "$\sim_{gm}$" between concrete states)**.**

```
Inductive match_states G: S → S → Prop :=
  | match_states_intro Σ⃗ f pc sp rs rs' m Σ⃗' f'
      (TRANSF: match_function f f')
```



(3): Correctness of the modulo liveness relation

Figure 6.2: Diagrammatic Proof of Blockstep Simulation.

```
    (MATCHI: match_invs (mk_iblock_ctx(G, sp, rs, m)) (f'.(fn_gm) pc) rs')
    (STACKS: list_forall2 (match_stackframes G) Σ Σ')
    : match_states G (S Σ f sp pc rs m) (S Σ' f' sp pc rs' m)
| match_states_call Σ Σ' f f' args m
    (STACKS: list_forall2 (match_stackframes G) Σ Σ')
    (TRANSF: match_fundef f f')
    : match_states G (C Σ f args m) (C Σ' f' args m)
| match_states_return Σ Σ' v m
    (STACKS: list_forall2 (match_stackframes G) Σ Σ')
    : match_states G (R Σ v m) (R Σ' v m)
```

Condition STACKS expresses that each stackframe of $\vec{\Sigma}$ is simulated by the corresponding one in $\vec{\Sigma}'$ (thanks to the `list_forall2` predicate from CoqLib). The MATCHI condition of the `match_states_intro` case expresses that when the execution has reached label `pc`, source register state `rs` is mapped to target register state `rs'` through the invariants at `pc`. Note that because our gluing invariants cannot transform the memory but only the registers, then the target and the source memories are identical (both written `m`). For calls, the `match_fundef` hypothesis states that only internal functions are subject to change under the `match_function` relation; external calls must remain identical.

6.1.2 *Sketch of the Blockstep Simulation Proof*

In Figure 6.1, the "$\sim_{gm}$" relation applies invariants on concrete states. The symbolic simulation test applies them to symbolic states instead. In order to prove the diagram of Figure 6.1 for normal blocksteps, we thus need to transform the relations over symbolic states resulting from the SE into relations over concrete states. Figure 6.2 sketches this proof: the expected diagram is built by composing the diagram of the simulation predicate (that I detail in §6.5.1, Figure 6.3) with 4 other subdiagrams, each of them being an intermediate lemma. In the figure, a thick arrow $ss \rightarrow^{e} S$ means that—in the current block execution context—the symbolic state ss evaluates into a concrete state matching S while emitting the observable event e. Similarly, a thick arrow $sis \rightarrow S$ means that (still under the block execution context) symbolic internal state sis evaluates into a concrete state matching S. In order to evaluate any symbolic state, we need to consider a well-chosen block execution context, and in particular a well-chosen call stack. This information is given at the top in Figure 6.2: the *context switch* from the source context to the target context is performed while applying $\mathcal{G}$ to (symbolic or concrete) states[2].

Here is the sketch of the proof. Under the assumptions that $S_1 \rightarrow^{e} S_2$ is a normal blockstep and that proposition $S_1 \sim_{gm} S_1'$ holds, we first deduce (from the TRANSF condition of Definitions 6.1.2 & 6.1.3) that the symbolic simulation test represented at diagram (0) succeeds. The first lemma

[2]Below, "source" and "target" modes refer to the variable domain used in the matching relation. See §6.4.

(6.4.11)—the correctness of symbolic execution in source mode, at diagram (1)—states that given a valid blockstep $S_1 \rightarrow^e S_2$, there exists an *initial* source symbolic internal state $\mathtt{Isis}_s$ and a final symbolic state ss_s, both evaluating to concrete states S_1 and S_2, respectively. Then, a second lemma (6.6.1)—the correctness of invariants application on the source, at diagram (2)—establishes that symbolic state ss_G evaluates (while emitting observational event e) to a concrete state S_G such that $S_2 \sim_{gm} S_G$. We then mimic the comparison $ss_G \succeq_t ss_t$ on concrete states with a third lemma (6.6.2)—at diagram (3)—which proves the correctness of the modulo liveness relation. This gives a concrete state S'_G such that symbolic state $ss_t \rightarrow^e S'_G$ and $S_G \equiv_t S'_G$: both concrete states coincide on live registers of the target program (see the variant of "$\equiv_E$" for states [⋄]). A last (and fourth) lemma (6.4.12)—the exactness of symbolic execution in target mode, at diagram (4)—establishes that there exists a blockstep $S'_1 \rightarrow^e S'_2$ such that $S'_2 \equiv_t S'_G$. Finally, we prove $S_2 \sim_{gm} S'_2$ from $S_2 \sim_{gm} S_G \equiv_t S'_2$.

I describe the proof in more details in §6.6, after the introduction of the simulation predicate corresponding to diagram (o) in Figure 6.2.

6.2 SYNTAX OF SYMBOLIC VALUES AND INVARIANTS

The preliminary step to formalize invariants is to define symbolic values analogously[3] as RTLpath, with a few improvements (most of the RTLpath limitations we overcome are not due to changes in the type of symbolic values but rather to the invariants and the—more general—representation of symbolic states). We then propose two possible representations for invariants: an *abstract* one for simplifying Coq proofs, and another one more adapted for *concrete* computations (both in the oracles and in the checker).

It is common to separate types for theory and implementation: we often want "naive" (but intuitive) theory, and "clever", efficient implementation.

6.2.1 *BTL Symbolic Values*

The aim is to represent the value of a register after a sequence of *concrete* BTL assignments. In RTLpath, there were two representations of symbolic values. An abstract representation for simulation theory; and a refined representation for the implementation. In use, this duplication of representations proved to be cumbersome and not very useful.

We have chosen here to simplify the approach by using directly in the simulation theory, the representation of the symbolic values used by the implementation. This choice has of course a defect: it exposes constructions which are only useful in the implementation and not in the theory. But this defect seems less annoying in use than the duplication of representation à la RTLpath.

These useless constructions in the theory are: the hash identifiers used for hash-consing and the `Sfoldr` constructor for rewriting affine forms (introduced in §7.6.2).

Definition 6.2.1 (BTL mutually inductive symbolic values [⋄])**.** Our theory will always instantiate the hash-consing identifiers ("*hid*" below) with a special value `unknown_hid` (see §7.2.4 for an extended explanation about the hash-consing mechanism). We will not use `Sfoldr`. The `Sop` constructor does not include the memory (operations do not modify it, and it is kept in a global context, see §6.3.1)[4].

$$
\begin{array}{rcl}
\mathtt{sval} & ::= & \mathtt{Sinput}(\mathit{reg}, \mathit{hid}) \mid \mathtt{Sop}(\mathit{op}, \mathtt{list\_sval}, \mathit{hid}) \\
& \mid & \mathtt{Sfoldr}(\mathit{op}, \mathtt{list\_sval}, \mathtt{sval}, \mathit{hid}) \mid \mathtt{Sload}(\mathtt{smem}, \mathit{trap}, \mathit{chk}, \mathit{addr}, \mathtt{list\_sval}, \mathit{hid}) \\
\mathit{with}\ \mathtt{list\_sval} & ::= & \mathtt{Snil}(\mathit{hid}) \mid \mathtt{Scons}(\mathtt{sval}, \mathtt{list\_sval}, \mathit{hid}) \\
\mathit{with}\ \mathtt{smem} & ::= & \mathtt{Sinit}(\mathit{hid}) \mid \mathtt{Sstore}(\mathtt{smem}, \mathit{chk}, \mathit{addr}, \mathtt{list\_sval}, \mathtt{sval}, \mathit{hid})
\end{array}
$$

Section 7.2.4.3 describes how we project fake symbolic values to normal ones with hash-consing.

To build the symbolic values of the theory, I use "fake constructors" that set hid fields to `unknown_hid`. In fact, it is also interesting in the implementation, to simplify the proofs on the rewrites of hash-consed terms. From here, I note those fake constructors using the same names as for symbolic values with an "`f`" prefix (e.g. `fSinput`, `fSop`, ...).

In our Coq code, we also define a mutually inductive *scheme* that is useful for proofs, and a *decidable equality* over symbolic values.

[3] Symbolic instructions are mostly close to concrete ones (they mimic BTL instructions, which are themselves semantically close to RTL ones).

[4] In RTLpath, the memory was removed from the Sop only in the refined representation—it simplifies in fact to remove it from the theory.

6.2.2 *Representations of Invariants*

See Example 6.2.1 for a comparison of both representations.

As introduced in §5.1, our invariants themselves represent *preconditioned parallel assignments of symbolic values*. However, Example 2.2.1 illustrates on block (B_2) that such parallel assignments may contain (exponential) term duplications w.r.t. a sequential representation. Because invariants are syntactically provided by the oracle under validation, we design a compact, simpler syntax, which represents them as sequences of assignments. The parallel assignments are never really built by the simulation test; it performs instead a hash-consed SE of the compact invariants. Nevertheless, having a parallel representation facilitates reasoning in the formal proof.

6.2.2.1 *Parallel Form*

We name this abstract representation "finite parallel assignment of symbolic values (FPASV)":

Definition 6.2.2 (Syntax of abstract invariants [◇])**.** Such invariants relate a finite set of "output" variables in function of "input" variables and an "input" memory (here, "input" means "at the block entry") through `fpa_reg`; they also express that symbolic values in `fpa_ok` do not trap (e.g. as with blocks' preconditions of Example 2.2.1).

$$\begin{aligned}\texttt{fpasv} \triangleq \{\, &\texttt{fpa\_ok} : \overrightarrow{sval};\ \texttt{fpa\_reg} :> reg \mapsto sval\ \texttt{option};\\ &\texttt{fpa\_wf} : \forall \texttt{r}\ sv, \texttt{fpa\_reg!r} = \texttt{Some}\ sv \implies sv \neq \texttt{Sinput} \implies sv \in \texttt{fpa\_ok}\,\}\end{aligned}$$

Where the ":>" notation indicates the default field to coerce with the record's type.

The wellformedness condition `fpa_wf` expresses that if a value is defined in the invariant, and if it is not a trivial `Sinput`, then it is in `fpa_ok`, and thus does not trap.
To compare FPASVs, we define the equality relation:

$$\begin{aligned}\texttt{fpa\_eq}\ si_1\ si_2 \triangleq \forall \texttt{r}, \texttt{r} \in si_1.(\texttt{fpa\_ok}) &\iff \texttt{r} \in si_2.(\texttt{fpa\_ok}) \wedge\\ si_1.(\texttt{fpa\_reg}) &= si_2.(\texttt{fpa\_reg})\end{aligned}$$

6.2.2.2 *Compact Form*

The sequential, concrete representation is slightly more complex, since we want to distinguish registers with an "input" prefix, indicating that the *input* (at the block entry) value is considered instead of the *current* (i.e. *last*) one within the assignment sequence. A sequential invariant is thus either a register, prefixed with input or last, or an operation involving current or input registers, or a load from the *current* memory.

Definition 6.2.3 (Invariant registers, operations and values [◇])**.** The type for registers represents this prefix with a Boolean:

```
Record ireg := { force_input: bool; regof:> reg }

(** Constructors of both register types *)
Definition input (r:reg) := {| force_input := true; regof := r |}
Definition last (r:reg) := {| force_input := false; regof := r |}
```

Operations supported by invariants are first defined without arguments:

$$\texttt{root\_op} ::= \texttt{Rop}(op) \mid \texttt{Rload}(trap, chk, addr)$$

The notion of "root" values is inspired by the thesis of Six [133, §7.3.3]; here, we extend its usage to invariant's values.

We call this abstraction "root" symbolic values, as it encodes the essential, generic information about basic (i.e. non-branching) instructions on registers. A root value is either an operation or a load (whether concrete or symbolic). Root values are intended to be completed with symbolic arguments and memory, allowing to directly convert them to ("fake") symbolic values by using smart constructors:

```
Definition root_apply (o : root_op) (args : list_sval) (sm : smem) : sval :=
  match o with
  | Rop op ⇒ fSop op args
  | Rload trap chunk addr ⇒ fSload sm trap chunk addr args
  end
```

And to specify a type of invariant values relying on `ireg` (*instead of* symbolic arguments), able to encode the distinction between input and current values:

$$\texttt{ival} ::= \texttt{Ireg}(\mathit{ireg}) \mid \texttt{Iop}(\mathit{root\_op}, \overrightarrow{\mathit{ireg}})$$

The compact representation is named "compact sequence of assignments of symbolic values (CSASV)":

Definition 6.2.4 (Syntax of sequential invariants [◇])**.** First, `aseq` is the sequence itself, and second, we export a finite set `output` that distinguishes live registers. Lazily, the simulation test considers that the set of registers `r` such that "`r` = `fSinput(r)`" is those not defined in `aseq` (i.e. it suffices to include them in `output`). In §6.3.12, we characterize CSASVs which only constrain liveness with a property being true when the `aseq` list is empty; hence, oracles must not set trivial "input" assignments (e.g. like "`r` = `fSinput(r)`") in `aseq`, because it would lead the verifier to reject the invariant while checking this "only liveness" constraint.

$$\texttt{csasv} \triangleq \{\, \texttt{aseq} : \overrightarrow{(\mathit{reg} * \mathit{ival})};\ \texttt{output} : \mathit{regset} \,\}$$

Example 6.2.1 (Link between both invariants' representations)**.** The two following compact invariants (CSASVs) "`([a ≔ y[5]; z ≔ a + input x; x ≔ a + input z], {x, z, t})`" and "`([z ≔ y[5]; x ≔ z + input z; z ≔ z + input x], {x, z, t})`" both represent parallel assignment (FPASV) "`x ≔ y[5] + z ‖ z ≔ y[5] + x ‖ t ≔ t`" with precondition "OK(`y[5]`)".

The *gm* field of `BTL` functions (Figure 5.1) is then a map from program points *pc* to invariant records with a CSASV for gluing invariants (GIs) and another one for history invariants (HIs).

Definition 6.2.5 (Symbolic invariant mapping)**.**

```
Record invariants ≔ {
  history: csasv; (* history (on the source) *)
  glue:> csasv    (* gluing (relating the source and the target) *)
}

Definition gluemap ≔ pc ↦ invariants. (* symbolic invariant map *)
```

6.3 CONCRETE SEMANTICS OF SYMBOLIC VALUES AND INVARIANTS

Symbolic values have an evaluation semantics that corresponds to concrete CompCert values, and which is dependent on the (symbolic) execution context. Abstract invariants (i.e. FPASVs) are *dictionaries* from registers to symbolic values, and are contextually evaluated using the *same mechanism*. On the other hand, compact invariants (i.e. CSASVs) restrict the set of possible operations to encode (with the *ival* type). Defining their contextual evaluation semantics thus implies a prior conversion into their abstract representation.

I present the contextual evaluation of symbolic values in §6.3.1, and the semantic relation for abstract invariants w.r.t. concrete register sets in §6.3.2. The translation of concrete to abstract invariants (i.e. to trees of symbolic values), and especially their *concrete semantics* is detailed in §6.3.3.

6.3.1 *Execution Context & Evaluation*

To alleviate the definitions of our framework, we parametrize the semantics of our SE by the *context of execution* of an *iblock*, encapsulating the common parameters:

Definition 6.3.1 (Execution context at block level [⋄])**.**

$$iblock_{ctx} \triangleq \{\, \mathtt{cG} : BTL.genv;\ \mathtt{csp} : val;\ \mathtt{crs}_0 : regset;\ \mathtt{cm}_0 : mem \,\}$$

It contains the global environment cG, the stack pointer of the current function csp (of type val for CompCert values, defined in §3.3.1), and the initial states for registers and memory $\mathtt{crs}_0$ and $\mathtt{cm}_0$, respectively. We note "$\mathtt{mk}\_iblock_{ctx}(\mathtt{G}, \mathtt{sp}, \mathtt{rs}, \mathtt{m})$" the function that builds a record of this type.

This context helps us to define symbolic states (see §6.4.3), and to instantiate the semantics of symbolic values (below); for the semantics of final symbolic values (see §6.4.1.1), we will also need the current BTL function (e.g. to allocate/deallocate on the stack). This led us to encapsulate the block context in order to specify it with the associated function:

Definition 6.3.2 (Encapsulated execution context with the BTL function [⋄])**.**

$$fct\_iblock_{ctx} \triangleq \{\, \mathtt{cf} : f_{\mathrm{btl}};\ \mathtt{cc} :> iblock_{ctx} \,\}$$

Non final symbolic values are contextually evaluated as follows:

Definition 6.3.3 (Evaluation of symbolic values [⋄])**.** Our symbolic values are evaluated with a partial function "$\sigma_{sv} : iblock_{ctx} \to sval \to \mathtt{val\ option}$". It is also defined for *list_sval* and *smem* types, again in a mutually inductive way: the list signature σ_{lsv} returns a "$\overrightarrow{\mathtt{val}}$ option", and the memory signature σ_{sm} a "mem option", with mem being the internal CompCert type used to represent the memory (cf. §3.3.3).

Here, I use the "_" wildcard to avoid noting the hid, since the definition ignores hash codes.

For instance, the definition of σ_{sv} to evaluate input symbolic values looks like:

$$\sigma_{sv}(\mathtt{ctx}, \mathtt{Sinput}(\mathtt{r}, \_)) \triangleq \mathtt{Some}(\mathtt{ctx}.(\mathtt{crs}_0)\#\mathtt{r})$$

Of course, the evaluation of symbolic values relying on an existing type—such as operations or loads—is close to their corresponding BTL semantics, and reuses the existing CompCert functions. As another example, with $\lfloor . \rfloor_C$ denoting the concrete BTL evaluation semantics (e.g. for operations, it would be the "`eval_operation`" function visible in Definition 5.3.2) under a context C, operations evaluate in the following way[5]:

$$\sigma_{sv}(\mathtt{ctx}, \mathtt{Sop}(\mathtt{op}, \mathtt{lsv}, \_)) \triangleq \mathtt{SOME\ args} \leftarrow \sigma_{lsv}(\mathtt{ctx}, \mathtt{lsv})\ \mathtt{IN}\ \lfloor(\mathtt{op}, \mathtt{args})\rfloor_{\mathtt{ctx}}$$

REMARK ON THE MEMORY DEPENDENCY OF SYMBOLIC OPERATIONS In Definition 6.2.1, I explained that unlike RTLpath, arithmetic operations (in fact it also applies to unsigned conditions) in BTL symbolic values do not need to depend on the current memory of the block. This improvement is possible thanks to Lemma 6.3.1 below, stating that their semantics only depend on the initial memory of the block.

Lemma 6.3.1 (Symbolic memory preserves valid pointers [⋄])**.** *The `Mem.valid_pointer` property returns true if the memory address at `(b,ofs)` has at least the "non-empty" permission (cf. §3.3.3, knowing that freeable, writable, and readable imply non-empty[6]) in `(cm0 ctx)`, and false if it is empty.*

Hence, the lemma states that the aforementioned pointer validity w.r.t. an initial memory of a given context still holds after evaluating a new symbolic memory under the same context (assuming the evaluation succeeds):

```
Lemma valid_pointer_preserv ctx sm:
  ∀ m b ofs, σ_sm(ctx, sm) = Some m →
  Mem.valid_pointer (cm0 ctx) b ofs = Mem.valid_pointer m b ofs
```

Proof. By using the fact that memory stores preserve valid pointers. □

[5] We reuse the option monad notation introduced by Definition 5.3.3.

[6] For more information about this, please refer to the documentation at [⋄].

6.3.2 *Relation Between Abstract Invariants and Concrete Registers*

We first define a matching relation between a symbolic register state—simply defined as a function "$reg \to sval$ option"—and a concrete one, under a block level execution context:

Definition 6.3.4 (Matching relation between a symbolic and a concrete regset [◇])**.** The symbolic regset sreg and concrete regset rs match if and only if:

$$\texttt{match\_sreg}\ (\texttt{ctx} : \mathit{iblock}_{ctx})\ (\texttt{sreg} : \mathit{reg} \to \mathit{sval}\ \texttt{option})\ (\texttt{rs} : \texttt{regset}) \triangleq$$
$$\forall \texttt{r}\ \texttt{sv}, \texttt{sreg r} = \texttt{Some sv} \implies \sigma_{\texttt{sv}}(\texttt{ctx}, \texttt{sv}) = \texttt{Some}(\texttt{rs\#r})$$

Second, we write a validity predicate on FPASVs (still under the block level context):

Definition 6.3.5 ("Ok" predicate for the trapping values of a FPASV [◇])**.**

$$\texttt{si\_ok ctx si} \triangleq \forall \texttt{sv}, \texttt{sv} \in \texttt{si.(fpa\_ok)} \implies \sigma_{\texttt{sv}}(\texttt{ctx}, \texttt{sv}) \neq \texttt{None}$$

Lastly, the combination of Predicates 6.3.4 and 6.3.5 forms the relation:

Definition 6.3.6 (Matching relation between a FPASV and a concrete regset [◇])**.**

$$\texttt{match\_si ctx si rs} \triangleq \texttt{si\_ok ctx si} \wedge \texttt{match\_sreg ctx si rs}$$

When the predicate's relation is true, it means that the trapping instructions of si are evaluating correctly and that they match the concrete regset.

This relation serves us to define the semantics of concrete invariants in §6.3.3.3.

6.3.3 *Linking Symbolic Values and Invariants*

We now define functions to access, set and *substitute* symbolic values within invariants, and especially convert sequential invariants into their abstract (i.e. tree) representation with symbolic values. The semantics of concrete invariants, based on this conversion, is formalized subsequently, along with some basic and *syntactic* properties needed to perform the symbolic test.

6.3.3.1 *Substitutions in Concrete Invariants*

Compact invariants are defined as a sequence of pairs where the first element is a register, and the second one a value of type ival (from Definition 6.2.3). This dedicated type is practical for oracles to generate, but we still need to convert ival values to symbolic ones in order to give them a semantics.

We start by considering ireg, the simplest case: it should either be translated as an input symbolic value; or as a more complex symbolic term taking into account previous assignments in the invariant sequence, according to its force_input field's value.

Definition 6.3.7 (Invariant registers substitution)**.** For an ireg, and given the substitution function to apply (I explain in §6.3.3.2 how to instantiate the latter), we have:

```
Definition ir_subst (subst: reg → sval) (ir: ireg): sval :=
  if ir.(force_input) then fSinput ir else subst ir
```

From here, we can easily extend the process to list of invariant registers (into list_sval):

```
Fixpoint lsvof (subst: reg → sval) (l: list ireg): list_sval :=
  match l with
  | nil ⇒ fSnil
  | ir::l ⇒ fScons (ir_subst subst ir) (lsvof subst l)
  end
```

These constructs are immediately reused to handle the invariant operation type:

Definition 6.3.8 (Invariant operations substitution)**.** Given an invariant operation of type root_op, the translation to a (non hash-consed) symbolic value is simply obtained by first substituting the arguments and second converting it with the root_apply function of Definition 6.2.3.

```
Definition rop_subst (subst: reg → sval) (rop: root_op) (lr: list ireg): sval :=
  let lsv := lsvof subst lr in
  root_apply rop (lsvof subst lr) fSinit
```

Finally, the conversion for an $ival$ is simply:

Definition 6.3.9 (Invariant values substitution)**.**

```
Definition iv_subst (subst: reg → sval) (iv: ival): sval :=
  match iv with
  | Ireg ir ⇒ ir_subst subst ir
  | Iop rop args ⇒ rop_subst subst rop args
  end
```

6.3.3.2 *Coerce From Concrete to Abstract Invariants*

Even if having both forms of invariants is interesting in practice, one would like to avoid proving some properties twice by defining a *coercion* between them. It is this coercion that reduces the semantic specification of compact invariants to the one of abstract invariants.

For a formal definition of this conversion from sequential to parallel invariants, please refer to the online Coq code.

The goal is to transform, exactly as shown in Example 6.2.1, a CSASV into its FPASV equivalent. Firstly, we need a function to *set* a value inside a FPASV: to satisfy the wellformedness property $\texttt{fpa\_wf}$, the new symbolic value must be added to both the $\texttt{fpa\_ok}$ list and the $\texttt{fpa\_reg}$ map. This results in a "$\texttt{si\_set} : reg \to sval \to \texttt{fpasv} \to \texttt{fpasv}$" function [◇].
Secondly, given a compact invariant $\texttt{csi}$, we decompose its conversion in two steps:

1. A naive accumulation "$\texttt{exec\_seq} : \overrightarrow{(reg * ival)} \to \texttt{fpasv} \to \texttt{fpasv}$" [◇] from the sequence of assignments to the new FPASV. The idea is to build the FPASV by accumulation, starting with an *empty* FPASV. For each (r, iv) tuple of the sequential invariant, the function updates the value r of the current FPASV (i.e. the accumulator) with the result of $\texttt{iv\_subst}$ on iv. The substitution function given to $\texttt{iv\_subst}$ is an extension of the accumulator's $\texttt{fpa\_reg}$ as a total function, returning $\texttt{fSinput}(r)$ for any undefined register r in the map.

2. A filtering step based on the $\texttt{csi.(output)}$ set of live variables "$\texttt{build\_alive} : (\texttt{subst} : reg \to sval) \to \texttt{regset} \to (reg \mapsto sval\ \texttt{option})$" [◇]. In fact, step 1. took *every* assignment from the $\texttt{csi.(aseq)}$ list, but only those whose result register appears in the "output" set must be added to $\texttt{fpa\_reg}$. Hence, $\texttt{build\_alive}$ creates a new, filtered $\texttt{fpa\_reg}$ map from a total function of the map built in the first step, and the "output" register set of the sequential invariant.

Definition 6.3.10 (Conversion from CSASV to FPASV [◇])**.** Following the two steps from above, we name $\texttt{siof}$ the semantics of the compact representation as abstract symbolic invariants. Below, we note "$\chi : (reg \mapsto sval\ \texttt{option}) \to reg \to sval$" the function that extends an abstract invariant's symbolic regset as a total function (with the behavior described in step 1. above).

```
Program Definition siof (csi: csasv): fpasv :=
let si := exec_seq csi.(aseq) si_empty in
{| fpa_ok := fpa_ok si;
   fpa_reg := build_alive (χ si) csi.(outputs) |}
```

Discharging the proof's obligation here is trivially done by using lemmas from the Coq `PTree` library (used for the $\texttt{fpa\_reg}$ implementation) and list induction.

6.3.3.3 *Concrete Semantics of Our Invariants*

From a record of sequential invariants (Definition 6.2.5), we leverage the coercion of §6.3.3.2 to reuse the matching relation of Definition 6.3.6 for each kind of invariant. The HI is supposed to match the initial regset from the context (which corresponds to that of the source concrete BTL state), while the GI is expected to match the final regset (the one of the target BTL state).

Definition 6.3.11 (Matching relation between both invariants and a concrete regset [◇]).

$$\begin{aligned}&\texttt{match\_invs}\ (\texttt{ctx}:\ \mathit{iblock}_{ctx})\ (\texttt{csix}:\texttt{invariants})\ (\texttt{rs}:\ \texttt{regset}) \triangleq\\ &\quad \texttt{match\_si ctx csix.(history) ctx.(crs}_0\texttt{)} \wedge\\ &\quad \texttt{match\_si ctx csix.(glue) rs.}\end{aligned}$$

Recall the concrete state Relation 6.1.3: ctx is instantiated with the register set of the source state, and rs with the one of the target (i.e. transformed) state.

6.3.3.4 *Liveness and Freedom Properties*

Before simulating block-by-block a BTL function transformed by an oracle, we need to ensure that invariants at the CFG *entry point* do not constrain any register, except liveness (of the function parameters). This is because in RTL and BTL semantics, registers except function parameters have an undefined initial value. And in our intra-procedural verification, the value of parameters must be considered as unknown. Technically, it means that entry invariants (both GIs & HIs) can only contain live variables (in the output field of CSASVs) but the sequence of assignments *must be empty* (the corresponding check is done just after returning from an oracle, see §7.5.4). More formally, entry invariants must satisfy:

Definition 6.3.12 ("Only liveness" property [◇]). We chose to write this definition on abstract invariants to facilitate future generalizations, but the actual test implementation operates on the compact form. In addition, such high-level characterization is easier to manipulate in the simulation proof. Given a parallel invariant si, we pose:

$$\begin{aligned}&\texttt{only\_liveness si} \triangleq \texttt{si.(fpa\_ok)} = [] \wedge\\ &\qquad\qquad \forall \texttt{r sv, si.(fpa\_reg)!r} = \texttt{Some sv} \implies \texttt{sv} = \texttt{fSinput(r)}\end{aligned}$$

Considering the conversion from a CSASV, if the above property holds, nothing was added to fpa_ok because the assignment sequence was empty; and variables from output were necessarily mapped to input symbolic values (by definition of build_alive). We implemented the trivial checker to validate this property and proved that if it succeeds, then the above definition holds.

Furthermore, in the presence of calls or built-ins, we also need to check that the applied invariant (i.e. the output one, after the block's SE) does not constrain the *output register* of the final instruction. Indeed, in the concrete execution, the value of this register will surely be clobbered by the callee, so we have to consider it as unknown. If the FPASV does not constrain a register r, we say it is free of r:

Definition 6.3.13 (Invariant freedom for a given register [◇]). Let si be the abstract invariant; it is free of register "r" if and only if:

$$\begin{aligned}\texttt{sifree(si, r)} \triangleq\ &\forall \texttt{sv, sv} \in \texttt{si.(fpa\_ok)} \implies \texttt{svfree(sv, r)} \wedge\\ &\forall \texttt{r}' \texttt{ sv, si.(fpa\_reg)!r}' = \texttt{Some sv} \implies\\ &\forall\ \texttt{hc, sv} \neq \texttt{Sinput(r}', \texttt{hc)} \implies \texttt{svfree(sv, r)}\end{aligned}$$

Both freedom definitions accept constraints like $\forall \texttt{hc}, \texttt{r} = \texttt{Sinput(r, hc)}$ *in the invariant because they are viewed as the trivial equation* $\texttt{r} = \texttt{r}$.

Incidentally, the constraint also applies to the memory, since the callee might modify it:

Definition 6.3.14 (Invariant freedom for the memory [◇]). Likewise, the abstract invariant si is said free from the memory if and only if:

$$\texttt{sifreem(si)} \triangleq \forall \texttt{sv, sv} \in \texttt{si.(fpa\_ok)} \implies \texttt{svfreem(sv)}$$

Where svfree [◇] (respectively svfreem [◇]) is a recursive property over a symbolic value sv that holds when sv *does not depend* on the given register (respectively the memory). Fortunately, simple syntactic verifications on the compact invariant are sufficient to validate or not both properties: we only have to ensure that the destination register of the call or built-in (or a memory dependent operation) *does not appear* in the sequence of assignments. Of course, the register may be live (i.e. in the "output" register set of the invariant). Here again, the defensive check must be performed on both types of invariants. This is done before applying the output invariant on the final symbolic

state—if the final instruction is a call or a built-in—by a syntactic checker ensuring that the invariant is free from the memory and from the result register of the instruction.

The compatibility of a CSASV w.r.t. a call/built-in result register is defined by grouping the freedom properties with a weaker only-live property concerning the result only (i.e. and not all registers as in Definition 6.3.12).

Definition 6.3.15 (Clobbered register compatibility with compact invariant [◇]). Observe that our property coerces compact to abstract invariants with `siof` below:

$$\begin{aligned}&\texttt{clobbered\_compat}\ (\texttt{csi} : \texttt{csasv})\ (\texttt{res} : \texttt{reg}) \triangleq \\ &\quad \texttt{sifree}(\texttt{siof}(\texttt{csi}), \texttt{res}) \ \wedge\ \texttt{sifreem}(\texttt{siof}(\texttt{csi})) \wedge \\ &\quad \forall \texttt{sv}, \texttt{siof}(\texttt{csi})!\texttt{res} = \texttt{Some}\ \texttt{sv} \implies \texttt{sv} = \texttt{fSinput}(\texttt{res})\end{aligned}$$

This illustrates how switching between representations can be handy to formalize properties that help writing proofs. Of course, neither `clobbered_compat` nor `siof` are executed (nor extracted) during the compilation process, since the actual compatibility check operates directly on CSASVs.

6.4 SYMBOLIC SEMANTICS OF BTL BLOCKS (*IN RELATION TO THEIR CONCRETE SEMANTICS*)

Our theory of symbolic execution improves upon Six et al. [135]†'s with mainly two features. First, liveness checking is optionally performed during symbolic execution. Indeed, we will see in §6.5 that the SE of the source block is performed without any liveness checking (similarly to [135]†), while the SE of the target block raises an error as soon as an undefined register is read: with the few additional checks given in §6.3.3.4, this suffices to validate the liveness information implicitly given by invariants. In the following, we thus distinguish between the "source" and the "target" mode. Only the latter involves liveness checking. Second, BTL SE supports arbitrary nested sequences of "if-then-else" instead of superblocks only. This required both a more general representation of the symbolic states generated by the SE and a kind of *trace-partitioning* within it.

We know sketch these ideas: §6.4.1 starts by introducing the symbolic syntax and semantics of BTL final values, and the evaluation function for conditional branches. Then, the simulation context instantiation is explained in §6.4.2, the syntax of symbolic states is provided in §6.4.3, and §6.4.4 details the execution semantics.

6.4.1 *Prerequisite: Symbolic Representations for Final Instructions and Conditions*

6.4.1.1 *Syntax and Semantics for Final Symbolic Values*

For BTL final instructions (of Figure 5.1), we define a separated symbolic type:

Definition 6.4.1 (Final symbolic values [◇]). Their syntax mimics the BTL *final* type:

$$\begin{aligned}\mathit{sfval} \quad ::= \quad & \texttt{Sgoto}(\mathit{pc}_{\text{succ}}) \mid \texttt{Scall}(\mathit{sig}, (\mathit{sval} \mid \mathit{id}), \mathit{list\_sval}, \mathit{reg}_{\text{dst}}, \mathit{pc}_{\text{succ}}) \\ \mid \quad & \texttt{Stailcall}(\mathit{sig}, (\mathit{sval} \mid \mathit{id}), \mathit{list\_sval}) \mid \texttt{Sbuiltin}(\mathit{ef}, \overrightarrow{\mathit{sval}_{\text{bargs}}}, \mathit{reg}_{\text{bdst}}, \mathit{pc}_{\text{succ}}) \\ \mid \quad & \texttt{Sjumptable}(\mathit{sval}_{\text{arg}}, \overrightarrow{\mathit{pc}_{\text{succ}}}) \mid \texttt{Sreturn}(\epsilon \mid \mathit{sval})\end{aligned}$$

except that source registers are replaced by symbolic values.

The semantics for values of the *sfval* type [◇] is very similar to the BTL *final* values' semantics (of Definition 5.3.5), except that it is parametrized by the function context of Definition 6.3.2:

```
Inductive sem_sfval (ctx: fct_iblock_ctx) Σ⃗: sfval → regset → mem → trace → state → Prop
```

Moreover, when the final instruction to evaluate contains symbolic values (e.g. the optional return value of `Sreturn`; the arguments of a `Scall`, `Stailcall`, or `Sbuiltin`; and the index value of a `Sjumptable`) then the semantics assumes they evaluate correctly. In the same fashion, the `find_function` from Definition 5.3.5 is replaced by a symbolic equivalent, that either evaluates the symbolic value containing the target function or searches for the identifier (according to the sum

type $(sval|id)$) in the global environment of the execution context. Note that we had to define specific evaluation functions for built-ins' arguments and result registers since their type is an inductive definition as briefly described in §5.2.2 (i.e. we have a symbolic evaluation property `eval_builtin_sargs` semantically equivalent to the concrete `eval_builtin_args` from mainline CompCert).

6.4.1.2 *Conditional Branches*

Observe that conditions are *not* directly represented in symbolic values, since branching is *implicitly encoded* by the structure of the binary decision tree formed during SE (see §6.4.3). Hence, we define the evaluation of conditions apart from the σ function, as:

Definition 6.4.2 (Evaluating symbolic conditions)**.**

```
Definition eval_scondition ctx (cond: condition) (lsv: list_sval): option bool :=
  SOME args ← σ_lsv ctx lsv IN eval_condition cond args (cm0 ctx)
```

6.4.2 *Instantiating Contexts*

We saw in §6.3.1 that evaluation was operating under a context of execution for the block, itself encapsulated into a larger context including the current function for the semantics of final symbolic values. Following this principle, we *instantiate* the whole symbolic simulation under a context to remember the *global environments* of both the source and target blocks:

Definition 6.4.3 (Simulation context [◇])**.**

$$\begin{aligned} simu_{ctx} \triangleq \{\, & \mathtt{sG}_1 : BTL.genv;\ \mathtt{sG}_2 : BTL.genv; \\ & \mathtt{sG\_match} : \forall \mathtt{s}, \mathtt{find\_symbol}(\mathtt{sG}_1, \mathtt{s}) = \mathtt{find\_symbol}(\mathtt{sG}_2, \mathtt{s}); \\ & \mathtt{ssp} : val;\ \mathtt{srs}_0 : regset;\ \mathtt{sm}_0 : mem \,\} \end{aligned}$$

Here, the sG_match dependent property indicates that both environments have the same set of symbols (i.e. identifiers).

The (refined) implementation of the simulation test (Chapter 7) thus *implies* the main simulation predicate for any context; with the `instantiate_context` predicate visible in Definition 6.1.1 and defined below:

Definition 6.4.4 (Instantiating the global context of simulation [◇])**.**

$$\begin{aligned} &\mathtt{instantiate\_context}\ (\mathtt{P} :\ simu_{ctx} \to gm \to iblock \to iblock \to pc \to \mathtt{Prop})\ (\mathtt{gm} :\ gm) \\ &\quad (\mathtt{ib}_1\ \mathtt{ib}_2 :\ iblock)\ (\mathtt{pc} : pc) \triangleq \forall (\mathtt{ctx} :\ simu_{ctx}), \mathtt{P\ ctx\ gm\ ib}_1\ \mathtt{ib}_2\ \mathtt{pc} \end{aligned}$$

Symbolic values' evaluation and states' preconditions are expected to be preserved by the context switch.

In other words, the initial values of the various contextual elements (e.g. register set, memory, etc.) do not matter, but are required to theoretically *reason* on the SE framework. When applying the gluing invariant (more information is provided in §6.5.3), a context switch is performed from the source context to the target one[7]. Both block level contexts are built from this simulation context, using $\mathtt{sG}_1$ for the source global environment and $\mathtt{sG}_2$ for the target. In the following, we note $\mathtt{Bctx}_1$ and $\mathtt{Bctx}_2$ [◇] the functions building those contexts from a simulation context.

6.4.3 *Symbolic States*

A symbolic state represents all possible blocksteps of a given BTL block. Formally, we define them as binary decision trees: each branch of the tree represents one possible execution path of the block.

Definition 6.4.5 (Symbolic states syntax [◇])**.** Type *sstate* is thus inductive on Scond, and has two kinds of leaves: either "Sabort" as an error case (e.g. the concrete branch never reaches a final instruction, or the liveness information is incorrect) or "Sfinal(*sistate*, *sfval*)" for final states.

[7] Such a switch is required to prove the overall block forward simulation of §6.1, as illustrated at the top of Figure 6.2.

In the latter, *sistate*—*a symbolic internal state*—represents the *preconditioned parallel assignments* realized by the sequence of basic instructions of the branch, and *sfval*—*a symbolic final value*, from Definition 6.4.1—represents the final instruction run by the block (where registers have been substituted by their final symbolic values).

$$sstate ::= \texttt{Sfinal}(sistate, sfval) \mid \texttt{Scond}(cond, list\_sval, sstate_{\text{so}}, sstate_{\text{not}}) \mid \texttt{Sabort}$$

Internal states are characterized in a quite abstract way (w.r.t. their refined variant of §7.2.1):

Definition 6.4.6 (Symbolic internal states syntax [⋄])**.** An internal state includes a precondition which must hold under the initial block execution context, and a dependent property ensuring the preservation of this precondition from context switching:

$$\begin{aligned} sistate \triangleq \{ & \texttt{sis\_pre} : iblock_{ctx} \to \texttt{Prop};\ \texttt{sis\_sreg} :> reg \to sval\ \texttt{option}; \\ & \texttt{sis\_smem} : smem; \\ & \texttt{sis\_pre\_preserved} : \forall(\texttt{pctx} : simu_{ctx}), \\ & \qquad \texttt{sis\_pre}(\texttt{Bctx}_1\ \texttt{pctx}) \iff \texttt{sis\_pre}(\texttt{Bctx}_2\ \texttt{pctx}) \} \end{aligned}$$

Observe that both the precondition and the register set are abstract (the former being in Prop, and the latter devoid of a concrete data structure, unlike the map of FPASVs).

Property sis_pre_preserved of such states appears in every Coq `Program`'s proof obligation defining a new state, and is deduced from three trivial lemmas on the mutual variants of symbolic values (*sval*, *list_sval*, and *smem*). They are summarized by lemma:

Lemma 6.4.1 (Sigma-evaluations are preserved by context switching [⋄])**.**

$$\forall(\texttt{ctx} : iblock_{ctx})\ sv, \sigma_{(sv|lsv|sm)}(\texttt{Bctx}_1\ \texttt{ctx}, sv) = \sigma_{(sv|lsv|sm)}(\texttt{Bctx}_2\ \texttt{ctx}, sv)$$

Proof. By mutual induction on sv, then by rewriting existing preservation properties about the evaluation of addressing modes and operations when the global environment changes. □

If we only consider a single execution path (terminated by a `Sfinal` leaf of an *sstate*), the outcome of the simulation is a record:

```
Record soutcome ≔ sout { _sis: sistate; _sfv: sfval }
```

obtained by recursively evaluating the considered path of the symbolic state:

Definition 6.4.7 (Computing the outcome of a symbolic state)**.**

```
Fixpoint get_soutcome ctx (ss:sstate): option soutcome ≔
  match ss with
  | Sfinal sis sfv ⇒ Some (sout sis sfv)
  | Scond cond args ifso ifnot ⇒
     SOME b ← eval_scondition ctx cond args IN
     get_soutcome ctx (if b then ifso else ifnot)
  | Sabort ⇒ None
  end
```

This function is partial, so that it returns `None` if the SE failed with an `Sabort` state.

In the simulation proof on concrete execution (which is formalized in paragraph §6.4.3.2), we reason (by case analysis) on a single blockstep. The latter step corresponds to the selection of an execution path of the symbolic state from the initial context: this is precisely what this function does. It is therefore very practical, because it will serve us to link symbolic and concrete executions (e.g. in Lemmas 6.4.6 and 6.4.7).

6.4.3.1 *A Theory of Liveness Frames*

The liveness of variables in our theory is simply encoded by their *presence* in the symbolic regset. Hence, we say that a register is alive if the below notation (which determines a property) is true:

Definition 6.4.8 (Alive notation and frame construction)**.**

```
Notation "'alive' o" := (o = None → False) (at level 0)
Definition build_frame {A} (si: reg → option A) (r: reg): Prop := alive (si r)
```

Given a gluing invariant `si` (that defines the live variables of the target block) , we name "frame" the abstract liveness property obtained with "`build_frame si`". This partial application gives us a type `reg → Prop` that we can reuse to formalize the notion of regset equality modulo live registers as:

Definition 6.4.9 (Equality of regsets "$\equiv_E$" under a given frame)**.**

```
Definition eqlive_reg (frame: reg → Prop) (rs1 rs2: regset): Prop :=
  ∀ r, (frame r) → rs1#r = rs2#r
```

This relation is *reflexive*, *transitive*, and *monotonic* [◇] (i.e. if `∀ r, frame1 r → frame2 r` then `∀ rs1 rs2, eqlive_reg frame2 rs1 rs2 → eqlive_reg frame1 rs1 rs2`). It is reused to define the modulo liveness equality over stack frames and concrete states (of Definition 5.3.1).

Moreover, if we have a valid instance of the relation for a frame that does not mention a register `r`, then *updating* `r` with a value v in both register sets and in the initial frame preserves the property:

Lemma 6.4.2 (Frame-scoped equality of regsets is preserved by updates [◇])**.**

```
Lemma eqlive_reg_update (frame: reg → Prop) rs1 rs2 r v:
  eqlive_reg (λ r1 ⇒ r1 <> r ∧ frame r1) rs1 rs2 →
  eqlive_reg frame (rs1 # r ← v) (rs2 # r ← v)
```

Proof. By rewriting get/set lemmas on register sets. □

For instance, the trivial frame hypothesis of an internal state `sis` (i.e. supposing every register as live) is built by `∀ r, build_frame sis r`.

To verify the liveness information in target mode (as stated at the beginning of this section), the simulation must ensure that every register marked as live in the *successor* blocks are indeed live when reaching the final instruction of the *current* block. For that, we characterize the frame of a final symbolic value using the gluemap stored in the `BTL` function (that the oracle is supposed to have filled):

Definition 6.4.10 (Building the output frame with GIs)**.** Let `f` be the current function, and `sfv` the final value of the block. It's output frame is built as:

```
Definition sfv_frame f sfv: reg → Prop :=
  match sfv with
  | Sgoto pc ⇒ build_frame (f.(fn_gm) pc)
  | Sreturn _ | Stailcall _ _ _ ⇒ λ r ⇒ False
  | Scall _ _ _ res pc ⇒ λ r ⇒ r <> res ∧ build_frame (f.(fn_gm) pc) r
  | Sjumptable _ tbl ⇒ λ r ⇒ ∃ pc, List.In pc tbl ∧ build_frame (f.(fn_gm) pc) r
  | Sbuiltin _ _ bres pc ⇒ λ r ⇒ (∀ res, reg_builtin_res bres = Some res → r <> res)
                                 ∧ build_frame (f.(fn_gm) pc) r
  end
```

Knowing that we want to verify the set of output registers, we can eliminate the cases where there are *no successors* (e.g. for returns and tail calls), since the set of variables that should be alive is obviously *empty* if there is no code supposed to read them. That is why the frame returns `False` in those situations.

For a built-in *without* result register or a goto instruction jumping at `pc`, the frame is simply the set of live variables at the entry of the block located at `pc`. Built-ins featuring a result register and calls follow the same principle, but (as we are computing the frame *before* the final symbolic value's execution), they exclude this—dead because of the affectation—result register, no matter the content of the GI.

Finally, the most subtle case is for jump tables: as long as *there exists* a `pc` in `tbl` such as the frame built with the invariant at `pc` contains a register `r`, then it is in the frame of the jump table. Put another way, the output frame of a jump table is the union of the input frames of all successors (I come back to this concept of union in §6.5.2.2).

6.4.3.2 *Intermediate Properties*

To prove correct our simulation test, we need some simple properties about the validity of symbolic states, and about their matching relation w.r.t. concrete states. All those properties depend on the (block) execution context (type $iblock_{ctx}$).

VALIDITY For a state sis to be valid, it has to satisfy three properties. Firstly, its precondition must hold; secondly, its symbolic memory must evaluate (through the σ function) without errors; and thirdly, the values of its symbolic regset must also evaluate correctly. Below are the validity predicates for symbolic regsets and internal states.

Definition 6.4.11 ("Ok" predicate for a symbolic regset [◇])**.**

$$\mathtt{sreg\_ok\ ctx\ sreg} \triangleq \forall \mathtt{r\ sv, sreg\ r} = \mathtt{Some\ sv} \implies \sigma_{sv}(\mathtt{ctx, sv}) \neq \mathtt{None}$$

Definition 6.4.12 ("Ok" predicate for a symbolic internal state [◇])**.**

$$\mathtt{sis\_ok\ ctx\ sis} \triangleq \mathtt{sis.(sis\_pre\ ctx)} \wedge \sigma_{sm}(\mathtt{ctx, sis.(sis\_smem)}) \neq \mathtt{None} \wedge \mathtt{sreg\_ok\ ctx\ sis}$$

MATCHING RELATIONS BETWEEN SYMBOLIC AND CONCRETE STATES Matching a symbolic internal state w.r.t. a *concrete pair* $(\mathtt{rs}, \mathtt{m})$ of a regset and a memory (still under context $\mathtt{ctx}$), also necessitates a valid precondition. However, it entails stronger properties concerning the symbolic memory and regset. This time, the memory has to not only evaluate correctly, but also the result must be exactly $\mathtt{m}$. Similarly, the symbolic regset must match the concrete one (i.e. reusing Definition 6.3.4).
We define the valid relation between an internal state and a concrete pair as follows:

Definition 6.4.13 (Semantic "ok" predicate for an internal state w.r.t. a concrete state [◇])**.**

$$\begin{aligned}\mathtt{sem\_sistate\ ctx\ sis\ rs\ m : Prop} \triangleq\ & \mathtt{sis.(sis\_pre)\ ctx} \\ & \wedge\ \sigma_{sm}(\mathtt{ctx, sis.(sis\_smem)}) = \mathtt{Some\ m} \\ & \wedge\ \mathtt{match\_sreg\ ctx\ sis\ rs}\end{aligned}$$

Note that for the purpose of comparing symbolic internal states, we want to reason modulo extensionality w.r.t. register sets (as explained in §3.3.2):

Lemma 6.4.3 (The semantic relation between symbolic internal and concrete states is deterministic [◇])**.** *Let* sis *be a state for which* $\mathtt{sem\_sistate}$ *holds for two pairs* $(\mathtt{rs}_1, \mathtt{m}_1)$ *and* $(\mathtt{rs}_2, \mathtt{m}_2)$*:*

$$\begin{aligned}&\mathtt{sem\_sistate\ ctx\ sis\ rs}_1\ \mathtt{m}_1 \implies \mathtt{sem\_sistate\ ctx\ sis\ rs}_2\ \mathtt{m}_2 \implies \\ &\mathtt{eqlive\_reg\ (build\_frame\ sis)\ rs}_2\ \mathtt{rs}_1 \wedge \mathtt{m}_1 = \mathtt{m}_2\end{aligned}$$

Proof. By rewriting the $\mathtt{match\_sreg}$ *property of the second hypothesis.* □

Furthermore, observe that for any context, state, regset, and memory, if property 6.4.13 is valid, it always implies property 6.4.12[8]:

Lemma 6.4.4 (Semantic "ok" implies valid "ok" [◇])**.**

$$\forall \mathtt{ctx\ sis\ rs\ m, sem\_sistate\ ctx\ sis\ rs\ m} \implies \mathtt{sis\_ok\ ctx\ sis}$$

Proof. By construction. □

By extension of Definition 6.4.13, one can deduce the semantic relation between a full symbolic state "$ss : sstate$" and a concrete CompCert transition. The latter being a pair (cs, e) of a concrete state and an observable trace that might be produced by ss under the initial (wrapped with function) context "$\mathtt{ctx} : fct\_iblock_{ctx}$".

Definition 6.4.14 (Semantic "ok" predicate for state transition)**.** To make this relation usable in both source and target reasoning, the symbolic value final frame ("SVFF") is passed as a parameter. In addition to the trace e, the concrete state cs, and the symbolic state ss, the predicate takes into account the call stack $\vec{\Sigma}$.

[8] And we have, logically, the same kind of implication between predicate 6.3.4 and 6.4.11 [◇].

```
Inductive sem_sstate (SVFF: function → sfval → reg → Prop)
(ctx: fct_iblock_ctx) ⃗Σ e cs: sstate → Prop :=
  | sem_Sfinal sis sfv rs m
     (SIS: sem_sistate ctx sis rs m)
     (LIVEOK: ∀ r, SVFF (cf ctx) sfv r → build_frame sis r)
     (SFV: sem_sfval ctx ⃗Σ sfv rs m e cs)
     : sem_sstate SVFF ctx ⃗Σ e cs (Sfinal sis sfv)
  | sem_Scond b cond args ifso ifnot
     (SEVAL: eval_scondition ctx cond args = Some b)
     (SELECT: sem_sstate SVFF ctx ⃗Σ e cs (if b then ifso else ifnot))
     : sem_sstate SVFF ctx ⃗Σ e cs (Scond cond args ifso ifnot)
  (* NB: Sabort: fails to produce a transition *)
```

ADDITIONAL PROPERTIES ON MATCHING RELATIONS Oppositely to the above relations, the below property catches situations where the SE fails:

Definition 6.4.15 (SE failure [⋄]). (The context being at the block level again)

$$\begin{aligned}\mathtt{abort\_sistate\ ctx\ sis} : \mathtt{Prop} \triangleq \neg(\mathtt{sis.(sis\_pre)\ ctx}) \\ \vee\ \sigma_{sm}(\mathtt{ctx}, \mathtt{sis.(sis\_smem)}) = \mathtt{None} \\ \vee\ \exists \mathtt{r}\ sv, \mathtt{sis\ r} = \mathtt{Some}\ sv\ \wedge\ \sigma_{sv}(\mathtt{ctx}, sv) = \mathtt{None}\end{aligned}$$

If either (i) the precondition does not hold; or (ii) the memory does not evaluate correctly; or (iii) there exists an assigned register whose symbolic value does not evaluate correctly, then the simulation aborts.

Naturally, we want to be sure that Properties 6.4.13 and 6.4.15 exclude each other:

Lemma 6.4.5 (Semantic "ok" and "ko" properties are mutually exclusive [⋄]).

$$\forall \mathtt{ctx\ sis\ rs\ m}, \mathtt{sem\_sistate\ ctx\ sis\ rs\ m} \implies \mathtt{abort\_sistate\ ctx\ sis} \implies \mathtt{False}$$

Proof. By congruence and rewriting the match_sreg *relation of* sem_sistate. □

Two other lemmas follow intuitively the definition of `sem_sstate`. First, assuming a valid semantic relation between a concrete and a symbolic state, it always exists an outcome (in the sense of Definition 6.4.7) as a pair ("sout sis sfv") for which the three properties of the `sem_Sfinal` constructor hold (Lemma 6.4.6). Second, and conversely, assuming the same outcome and the three properties of `sem_Sfinal`, it is always possible to reconstruct a valid `sem_sstate` predicate (Lemma 6.4.7).

Lemma 6.4.6 (The semantic "ok" relation with a concrete state implies a valid SE run [⋄]).

```
Lemma sem_sstate_run (SVFF: function → sfval → reg → Prop) (ctx: fct_iblock_ctx) ⃗Σ ss e cs:
  sem_sstate SVFF ctx ⃗Σ e cs ss →
  ∃ sis sfv rs m,
    get_soutcome ctx ss = Some (sout sis sfv) ∧ sem_sistate ctx sis rs m ∧
    (∀ r, SVFF (cf ctx) sfv r → build_frame sis r) ∧ sem_sfval ctx ⃗Σ sfv rs m e cs
```

Proof. Trivial induction on the first hypothesis. □

Lemma 6.4.7 (A valid SE run implies the semantic "ok" relation with a concrete state [⋄]).

```
Lemma run_sem_sstate (SVFF: function → sfval → reg → Prop) (ctx: fct_iblock_ctx) ss sis sfv:
  get_soutcome ctx ss = Some (sout sis sfv) →
  ∀ rs m ⃗Σ cs e,
  sem_sistate ctx sis rs m →
  (∀ r, SVFF (cf ctx) sfv r → build_frame sis r) →
  sem_sfval ctx ⃗Σ sfv rs m e cs →
  sem_sstate SVFF ctx ⃗Σ e cs ss
```

Proof. Trivial induction on ss. □

6.4.3.3 *Writing in Internal States*

Two operations are needed on symbolic states: assigning a value to a register and setting the memory.

Definition 6.4.16 (Assign a symbolic value to a register in a sistate [⋄]). Here again, the Coq definition use the `Program` keyword to maintain the preservation property:

```
Program Definition set_sreg (r:reg) (sv:sval) (sis:sistate): sistate :=
{| sis_pre := (λ ctx ⇒ (∀ sv, sis r = Some sv → σ_sv(ctx, sv) <> None)
                       ∧ (sis.(sis_pre) ctx));
   sis_sreg := λ y ⇒ if r = y then Some sv else sis y;
   sis_smem := sis.(sis_smem)|}
```

In the above, we ensure that the assigned symbolic value evaluates correctly, to preserve the semantic validity of the state (according to Definition 6.4.13). This is proved by:

Lemma 6.4.8 (Correctness of the register assignment function on internal states [⋄]).

```
Lemma set_sreg_correct ctx dst sv sis (rs rs': regset) m:
  sem_sistate ctx sis rs m →
  (σ_sv(ctx, sv) = Some (rs'#dst)) →
  (∀ r, build_frame sis r → r <> dst → rs'#r = rs#r) →
  sem_sistate ctx (set_sreg dst sv sis) rs' m
```

Proof. It consists of three cases (those of Definition 6.4.13):

- *Precondition preservation: the new precondition is easily proved using both the old precondition and the old regset matching property of the first hypothesis;*
- *The memory has not changed so it still evaluates well;*
- *And here we have two subcases: either* $\mathtt{dst} = \mathtt{r}$ *and this is trivial by applying the second hypothesis; or* $\mathtt{dst} \neq \mathtt{r}$ *and we can use the liveness equivalence on the frame (the third hypothesis) to solve the goal.*

□

Overwriting the memory also requires updating `sis_pre`, but this time we do not have to check if it is live, since memory is always live:

Definition 6.4.17 (Assign a new memory to a sistate [⋄]).

```
Program Definition set_smem (sm:smem) (sis:sistate): sistate :=
{| sis_pre := (λ ctx ⇒ σ_sm(ctx, sis.(sis_smem)) <> None ∧(sis.(sis_pre) ctx));
   sis_sreg := sis.(sis_sreg); sis_smem := sm |}
```

In the same way, the memory assignment is proved correct by:

Lemma 6.4.9 (Correctness of the memory assignment function on internal states [⋄]).

```
Lemma set_smem_correct ctx sm sis rs m m':
  sem_sistate ctx sis rs m →
  (σ_sm(ctx, sm) = Some m') →
  sem_sistate ctx (set_smem sm sis) rs m'
```

Proof. By rewriting the second hypothesis (properties 1 and 3 of `sem_sistate` *are trivial).* □

6.4.4 *Symbolic Execution*

Before introducing the recursive SE of BTL blocks, we begin with intermediate functions for executing lists of concrete arguments, final symbolic values and some BTL basic instructions.

6.4.4.1 *Arguments and Liveness Checking*

For concrete instructions featuring a list of arguments (i.e. pseudo-registers) `args`, we transform it into a list of symbolic values (of the *list_sval* type). This computation may fail if a register does not evaluate correctly. Moreover, in target mode, when `args` contains a dead register, the transformation returns `None`, which is then implicitly propagated to the rest of the execution by the `SOME` monadic binder (from notation in Definition 5.3.3).

Definition 6.4.18 (Mapping a register list to a symbolic value list using a symbolic regset)**.**

```
Fixpoint lmap_sv {A} (sreg: A → option sval) (l: list A): option list_sval :=
  match l with
  | nil ⇒ Some (fSnil)
  | r::l' ⇒ SOME sv ← sreg r IN SOME lsv ← lmap_sv sreg l' IN Some (fScons sv lsv)
  end
```

We say that the above function is correct if, given a matching relation between a symbolic regset $sreg$ and a concrete regset rs under context ctx, for all lists of registers l, the result of `lmap_sv` on l evaluates to $rs\#\#l$ [9].

Lemma 6.4.10 (Correctness of the arguments' execution [$\diamond$])**.** *With* ctx, $sreg$, l *and* rs, *we have:*

$$match\_sreg\ ctx\ sreg\ rs \implies \forall lsv, lmap\_sv(sreg, l) = Some\ lsv \implies \sigma_{lsv}(ctx, lsv) = Some(rs\#\#l)$$

Proof. By simple induction on l. □

6.4.4.2 *Final Values*

Given a final `BTL` instruction and a symbolic regset $sreg$, the execution is a partial function producing an *sfval* (in the option monad):

Definition 6.4.19 (SE for final instructions)**.**

```
Definition sexec_final_sfv (i: final) (sreg: reg → option sval): option sfval :=
  match i with
  | Bgoto pc ⇒ Some (Sgoto pc)
  | Bcall sig ros args res pc ⇒
    SOME svos ← sum_left_optmap sreg ros IN
    SOME sargs ← lmap_sv sreg args IN
    Some (Scall sig svos sargs res pc)
  | Btailcall sig ros args ⇒
    SOME svos ← sum_left_optmap sreg ros IN
    SOME sargs ← lmap_sv sreg args IN
    Some (Stailcall sig svos sargs)
  | Bbuiltin ef args res pc ⇒
    SOME sargs ← bamap_opt (map_builtin_arg_opt sreg) args IN
    Some (Sbuiltin ef sargs res pc)
  | Breturn None ⇒ Some (Sreturn None)
  | Breturn (Some r) ⇒
    SOME sv ← sreg r IN
    Some (Sreturn (Some sv))
  | Bjumptable reg tbl ⇒
    SOME sv ← sreg reg IN
    Some (Sjumptable sv tbl)
  end
```

Here, `sreg` is supposed to be the symbolic regset resulting from the execution of previous instructions in the block. Above, cases for gotos, returns, and jump tables are trivial. For calls and tail calls, we use an auxiliary function `sum_left_optmap: ∀ A B C : Type, (A → option B) → A + C → option (B + C)`

[9] Recall notation $rs\#\#\_$ used to access a list of arguments in rs.

that transforms the `ros` argument (which is a sum type `reg + ident`) into type `svos: sval + ident`. The SE of lists of concrete arguments is delegated to function `lmap_sv` (Definition 6.4.18). In the built-in case, we use an equivalent transformation from built-ins' registers to built-ins' symbolic values.

6.4.4.3 *Block Execution*

Let *sis* be the current symbolic internal state. Assuming a BTL "`Bop op args dst _`" instruction, its SE corresponds to the function:

```
Definition sexec_op op args dst (sis: sistate): option sistate :=
  SOME args ← lmap_sv sis args IN
  Some (set_sreg dst (fSop op args) sis)
```

Likewise, the SE functions for `Bload` and `Bstore` are:

```
Definition sexec_load trap chunk addr args dst (sis: sistate): option sistate :=
  SOME args ← lmap_sv sis args IN
  Some (set_sreg dst (fSload sis.(sis_smem) trap chunk addr args) sis)
```

and

```
Definition sexec_store chunk addr args src (sis: sistate): option sistate :=
  SOME args ← lmap_sv sis args IN
  SOME src ← sis src IN
  Some (set_smem (fSstore sis.(sis_smem) chunk addr args src) sis)
```

Our SE model of a BTL block starts from an initial internal state and constructs (by computing recursively over the block) a symbolic state of type *sstate*. Alike the SOME monadic binder notation, we write a STBIND notation which returns `Sabort` in case a `None` value is caught in the option monad:

Definition 6.4.20 (Option monad binder to Sabort)**.**

```
Notation "'STBIND' X ← A 'IN' B" :=
  (match A with Some X ⇒ B | None ⇒ Sabort end)
  (at level 200, X name, A at level 100, B at level 200) : option_monad_scope
```

By lifting `None` to `Sabort`, we implicitly propagate errors from access to dead registers.

The trace partitioning is then realized in continuation passing style (CPS) as follows:

Definition 6.4.21 (Recursive SE of a BTL block)**.** With `sinit` the initial internal symbolic state, and `ib` the block to simulate:

```
Fixpoint sexec_rec ib (sis:sistate) (k: sistate → sstate): sstate :=
  match ib with
  | BF fin _ ⇒
    STBIND sfv ← sexec_final_sfv fin sis IN Sfinal sis sfv
  (** basic instructions *)
  | Bnop _ ⇒ k sis
  | Bop op args res _ ⇒
     STBIND sis' ← sexec_op op args res sis IN k sis'
  | Bload trap chunk addr args dst _ ⇒
     STBIND sis' ← sexec_load trap chunk addr args dst sis IN k sis'
  | Bstore chunk addr args src _ ⇒
     STBIND sis' ← sexec_store chunk addr args src sis IN k sis'
  (** composed instructions *)
  | Bseq ib1 ib2 ⇒
      sexec_rec ib1 sis (λ sis2 ⇒ sexec_rec ib2 sis2 k)
  | Bcond cond args ifso ifnot _ ⇒
      let ifso := sexec_rec ifso sis k in
      let ifnot := sexec_rec ifnot sis k in
      STBIND args ← lmap_sv sis args IN Scond cond args ifso ifnot
  end

Definition sexec ib sinit := sexec_rec ib sinit (λ _ ⇒ Sabort)
```

Continuation `k` represents how SE should "normally" continue on updates of the internal state. It is initialized as "`(λ _ ⇒ Sabort)`" reflecting the fact that each BTL blockstep must reach a final instruction.

To ensure that our SE function is a correct model, we prove that each concrete execution (as defined in Chapter 5) can be executed on the symbolic state produced from `sexec`.

Theorem 6.4.11 (The SE is a correct over-approximation [◇])**.**

```
Theorem sexec_correct (ctx: fct_iblock_ctx) sinit ⃗Σ ib e s:
  iblock_step (cG ctx) ⃗Σ (cf ctx) (csp ctx) (crs0 ctx) (cm0 ctx) ib e s →
  (* NB: the two properties below give the correctness property of
     an "history invariant" *)
  sem_sistate ctx sinit (crs0 ctx) (cm0 ctx) → (* prop 1 *)
  (∀ r : reg, build_frame sinit r) →           (* prop 2 *)
  sem_sstate triv_frame ctx ⃗Σ e s (sexec ib sinit)
```

Proof. The proof is an induction on `ib`*. It is quite straightforward thanks to the trivial liveness assumption.* □

The correctness property is exploited in §6.1.2 to justify the existence of the symbolic states corresponding to the source concrete states (both the initial and final ones). As in source mode, the execution does not involve liveness verification, the `triv_frame` *above is the trivial frame for final symbolic values (i.e. a function "*`λ (_: function) (_: sfval) (_: reg) ⇒True`*").*

To formally verify the block-by-block simulation, we will also need the reverse property, stating that each execution of a symbolic state produced from `sexec` represents a concrete execution. We thus prove it thanks to the below theorem:

"Inverting" an hypothesis means introducing a goal for each non self-contradictory constructor of a (co-)inductive type. See the Coq "inversion" tactic documentation online.

Theorem 6.4.12 (The SE is exact [◇])**.**

```
Theorem sexec_exact (ctx: fct_iblock_ctx) ⃗Σ ib e s1 sinit rs m:
  sem_sistate ctx sinit rs m →
  sem_sstate sfv_frame ctx ⃗Σ e s1 (sexec ib sinit) →
  ∃ s2, iblock_step (cG ctx) ⃗Σ (cf ctx) (csp ctx) rs m ib e s2
        ∧ s1 ≡ₜ s2
```

Proof. Here also, we reason by induction on `ib`*. The final case goal is solved by inverting the* `sem_sstate` *hypothesis and by applying Lemma 6.4.3 to recover an extensional equality between regsets. The induction for* `Bseq` *requires applying Lemma 6.4.5 (on the left side) to prove it does not fail. Finally, the exactness is first shown w.r.t.* `iblock_istep_run` *and relies on Lemma 5.3.1 to reach the final goal.* □

Recall the "$\equiv_t$*" notation to indicate equality of states modulo the liveness from GIs. This exactness theorem serves us in the same place as the correctness one (cf. §6.1.2), to prove the reverse direction: coming back to concrete states from a symbolic state computed in target mode (and so we use the non-trivial* `sfv_frame` *liveness function for final symbolic values).*

6.5 SIMULATION PREDICATE MODULO ABSTRACT INVARIANTS

Examples 1.2.1 (for GIs) and 5.1.1 (for both GIs & HIs) gave an idea of the simulation scheme, and in particular on how (and *when*) to execute invariants. I informally describe this scheme in §6.5.1, and its formalization in our SE theory w.r.t. symbolic states in §6.5.2. The overall simulation predicate is finally built in §6.5.3, Definition 6.5.14.

6.5.1 *Simulation Scheme*

We aim to deduce the simulation between blocks from the simulation of the symbolic states produced by their SE. The simulation of a symbolic state ss_1 by a symbolic state ss_2 is defined as a *relation* written $ss_1 \succeq ss_2$, which holds if and only if (i) both decision trees have the same structure with identical conditions; (ii) their leaves are pairwise identical except for the *definiteness condition*: ss_2 leaves may assign registers not defined in ss_1—their value is simply ignored, these registers being interpreted as dead; (iii) the trapping symbolic values in ss_1 leaves include those of in ss_2 leaves.

We have seen that the set of registers defined in ss_1 depends on the source or target mode. In source mode, all registers are defined, thus the definiteness condition is trivial; it is only non-trivial in target mode. To make this explicit, we note "$\succeq_t$" if ss_1 is viewed in the target mode, or "$\succeq_s$" otherwise.

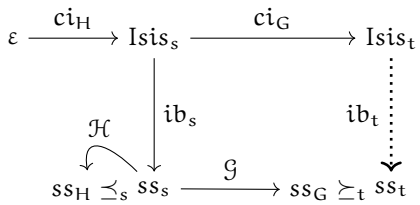


Figure 6.3: Symbolic Simulation of ib_s by ib_t.

We now explain how we reduce invariant preservation with liveness checking to comparison of symbolic states. The computations performed by our symbolic simulation test are pictured in Figure 6.3. Blocks ib_s and ib_t represent respectively the source and the target. Invariants ci_G and ci_H are respectively the gluing and the history invariants annotating the target block entry point. Each arrow represents a SE of either a block or some invariants. The bold dotted arrow, labelled by ib_t, means that its SE is in target mode. Otherwise, SEs are in source mode. Hence, starting from an empty (symbolic) internal state written ε, the execution of input HI ci_H produces a (symbolic) internal state $Isis_s$. From $Isis_s$, the SE of ib_s produces state ss_s. Still from $Isis_s$, execution of ci_G produces an internal state $Isis_t$, from which execution of ib_t in target mode produces ss_t. From ss_s, executing gluing (respectively history) invariants on each exit—as represented by the $\mathcal{G}$ (respectively $\mathcal{H}$) arrow—produces a new state ss_G (respectively ss_H). Finally, we check the preservation of GIs with the comparison $ss_G \succeq_t ss_t$. And, we check—*independently* of GIs—the preservation of HIs with the comparison $ss_s \succeq_s ss_H$ (actually, if the comparison is true then $ss_H = ss_s$).

Two technical details of this big picture now need to be clarified. First, our checker validates that GIs $\mathcal{G}$ do not trap. When computing ss_s, it also needs to remember that ci_G has not trapped. In Figure 6.3, this corresponds to consider that the may-trapping expressions of ci_G & $\mathcal{G}$ are actually implicitly part of ci_H & $\mathcal{H}$ [10]. Second (as stated in §6.3.3.4), when applying invariants of $\mathcal{G}$ (or of $\mathcal{H}$) to a call or built-in, we also need to check that they do not constrain the destination register (i.e. ensuring property 6.3.15).

6.5.2 *Application of Invariants on States*

A symbolic internal state is transferred from the application of an invariant. In the theory, the transfer is a high-level abstraction operating on FPASVs (its implementation is explained in §7.3.4), which involves updating the state's precondition, and changing the function of its symbolic register set.

Definition 6.5.1 (Transfering symbolic internal states [⋄]). Let sis and si be the state to transfer and the FPASV, respectively. The tr_sis function below takes a Boolean sis_input_init indicating the default *initialization mode* for the symbolic regset: when it is true, then every register not in the invariant is set to its input value (i.e. source mode); otherwise, unmentioned registers are left to None (i.e. target mode).

$$
\begin{array}{l}
\texttt{tr\_sis sis si sis\_input\_init} \triangleq \\
\{\quad \texttt{sis\_pre} \quad = \quad \lambda \texttt{ctx}. \quad \texttt{sis.(sis\_pre) ctx} \wedge \texttt{ sreg\_ok ctx sis} \wedge \\
\qquad\qquad\qquad\qquad\quad \forall sv, sv \in \texttt{si.(fpa\_ok)} \implies \\
\qquad\qquad\qquad\qquad\qquad\quad \sigma_{sv}(\texttt{ctx}, [(\texttt{sis.(sis\_smem)}, \texttt{sis})/sv]) \neq \texttt{None}; \\
\qquad \texttt{sis\_sreg} \quad = \quad \lambda r. \quad \textbf{match } \texttt{si!r } \textbf{with} \\
\qquad\qquad\qquad\qquad\quad \mid \texttt{Some } sv \to \texttt{Some}([(\texttt{sis.(sis\_smem)}, \texttt{sis})/sv]) \\
\qquad\qquad\qquad\qquad\quad \mid \texttt{None} \to \texttt{sis\_input\_init ? Some(fSinput(r)) : None};
\end{array}
$$

[10] In other words, the traps of $\mathcal{G}$ are part of H in the "$H(r_s, m) \wedge r_t \equiv_t G(r_s, m)$" semantics given in §5.1.

$$\mathtt{sis\_smem} = \mathtt{sis.(sis\_smem)};\ \mathtt{sis\_pre\_preserved} = \rho\,\}$$

Where the notation between square brackets refers to the substitution function sv_subst for parallel symbolic invariants whose signature looks like "$[(\mathtt{substm} : \mathit{smem}, \mathtt{subst} : \mathit{reg} \to \mathit{sval}\ \mathtt{option})/sv : \mathit{sval}] \to \mathit{sval}$" [⋄]. The two parameters substm and subst are the new symbolic memory and registers (respectively) to substitute *inside* sv. When sv is a symbolic operation, fold right, load, or store, the substitution is applied on (and thus replaces) their symbolic arguments; the mutual recursion on list of symbolic values simply calls sv_subst on each element; and, when $sv = \mathtt{Sinput}(r, \_)$, we replace it by subst r. Note that we only substitute with source internal states, hence, forall r, subst r should never return None[11]. Warning: this notion of substitution must not be confused with the one of §6.3.3.1 (that aims to convert sequential invariants to their parallel representation); here, we aim to make the FPASV values take into account the registers already defined in the current symbolic state.

In practice, sv_subst is a `Fixpoint` *structurally (and mutually) recursing on sv (which is, alike other functions on FPASV, never executed in practice).*

Above, when si!r is not defined (in the sis_sreg lambda function) we look at the value of sis_input_init[12] to decide if the default value (i.e. $\mathtt{fSinput}(r)$) should be used or not.

The proof obligation ρ is easily proved by rewriting Lemma 6.4.1 on sigma-evaluations.

Given a concrete pair (rs, m), a context ctx, a symbolic value sv to substitute, and with the subst and substm functions that evaluate to rs and m (respectively), the substitution on sv is correct if the result of its evaluation corresponds to evaluating sv directly under a context updated with the concrete pair.

Lemma 6.5.1 (Substitution is correct [⋄])**.**

$$\begin{aligned}&\sigma_{sm}(\mathtt{ctx}, \mathtt{substm}) = \mathit{Some}\ m \implies\\ &\forall r, \mathtt{subst}\ r = \mathit{Some}\ sv \wedge \sigma_{sv}(\mathtt{ctx}, sv) = \mathit{Some}(rs \# r) \implies\\ &\sigma_{sv}(\mathtt{ctx}, [(\mathtt{substm}, \mathtt{subst})/sv]) = \sigma_{sv}(\{\,\mathtt{ctx}\ \mathit{with}\ \mathtt{crs}_0 = rs;\ \mathtt{cm}_0 = m\,\}, sv)\end{aligned}$$

Where $\{\,\mathtt{ctx}\ \mathit{with}\ \mathtt{crs}_0 = rs;\ \mathtt{cm}_0 = m\,\}$ *is equivalent to* $\mathtt{mk\_}\mathit{iblock}_{ctx}(\mathtt{ctx.(cG)}, \mathtt{ctx.(csp)}, rs, m)$.
Proof. By mutual induction on sv. □

AUXILIARY PROPERTIES ON SUBSTITUTION To prove that tr_sis has the expected behavior, we need a variant of Definition 6.3.5 with substitution.

Definition 6.5.2 ("Ok" predicate for substituted symbolic invariants [⋄])**.**

$$\begin{aligned}&\mathtt{si\_ok\_subst\ sis\ si\ ctx} \triangleq \forall sv, sv \in \mathtt{si.(fpa\_ok)} \implies\\ &\sigma(\mathtt{ctx}, [(\mathtt{sis.(sis\_smem)}, \mathtt{sis})/sv]) \neq \mathtt{None}\end{aligned}$$

The property states that if a symbolic value exists in the "ok" list of an invariant, then its substitution with the content of sis must evaluate without error.

The fact that invariants do not trap w.r.t. to a valid internal state used for the substitution is encoded by the implication between Definitions 6.4.12 and 6.5.2.

Definition 6.5.3 (Validity of a SI substituted with a valid internal state [⋄])**.**

$$\mathtt{tr\_sis\_ok\ ctx\ sis\ si} \triangleq \mathtt{sis\_ok\ ctx\ sis} \implies \mathtt{si\_ok\_subst\ sis\ si\ ctx}$$

Furthermore, matching the parallel invariant with a concrete regset requires *building* the latter by evaluating in ctx the result of sis on si. We decompose this goal by implementing a generic builder function for an arbitrary map to apply on a given symbolic regset:

Definition 6.5.4 (Regset builder from correctly evaluating mapped values)**.** Let ctx a block level context, "$\mathtt{map} : \mathit{sval} \to \mathit{sval}$" a symbolic value mapping, and "$\mathtt{si} : \mathit{reg} \mapsto \mathit{sval}\ \mathtt{option}$" a partial map from registers to symbolic values. The goal here is to build a new register state with the same default value (regsets are a pair of a default value and a map, as said in §3.3.2); and including, in addition to values already present in $(\mathtt{crs}_0\ \mathtt{ctx})$, any correctly evaluating value of si after applying map. More formally, we construct the register state by applying the following steps:

[11] In the actual definition, we return $\mathtt{fSinput}(r)$ in case the substitution on r is undefined. We enforce here a *sval* result (instead of an *sval* option) to simplify proofs.

[12] Here, I denote this test with the syntax of ternary conditions for readability.

1. Extracting the default value v_0 of the concrete regset in $\mathtt{ctx}$ so that the resulting regset will also have v_0 as default;
2. Mapping $\mathtt{si}$'s symbolic values to concrete values: $\forall \mathtt{r}, \mathtt{si!r} = \mathtt{Some}\ \mathtt{sv}$, if $\sigma_{\mathtt{sv}}(\mathtt{ctx}, \mathtt{map}\ \mathtt{sv}) = \mathtt{Some}\ v$, then we take v as the concrete value; otherwise, we take $(\mathtt{crs}_0\ \mathtt{ctx})\#\mathtt{r}$. The result of this map is named $\mathtt{rsmap}$;
3. Performing the union of $(\mathtt{crs}_0\ \mathtt{ctx})$ and $\mathtt{rsmap}$, with priority given to the latter: if a value is defined in both concrete regsets, we pick the version of $\mathtt{rsmap}$.

"PTree.t sval" is the type corresponding to "reg ↦ sval option", from the PTree library.

In Coq, this corresponds to the following definition:

```
Definition eval_map_sreg (ctx:iblock_common_context)
(map: sval → sval) (si : PTree.tree sval): regset :=
  (* (1) Extract the default value from (crs0 ctx) *)
  let default := fst (crs0 ctx) in
  (* (2) Mapping to create rsmap *)
  let rsmap := PTree.map (λ r sv ⇒
       match eval_sval ctx (map sv) with
       | Some v ⇒ v
       | None ⇒ (crs0 ctx)#r end) si
  (* (3) Union with priority given to rsmap *)
  in (default,
    PTree.combine (λ oa ob ⇒
      match ob with
      | Some x ⇒ Some x
      | None ⇒ oa
      end) (snd (crs0 ctx)) rsmap)
```

Thanks to this generic definition, building the concrete regset we need is now trivial. For each symbolic value $\mathtt{sv}$ defined in a given $\mathtt{si}$ for register $\mathtt{r}$, the concrete regset will contain either the result of the evaluation after *substituting* with (a given) $\mathtt{sis}$; or (if the evaluation is not defined) the value $\mathtt{ctx}.(\mathtt{crs}_0)\#\mathtt{r}$. We simply construct this regset by instantiating the above builder as:

Definition 6.5.5 (Regset builder from correctly evaluating values in invariant after substitution with an internal state [◇]).

$$\mathtt{eval\_subst\_si} : \mathit{iblock}_{\mathit{ctx}} \to \mathit{sistate} \to (\mathit{reg} \mapsto \mathit{sval}\ \mathtt{option}) \to \mathit{regset} \triangleq$$
$$\mathtt{eval\_map\_sreg}\ \mathtt{ctx}\ (\mathtt{sv\_subst}\ \mathtt{sis}.(\mathtt{sis\_smem})\ \mathtt{sis})$$

Notice how we partially apply the substitution: by fixing the memory and register set to use in $\mathtt{sv\_subst}$, the second argument (passed to $\mathtt{eval\_map\_sreg}$) becomes a $\mathtt{map}$ of type $\mathit{sval} \to \mathit{sval}$ as required by the generic builder (Definition 6.5.4).

Its correctness property is provided by lemma:

I specified this lemma to give an idea of the behavior of the function. Notice that it only covers the case where r is defined in si.

Lemma 6.5.2 (Builder of invariant substitution with internal state is correct [◇]).

```
Lemma eval_subst_si_correct ctx sis (si:fpasv) sv r:
  si!r = Some sv →
  (eval_subst_si ctx sis si)#r =
       match σsv(ctx, [sis.(sis_smem), sis/sv]) with
       | Some v ⇒ v
       | None ⇒ (crs0 ctx)#r
       end
```

Proof. Only by using the correctness lemmas of the PTree library for map and combine. □

CORRECTNESS OF THE SYMBOLIC STATE TRANSFER FOR A GIVEN FPASV Assuming an initial source (i.e. with a trivial frame) internal state $\mathtt{sis}$ matching a concrete pair $(\mathtt{rs}, \mathtt{m})$ under $\mathtt{ctx}$, and for a valid $\mathtt{si}$ in the sense of Definition 6.5.3, we have:

Lemma 6.5.3 (tr_sis is correct [◇]).

$$\begin{aligned}&\texttt{sem\_sistate ctx sis rs m} \implies\\ &\forall \texttt{r}, \texttt{build\_frame sis r} \implies \texttt{tr\_sis\_ok ctx sis si} \implies\\ &\texttt{sem\_sistate ctx (tr\_sis sis si false) (eval\_subst\_si ctx sis si) m} \wedge\\ &\texttt{match\_si}\,\{\texttt{ctx with crs}_0 = \texttt{rs};\ \texttt{cm}_0 = \texttt{m}\}\,\texttt{si (eval\_subst\_si ctx sis si)}\end{aligned}$$

Proof. First, we exploit Lemma 6.4.4 so we know that sis *is valid from the first hypothesis. Then, the* match_sreg *relation of* sem_sistate *is proved using map and combination properties of the* PTree *library (used in the implementation of FPASV's maps). The* match_si *conclusion is easily solved by using Lemma 6.5.1. Other subgoals are trivial.* □

6.5.2.1 *Input Invariants and Initial States*

The base for building the initial symbolic internal states Isis_s and Isis_t is to define the empty state ε.

Definition 6.5.6 (Empty symbolic internal state [◇]).

$$\begin{aligned}\varepsilon \triangleq \{&\texttt{sis\_pre} = \lambda(\_ : \mathit{iblock}_{ctx}).\,\texttt{True};\ \texttt{sis\_sreg} = \lambda(\texttt{r} : \mathit{reg}).\,\texttt{Some}(\texttt{fSinput}(\texttt{r}));\\ &\texttt{sis\_smem} = \texttt{fSinit};\ \texttt{sis\_pre\_preserved} = \rho\}\end{aligned}$$

Property sis_pre_preserved is trivially solved for an empty state.

Let "csix : invariants" be the invariants at the current pc, and let us note sis_H and sis_{HG} the states (sequentially) obtained by "tr_sis ε csix.(history) true" and "tr_sis sis_H csix.(glue) true", respectively. At first sight, following the informal explanation of §6.5.1, we would expect Isis_s to be exactly the application of ci_H on ε (i.e. that $\text{Isis}_s = \text{sis}_H$). In fact, only the symbolic regset of Isis_s is equal to the one of sis_H, but the precondition and the symbolic memory are assigned with the value of sis_{HG}[13]. More formally, we have:

Definition 6.5.7 (Source initial state [◇]).

$$\text{Isis}_s \triangleq \{\texttt{sis\_pre} = \text{sis}_{HG}.(\texttt{sis\_pre});\ \texttt{sis\_sreg} = \text{sis}_H;\ \texttt{sis\_smem} = \text{sis}_{HG}.(\texttt{sis\_smem})\}$$

This stems from the first technical detail mentioned in the end of §6.5.1: our checker must consider the trapping instructions of ci_G (and $\mathcal{G}$) as part of ci_H (and $\mathcal{H}$).

Each initialization is a Coq **Program** *asking for a proof of context preservation (ommited for simplicity).*

Intuitively, the target initial state Isis_t is thus built using sis_{HG}, and by filtering the symbolic regset to eliminate registers that are *not live* in the gluing invariant (this is needed because sis_{HG} was constructed with sis_input_init = true):

Definition 6.5.8 (Target initial state [◇]).

$$\begin{aligned}\text{Isis}_t \triangleq \{&\texttt{sis\_pre} = \text{sis}_{HG}.(\texttt{sis\_pre});\\ &\texttt{sis\_sreg} = \lambda \texttt{r}.\,\texttt{SOME sv} \leftarrow \texttt{csix.(glue) r IN sis}_{HG}\,\texttt{r};\\ &\texttt{sis\_smem} = \text{sis}_{HG}.(\texttt{sis\_smem})\}\end{aligned}$$

6.5.2.2 *Output Invariants and Final States*

The internal state application from the previous section serves us to build initial states; but, when the SE is finished (as pictured in Figure 6.3), the output invariants (i.e. either for GI or HI) must be applied on *all the final symbolic values* of a *sstate*. A fixpoint named "$\texttt{tr\_sstate} : \mathit{iblock}_{ctx} \to (\texttt{invs} : \mathit{pc} \mapsto \texttt{csasv}) \to \mathit{sstate} \to \mathit{sstate}$" [◇] is dedicated to the transfer of a binary decision tree with an output FPASV. Its "invs" parameter being a map (of invariants, built from the original gluemap) from a CFG node to either HIs or GIs, so the method works for both kinds of invariants. For each $\texttt{Sfinal}(\texttt{sis}, \texttt{sfv})$ leaf (the Sabort case is just the identity), it calls a function "$\texttt{si\_sfv} : \mathit{iblock}_{ctx} \to \mathit{sistate} \to (\texttt{invs} : \mathit{pc} \mapsto \texttt{csasv}) \to \mathit{sfval} \to \texttt{fpasv option}$" [◇], where the internal state and the final

Here again, these functions are not intended to be executed.

[13] The memory, however, is still equal to the one of ε, since it is untouched by tr_sis.

value are instantiated with `sis` and `sfv` (respectively). After having computed the tree of invariants corresponding to the leaf with `si_sfv`, `tr_sstate` delegates its application to `tr_sis` (Definition 6.5.1).

When the final instruction is a `Sgoto(pc)`, `si_sfv` simply returns the FPASV representation of the invariant at "`gm pc`" (i.e. thanks to Definition 6.3.10). For tail calls and returns (which have no successors), the function's result is the empty FPASV. Calls and built-ins cases are a bit more complex, and require checking that their result register—when applicable[14]—is *not mentioned* (i.e. using the "cloberrable" property of 6.3.15) in the invariant of their (unique) successor (otherwise `si_sfv` is undefined and returns `None`). In addition, if they feature a result register, then it must be removed from the "output" set of the invariant before converting it to a FPASV. Finally, and as indicated in Definition 6.4.10, the application for jump tables requires computing the *union* of the successors' invariants. For the list of successors `lpc`, we apply a map with `gm` to obtain a list of CSASVs, and we perform the unions—by accumulation, after converting the current compact invariant to its parallel form—until reaching the empty list (and thus performing the union with the empty FPASV at the end).

SIDE-NOTE ON THE UNION OF FPASVS [⋄] Formally defining this union in Coq was in fact a quite difficult part. To illustrate that, let us consider two FPASVs' trees $\texttt{fpa}_1$ and $\texttt{fpa}_2$ that we want to combine into a new tree $\texttt{fpa}_3$. For a register `r`, is the combination of $\texttt{fpa}_1\texttt{!r}$ and $\texttt{fpa}_2\texttt{!r}$ possible, and if so, *which value should we keep?* Naturally, if *neither* of them has a value for `r`, the combination is trivial ($\texttt{fpa}_3$ also not defines a value for `r`); similarly, if *only one* of them has a value for `r`, there is no choice to make, so we keep the value (i.e. $\texttt{fpa}_3$ is the result of a "most-defined" relation). On the other hand, when *both* trees have a mapping for `r` (e.g. $\texttt{fpa}_1\texttt{!r} = sv_1$ and $\texttt{fpa}_2\texttt{!r} = sv_2$), we have to ensure consistency w.r.t. the final internal state (and under the block level context). From a theoretical point of view, the following function must be true for sv_1 and sv_2:

```
(* A notion of symbolic equality over substitution for invariants' unification. *)
Definition symbolic_eq (ctx: iblock_ctx) sis (sv1 sv2: sval): bool :=
  match σ_sv(ctx, [(sis.(sis_smem), sis), sv1]), σ_sv(ctx, [(sis.(sis_smem), sis), sv2]) with
  | Some v1, Some v2 ⇒ Val.eq v1 v2
  | Some _, None | None, Some _ ⇒ false
  | None, None ⇒ true
  end
```

The principle is to compare the values obtained after evaluating the substitution of the symbolic memory and register set of `sis` in sv_1 and sv_2. The Boolean output of `symbolic_eq` indicates if the union is possible or not. If it is *true*, then we can keep either sv_1 or sv_2 (they are *equivalent* after substitution). In the above, when both evaluations succeed, the comparison is reduced to the equality over concrete values; when both fail, values are undefined, so we can consider them equal (it does not matter anyway). Otherwise, if one value has evaluated correctly after substitution but not the other, it means that the invariant is *inconsistent*. In such a situation, the union cannot be computed, and the combine operation will *fail* (and the whole validation as well, see §7.3.4.2).

The jump tables case took me about two weeks of full-time work, and is a typical example of why, in formal proof, we often say that "the devil lies in the details".

The `fpa_reg` field of FPASVs being implemented using the `PTree` library [8], we had to formalize a notion of error-prone combination of such trees. Indeed, even if the library already features a "combine" function[15], we cannot use it directly because its return type imposes that it must always succeed. To solve this issue, I extended their library with a "may combine" operation inside the option monad [⋄], so that it returns `None` in case of combination failure. In addition, my extension also provides an implementation in the *Impure* monad [⋄] to maintain compatibility with the verifier's implementation. Overall, these two versions of "may combine" and their proof of correctness represent about 450 new sloc (significant lines of Coq code) in the library. In the theory, the correctness proof of the FPASVs' unification represents about 200 sloc (excluding the implementation proof, which is quite the same except for the Impure monad bureaucracy).

[14] Calls always have a destination register, but it is not always the case for built-ins.

[15] Which is already quite complex, since the naive implementation consists of 49 cases and must be simplified using the "views" design-pattern [8, §5].

6.5.3 *Matching Simulations in a Predicate*

The symbolic execution success is encoded by a predicate yielding, from a block level context and a binary decision tree of type *sstate*, the final internal state and its associated final symbolic value:

Definition 6.5.9 ("Ok" predicate for SE [◇])**.**

$$\mathtt{symb\_exec\_ok}\ \mathtt{ctx}\ \mathtt{ss}\ \mathtt{sis}\ \mathtt{sfv} \triangleq \mathtt{get\_soutcome}\ \mathtt{ctx}\ \mathtt{ss} = \mathtt{Some}(\mathtt{sout}\ \mathtt{sis}\ \mathtt{sfv}) \wedge \mathtt{sis\_ok}\ \mathtt{ctx}\ \mathtt{sis}$$

The get_soutcome function from Definition 6.4.7 retrieves the pair $(\mathtt{sis}, \mathtt{sfv})$ from state ss. For the predicate to hold, sis must be valid in the sense of Definition 6.4.12.

To check the invariant application in the global simulation predicate, we specify a variant of predicate 6.5.9 using existential quantifiers:

Definition 6.5.10 ("Ok" predicate for symbolic state transfer [◇])**.**

$$\mathtt{trss\_ok}\ \mathtt{ctx}\ \mathtt{ss} \triangleq \exists \mathtt{sis}\ \mathtt{sfv}, \mathtt{symb\_exec\_ok}\ \mathtt{ctx}\ \mathtt{ss}\ \mathtt{sis}\ \mathtt{sfv}$$

In the hash-consed implementation, we replace the structural comparisons used for reasoning with efficient pointer equalities.

For final symbolic values, we define the simulation predicate syntactically. Actually, this trick is needed to avoid a *circularity* issue: the semantic simulation on final values involves concrete states, and here, we aim to provide a notion of simulation to be used with such states. With the abstraction level of our theory, this syntactical comparison is just an inductive predicate "$\mathtt{sfv\_simu} : \mathit{iblock}_{ctx} \to \mathit{sfval} \to \mathit{sfval} \to \mathtt{Prop}$" [◇]. The constructors of both *sfval* must of course be the same, and, for each pair of parameters which are not symbolic values (i.e. neither *list_sval* or *smem*), the comparison is *structural*. Symbolic values and their mutual variants are compared after evaluation with σ under ctx.

In contrast, symbolic internal states are compared semantically:

Definition 6.5.11 (Simulation predicate for internal states [◇])**.** Given two states $\mathtt{sis}_1$ and $\mathtt{sis}_2$ under ctx, we have:

$$\begin{aligned}\mathtt{sistate\_simu}\ \mathtt{ctx}\ \mathtt{sis}_1\ \mathtt{sis}_2 \triangleq \forall \mathtt{rs}_1\ \mathtt{m}, \mathtt{sem\_sistate}\ \mathtt{ctx}\ \mathtt{sis}_1\ \mathtt{rs}_1\ \mathtt{m} \implies \\ \exists \mathtt{rs}_2, \mathtt{sem\_sistate}\ \mathtt{ctx}\ \mathtt{sis}_2\ \mathtt{rs}_2\ \mathtt{m} \wedge \\ \mathtt{eqlive\_reg}\ (\mathtt{build\_frame}\ \mathtt{sis}_1)\ \mathtt{rs}_1\ \mathtt{rs}_2\end{aligned}$$

It may seem confusing to observe that the modulo liveness equality here is performed on $\mathtt{sis}_1$, but in fact, $\mathtt{sis}_1$ in this definition is intended to be instantiated (in Relation 6.5.12) with the internal state *after* having applied the GI (so that variables dead in the target will have already been filtered)[16].

Both executed blocks $\mathtt{ss}_G$ (the source SE plus the output invariants) and $\mathtt{ss}_t$ (the input invariants plus the target SE), must satisfy:

Definition 6.5.12 (Modulo liveness relation "$\succeq_t$" for both blocks [◇])**.** Verifying the SE result, the liveness information, and the final values' equality together:

$$\begin{aligned}\mathtt{match\_sexec\_live}\ \mathtt{ctx}\ \mathtt{ss}_G\ \mathtt{ss}_t \triangleq \forall \mathtt{sis}_G\ \mathtt{sfv}_G, \mathtt{symb\_exec\_ok}\ \mathtt{ctx}\ \mathtt{ss}_G\ \mathtt{sis}_G\ \mathtt{sfv}_G \implies \\ \exists \mathtt{sis}_t\ \mathtt{sfv}_t,\quad \mathtt{get\_soutcome}\ \mathtt{ctx}\ \mathtt{ss}_t = \mathtt{Some}(\mathtt{sout}\ \mathtt{sis}_t\ \mathtt{sfv}_t) \wedge \\ (\forall \mathtt{r}, \mathtt{build\_frame}\ \mathtt{sis}_G\ \mathtt{r} \implies \mathtt{build\_frame}\ \mathtt{sis}_t\ \mathtt{r}) \wedge \\ \mathtt{sistate\_simu}\ \mathtt{ctx}\ \mathtt{sis}_G\ \mathtt{sis}_t \wedge \\ (\forall \mathtt{rs}\ \mathtt{m}, \mathtt{sem\_sistate}\ \mathtt{ctx}\ \mathtt{sis}_G\ \mathtt{rs}\ \mathtt{m} \implies \mathtt{sfv\_simu}\ \mathtt{ctx}\ \mathtt{sfv}_G\ \mathtt{sfv}_t)\end{aligned}$$

This relation states that for all pairs $(\mathtt{sis}_G, \mathtt{sfv}_G)$ obtained from the source execution $\mathtt{ss}_G$, there exists a pair $(\mathtt{sis}_t, \mathtt{sfv}_t)$ that can be retrieved from the target's execution $\mathtt{ss}_t$. The first conclusion ensures, with get_soutcome, that $\mathtt{ss}_t$ indeed leads to a target pair, and is correct. Second, the implication between the source and the target frames models the fact that the live variables from $\mathtt{ss}_G$ (already filtered with the assignments of the GI) are also live in $\mathtt{ss}_t$ (a bi-implication is not needed here,

[16] The predicate is also used in Relation 6.5.13 for the HI, but the latter does not impact liveness checking.

meaning that the ss_t execution might set more live variables than those filtered in the GI). Third, both final internal states must simulate in the sense of Definition 6.5.11, and fourth, for all concrete pairs (rs, m) such that Relation 6.4.13 holds with sis_G, final symbolic values sfv_G and sfv_t must be syntactically equal.

With this relation on simulated blocks, we enforce the GIs' correctness (and thus the liveness information). In Figure 6.3, predicate 6.5.12 corresponds to verifying that $ss_G \succeq_t ss_t$ holds given both states. Nevertheless, we still need a predicate to model the $ss_H \preceq_s ss_s$ relation of our simulation scheme:

Definition 6.5.13 ("Redundancy" relation "$\preceq_s$" for the source's output history invariant [$\diamond$])**.** The output HI is correct if the symbolic state ss_H obtained after executing it is redundant with the symbolic internal state sis_s from ss_s (note that checking the final symbolic value is useless, because we know that it is not changed by applying the HI on ss_s). Hence, the relation focuses on internal states:

$$\begin{aligned}&\mathit{match\_sexec\_redundant}\ ctx\ ss_H\ sis_s \triangleq \forall sis_H\ sfv_H,\\ &\mathit{get\_soutcome}\ ctx\ ss_H = \mathtt{Some}(\mathit{sout}\ sis_H\ sfv_H) \implies \mathit{sistate\_simu}\ ctx\ sis_H\ sis_s\end{aligned}$$

Finally, the complete verification function details how states are built and how the previous relations aggregate together to capture the simulation scheme of Figure 6.3.

Definition 6.5.14 (Simulation modulo abstract symbolic invariants)**.** For a *simulation* context "$ctx : simu_{ctx}$", a gluemap gm, the source and target BTL blocks ib_s and ib_t (respectively), and the current program counter pc, the whole simulation consists of:

1. Building the initial source state $Isis_s$;
2. Executing $Isis_s$ with "$sexec\ ib_s\ Isis_s$" to obtain ss_s,
   and $Isis_t$ with "$sexec\ ib_t\ Isis_t$" to obtain ss_t;
3. Computing "$ss_H = tr\_sstate\ (Bctx_1\ ctx)\ (\lambda pc'.(gm\ pc').(history))\ ss_s$" and
   "$ss_G = tr\_sstate\ (Bctx_2\ ctx)\ (\lambda pc'.(gm\ pc').(glue))\ ss_s$";
4. Defining the predicate:

$$\begin{aligned}&\mathit{match\_sexec\_si}\ ctx\ ss_H\ ss_G\ ss_s\ ss_t : \mathrm{Prop} \triangleq \forall sis_s\ sfv_s\\ &\mathit{symb\_exec\_ok}\ (Bctx_1\ ctx)\ ss_s\ sis_s\ sfv_s \implies\\ &\quad \mathit{trss\_ok}\ (Bctx_1\ ctx)\ ss_H \wedge \mathit{trss\_ok}\ (Bctx_1\ ctx)\ ss_G \wedge\\ &\quad \mathit{match\_sexec\_redundant}\ (Bctx_1\ ctx)\ ss_H\ sis_s \wedge\\ &\quad \mathit{match\_sexec\_live}\ (Bctx_2\ ctx)\ ss_G\ ss_t\end{aligned}$$

In Coq, this corresponds to the following definition:

```
Definition match_sexec_si ctx gm ibs ibt pc: Prop := ∀sis_s sfv_s,
  let ss_s := sexec ibs Isis_s in
  symb_exec_ok (Bctx1 ctx) ss_s sis_s sfv_s →
  let ss_H := tr_sstate (Bctx1 ctx) (λ pc ⇒ history (gm pc)) ss_s in
  let ss_G := tr_sstate (Bctx2 ctx) (λ pc ⇒ glue (gm pc)) ss_s in
  trss_ok (Bctx1 ctx) ss_H ∧ trss_ok (Bctx1 ctx) ss_G ∧
  match_sexec_redundant (Bctx1 ctx) ss_H sis_s ∧
  match_sexec_live (Bctx2 ctx) ss_G (sexec ibt Isis_t)
```

Notice the context switch appearing at step 3.: the (output) HI is executed on the source block context, while we take the target context for applying the (output) GI (as explained in §6.1.2). The ok predicate validating the transfer of symbolic states concerns only the source side, so it always takes the source context. The same applies for the redundancy relation. On the other hand, the modulo liveness relation being specifically designed to check the target liveness, it has to be instantiated with the target context.

Recall that in the `match_function` Relation 6.1.1, we *universally quantify* the input `ctx` of the above predicate using Definition 6.4.4.

6.6 MORE DETAILS ON THE BLOCKSTEP SIMULATION PROOF

The proof presented here is not essential for a correct understanding of this thesis.

We proved in §6.4.4.3 that the symbolic execution theory was both correct and exact w.r.t. concrete BTL states. Hence, two of the four lemmas needed to complete the blockstep proof of Figure 6.2 remain to be proved: the correctness of invariants' application and the correctness of the modulo liveness relation. Below, §6.6.1 focuses on the former, and §6.6.2 on the latter; finally, the full simulation theorem is proved in §6.6.3.

6.6.1 *Correctness of Invariants' Transfer on Final Symbolic States*

We want to prove the subdiagram at label (2) of Figure 6.2. The lemma assumes a valid semantic relation between source concrete final state S_2 and final symbolic state ss_s (in the sense of Definition 6.4.14), and a successfully terminating SE (i.e. whose outcome from function 6.4.7 is a pair of an internal state and a final symbolic value "$\texttt{sout}\ sis_s\ sfv_s$"). Moreover, since the relation on the source and target BTL functions (i.e. the `match_function` of 6.1.1) is expected to hold[17], we know that both `tr_sstate` applications from step 3. in simulation Property 6.5.14 led to valid ss_H and ss_G symbolic states according to Property 6.5.10. The simulation predicate assumption also implies that ss_H is redundant w.r.t. sis_s (from Relation 6.5.13). Under these hypotheses, the invariants transfer is correct if there exists a concrete state S_G (using the same names as in Figure 6.2) resulting from the evaluation of ss_G, and if the matching Relation 6.1.3 holds for S_2 and S_G.

Theorem 6.6.1 (The transfer of invariants after symbolic execution is correct [◇]). *Let* f_1 *and* f_2 *the source and target BTL functions, respectively; and let* G *the global environment. Under a simulation context* $ctx : simu_{ctx}$*, we build the function contexts of* f_1 *and* f_2 *as* $fctx_1 : fct\_iblock_{ctx} = \{\texttt{cf} = f_1;\ \texttt{cc} = \texttt{Bctx}_1\ ctx\}$ *and* $fctx_2 : fct\_iblock_{ctx} = \{\texttt{cf} = f_2;\ \texttt{cc} = \texttt{Bctx}_2\ ctx\}$*, respectively. Then, for all source and target call stacks* $\vec{\Sigma}$ *and* $\vec{\Sigma}'$*, trace* e*, concrete state* S_2*, symbolic internal state* sis_s*, final symbolic value* sfv_s*, and symbolic states* ss_s*,* ss_H*, and* ss_G*, we have:*

$$
\begin{aligned}
&\texttt{sem\_sstate triv\_frame}\ fctx_1\ \vec{\Sigma}\ e\ S_2\ ss_s \implies\\
&\texttt{get\_soutcome}\ (\texttt{Bctx}_1\ ctx)\ ss_s = \texttt{Some}(\texttt{sout}\ sis_s\ sfv_s) \implies\\
&\texttt{tr\_sstate}\ (\texttt{Bctx}_1\ ctx)\ (\lambda pc'.\ (f_2.(\texttt{fn\_gm})\ pc').(\texttt{history}))\ ss_s = ss_H \implies\\
&\texttt{tr\_sstate}\ (\texttt{Bctx}_2\ ctx)\ (\lambda pc'.\ (f_2.(\texttt{fn\_gm})\ pc').(\texttt{glue}))\ ss_s = ss_G \implies\\
&\texttt{match\_sexec\_redundant}\ (\texttt{Bctx}_1\ ctx)\ ss_H\ sis_s \implies\\
&\texttt{trss\_ok}\ (\texttt{Bctx}_1\ ctx)\ ss_H \implies \texttt{trss\_ok}\ (\texttt{Bctx}_1\ ctx)\ ss_G \implies\\
&\texttt{list\_forall2}\ (\texttt{match\_stackframes}\ G)\ \vec{\Sigma}\ \vec{\Sigma}' \implies\\
&\texttt{match\_function}\ f_1\ f_2 \implies\\
&\exists S_G, \texttt{sem\_sstate sfv\_frame}\ fctx_2\ \vec{\Sigma}'\ e\ S_G\ ss_G \wedge\\
&\qquad \texttt{match\_states}\ G\ S_2\ S_G
\end{aligned}
$$

Proof. By induction on the first hypothesis: the `Sabort` *case is thus directly eliminated. The inductive case on* `Scond` *is easily demonstrated by exploiting the induction hypothesis, which becomes applicable by showing that solving the goal for an* `Scond` *state is equivalent to solving it for the successor's state that results from the evaluation of the condition (Definition 6.4.2).*

When the state is of type `Sfinal`*, each kind of final symbolic value must be considered. All of those share the same initial step: the idea is to exploit* `tr_sis_correct` *with the appropriate instance of FPASV (variable "*`si`*" in Lemma 6.5.3). The "*`si`*" instance must correspond to the FPASV built by* `si_sfv` *(the function described in §6.5.2.2) for the final value of the current goal.*

Hence, mimicking the description of `si_sfv`*, if the final value has no successor, "*`si`*" must be instantiated with the empty FPASV* `si_empty` *(e.g. for tail calls and returns). Otherwise, the theorem must be instantiated*

[17] In practice, the matching relation between BTL functions is the main check performed by our validator. One may notice that since it already contains the complete simulation property in `match_sexec_ok`, some other hypotheses are thus redondant (because they follow from this main matching relation). Albeit relevant, we usually prefer to ignore this remark and keep redundancies in the proof's signature when they avoid to repeat some bureaucratic proof steps and facilitate the overall understanding.

*with the symbolic invariant of the successor, from which the result register of the final value has been removed beforehand (if applicable, i.e. for built-ins featuring a result register or calls). For gotos instructions and built-ins without result register, the FPASV of the successor is taken as is. Last, the jump table case simply instantiates "*si*" with the union of the successors' invariants.*

The rest of the proof does not contain any subtleties, and is only about either making use of assumptions or rewriting with intermediate preservation properties on context switching. For example, it is necessary (for all `Sfinal` *cases) to show that if Predicate 6.4.13 holds for the source context, then it also holds for the target one. The latter property trivially follows from the constraint* sis_pre_preserved *on internal symbolic states (Definition 6.4.6) and from Lemma 6.4.1 (preservation of sigma-evaluations).* □

6.6.2 *Correctness of the Modulo Liveness Relation w.r.t. Concrete States*

Implications between implemented checks and theoretical relations (as the modulo liveness one, or the syntactical equalities of Definition 6.1.1) are detailed in §7.5.3.

Given the assumption that the match_sexec_live predicate from Definition 6.5.12 holds for the source symbolic state after applying the output GI (ss_G) and the target symbolic state after executing ib_t (ss_t) in Figure 6.2, the goal is to prove the subdiagram at label (3). The second important hypothesis is exactly the first conclusion of the previous theorem, stating that concrete state S_G is the result obtained by semantically evaluating symbolic state ss_G (i.e. the sem_sstate relation). And of course, we still have the matching relation over BTL functions (of Definition 6.1.1), as in the previous theorem[17]. The two first hypotheses correspond to the top horizontal relation ($ss_G \succeq_t ss_t$) and the left vertical concretization ($ss_G \rightarrow^e S_G$) of subdiagram (3) of Figure 6.2, respectively. Therefore, in order to complete the subdiagram, we must show that the modulo liveness relation on symbolic states is correct if there exists a concrete state S'_G resulting from the evaluation of the target symbolic state (i.e. the concretization step $ss_t \rightarrow S'_G$) and if both concrete states S_G and S'_G are equivalent modulo the target program liveness (i.e. relation $\equiv_t$).

Theorem 6.6.2 (The modulo liveness relation is correct [$\diamond$]). *Again, we name f_1 and f_2 the source and target BTL functions, respectively. This time, we only need the function level context of the target function $fctx_2 : fct\_iblock_{ctx} = \{cf = f_2;\ cc = Bctx_2\ ctx\}$, built from the simulation context ctx. Indeed, subdiagram (3) being about concretizing symbolic states after the application of the gluing invariant, the context already switched. For the same reason, the theorem only considers the target call stack $\vec{\Sigma}'$ here. The trace emitted by the program is still noted e. Given the concrete state S_G, and symbolic states ss_G and ss_t, the theorem is:*

$$\begin{aligned}
&\texttt{match_sexec_live}\ (Bctx_2\ ctx)\ ss_G\ ss_t \implies \\
&\texttt{sem_sstate}\ sfv_frame\ fctx_2\ \vec{\Sigma}'\ e\ S_G\ ss_G \implies \\
&\texttt{match_function}\ f_1\ f_2 \implies \\
&\exists S'_G, \texttt{sem_sstate}\ sfv_frame\ fctx_2\ \vec{\Sigma}'\ e\ S'_G\ ss_t \wedge S_G \equiv_t S'_G
\end{aligned}$$

Proof. The initial step is to deduce a valid symbolic execution from the sem_sstate *assumption on ss_G, thanks to Lemma 6.4.6, leading to an outcome "*sout sis_G sfv_G*" and characterized by* sem_sistate*. After that, we exploit Lemma 6.4.4 on the resulting* sem_sistate *hypothesis. This gives us property "*sis_ok ($Bctx_2$ ctx) sis_G*", which is needed, along with the* get_soutcome *from the initial step, to exploit the* match_sexec_live *assumption.*

*All the conclusions of Definition 6.5.12 are now available: the result of the SE of ss_t as a pair "*sout sis_t sfv_t*", the implication between frames of sis_G and sis_t (respectively), the simulation predicate of those two internal states, and the implication, for any pair (rs, m), between the "ok" predicate for sis_G and the "*sfv_simu ($Bctx_2$ ctx) sfv_G sfv_t*" simulation property on final symbolic values. The latter implication is trivial to resolve since we already know from the first step that there is a valid concretization of sis_G (the previous* sem_sistate*).*

Next, we exploit simulation predicate sistate_simu ($Bctx_2$ ctx) sis_G sis_t *(Definition 6.5.11) that we got after decomposing 6.5.12. Its prerequisite is exactly, here also, the valid concretization from the first step. This step proves the existence of a target regset resulting from the concrete evaluation of sis_t, and its modulo liveness equivalence with the corresponding one for sis_G.*

In other words, the evaluation $ss_t \rightarrow S'_G$ is demonstrated until the second last instruction, but we still have to prove it for the final instruction. The rest of the proof is thus by case analysis on final symbolic values: we split the main goal by inverting the sfv_simu *hypothesis.*

To construct the whole $\mathtt{ss}_t \to S'_G$ *transition, which is encoded by the* sem_sstate *predicate, each final case must exploit Lemma 6.4.7. Doing so allows us to define the concrete state* S'_G *to instantiate the existential quantifier of the conclusion. This solves the left part of the goal for all final cases. The equality modulo liveness on concrete states—i.e. the right part of the goal—is easily proved by eventually rewriting (according to the considered final case) the monotonicity (cf. §6.4.3.1) and update (Lemma 6.4.2) properties of the frame-scoped regset equality, and by assumption from the previously decomposed* sistate_simu *predicate.* □

6.6.3 *Correspondence With the BTL Operational Semantics*

Let us explain how we put it all together to obtain the complete proof of diagram 6.2. In the blockstep transition predicate of the BTL IR, defined in 5.3.7, the diagram actually corresponds to the `exec_iblock` transition. Hence, I start by proving the latter as a separate theorem, before constructing the entire blockstep proof covering the three other types of transitions (for internal/external function calls, and return states).

6.6.3.1 *Normal Blockstep Transition Simulation*

We assume a valid blockstep execution of a source block $\mathtt{ib}_1$: *iblock_info* (in the sense of Definition 5.3.6), located at pc in the source BTL function $\mathtt{f}_1$, and leading to concrete state S_2. As before, we know from the validator check that `match_function` (Definition 6.1.1) holds between $\mathtt{f}_1$ and the target function $\mathtt{f}_2$. In addition, since the standard forward simulation scheme assumes that initial (concrete) states match (i.e. that they are related by Definition 6.1.3), we add as hypothesis the concrete invariant semantics (the match_invs Relation 6.3.11). Still from the concrete state matching relation, call stacks are expected to match (two-by-two with `list_forall2` and Definition 6.1.2). The goal is therefore to show the existence of a target block, located at the same pc in $\mathtt{f}_2$, and whose blockstep execution results in a concrete state S'_2, related to S_2 by `match_states`.

Theorem 6.6.3 (Iblock step simulation (Figure 6.2) [⋄])**.** *With* G_1 *and* G_2 *the source and target global environments (respectively),* $\vec{\Sigma}$ *and* $\vec{\Sigma}'$ *the call stacks,* $\mathtt{f}_1$ *and* $\mathtt{f}_2$ *the source and target functions (respectively),* sp *the stack pointer,* $\mathtt{rs}_1$ *and* $\mathtt{rs}_2$ *the source and target initial regsets (respectively),* m *the memory,* e *the trace, and* $\mathtt{ib}_1$ *the source block located at* pc *and leading to state* S_2*, we write:*

$$
\begin{aligned}
&\mathtt{iblock\_step}\ G_1\ \vec{\Sigma}\ \mathtt{f}_1\ \mathtt{sp}\ \mathtt{rs}_1\ \mathtt{m}\ \mathtt{ib}_1.(\mathtt{entry})\ \mathtt{e}\ S_2 \implies\\
&(\mathtt{fn\_code}\ \mathtt{f}_1)!\mathtt{pc} = \mathit{Some}\ \mathtt{ib}_1 \implies\\
&\mathtt{match\_function}\ \mathtt{f}_1\ \mathtt{f}_2 \implies\\
&\mathtt{match\_invs}\ (\mathtt{mk\_}\mathit{iblock}_{\mathtt{ctx}}(G_1, \mathtt{sp}, \mathtt{rs}_1, \mathtt{m}))\ (\mathtt{f2}.(\mathtt{fn\_gm})\ \mathtt{pc})\ \mathtt{rs}_2 \implies\\
&\mathtt{list\_forall2}\ (\mathtt{match\_stackframes}\ G_1)\ \vec{\Sigma}\ \vec{\Sigma}' \implies\\
&\exists \mathtt{ib}_2\ S'_2, (\mathtt{fn\_code}\ \mathtt{f}_2)!\mathtt{pc} = \mathit{Some}\ \mathtt{ib}_2 \wedge\\
&\qquad \mathtt{iblock\_step}\ G_2\ \vec{\Sigma}'\ \mathtt{f}_2\ \mathtt{sp}\ \mathtt{rs}_2\ \mathtt{m}\ \mathtt{ib}_2.(\mathtt{entry})\ \mathtt{e}\ S'_2 \wedge\\
&\qquad \mathtt{match\_states}\ G_1\ S_2\ S'_2
\end{aligned}
$$

Proof. To rely on the simulation test performed by the validator, we exploit the `match_sexec_ok` *property of* match_function *(see Definition 6.1.1). We then demonstrate subdiagram (1) of Figure 6.2 thanks to Theorem 6.4.11. This requires proving two intermediate goals: first, that the transition* $\mathtt{Isis}_s \to S_1$ *is valid; and second, that the initial frame of* $\mathtt{Isis}_s$ *is well-defined. The former relation is deduced from the* match_invs *hypothesis, and the frame needed for the second goal is trivial from the definition of* $\mathtt{Isis}_s$*.*

At that point, we apply Theorems 6.6.1 and 6.6.2 to complete subdiagrams (2) and (3), respectively. Finally, the exactness of symbolic execution (Theorem 6.4.12) gives us the proof for subdiagram (4). Its application requires, symmetrically to the correctness subdiagram, to show that transition $\mathtt{Isis}_t \to S'_1$ *is valid. The only (little) subtlety here concerns the transitivity of the modulo liveness regset equality (cf. §6.4.3.1): indeed, such a property is needed at the end of the proof to state that* $S_G \equiv_t S'_G \equiv_t S'_2$*.* □

6.6.3.2 *Overall Forward Step Simulation*

The forward simulation of the whole symbolic execution blockstep pass is following a classical lock-step scheme (Figure 6.1):

Theorem 6.6.4 (BTL Block simulation proof [◇]). *Sticking to the notations of Figure 6.1, let* S_1 *and* S_2 *the source initial and final states, and* S_1' *and* S_2' *the target initial and final states. The trace is still noted* e*, and the source and target global environments* G_1 *and* G_2*, respectively.*

$$\texttt{step}\ G_1\ S_1\ e\ S_2 \implies \texttt{match\_states}\ G_1\ S_1\ S_1' \implies$$
$$\exists S_2', \texttt{step}\ G_2\ S_1'\ e\ S_2' \wedge \texttt{match\_states}\ G_1\ e\ S_2\ S_2'$$

Proof. The step hypothesis is destructed to obtain the four kinds of transitions. The `exec_iblock` *case is straightforward by applying Theorem 6.6.3, and the* `exec_return` *transition is solved using constructors from both conclusions.*

The external call case (i.e. `exec_function_external`*) requires showing that call symbols are preserved between* G_1 *and* G_2*, and this is trivial considering that our validation mechanism do not change them at all.*

When the transition is an internal call (i.e. `exec_function_internal`*), the preservation of the stack size gives us the step conclusion directly, thanks to the* `preserv_fnstacksize` *field of Relation 6.1.1. The state matching is also resolved using properties of* `match_function`*: the two properties on invariants at the entry point of the function (i.e.* `trivial_glueinv_entrypoint` *and* `trivial_histinv_entrypoint`*), and the preservation checks for parameters and the entry point (i.e.* `preserv_fnparams` *and* `preserv_entrypoint`*, respectively).*

Property `preserv_fnsig` *is also needed in the proof to show that given* S_1*, the target initial state* S_1' *exists and matches the former.* □

Thanks to the above theorem, the symbolic simulation validator is fully integrated and proved as a normal CHAMOIS-COMPCERT pass from BTL to BTL. By sequentially combining it with the *translation passes* between RTL and BTL (Chapter 8), we get a classical RTL to RTL pass that can be *placed* wherever we want in the existing RTL sequence of passes.

7

SYMBOLIC SIMULATION REFINEMENT AND IMPLEMENTATION†

This chapter only sketches proofs of the main SE theorems and lemmas; see the complete Coq proof online.

This chapter describes how the symbolic execution theory (of Chapter 6) is refined and efficiently implemented using normalized rewriting, hash-consing, and concrete data structures. The refinement of a high-level specification by a lower-level one is inspired by the RTLpath approach (cf. §4.4.4). As documented in §9.1, compared to RTLpath and other prior experiments, this refinement is much more intricate. The main novelties are the generalized block structure, the invariants, and the rewriting methods employed, such as the affine forms essential for strength-reduction validation. Figure 7.1 and Section 7.1 below give an outline of this chapter.

7.1 HIGH-LEVEL VIEW OF THE ARCHITECTURE

The primary goal of our refinement technique is to circumscribe the reasoning on impure computations as much as possible. These computations mainly concern the hash-consing mechanism, which is managed through the IMPURE monad (presented in §2.4). The implementation—i.e. the code that will be actually extracted and executed—is split into four modules:

PRELUDE: the base module defining refined states and their (refinement) relation with theoretical ones, generic operations over refined states, and common hash-consing tools to manipulate and compare data;

SYMBOLIC EXECUTION: simulation functions for BTL blocks, and in particular refined versions of definitions from §6.4.4;

SYMBOLIC INVARIANTS: functions to transfer invariants on refined symbolic states, and related properties;

TEST: main refined simulation test, that implies Definition 6.5.14 of the theory.

The overall architecture is pictured in Figure 7.1, where the plain arrow denotes the refinement step (e.g. A → B means A *refines* B) and dashed arrows represent dependencies between Coq modules. Notice that the execution module is independent of the symbolic invariants one. Normalization rules applied during SE are written and proven in an architecture dependent module, that is specified in §7.2.5 and implemented for two specific optimizations in §7.6.

REMARK ON THE REFINEMENT STRATEGY In order to reason on concrete types and to make the link with the theory without being hindered by the monadic encapsulation of IMPURE while writing proofs, some parts of our implementation can be refined in *two steps*. The idea is to incrementally refine certain functions by separating the concrete data structure refinement from the hash-consing encapsulation. It results in an intermediate refinement layer which is easier to prove, and whose

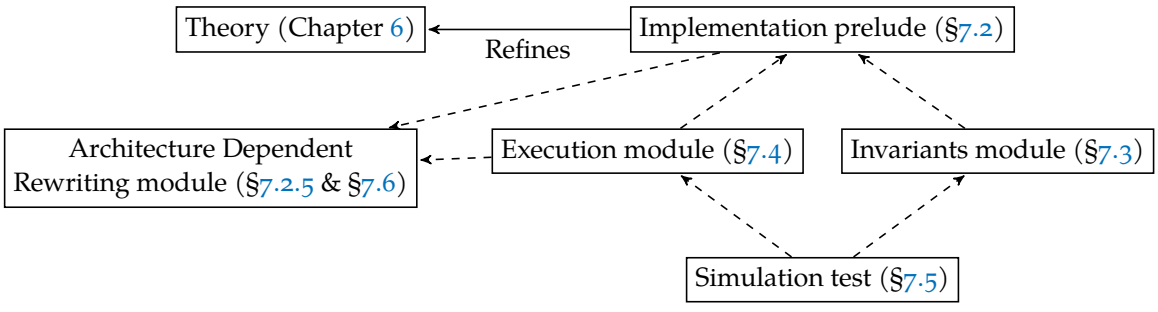


Figure 7.1: Architecture of the Refinement Layers.

proof scheme is often applicable to the fully refined layer defined within the monad. However, this strategy being time-consuming to maintain when Coq proofs evolve, we only exploited it for a few proofs. A typical example is given with functions accessing a register in a symbolic state: function `sis_sreg` of the theory is refined with Definitions 7.2.3 and 7.2.9.

7.2 CONCRETE DATA STRUCTURES AND OPERATIONS

The types for symbolic values, whether final or not, are the same between the theory and the refinement. Only internal and final symbolic states are subject to changes. Refined states, discussed in §7.2.1, are related to their theoretical models in §7.2.3. Section 7.2.2 outlines an *intermediate refinement* model for accessing a refined symbolic regset. The hash-consing mechanism and a function to hash-cons "fake" symbolic values are presented in §7.2.4, supporting the specification of the rewriting engine functions in §7.2.5. An efficient implementation of this getter model, utilizing hash-consing, is found in §7.2.6. Finally, §7.2.7 breaks down the process of setting and rewriting a value within a refined, hash-consed symbolic regset.

7.2.1 *Refined Symbolic States*

We propose a refined version of the theoretical internal states from Definition 6.4.6, where the precondition is replaced by a *list of potentially trapping symbolic values*, and with concrete data structures in place of the $reg \to sval$ abstract register state.

Definition 7.2.1 (Refined symbolic internal states [◇])**.** The memory representation stays the same: `ris_smem` contains the current symbolic memory evaluation. Registers are now stored in a concrete map from positive integers to symbolic values (with the `PTree` library) in `ris_sreg`. This map being finite, we keep a `ris_input_init` Boolean to indicate if registers should have a default value or not. The states' precondition from our theory is encoded as a list `ok_rsval` of "ok" values.

$$
\begin{array}{lll}
ristate \triangleq \{ & \texttt{ris\_smem} : smem; & \text{Same type as in theory.} \\
& \texttt{ris\_input\_init} : bool; & \text{When true, the state is in "source" mode.} \\
& \texttt{si\_mode} : bool; & \text{When true, any trapping value assigned to} \\
& & \texttt{ris\_sreg} \text{ must be included in } \texttt{ok\_rsval}. \\
& \texttt{ok\_rsval} : \overrightarrow{sval}; & \text{List of seen trapping values.} \\
& \texttt{ris\_sreg} : reg \mapsto sval\ \texttt{option}\ \} & \text{Symbolic register state.}
\end{array}
$$

Final symbolic states (i.e. binary decision trees) are refined simply by replacing theoretical internal states by concrete ones:

Definition 7.2.2 (Refined symbolic states [◇])**.**

$$rstate ::= \texttt{Rfinal}(ristate, sfval) \mid \texttt{Rcond}(cond, list\_sval, rstate_{\text{so}}, rstate_{\text{not}}) \mid \texttt{Rabort}$$

Exactly as in the theory, we define a "`get_routcome` : $iblock_{ctx} \to rstate \to$ `routcome option`" [◇] where `routcome` has the same definition as `soutcome` but with a refined internal state instead of a theoretical one.

7.2.2 *Model of Symbolic Register Access and Default Values*

To imitate the abstract `sis_sreg` field of theoretical symbolic states, we define a function, called `ris_sreg_get`, which returns the symbolic value possibly associated with a register in a certain *ristate*. This function seeks the `ris_sreg` dictionary. If the accessed register is defined, it returns its value; otherwise, the behavior depends on the `ris_input_init` Boolean. When the Boolean is true in a refined symbolic internal state, accessing an undefined symbolic register `r` returns the default "input" symbolic value. Otherwise, accessing `r` results in a validation failure. In other words, the getter function is total in *source* mode, and partial in *target* mode. The definition uses an assert notation `Notation "'ASSERT' A 'IN' B" := (if A then B else None)` for the option monad.

Definition 7.2.3 (Intermediate refinement of the symbolic register getter)**.**

```
Definition ris_sreg_get (ris: ristate) r: option sval :=
  match ris.(ris_sreg)!r with
  | None ⇒ ASSERT ris_input_init ris IN Some (fSinput r)
  | Some sv ⇒ Some sv
  end
```

This definition is not the getter of the actual implementation (defined in §7.2.6): it corresponds to an intermediate refinement layer, devoid of hash-consing. Hence, when `ris_input_init` is true, the above returns the "fake" symbolic value using `fSinput`. From here, register accesses through this intermediate layer are *coerced* as "`ris r`". It is part of our semantics of *ristate*, as defined below by the `ris_refines` relation.

7.2.3 *Validity and Refinement Relation*

In order to encode the relation between theoretical and refined symbolic internal states, we adapt the validity property of Definition §6.4.12 as follows:

Definition 7.2.4 (Refined version of the "`sis_ok`" predicate [⋄])**.** Given a refined symbolic internal state `ris`, and under the block level context `ctx`, we impose that all values defined in `ok_rsval` evaluate without error.

$$\begin{aligned}\texttt{ris\_ok ctx ris} \triangleq\ & \sigma_{\texttt{sm}}(\texttt{ctx}, \texttt{ris.(ris\_smem)}) \neq \texttt{None} \wedge \\ & \forall \texttt{sv}, \texttt{sv} \in \texttt{ris.(ok\_rsval)} \implies \sigma_{\texttt{sv}}(\texttt{ctx}, \texttt{sv}) \neq \texttt{None}\end{aligned}$$

The theory's version of validity is *stronger*, as it imposes that *all* defined registers evaluate correctly. For the symbolic memory, we keep the same constraint.

We now build the refinement relation as the conjunction of four properties:

Definition 7.2.5 (Refinement relation between symbolic internal states [⋄])**.** Still under a block context `ctx`, and considering `ris` and `sis` as the two internal states:

$$\begin{aligned}\texttt{ris\_refines ctx ris sis} \triangleq\ & \texttt{sis\_ok ctx sis} \iff \texttt{ris\_ok ctx ris} \wedge \\ & \texttt{ris\_ok ctx ris} \implies \sigma_{\texttt{sm}}(\texttt{ctx}, \texttt{ris.(ris\_smem)}) = \\ & \qquad\qquad \sigma_{\texttt{sm}}(\texttt{ctx}, \texttt{sis.(sis\_smem)}) \wedge \\ & \texttt{ris\_ok ctx ris} \implies \forall \texttt{r}, \texttt{ris r} = \texttt{None} \iff \texttt{sis r} = \texttt{None} \wedge \\ & \texttt{ris\_ok ctx ris} \implies \forall \texttt{r}, |\sigma_{\texttt{sv}}|(\texttt{ctx}, \texttt{ris r}) = |\sigma_{\texttt{sv}}|(\texttt{ctx}, \texttt{sis r})\end{aligned}$$

For `ris` to be a valid refinement of `sis`, the equivalence of their validity predicates is necessary. Furthermore, assuming that `ris` is a *valid* refined state:

- both symbolic memories from `ris` and `sis` must evaluate to the same concrete value;
- any dead register in `ris` must also be dead in `sis`, and vice-versa;
- conversely, any live register in `ris` must also be live in `sis`, and evaluate to the same concrete value, and vice-versa.

In the last clause, we encapsulate the $\sigma_{\texttt{sv}}$ function into a new one $|\sigma_{\texttt{sv}}|$ [⋄], of type "$\mathit{iblock}_{\texttt{ctx}} \to \mathit{sval}$ `option` $\to$ `val option`". It returns $\sigma_{\texttt{sv}}(\texttt{ctx}, \texttt{sv})$ when the option argument is "`Some sv`", and `None` otherwise.

Note that we cannot decompose this refinement relation into an abstraction function *ristate* $\to$ *sistate* and an equivalence relation between the result and the original state. Indeed, *any symbolic register of a valid abstract state* evaluates correctly. This is generally not true for refined states (i.e. their validity condition is weaker) unless, precisely, the refinement relation holds. An alternative design that would enable the definition of `ris_refines` as the composition of an abstraction function and an equivalence on *sistate* would involve adding a wellformedness field to both the abstract and the refined types.

Extending Relation 7.2.5 for *rstate* is straightforward with an inductive property:

Definition 7.2.6 (Refinement relation between symbolic states)**.** We parametrize the property with a Boolean, so we can instantiate it for either the source or target states. In the inductive case, we reuse the evaluation of conditions from Definition 6.4.2.

```
Inductive rst_refines (input_init: bool) ctx: rstate → sstate → Prop :=
  | Reffinal ris sis rfv sfv
      (RIS: ris_refines ctx ris sis)
      (RIS_init: ris_input_init ris = input_init)
      (RFV: ris_ok ctx ris → rfv_refines ctx rfv sfv)
      : rst_refines input_init ctx (Rfinal ris rfv) (Sfinal sis sfv)
  | Refcond rcond cond rargs args rifso rifnot ifso ifnot
      (RCOND: eval_scondition ctx rcond rargs = eval_scondition ctx cond args)
      (REFso: eval_scondition ctx rcond rargs = Some true →
        rst_refines input_init ctx rifso ifso)
      (REFnot: eval_scondition ctx rcond rargs = Some false →
        rst_refines input_init ctx rifnot ifnot)
      : rst_refines input_init ctx (Rcond rcond rargs rifso rifnot)
                                   (Scond cond args ifso ifnot)
  | Refabort
      :rst_refines input_init ctx Rabort Sabort
```

Here, "rfv_refines : $iblock_{ctx} \to sfval \to sfval$" [⋄] represents the refinement relation for final values. It features one constructor per final symbolic value, and holds if (i) both final values are of the same type; (ii) their symbolic arguments evaluate to the same concrete values; (iii) other arguments (e.g. call identifiers) are structurally equal.

The validity predicate, along with the refinement relations for internal states and final values, are preserved during context switching[1].

7.2.4 *The Hash-Consing Mechanism*

Knowing that storing and comparing structurally symbolic values is *very costly* due to duplications, our implementation leverages a hash-consing factory, enabling us to share identical subtrees and providing a constant-time pointer equality check. To ensure the safety of this latter check, as justified in §2.4.3, we embed it within the IMPURE monad, which prevents unsafe reasoning about the purity of functions. I have already briefly introduced the defensively checked hash-consing in §2.4.2, as well as the generic method to compare terms in §2.4.4. When applied to symbolic values, we only use pointer equality to compare *subterms* (i.e. of type *sval*, *list_sval*, and *smem*). Other data types (e.g. Coq constructors, registers' IDs, etc.) are compared structurally.

NOTATIONS FOR THE IMPURE MONAD In this chapter, I keep the "⇝" and "RET" notations for the may-return and unit operators of §2.4.4. Any function within IMPURE (i.e. with a may-return type ??A) can either succeed by yielding a result (with "RET"), or fail and abort the whole compilation process. In the following sections, we denote "FAILWITH msg" the latter, to abort with error message msg. From the caller's point of view, a "DO x ⇜ f;; C" notation attempts to extract the IMPURE result of f into x before executing C, and aborts execution if the function has failed. Technically, the failing case is extracted to a `raise` instruction in OCaml.
For a full description of IMPURE, please refer to Boulmé [23, Chapter 2]'s habilitation thesis.

7.2.4.1 *Equality Checkers*

To parametrize the hash-consing factory and implement the simulation test, we use two equality checks from IMPURE: the physical (pointer) equality (cf. §2.4.4), and a structural check with a stronger correctness property.

[1] These simple demonstrations are available at [⋄] (validity); [⋄] (refinement relation for internal states); [⋄] (refinement relation for whole states); [⋄] (refinement relation for final values).

STRUCTURAL EQUALITY Unlike the physical equality's partial correctness axiom, which only gives information when the result is *true* (but does not allow concluding anything when the result is false), the structural equality covers both cases. It is defined as an axiom extracted to the standard OCaml equality:

```
Axiom struct_eq: ∀ {A}, A → A → ?? bool
Extract Inlined Constant struct_eq ⇒ "(=)"
Axiom struct_eq_correct: ∀ A (x y:A), struct_eq x y ⇝ b → if b then x=y else x<>y
```

Note the difference with `phys_eq`: when the structural equality returns false, we *know* that both values are different w.r.t. the Coq (i.e. Leibniz) equality[2]. From here, physical equality is denoted with operator "`==`".

PURE POINTER EQUALITY In the upcoming discussions, we will identify a need for a more streamlined and efficient approach to working with the IMPURE monad. Specifically, in §7.6.2, we propose a normalization procedure for affine forms that necessitates a *total order* over symbolic values. Additionally, in §7.2.4.3, we will find that it is advantageous for our rewriting engine (which goes beyond the affine normalization) to avoid working inside the IMPURE monad. Addressing these needs, we introduce a *significant improvement*: a pure pointer equality as seen from a Coq perspective.

Type "`pstring`" here abstracts strings from the Coq and OCaml standards together; it is defined by IMPURE.

To implement this idea, we start by defining a notion of impure assertion:

```
Definition assert_k (k: ?? bool) (msg: pstring): ?? unit
  ≔ DO b ↜ k;; if b then RET tt else FAILWITH msg
```

This mechanism extracts the impure computation as an assertion that either returns ??tt ("tt" being the "unit" type in Coq) or fails. It satisfies:

```
Lemma assert_k_correct (k: ?? bool) (msg: pstring):
  assert_k k msg ⇝ _ → k ⇝ true
  (* Trivial by definition of assert_k. *)
Qed
```

Then, we define the notion of a safe exit coercion from the monad as:

```
Definition safe_coerce (k: ?? bool) (msg: pstring): bool ≔ has_returned (assert_k k msg)
```

By encapsulating the assertion into the IMPURE's termination operator (for Boolean functions), "`safe_coerce`" exits the monad and returns a Boolean result. At runtime, the result is never false (recall the definition of "`has_returned`" in §2.4.4): the only alternative is non-termination (an abort inside the impure computation). Hence, when "`safe_coerce`" yields true, we *prove* that the impure computation terminated successfully:

```
Lemma safe_coerce_correct (k: ?? bool) (msg: pstring):
  safe_coerce k msg = true → k ⇝ true
  (* By unfolding safe_coerce and exploiting the has_returned correctness axiom
  The remaining goal corresponds to lemma assert_k_correct above. *)
Qed
```

Deducing the pure pointer equality we aim at using this safe coercion is quite easy:

Definition 7.2.7 (A fast, pure pointer equality test)**.**

```
Definition very_fast_eq_msg: pstring ≔ "very_fast_eq failed"

Definition very_fast_eq {A} (x:A) (y:A): bool ≔ safe_coerce (phys_eq x y) very_fast_eq_msg
```

Here again, the above definition is pure since it can never return false. If `x` is different from `y`, function `very_fast_eq` will abort the compiler.

[2]Please note, declaring this polymorphic `struct_eq` as a pure function would amount to positing the axiom that Coq's polymorphic equality is decidable, which is a very strong and potentially dangerous assumption. With the IMPURE monad, those axioms have a very simple model in Coq: the function that never returns. The `struct_eq_correct` axiom is also presumably consistent with extraction to OCaml even if function equality is axiomatized as extensional (which is the case in COMPCERT). This is because OCaml's structural equality raises an exception in case of function comparisons.

Technically, we could completely do away with `struct_eq`, with the usual technique in Coq consisting of defining decidable monomorphic equalities. It is just a convenience when we are already computing in the IMPURE monad. And it is probably also a bit more efficient.

Let me explain why doing so is interesting in practice. Knowing that pointer equality *always implies* structural equality, we can prove that:

Lemma 7.2.1 (Very fast equality is correct)**.**

```
Lemma very_fast_eq_correct {A} (x y: A): very_fast_eq x y = true → x=y
```

Proof. By unfolding `very_fast_eq` *and exploiting the* `safe_coerce_correct` *lemma, we reduce the goal to exactly lemma* `phys_eq_correct` *(from §2.4.4).* □

TOTAL ORDER OVER SYMBOLIC VALUES The above enhanced pointer equality solves two issues: its purity avoids the monad, and when used in conjunction with hash-consing, it enables to define a total order over symbolic terms. In fact, each symbolic value stores an integer in its root, which is expected to uniquely identify it [23, §3.3.2]. This identifier provides a simple and efficient total order over symbolic values that we do not even need to prove correct.

In IMPURE, unique hash-consing identifiers are abstracted in a type called "`hashcode`" [◇]. This type is declared as a Coq *axiom* (i.e. stating that there exists an opaque type called "`hashcode`"), and is extracted to the OCaml type used by hash-consing oracles (the integer—"`int`"—type). The comparison of hash codes must also be performed via axioms, as this type is abstract for Coq:

```
Axiom hashcode_eq: hashcode → hashcode → bool
Extract Inlined Constant hashcode_eq ⇒ "(=)"

Axiom hashcode_lt: hashcode → hashcode → bool
Extract Inlined Constant hashcode_lt ⇒ "(<)"
```

Observe the return type of these axioms: contrarily to structural and pointer equalities, we use a pure Boolean value here.

In theory, two values identified with the same unique hash code should always be equal. Nonetheless, deducing pointer equality from the equality of hash codes could be dangerous in case of a poor management of the hash-consing mechanism. Our framework never relies on hash codes to infer equality: even when hash codes are equal, pointer equality is *still upheld*:

```
Definition fast_eq {A} (h: A → hashcode) (x y: A): bool :=
  if hashcode_eq (h x) (h y) then very_fast_eq x y else false

Lemma fast_eq_correct A (h: A → hashcode) (x y: A):
  fast_eq h x y = true → x=y
  (* By congruence for the case where (h x) <> (h y),
  and by applying very_fast_eq_correct otherwise. *)
Qed
```

With the `fast_eq` definition, we are now able to provide a full comparison for hash codes. The latter is used *exclusively for the normalization process* of our rewriting engine.

Definition 7.2.8 (A total order over symbolic, hash-consed terms)**.**

```
Definition fast_cmp {A} (h: A → hashcode) (x y: A): comparison :=
  if fast_eq h x y then Eq
  else if hashcode_lt (h x) (h y) then Lt else Gt

Lemma fast_cmp_Eq_correct A (h: A → hashcode) (x y: A):
  fast_cmp h x y = Eq → x=y
  (* For the ifso case, by applying fast_eq_correct, and by congruence otherwise. *)
Qed
```

Similarly to [23] & [135]†, the correctness of our normalization modulo hash-consing is therefore only derived from a sound Coq model of OCaml pointer equality. Note that a bug in the hash-consing mechanism makes—at worse—the verifier fail to prove some expected equalities.

However, the impure version of physical equality is *not* entirely replaced in the framework. Indeed, the pure test, which aborts the program in case of unequal values, is *not suitable for use in the hash-consing factory*, where it is necessary to compare values without aborting the program.

7.2.4.2 *Instantiating the Factory*

The Impure library provides a polymorphic $\mathtt{hCons} : \mathtt{hashP}\ A \to ??(\mathtt{hashConsing}\ A)$ factory that creates a hash-consing function for any type A. To do so, it requires three pieces of information about the type to be hash-consed, transmitted with the $\mathtt{hashP}\ A$ record [◇]:

$$\begin{aligned}\mathtt{hashP}\ A \triangleq \{\ & \mathtt{hash\_eq} : A \to A \to ??\mathtt{bool};\\ & \mathtt{get\_hid} : A \to \mathtt{hashcode};\\ & \mathtt{set\_hid} : A \to \mathtt{hashcode} \to A;\}\end{aligned}$$

The hash codes getters and setters are available in Coq here [◇]; and the hash equality functions here [◇].

We want to hash the three mutual variants of our symbolic values, so we have to define a hashP record for each of them. Our three get_hid getters are very simple: they contain a pattern-matching on each constructor of the mutual variant considered, and return their "hid" field. The set_hid setters also match constructors to update their "hid" field and return the new symbolic value variant. And hash_eq must be filled with an equality function (in the monad) that compares two values of type A. For instance, the *smem* mutual variant of hash_eq is defined as follows:

```
Definition smem_hash_eq (sm1 sm2: smem): ?? bool :=
  match sm1, sm2 with
  | Sinit _, Sinit _ ⇒ RET true
  | Sstore sm1 chk1 addr1 lsv1 sv1 _, Sstore sm2 chk2 addr2 lsv2 sv2 _ ⇒
     DO b1 ←~ lsv1 == lsv2;;
     DO b2 ←~ sm1 == sm2;;
     DO b3 ←~ sv1 == sv2;;
     DO b4 ←~ struct_eq chk1 chk2;;
     if b1 && b2 && b3 && b4 then struct_eq addr1 addr2 else RET false
  | _, _ ⇒ RET false
  end
```

Above, smem_hash_eq uses pointer equality to compare symbolic value's variants, and structural equality otherwise (here for chunks and addresses). Our comparison only checks addresses if necessary to be more efficient. The *sval* and *list_sval* versions of hash_eq are similar, and do not require recursion (e.g. for *list_sval*, list constructors $\mathtt{Scons}(sv, lsv)$ are directly compared by applying "==" on both sv and lsv).

Once we have those three hashP records, applying hCons to them gives us the three corresponding hash-consing functions hC_sv, hC_lsv and hC_sm:

```
Record HashConsingFcts := {
    hC_sv: hashinfo sval → ?? sval;
    hC_lsv: hashinfo list_sval → ?? list_sval;
    hC_sm: hashinfo smem → ?? smem
}
```

Where hashinfo is a record containing the value and a list of hash codes corresponding to the hash IDs of its subterms.

```
Record hashinfo (A : Type) : Type := Build_hashinfo { hdata : A;  hcodes : list hashcode }
```

Lists of hash codes are built by generating—via the library hash function—the hash IDs of each argument. For instance, a Sop symbolic value's list of hash codes will contain the hash ID of the operation and of its list of arguments (e.g. the hid field of type *list_sval*). We expect that hashinfo values verify the equivalence $\forall(x\ y : \mathtt{hashinfo})$,
$\mathtt{hash\_eq}\ (\mathtt{hdata}\ x)\ (\mathtt{hdata}\ y) \iff (\mathtt{hcodes}\ x) = (\mathtt{hcodes}\ y)$.

The correctness properties of this instantiation are proved thanks to a lemma from Impure [◇], stating that hash-consing functions obtained with hCons behave like the identity if we ignore hash codes. In our Coq code, we group these properties in a class:

```
Class HashConsingHyps (HCF: HashConsingFcts) := {
    hC_sv_correct: ∀ s, HCF.(hC_sv) s ⇝ s' → ∀ ctx,
      σ_sv(ctx,hdata s) = σ_sv(ctx,s');
    hC_lsv_correct: ∀ l, HCF.(hC_lsv) l ⇝ l' → ∀ ctx,
      σ_lsv(ctx,hdata l) = σ_lsv(ctx,l');
```

```
    hC_sm_correct: ∀ m, HCF.(hC_sm) m ⇝ m' → ∀ ctx,
      σ_sm(ctx,hdata m) = σ_sm(ctx,m')
}
```

This allows each implementation module to be parametrized by those functions: the prelude, and both the symbolic execution and the symbolic invariants modules. They are declared as Coq sections taking as context variables the above record and class.

7.2.4.3 *Projection of "Fake" Symbolic Values*

All symbolic values that will be created during SE will be *immediately hash-consed*; however, to simplify the rewriting engine specified in §7.2.5, it is easier to let it produce "fake" (i.e. *partially* hashed) symbolic values, so that it does not need to operate inside the IMPURE monad, and will be easier to prove correct (i.e. hash-consing is delayed). On the other hand, when these partially hashed values return from the rewriting engine to the SE, they need to be *fully* hashed prior to being assigned to a symbolic register. Moreover, supposing arbitrary rewrites, the resulting "fake" values might be nested, which requires us to potentially hash them recursively.

The prelude module thus defines a (recursive) projection function "`fsval_proj` : $sval \to ??sval$" [⋄] that transforms any partially hashed symbolic value into its hashed version. If the argument was already hash-consed (we verify this by comparing its `hid` field with the default `unknown_hid`, equal to -1), the value is not touched; otherwise, it is hash-consed (recursively) and equipped with a new `hid`. The result is returned within the IMPURE monad, so that it cannot be considered as deterministic. Function `fsval_proj` thus assumes that in the input term, every subterm with a `hid` different from `unknow_hid` is fully hashed. If the rewrites do not respect this precondition, the only possible consequence is that the verifier will fail to prove equalities, which does not contradict its correctness.

The *partial correctness* theorem of our projection function states that [⋄]:

$$\forall \mathtt{sv}, \mathtt{fsval\_proj}\ \mathtt{sv} \rightsquigarrow \mathtt{sv}' \implies \forall \mathtt{ctx}, \sigma_{\mathtt{sv}}(\mathtt{ctx}, \mathtt{sv}) = \sigma_{\mathtt{sv}}(\mathtt{ctx}, \mathtt{sv}')$$

7.2.5 *Specification of the Rewriting Engine*

Two different applications of this rewriting mechanism are proposed in §7.6.

Our SE framework includes a rewriting engine based on proven normalization rules. We use it to rewrite classical operations in §7.2.7.3 and conditional branches in §7.4.2. Rewriting occurs on *both the source and target sides* of SE. For operations, rewriting also occurs on the invariants' execution[3]. The engine is called each time an operation is assigned to a symbolic register, and at each execution of a BTL block's conditional branch.

7.2.5.1 *Our Generic Rewriting System*

The `rewrite_ops` subroutine of Definition 7.2.11 handles operations, and has the following signature "`rewrite_ops` : $R \to op \to list\_sval \to sval$ `option`". Similarly, branches are managed with the `rewrite_cbranches` subroutine from Definition 7.4.2, whose specification is "`rewrite_cbranches` : $R \to ristate \to cond \to \overrightarrow{reg_{\text{arg}}} \to (cond * list\_sval)$ `option`". In both signatures, R is a sum type—defined in §7.6—indicating the set of normalization rules to use. For operations, this simply means that given a BTL operation and its list of symbolic arguments, the engine may yield a rewritten symbolic value. The case of branches is a bit different: instead of directly taking the symbolic arguments, we use the concrete arguments (i.e. registers) along with an instance of the current internal state to retrieve them[4]. Since rewriting a conditional can not only change its (symbolic) arguments, but also the condition type itself, `rewrite_cbranches` may return a tuple containing both the (potentially) new condition and the new symbolic arguments.

Our normalization rules are always applied *at the top of hash-consed terms*, like a smart constructor. The rewritten terms are then turned into proper hash-consed terms using the projection function of §7.2.4.3. This both simplifies the implementation of normalization rules and its formal proof. Function `rewrite_ops` yields a single "fake" symbolic value, and the pair returned by `rewrite_cbranches`

[3] This is not the case for branches, since they are not represented in our invariants.

[4] Note that the same signature used for operation rewrites could have been applied here; the difference is merely technical.

contains "fake" symbolic arguments. We do not hash-cons conditions, meaning that the new pair is directly incorporated into the symbolic binary decision tree, as visible in Definition 7.4.3.

7.2.5.2 *Correctness Property to Satisfy*

When implementing the rewriting mechanism by adding normalization rules specific to a given optimization, one needs to maintain the related correctness property.
For operation rewrites, it is expressed as follows:

Theorem 7.2.2 (The rewriting procedure for operations is correct [◇]). *Let "*$\mathtt{R} : R$*" the set of rules,* $\mathtt{ctx}$ *the block level context,* $\mathtt{op}$ *the operation, and* $\mathtt{lsv}$ *its symbolic arguments. For all* $\mathtt{lsv}'$ *and* $\mathtt{fsv}$ *of types* $\mathit{list\_sval}$ *and* sval*, respectively, and for all lists of concrete arguments (as CompCert values)* $\overrightarrow{\mathtt{arg}}$*, we want:*

$$\mathtt{rewrite\_ops\ R\ op\ lsv} = \mathit{Some}\ \mathtt{fsv} \implies \sigma_{\mathtt{lsv}}(\mathtt{ctx}, \mathtt{lsv}) = \sigma_{\mathtt{lsv}}(\mathtt{ctx}, \mathtt{lsv}') \implies$$
$$\sigma_{\mathtt{lsv}}(\mathtt{ctx}, \mathtt{lsv}') = \mathit{Some}\ \overrightarrow{\mathtt{arg}} \implies \sigma_{\mathtt{sv}}(\mathtt{ctx}, \mathtt{fsv}) = \lfloor(\mathtt{op}, \overrightarrow{\mathtt{arg}})\rfloor_{\mathtt{ctx}}$$

We first assume that the rewrite "$(\mathtt{op}, \mathtt{lsv})_{\mathtt{R}} \mapsto \mathtt{fsv}$*" succeeded. Second, we assume that* $\mathtt{lsv}'$ *is semantically equivalent to* $\mathtt{lsv}$*, and that evaluating* $\mathtt{lsv}'$ *yields the list of the operation's concrete arguments. Then, the theorem states that evaluating the rewritten "fake" symbolic value* $\mathtt{fsv}$ *leads to the same result as executing the concrete operational semantics on* $(\mathtt{op}, \overrightarrow{\mathtt{arg}})$*. All steps are performed under the execution context* $\mathtt{ctx}$*.*

Note that we do not even require $\mathtt{lsv}$ *and* $\mathtt{lsv}'$ *to be syntactically equal here: they only need to evaluate to the same result.*

The correctness property for branch rewrites makes use of the refinement relation to prove the mapping from concrete to symbolic arguments[5]. In function `rewrite_cbranches`, we thus retrieve symbolic arguments by accessing the state using Definition 7.2.3.

Theorem 7.2.3 (The rewriting procedure for branches is correct [◇]). *We consider the rules set* $\mathtt{R}$*, the context* $\mathtt{ctx}$*, the current internal refined state* $\mathtt{hrs}$*, its theoretical equivalent* $\mathtt{sis}$*, and the* $(\mathtt{cond}, \vec{\mathtt{r}})$ *pair containing the condition type and its concrete arguments (which is, this time, provided as a list of registers). Then, for all rewritten pair* $(\mathtt{cond}', \mathtt{flsv})$ *with* $\mathtt{flsv}$ *a "fake" (i.e. partially hash-consed)* $\mathit{list\_sval}$*, and for all "*$\mathtt{lsv} : \mathit{list\_sval}$*", the property to verify is:*

$$\mathtt{rewrite\_cbranches\ R\ hrs\ cond}\ \vec{\mathtt{r}} = \mathit{Some}\ (\mathtt{cond}', \mathtt{flsv}) \implies \mathtt{ris\_refines\ ctx\ hrs\ sis} \implies$$
$$\mathtt{ris\_ok\ ctx\ hrs} \implies \mathtt{lmap\_sv\ sis}\ \vec{\mathtt{r}} = \mathit{Some}\ \mathtt{lsv} \implies$$
$$\mathtt{eval\_scondition\ ctx\ cond\ lsv} = \mathtt{eval\_scondition\ ctx\ cond}'\ \mathtt{flsv}$$

Here also, we assume a successful rewrite "$(\mathtt{cond}, \vec{\mathtt{r}})_{\mathtt{R}} \mapsto (\mathtt{cond}', \mathtt{flsv})$*". In addition, we need a valid refinement relation between* $\mathtt{hrs}$ *and* $\mathtt{sis}$*, and a valid* $\mathtt{hrs}$ *in the sense of property 7.2.4. The theorem assumes, with the help of Definition 6.4.18, a correct mapping from concrete arguments to symbolic ones through* $\mathtt{sis}$*, yielding the list* $\mathtt{lsv}$*. If these conditions are satisfied, then pairs* $(\mathtt{cond}, \mathtt{lsv})$ *and* $(\mathtt{cond}', \mathtt{flsv})$ *are proved to evaluate to the same result (of type "*$\mathtt{bool\ option}$*", cf. Definition 6.4.2).*

Of course, the proofs of both theorems depend on the rewrites that are actually implemented in the backend. Normalization rules, which are in the option monad, must be proved to always yield an output semantically equivalent to their input when they succeed.

7.2.6 *Hash-Consed Symbolic Register Access*

With our hash-consing instance, we can now propose a hash-consed version of Definition 7.2.3, that builds a real hashed input term when the value to access does not exist. Of course, this forces our new getter to be within the Impure monad.

Definition 7.2.9 (Implementation of symbolic register getter). The code is almost the same as in the intermediate refinement version, except that the "`None`" case of the option monad is replaced by the failure case of the Impure monad, which aborts the compilation process with an error.

[5] This is a technical detail. It comes from the fact that operations are rewritten during the assignment procedure of symbolic regsets, which also requires symbolic arguments. We thus compute them before for operations, in Definition 7.3.4. We could have done the same for branches, although this conception is a remnant of RTLpath.

```
Definition hrs_sreg_get (hrs: ristate) r: ?? sval :=
  match hrs.(ris_sreg)!r with
  | None ⇒ if ris_input_init hrs then hSinput r
           else FAILWITH "hrs_sreg_get: dead variable"
  | Some sv ⇒ RET sv
  end
```

Where "`hSinput`" constructs a hash-consed `Sinput` value. If the requested symbolic register ID is undefined in target mode (when `ris_input_init` is false), it means that the execution tried to access a *dead variable*, as indicated by the error message of the monad.

This new definition is proved correct by two separate properties. First, by relating it to the intermediate refinement's version:

Lemma 7.2.4 (Symbolic register access is correct [◇])**.**

$$\begin{aligned}&\forall(\mathtt{hrs} : \mathit{ristate})\ \mathtt{r}, \mathtt{hrs\_sreg\_get\ hrs\ r} \rightsquigarrow \mathtt{hsv} \implies\\ &\forall \mathtt{ctx}\ (\mathtt{sreg} : \mathtt{reg} \to \mathtt{sval\ option}), (\forall \mathtt{r}', |\sigma_{\mathtt{sv}}|(\mathtt{ctx}, \mathtt{hrs\ r}') = |\sigma_{\mathtt{sv}}|(\mathtt{ctx}, \mathtt{sreg\ r}')) \implies\\ &|\sigma_{\mathtt{sv}}|(\mathtt{ctx}, \mathit{Some}\ \mathtt{hsv}) = |\sigma_{\mathtt{sv}}|(\mathtt{ctx}, \mathtt{sreg\ r})\end{aligned}$$

Proof. By extracting the monad result (since the first hypothesis assumes a success) and by rewriting the second hypothesis. □

And second, by showing that if accessing register r in refined state hrs returns a value, and if hrs is a valid refinement of theoretical state sis, then sis cannot be undefined for r.

Lemma 7.2.5 (A successful register access on refined state cannot be undefined on its abtract equivalent [◇])**.**

$$\begin{aligned}&\forall(\mathtt{hrs} : \mathit{ristate})\ \mathtt{r}, \mathtt{hrs\_sreg\_get\ hrs\ r} \rightsquigarrow \mathtt{hsv} \implies\\ &\forall \mathtt{ctx\ sis}, \mathtt{ris\_ok\ ctx\ hrs} \implies \mathtt{ris\_refines\ ctx\ hrs\ sis} \implies\\ &\mathtt{sis\ r} = \mathit{None} \implies \mathtt{False}\end{aligned}$$

Proof. By rewriting the alive equivalence from the third hypothesis. □

7.2.7 *Setting Values in a Symbolic Register*

7.2.7.1 *Set & Reduce Operation on Refined Symbolic Regsets*

When a refined internal state is in source mode, we saw that any undefined register r will be mapped by default to "`fSinput r`". Hence, executing the assignment "`ris r ← fSinput r`" when "$\mathtt{ris.(ris\_input\_init)} = \mathtt{true}$" would be *redundant* with the default value. We avoid such useless assignments using a set & reduce operation over the symbolic regset:

Definition 7.2.10 (Set and reduce value into the symbolic regset of refined states)**.**

```
Definition red_PTree_set (r: reg) (sv: sval) (ris: ristate): PTree.t sval :=
  match (ris_input_init ris), sv with
  | true, Sinput r' _ ⇒
      if r = r' then (* Remove useless assignment *)
        PTree.remove r' ris.(ris_sreg)
      else ris.(ris_sreg)!r ← sv
  | _, _ ⇒ ris.(ris_sreg)!r ← sv
  end
```

The function returns a new `PTree` that should replace the old one in ris. It features several related properties, available in Coq online [◇]; I summarize some of them below:

1. Correctness "Some": $\forall \mathtt{ctx}\ (\mathtt{r}\ \mathtt{r0} : \mathit{reg})\ \mathtt{ris}\ \mathtt{sv}\ \mathtt{sv}'$,
   $\{\mathtt{ris\ with\ sreg} = \mathtt{red\_PTree\_set\ r\ sv\ ris}\}\ \mathtt{r0} = \mathtt{Some}\ \mathtt{sv} \implies$
   $\{\mathtt{ris\ with\ sreg} = \mathtt{ris.(ris\_sreg)!r} \leftarrow \mathtt{sv}\}\ \mathtt{r0} = \mathtt{Some}\ \mathtt{sv}' \implies$
   $\sigma_{\mathtt{sv}}(\mathtt{ctx}, \mathtt{sv}) = \sigma_{\mathtt{sv}}(\mathtt{ctx}, \mathtt{sv}')$

2. Correctness "None": $\forall$ctx (r r0 : *reg*) ris sv,
   {ris *with* sreg = red_PTree_set r sv ris} r0 = None $\iff$
   {ris *with* sreg = ris.(ris_sreg)!r $\leftarrow$ sv} r0 = None

3. Refines "Some": $\forall$ctx (r r0 : *reg*) ris sis sv sv′ osv,
   σ_{sv}(ctx, sv) = σ_{sv}(ctx, sv′) $\implies$
   ris_ok ctx ris $\implies$ ris_refines ctx ris sis $\implies$
   {ris *with* sreg = red_PTree_set r sv ris} r0 = osv $\implies$
   $|\sigma_{sv}|$(ctx, osv) = $|\sigma_{sv}|$(ctx, r = r0 ? Some sv′ : sis r0)

4. Refines "None": $\forall$ctx (r r0 : *reg*) ris sis sv sv′,
   ris_ok ctx ris $\implies$ ris_refines ctx ris sis $\implies$
   {ris *with* sreg = red_PTree_set r sv ris} r0 = None $\iff$
   set_sreg r sv′ sis r0 = None

7.2.7.2 *Root Symbolic Values as Right-Hand Sides of Affectations*

The set & reduce operation of Definition 7.2.10 assigns a symbolic value to a refined internal state, producing a new symbolic regset. Nonetheless, our symbolic simulation is designed to handle either executing BTL blocks (i.e. BTL instructions of Figure 5.1) or compact symbolic invariants (i.e. values of type *ival* from Definition 6.2.3). Therefore, we require an operation that can assign both types of values—BTL instructions and *ival*—to a given internal state, by transforming them into symbolic values beforehand.

Since only two types of BTL instructions modify the symbolic register set (arithmetic operations and memory loads[6], cf. §6.2.2.2), our solution defines an assignment operation working with the *root symbolic values* of Definition 6.2.3. An ival of type Iop is natively compatible (as it directly *includes* a root_op), and type Ireg trivially reduces to a symbolic value (it suffices to retrieve the value attached to its register, see §7.3.2). For BTL's arithmetic operations and loads, the conversion to root_op is a natural abstraction.

In the next section, we convert these root symbolic values to actual ones while simplifying them (with eventual rewrites). The resulting *sval* is then assigned to the symbolic regset. However, potentially trapping symbolic values must be appended to the list of "ok" values of the state without being rewritten. Doing so require a preserving (without rewriting) conversion from root_op to *sval*. In the theory, this was the role of function root_apply; its implementation—namely root_happly [$\diamond$]—is almost the same except that it replaces "fake" constructors (fSop and fSload) with hash-consing ones. Consequently, its old *sval* return type becomes ??*sval*.

7.2.7.3 *Simplifying Right-Hand Sides Before Writing in the State*

To validate the replacement of an instruction sequence with an equivalent, more efficient sequence, we must rewrite symbolic values before they are assigned to the symbolic state. For this purpose, we define a *simplification* function, which transforms a root symbolic value into an actual one while potentially rewriting it. Below, hSop and hSload are the hash-consed constructors for operations and loads, respectively.

Definition 7.2.11 (Simplify a root symbolic value)**.**

```
Definition simplify (rsv: root_op) (lsv: list sval) (lhsv: list_sval) (sm: smem): ?? sval :=
  match rsv with
  | Rop op ⇒
      match is_move_operation op lsv with
      | Some arg ⇒ fst_lhsv_imp lhsv
      | None ⇒
        match rewrite_ops R op lsv with
        | Some fsv ⇒ fsval_proj fsv
        | None ⇒ hSop op lhsv
```

[6] We do not consider final values because these are handled differently with their symbolic equivalent and are never applied to internal symbolic states.

```
      end
    end
| Rload _ chunk addr ⇒ hSload sm NOTRAP chunk addr lhsv
end
```

The mini rewrites for moves and loads originate from the RTLpath implementation of Six [133, §7.3.3].

Given a root value `rsv`, two representations of its arguments (here `lsv` and `lhsv`), and a symbolic memory:

- When `rsv` represents an arithmetic operation, we check if it is a move (a copy from a register to another) by examining `op` and the number of symbolic arguments (which should be exactly one). If it is indeed a move, we directly replace it with its first symbolic argument to lighten the symbolic state (function `fst_lhsv_imp` [⋄] extracts the head of `lhsv`). Otherwise, we call the rewriting engine for operations `rewrite_ops` with `op`, `lsv`, and a normalization rules policy R (see §7.6). If no rewriting applies on the operation, the function will yield `None` as the option result and will simply convert `rsv` to a hash-consed symbolic value (mimicking the behavior of `root_happly`). Otherwise, a partially hashed rewritten value is yielded, before being projected to its fully hashed equivalent with `fsval_proj`.
- When `rsv` is a memory load, we build a corresponding hash-consed symbolic value with the "trapping" flag set to `NOTRAP` (i.e. false). This mini-rewriting allows to normalize the loads assigned in the symbolic register state (they are always marked as non-trapping) and does not cause any issues since trapping loads are in the precondition (list `ok_rsval`) anyway.

The list of symbolic arguments is passed in two formats because it is simpler, and avoids multiple conversions[7].

7.2.7.4 *Assigning a Root Symbolic Value to a Refined Internal State*

The assignment of a `root_op` to a *ristate* is split in two functions: one for the potentially trapping case, and one for the non-trapping case. Both functions take the two representations of arguments, the root value, the refined state, and the destination register `r`. Assigning a non-trapping value is quite simple, and only requires computing the rewritten symbolic value with Definition 7.2.11 before setting the result in the state:

Definition 7.2.12 (Non-trapping root symbolic value assignment [⋄])**.**

$$\begin{aligned}&\texttt{hrs\_rsv\_set\_notrap}\ \texttt{r}\ (\texttt{lsv} : \overrightarrow{\mathit{sval}})\ (\texttt{lhsv} : \mathit{list\_sval})\ \texttt{rsv}\ \texttt{hrs} : ??\mathit{ristate} \triangleq\\ &\quad \texttt{DO simp} \leftsquigarrow \texttt{simplify rsv lsv lhsv hrs.(ris\_smem);;}\\ &\quad \texttt{RET}\ \{\,\texttt{hrs with sreg} = \texttt{red\_PTree\_set r simp hrs}\,\}\end{aligned}$$

For values that may trap, the procedure is a bit more complex: in addition to calling the simplification procedure, we might have to modify the state's precondition. In fact, it depends on the `si_mode` Boolean, which is false by default, and true only when applying the (history or gluing) invariant on a final state (i.e. in *output*). Intuitively, this simply encodes the fact that transferring an invariant after symbolic execution must not add a potential failure (i.e. an operation that may trap) to the final state. In our definition, we enforce this constraint by verifying that the value to assign is *already present* in the precondition of `hrs` (when `si_mode` = `true`). When we apply the input invariant, or during SE (i.e. when `si_mode` = `false`), we only *append the value* to `ok_rsval`.

Definition 7.2.13 (Possibly-trapping root symbolic value assignment [⋄])**.**

$$\begin{aligned}&\texttt{hrs\_rsv\_set\_trap}\ \texttt{r}\ (\texttt{lsv} : \overrightarrow{\mathit{sval}})\ (\texttt{lhsv} : \mathit{list\_sval})\ \texttt{rsv}\ \texttt{hrs} : ??\mathit{ristate} \triangleq\\ &\quad \texttt{DO sv} \leftsquigarrow \texttt{root\_happly rsv lhsv ris.(ris\_smem);;}\\ &\quad \texttt{DO ok\_lsv} \leftsquigarrow (\\ &\qquad \textbf{if}\ \neg\texttt{hrs.(si\_mode)}\ \textbf{then}\ \texttt{RET}\ (\texttt{sv} :: \texttt{hrs.(ok\_rsval)})\ \textbf{else}\ (\\ &\qquad\quad \textbf{if}\ \texttt{sv} \in \texttt{hrs.(ok\_rsval)}\ \textbf{then}\ \texttt{RET hrs.(ok\_rsval)}\end{aligned}$$

[7] For instance, the `is_move_operation` comes from a `RTL` module, and requires a Coq list, while our hash-consed constructors operate with the inductive `list_sval` type.

```
      else FAILWITH "adding potential failure"));;
    DO simp ⇜ simplify rsv lsv lhsv hrs.(ris_smem);;
    RET {hrs with sreg = red_PTree_set r simp hrs; ok_rsval = ok_lsv}
```

Technically, the inclusion test compares (within the IMPURE monad) iteratively each value of `hrs.(ok_rsval)` with *sv* using the efficient pointer equality check [⋄]. We could probably reduce the cost of this test to a constant amortized cost with a more efficient representation of the `ok_rsval` list. However, we have not observed any efficiency issues in our experiments so far.

7.3 REFINED EXECUTION OF SYMBOLIC INVARIANTS

The goal is to execute our sequential, compact symbolic invariants equivalently to the `tr_sis` function of the theory (Definition 6.5.1). Especially, when applying an invariant on a refined state, one must take into account the current symbolic regset.

A compact sequence of assignments of symbolic values, as formalized in §6.2.2.2, contains values of type `ival`, whose arguments (of type `ireg`) refer to either the input value or to the current one (depending on their `force_input` Boolean). When the invariant is applied to a (whether final or not) state, taking the "input" value of an argument means taking the value as defined in this state. On the other hand, the "current" value means the one from the state in which we have already applied some part of the sequence. In any case, if an invariant's value depends on symbolic values defined in the initial or current symbolic state, those must be retrieved and substituted within the invariant's value for the result to be correct.

7.3.1 *Sequential Execution*

Our idea is to start by *duplicating* the refined state on which we want to apply the sequential invariant: one version will never be modified, and will serve us as the "input" reference, and the other will change incrementally by executing invariants in the sequence.

In point 1. of §6.3.3.2, I explained how to build the finite parallel assignment of symbolic values by accumulating sequential assignments, with the "*exec_seq*" function. Now, we define *exec_seq_imp* as the implementation of this accumulation for refined symbolic internal states:

Definition 7.3.1 (Sequential execution of compact invariants on refined states)**.**

```
Fixpoint exec_seq_imp (hrs hrs_old: ristate) (l: list (reg*ival)): ?? ristate :=
  match l with
  | nil ⇒ RET hrs
  | (r,iv)::l ⇒
      DO hrs' ⇜ hrs_ival_set hrs hrs_old r iv;;
      exec_seq_imp hrs' hrs_old l
  end
```

We iterate over the list of invariants `l` (it should come from the `aseq` field of a CSASV), and with the two refined states `hrs` and `hrs_old` that are expected to be initially equal (i.e. they should be instantiated with the state to transfer). Then, function `hrs_ival_set` yields a new state `hrs'` by assigning to `r` the current invariant `iv`, and we recurse with the remaining sequence, and with `hrs'` in place of `hrs` (but `hrs_old` stays the same).

7.3.2 *Assigning an Invariant's Value to a Refined Internal State*

The behavior of `hrs_ival_set` depends on `iv`: if the invariant is of type "`Ireg ir`", we retrieve "`regof ir`" from either `hrs` or `hrs_old`. Otherwise, the "`root_op`" inside "`Iop`" has to be executed after possibly substituting arguments. We define it as follows, with the help of Definitions 7.3.3 and 7.3.4 below:

Definition 7.3.2 (Setter for an invariant value [◇])**.**

$$\begin{aligned}&\texttt{hrs\_ival\_set}\ (\texttt{hrs}\ \texttt{hrs\_old} : \mathit{ristate})\ (\texttt{r} : \mathit{reg})\ (\texttt{iv} : \mathit{ival}) : ??\mathit{ristate} \triangleq\\ &\quad \textbf{match}\ \texttt{iv}\ \textbf{with}\\ &\quad |\ \texttt{Ireg ir} \to \texttt{DO sv} \leftsquigarrow \texttt{hrs\_ir\_get hrs hrs\_old ir;;}\\ &\qquad \{\texttt{hrs with sreg} = \texttt{red\_PTree\_set r sv hrs}\}\\ &\quad |\ \texttt{Iop rop args} \to \texttt{hrs\_rsv\_set r args rop hrs hrs\_old}\end{aligned}$$

Where hrs_ir_get is a simple wrapper around the normal getter function over symbolic regset of Definition 7.2.9:

Definition 7.3.3 (Access to an invariant register by selecting the right refined state)**.**

```
Definition hrs_ir_get hrs hrs_old ir: ?? sval :=
  if force_input ir then hrs_sreg_get hrs_old (regof ir) else hrs_sreg_get hrs (regof ir)
```

To manage the list of *ireg* arguments of an Iop, we implement a recursive function "$\texttt{hrs\_lir\_get} : \mathit{ristate} \to \mathit{ristate} \to \mathit{ireg} \to ??(\overrightarrow{\mathit{sval}} * \mathit{list\_sval})$" [◇]. It converts, relying on Definition 7.3.3, the vector of *ireg* into two representations of lists of symbolic values: a Coq list (of *sval*), and a *list_sval* inductive list. Both contain the same elements, in the same order, and are built simultaneously.

Then, hrs_rsv_set delegates the assignment to Definitions 7.2.12 & 7.2.13. To do so, it determines if the root symbolic value may trap or not, and converts the *ireg* list into both formats of symbolic value lists, as required by the assignment functions.

Definition 7.3.4 (Generic root symbolic value assignment [◇])**.** The may_trap function below is true when rsv is either a trapping load or a trapping operation, see [◇].

$$\begin{aligned}&\texttt{hrs\_rsv\_set r}\ (\texttt{lir} : \overrightarrow{\mathit{ireg}})\ \texttt{rsv hrs hrs\_old} : ??\mathit{ristate} \triangleq\\ &\quad \texttt{DO}\ (\texttt{lsv}, \texttt{lhsv}) \leftsquigarrow \texttt{hrs\_lir\_get hrs hrs\_old lir;;}\\ &\quad \textbf{if}\ (\texttt{may\_trap rsv lir})\ \textbf{then}\ \texttt{hrs\_rsv\_set\_trap r lsv lhsv rsv hrs}\\ &\quad \textbf{else}\ \texttt{hrs\_rsv\_set\_notrap r lsv lhsv rsv hrs}\end{aligned}$$

Symbolic values in lists lsv and lhsv are therefore either from hrs or hrs_old, in function of the force_input value of their corresponding *ireg*.

7.3.3 *Liveness Filtering*

The refined symbolic state obtained by sequentially executing a CSASV contains, in its symbolic regset, all the values assigned by the invariant sequence. To comply with the simulation theory, we now have to *filter* this symbolic regset to only keep variables defined as live in the "output" set of the CSASV. In other words, the idea is to provide an implementation for the "build_alive" function outlined in point 2. of §6.3.3.2.

Definition 7.3.5 (Building a symbolic regset of the compact invariant's live variables)**.** Given loutputs a list made from the "output" regset of a CSASV, and hrs the state on which we executed the CSASV's sequence (i.e. its "aseq" field), we build—from an empty symbolic regset PTree.empty—the filtered symbolic regset as follows:

```
Fixpoint build_alive_imp (loutputs: list reg) (hrs: ristate): ?? (PTree.t sval) :=
  match loutputs with
  | nil ⇒ RET (PTree.empty sval)
  | r :: tl ⇒
      DO sreg ↜ build_alive_imp tl hrs;;
      DO hsv ↜ hrs_sreg_get hrs r;;
      RET (sreg!r ← hsv)
  end
```

7.3.4 *Transferring Compact Invariants on a Refined State*

7.3.4.1 *Applying a Single Compact Invariant*

Grouping the sequential execution and the liveness filtering together gives us the invariant transfer function for a single CSASV on a symbolic (refined) internal state:

Definition 7.3.6 (Single compact invariant transfer function [◇])**.**

```
tr_hrs_single hrs (csi : csasv) : ??ristate ≜
  DO hrs' ⇜ exec_seq_imp hrs hrs csi.(aseq);;
  DO sreg ⇜ build_alive_imp (Regset.elements csi.(outputs)) hrs';;
  RET {hrs' with sreg = sreg}
```

7.3.4.2 *Iteratively Unifying Compact Invariants*

Nevertheless, when applying the output invariant to a BTL block having several successors (e.g. with a jump table), the successors' invariants must be unified. In practice, we iteratively apply each successor's invariant on the final state, while ensuring that the resulting union is *consistent*: a symbolic register should never be defined with two different symbolic values.

In accordance with what I wrote in §6.5.2.2, we use a specific "most-defined" relation over symbolic values to perform the union of symbolic regsets. If both hold a value for the same register, we compare the two values by pointer equality and fail in case of a negative result:

Definition 7.3.7 (Symbolic equality implementation for union of symbolic regsets [◇])**.**

```
symbolic_eq_sv_imp sv1 sv2 : ??sval ≜
  if sv1 == sv2 then RET sv1
  else FAILWITH "symbolic_eq_sv_imp: no more defined symbolic value"
```

Notice how this implementation is much simpler than the `symbolic_eq` function from §6.5.2.2: here, we directly compare sv_1 and sv_2 without prior substitution or evaluation. Below, I note "$sreg_1 \cup_{\text{mostdef}} sreg_2$" the most-defined union operation for $sreg_1$ and $sreg_2$ using this consistency check to solve conflicting cases.

To unify output invariants, we implement a recursive transfer function working with a list of CSASVs. The latter is intended to be used only on final symbolic values. Input invariants on initial states never require a union[8], and are executed using Definition 7.3.6.

Definition 7.3.8 (Recursive compact invariants transfer function [◇])**.**

```
tr_hrs_rec hrs (lcsi : csasv⃗) : ??ristate ≜
  match lcsi with
  | [] → DO hrs' ⇜ tr_hrs_single hrs {aseq = []; outputs = ∅};;
         RET {hrs with ris_input_init = false}
  | [csi] → DO hrs' ⇜ tr_hrs_single hrs csi;;
         RET {hrs with ris_input_init = false}
  | csi :: lcsi' →
    DO hrsr ⇜ tr_hrs_rec hrs lcsi';;
    DO hrsl ⇜ tr_hrs_single hrs csi;;
    DO sreg ⇜ hrsl.(ris_sreg) ∪mostdef hrsr.(ris_sreg);;
    RET {hrsr with sreg = sreg;
          ok_rsval = hrsl.(ok_rsval) ++ hrsr.(ok_rsval)}
```

[8] Since each block has its own input invariant.

When `lcsi` is empty (e.g. for tail calls), the transfer is performed with the empty CSASV; otherwise, when it contains a unique compact invariant, we rely on Definition 7.3.6. The (head-)recursive case computes the transfer for the remaining list of CSASVs and for the current one. Then, it applies the most-defined union on their regsets, and concatenates their preconditions (with operator "++").

After the execution of an output invariant, we do not want to keep the default value for dead variables (precisely because we just filtered them with the invariant's output set). Hence, terminal cases of `tr_hrs_rec` switch the state to target mode by setting its `ris_input_init` field to false.

Finally, proving the correctness of the output invariants' transfer requires showing that it is only called on states which are in source mode, and whose `si_mode` Boolean is true. Indeed, output invariants are always applied on the source symbolic state, and must never be able to add a potential failure. We verify this constraint by wrapping our recursive transfer function into a "`tr_hrs`" function with the same signature [$\diamond$], that raises an error and aborts the simulation in case one of the two Booleans is not true.

7.3.4.3 *Application on Final Symbolic Values & States*

Section §6.5.2.2 of the simulation theory gave the signature of a `si_sfv` function that models the building method of an output FPASV according to a final symbolic value. Its Coq implementation (i.e. with a CSASV) is written as follows:

Definition 7.3.9 (Applying output invariants for a refined internal state)**.**

```
Definition tr_sfv hrs (gm_select: pc → csasv) sfv: ?? ristate :=
  match sfv with
  | Sgoto pc ⇒ tr_hrs hrs [gm_select pc]
  | Scall sig svid args res pc ⇒
      if test_clobberable (gm_select pc) res then
        tr_hrs hrs [(csi_remove res (gm_select pc))]
      else FAILWITH "ri_sfv: Scall res not clobberable"
  | Sbuiltin ef args bres pc ⇒
      match reg_builtin_res bres with
      | Some r ⇒
          if test_clobberable (gm_select pc) r then
            tr_hrs hrs [(csi_remove r (gm_select pc))]
          else FAILWITH "ri_sfv: Sbuiltin res not clobberable"
      | None ⇒
          if test_csifreem (gm_select pc) then
            tr_hrs hrs [gm_select pc]
          else FAILWITH "ri_sfv: Sbuiltin memory dependent"
      end
  | Stailcall sig svid args ⇒ tr_hrs hrs []
  | Sreturn osv ⇒ tr_hrs hrs []
  | Sjumptable sv lpc ⇒ tr_hrs hrs (List.map (λ pc ⇒ gm_select pc) lpc)
  end
```

The partial application `gm_select` is expected to be instantiated with either HIs or GIs. For each kind of final value, we call the `tr_hrs` transfer function accordingly. As indicated in §6.5.2.2, the goto case consists simply of executing the invariant of the successor at `pc`, while tail calls and returns instantiate the transfer function with an empty list (because they are devoid of successors). The list of successors of a jump table is mapped to a list of CSASVs via `gm_select`. For calls and built-ins, we must still verify that neither their destination register nor the memory is clobbered (i.e. overwritten) by the invariant. Above, `test_clobberable` validates both requirements by iterating on the invariant's sequence of assignments; it implies property 6.3.15 when its result is true. We can use this check for calls, and for built-ins featuring a result register. If the built-in does not have a result register, we only ensure the invariant's freedom for the memory with `test_csifreem`; which, conversely, implies property 6.3.14.

To execute CSASVs over binary decision trees (i.e. whole refined states), we concretize function `tr_sstate` sketched in §6.5.2.2:

Definition 7.3.10 (Applying output invariants on the binary decision tree leaves)**.**

```
Fixpoint tr_hstate (gm_select: pc → csasv) (hst: rstate): ?? rstate :=
  match hst with
  | Rfinal hrs sfv ⇒
      DO hrs' ⇜ tr_sfv {hrs with si_mode = true} gm_select sfv;;
      RET (Rfinal hrs' sfv)
  | Rcond cond args ifso ifnot ⇒
      DO ifso' ⇜ tr_hstate gm_select ifso;;
      DO ifnot' ⇜ tr_hstate gm_select ifnot;;
      RET (Rcond cond args ifso' ifnot')
  | Rabort ⇒ RET Rabort
  end
```

The function being designed for output invariants, it sets the value of the si_mode Boolean to *true* for each leaf before calling tr_sfv.

7.4 REFINED SYMBOLIC EXECUTION OF BTL BLOCKS

In this section, we refine the symbolic execution with continuation of Definition 6.4.21.

7.4.1 *Mapping Registers to Symbolic Values and Executing Final Values*

The map operation from a register list to a symbolic value list (Definition 6.4.18) is implemented by iteratively calling our getter hrs_sreg_get (Definition 7.2.9) on each argument, with a function named hlist_sreg_get [◇].

Next, we refine the execution of final symbolic values with a function "hrexec_final_sfv : *final* → *ristate* → ??*sfval*" [◇]. The latter is very similar to Definition 6.4.19, except that it uses the hash-consed symbolic register getters to retrieve symbolic values, and is incorporated in the IMPURE monad. In particular, we had to adapt the auxiliary transformations for calls, tail calls, and built-ins' arguments (e.g. the sum_left_optmap and the mapping for built-ins' registers) to port them in the monad as well.

The execution of final values is proven correct w.r.t. the theory by showing that if we have two states hrs and sis such that hrs is valid (in the sense of property 7.2.4) and refines sis (in the sense of Relation 7.2.5), then the result of hrexec_final_sfv on a BTL instruction fi over hrs refines the one from the theoretical execution of fi over sis.

Lemma 7.4.1 (The refined SE of BTL final values is correct [◇])**.** *For any block level context* ctx*:*

$$\begin{aligned}&\texttt{hrexec\_final\_sfv fi hrs} \rightsquigarrow \texttt{sfv} \implies \forall \texttt{sfv}',\\ &\texttt{ris\_ok ctx hrs} \implies \texttt{ris\_refines ctx hrs sis} \implies\\ &\texttt{sexec\_final fi sis} = \mathit{Some}\ \texttt{sfv}' \implies\\ &\texttt{rfv\_refines ctx sfv sfv}'\end{aligned}$$

Proof. By case analysis on fi*, and by inverting the third and fourth hypotheses, it suffices to apply the right constructor of* rfv_refines*.* □

7.4.2 *Implementation of Block Execution*

To alleviate the proofs of our framework, our symbolic register assignment method reuses Definition 7.3.4. We wrap this definition so that both internal states are instantiated with the same parameter, and the list of *ireg* is built with an always true force_input Boolean:

Definition 7.4.1 ("Input" mode root symbolic value assignment [◇])**.**

$$\begin{aligned}&\texttt{hrs\_rsv\_set\_input r (args} : \overrightarrow{reg}\texttt{) rsv hrs} : ??\mathit{ristate} \triangleq\\ &\quad\texttt{hrs\_rsv\_set r (map input args) rsv hrs hrs}\end{aligned}$$

Here, both the `hrs` and `hrs_old` parameters of the invariant value setter are instantiated with the same current state `hrs`, and the list of *reg* arguments is mapped into an *ireg* list through function `input` from Definition 6.2.3.

Following the way paved by our theory, we start by implementing an execution function for store instructions:

```
Definition hrexec_store chunk addr args src hrs: ?? ristate :=
  DO hargs <~ hlist_sreg_get hrs args;;
  DO hsrc <~ hrs_sreg_get hrs src;;
  DO hm <~ hSstore hrs chunk addr hargs hsrc;;
  RET (rset_smem hm hrs)
```

With `hSstore` being the constructor for hash-consed `Sstore` symbolic memory values.

The efficient block symbolic execution for a whole `BTL` block resembles the theoretical definition, except for conditional branches instructions, that are rewritten on-the-fly. The latter rewriting is performed by function `cbranch_expanse` below [◇], which returns a pair containing a new type of condition and new (symbolic) arguments. If no rewriting applies, the returned pair is identical to the original condition's parameters.

Definition 7.4.2 (Rewriting branches during SE)**.** Exactly as function `rewrite_ops` from Definition 7.2.11, function `rewrite_cbranches` below takes a normalization rules policy `R` (see §7.6).

```
Definition cbranch_expanse (prev: ristate) (cond: condition)
  (args: list reg): ?? (condition * list_sval) :=
  match rewrite_cbranches R prev cond args with
  | Some (cond', vargs) =>
      DO vargs' <~ @fsval_list_proj HCF vargs;;
      RET (cond', vargs')
  | None =>
      DO vargs <~ hlist_sreg_get prev args ;;
      RET (cond, vargs)
  end
```

Definition 7.4.3 (Implementation of the recursive block SE)**.** Alike its equivalent in the theory, the `SE` works in continuation passing style. Below, `hrinit` is the initial (refined) symbolic internal state to start with.

```
Fixpoint hrexec_rec ib hrs (k: ristate → ?? rstate): ?? rstate :=
  match ib with
  | BF fin _ =>
      DO sfv <~ hrexec_final_sfv fin hrs;; RET (Rfinal hrs sfv)
  | Bnop _ => k hrs
  (** Invoke the rewriting engine for ops/loads *)
  | Bop op args dst _ =>
      DO next <~ hrs_rsv_set_input dst args (Rop op) hrs;; k next
  | Bload trap chunk addr args dst _ =>
      DO next <~ hrs_rsv_set_input dst args (Rload trap chunk addr) hrs;; k next
  | Bstore chunk addr args src _ =>
      DO next <~ hrexec_store chunk addr args src hrs;; k next
  | Bseq ib1 ib2 =>
      hrexec_rec ib1 hrs (λ hrs2 => hrexec_rec ib2 hrs2 k)
  | Bcond cond args ifso ifnot _ =>
      (** Invoke the rewriting engine for conditions *)
      DO res <~ cbranch_expanse hrs cond args;;
      let (cond, vargs) := res in
      DO ifso <~ hrexec_rec ifso hrs k;;
      DO ifnot <~ hrexec_rec ifnot hrs k;;
      RET (Rcond cond vargs ifso ifnot)
  end

Definition hrexec ib hrinit := hrexec_rec ib hrinit (λ _ => RET Rabort)
```

7.4.3 *Correctness of the Hash-Consed Symbolic Execution*

We prove two separate properties on Definition 7.4.3. Both of them assume a BTL block ib, an initial internal state hrinit refining the abstract internal state sinit, a block level context ctx, and a final symbolic state hst resulting from "hrexec ib hrinit".

Lemma 7.4.2 (Refined Block SE is correct on the source side [⋄]). *We assume a source initial refined state, whose* ris_input_init *Boolean is true, and successful refined and theoretical* SEs *of* ib *from initial states* hrinit *and* sinit*, respectively. If* sis*, the internal state resulting from the theoretical execution, satisfies validity property 6.4.12, and if* hrinit *is a valid refinement of* sinit*, then the result of the hash-consed execution must be a valid refinement of the one from the theoretical execution.*

$$\begin{aligned}
&\texttt{hrexec ib hrinit} \rightsquigarrow \texttt{hst} \implies \\
&\texttt{get_soutcome ctx (sexec ib sinit)} = \textit{Some}\ \texttt{(sout sis sfv)} \implies \\
&\texttt{sis_ok ctx sis} \implies \texttt{ris_refines ctx hrinit sinit} \implies \\
&\texttt{hrinit.(ris_input_init)} = \texttt{true} \implies \\
&\texttt{rst_refines true ctx hst (sexec ib sinit)}
\end{aligned}$$

Proof. We reason by induction on ib*. The final case is solved by exploiting Lemma 7.4.1. Other non-inductive instructions are proved in a similar fashion, using the fact that executing an instruction never changes the* ris_input_init *Boolean. The operation and load cases require an additional intermediate property stating that* hrs_lir_get *(from §7.3.2) and Definition 6.4.18 are equivalent when concrete arguments were mapped to* input *(i.e. to invariant values with a true* force_input *Boolean) beforehand. The* Bseq *and* Bcond *cases are solved using induction hypotheses, and by ensuring that the recursive* SE *preserves validity property 6.4.12.* □

Lemma 7.4.3 (Refined Block SE is correct on the target side [⋄]). *The target lemma is similar, but assumes a false* ris_input_init *Boolean for* hrinit*. Moreover, instead of expecting a valid theoretical* SE *outcome, the hypothesis concerns the refined execution. Of course, since we reason on the target state, the refinement relation* rst_refines *is instantiated with a false* input_init *Boolean, contrary to the previous lemma.*

$$\begin{aligned}
&\texttt{hrexec ib hrinit} \rightsquigarrow \texttt{hst} \implies \\
&\texttt{get_routcome ctx hst} = \textit{Some}\ \texttt{(rout hrs rfv)} \implies \\
&\texttt{ris_ok ctx hrs} \implies \texttt{ris_refines ctx hrinit sinit} \implies \\
&\texttt{hrinit.(ris_input_init)} = \texttt{false} \implies \\
&\texttt{rst_refines false ctx hst (sexec ib sinit)}
\end{aligned}$$

Proof. The proof follows the same scheme as the previous lemma, but requires additional "no-fail" properties for hrexec_final_sfv*, and the concrete to symbolic registers mapping. These "no-fail" lemmas are of the form of Lemma 7.2.5: they demonstrate that if an implementation's function succeeds on a state which is a valid refinement of a theoretical state, then the theory's equivalent function cannot fail. Here also, we exploit Lemma 7.4.1, and preservation lemmas about the validity relation and the* ris_input_init *Boolean.* □

7.5 SIMULATION TEST

The implementation of our simulation test exploits the hash-consed SE for invariants and blocks in order to produce the binary decision trees of both blocks to compare.

We unfold this impure test backward, starting with a detailed examination of the main simulation test for a specific *pair of blocks* in §7.5.1. Subsequently, we describe the underlying comparison for final symbolic states in §7.5.2. The correctness proof of our implementation, which implies the simulation property of our theory, is briefly sketched in §7.5.3. Finally, §7.5.4 explains how we elevate this simulation test to validate the source and target functions.

7.5.1 *Instantiating the Framework for a Pair of Blocks*

The preliminary step of the Coq implementation is to instantiate hash-consing functions and hypotheses (cf. §7.2.4.2). These are then set as global parameters for the *current* simulation instance. Hence, each pair of blocks to simulate starts with *three empty hash tables* for the three kinds of symbolic types (values, lists of values, and memories).

Our implementation also requires a refined notion of an empty symbolic internal state, as given by Definition 7.2.1:

Definition 7.5.1 (Empty refined symbolic internal state [◇])**.**

$$\begin{aligned}\varepsilon_{\text{ref}} \triangleq \{\, &\texttt{ris\_smem} = \texttt{fSinit};\ \texttt{ris\_input\_init} = \texttt{true};\ \texttt{si\_mode} = \texttt{false};\\ &\texttt{ok\_rsval} = [];\ \texttt{ris\_sreg} = \texttt{PTree.empty sval}\,\}\end{aligned}$$

Let us name $\texttt{ib}_s$ and $\texttt{ib}_t$ the source and target BTL blocks, respectively. Given these two blocks, the rewriting policy R, the current program counter pc, and the gluemap gm provided by the oracle, we define the test as follows:

Definition 7.5.2 (Main refined simulation test [◇])**.** First, we compute the application of both gluing and history invariants on the empty refined state as intermediate results:

$$\begin{aligned}&\texttt{DO}\ \texttt{hrs}_H \leftsquigarrow \texttt{tr\_hrs\_single}\ \varepsilon_{\text{ref}}\ (\texttt{gm pc}).(\texttt{history});;\\ &\texttt{DO}\ \texttt{hrs}_{HG} \leftsquigarrow \texttt{tr\_hrs\_single}\ \texttt{hrs}_H\ (\texttt{gm pc}).(\texttt{glue});;\end{aligned}$$

Second, we manipulate these two states to construct the source and target initial states $\texttt{Ihrs}_s$ and $\texttt{Ihrs}_t$, respectively:

$$\begin{aligned}\texttt{Ihrs}_s &= \{\, \texttt{hrs}_{HG}\ \textit{with}\ \texttt{sreg} = \texttt{hrs}_H.(\texttt{ris\_sreg})\,\}\\ \texttt{Ihrs}_t &= \{\, \texttt{hrs}_{HG}\ \textit{with}\ \texttt{ris\_input\_init} = \texttt{false}\,\}\end{aligned}$$

As intended, those two states correspond the theoretical ones of Definitions 6.5.7 & 6.5.8. Finally, we simply reproduce the simulation diagram from Figure 6.3:

$$\begin{aligned}&\texttt{simu\_check\_single R}\ \texttt{ib}_s\ \texttt{ib}_t\ \texttt{pc gm} : \texttt{??unit} \triangleq\\ &\quad\texttt{DO}\ \texttt{hst}_s \leftsquigarrow \texttt{hrexec}\ \texttt{ib}_s\ \texttt{Ihrs}_s;;\ \texttt{DO}\ \texttt{hst}_t \leftsquigarrow \texttt{hrexec}\ \texttt{ib}_t\ \texttt{Ihrs}_t;;\\ &\quad\texttt{DO}\ \texttt{hst}_H \leftsquigarrow \texttt{tr\_hstate}\ (\lambda\texttt{pc}.\,(\texttt{gm pc}).\texttt{history})\ \texttt{hst}_s;;\\ &\quad\texttt{DO}\ \texttt{hst}_G \leftsquigarrow \texttt{tr\_hstate}\ (\lambda\texttt{pc}.\,(\texttt{gm pc}).\texttt{glue})\ \texttt{hst}_s;;\\ &\quad\texttt{DO}\ \_ \leftsquigarrow \texttt{rst\_simu\_check true}\ \texttt{hst}_H\ \texttt{hst}_s;;\ \texttt{rst\_simu\_check false}\ \texttt{hst}_G\ \texttt{hst}_t.\end{aligned}$$

This definition makes the link with the theory. Initial states $\texttt{Ihrs}_s$ and $\texttt{Ihrs}_t$ refine $\texttt{Isis}_s$ and $\texttt{Isis}_t$, respectively. States $\texttt{hst}_s$ and $\texttt{hst}_t$ correspond to $\texttt{ss}_s$ and $\texttt{ss}_t$, respectively. States $\texttt{hst}_H$ and $\texttt{hst}_G$ correspond to $\texttt{ss}_H$ and $\texttt{ss}_G$, respectively.

In line with Figure 6.3, two properties require verification, namely $\texttt{hst}_H \preceq_s \texttt{hst}_s$ for source mode and $\texttt{hst}_G \succeq_t \texttt{hst}_t$ for target mode. They are checked by the two calls to function rst_simu_check, which recursively compares symbolic binary decision trees.

7.5.2 *Efficient Comparison of Refined Symbolic States*

The comparison is defined as a fixed point that recurses over both symbolic states. It expects both trees to have exactly *the same structure*, and their branches and leaves should be structurally equal. This requirement on the structure of symbolic states forbids transformations that would change the control flow: both states must be composed of the same execution paths. Thanks to hash-consing, we know that structurally equal symbolic terms will always share a common hash table cell, and be bound to the same pointer. Hence, structural comparison of symbolic terms is reduced to constant-time pointer equality.

Definition 7.5.3 (Recursive comparison of refined symbolic states)**.**

```
Fixpoint rst_simu_check (src_mode: bool) (hst1 hst2: rstate) :=
  match hst1, hst2 with
  | Rfinal hrs1 sfv1, Rfinal hrs2 sfv2 ⇒
      hrs_simu_check src_mode hrs1 hrs2;;
      sfval_simu_check sfv1 sfv2
  | Rcond cond1 lsv1 rsL1 rsR1, Rcond cond2 lsv2 rsL2 rsR2 ⇒
      struct_eq cond1 cond2;;
      lsv1 == lsv2;;
      rst_simu_check src_mode rsL1 rsL2;;
      rst_simu_check src_mode rsR1 rsR2
  | _, _ ⇒ FAILWITH "rst_simu_check: simu_check failed"
  end
```

If the structures of $\mathtt{hst}_1$ and $\mathtt{hst}_2$ differ, the validation fails. When both states are `Rcond` nodes, their conditions and arguments must be syntactically (i.e. structurally) equal. Conditions are compared using a real structural equality (since they are not hash-consed), while lists of symbolic values are compared by pointer equality. Finally, we compare leaves in two steps. First, we ensure the modulo invariants relation for internal states with Definition 7.5.4 below. Second, we check their final symbolic value syntactically with function `sfval_simu_check` [◇]. The latter implements syntactical property $\mathtt{sfv\_simu}$ from §6.5.3 and uses pointer equality to compare final values' symbolic arguments[9].

For refined internal states, our test distinguishes the source and target modes with the $\mathtt{src\_mode}$ Boolean:

Definition 7.5.4 (Matching refined symbolic internal states in both modes [◇])**.**

$$
\begin{aligned}
&\mathtt{hrs\_simu\_check}\ (\mathtt{src\_mode} : \mathtt{bool})\ (\mathtt{hrs}_1\ \mathtt{hrs}_2 : \mathtt{ristate}) : \mathtt{??unit} \triangleq \\
&\quad \mathtt{hrs}_1.(\mathtt{ris\_smem}) == \mathtt{hrs}_2.(\mathtt{ris\_smem});; \\
&\quad (\textbf{if}\ \mathtt{src\_mode}\ \textbf{then}\ \mathtt{hrs}_1.(\mathtt{ris\_input\_init}) = \mathtt{false} \\
&\quad \textbf{else}\ (\mathtt{hrs}_1.(\mathtt{ris\_input\_init}) = \mathtt{hrs}_2.(\mathtt{ris\_input\_init}) = \mathtt{false};; \\
&\qquad \mathtt{hrs}_2.(\mathtt{ok\_rsval}) \sqsubseteq \mathtt{hrs}_1.(\mathtt{ok\_rsval})));; \\
&\quad \mathtt{hrs}_1.(\mathtt{ris\_sreg}) \sqsubseteq \mathtt{hrs}_2.(\mathtt{ris\_sreg})
\end{aligned}
$$

This pseudocode gives an easy-to-read overview of the comparison: in any case, both symbolic memories must be structurally (using pointer equality) equal, and the register set from the first state must be included in the second one. Implicitly, this second constraint encodes the fact that $\mathtt{hrs}_2$ *might define new variables* that were dead in $\mathtt{hrs}_1$. Note that here, we expect $\mathtt{hrs}_1$ to be free of variables dead in $\mathtt{hrs}_2$ (such dead variables are removed by the $\mathtt{tr\_hstate}$ transfer function).

The rest of the test depends on the $\mathtt{src\_mode}$ Boolean. If true, we only ensure that the source state was switched to target mode (i.e. with $\mathtt{ris\_input\_init} = \mathtt{false}$). If false, the constraint is *stronger*: both states must have been switched to target mode, and the list of trapping operations from the target state must be included in that of the source state. The latter inclusion prevents the transformed code from having *more traps* than the source one.

7.5.3 *Proof of Correctness w.r.t. the Theory*

We want to prove that when our implemented simulation test from Definition 7.5.2 succeeds, then it implies the theoretical simulation property from Definition 6.5.14.

Theorem 7.5.1 (The refined simulation test is correct [◇])**.** *Given the same variables* $\mathtt{ctx}$, $\mathtt{gm}$, $\mathtt{ib}_s$, $\mathtt{ib}_t$, *and* $\mathtt{pc}$, *as in Definition 6.5.14, and under any simulation context* (*using Definition 6.4.4*), *we pose:*

$$
\begin{aligned}
&\mathtt{simu\_check\_single}\ \mathtt{R}\ \mathtt{ib}_s\ \mathtt{ib}_t\ \mathtt{gm}\ \mathtt{pc} \rightsquigarrow \_ \implies \\
&\mathtt{instantiate\_context}\ \mathtt{match\_sexec\_si\_ref}\ \mathtt{gm}\ \mathtt{ib}_s\ \mathtt{ib}_t\ \mathtt{pc}
\end{aligned}
$$

[9] For the sake of conciseness, we do not provide its definition here as it is essentially a straightforward syntactic check.

I do not detail the proof here, as it basically consists in decomposing `simu_check_single` *and linking it with the theory. Roughly, we compose other correctness proofs of this chapter to show that our implementation correctly refines the theory. Notably, we exploit Lemmas 7.4.2 and 7.4.3 for both the source and target SEs.*

7.5.4 *Validating an Entire Target Function*

After each application of Definition 7.5.2, we leave the IMPURE monad using the exit operator for Boolean computations. This allows us to lift the simulation test inside the result monad of Coq: if "`has_returned (simu_check_single R ib_s ib_t gm pc)`" is true, we safely return the "OK" result; otherwise, an error is raised [◇]. The lifted, pure version of our simulation test is named "`simu_check`". It takes the same arguments as its impure equivalent.

To guarantee the `match_sexec_ok` property of Definition 6.1.1, we encapsulate `simu_check` into a function `check_symbolic_simu` that invokes it over all pairs of blocks.

Definition 7.5.5 (Transfer function of a BTL pass)**.** After translating a RTL function to BTL, the whole Coq pass is implemented as follows:

```
Definition transf_function (f: BTL.function) :=
  let (tcfi, gm) := btl_optim_oracle f in
  let (tc, fi) := tcfi in
  let tf := BTL.mkfunction (fn_sig f) (fn_params f) (fn_stacksize f) tc
    (fn_entrypoint f) gm fi in
  do _ ← check_only_liveness tf;
  do _ ← check_symbolic_simu f tf;
  OK tf
```

The source function `f` is first given to the oracle (which is a parameter of the pass, cf. §6.1). It yields three elements: the new CFG `tc` (stands for Transformed Code), the new shadow field of the function `fi` (used only to propagate information between untrusted oracles), and the gluemap `gm` (containing invariants). A new function `tf` is built from these three elements; but the other fields stay the same. Two checks are then performed in the result monad to satisfy matching Relation 6.1.1 between BTL functions. First, that both GIs & HIs contain only pure liveness affectations, to ensure conditions `trivial_glueinv_entrypoint` and `trivial_histinv_entrypoint`. Second, that our simulation test succeeds for all pairs of blocks of `f` and `tf` to guarantee `match_sexec_ok`. The latter check retrieves the list of nodes of the source function—i.e. in `f.(fn_code)`—as a list of "`pc`" (program counters that identify a block in the CFG). It then checks, for all `pc`, that both `f.(fn_code)!pc` and `tf.(fn_code)!pc` are defined, aborting otherwise. If this is the case, it invokes the simulation test for the current pair of blocks (by calling the pure `simu_check` function mentioned above). This process continues until either a simulation failure, or after having compared all block pairs.

Both `check_only_liveness` and `check_symbolic_simu` above are proved by induction over the list of block pairs. The former is demonstrated using the correctness lemma of the only liveness checker, which implies Definition 6.3.12; and the latter by applying Theorem 7.5.1.

7.6 APPLICATIONS OF THE REWRITING ENGINE

Our term rewriting system is applied on both sides of the simulation using the same set `R` of normalization rules. In the current implementation, we define two possible sets of rules (in addition to the empty set, where nothing is rewritten): one for the expansion oracle, and another for the lazy code transformations (LCT) oracle. The expansion oracle is a BTL port of my preliminary contribution to RTLpath (cf. §4.5.2).

Recall from §4.5.2.2 that our rules are designed to always transform symbolic values to their *normal forms*. Final symbolic states are therefore always built canonically, enabling their syntactic comparison in the simulation test. Technically, to indicate the set of rules to rewrite, the SE framework is configured with an element of the following sum type [◇]: $R \triangleq$ `RRexpansions | RRlct | RRnone`.

In the following, §7.6.1 presents the expansions for operations and branches. The affine normalization mechanism used for the LCT is formalized in §7.6.2.

7.6.1 *Rules for the Expansions of Operations and Branches*

When the expansion oracle is enabled, macro-instructions of the source block are replaced with their assembly level equivalent in the target block. As in RTLpath, we aim to take advantage of these earlier expansions to eliminate their eventual redundancies (as in Example 4.5.1) and schedule them intelligently. The principle is exactly the same: normalization rules are applied on the source block's SE. In practice, rewrites are performed each time a non-expanded operation or branch is encountered during SE.

I will not go into detail into the specifics of these previous Asm-level expansions, but the interested reader can find online the exhaustive lists for operations [◇] and branches [◇].

Furthermore, the expansion mechanism is also a way for us to delay the (early) CompCert's instruction selection at the BTL level. Indeed, some middle-end optimizations are more efficient or easier to program if we keep a high-level representation of instructions. For instance, a CSE pass may be unable to eliminate certain redundancies after selection, when it could have done so before.

LIMITATIONS & WORKAROUND FOR UNEXPECTED VALUES With this expansion mechanism—whether we consider its preliminary version on RTLpath or this port to BTL—we moved *most of the assembly expansions* expressed during the Mach to Asm translation to the RTL (here BTL) level. This required us to overcome a little issue: while the forward simulation proof of Mach to Asm supports that expansions replace the `Vundef` value by any other value, this is not supported in the proof of our rewriting rules. To accept this type of rewriting, we would need to specialize the correction property of the normalization according to the side where it is applied (source side or target side). This would be more expressive, but would likely significantly increase the size of the simulation test proof.

In CompCert, `Vundef` represents a potential undefined behavior that has not yet been observed (cf. §3.3.1). In our case, `Vundef` may appear when evaluating some RISC-V macro-operations on unexpected immediate arguments (e.g. the `shrximm` macro-operation used to model division by a power of two, expanded into a sequence of right shifts and additions). The Mach to Asm expansion typically replaces `Vundef` by a silent arithmetic overflow. Our simple workaround is to introduce within normalization rules some dedicated pseudo-instructions able to generate the necessary `Vundef` (hence, acting like defensive tests): these extra pseudo-instructions are further removed in the Mach to Asm pass. Note that these extra pseudo-instructions do not disturb the prepass scheduling because they are assigned 0 latency and 0 resource. On complex ISAs, such as AArch64, many expansions of Mach to Asm pass cannot be expressed at RTL level. This is due to limitations of RTL, which does not allow instructions modifying several pseudo-registers in parallel, such as instructions with side effects on flags (recall §3.4.2). Even on RISC-V, expansions that involve stack-accessing instructions cannot really be expressed at RTL level, still because of the IR's limitations.

7.6.2 *Fold Right and Affine Forms*

Example 5.1.1 illustrated the different (but equivalent) terms produced by a strength-reduction transformation, due to the order of calculations. We focus on the SR of multiplications by a `long` constant only (see §10.5.2); typically, the verifier will have to (automatically) prove equations of the form "$(i + c) \times f = (i \times f) + c'$", where f is the multiplication factor, c the constant added at each loop iteration, and with "$c' = f \times c$". To do this, our idea is to normalize both terms in a *canonical affine form* "$c' + (f \times i)$". Nonetheless, such a representation is not generic enough to support every kind of multiplication SR, considering that calculations may be nested and involve additions. For instance, one would need to normalize a decomposed multiplication $(i \times f_1 + c_1) + (i \times f_2 + c_2)$ into $(c_1 + c_2) + (f_1 \times i) + (f_2 \times i)$. The latter can be viewed, intuitively, as a fold right operation: we want to sum a list of terms (e.g. multiplications) to produce the final result.

The high-level Coq formalization of affine forms is available here [◇]; and its lower level specification for the RISC-V architecture here [◇]. For simplicity, this section presents our affine normalization engine from a high-level point of view.

$$c \cdot (v_1 + v_2) = (c \cdot v_1) + (c \cdot v_2) \qquad c_1 \cdot (c_2 \cdot v) = (c_1 c_2) \cdot v \qquad c_1 \cdot c_2 = c_1 c_2$$
$$v_1 + v_2 = v_2 + v_1 \qquad (v_1 + v_2) + v_3 = v_1 + (v_2 + v_3) \qquad 0 + (v_1 + v_2) = v_1 + v_2$$
$$0 + (c \cdot v) = c \cdot v \qquad (c_1 \cdot v) + (c_2 \cdot v) = (c_1 + c_2) \cdot v \qquad (c \cdot v) + v = (c + 1) \cdot v \qquad v + v = 2 \cdot v$$

Figure 7.2: Affine Arithmetic of CompCert 64-bit Integer Operators on Values.

$$0 + v = v \qquad 0 \cdot v = 0 \qquad 1 \cdot v = v \qquad (v + v) - v = v + (v - v)$$

Figure 7.3: Examples of Invalid Equalities for CompCert 64-bit Integer Operators.

$$\pi ::= v \mid c \cdot v \qquad \phi ::= v \mid c + \Sigma_{i=1}^{n} \pi_i \quad \text{where } n \geqslant 0 \text{ and } (\pi_i)_{i \geqslant 1} \text{ is strictly increasing}$$

Figure 7.4: Representation of Our Affine Forms.

7.6.2.1 *Equational Theory*

Our SR is validated in the variant of affine arithmetic given in Figure 7.2, where c represents a 64-bit integer constant and v is a CompCert value. Actually, we consider this theory extended with specific operators such as $v \ll c = 2^c \cdot v$. However, note that some usual equations—such as those given in Figure 7.3—do not hold. For example, if one of their argument is not a long integer or not a pointer—e.g. a float—64-bit integer operations return the absorbing `Vundef` value. Moreover, operation "+" also performs pointer arithmetic in the abstract CompCert model of pointers (and our SR leverages this opportunity). In this model, on a 64-bit architecture, if v is a pointer, then $v + v' \neq$ `Vundef` if and only if v' is a 64-bit integer (in this case, v' is seen as a relative offset w.r.t. v). And $c \cdot v =$ `Vundef` if v is a pointer. This explains why we never identify v and $1 \cdot v$. But, if v is a pointer, then $v + v =$ `Vundef` and we still have $v + v = 2 \cdot v$. Last, if v and v' are two pointers in the same block, then $v - v'$ determines their relative offset. Thus, if v is a pointer, then $v + (v - v) = v$ but $(v + v) - v =$ `Vundef` [10].

7.6.2.2 *Normal Forms*

Since our symbolic values are evaluated to CompCert values (for a given block execution context), Figure 7.2 also induces semantic equalities about symbolic values. As noticed in §2.2.3.2, normalized rewriting (when applicable) reduces such semantic equalities to structural equalities. Our representation of normal (i.e. canonical) forms is given in Figure 7.4, where v represents now a *variable* (representing itself a symbolic value). Due to the commutativity of "+", normal forms depend on a total order over variables. Let us assume such an order. Because $1 \cdot v$ may not be v, we introduce a notion of *pseudo-product*, written π (see Figure 7.4). We then lift the *total order* over variables into a *total preorder* over pseudo-products. At that point, we define a normal form, written ϕ, as either a variable v or as the sum of a scalar c (possibly null) with a strictly increasing sequence (possibly empty) of pseudo-products (see Figure 7.4). Last, the affine normalization is mainly reduced to two operations "$c \cdot \phi$" and "$\phi_1 + \phi_2$" preserving normal forms by applying Figure 7.2 equations.

The `list_sval` type is represented into square brackets here for the sake of simplicity.

To facilitate the proof of recursive operations over affine forms and future extensions, we encode them in symbolic values (Definition 6.2.1) with the "`Sfoldr`$(op, \mathtt{l}sv, sv_0, hid)$" dedicated constructor, which is semantically equivalent to "`Sop`$(op, [sv_0, \mathtt{Sop}(op, [sv_1, \ldots])], hid)$". This enables us to represent affine forms written "$c + \Sigma\vec{\pi}$" in Figure 7.4 as a fold right "`Sfoldr`$(+, \vec{\pi}, c, hid)$" within our implementation.

[10] This lack of associativity of operator "−" w.r.t. the additive operator (for pointers) explains why it is not yet supported in our SR.

7.6.2.3 *Normalization Process and Correctness*

Our affine normalization needs to integrate affine forms with symbolic values that do not represent pure affine computations. In particular, affine variables in Fig. 7.4 actually reify symbolic values whose root is not an affine computation (i.e. neither a 64-bit immediate, nor a "$\cdot$", nor a "$+$"). In practice, we do not introduce explicit affine variables, but rather use the total order of Definition 7.2.8 on reified symbolic values.

In order to normalize an affine operation "$c \cdot sv$" or "$sv_1 + sv_2$", we first define a function $\mathcal{A}$ [$\diamond$], which maps any symbolic value sv to an affine form and satisfies the below theorem.

Theorem 7.6.1 (Affine form construction properties)**.** *For any block execution context* ctx*:*

$$\sigma_{sv}(ctx, c \cdot sv) = \sigma_{sv}(ctx, c \cdot \mathcal{A}(sv)) \qquad \sigma_{sv}(ctx, sv_1 + sv_2) = \sigma_{sv}(ctx, \mathcal{A}(sv_1) + \mathcal{A}(sv_2))$$

In other words, within the context of an affine operation, the normal forms returned by $\mathcal{A}$ preserve the semantics. Because of the invalid equations in Figure 7.3, it would be too strong to simply require "$\sigma_{sv}(ctx, \mathcal{A}(sv)) = \sigma_{sv}(ctx, sv)$". Those properties are easily proved using auxiliary lemmas about the semantics of CompCert values (defined in §3.3).

In practice, the normalization is applied after each assignment, so $\mathcal{A}$ only needs to perform a simple case analysis on the root of its argument:

Definition 7.6.1 (Affine form construction)**.**

$$\mathcal{A}(sv) \triangleq \begin{cases} sv & \text{if } sv \text{ matches } \texttt{Sfoldr}(+, \_, \_, \_) \\ \texttt{fSfoldr}(+, [\,], c) & \text{if } sv \text{ matches } \texttt{Sop}(c, [\,], \_) \text{ where } c \text{ is a 64-bit integer} \\ \texttt{fSfoldr}(+, [sv], 0) & \text{otherwise} \end{cases}$$

Using classical Coq definitions, pattern-matching with wildcards on sum-types featuring many constructors often generates many identical goals; to circumvent this issue, a solution is to rely on the `Function` *keyword.*

Then, the normalization of "$c \cdot \mathcal{A}(sv)$" (respectively "$\mathcal{A}(sv_1) + \mathcal{A}(sv_2)$") reduces to a computation of the form "$c \cdot \texttt{fSfoldr}(+, \vec{\pi}, c_0, hc)$" (respectively "$\texttt{Sfoldr}(+, \vec{\pi_1}, c_1, hc_1) + \texttt{Sfoldr}(+, \vec{\pi_2}, c_2, hc_2)$").

Observe that Definition 7.6.1 and Theorem 7.6.1 are not architecture-independent: they must be overridden for each backend to specify the underlying additive and multiplicative RTL operations. In this thesis, the work was done on the RISC-V 64-bit architecture. Nevertheless, operations on affine forms (i.e. scaling with a constant & adding and merging to affine terms) are defined once and for all by our theory thanks to a template module parametrized by the additive and multiplicative operations and their related properties. Therefore, porting the SR normalization rules to another backend should require minimal amount of work.

SCALE OPERATION The computation of "$c \cdot \texttt{Sfoldr}(+, \vec{\pi}, c_0, hc)$" returns "$\texttt{fSfoldr}(+, c \cdot \vec{\pi}, c\ c_0)$" where "$c \cdot \vec{\pi}$" is an instance of a "list-map" operation over pseudo-product list $\vec{\pi}$ (and is verified by applying the three equalities at the top line in Figure 7.2).

ADD & MERGE OPERATIONS The computation of "$\texttt{Sfoldr}(+, \vec{\pi_1}, c_1, hc_1) + \texttt{Sfoldr}(+, \vec{\pi_2}, c_2, hc_2)$" returns "$\texttt{fSfoldr}(+, \vec{\pi_1} + \vec{\pi_2}, c_1 + c_2)$", where "$\vec{\pi_1} + \vec{\pi_2}$" is very similar to the merging of sorted lists $\vec{\pi_1}$ and $\vec{\pi_2}$ for the pseudo-product preorder, except that when two compared pseudo-products are equivalent for the preorder, they are themselves merged by an operation described just below.

Definition 7.6.2 (Testing the equivalence induced by the preorder over pseudo-products)**.**

$$equiv(\pi_1, \pi_2) \triangleq \begin{cases} sv_1 == sv_2 & \text{if } (\pi_1, \pi_2) \text{ matches } (c_1 \cdot sv_1, c_2 \cdot sv_2) \\ sv == sv' & \text{else if } (\pi_1, \pi_2) \text{ matches } (sv, c \cdot sv') \text{ or } (c \cdot sv, sv') \\ \pi_1 == \pi_2 & \text{otherwise} \end{cases}$$

The equivalence test on pseudo-products uses pointer equality "==" to validate that two pseudo-products can be merged.

In Figure 7.2, the three equations on the bottom line (from left to right) correspond to the three cases of $equiv$ (from top to bottom). Each of these cases in $equiv$ is thus associated with a normalization rule that merges the pseudo-products by applying the corresponding equation from left to right. For instance, supposing we have $equiv(\pi_1, \pi_2) = \texttt{true}$ for the third case, then the merge of these two pseudo-products is $2 \cdot \pi_1$, because then we know that $\pi_1 = \pi_2$ and that π_1 is a "reified" symbolic value (i.e. not having an affine computation at its root).

8

BILATERAL RTL-BTL TRANSLATION†

This chapter presents our translation from RTL to BTL and back. In particular, it details the design of a defensive translation validator able to verify translations in both directions. These translations perform some structural transformations involving code duplications/factorizations (according to the direction) and insertion of so-called synthetic nodes. In Section 8.1, we briefly introduce the idea of morphisms as a generic abstraction to prove such duplications, factorizations, or translations between RTL & BTL control-flow graphs. Applying this principle, Section 8.2 describes two oracles that perform the translations between RTL & BTL in both directions. These oracles produce a morphism as a certificate to guide the projection checker of Section 8.3, which then validates their result.

8.1 SETTING: THE NOTION OF CFG MORPHISMS

I explained in §4.1.4.2 that to increase the number of scheduling opportunities, it was of great interest to perform code duplications to widen the size of the optimization window. As detailed in [65, §2.1]† (and previously in [135, §4.4]† & [133]), we specialized the notion of *graph homomorphism* into *CFG morphisms* between two RTL CFGs. We proposed a translation validation of CFG restructurings, given a mapping (i.e. an FVDP certificate provided by the oracle) between nodes of both CFGs. For instance, duplications like those of Figure 4.1 have a CFG morphism corresponding to the mapping that forgets numeric indices on node contents [133, Figure 6.1]. This mapping, guiding the validation mechanism, is expected to maintain the integrity of instruction contents, successor relationships, successor order, and the CFG entry point. These criteria are checked syntactically by comparing instructions obtained through the mapping across both CFGs. Proving such a validator is almost trivial: two CFGs related by a CFG morphism are semantically equivalent and, consequently, bisimulable.

This morphism checker was implemented by Cyril Six on RTL to validate the various kinds of code duplications depicted in Figure 4.1; and is still used in the current version of Chamois-CompCert[1]. Yet, when conceiving BTL, we noticed that the reverse transformations—i.e. code factorizations—can also be verified using the same mechanism (thanks to the reverse simulation). See §8.2.2.

Moreover, loops are not the only interesting code fragments to duplicate. In particular, code motion (CM) algorithms on BTL can greatly benefit from the insertion of empty blocks—called synthetic nodes—due the event-based correctness model of CompCert (I justify this in Example 10.2.1).

Therefore, we generalized Cyril Six's morphism checker to validate a bisimulation between RTL and BTL (in both directions), synthetic nodes' insertion, and a CFG factorization pass.

8.2 TRANSLATION ORACLES

Each translation direction is performed by a dedicated oracle.

8.2.1 *Block Selection Oracle, From RTL to BTL*

When partitioning the RTL control-flow graph to translate it to BTL, we can arbitrarily choose how to select blocks (as long as they respect BTL's structure of loop-free blocks).
In Chamois-CompCert, we implemented three different block selections:

I do not detail these block selection processes, but their code is available here [◇].

STRAIGHTFORWARD BASIC BLOCKS: simply cut the code into basic blocks, by running forward through the RTL program and instantiating a new block each time a final instruction (of type *final*) or a branch is encountered.

[1] We have not ported Six's duplications to BTL, as applying them on RTL is quite flexible in practice (there is no need to translate to another language, and they can be applied between two existing RTL passes), but doing this port would be very simple using the projection checker of this chapter.

BASIC BLOCKS WITH SYNTHETIC NODES: same principle, but this time the oracle inserts a synthetic node on every edge leading to a join point in the RTL CFG. Synthetic nodes are empty blocks that only contain a `Bgoto` instruction pointing to the detected join point. They are necessary for our LCT algorithm, see §10.2.1.1.

SUPERBLOCKS: this last option was ported from [133, §5.3.1]. In contrast to basic block selectors, here we allow predicted branches to appear in superblocks.

Hence, the Coq pass is parametrized by one of the three oracles mentioned above. In addition to the block selection, the translation oracle also applies a post-order renumbering of the BTL code[2] (see its code online [◇]), and builds the morphism mapping needed to drive the validation.

All those three untrusted oracles share the same Coq signature:

```
Parameter rtl2btl: f_rtl → cfg_btl * pc * finfo * (pc ↦ pc option)
```

Given a RTL function, the oracle yields a tuple of the BTL translated CFG, its entry point of type *pc*, a fresh shadow field for function's information, and the morphism as a map from BTL to RTL nodes.

Generating the mapping for the RTL to BTL translation is trivial: we *only* insert identity mappings. When using the synthetic nodes generation, we simply leave synthetic nodes undefined in the mapping, since they do not have a RTL equivalent (see §8.3).

8.2.2 *Flattening and Factorization, From BTL to RTL*

For the resulting RTL code to be as close as possible to the original one, we start by renumbering the BTL code using the old indices that were stored in shadow fields during the initial selection. This is a preserving approach; however, if many changes were made to the BTL CFG, the regenerated RTL structure will be different. Translating a BTL program to RTL then simply involves flattening blocks to rebuild a single instruction CFG. After the translation, we optionally apply a CFG minimization pass which mimics Moore's algorithm for deterministic finite automaton (DFA) minimization.

Both the flattening and the optional factorization are performed by the same oracle, which has the following type:

```
Parameter btl2rtl: f_btl → cfg_rtl * pc * (pc ↦ pc option)
```

The result is a tuple of the new RTL CFG, entry point, and mapping. Without factorization, the CFG morphism trivially maps *BTL block IDs to their first corresponding instruction in the RTL graph*.

The code factorization pass was designed and implemented by Alexandre Bérard [4, §4]: an example is given in §10.2.7, and a more complete description is provided in §11.4.

8.3 BILATERAL MATCHING: THE BTL PROJECTION CHECKER

We use a variant of the duplications' morphism checker [135, §4.4][†]—called the *BTL projection checker*, to validate both directions of translation between RTL and BTL.

Below, we outline the relational specification of this validator in a backward manner in §8.3.1, before proving it correct for both directions in Sections 8.3.2 and 8.3.3.

8.3.1 *Specification of Our Validator*

At the top level, we expect a matching relation between a BTL function `fB` and a RTL one `fR`:

The morphism `m` is implemented using the `PTree` library.

Definition 8.3.1 (Matching relation between BTL and RTL at the function level)**.** Given the mapping `m` (provided by a translation oracle as a certificate), and a Boolean "`indirect`" that enables the support of synthetic nodes (i.e. if false, the validator will reject new empty blocks in the BTL function)[3], we set:

[2] This renumbering has two purposes: first, it accelerates work-list based fixed points in optimizations (see Chapter 10); second, it eases the translation back to RTL by remembering the old indices in the instructions' shadow fields.

[3] We enable this Boolean only for the RTL to BTL direction, and only when the chosen oracle is the "basic blocks with synthetic nodes" one.

```
Record match_function (m: pc ↦ pc option) (fB: f_btl) (fR: f_rtl) (indirect: bool): Prop := {
  map_correct: match_cfg m (BTL.fn_code fB) (RTL.fn_code fR) indirect;
  map_entrypoint: match_nodes m (BTL.fn_code fB) indirect
                    (BTL.fn_entrypoint fB) (RTL.fn_entrypoint fR);
  preserv_fnsig: BTL.fn_sig fB = RTL.fn_sig fR;
  preserv_fnparams: BTL.fn_params fB = RTL.fn_params fR;
  preserv_fnstacksize: BTL.fn_stacksize fB = RTL.fn_stacksize fR
}
```

The function's signature, parameters, and stack size are expected to be preserved. Entry points and other CFG nodes from the two functions should be related using the mapping: this is ensured by properties `match_nodes` and `match_cfg` above, respectively.

Let us continue with the relation for nodes:

Definition 8.3.2 (Matching relation between nodes)**.**

```
Inductive match_synthetic_node (m: pc ↦ pc option) (cfgB: cfg_btl): bool → pc → pc → pc → Prop :=
  | mn_direct pcB pcR indirect:
      m ! pcB = Some pcR →
      match_synthetic_node m cfgB indirect pcB pcB pcR
  | mn_indirect pcX pcB pcR ib iinfo:
      m ! pcX = None →
      cfgB ! pcX = Some ib →
      ib.(entry) = BF (Bgoto pcB) iinfo →
      m ! pcB = Some pcR →
      match_synthetic_node m cfgB true pcX pcB pcR

Definition match_nodes m cfgB indirect pcX pcR :=
  ∃ pcB, match_synthetic_node m cfgB indirect pcX pcB pcR
```

We voluntarily isolate the existential binder here, so that we can instantiate the relation for a specific pcB (see Definition 8.3.10).

In other words, given two nodes `pcX` and `pcR`, two alternatives are possible: either (`mn_direct`) there is a direct mapping between the BTL and the RTL node in `m`, or (`mn_indirect`) there is no mapping from `pcX` in `m`, but `pcX` points to an intermediate—synthetic—node in `cfgB` whose immediate successor leads to a program point `pcB` mapped to the RTL node in `m`. The second clause is conditioned by the `indirect` Boolean.

Relating entry points from both functions using the above definition allows us to insert a synthetic node *before* the original RTL entry point. This is useful in case the RTL's CFG entry was also a loop header: indeed, without such an empty pre-header, it would be impossible to anticipate loop invariant code in that situation (which remain, however, quite rare).

Beside these indirections, our projection checker also needs to relate the content of all blocks with their corresponding RTL instructions. We achieve this by enforcing an inductive relation for every BTL block:

Definition 8.3.3 (Matching all BTL blocks with the right RTL sequence)**.**

```
Definition match_cfg m (cfgB: cfg_btl) (cfgR: cfg_rtl) indirect: Prop :=
    ∀ pcB pcR, m!pcB = Some pcR →
    ∃ ib, cfgB!pcB = Some ib ∧
          match_iblock m cfgB cfgR indirect true pcR ib.(entry) None
```

Meaning that for each defined node in the CFG morphism, the node must also be defined in the BTL CFG itself, and satisfies `match_iblock`:

An essential difference between RTL and BTL is that in RTL, each instruction designates at least one explicit successor in the CFG, whereas in BTL, only final instructions (of type *final*) indicate a successor in the CFG. Some BTL final instructions like the `Bgoto` have no direct RTL analogue: typically, `Bgoto` actually designates the successor (in another BTL block) of a certain RTL instruction. So, to correspond to RTL code, a `Bgoto` instruction must necessarily *be preceded* (in its block) by another BTL instruction. This preceding instruction corresponds to the "*beginning*" of a RTL instruction.

Another important concern is no-operation instructions. In Figure 5.1, the syntax for Bnop optionally defines an *iinfo* field. We actually use the presence of this shadow field to distinguish no-ops that must be kept in RTL (when the field is set) from those that we want to forget during the translation. To illustrate these differences, let us examine the RTL codes corresponding to Figures 5.2 and 5.3:

Example 8.3.1 (Correspondence between RTL and BTL syntaxes)**.** For Figure 5.2, if we assume the conditional branch to be at position pc2 in the RTL graph, the RTL pseudocode (in post-order numbering) would look like:

```
pc2: Icond(_>=_, [x;y], L, pc1)
pc1: Iop(_<<, [z], x, pc0)
pc0: Ireturn(x)
```

where L could point to any CFG node. In this example, the branch (at pc2) is a RTL Icond instruction which contains syntactically the addresses of its two possible successors. The BTL Bgoto, "Bnop None" (in Figure 5.2, the None argument was omitted for simplicity), and Bseq instructions have no RTL equivalent and are thus absent in the above translation. The sole purpose of the Bgoto in the BTL version of the code was to encode the fact that one branch exits the block. In contrast, the BTL Bnop of the "else" branch was simply there to "fill" the "else" block, since the actual "else" code in Figure 5.2 is stored in the enclosing Bseq.

This *dummy* Bnop in the "else" branch of Figure 5.2 could theoretically be removed by directly replacing it with the sequence of both the shift and the return. On the other hand, if we consider a variant of this example, the presence of the dummy Bnop becomes essential to avoid code redundancies. Imagine a situation where, instead of the Bgoto(L) of the if branch, we would have an arithmetic operation such as "z = z * 2". In that case, not using the dummy Bnop would force us to duplicate the "else" branch both in the right part of the condition and in the enclosing sequence!

Another argument, though less consequential but aesthetic, is that since BTL is heavily used to represent superblocks, combining a dummy Bnop with a goto provides a convenient way to represent side-exits as a single instruction, which is easier to manipulate in oracles.
As a simpler example, the internal join from Figure 5.3 would be translated as:

```
pc3: Icond(_==0, [i], pc1, pc2)
pc2: Iop(Omove, [b], x, pc0)
pc1: Iop(Omove, [a], x, pc0)
pc0: Ireturn(x)
```

where the only BTL instructions skipped during translation are those of type Bseq.

I now explain how to define a matching relation between both syntaxes encoding those differences.

Definition 8.3.4 (Inductive relation between a single BTL block and its corresponding RTL sequence)**.** Here "match_iblock m cfgB cfgR indirect (isfst: bool) pcR ib (opc: option pc)" means that ib matches a RTL code sequence starting at pcR (in the RTL CFG). Parameter isfst, when true, indicates that no RTL step in the surrounding block has been started before entering pcR, i.e. this encodes an "input" information. This information is used to forbid certain BTL code that are not transformable in RTL, e.g. Bgoto not preceded by the beginning of a step. Conversely, opc is an "output" information: it is equal to None when all branches of the block end on a final instruction, and to "Some pcR'" when all branches (of the block), that do not end on a final BTL instruction, join on pcR' in the RTL CFG.

```
Inductive match_iblock (m : pc ↦ pc option) (cfgB : cfg_btl) (cfgR : cfg_rtl) (indirect: bool):
bool → pc → iblock → (option pc) → Prop :=
  | mib_BF isfst fi pcR i iinfo:
      cfgR!pcR = Some i →
      match_final_inst m cfgB indirect fi i →
      match_iblock m cfgB cfgR indirect isfst pcR (BF fi iinfo) None
  | mib_nop_on_rtl isfst pcR pcR' iinfo:
      cfgR!pcR = Some (Inop pcR') →
      match_iblock m cfgB cfgR indirect isfst pcR (Bnop (Some iinfo)) (Some pcR')
  | mib_nop_skip pcR:
      match_iblock m cfgB cfgR indirect false pcR (Bnop None) (Some pcR)
  | mib_op isfst pcR pcR' op lr r iinfo:
```

```
    cfgR!pcR = Some (Iop op lr r pcR') →
    match_iblock m cfgB cfgR indirect isfst pcR (Bop op lr r iinfo) (Some pcR')
| mib_load isfst pcR pcR' trap m0 a lr r iinfo:
    cfgR!pcR = Some (Iload trap m0 a lr r pcR') →
    match_iblock m cfgB cfgR indirect isfst pcR (Bload trap m0 a lr r iinfo) (Some pcR')
| mib_store isfst pcR pcR' m0 a lr r iinfo:
    cfgR!pcR = Some (Istore m0 a lr r pcR') →
    match_iblock m cfgB cfgR indirect isfst pcR (Bstore m0 a lr r iinfo) (Some pcR')
| mib_exit pcX pcR iinfo:
 (* NB: on RTL side, we exit the block by a "basic" instruction (or Icond)
    Thus some step should have been executed before [pcX] in the RTL code. *)
    match_nodes m cfgB indirect pcX pcR →
    match_iblock m cfgB cfgR indirect false pcR (BF (Bgoto pcX) iinfo) None
| mib_seq_Some isfst b1 b2 pcR1 pcR2 opc:
    match_iblock m cfgB cfgR indirect isfst pcR1 b1 (Some pcR2) →
    match_iblock m cfgB cfgR indirect false pcR2 b2 opc →
    match_iblock m cfgB cfgR indirect isfst pcR1 (Bseq b1 b2) opc
| mib_cond isfst c lr bso bnot pcso pcnot pcR opc1 opc2 opc i iinfo:
    cfgR!pcR = Some (Icond c lr pcso pcnot i) →
    match_iblock m cfgB cfgR indirect false pcso bso opc1 →
    match_iblock m cfgB cfgR indirect false pcnot bnot opc2 →
    is_join_opt opc1 opc2 opc →
    match_iblock m cfgB cfgR indirect isfst pcR (Bcond c lr bso bnot iinfo) opc
```

Above, values of `ib` that correspond to the beginning of a RTL instruction (i.e. all BTL instructions except `Bseq`, `Bgoto`, and "`Bnop None`") are syntactically compared with their RTL equivalent match at `pcR`. When `ib` is a branch (`Bcond`) or a sequence (`Bseq`), the relation is defined inductively over sub-blocks. For sequences, rule `mib_seq_Some` prohibits the presence of dead code sequentially following a final instruction. Such a restriction simplifies the formal proof and ensures that no unnecessary code is generated.

In the following, we call *real node* a BTL node which is an actual RTL node, and not a synthetic node. As sketched before, BTL gotos cannot be isolated in a block—as implied by the false value of `isfst` in rule `mib_exit`, except in synthetic nodes (which are not bound in the morphism). In fact, this is encoded by the `match_nodes` prerequisite: a goto terminating a real node can be followed by a synthetic node[4]. A synthetic node may also follow non-returning final instructions (i.e. calls, built-ins, and jump tables). Predicate `match_final_inst` [◇] compares final values[5] syntactically and checks that each exit of a final instruction leads to a real node modulo at most one *indirection* by a synthetic node (e.g. with Definition 8.3.2).
Finally, the `Bcond` case enforces a special `is_join_opt` output property for branches:

```
Inductive is_join_opt {A}: (option A) → (option A) → (option A) → Prop :=
  | ijo_None_left o: is_join_opt None o o
  | ijo_None_right o: is_join_opt o None o
  | ijo_Some x: is_join_opt (Some x) (Some x) (Some x)
```

We implemented a functional checker proved to imply the specification of Relation 8.3.1 when the result is true (this step being very simple, I do not describe it, see [◇]).

8.3.2 *BTL to RTL Proof*

Proving the translation back to RTL is the simplest: synthetic nodes in the checker are always disabled (i.e. `indirect = false`). The simulation is following a variant of "plus" scheme (cf. top-right of Figure 3.2), meaning that a single BTL blockstep is simulated by at least one instruction on the RTL side. Proofs below are presented in a backward style.
More formally, we relate RTL and BTL states as follows (assuming a successful validation to obtain `match_function` from Relation 8.3.1):

[4] The RTL to BTL oracle thus never generates an isolated goto which is not a synthetic node.
[5] Except gotos, that are already handled in `match_iblock` since not present in RTL.

Definition 8.3.5 (Mathing relation between stack frames—BTL to RTL).

```
Inductive match_stackframes: Σ_B → Σ_R → Prop :=
  | match_stackframe_intro m res fB sp pcB rs0 fR pcR
      (TRANSF: match_function m fB fR false)
      (HMAP: m!pcB = Some pcR)
      : match_stackframes (Σ_B res fB sp pcB rs0) (Σ_R res fR sp pcR rs0)
```

States for RTL and BTL below are denoted as in Definition 5.3.1.

Definition 8.3.6 (Matching relation between states—BTL to RTL).

```
Inductive match_states: BTL.S → RTL.S → Prop :=
  | match_states_intro m Σ_B Σ_R fB fR sp pcB pcR rs0 m0
      (STACKS: list_forall2 match_stackframes Σ_B Σ_R)
      (TRANSF: match_function m fB fR false)
      (HMAP: m!pcB = Some pcR)
      : match_states (BTL.S Σ_B fB sp pcB rs0 m0) (RTL.S Σ_R fR sp pcR rs0 m0)
  | match_states_call Σ_B Σ_R fB fR args m0
      (STACKS: list_forall2 match_stackframes Σ_B Σ_R)
      (TRANSF: match_fundef fB fR)
      : match_states (BTL.C Σ_B fB args m0) (RTL.C Σ_R fR args m0)
  | match_states_return Σ_B Σ_R v m0
      (STACKS: list_forall2 match_stackframes Σ_B Σ_R)
      : match_states (BTL.R Σ_B v m0) (RTL.R Σ_R v m0)
```

The simulation proof of internal calls, external calls, and returns is trivial: it demonstrates a lock-step between the BTL blockstep and the RTL instruction; so I do not detail it here (see [⋄]). When the step is of type `exec_iblock` (cf. Definition 5.3.7), a BTL step decomposes into the execution of a sequence of non-final instructions (the body), followed by a final instruction. Two possibilities arise. (i) Either this final instruction exists in RTL, and we simulate the body's execution by a "star" step (potentially empty) of BTL instructions. The simulation of the final instruction then corresponds to *exactly one* RTL step. Or (ii) this final instruction is a goto and the matching conditions of Definition 8.3.4 ensure that the body's execution contains a BTL instruction corresponding to the beginning of a RTL instruction just before the goto. In this situation, the body execution is simulated by a "plus" step[6] of RTL instructions, and the simulation of the goto is already done at the end of this plus step. Put another way, the goto simulation represents a *stuttering* step on the RTL side.

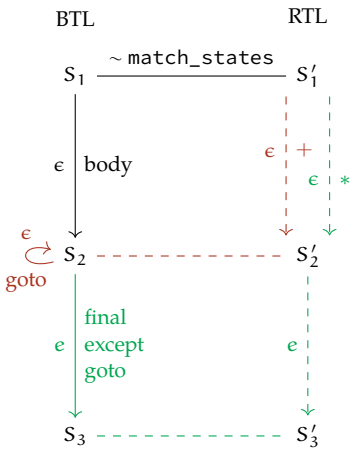


Figure 8.1: Simulation From BTL to RTL: goto case (red) and other final instructions (green); see Lemma 8.3.2.

We encode that using a variant of star RTL step (see Figure 8.1):

Definition 8.3.7 (Conditional variant of the RTL star simulation). The difference with the classical star step is that here, the stuttering case requires that proposition P holds.

```
Inductive cond_star_step (P: Prop): RTL.S → trace → RTL.S → Prop :=
  | css_refl s: P → cond_star_step P s ϵ s
  | css_plus s1 e s2: plus RTL.step tge s1 e s2 →
                      cond_star_step P s1 e s2
```

Notice that it implies the star simulation:

Lemma 8.3.1 (Conditional reflexive step or plus step imply star step [⋄]). *Let us note Definition 8.3.7 from state S_1 to state S_2 as "$S_1 \xrightarrow{C} S_2$". Star and plus steps are noted "$S_1 \xrightarrow{*} S_2$" and "$S_1 \xrightarrow{+} S_2$", respectively.*

[6] The validator here expects the CFG to be free of synthetic nodes, that is why the simulation is "plus" and not "star".

In the `css_refl` *case, for all* P, *we always have "*$S \xrightarrow{*} S$*", as no step is performed. It is hence a particular case of star simulation. Conversely, for all* S_1 *and* S_2, *"*$S_1 \xrightarrow{+} S_2 \implies S_1 \xrightarrow{*} S_2$*", meaning that a plus simulation is also a particular case of star simulation. Hence, "*$S_1 \xrightarrow{C} S_2 \implies S_1 \xrightarrow{*} S_2$*".*

Lemma 8.3.2 (BTL block plus step simulation is correct)**.** *This simulation property decomposes the body (hypothesis* `IBIS` *below) and the final instruction (hypothesis* `FS`*) of block* `ib`*.*

```
Lemma iblock_step_simulation sp m Σ_B Σ_R fB fR ib rs0 m0 rs1 m1 pcR0 fin e s2
  (STACKS: list_forall2 match_stackframes Σ_B Σ_R)
  (TRANSF: match_function m fB fR false)
  (IBIS: iblock_istep G_B sp rs0 m0 ib rs1 m1 (Some fin))
  (MIB: match_iblock m (BTL.fn_code fB) (RTL.fn_code fR) false true pcR0 ib None)
  (FS: final_step G_B Σ_B fB sp rs1 m1 fin e s2)
  : ∃ s2', plus RTL.step G_R (RTL.S Σ_R fR sp pcR0 rs0 m0) e s2' ∧ match_states s2 s2'
```

Proof. We handle the body using Lemma 8.3.3 (defined later in this section), so that we obtain a new `match_iblock` *property with the final instruction of* `ib`*, and a conditioned star step (cf. Definition 8.3.7) for the execution of the body. The former has an unknown value for* `isfst`*, used to define the* `P` *predicate of the latter* `cond_star_step` *as "*`true = isfst`*". Hence,* `P` *never holds for gotos and no-op that should be skipped. This leads to two cases:*

- *The final instruction is not a goto: we can apply Lemma 8.3.4 that gives us the single* `RTL` *step corresponding to the final instruction, and a matching property between* `s2` *and* `s2'`*. We thus discharge the existential quantifier with this new* `s2'`*, and prove the* `match_states` *goal. We know from a CompCert's existing theorem that our "*$S_2 \xrightarrow{+} S_2'$*" step can be decomposed into "*$\exists S_{2i}, S_2 \xrightarrow{*} S_{2i} \wedge S_{2i} \to S_2'$*" (i.e. right unfolding). The star result is proved by Lemma 8.3.1 above, and the single step goal is exactly the* `RTL` *step of the last instruction.*
- *The final instruction is a goto, so* `P` *is "*`true = false`*" and reduces to a plus simulation from the conditional star hypothesis. This plus instantiates* `s2'` *perfectly and solves the first goal. Since* `indirect` *is false, we deduce the "*`m!pcB = pcR`*" equality needed to prove the matching between* `s2` *and* `s2'`*.*

□

Lemma 8.3.3 (Conditional star step simulation of the body is correct)**.** *In case of a final instruction in* `ofin` *below, it must be matched with the corresponding* `RTL` *one located at* `pcR1`*; predicate* `P` *for* `cond_star_step` *is then defined as the equality between* `isfst` *(at the block entry) and* `isfst'` *(at the last instruction). This predicate might be true e.g. if the only instruction in the block is a non-goto but final instruction. Otherwise, when "*`ofin = None`*", we just impose (in addition to the conditional star step) that* `opc` *from the relation in assumption equals "*`Some pcR1`*".*

```
Lemma iblock_istep_simulation sp m Σ_R fB fR rs0 m0 ib rs1 m1 ofin
  (IBIS: iblock_istep G_B sp rs0 m0 ib rs1 m1 ofin): ∀pcR0 opc isfst
  (MIB: match_iblock m (BTL.fn_code fB) (RTL.fn_code fR) false isfst pcR0 ib opc),
   match ofin with
   | None ⇒ ∃ pcR1,(opc = Some pcR1) ∧
       cond_star_step (isfst = false) (RTL.S Σ_R fR sp pcR0 rs0 m0) ϵ
                                      (RTL.S Σ_R fR sp pcR1 rs1 m1)
   | Some fin ⇒ ∃ isfst' pcR1 iinfo,
       match_iblock m (BTL.fn_code fB) (RTL.fn_code fR) false isfst' pcR1 (BF fin iinfo) None ∧
       cond_star_step (isfst = isfst') (RTL.S Σ_R fR sp pcR0 rs0 m0) ϵ
                                       (RTL.S Σ_R fR sp pcR1 rs1 m1)
   end
```

Proof. We proceed by induction on `IBIS`*, and obtain eight subcases (cases for* `Bop`*,* `Bload`*, and* `Bstore` *are merged below):*

- *"*`ib = BF (fin iinfo)`*". For both the* `mib_BF` *and* `mib_exit` *cases,* `opc` *is equal to* `None`*. As there is a final instruction, we must prove the matching relation for* `pcR1`*, which is exactly* `MIB`*. The predicate* `P` *is thus "*`isfst = isfst`*", so we apply constructor* `css_refl` *and prove* `P` *by reflexivity.*

- *"`ib = Bnop oiinfo`". We invert hypothesis `MIB` to distinguish the `Some` and the `None` cases. In the former, `pcR1` is instantiated with the successor of the `RTL` corresponding no-op, and by applying the `css_plus` constructor, we prove there exists a plus one (i.e. single step) simulation matching the `RTL` step. Conversely, in the latter case, "`pcR1 = pcR0`" and "`isfst = false`", so that we apply the `css_refl` constructor emitting no step.*
- *"`ib = Bop` or `Bload` or `Bstore`". Here, `pcR1` is always instantiated with the `RTL` successor of the instruction. We always perform a single step using the `css_plus` simulation. Those three cases require proving some usual properties about the preservation of symbols, which are trivial.*
- *"`ib = (Bseq b1 b2)`" when `b1` ends with a final instruction (i.e. meaning `b2` is dead code). We invert `MIB` to obtain the inductive relation for `b1`, and directly apply the induction hypothesis from `IBIS` (which contains the `Some` conclusion).*
- *"`ib = (Bseq b1 b2)`" with `b1` at `pcR0` continuing to `b2` at `pcR1`. Again, we invert `MIB` for constructor `mib_seq_Some` and exploit the inductive relation for `b1`. This shows the existence of a step "`pcR0` $\overset{C}{\to}$ `pcR1`" (as we know that `b1` does not end with a final instruction). Then, exploiting the inductive relation for `b2` gives us the second step starting from `pcR1` to a next `RTL` program counter named `pcR2`. Two cases are possible depending on if `b2` ends with a final instruction or not. If so, we know that the last instruction of `b2` satisfy a `match_iblock` relation with the `RTL` one at `pcR2`. We use this relation to prove the corresponding goal. The remaining objective is to demonstrate "`pcR0` $\overset{C}{\to}$ `pcR2`" with `P` being the equality between the `isfst` Booleans for `b1` and for the last instruction of `b2`. By transitivity of the $\overset{C}{\to}$ relation, this amounts to prove "`pcR0` $\overset{C}{\to}$ `pcR1` $\wedge$ `pcR1` $\overset{C}{\to}$ `pcR2`". These two clauses correspond exactly to the two steps obtained from the induction hypotheses on `b1` and `b2`, respectively.*

  Otherwise, when `b2` does not end with a final instruction, we still know the existence of a `pcR2` and of a transition `pcR1` $\overset{C}{\to}$ `pcR2`. The whole transition is proved by transitivity following the same pattern.
- *"`ib = (Bcond bso bnot)`". By destructing the conditional test and exploiting the induction hypothesis, we obtain the step for either the `bso` or the `bnot` block. Each time, we also destruct `ofin` to distinguish the case where the (considered) branch ends with a final instruction. For both branches, when there is a final instruction, we already have the matching relation from the induction hypothesis, and we prove the conditional star step by transitivity. The `is_join_opt` hypothesis from `MIB` gives us, for both non-ending (i.e. with a final instruction) branches, two goals: one where both branches join on the same PC, and one where only the left (for `bso`) or the right (for `bnot`) branch continues. All the four cases are proved by transitivity of relation $\overset{C}{\to}$.*

□

Lemma 8.3.4 (Final step simulation is correct)**.**

```
Lemma final_simu_except_goto sp m Σ_B Σ_R fB fR rs1 m1 pcR1 fin e s
  (STACKS: list_forall2 match_stackframes Σ_B Σ_R)
  (TRANSF: match_function m fB fR false)
  (FS: final_step G_B stack fB sp rs1 m1 fin e s)
  (i: instruction)
  (ATpc1: (RTL.fn_code fR) ! pcR1 = Some i)
  (MF: match_final_inst m (BTL.fn_code fB) false fin i)
  : ∃ s', RTL.step G_R (RTL.S Σ_R fR sp pcR1 rs1 m1) e s' ∧ match_states s s'
```

Proof. We split cases by inverting `MF` and `FS`. We obtain five goals depending on the final instruction, which cannot be a goto (by definition of `MF`). All cases are proven in a similar fashion: first by applying the `RTL` step constructor of the current final instruction and some preservation properties concerning the stack size, the function's signature, and symbols; second, by constructing the matching relation between states from the hypotheses. □

8.3.3 *RTL to BTL Proof*

The RTL to BTL direction is much more complex. Indeed, whereas for the forward simulation of each BTL step by RTL steps at the previous section, we only had to define the simulation relation at

block entries, in the reverse direction, we need to define the simulation relation between each RTL step; that is, after each internal instruction. Hence, our simulation theorem comprises two cases, as illustrated in the two diagrams of Figure 8.2. Moreover, we also need to deal with indirections due to synthetic nodes. Thus, while the simulation seems quite obvious, its formal proof is rather intricate. I also present this proof backward.

8.3.3.1 *Matching Relations and Normalization*

For internal instructions, the simulation is of the "star" type. It does not emit observable events, and is shown to be well-founded with a decreasing measure "|ib|" [◇] of the number of remaining BTL instructions in block ib (excluding Bseq and Bcond).

As previously explained, a Bgoto represents either the remainder of a RTL instruction that ends a block, or a synthetic node (which does not exist in RTL). In our simulation relation, we impose that the RTL state never matches a Bgoto. In this way, both executions remain easily synchronized. In particular, the RTL state never matches a synthetic node.

When a synthetic node follows a real node on the BTL side, the BTL simulation takes two steps, whereas the matching RTL final instruction (emitting event e) is executed in one step. Hence, the final case follows a "plus" simulation to allow for either one (direct) or two (indirect) steps on the BTL side.

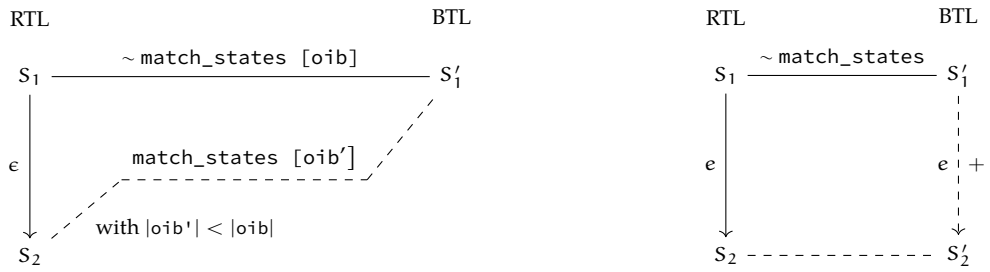


Figure 8.2: Simulation From RTL to BTL: Internal Instructions (left) and Last Instruction (right) (Theorem 8.3.5).

The matching relation between stack frames is very similar to Definition 8.3.5, except that the direct matching hypothesis "m!pcB = Some pcR" is replaced with match_nodes:

Definition 8.3.8 (Mathing relation between stack frames—RTL to BTL)**.**

```
Inductive match_stackframes: ΣR → ΣB → Prop :=
  | match_stackframe_intro m indirect res fR sp pcR rs0 fB pcX
      (TRANSF: match_function m fB fR true)
      (MN: match_nodes m (fn_code fB) indirect pcX pcR)
      : match_stackframes (ΣR res fR sp pcR rs0) (ΣB res fB sp pcX rs0)
```

If a synthetic node was inserted, then match_stackframes holds for the pc corresponding to it.

To make the simulation proof simpler, we start by normalizing BTL blocks. This normalization does not change the semantics, and preserves the match_iblock relationship (of Definition 8.3.4): it just reduces the proof to a *smaller subset of BTL blocks* than syntactically allowed.

Definition 8.3.9 (Normalizing the struture of BTL blocks)**.**
We define the following formal specification:

```
Inductive is_normRTL: iblock → Prop :=
  | norm_Bseq ib1 ib2:
     is_RTLbasic ib1 = true →
     is_normRTL ib2 →
     is_normRTL (Bseq ib1 ib2)
  | norm_Bcond cond args ib1 ib2 i:
     is_normRTL ib1 →
     is_normRTL ib2 →
```

```
    is_normRTL (Bcond cond args ib1 ib2 i)
| norm_others ib:
    is_RTLatom ib = true →
    is_normRTL ib
```

Where `is_RTLbasic` identifies all non-final RTL instructions (i.e. everything except `Bseq`, `Bcond`, "`Bnop None`", and "`BF _ _`"); and where `is_RTLatom` does the same but including final (`BF`) instructions.

The principle of this transformation is to orientate all `Bseq` the same way: given a sequence `(Bseq ib1 ib2)`, `ib1` is expected to correspond to the "beginning" of a RTL basic instruction (but `ib2` can contain another `Bseq`).

Our implementation of this specification operates in continuation passing style, and removes all no-operations marked as not needed in RTL (i.e. those whose argument is `None`)[7].

```
(* NB: [k] is a "continuation" (e.g semantically [normRTLrec ib k] is like [Bseq ib k]). *)
Fixpoint normRTLrec (ib: iblock) (k: iblock): iblock :=
  match ib with
  | Bseq ib1 ib2 ⇒ normRTLrec ib1 (normRTLrec ib2 k)
  | Bcond cond args ib1 ib2 iinfo ⇒
     Bcond cond args (normRTLrec ib1 k) (normRTLrec ib2 k) iinfo
  | BF fin iinfo ⇒ BF fin iinfo
  | Bnop None ⇒ k
  | ib ⇒ Bseq ib k
  end

Definition normRTL ib := normRTLrec ib (Bnop None)
```

We proved two preservation lemmas about `normRTL`: w.r.t. first, the functional semantics of BTL internal instructions (Definition 5.3.4) [◇]; and second, Relation 8.3.4 [◇]. Furthermore, under a valid `match_iblock` hypothesis and for any block `ib`, "`is_normRTL (normRTL ib)`" holds [◇].

Formalizing a matching relation between states that *stutters* (with a decreasing measure) on the BTL side requires *decomposing a BTL blockstep* into multiple small RTL steps. This implies defining an internal correspondence relation between RTL and BTL states: on the RTL side, each step leads to a new state; while on BTL, we want a notion of internal state that corresponds to a *partial* block execution.

Figure 8.3 sketches a relation (defined just below) that corresponds to the simulation invariant between each instruction in a (normalized) block. It expresses the fact that we are *accumulating small RTL steps* that were simulated by the start of execution of the BTL block (i.e. by a partial execution). In this manner, the relation determines the sub-block that remains to be executed in order to continue the simulation.

Partial execution of block starting at `ib0` until `ib`
Red: optional step for pre indirection
Green: `match_strong_state` hypotheses

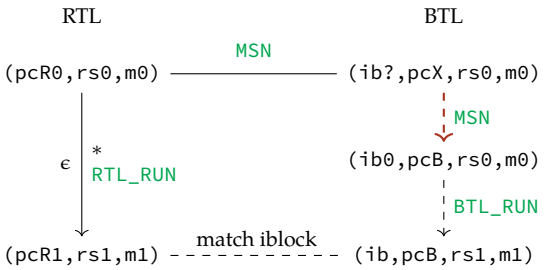


Figure 8.3: Relation `match_strong_state`.

Definition 8.3.10 (Matching relation between states—RTL to BTL)**.** We start by defining the auxiliary predicate `match_strong_state` pictured in Figure 8.3, that matches each single RTL instruction within a blockstep to a BTL instruction (i.e. block) `ib`. It represents the aforementioned sub-block of remaining instructions to simulate. The current matching instruction `ib` is extracted from a surrounding BTL block called `ib0`, itself at label `pcB` (hypothesis `ATpcB` below), where `pcB` matches (after a possible indirection from `pcX`) the RTL entry point of the block at label `pcR0`; the RTL instruction corresponding to `ib` being at label `pcR1`. The `MSN` hypothesis (below and in Figure 8.3) has two roles: it makes the link between `pcR0` and `pcX`; and it tells us if either `pcX` is an indirection with a synthetic node pointing to `pcB` or if `pcX` is a real node (meaning that `pcX=pcB`).

[7] This transformation exponentially increases the number of instructions in the worst case. However, this is not a problem here, since we do not use this transformation for compilation: its sole purpose is to facilitate reasoning.

```
Inductive match_strong_state
m Σ⃗_R Σ⃗_B fR fB sp rs1 m1 rs0 m0 pcX pcB pcR0 pcR1 ib ib0 isfst: Prop :=
  | match_strong_state_intro indirect
      (STACKS: list_forall2 match_stackframes Σ⃗_R Σ⃗_B)
      (TRANSF: match_function m fR fB)
      (MSN: match_synthetic_node m (fn_code fB) indirect pcX pcB pcR0)
      (ATpcB: (fn_code fB)!pcB = Some ib0)
      (MIB: match_iblock m (fn_code fB) (RTL.fn_code fR) indirect isfst pcR1 ib None)
      (IS_EXPD: is_normRTL ib)
      (RTL_RUN: star RTL.step G_R (RTL.S Σ⃗_R fR sp pcR0 rs0 m0) ϵ (RTL.S Σ⃗_R fR sp pcR1 rs1 m1))
      (BTL_RUN: iblock_istep_run G_B sp ib0.(entry) rs0 m0 = iblock_istep_run G_B sp ib rs1 m1)
      : match_strong_state m Σ⃗_R Σ⃗_B fR fB sp rs1 m1 rs0 m0 pcX pcB pcR0 pcR1 ib ib0 isfst
```

In the RTL_RUN hypothesis, G_R is the RTL global environment. We set G_R (and G_B for BTL) as parameters for the entire Coq proof module.

Let me detail the other hypotheses of `match_strong_state`. As for any normal state simulation, we need the matching relations for stack frames and functions as prerequisites. In addition, `MIB` must hold for `ib` and its matching program counter `pcR1`. This requires defining what was already simulated in `ib0` (i.e. all instructions from `ib0` up to—and excluding—`ib`). Therefore, we have two hypotheses `RTL_RUN` and `BTL_RUN` that relate both executions. For `RTL`, it states that n steps (where n can be 0, since it is a star simulation) were already executed, starting from an initial state `(pcR0,rs0,m0)` to a current state `(pcR1,rs1,m1)`. These steps are internal, so their trace should be empty (i.e. "ϵ"). Similarly, the `BTL` run stipulates that starting the execution from a state `(ib0,pcB,rs0,m0)` leads to the same state as the one obtained starting from `(ib,pcB,rs1,m1)` (cf. Definition 5.3.4). On the `BTL` side, the current pc stays `pcB` since we do not change block. Both the `RTL` and `BTL` run hypotheses are drawn in Figure 8.3.

Finally, notice the `IS_EXPD` hypothesis. The latter ensures that current sub-block `ib` respects the canonical form specified in Definition 8.3.9. This requirement is of great help to prove the lemmas depicted in Figure 8.4.

Since we want `ib` to always start with the beginning of a `RTL` instruction, and since our strong matching property handles the simulation step for a possible previous synthetic node, `ib` must always be a real block in `match_states`, as ensured with the `NGOTO` assumption below.

Note that the call and return cases are almost identical to those of Definition 8.3.6.

```
Inductive match_states: (option iblock) → RTL.S → BTL.S → Prop :=
  | match_states_intro m Σ⃗_R Σ⃗_B fR fB sp rs1 m1 rs0 m0 pcX pcB pcR0 pcR1 ib ib0 isfst
      (MSTRONG: match_strong_state
                  m Σ⃗_R Σ⃗_B fR fB sp rs1 m1 rs0 m0 pcX pcB pcR0 pcR1 ib ib0 isfst)
      (NGOTO: is_goto ib = false)
      : match_states (Some ib) (RTL.S Σ⃗_R fR sp pcR1 rs1 m1) (BTL.S Σ⃗_B fB sp pcX rs0 m0)
  | match_states_call Σ⃗_R Σ⃗_B fR fB args m0
      (STACKS: list_forall2 match_stackframes Σ⃗_R Σ⃗_B)
      (TRANSF: match_fundef fR fB)
      : match_states None (RTL.C Σ⃗_R fR args m0) (BTL.C Σ⃗_B fB args m0)
  | match_states_return Σ⃗_R Σ⃗_B v m0
      (STACKS: list_forall2 match_stackframes Σ⃗_R Σ⃗_B)
      : match_states None (RTL.R Σ⃗_R v m0) (BTL.R Σ⃗_B v m0)
```

In `match_states` above, the "`option iblock`" parameter represents the current `BTL` execution state.

8.3.3.2 *Proof Sketch*

The forward simulation requires proving that: (i) the measure is well-founded, which is trivial and already proved in CompCert for any "less than" based measure; (ii) initial states match; (iii) final states match; and (iv) moving a step forward on the `RTL` side preserves Relation 8.3.10.

Proving the correctness for initial and final states is really straightforward, so I omit that here (see the Coq lemmas [◇]). Let us rather focus on main theorem:

Theorem 8.3.5 (Forward simulation between RTL and BTL)**.**

```
Theorem opt_simu s1 e s2 oib s1':
  RTL.step G_R s1 e s2 →
```

```
match_states oib s1 s1' →
∃ (oib' : option iblock), (|oib'| < |oib| ∧ e=ϵ ∧ match_states oib' s2 s1')
                        ∨ (∃ s2', plus step G_B s1' e s2' ∧ match_states oib' s2 s2')
```

Proof. The left and right alternatives of this theorem's conclusion correspond to the left and right schemes of Figure 8.2, respectively. By decomposing the `match_states` hypothesis, we obtain cases for normal, call, and return states. The call state is itself decomposed in two for internal and external calls. For internal calls, external ones, and return states, the proof requires splitting the direct and indirect cases, and taking the right alternative (i.e. either a plus one or a plus two simulation).

Most of the complexity here lies in the normal state simulation. After unfolding the `match_strong_state` hypothesis, we obtain a `match_iblock` hypothesis where the last parameter is `None` (since in Definition 8.3.10, we only match blocks whose execution ends with a final instruction). This leads to a new decomposition in four cases: final (`mib_BF`), goto (`mib_exit`), sequence (`mib_seq_Some`) and branch (`mib_cond`):

- `mib_BF`. We first exploit Lemma 8.3.6 below to show the existence of `oib'`, `s2'`, a `BTL` hypothesis `BSTEP` from `s1'` to `s2'`, and a matching relation `MS` between `s2` and `s2'`. Since we reason on final instructions, we take right conclusion alternative for each of them. The plus step is either direct, which corresponds exactly to `BSTEP`; or indirect in two steps. In the latter case, the first step is the synthetic node (i.e. semantically, the identity), solved by application of Lemma 8.3.7, and the second is `BSTEP` again. For both the direct and indirect cases, the matching relation between states is exactly `MS`.

- `mib_exit`. Contradiction with hypothesis `NGOTO`.

- `mib_seq_Some`. Let "`ib = (Bseq b1 b2)`". By discriminating constructors of the `IS_EXPD` hypothesis, we ensure that `b1` matches a non-final equivalent `RTL` instruction, and that `b2` is correctly normalized.

  As we are in the sequences inductive case, we have a `match_iblock` hypothesis for both `b1` and `b2`. Considering that "`is_RTLbasic b1 = true`", inverting Definition 8.3.4 leads to four cases: "`Bnop (Some _)`", `Bop`, `Bload`, and `Bstore`.

  To solve them, we use the `match_strong_state` property for inductive instructions of Lemma 8.3.8. The measure to prove is "`|b2| < |b2| + 1`", which is trivial. The strong matching for `b1` is proved by assumption, and the step by using the `RTL` operational semantics for the considered instruction. When `b1` is an operation or memory operation, the strong matching for `b2` requires proving that the semantics is preserved by switching from G_R (the `RTL` global environment) to G_B (the `BTL` one). This is trivial since symbols are preserved by translation.

- `mib_cond`. Here also, we know that both the `bso` (true) and the `bnot` (false) blocks are in their normal form. In addition, as we match a terminating (i.e. with a final instruction) block, the `is_join_opt` property from `match_iblock` tells us that both branches end with a final instruction.

  According to `b` the Boolean result of the condition, we apply Lemma 8.3.8 with the current `Bcond` as `ib1` and with either `bso` or `bnot` for `ib2`. To demonstrate the strong matching hypothesis, we simplify the `BTL` run hypothesis by splitting cases for `b`. Finally, we prove the trivial decay of the measure "`|if b then bso else bnot| < |bso| + |bnot|`".

□

Lemma 8.3.6 (Correctness of the real final blockstep[5])**.**

```
Lemma opt_simu_direct
  m indirect Σ_R Σ_B fR fB sp rs1 m1 rs0 m0 pcB pcR0 pcR1 ib0 s2 e (fin: final) inst iinfo
  (TRANSF: match_function m fR fB)
  (STACKS: list_forall2 match_stackframes Σ_R Σ_B)
  (STEP: RTL.step G_R (RTL.S Σ_R fR sp pcR1 rs1 m1) e s2)
  (NGOTO: is_goto (fin iinfo) = false)
  (MAP: m ! pcB = Some pcR0)
  (MFI: match_final_inst m (fn_code fB) indirect fin inst)
  (ATpcB: (fn_code fB) ! pcB = Some ib0)
  (RTL_RUN: star RTL.step G_R (RTL.S Σ_R fR sp pcR0 rs0 m0) ϵ
                              (RTL.S Σ_R fR sp pcR1 rs1 m1))
  (BTL_RUN: iblock_istep_run G_B sp (entry ib0) rs0 m0 =
```

```
          iblock_istep_run GB sp (fin iinfo) rs1 m1)
  (PC: (RTL.fn_code fR) ! pcR1 = Some inst)
  : ∃ oib' s2', step GB (BTL.S Σ⃗B fB sp pcB rs0 m0) e s2' ∧ match_states oib' s2 s2'
```

Proof. We split the goal for each final instruction in `match_final_inst`[5]. Every case is discharged using a similar pattern. We assume that `oib'` and `s2'` exist, to then split the goal with on one side, the step relation, and on the other, the matching relation.

By construction, proving `step` reduces to prove Definition 5.3.6: the relational semantics for the block body and final instruction. The body, encoded by Relation 5.3.2 as a predicate `iblock_istep`, goes from state `(rs0, m0)` to `(rs1, m1)` for the current final instruction. In fact, this is simply the relational form of hypothesis `BTL_RUN`, which we can convert using Lemma 5.3.1. For the final instruction, it suffices to apply the `final_step` (cf. Relation 5.3.5) constructor corresponding to the current instruction, and to discharge its prerequisites (for instance, if the final instruction is a return, one need to prove that the function's stack frame was freed: "`Mem.free m0 stk 0 (fn_stacksize fB) = Some m0'`"). These prerequisites being identical to their `RTL` version, we obtain them by inverting `STEP`. Nonetheless, when they depend on the function's attributes (e.g. the stacksize, the signature, etc), it is needed to first rewrite preservation properties from `match_function`.

Solving the final step goal gives a concrete instantiation of `s2'`, so that we demonstrate `match_states` by construction. □

Lemma 8.3.7 (Correctness of the synthetic node blockstep).

```
Lemma opt_simu_indirect m indirect Σ⃗B fB sp rs0 m0 pcX pcB ib (fin: final) inst iinfo
  (PC: (fn_code fB) ! pcX = Some ib)
  (BLK: entry ib = Bgoto pcB iinfo)
  (MFI: match_final_inst m (fn_code fB) indirect fin inst)
  : step GB (BTL.S Σ⃗B fB sp pcX rs0 m0) ϵ (BTL.S Σ⃗B fB sp pcB rs0 m0)
```

Note that here, proving only the case where `indirect` is true would suffice to complete the proof, but general case also holds.

Proof. Inverting `MFI` splits the goal in all final[5] instructions. By construction, the step for an iblock (cf. Definition 5.3.7) requires proving two properties. First, the block's existence in the CFG (i.e. that "`(fn_code fB) ! pcX = Some ib)`". This is exactly hypothesis `PC`. Second, a valid `iblock_step` relation: this is trivial by rewriting hypothesis `BLK`, and by repeated application of the inductive types' constructors. □

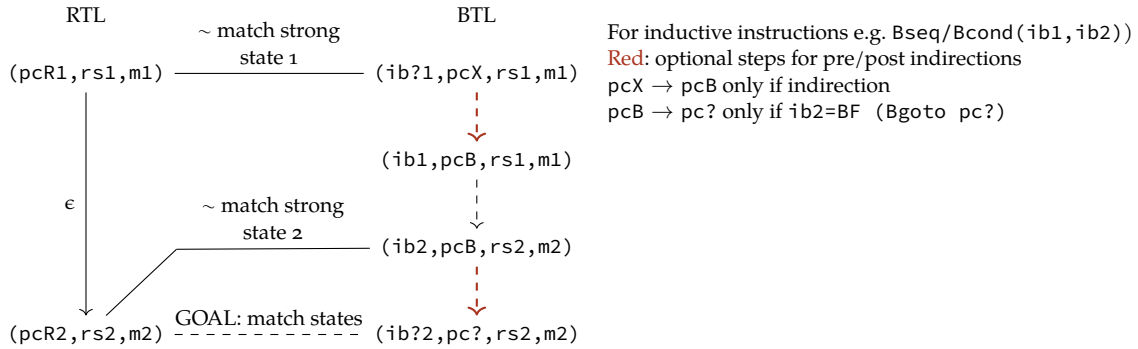


Figure 8.4: Decomposition of Theorem 8.3.5 into Lemmas 8.3.6 (bottom subdiagram for final instructions) and 8.3.8 (with two `match_strong_state` relations for inductive instructions).

Lemma 8.3.8 (Simulation property for inductive instructions).

```
Lemma match_strong_state_simu
  m Σ⃗R Σ⃗B fR fB sp rs2 m2 rs1 m1 rs0 m0 pcX pcB pcR0 pcR1 pcR2 isfst ib1 ib2 ib0 n s2
  (EQs2: s2=(RTL.S Σ⃗R fR sp pcR2 rs2 m2))
  (STEP: RTL.step GR (RTL.S Σ⃗R fR sp pcR1 rs1 m1) ϵ s2)
  (MSS1: match_strong_state m Σ⃗R Σ⃗B fR fB sp rs1 m1 rs0 m0 pcX pcB pcR0 pcR1 ib1 ib0 isfst)
  (MSS2: match_strong_state m Σ⃗R Σ⃗B fR fB sp rs2 m2 rs0 m0 pcX pcB pcR0 pcR2 ib2 ib0 false)
  (MES: |ib2| < n)
  : ∃ (oib' : option iblock), (|oib'| < n ∧ match_states oib' s2 (BTL.S Σ⃗B fB sp pcX rs0 m0))
      ∨ (∃ s2', plus step GB (BTL.S Σ⃗B fB sp pcX rs0 m0) ϵ s2'
          ∧ match_states oib' s2 s2')
```

See the proof scheme of Figure 8.4.

Proof. We first destruct equality "`ib2 = BF (Bgoto s iinfo)`".

- *When `ib2` is a goto, four cases are possible. Node `ib1` can be preceded or not by a synthetic node; while node `ib2` can be followed or not by a synthetic node (i.e. according to the content of its successor `s`). Assuming a successful run of our validator, we know that the real, non-synthetic successor of `ib2` is always mapped in `m` to the `RTL` node at `pcR2` (cf. the prerequisite of Definition 8.3.3). Hence, we pose two trivial auxialiary lemmas: `match_states_entry_direct` [◇] that guarantees the matching relation for states in the direct case; and its equivalent for indirect paths named `match_states_entry_indirect` [◇]. The former is correct by construction, and the latter by applying the former and by construction also.*

  The proofs of these two secondary lemmas are made simpler thanks to `is_normRTL`.

  From here, we prove the four cases as follows: `oib'` and `s2'` are assumed to exist, and we always chose the second goal. For the two cases where `ib2` is not followed by a synthetic node, we prove the matching relation with lemma `match_states_entry_direct`. Otherwise, when there is a synthetic node after `ib2`, we apply `match_states_entry_indirect`.

  Conversely, the "plus step" conclusion requires a single step when `ib1` is not preceded by a synthetic node; and two steps otherwise.

  Then, by assumption and by applying Lemma 5.3.1, we finish the step demonstration for all goals.

- *When `ib2` is any other instruction, we state that "`oib'= Some ib2`". We focus on the left conclusion alternative: we already know by hypothesis that "`|ib2| < n`", and `match_states` is proved correct by applying the "intro" constructor.*

□

9

CLOSING REVIEW ON BTL†

The last four chapters introduced two defensive and formally verified translation validators for the block transfer language intermediate representation: a symbolic execution engine, and a CFG morphism checker. This chapter concludes our work on BTL*. We review some features and limitations of this approach, and some possible future improvements that would be interesting to work on. Finally, we position BTL w.r.t. related work, and summarize the key points of this part.

9.1 DEVELOPMENT SIZE

Here is a summary of code size for the whole BTL framework presented in this part (again, in significant lines of code, sloc). Concerning OCaml oracles, the RTL to BTL one fits in approximately 200 sloc, and the BTL to RTL one is slightly longer, near 300 sloc (comprising the factorization of §8.2.2).

As in §4.3, column "Defs" comprises both Coq definitions and types.

Project	Defs	Proofs
BTL IR	252	20
BTL projection checker (Section 8.3.1)	296	121
RTL → BTL (Section 8.3.3)	313	377
BTL → RTL (Section 8.3.2)	146	249
BTL SE theory (Chapter 6)	1844	1862
BTL SE refinement (Chapter 7)	1612	1411
BTL rewriting engine (RISC-V only, Section 7.6)	1209	1038
BTL passes module (Section 11.7)	122	60
Total	5794	5138

COMMENT ON OUR CFG MORPHISM OF CHAPTER 8 Tristan and Leroy [142] proposed a "graph-of-trees" intermediate representation for validating some trace scheduling over Mach programs. Their "graph-of-trees" are close to our normalized BTL CFG (see `is_normRTL` in Definition 8.3.9). Actually, our conversion between RTL and BTL generalizes their conversion between Mach programs and graph-of-trees (§5.4 of their paper) on several points: a more general notion of block; support of (un)foldings through CFG morphisms (their implicit CFG morphism is identity); support of synthetic nodes. Despite these generalizations, our Coq code (755 sloc of definitions and types plus 747 sloc of proofs, i.e. by summing both columns for the projection checker and translation directions in the above table) seems significantly simpler that their one (986 sloc of definitions and types + 2418 sloc of proofs—cf. §6 of their paper). In particular, the complexity of their code leads them to conclude: "*the part labeled "tree semantics" [in their Figure 5], which includes the definition and semantics of trees plus the validation of the conversion from list-of-instructions to graph-of-trees, is the largest and most difficult part of this development.*" We suspect that their conversion proof was much harder, because Mach (being the last IR before assembly) represents a program as a list-of-instructions instead of an explicit CFG. This illustrates that the complexity of proofs is very dependent on data representations. In our development, the conversion between RTL and BTL was much smaller and easier than the symbolic simulation test, which we discuss below.

COMMENT ON THE INTRICACY OF CHAPTERS 6 & 7 The technique of refining (in Coq) an efficient symbolic simulation test using hash-consing from a purely functional model was introduced by Sylvain Boulmé in the design of AbstractBasicBlock [23, §3.3]. Given that the simulation theory of

*Some text of this chapter is reused from our OOPSLA'23 paper [65]†.

AbstractBasicBlock is relatively simple (less than 350 sloc) described by two pages in [23, §3.3.1], its refinement can be summarized in under two pages, see [23, §3.3.3]. Later, this technique was substantially generalized by Cyril Six for RTLpath: the theory of near 1.8K sloc, is detailed in 26 pages [133, §7.1, §7.2]. The refinement (of around 1.5K sloc) is summarized in 16 pages. It has almost no rewriting rules, except for trapping loads. In the case of BTL, the theory is even more intricate (mainly because of invariants). It takes 3.7K sloc (see the above table) described in roughly 30 pages (Chapter 6). Its refinement takes around 3K sloc for the core engine, plus 2K sloc for normalized rewritings. In summary, it has become complicated to give a simple description of this implementation without hiding too many details.

9.2 GENERAL REMARKS

In Chapter 10, we will see that basic blocks fit very well for complex optimizations.

BLOCKS' SIZE Thanks to our intra-procedural simulation technique illustrated through this part, the need to select larger and larger blocks to extend the scope of optimizations became less important. Nonetheless, this need still exists in some situations: for instance, our simulation test modulo invariants cannot reason on conditions, hence the necessity of selecting superblocks for if-lifting (see §9.3 below). Actually, using a block structure larger than superblocks would tend to increase the *simulation's complexity* (cf. §2.2.3.3 and §9.3 below), which is of course not desirable. Furthermore, it is for instance not clear what would be a scheduling optimization on non-linear code sequences, while it is well defined on basic and superblocks. Invariants enable applying *most* intra-procedural transformations on a code cut in basic blocks.

PARTITIONING The idea of using blocks to perform global optimizations might be counter-intuitive at first sight, but in fact it has the advantage of *drastically limiting* the amount of invariants to verify: without blocks, one would need to enforce the correctness of invariants at each instruction, which would obviously be much more expensive.

SIMULATION INVARIANT I wrote in the margin notes of §1.2.2 that the name "invariant" was referring to the simulation invariant. In RTLpath, this simulation invariant was defined as the equality between the source and target states (that is, the registers—eventually modulo liveness—and memory) at the entry of each block.

Inter-block transformations cannot be validated using this invariant for two reasons. Firstly, a transformation only needs to consider initial states at the block entry that are *reachable* by the program, whereas the RTLpath simulation test requires proving the preservation *for any values* of the registers and memory at the block (in fact, path) entry. Secondly, the source and target states at the entry of the block *do not need to be equal*. For instance, DCE only maintains the equality of the live registers; and some transformations use auxiliary registers, which hold meaningful values only in the target program. Indeed, recall that the original RTLpath, as presented in [133], could validate neither DCE nor the introduction of fresh registers. With the extension I proposed in §4.5.1, validating the introduction of fresh registers became possible, but DCE remained unsupported. This is because RTLpath only validates a weak liveness analysis of the source program.

In contrast, we do not use an ad-hoc validation of register liveness. While our oracles generate only invariants for live registers of the *target* program, the validation of this liveness analysis implicitly results from the preservation of gluing invariants between source and target registers. This corresponds to a generalization of the RTLpath's simulation invariant. An incorrect liveness analysis will result in an invariant that is invalid after substitution of the target registers, because it will still involve a target register not itself bound to a symbolic expression of source registers.

Usually, we fill our invariants with a *strong* liveness information computed using the standard data-flow algorithm. Doing so forces us to apply DCE before executing the simulation test, as the liveness validation would fail otherwise. Ultimatly, one could still provide a *weaker* liveness information in the invariants to avoid applying DCE, albeit this does not seem to be very useful in practice.

DEAD CODE ELIMINATION The DCE already provided by CompCert was directly proved to be correct. In contrast, in this work, we validate our DCE "for free" as a block transformation with our symbolic simulation test, which illustrates the versatility of that approach. Nevertheless, the

existing DCE of CompCert is slightly more powerful than ours, because it also eliminates some redundant conditions. Validating exactly CompCert's DCE with our validator would require a non-trivial extension, not of priority interest. Our DCE oracle was designed to be light and to eliminate directly useless assignments generated by our SR oracle (see §10.4).

9.3 LIMITATIONS

Our simulation test has two kinds of limitations: performance ones (impacting CompCert running times) and expressivity ones (restricting the class of simulations that can be validated).

PERFORMANCE In theory, any piece of code without loops may be represented as a BTL block. However, we saw in §2.2.3.3 that in practice, due to our naive trace partitioning, SE is exponential over the number of internal joins of the input block. But, as we currently only apply our checker to *at most* extended blocks[1], which, by definition, do not have such joins, this is not an issue. Furthermore, for blocks with a bounded number of internal joins, and without rewriting rules, our symbolic execution is linear in the size of invariants and blocks. In the general case, its cost depends on the normalization system. For example, for the normalization of affine forms (for SR), it is expected to be quadratic in the worst case. Lastly, the comparison of symbolic states costs $\mathcal{O}((l+t)\times e)$ where l is the maximal number of liveout registers and t is the maximal number of trapping instructions, both per execution path, and with e the number of execution paths (coinciding with the number of exits for blocks without internal joins). Block selection is a way to finely control e, and thus checker performance.

EXPRESSIVITY The relative simplicity and efficiency of our checker comes at a price: its expressive power is limited. (i) Our invariants only support equations of the form "$r = sv$" but not the more general "$sv_1 = sv_2$": this limitation avoids the need of costly saturation techniques. (ii) Our simulation test performs no reasoning on *conditions*. It simply checks that the two symbolic states under comparison have the same binary decision tree structure, with syntactically equals conditions on nodes. Future works include supporting conditions within invariants with a more expressive comparison of decision trees and preconditioned rewriting rules. (iii) Our invariants implicitly express that their trapping expressions are actually safe in the execution context. This forbids the target to *anticipate* traps w.r.t. the source. Avoiding this restriction would require prophecies [2] ensuring that these traps will *eventually be observed on the source before any subsequent observable event*. Besides generalizing the semantics of our invariants, this would need introducing a notion of "decreasing variant" forbidding never-realized prophecies. Currently, we partly overcome this restriction with the help of CFG morphisms (e.g. see §10.2.6.2). (iv) Our symbolic simulations cannot deal with invariants over an abstract domain. It seems however desirable to at least enable symbolic simulations to benefit from previous formally verified static analyses. (v) Our memory model is very limited: the whole memory is considered as a single variable. It seems however desirable to integrate a finer memory model, with some alias analyses. (vi) Our invariants cannot express transformations over memories. This forbids for example validating the loop-invariant code motion of a memory update after a loop. Alias analysis is a prerequisite for such a feature.

Possible solutions for points (iv), (v) and (vi) are provided in §11.2.

9.4 SOME RELATED WORK

Refer to §2.5 for a more complete state of the art.

As explained in §2.5.2, our approach induces very different concerns than classical approaches of translation validation like those in [31, 78] based on SMT solving.

A CO-DESIGN STRATEGY In our work, "*synchronization points*" and "*invariants*" between source and target code (aka "*program alignment*") are directly given by the oracles that actually perform the translations. Generating this information inside the transformation phase is not very difficult: it requires quite simple refinements of translation algorithms; in contrast, reconstructing them from compiler output is hard. We thus do not really experience "*false alarms*", because our translations

[1] We use basic blocks for code motion & strength-reduction, superblocks for scheduling, and extended blocks for if-lifting—see §11.3.1.

are designed with the validator limitations in mind. In addition, the design of our validators is very constrained, because we want them to be formally verified, lightweight at compile-time (i.e. quasi-linear in practice), and predictable on "*false alarms*". This prevents us from using SMT-solvers in the current state of the art.

FORMALLY VERIFIED SSA OPTIMIZATIONS The Coq-verified translation validators for SSA optimizations proposed in [45–47] for CompCert rely on strong SSA invariants (e.g. dominator sets). In an alternative design, we could imagine extending BTL with optional parallel moves of registers at exit points. This would allow representing (partial) SSA forms within BTL using Appel [7]'s representation: without explicit ϕ-nodes, but rather by encoding them with explicit parallel moves on joining edges. The validator would completely ignore SSA-invariants, but would be able to compare SSA forms with non-SSA ones. Moreover, only SSA oracles would have to maintain SSA-invariants, without need of formal proof of this.

LOOP OPTIMIZATIONS As I explain in §11.3.1, our framework validates superblock scheduling which interleaves the computations of successive iterations within a loop. Tristan and Leroy [144] showed that symbolic simulation is able to validate even more advanced scheduling techniques, such as *software pipelining* [89]. It remains however to understand how their technique could be integrated to our framework.

I discuss existing work about redundancy elimination in conclusion of Chapter 10.

Clément and Cohen [33]'s work on validating optimizations in the polyhedral model supports much more advanced loop transformations than our does; but we support a much wider class of input programs within a general-purpose compiler. While special-purpose translation validation is in the spirit of CompCert's design, it seems very challenging (but very interesting) to integrate such advanced techniques within a formally verified general-purpose compiler.

9.5 IN SUMMARY

We mainly combine two kinds of translation validators: the first one, described in Chapter 8, targets code duplications or factorizations; the second one, described in Chapters 6 & 7, targets what we call inter-block transformations. At high-level, each of our optimizations can be viewed as a composition of several transformations on the RTL code, with generally "preprocessing passes" (e.g. loop-unrolling or register renaming), the core of the optimization (e.g. superblock scheduling) and possibly some "post-processing passes" (e.g. code factorization). Each transformation must be checked by a validator. Distinct transformations may be checked by the same validator. If each transformation in a sequence can be checked by the same validator, then the oracles performing them can sometimes be composed into a single oracle requiring a single validator run at the end.

1. BTL is an IR close to RTL, and it shares the same semantic limitations as those listed in §3.4.2. On the other hand, this similarity simplify the compatibility with RTL and the related translations passes between both IRs.
2. Structuring the CFG with syntactically defined blocks related by invariants is a way to generalize the RTLpath representation, and to avoid the trade-off between local—but scoped—optimizations (e.g. scheduling) and the need for global ones (e.g. SR or CSE).
3. The generic block format leaves optimizations free to change the block type: the "if-lifting" limitation of RTLpath (recall §4.4.5) is no longer a problem with BTL.
4. More generally, all limitations (and their consequences) mentioned in §4.4.5 are no longer a concern in BTL.
5. Comparing to the explicit liveness validation of RTLpath, the implicit validation in BTL is strictly more expressive (i.e. by allowing both DCE and the introduction of fresh registers).
6. Validation is helped by *hints* provided by oracles: information easy for the oracle to yield, but that would be hard to have the validators reconstruct.

Part III

OPTIMIZATION ORACLES

I present in this part my second main contribution: a code motion and strength-reduction algorithm named lazy code transformations. This new optimization is detailed in Chapter 10 and is verified with the framework presented in Part ii.

Then, Chapter 11 examines other optimizations that demonstrate the versatility of our defensive validation approach.

10

LAZY CODE TRANSFORMATIONS†

We *combined* and *enhanced* the lazy code motion (LCM) & lazy strength-reduction (LSR) algorithms of Knoop, Rüthing, and Steffen. The resulting oracle, that we named lazy code transformations (LCT) [64]†, is implemented in OCaml, and was co-designed with the validation framework presented in the previous part. LCT operates over BTL *basic blocks*, and generates invariant annotations from data-flow analyses as *certificates* for our defensive validator. It is, as far as we know, and at the time of writing, the first formally verified strength-reduction of loop-induction variables.
We introduce several refinements w.r.t. the original papers, that I explain in this chapter*.

The main contributions suggest a generalization of LSR: (i) that operates over basic blocks by adapting the analysis of Knoop, Rüthing, and Steffen [85], as it was done in [83] for LCM; (ii) performed together with LCM in a single transformation; (iii) which integrates a rewriting procedure to widen the scope of strength-reduction over sequences of operations, rather than on each instruction independently; (iv) inferring the invariants needed for the translation validation from data-flow equations (including liveness analysis).

Below, Section 10.1 motivates our choice for LCM & LSR and shows the limitations we overcame w.r.t. the original algorithms. Then, LCM is presented in Section 10.2 along with our improved LCT implementation; and Section 10.3 explains how we refine this enhanced code motion to also integrate LSR. The method we use to generate history and gluing invariants is detailed in Section 10.4. Finally, Section 10.5 discusses related work and concludes.

10.1 INTRODUCTION: CODE MOTION, STRENGTH-REDUCTION, AND RISC-V

We saw in §4.1 that reordering instructions was effective and important for simple cores ("simple" in the sense of embedded, in-order, or featuring a reduced ISA). Here, I give a short introduction to code motion (CM) & strength-reduction (SR), and explain why these optimizations are important for simple cores. In addition, the section details the reasons that motivate our choice for LCM & LSR, and the challenges it brings.

From here, we note "LCM" & "LSR" to refer to the original algorithms of Knoop et al., and "LCT" to refer to our new oracle—which combines and improves LCM & LSR—implemented in CHAMOIS-COMPCERT.

10.1.1 *Main Concepts and "Lazy" Transformations*

Code motion consists in *anticipating some instructions* in order to remove redundant computations. For example, by data-flow analysis, we may detect expressions remaining constant within a loop and anticipate their computation before the loop: this is loop-invariant code motion (LICM). However, if done carelessly, this transformation may anticipate a loop-invariant expression that traps (e.g. a memory load from a potentially invalid pointer, or a division operation on some architectures), whereas this computation is unreachable in the original loop. Safe elimination of such computations—that are redundant on *some but not all* program paths—is called partial redundancy elimination (PRE). In contrast, full redundancy elimination (FRE) eliminates computations that are redundant on *all* paths. According to Bodík, Gupta, and Soffa [22], "*to achieve a complete PRE, control flow restructuring must be applied. However, the resulting code duplication may cause code size explosion.*" They propose to guide these CFG restructuring with path-profiling and data-flow frequency analysis.

Lazy code motion [84] performs safe and optimal PRE without CFG unrolling, while limiting the register pressure induced by code motion. In fact, lifting operations in the CFG of the program certainly allows removing common subexpressions, but it can also increase the live range (i.e. the interval during which a variable is live). Instructions are safely anticipated but not earlier than

*A large part of this chapter is adapted from my ICOOOLPS'23 paper [64]†, and a smaller part from our OOPSLA'23 paper [65]†.

```
ldr  x0,[x0,w1,sxtw#3]
```

```
slli x6,x11,3
add  x6,x10,x6
ld   x6,0(x6)
```

Figure 10.1: AArch64 (left) vs. RISC-V (right) Addressing.

the minimum necessary—hence the name "lazy"—to reach computational optimality (i.e. with a minimal average running time for PRE without CFG unrolling). In the words of Knoop, Ruthing, and Steffen [83], "*the point of this algorithm* [LCM] *is to place computations as late as possible in a program, while maintaining computational optimality.*" Essentially, among computationally optimal code motions, LCM selects those that minimize register pressure.

Extending their work on code motion, Knoop, Rüthing, and Steffen [85] proposed a lazy strength-reduction algorithm and an implementation-oriented paper about LCM [83][1].

Rather than just moving instructions, or simply replacing (sequences of) computations by semantically equivalent—but more efficient—ones, we tackle a much more advanced approach: LSR. It is a generalization of LCM that moves instructions (e.g. out of a loop) modulo insertion of compensation code (e.g. within the loop), in order to operate a *strength-reduction*: to replace the moved instructions by simpler (e.g. faster at runtime) ones. In other words, LSR reduces computations *while* moving them. To get a more concrete idea of the kind of transformations performed by LCM and LSR, the reader can refer to Example 5.1.1, which was obtained by applying our LCT oracle. More complex examples are provided in Sections 10.2.7 and 10.3.5. Note that simpler forms of SR, which for instance replace a multiplication by a power of two with a left shift, are already implemented in CompCert.

10.1.2 *Why Does RISC-V Need More Optimization?*

Despite the fact that RISC-V is rising for safety-critical systems (SCS) applications (cf. §1.2.1), CompCert is still *less efficient* than GCC on this architecture. Indeed, the instruction set architecture being truly reduced [145], the compiler must be clever to generate efficient assembly code.

Some architectures provide instructions or addressing modes for commonly found patterns, such as array addressing. For example, on KVX, AArch64, and x86 (this list is not exhaustive), there are addressing modes that directly implement array accesses: given a base pointer a and index i, load or store $a[i]$, they automatically compute the address "$a + si$" where s, also known as the stride, is the size of the data type (number of bytes). They may even perform a signed or unsigned extension over i, since i is typically a 32-bit integer while a is 64-bit on architectures with 64-bit pointers. In truly reduced instruction sets like RISC-V, these patterns instead result in a multi-cycle sequence of instructions, amenable to SR. Figure 10.1 shows the single AArch64 load generated for array access "$x = a[i]$" (with an addressing mode that shifts an index by three bits and adds it to a base address) compared to the succession of RISC-V instructions that shift, add, then load. The sequence can even contain five instructions if an unsigned extension is needed!

If load from L1 cache takes 3 cycles, and each basic arithmetic instruction takes 1 cycle, the overall sequence could take 5 or even 7 cycles whereas on other architectures it would take 1.

As another example, on most architectures, extending a 32-bit unsigned integer value to 64-bit takes one instruction; but on RISC-V it takes two instructions (a logical left shift of 32-bit followed by a logical right shift of 32-bit)[2].

The lack of SR for such sequences may explain why CompCert performs poorer compared to GCC on RISC-V than on other architectures. When such a triplet of instructions appears in a loop, strength-reduction becomes particularly beneficial in order to minimize the number of cycles per iteration.

[1] This paper proposes a realistic and practical approach to integrate LCM in an existing compiler.

[2] The integer promotion pass of Benjamin Bonneau, briefly described in §11.2, performs an interval analysis and then replaces unsigned conversions over numbers that anyway are always non-negative with signed conversions, whose cost is null on RISC-V since 32-bit operations after the upper bits as though their results were signed.

10.1.3 *Why Choose the LCM & LSR Data-Flow Based Algorithms?*

The principle of a data-flow algorithm like LCM or LSR is to infer information from data-flow analyses on a CFG. This consists in assessing the validity of certain predicates for each potential candidate detected by the algorithm. A *candidate* designates an expression (i.e. an instruction) of the underlying intermediate representation—here BTL—that will possibly be optimized (e.g. by code motion or strength-reduction). The LCM & LSR predicates define a Boolean value for each candidate instruction, at each CFG node.

These algorithms were originally designed for single instruction CFGs in [84, 85]. Later, the authors adapted LCM for basic blocks in [83]. To our knowledge, such a work was never published for LSR.

Our focus was on demonstrating our defensive validation mechanism rather than exploring various optimization algorithms. For an overview of state-of-the-art loop SR algorithms, refer to §10.5.3.

Our choice to adapt, combine, and implement LCM & LSR rather than another similar algorithm is mainly motivated by three reasons:

1. They operate over basic blocks, which lowers the amount of *hints* (i.e. invariants) needed for the validation compared with single instruction based algorithms. Hence, the communication between the oracle and the validator is more efficient, as well as the validator itself. Moreover, the basic block structure reduces the number of nodes in the CFG, as nodes become blocks, thereby reducing the length of predicates to store;
2. The data-flow approach helps in generating invariants (see §10.4);
3. They are among the most efficient algorithms of this family (CM and SR of loop-induction variables) not based on static single assignment (SSA). Actually, as mentioned in §2.5.3 for the de-SSA pass, and as expressed by Demange [45, §5.1.1, §5.11] for the translation *to* SSA, "*they* [*SSA generation algorithms*] *rely on complex properties of graphs, that are difficult to justify formally.*"

The code of our LCT oracle is split in three parts. Their *online documentation* is available here: (i) core module containing high-level types and operations [⋄]; (ii) main module containing all analyses and rewriting procedures [⋄]; (iii) backend specific module for the SR part on RISC-V [⋄].

10.1.4 *Limitations of LCM & LSR*

We want to combine the basic block LCM with LSR; this requires generalizing the latter to operate on basic blocks. Indeed, due to the discrepancy in the representations supported by these two algorithms, one might be compelled to redo some computations already performed by LCM (on basic blocks) in the implementation of LSR, which relies on the same base of logical predicates.

On another note, the original algorithms do not specify any order of treatment for candidate instructions, and insert a move in place of each replaced instruction. The problem with this behavior is that it prevents them from moving or reducing *sequences* of operations. In real compilers, these sequences are often generated by *instruction selection*. For example, on RISC-V, a multiplication $c = i \times 10$ can be replaced with the less costly sequence "$a = i \ll 1; b = i \ll 3; c = a + b$". Usually, in a compiler like GCC or LLVM, this would not really be an issue since one could still apply the optimization *before* the selection. However, to facilitate its formal proof of correctness, the instruction selection pass in CompCert operates on a structured intermediate representation, placed upstream of most other optimizations (cf. Figure 3.1). Our LCT, which works on BTL, is located downstream. Let us imagine a loop containing the above sequence: an addition of the results of two shifts[3], where i is an induction variable incremented by 3 at each iteration. One would like to take out of the loop the whole sequence, and to insert an addition $c = c + 30$ instead (i.e. $30 = 3 \cdot 2^1 + 3 \cdot 2^3$). The increment on i would not be modified, and the resulting code would be much more efficient. But, the insertion of moves constrains the analyses by creating new dependencies and makes LCM & LSR unable to handle such sequences.

In practice, instruction selection is not the only source of reducible sequences: they may also appear directly in the source code, or be produced for calculations of memory addresses during translations (like the left shift of Figure 10.1).

Our goal in this chapter is to co-design an enhanced version of LSR, one that integrates LCM, overcomes those limitations, and is capable of providing the validator with the expected, correct

[3] This pattern is actually implemented in the mainline CompCert for RISC-V.

See Chapter 12 for an experimental evaluation of the runtime performance of the generated code.

invariants. We have measured that adding these optimizations significantly improves the performance of the code generated by CompCert on 64-bit RISC-V, without degrading compilation times (including formally verified defensive checks).

10.2 LAZY CODE MOTION

For LCM, candidates can be either arithmetic *operations or loads* (i.e. `Bop` or `Bload`). We do not aim at moving stores, no-operations, conditionals, and final instructions. Predicates can be of two types: either *global*, if their value at node n depends on the other nodes of the graph; or *local*, if their value at node n can be determined only by examining n. Global predicates are obtained either by data-flow analysis (i.e. computed as a fixed point), or by logical combinations (i.e. conjunction and disjunction) of other predicates. LCM is built on the seminal work of Morel and Renvoise [111][4].

SIDE-NOTE ABOUT DATA-FLOW COMPLEXITY One significant advantage of LCM (and therefore of LCT) over many other similar code motion algorithms (and notably to Morel and Renvoise [111]), is that it exclusively relies on unidirectional data-flow analyses. Unidirectional here means that the fixed point computation steps either forward with the "successor" relation or backward with the "predecessor" relation, *but not both*. Data-flow equation systems involving both directions are referred to as *bidirectional*.

Intuitively, one might expect that unidirectional fixed points are less complex. According to Knoop, Ruthing, and Steffen [83], "*Bidirectional algorithms, however, are in general conceptually and computationally more complex than unidirectional ones: e.g., in contrast to the unidirectional case, where reducible programs can be dealt with in* $O(n \log(n))$ *bit vector steps, where* n *characterizes the size of the argument program (e.g., number of statements) [in our case, in a* `BTL` *function], the best known estimation for bidirectional analyses is* $O(n^2)$ *(cf. [49, 50]).*"

In their paper, Knoop et al. clarify these claims in a footnote: "*In [51], the complexity of bidirectional problems has been estimated by* $O(n \times w)$*, where* w *denotes the width of a flowgraph.*" This notion of width does not solely depend on the flowgraph structure, but also on the considered bit vector problem. In particular, and as stipulated in the same footnote, the width is "*larger for bidirectional problems than for unidirectional ones, and in the worst-case it is linear in the size of the flowgraph.*"

Roughly, LCM performs the following steps:

1. **Preparing the CFG (§10.2.1):** in order for the data-flow analysis to function correctly, some pre-processing of the CFG is necessary;
2. **Detecting candidates (§10.2.2):** by examining each basic block, the algorithm detects all candidate expressions that might be moved or reduced;
3. **Computing local and global predicates (§10.2.3);**
4. **Effectively rewriting the graph (§10.2.4):** depending on the results of the analyses for each candidate expression, we apply the needed transformations in the code.

The original papers from Knoop et al. focus on explaining and proving analyses, and leave out many details. I do not repeat the proof work here, nor the detailed explanation of the data-flow equations, except on the points where they differ from the original papers. Please refer to those papers for details. Fundamentally, translation validation saves us from having to formalize in Coq the theory of these algorithms—described in those papers, which seems to be a very substantial task.

Apart from that, the authors mentioned the preparation, detection, and rewriting phases (points 1., 2., and 4. above) but, since they depend on the intermediate representation, their technical implementation is not covered. In this section, we specify all four steps in the frame of BTL. Furthermore, §10.2.5 suggests an alternative, more efficient way of managing candidates; §10.2.6 explains how we adapted LCM to our validation constraints; and §10.2.7 exemplifies the code motion part of LCT on a realistic example.

[4] Which performs a loop-invariant code motion and common subexpression elimination in a single computation.

10.2.1 *Prerequisites for the CFG*

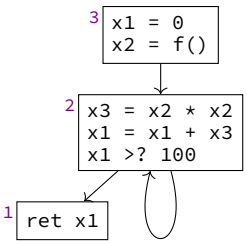
In their implementation paper, Knoop et al. presuppose a CFG with *a single exit point*. However, in practice, some embedded programs are modeled using an infinite loop; thus, their main function contains no exit point. Moreover, a C function might contain multiple return statements. Hence, we assume working on a control-flow graph $G = (B, E, s)$ with B the set of nodes (basic blocks), E the set of edges, and s the unique entry point of the function. Contrary to Knoop et al., we do not impose any restriction on exit points. We note $succ(n) \triangleq \{ m \mid (n, m) \in E \}$ and $pred(n) \triangleq \{ m \mid (m, n) \in E \}$, the functions that give the sets of immediate successors and predecessors of node n, respectively.

10.2.1.1 *Insertion of Synthetic Nodes*

Another important constraint of LCM concerns *critical edges* [83, §2.2]: edges going from nodes with multiple successors to nodes with multiple predecessors. In some cases, such edges may block the code motion process. Rüthing [128] experimented an approach to still perform code motion in the presence of critical edges, while keeping unidirectional analyses. On the other hand, its conclusion does not encourage one to leave critical edges: "*Although this may sound like an advocacy for the non-splitting of critical edges it should kept in mind that the computational quality of expression motion in the absence of critical edges is unexcelled when compared to the situation where critical edges are present.*"

The common practice, applied in the LCM implementation paper, involves splitting critical edges by incorporating synthetic nodes, essentially empty blocks. In reality, within BTL, simply splitting critical edges proves to be insufficient: we illustrate this with an example below.

Example 10.2.1 (BTL has stronger constraints on critical edges)**.**
Consider the trivial CFG on the right, where `x3` is an argument of the function. LCM would want to lift the "`x2 * x2`" before the loop, as it is loop invariant. However, altering the result register of calls is prohibited by our validator, and by construction, it is unfeasible to have a call in the middle of a block[5]. Thus, in this example, we would be hindered by the constraints of our IR to perform the optimization. Additionally, splitting critical edges *would not suffice* here, as the only critical edge in this scenario is the back edge from block 2 to itself.



Our approach involves a more aggressive split: rather than merely cutting critical edges, we cut every edge that leads to a join point. In this case, block 2 is a join point for edges $(3, 2)$ and $(2, 2)$, meaning that our technique would introduce a synthetic node before the loop entry, and following block 3. Our oracle can then leverage this empty node to lift the invariant instruction without being blocked.

As well explained by Knoop, Ruthing, and Steffen [83, Lemma 2.1], a CFG devoid of critical edges invariably satisfies the following two properties: (i) $\forall n \in B, |pred(n)| \geqslant 2 \implies succ(pred(n)) = \{n\}$ and (ii) $\forall n \in B, |succ(n)| \geqslant 2 \implies pred(succ(n)) = \{n\}$.

The insertion of synthetic nodes is performed during the RTL to BTL translation validation pass (cf. §8.2.1). Synthetic nodes that remained unused for inserting a new computation are conveniently removed by the subsequent "tunneling" passes of CompCert [92, §9].

10.2.1.2 *Unrolling the CFG*

Proving the anticipation of trapping instructions would require an anticipability (a.k.a. inevitability) analysis during the validation (see §10.5.3).

Recall that our validator restricts the anticipation of trapping instructions like loads (point (iii) of §6.5.1). Indeed, we cannot anticipate a trapping instruction w.r.t. to the source, since doing so would *add a potential trap* in the code. Hence, we were compelled to modify LCM to prevent it from anticipating those instructions. Fortunately, we can still mitigate this limitation thanks to CFG restructurings (duplications), as the ones depicted in Figure 4.1 (and whose verification is outlined in Chapter 8). Example 10.2.2 shows how this technique applies in practice.

[5] In the BTL semantics, a call in the middle of a block exits the block (to a return block encoded in the instruction, cf. Figure 5.1). Any code that sequentially follows a call in the same block is actually unreachable (recall explanations in §5.2).

Regardless of this limitation to move trapping instructions, unrolling the CFG improves the LCM efficiency. In addition, factorizing the CFG (using the pass in §8.2.2) enables us to undo certain superfluous duplications (see the example of §10.2.7).

10.2.1.3 *Preliminary CSE and Conceptual Refinement of Basic Block Predicates*

In order for the basic block based LCM (and consequently LCT) to perform solely unidirectional analyses, it needs to conceptually split basic blocks into two parts, for each candidate. An *entry* part containing every instruction up to and including the last modification of the candidate's dependencies; and an *exit* part, consisting of all remaining statements [83, §2.3, Figure 4]. A block's entry part contains at most one computation of the candidate before the first modification of its dependencies. When such a computation exists, we denote it as the *entry computation*. The same applies for exit parts: they must also contain at most one computation of the candidate, namely the *exit computation*.

For these virtual, internal borders of basic blocks to be well-defined, LCM requires a prior local common subexpression elimination on each basic block (e.g. to avoid having more than one entry/exit computation). This was not a problem for us, thanks to the trivial CSE of CompCert, applied automatically before reaching BTL. This conceptual refinement of basic blocks into two parts[6] is used in two ways. First, to compute data-flow equation systems, which, as a result, feature *two equations*: one for each conceptual part. Second, to compute the *ideal insertion offset* for a new, anticipated computation within a block. In [83], the authors rely on this decomposition to formalize LCM, and they demonstrate that offset points computed with this method guarantee a locally minimal lifetime of registers.

The calculation of insertion offsets is detailed in §10.2.4.

10.2.2 *Detecting Code Motion Candidates*

We want the lazy code motion part of LCT to be *independent* of the target architecture. An important point here is to distinguish trapping instructions in our candidate model; the fact is some operations can be trapping on certain backends (e.g. divisions by zero), and there also exists backends where loads can be non-trapping (e.g. on KVX [134]). In the specific case of RISC-V, none of the BTL operations can fail, and loads are always trapping.

For our oracle to work with both operations and loads, we define a type representing right hand-sides (RHSs) of register assignments in Figure 10.2, that serves as a key to a hash table whose values, in Figure 10.3, record candidate information (bold fields are mutable). The candidates' hash table is defined as: "$\texttt{candidates} : cm\_key\_t \mapsto cand\_t$".

$$
\begin{aligned}
&\texttt{cm\_ckey\_t} ::= \\
&\quad \mid \texttt{CMop}(trap, op, \overrightarrow{reg_{\text{arg}}}) \\
&\quad \mid \texttt{CMload}(trap, chk, addr, \overrightarrow{reg_{\text{arg}}})
\end{aligned}
$$

Figure 10.2: Code Motion Candidates' Key Type.

$$
\begin{array}{lll}
\texttt{cand\_t} \triangleq \{ & & \\
\quad \textbf{lhs} & : pc \mapsto \mathrm{S}_{\texttt{ofs}}; & \text{Left hand-sides positions} \\
\quad \texttt{state} & : predicates; & \text{Record of bit vectors} \\
\quad \textbf{vaux} & : reg\ option; & \text{Candidate's fresh variable} \\
\quad \texttt{memdep} & : bool; & \text{Memory read dependency} \\
\quad \textbf{was\_reduced} & : bool; & \text{SR confirmation} \\
\quad \textbf{updated\_args} & : \overrightarrow{reg_{\text{arg}}}\ option; & \text{Substituted arguments} \\
\quad \texttt{orig\_args} & : \overrightarrow{reg_{\text{arg}}}; \} & \text{Original arguments}
\end{array}
$$

Figure 10.3: LCT Candidates' Value Type.

Thanks to this structure, we rebuild BTL instructions from candidates' keys, and match right hand-sides of instructions to existing candidates efficiently using the hashed key. Note that both constructors of Figure 10.2 include a Boolean to indicate if the candidate may trap or not. Furthermore, some operations are subject to a memory-read dependency that is not syntactically modeled by

Other fields of Figure 10.3 will be introduced as they are explained.

[6] Knoop, Ruthing, and Steffen [83, §2.3] refer to this concept as a "refined flowgraph" used only for reasoning.

BTL (encoded by the `memdep` Boolean of Figure 10.3). Predicates are stored in the `state` field of each candidate's record (Figure 10.3). They are encoded as bit vectors, such that each bit corresponds to a CFG node and indicates if the predicate holds or not.

The detection phase of our LCT traverses each block and inserts a new mapping for every operation or load. The `lhs`—stands for left hand-side (LHS)—field of Figure 10.3 is a map from block IDs (i.e. positions "pc" in the CFG) to sets of offsets (i.e. positions in a given block), recording the points where the candidate was seen (assigned to a LHS) in the CFG. So if a candidate is detected in multiple places, we simply update the `lhs` map with the new position. Once the detection phase is completed, candidates are sorted first by their appearance block, and second by their offset within that block. Sorting is made possible thanks to a prior post-order CFG renumbering, which also accelerates fixed point calculations. They are then handled one by one in this topological order by the oracle. Sorting candidates topologically is a *simple but powerful improvement* w.r.t. the original LCM: in particular, this technique is the first step towards the reduction of sequences like the one mentioned in §10.1.4. By processing *both* code motion and strength-reduction candidates in that order, one can avoid being blocked in optimizing a candidate because its dependencies were not treated before.

Yet, sorting candidates is not sufficient. We must also apply modifications iteratively: this is the topic of §10.2.5.

10.2.3 *Analyses*

The base of the LCT algorithm reproduces the same analyses as the basic block LCM. Every predicate P (except transparency, see below), concerns either the entry (noted $\lceil P \rceil$) or the exit (noted $\lfloor P \rfloor$) part of a block, w.r.t. the considered candidate instruction.

10.2.3.1 *Local Predicates*

Recall (from the beginning of this Section, §10.2), that "local" here means w.r.t. a block: the calculus of a local predicate only depends on the considered block.

The simplest local information to calculate is transparency, denoted as `TRANSP`. This predicate relates to *an entire basic block* and is true when the dependencies of the candidate under consideration are not clobbered within the block. If a candidate's input register, or memory (in case the candidate depends on it), is modified in the block, `TRANSP` is false.

The other two local predicates, namely "$\lceil$`COMP`$\rceil$" and "$\lfloor$`COMP`$\rfloor$", are variations of the local anticipability and availability of Morel and Renvoise [111], respectively. These predicates, instead of considering these properties across an entire basic block (such as those of Morel and Renvoise), utilize the conceptual entry and exit parts. By definition, each part can contain at most one occurrence of the candidate. If the targeted candidate appears in the entry part of a block, it indicates that it is also locally anticipable (i.e. its dependencies are not modified before the occurrence). Symmetrically, if it is present in the exit part, we know that it is locally available (i.e. its dependencies are not modified after the occurrence). Therefore, "$\lceil$`COMP`$\rceil$" and "$\lfloor$`COMP`$\rfloor$" simply hold when the candidate is computed in the entry and exit parts, respectively.

Obtaining the three local predicates described here is easily done by running through each block, for each candidate.

10.2.3.2 *Global Predicates*

Data-flow predicates are denoted with a prefix "↑" or "↓" according to whether they require a backward or forward analysis, respectively. Predicates without any prefix in front of their names are not data-flow based (like local properties above, but there are also global, non data-flow predicates). The finite conjunction and disjunction over sets of successors and predecessors (from functions *succ* and *pred*) are noted[7] with "$\prod$" and "$\sum$", respectively. The negation of a predicate P is noted $\overline{P}$. All data-flow predicates' bit vectors are initialized to *false*. Applied over an empty set, $\prod$ is always true, while $\sum$ is always false. Recall that "s" is the entry point of the graph.

Below is the list of all global properties solved by the LCM. Note that we did not adapt those equations in the LCT; I only provide them for completeness of this document.

[7] Reusing the notations of Knoop, Ruthing, and Steffen [83].

UP-SAFETY: (noted U-SAFE, a.k.a. availability in [111]) holds if a computation at node n does not introduce a new value for every path *leading* at n; it is computed by forward data-flow analysis:

$$\downarrow \begin{cases} \lceil \texttt{U-SAFE} \rceil(n) \triangleq \begin{cases} \mathit{false} \text{ if } n = s \\ \prod_{m \in pred(n)} (\lfloor \texttt{COMP} \rfloor(m) \vee \lfloor \texttt{U-SAFE} \rfloor(m)) \end{cases} \\ \lfloor \texttt{U-SAFE} \rfloor(n) \triangleq \texttt{TRANSP}(n) \wedge (\lceil \texttt{COMP} \rceil(n) \vee \lceil \texttt{U-SAFE} \rceil(n)) \end{cases}$$

DOWN-SAFETY: (noted D-SAFE, a.k.a. anticipability in [111]) holds if a computation at node n does not introduce a new value for every path *starting* from n; computed backward:

$$\uparrow \begin{cases} \lceil \texttt{D-SAFE} \rceil(n) \triangleq \lceil \texttt{COMP} \rceil(n) \vee \texttt{TRANSP}(n) \wedge \lfloor \texttt{D-SAFE} \rfloor(n) \\ \lfloor \texttt{D-SAFE} \rfloor(n) \triangleq \lfloor \texttt{COMP} \rfloor(n) \vee \prod_{m \in succ(n)} (\lceil \texttt{D-SAFE} \rceil(m)) \end{cases}$$

EARLIESTNESS: (EARL) is true if the candidate cannot be safely placed earlier; it does *not* require a data-flow analysis and is obtained from safety and transparency properties:

$$\begin{cases} \lceil \texttt{EARL} \rceil(n) \triangleq \lceil \texttt{D-SAFE} \rceil(n) \wedge \\ \qquad \prod_{m \in pred(n)} (\overline{\lfloor \texttt{U-SAFE} \rfloor(m) \vee \lfloor \texttt{D-SAFE} \rfloor(m)}) \\ \lfloor \texttt{EARL} \rfloor(n) \triangleq \lfloor \texttt{D-SAFE} \rfloor(n) \wedge \overline{\texttt{TRANSP}(n)} \end{cases}$$

DELAYABILITY: (DELAY) encodes the possibility to safely move the inserted value from its earliest down-safe point (i.e. the computation is *delayable*); computed forward:

$$\downarrow \begin{cases} \lceil \texttt{DELAY} \rceil(n) \triangleq \lceil \texttt{EARL} \rceil(n) \vee \begin{cases} \mathit{false} \text{ if } n = s \\ \prod_{m \in pred(n)} \begin{pmatrix} \overline{\lfloor \texttt{COMP} \rfloor(m)} \wedge \\ \lfloor \texttt{DELAY} \rfloor(m) \end{pmatrix} \end{cases} \\ \lfloor \texttt{DELAY} \rfloor(n) \triangleq \lfloor \texttt{EARL} \rfloor(n) \vee \lceil \texttt{DELAY} \rceil(n) \wedge \overline{\lceil \texttt{COMP} \rceil(n)} \end{cases}$$

LATESTNESS: (LATEST) represents the optimality of delayability, the maximum delay; without data-flow from the latter:

$$\begin{cases} \lceil \texttt{LATEST} \rceil(n) \triangleq \lceil \texttt{DELAY} \rceil(n) \wedge \lceil \texttt{COMP} \rceil(n) \\ \lfloor \texttt{LATEST} \rfloor(n) \triangleq \lfloor \texttt{DELAY} \rfloor(n) \wedge \begin{pmatrix} \lfloor \texttt{COMP} \rfloor(n) \vee \\ \sum_{m \in succ(n)} (\overline{\lceil \texttt{DELAY} \rceil(m)}) \end{pmatrix} \end{cases}$$

We noticed a minor discrepancy in the isolation equation system from [83, Table 7]: the entry part equation is missing the "$\overline{\lfloor \texttt{COMP} \rfloor(n)}$" clause, likely a typo, as it is not reflected in the authors' proof.

ISOLATION: (ISOL) detects the case where a computation inserted at a given node would only be used (i.e. isolated) in this node; computed backward:

$$\uparrow \begin{cases} \lceil \texttt{ISOL} \rceil(n) \triangleq \lfloor \texttt{EARL} \rfloor(n) \vee \overline{\lfloor \texttt{COMP} \rfloor(n)} \wedge \lfloor \texttt{ISOL} \rfloor(n) \\ \lfloor \texttt{ISOL} \rfloor(n) \triangleq \prod_{m \in succ(n)} \begin{pmatrix} \lceil \texttt{EARL} \rceil(m) \vee \\ \overline{\lceil \texttt{COMP} \rceil(m)} \wedge \lceil \texttt{ISOL} \rceil(m) \end{pmatrix} \end{cases}$$

NOTE ON THE IMPLEMENTATION The whole LCT is implemented in OCaml. To compute data-flow equations[8], we use a polymorphic work-list based algorithm that solves any of the above data-flow equation systems. The code of this fixed point solver is accessible here [◇]. For non data-flow predicates, we apply efficient logical bitwise operations over bit vectors.

10.2.3.3 *Insertion and Replacement Points*

Finally, the LCM insertion and replacement points (IR-points) are deduced using the formulas below[9]:

[8] Which are encoded as bit vectors using the "Bitv" library of Jean-Christophe Filliâtre: https://github.com/backtracking/bitv.

[9] Either for entry or exit parts by substituting predicates accordingly.

- $\texttt{INSERT}(\texttt{pc}) \triangleq \texttt{LATEST}(\texttt{pc}) \wedge \overline{\texttt{ISOL}(\texttt{pc})}$
- $\texttt{REPLACE}(\texttt{pc}) \triangleq \texttt{COMP}(\texttt{pc}) \wedge (\overline{\texttt{LATEST}(\texttt{pc}) \wedge \texttt{ISOL}(\texttt{pc})})$

When INSERT is true, the candidate is stored in its allocated, unique auxiliary variable (i.e. fresh register), specified in the vaux field of Figure 10.3. In every node marked as REPLACE, LCM replaces the candidate with a move from vaux into the original destination.

10.2.4 *Insertion Offset and Forward Propagation*

We compute insertion offsets of candidates within blocks exactly as LCM [83, §2.4], but our replacement method (i.e. for nodes satisfying REPLACE for a given candidate) is more general.

10.2.4.1 *Finding the Internal Insertion Offset*

Given a candidate c that must be inserted at node n, we encounter two scenarios: c must either be inserted in the entry part (if $\lceil\texttt{INSERT}\rceil(n)$) or in the exit part (if $\lfloor\texttt{INSERT}\rfloor(n)$). In the former situation, the optimal offset is immediately before the entry computation, if such computation exists. If not, it is immediately before the first modification of the candidate's dependencies, if applicable. If there is neither an entry computation nor a modification, then it is at the end of the entry part. For the latter situation, when inserting in the exit part, the optimal offset is immediately before the exit computation if applicable, or at the end of the exit part otherwise.

10.2.4.2 *Substituting Auxiliary Variables and Delaying the Move*

In the case of a sequence of instructions, the ordered treatment of candidates would cause LCT to lift the first instruction of the sequence. Nevertheless, following the LCM method for replacing candidates, we would replace the original occurrence with a move, and then be blocked by the dependencies to handle the second candidate.

To avoid that, we propose a new algorithm to substitute a candidate's auxiliary variable within the subsequent instructions, and delay the move insertion as late as possible in the block (we describe it in §10.3.4). When the auxiliary variable is not used anywhere after substitution, and is not live in the successor blocks, the move is even removed by dead code elimination. This improvement requires adapting the LCM stages compared with the four steps at the beginning of this section (§10.2). Combined with the topological treatment of candidates mentioned earlier, our LCT becomes capable of reducing instruction sequences, and thus improves CompCert's instruction selection. This substitution process unlocks many new instruction movements that were previously inaccessible, for *both* CM and SR.

10.2.5 *An Iterative Treatment of Candidates*

In fact, instead of computing predicates (step 3.) for all candidates, and then rewrite the CFG for all of them in step 4., we loop through steps 3. and 4. for each candidate (again, recall the four LCM steps). In other words, we perform iteratively the analysis and rewriting steps, candidate by candidate. Hence, when a candidate is moved, its newly inserted occurrence and the substitution of its auxiliary variable are effectively written into the CFG. This will then benefit upcoming candidates, whose analysis phase will take previous changes into account.

After having detected and sorted candidates from the hash table as a list of (key, value) pairs, our LCT repeats the four steps below for each pair.

1. *Update* the current candidate: if its original arguments (stored in the orig_args field of Figure 10.3) were modified by previous substitutions (through the updated_args field), then its key (cf. Figure 10.2) is modified with the substituted arguments[10].
2. Predicates' initialization (with their default values) and *local analysis*;

[10] We replace the candidate in the hash table (and take the updated key in the sorted list). If a candidate with the new arguments already exists, the new one is *merged* with the old one (by unifying their lhs fields).

3. *Data-flow analysis*;
4. *Rewrite the CFG* if the candidate was moved.

10.2.6 *The Case of Trapping Instructions*

IR-points of §10.2.3.3 let LCM anticipates trapping instructions, while our validator only allows one to move them if they were already computed before in the source.

10.2.6.1 *Restricting IR-Points for Trapping Instructions*

We sketch a *restrictive algorithm* to calculate IR-points for trapping instructions. The idea is to start by computing the set of block IDs where we may *replace* a trapping candidate[11]. We traverse the CFG from its entry point s, and remember each block ID satisfying two *necessary* conditions: (i) the candidate appears in the entry part; and (ii) the entry part is "up-safe". Indeed, as stated by (i), we cannot eliminate a trapping instruction if its dependencies are modified: this means that replaceable trapping candidates are located in entry parts of blocks. Point (ii) reflects the availability condition (we cannot eliminate an unavailable computation). The result is returned by the COMPUTE_POT_REP(s) function as a set $P = \{pc_p \mid \lceil \texttt{COMP} \rceil(pc_p) \wedge \lceil \texttt{U-SAFE} \rceil(pc_p)\}$ in Algorithm 1.

From there, we need to ensure that these points are actually reachable from a previous calculation of the candidate. For a given $pc_p \in P$, FILTER_COMP_BLOCKS(pc_p) finds the set I of available previous calculations (e.g. usable to factorize the candidate). It is defined as the set of pc_i such that $pc_i \neq pc_p \wedge (\lceil \texttt{COMP} \rceil(pc_i) \vee \lfloor \texttt{COMP} \rfloor(pc_i))$, and such that *there exists a path* from pc_i to pc_p preserving the transparency property of the candidate. Thus, I groups nodes where we should insert *and* replace the candidate.

Algorithm 1 Insertion and Replacement Points for Trapping Instructions [◇].

```
1:  procedure COMPUTE_LCM_TARGETS_TRAP(s : pc, cand : cand_t)
2:      st ← cand.state
3:      P ← COMPUTE_POT_REP(s)
4:      for pc_p ∈ P do
5:          I ← FILTER_COMP_BLOCKS(pc_p)
6:          if |I| > 0 then
7:              st.⌈REPLACE⌉(pc_p) ← true
8:              if st.⌊COMP⌋(pc_p) then
9:                  st.⌊INSERT⌋(pc_p) ← true
10:                 st.⌊REPLACE⌋(pc_p) ← true
11:         for pc_i ∈ I do
12:             if st.⌈COMP⌉(pc_i) then
13:                 st.⌈INSERT⌉(pc_i) ← true
14:                 st.⌈REPLACE⌉(pc_i) ← true
15:             else if st.⌊COMP⌋(pc_i) then
16:                 st.⌊INSERT⌋(pc_i) ← true
17:                 st.⌊REPLACE⌋(pc_i) ← true
```

Using those two functions, we define in Algorithm 1 the main procedure used to compute restricted variants of `INSERT` and `REPLACE`. When $I = \emptyset$, we abandon the potential replacement in block at pc_p (equivalently to the isolation predicate). Otherwise, $\lceil \texttt{REPLACE} \rceil(pc_p)$ is set to true. Moreover, if the block also contains an exit computation of the candidate, then the latter must be saved into its auxiliary variable[12] (lines 8-10 in Algorithm 1). Finally (lines 11-17), both `INSERT` and `REPLACE` predicates are set to true for all $pc_I \in I$ (we set their entry variant if the node has an entry computation, and their exit one otherwise).

[11] We only *factorize* trapping candidates with a previous computation of the instruction, so insertion points are also replacement points.

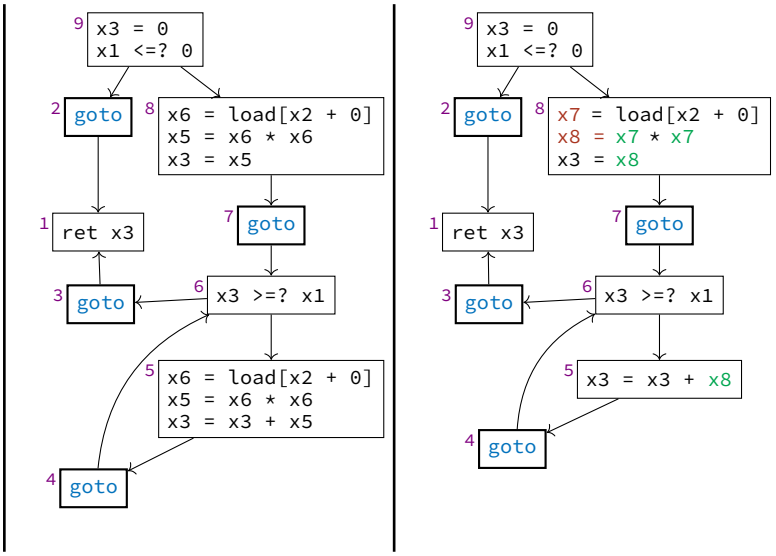
[12] The candidate being present in both block parts, it probably needs to be recalculated: otherwise it would have been removed by the CompCert's CSE beforehand.

10.2.6.2 *Restructuring the CFG to Enable Load Anticipation*

A code duplication pass was implemented in CompCert by Six [133, §6]. The objective was initially to lengthen superblocks, in order to widen the scope of scheduling. Nonetheless, Monniaux and Six [109] suggested a very interesting alternate use of this pass. Their paper presents a common subexpression elimination algorithm, named "CSE3", that leverages the code duplication pass to perform loop-invariant code motion. Our LCT uses the same technique to circumvent the anticipation limitation of §10.2.1.2, as exemplified below:

Example 10.2.2 (Unrolling the first loop iteration to lift loop invariant trapping instructions)**.** Let us consider the C source code depicted below on the left. We ran a loop peeling (on the first iteration) prior to applying LCT, as shown with the BTL CFG before optimization in the middle column. The result, after executing LCT, is visible in the right column. Synthetic nodes are framed with a thick border. All nodes are numbered—in violet—in a post-order fashion. New assignments to fresh variables are denoted in red, and compensation code or substitutions in green.

```
int foo(int *x, int n) {
  int s = 0;
  while (s < n)
    s += *x * *x;
  return s;
}
```



Observe what happened here: the load of the source variable `x` was duplicated by loop unrolling. Its first occurrence, in block 8, is now outside the loop. Thanks to Algorithm 1, LCT ensures that the load of the loop body—in block 5—is reachable from the duplicated one. Then, it allocates a fresh pseudo-register `x7` to save the value of the duplicated load, and eliminate the redundancy. Leveraging the substitution method of §10.2.4.2 to delay the move, and the iterative process of §10.2.5, the elimination of the load leads to an intermediate state of the loop body "`x5 = x7 * x7; x3 = x3 + x5; x6 = x7`". As expected, in this intermediate code, `x7` was substituted in the multiplication (replacing `x6`) and the move "`x6 = x7`" was delayed at the end of the block.

In addition to this simplification of the load, the topologically ordered treatment of candidates allows us to continue the process and eliminate the "`*x * *x`" calculation (i.e. "`x7 * x7`" in the intermediate code), which is also loop invariant! Hence, fresh register `x7` is used to compute the multiplication into fresh register `x8` in block 8, out of the loop. Again, the substitution is applied, so that variable `x8` is used in place of `x5` in the addition "`x3 = x3 + x8`" of the body. The move from `x8` to `x5` is also delayed at the end, but, since both the `x7` and `x8` moves are dead at this point, they are removed by the subsequent DCE pass (and they do not appear in the right column above).

The next section explains why LCT is more powerful than CSE3 to perform that kind of anticipation.

It should be noted that to make this optimization evident, we had to disable CSE3; otherwise, CSE3 would have eliminated the load redundancy before LCT could in this example.

10.2.7 *An LCT Example of Code Motion*

We present below an application of our formally verified LCT: §10.2.7.1 shows how it optimizes the example in Figure 10.4; §10.2.7.2 details the validation of our LCT oracle on this example; and §10.2.7.3 provides additional information on the computed predicates for one of the example's candidates.

10.2.7.1 *Performing LICM by PRE*

Figure 10.5 presents an extract of the RISC-V code produced by CompCert with CSE3 of [109, 110] activated for the source C code in Figure 10.4. The computation of `a[0]` has been factorized in register `f3` over the whole program. But, computations (in red color) of `a[1]` in `f0`, `a[2]` in `f2`, and loading of floating-point `7` in `f1` are performed at each iteration of the loop of label `.L102`.

```
double approx(double *a) {
  double r = 2;
  if (a[0] < 2) return 2;
  while (r < a[1])
    if (r >= a[2]) r -= a[0];
    else r *= 7;
  return r;
}
```

Figure 10.4: Four Candidates for LICM.

In contrast, in Figure 10.6 (the four first lines are omitted because identical), after unrolling the first iteration, our LCT moves all these computations before the loop, starting now at label `.L106`. Notice that if the condition of the loop is initially false, `a[2]` is not computed by the original loop, but may trap if the address is invalid. Thus, unrolling is necessary here to anticipate the computation of `a[2]` before the loop. However, it may not suffice. For example, if the test "`a[0] < 2`" was omitted, then simply unrolling the first iteration would not suffice to allow `a[0]` to be moved before the loop. Indeed, if "`r < a[2]`" at the first iteration, then `a[0]` is not computed and may still trap afterward. Actually, following Bodík, Gupta, and Soffa [22], we may find an unrolling (validated by the CFG morphism checker of Chapter 8) that enables it. But this would cost even more code duplications than those of Figure 10.6.

```
  fld    f3,0(x10)
  fld    f10,.L100,x31
  flt.d  x31,f3,f10
  bne    x31,x0,.L101
.L102: ; Loop Entry
  fld    f0,8(x10)
  flt.d  x31,f10,f0
  beq    x31,x0,.L101
  fld    f2,16(x10)
  fle.d  x31,f2,f10
  bne    x31,x0,.L103
  fld    f1,.L104,x31
  fmul.d f10,f10,f1
  j      .L102
.L103:
  fsub.d  f10,f10,f3
  j      .L102
.L104: ...; 7.0 in hexa
.L100: ...; 2.0 in hexa
```

Figure 10.5: CSE3 Alone.

```
  ... ; Same prolog
  fld    f0,8(x10)
  flt.d  x31,f10,f0
  beq    x31,x0,.L101
  fld    f2,16(x10)
  fle.d  x31,f2,f10
  bne    x31,x0,.L103
  fld    f10,.L105,x31
  j      .L102
.L103:
  fsub.d f10,f10,f3
.L102:
  fld    f1,.L104,x31
.L106: ; Loop Entry
  flt.d  x31,f10,f0
  beq    x31,x0,.L101
  fle.d  x31,f2,f10
  bne    x31,x0,.L107
  fmul.d f10,f10,f1
  j      .L106
.L107:
  fsub.d f10,f10,f3
  j      .L106
.L105: ...; 14.0 in hexa
.L104: ...; 7.0 in hexa
```

Figure 10.6: Unroll+LCT.

Let us now explain why LCT is more powerful than CSE. Applying CSE3 after unrolling produces almost the same code as the one of Figure 10.6 except that the load of floating-point `7` is not factorized[13]. This is due to the fact that some execution path of the first iteration does not load floating-point `7` into `f1`. Indeed, CSE3 can only eliminate computations that are available on all incoming paths. Thus, CSE3 only performs some *full redundancy elimination*: it misses FRE if the same value is available on different incoming paths, but in different registers (because unlike LCT,

[13]The original CSE of CompCert does not even eliminate the redundant "`a[0]`". This contrast with "`gcc -O1`" (version 9.4.0) which performs a PRE with slightly less code duplications than ours on this example. However, the original CSE of CompCert factorizes the load of floating-point 2 into register "`f10`".

CSE3 does not introduce any fresh register). In contrast, LCT is able to perform any FRE and even non-trapping PRE *without unrolling*. On Figure 10.4 example, the load of floating-point 7 is anticipated even without any loop unrolling. In the original version of [83], LCT also safely moves `a[1]` out of the loop without any loop transformation: this is a FRE, since `a[1]` is present in the condition of the loop, which is at least run once. Nevertheless, since our simulation test forbids the anticipation of trapping code, our LCT can only eliminate `a[1]` within the loop, after at least a loop rotation (see Figure 4.1). This is not an issue on this very simple example: thanks to the CFG minimization applied when returning to RTL (cf. §8.2.2), we still finally achieve the FRE of `a[1]` without any code duplication.

10.2.7.2 *Validating LCT on Our Example*

The CFG of the example is represented in Figure 10.7, using the same color code as in Example 10.2.2. The four candidates detected by the oracle have been inserted at their optimal points, by assigning them to a fresh variable.

For instance, the load of floating-point 7 illustrates this "lazy" behavior of LCT as it is inserted in two different blocks (14 and 16) to minimize the live range. The calculation is therefore duplicated on two branches, and both chosen blocks are the last possible ones before the loop. However, this code duplication does not appear in the final assembly code of Figure 10.6, because it is factorized by our subsequent CFG minimization pass (cf. §8.2.2). In Figure 10.7, it also appears that some fresh variables, such as `x10` and `x11`, are duplicated through a compensation move (in green). Note that these pseudo-register duplications do not increase the actual live range since they will be removed by the subsequent register allocator.

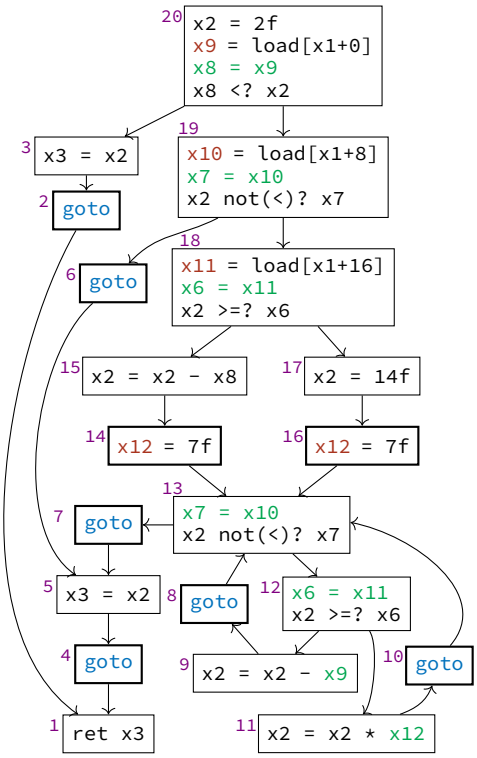


Figure 10.7: Full BTL CFG of Figure 10.6.

Now let us detail which invariants are generated by our LCT oracle before sending the whole result to the verifier. Code motion only requires gluing invariants (GIs): thus, history invariants (HIs) remain trivially empty here. Invariants are generated in their sequential representation (cf. §6.2.2.2). As explained in §6.3.3.4, the invariant of the entry block (here block 20) is always reduced to a liveness set. Besides liveness sets, invariants are updated for each candidate just after they appear in code: at the entry of block 19, we have "`([x9 ≔ load[x1+0]], {x1,x2,x8,x9})`" to remember the load, and because these four variables are live. The second (respectively third) load is added to the gluing invariant at block 18 (respectively blocks 15 and 17). Thus, for all blocks with a label in 14 . . . 17, the invariant contains the same sequence of assignments (but the sets of live variables are different): "`x9 ≔ load[x1+0]; x10 ≔ load[x1+8]; x11 ≔ load[x1+16]`". From block 13 and down to block 8 (included), we append to this list the assignment "`x12 ≔ 7f`". Finally, blocks 1 to 7 only contain pure liveness invariants, as the verification need not remember the values of candidates anymore.

Example 10.2.3 (Validating symbolic simulation for block 14)**.**
$\mathcal{G}(14) =$ `([x9 ≔ load[x1+0]; x10 ≔ load[x1+8]; x11 ≔ load[x1+16]], {x2,x9,x10,x11})`
This leads to the following initial states:

$$\mathtt{sis_s} = \{\mathtt{sis\_pre} = \varphi;\ \mathtt{sis\_sreg} = \lambda \mathtt{r}.\,\mathtt{None};\ \mathtt{sis\_smem} = \mathtt{Sinit}\}$$
$$\mathtt{sis_t} = \{\mathtt{sis\_pre} = \varphi;\ \mathtt{sis\_sreg} = \mathtt{x2 \coloneqq x2 \parallel x9 \coloneqq load[x1+0] \parallel x10 \coloneqq load[x1+8] \parallel x11 \coloneqq load[x1+16]};\ \mathtt{sis\_smem} = \mathtt{Sinit}\}$$
$$\text{where } \varphi = \mathrm{OK}(\mathtt{load[x1+0]}) \wedge \mathrm{OK}(\mathtt{load[x1+8]}) \wedge \mathrm{OK}(\mathtt{load[x1+16]})$$

After the symbolic execution of the blocks:

$$\mathtt{ib_s} = \mathtt{BF}(\mathtt{Bgoto}(13), \_) \qquad \text{and} \qquad \mathtt{ib_t} = \mathtt{Bseq}(\mathtt{Bop}(7\mathtt{f}, [\,], \mathtt{x12}, \_), \mathtt{BF}(\mathtt{Bgoto}(13), \_))$$

we obtain:

$$\mathtt{ss_s} = \mathtt{Sfinal}(\mathtt{sis_s}, \mathtt{Sgoto}(13))$$
$$\mathtt{ss_t} = \mathtt{Sfinal}((\varphi, \mathtt{x2} \coloneqq \mathtt{x2} \parallel \mathtt{x9} \coloneqq \mathtt{load[x1+0]} \parallel \ldots \parallel \mathtt{x12} \coloneqq \mathtt{7f}), \mathtt{Sgoto}(13))$$

The two decision trees have the same structure and their unique leaves match. In particular, the gluing invariant of the successor block 13 is satisfied:
$\mathcal{G}(13) = (\mathtt{[x9 \coloneqq load[x1+0]; x10 \coloneqq load[x1+8]; x11 \coloneqq load[x1+16]; x12 \coloneqq 7f]},$
$\mathtt{\{x2,x9,x10,x11,x12\}})$
For instance, the right hand-side expression for `x9` evaluates to `load[x1+0]` in $\mathtt{ss_s}$, which matches the value of `x9` in $\mathtt{ss_t}$.

10.2.7.3 *Observation of the LCT Predicates for One Candidate of Our Example*

Let us focus on the immediate (i.e. *not* memory-dependent) load of floating-point 7, relocated in blocks 14 and 16 in the optimized CFG of Figure 10.7. In the source CFG, its was originally in the entry of block 11, inside the loop (so that $\lceil$`COMP`$\rceil$ is true for block 11 and false otherwise, and $\lfloor$`COMP`$\rfloor$ is always false). Since it has no input registers, it is transparent in all blocks, and its earlier possible insertion point is the entry part of the entry block. Hence, $\lceil$`EARL`$\rceil$ is true for node 20 and false otherwise, and $\lfloor$`EARL`$\rfloor$ is always false. For all nodes n, the candidate's dependencies are never redefined in a path starting from n; so $\lceil$`D-SAFE`$\rceil$ and $\lfloor$`D-SAFE`$\rfloor$ are always true. In contrast, the candidate is defined on all paths leading to n only in node 10, and in the exit part of node 11, so $\lceil$`U-SAFE`$\rceil$ is true only for node 10 and $\lfloor$`U-SAFE`$\rfloor$ for nodes 10 and 11. Considering that we need the immediate for the comparison in node 13, the immediate load is not delayable (neither $\lceil$`DELAY`$\rceil$ nor $\lfloor$`DELAY`$\rfloor$) in the loop body; blocks 8 . . . 13. Consequently, its latest useful positions are the exit (the entry would not be latest) parts of nodes 14 and 16; so $\lfloor$`LATEST`$\rfloor$ is true for these blocks and false otherwise, and $\lceil$`LATEST`$\rceil$ is always false[14]. The isolation analysis is false (both in entry and exit parts) for all nodes whose ID is less than or equal to 7.

Therefore, the only conceptual parts of nodes that are latest but not isolated are the exit parts of blocks 14 and 16; so $\lceil$`INSERT`$\rceil$ is always false, and $\lfloor$`INSERT`$\rfloor$ true only for these nodes. Similarly, the only conceptual part which contains an occurrence, and is neither latest nor isolated is the entry of node 11; meaning that $\lceil$`REPLACE`$\rceil$ is true for this node and false otherwise, and that $\lfloor$`REPLACE`$\rfloor$ is always false.

10.3 LAZY STRENGTH-REDUCTION

Akin to Knoop, Rüthing, and Steffen [85], we refined our code motion into a strength-reduction algorithm. Lazy strength-reduction candidates are multiplications (in the broadest sense, including e.g. left shifts) of the form "$v \times c$", between a variable v and a constant c. The LSR principle is to weaken the LCM notion of transparency by considering that *additions* $v = v + c'$ with a constant c' do not break the transparency. Such additions are named injuring operations. Thus, multiplications are moved as if they were CM candidates. To compensate the effect of additions on v, the algorithm inserts *update* assignments: for a candidate relocated in auxiliary variable v', an addition "$v' = v' + (c \times c')$" may be inserted in each block containing an injuring addition.

In [85], Knoop et al. first introduce a "simple" LSR working with only three data-flow analyses. Then, they refine their—non basic block based—LSR in three stages to overcome some of its limitations:

1. **R1**: avoids inserting an update addition (i.e. an increment of the SR variable) if a multiplication (i.e. the candidate itself) must be inserted on the same path. In such cases, a multiplication might be saved at the cost of both a new multiplication, and an update addition, which is even worse than doing nothing.

2. **R2**: finds the "best" insertion point (for the multiplication), considering lifetime using the delay, latest, and isolation analyses.

[14] I say "useful positions" because in fact, $\lceil$`DELAY`$\rceil$ and $\lfloor$`DELAY`$\rfloor$ are also true for nodes 2, 3, 6, and 14 . . . 20, and $\lfloor$`LATEST`$\rfloor$ is true for nodes 2 and 6. See how the isolation equations exclude these nodes.

3. **R3**: avoids having multiple update additions on the same path for the same variable by accumulating them into a single update.

R1 and R2 concern insertion and replacement points: the former finds substitutes for the original insertion points (without changing replacement points); and the three additional analyses of the latter minimize the live range induced by code motion. R3 does not change IR-points, but tries to accumulate update assignments. Knoop, Rüthing, and Steffen [85, §3.1.3] first compute a naive code motion (where INSERT = D-SAFE ∧ EARL), and then apply R1, R2, and R3 (in that order). Our basic block implementation of §10.2 (inspired from Knoop, Ruthing, and Steffen [83]), includes R2 *without* R1 (noted R2$^\flat$) "for free" by unifying the code motion part of predicate inference[15]. Rather than using R3 directly, we suggest an alternative, generic representation based on affine forms (as in the rewriting engine of §7.6.2). Last, we exploit the new substitution technique briefly described in §10.2.4.2 for CM to propagate results locally from previous iterations, as a fourth refinement R4 (coming after R3).

Unlike the original LSR, we refine our code motion LCT as follows:
R2$^\flat$ (§10.3.1) → R1 (§10.3.2.1) → R2 (§10.3.2.2) → *alternative* R3(§10.3.3) → *new* R4 (§10.3.4).
A complete example of LCT with both code motion and strength-reduction is given in §10.3.5.

10.3.1 *Extending Our LCT to Integrate the* R2$^\flat$ *LSR*

We extend our lazy code transformations algorithm of §10.2, which only includes LCM, to perform both algorithms together. The `state` field (Figure 10.3) now contains either the "real" transparency (TRANSP) for code motion candidates, or the weak transparency (noted SR-TRANSP) for strength-reduction ones. Specific SR predicates are also stored in `state`.

10.3.1.1 *Constants and Injuring Predicate*

As LSR targets multiplications with a constant, we perform a simple *constant detection* before the candidate detection of §10.2.2. This small preliminary step builds a hash map $reg \mapsto (pc, op)$ from destination registers (left hand-sides), to a pair containing the constant immediate load of type op and its position (i.e. its block ID of type pc). An instruction is considered constant as long as its destination register is never rewritten[16].

In the local analysis, we add an "injuring" predicate (noted INJURED) to the `state` being true when an argument is only "injured" by an additive operation (preserving SR-TRANSP). The TRANSP predicate (still needed in R1) is rebuilt trivially knowing that TRANSP = SR-TRANSP ∧ $\overline{\text{INJURED}}$. Executing the analyses of §10.2.3 with this notion of weak transparency gives us the R2$^\flat$ IR-points.

10.3.1.2 *Additive Candidates for Strength-Reduction*

Our LCT, in contrast to LSR, considers additions as strength-reduction candidates, as long as: (i) they do not modify the input of a multiplicative candidate (but they can modify its output), and (ii) they operate over the result of another SR candidate. Example 10.3.1 highlights the difference between multiplicative and additive SR candidates, and the separation with LCM-only candidates. In practice, we encapsulate the type of Figure 10.2 with the new candidate key defined in Figure 10.8.

```
sr_t    ::=  SRmul | SRadd
ckey_t  ::=  CCM(cm_ckey_t)
         |   CSR(sr_t, op, reg_arg⃗)
```

Figure 10.8: LCT Candidates' Key Type.

Example 10.3.1 (Sequence reduction of a multiplicative candidate and an additive candidate)**.** CompCert's instruction selection tries to *decompose* multiplications into a sequence of one or two left shifts (powers of two). When there are two shifts, an addition of their results is appended to the sequence. Consider the decomposition "$x1 = 5 \times x2$" into "$x3 = x2 \ll 2;\ x1 = x2 + x3$", and assume

[15] In other words, our algorithm naturally includes (and infers) R2$^\flat$, while R1 is calculated only when necessary, and after R2$^\flat$.

[16] If the same constant is assigned several times to a given register, we keep the oldest (topologically) occurrence.

it is inside a loop with an injuring increment over x2 (i.e. $x2 = x2 + 1$). The reduction starts by lifting the shift out of the loop in an auxiliary variable xA, and inserts an update assignment $xA = xA + 4$ just before the increment.

Then, we improve this first transformation by noticing that in most cases, the shift's intermediate result is only used to compute the addition. If applicable, we thus lift the addition too using an auxiliary variable xA', and apply the update (i.e. adding 5, as "$4 \cdot 1 \cdot x2 + x2 = 5 \cdot x2$") on xA'.

Hence, LCT is capable of strength-reducing additions that follow an already reduced multiplication. Other additions are "downgraded" to a normal code motion candidate. After updating the arguments of the candidate to optimize in step 1. of §10.2.5, we check if: (i) the candidate is an addition; (ii) none of its input registers is an auxiliary variable of a previously reduced multiplicative candidate. Indeed, thanks to the replacement with substitutions of §10.2.4.2, if an additive candidate has for argument a reduced multiplication, then its corresponding register should have been substituted with the multiplication's fresh variable. If not, we know that the addition does not operate with any SR multiplication, and can be relegated to a simple code motion candidate.

Compensatory additions (i.e. updates assignments) are always inserted in blocks containing an injuring operation on the candidate, but not necessarily in all of them. More specifically, an injured node must receive an update assignment either if it contains an occurrence of the candidate (whether in its entry or exit part), or if it has at least one successor not marked as an insertion point (for both the entry and exit parts) but identified as an update point. These blocks are characterized by the least solution of the below equation, which covers a whole basic block:

$$\uparrow \texttt{UPDATE}(n) \triangleq \begin{array}{l} \lceil \texttt{COMP} \rceil(n) \vee \lfloor \texttt{COMP} \rfloor(n) \vee \\ \sum_{m \in succ(n)} \left(\begin{array}{l} \overline{\lceil \texttt{INSERT} \rceil(m)} \wedge \overline{\lfloor \texttt{INSERT} \rfloor(m)} \\ \wedge \texttt{UPDATE}(m) \end{array} \right) \end{array}$$

After having inserted and replaced candidates, LSR inserts update additions in every node satisfying both `INJURED` *and* `UPDATE`.

10.3.2 *Generalizing LSR on Basic Blocks*

Remind Example 5.1.1: a multiplication inside the loop was replaced by an addition. To keep the code correct, the multiplication had to be inserted before the loop. In some complex cases (e.g. the nested loops of Knoop, Rüthing, and Steffen [85, Figure 3]), such an insertion of the multiplication may itself need to be compensated by an addition. This is precisely what R1 seeks to avoid: not placing a multiplication too early, so that a supplementary addition is unnecessary. The applicability of R1 depends on the candidate kind: since additive strength-reduction candidates are always preceded by a multiplication (otherwise we downgrade them as code motion candidates), they do not require R1. The latter is therefore only computed for multiplicative SR candidates.

Technically, the first refinement (R1) of Knoop, Rüthing, and Steffen [85] computes a set of critical points from which *there exists a path with no other occurrence of the candidate before the injuring operation*. Then, critical-insertion points are *both* critical and marked as insertion (in the sense of $R2^\flat$), and represent places where the "naive" (without R1) LSR would place both a multiplication *and* an update assignment on the same path. To optimize this inefficiency, the authors define a new predicate substitution-critical that encodes the set of *substitutes* (i.e. alternatives) of critical-insertion points. Intuitively, R1 *delays* critical-insertion points until their first reachable, non-critical point.

10.3.2.1 *New Data-Flow Equations for* R1

SR additive candidates keep $R2^\flat$ IR-points from the first step; while the `state` of `SRmul` candidates is extended with results of R1. Our method to compute R1 on top of $R2^\flat$ leads to insertion points equivalent to the original R2. We adapted the original (backward) "critical" predicate below:

$$\uparrow \texttt{CRIT}(n) \triangleq \overline{\texttt{COMP}(n)} \wedge (\overline{\texttt{TRANSP}(n)} \vee \sum_{m \in succ(n)} (\texttt{CRIT}(m)))$$

to a basic blocks based analysis by splitting it into:

$$\uparrow \begin{cases} \lceil \texttt{CRIT} \rceil(n) \triangleq \overline{\lceil \texttt{COMP} \rceil(n)} \wedge (\overline{\texttt{TRANSP}(n)} \vee \lfloor \texttt{CRIT} \rfloor(n)) \\ \lfloor \texttt{CRIT} \rfloor(n) \triangleq \overline{\lfloor \texttt{COMP} \rfloor(n)} \wedge \sum_{m \in succ(n)} (\lceil \texttt{CRIT} \rceil(m)) \end{cases}$$

Deducing the above equations is straightforward; the first step is to duplicate the original predicate in two variants with $\lceil \texttt{COMP} \rceil$ and $\lfloor \texttt{COMP} \rfloor$. Since it must be solved backward (i.e. it depends on the successor relationship), the existential $\sum$ in the first equation is replaced with $\lfloor \texttt{CRIT} \rfloor$. For the exit equation, we remove the transparency term (as it does not depend on basic blocks parts, and is already present in the entry equation); finally, noticing that the successor of an exit part is obviously an entry part, the disjunction over successors is updated with the entry equation.

The bitwise "and" between the entry or exit variants of R2^b INSERT and CRIT gives us the entry or exit "critical-insertion" points noted $\lceil \texttt{CRITINS} \rceil$ or $\lfloor \texttt{CRITINS} \rfloor$ (i.e. $\texttt{CRITINS} = \texttt{INSERT} \wedge \texttt{CRIT}$). Those are needed to adapt the original "substitution-critical" forward equation below, in the same fashion as before.

$$\downarrow \texttt{SUBSTCRIT}(n) \triangleq \begin{array}{l} \texttt{CRITINS}(n) \vee \\ \sum_{m \in pred(n)} (\overline{\texttt{COMP}(m)} \wedge \texttt{SUBSTCRIT}(m)) \end{array}$$

which is decomposed, from a reasoning symmetrical to that of the CRIT predicate, into:

$$\downarrow \begin{cases} \lceil \texttt{SUBSTCRIT} \rceil(n) \triangleq & \lceil \texttt{CRITINS} \rceil(n) \vee \\ & \sum_{m \in pred(n)} \left(\begin{array}{l} \overline{\lfloor \texttt{COMP} \rfloor(m)} \wedge \\ \lfloor \texttt{SUBSTCRIT} \rfloor(m)) \end{array} \right) \\ \lfloor \texttt{SUBSTCRIT} \rfloor(n) \triangleq & \lfloor \texttt{CRITINS} \rfloor(n) \vee \\ & (\overline{\lceil \texttt{COMP} \rceil(n)} \wedge \lceil \texttt{SUBSTCRIT} \rceil(n)) \end{cases}$$

10.3.2.2 *Pushing Critical Insertion Points Forward*

We now update R2^b insertion points based on R1, in order to obtain an INSERT predicate equivalent to the one of R2. Recall that replacement points are not impacted by R1.

Algorithm 2 R2 Insertion Points From R1 & R2^b [$\diamond$].

```
1:  procedure FIND_CRIT_TARGETS_REC(n : pc, cand : cand_t)
2:      st ← cand.state
3:      VISIT(pc)
4:      for n ∈ succ(pc) do
5:          if ¬st.⌈CRIT⌉(n) ∧ st.⌈SUBSTCRIT⌉(n) then
6:              st.⌈INSERT⌉(n) ← true
7:          else if ¬st.⌊CRIT⌋(n) ∧ st.⌊SUBSTCRIT⌋(n) then
8:              st.⌊INSERT⌋(n) ← true
9:          else if ¬VISITED(n) then
10:             FIND_CRIT_TARGETS_REC(n, cand)
11: procedure FIND_CRIT_TARGETS_GEN(cand : cand_t, p_ins, p_ins_crit)
12:     RESET_VISITED_BLKS(void)
13:     for pc ∈ {pc_i | p_ins_crit(pc_i) = true} do
14:         p_ins(pc) ← false
15:         FIND_CRIT_TARGETS_REC(pc, cand)
```

The FIND_CRIT_TARGETS_GEN procedure of Algorithm 2 pushes forward (in the direction of the control-flow) insertion points for SR candidates. For each of them, the procedure is called only when R1's $\lceil \texttt{CRITINS} \rceil$ (respectively $\lfloor \texttt{CRITINS} \rfloor$) is *not* full of zeros. If the entry or exit critical insertion vector is always false, we do not need to push the respective entry or exit insertion points. Parameters *p_ins* and *p_ins_crit* of FIND_CRIT_TARGETS_GEN are instantiated with either $\lceil \texttt{INSERT} \rceil$ and $\lceil \texttt{CRITINS} \rceil$ (for entry parts) or with $\lfloor \texttt{INSERT} \rfloor$ and $\lfloor \texttt{CRITINS} \rfloor$ (for exit parts). If there are both entry and exit critical

type C `affine_form` ::= `Aff_term`(C, *reg*, C `affine_form`) | `Aff_const`(C)

Figure 10.9: Polymorphic Affine Forms Over a Type of Constants C.

insertion points, the procedure is thus invoked two times. First, it sets the insert predicate (which can be either the entry or exit one) to false for every block satisfying the given (entry or exit) critical-insertion predicate (line 13 and 14). Second, the FIND_CRIT_TARGETS_REC procedure replaces insertion points: it recurses over successors of the critical-insertion block, and stops when encountering an already visited block. For entry and exit parts, if a successor is not critical but is a valid substitute (i.e. that satisfies the "substitution-critical" predicate, lines 5 and 7 of Algorithm 2), then its related (entry or exit) `INSERT` predicate is set to true. This makes our insertion points equivalent to the second refinement of Knoop, Rüthing, and Steffen [85, §4.1].

10.3.3 *Affine Forms Strength-Reduction*

The third refinement of Knoop, Rüthing, and Steffen [85] *accumulates* update assignments when the source includes multiple injuring operations (as illustrated in Example 10.3.1). Their solution is to first record program points where an accumulated update should be inserted, and second to define a function that calculates the accumulation effect. Nonetheless, this idea involves a prior detection of *extended basic blocks* [85, footnote 15]. Mimicking this technique would be possible with our block-based LSR, even if it seems a bit heavy in our formally verified defensive framework. Moreover, this mechanism is subsumed by noticing that candidates can either multiply or add values, which amounts to manipulate affine forms (like our validator). We simply define addition and scalar multiplication of affine terms (forming a semimodule [62]), to accumulate "injuries" over induction variables, to reduce products between constants, and to factorize additions on the same variable (cf. the sequence in Example 10.3.1).

Hence, we improve R3 (but only for basic blocks) with the affine forms of Figure 10.9, where C is the type for constants (e.g. on my RISC-V implementation, C is `int64`). They model sequences of the form "$c_0 \cdot r_0 + c_1 \cdot r_1 + \ldots + c_N \cdot r_N + c$" (a sum of multiplications between a constant of type C and a register, whose last element is a final constant c). The oracle maintains a hash table $(pc, reg) \mapsto (\text{C } affine\_form)$, so we map (block ID, register) pairs to affine values. The detection phase applies operations over these forms as they occur, and the local substitution mechanism of §10.2.4.2 (which is further detailed in the next section) keep the table up-to-date with auxiliary variables. When inserting the update assignment, we invoke a function that takes a list of block IDs and the candidate's auxiliary destination register (fresh register) to retrieve the compensation amount that needs to be added (by seeking in our affine forms table).

10.3.4 *Details on the Forward Substitution of Auxiliary Variables*

The code of this forward substitution is available here [◇].

Let us propose a more complete description of the substitution mechanism of §10.2.4.2, which handles candidates satisfying the `REPLACE` predicate. The idea is to enable the propagation of candidates' fresh variables locally inside the block in which we replace them. For SR candidates, this process also enables the propagation of their associated affine expression.

Let pc_t and off_t be the target block ID and an offset inside this block (respectively) where we are going to replace the candidate[17]. The function traverses the basic block starting from the entry, and, depending on the current offset off_c:

- If $off_c < off_t$ (meaning that we have not yet reached the original occurrence); simply continue and increment off_c.

[17] Here, we ignore blocks' conceptual parts, and work directly using off_t. In other words, either $\lceil \texttt{REPLACE} \rceil(pc_t)$ or $\lfloor \texttt{REPLACE} \rfloor(pc_t)$ is true.

- If $off_c = off_t$; ensure a match between the current instruction and the candidate to replace, replace it by a no-op (no-operation) instruction (rather than directly by a move), and continue[18]. The algorithm saves the original destination of the replaced instruction.
- If $off_c > off_t$ (meaning the candidate was already replaced by a `no-op`); there are two possible subcases:
  - A final case when either (i) the auxiliary variable or the original destination of the candidate is rewritten; (ii) the current instruction is another occurrence of the candidate or an injuring operation; (iii) we are reaching the end of the block.
    If so, we insert the move from the auxiliary variable and stop the substitution algorithm;
  - A recursive case otherwise, where we substitute the previously saved original destination by its auxiliary variable within the `updated_args` field of the current candidate.

10.3.5 *A Full Example of Lazy Code Transformations*

We specialized our LCT algorithm in order to strength-reduce multiplicative and additive computations on the RISC-V (64-bit) backend. Note that for the SR part of LCT, both types of invariants are exploited, because the simulation have to remember the value of constants when verifying the correctness of newly inserted instructions. Since the feasibility of SR for a given candidate is conditioned by the existing dependencies on its variables, we apply a pass of *move forwarding* in the first place. The latter removes read-after-write dependencies coming from move instructions, that might be obstacles to LCT.

The C code of Figure 10.10 initializes a slice `[i,n)` of a vector `x` with scalar "`l * i`", and contains two candidates to be reduced. Indeed, in addition to the product itself, the addressing computation to access `x[i]` can be rewritten as well. The original and optimized BTL codes are set side-by-side in Figure 10.11. The orange comment on the left gives the correspondence between registers and variables from the source C program. The multiplication "`l * i`" corresponds to "`x1 * x4`" in both codes, and, on the left code, the sequence "`x8 = x1 << 3; x6 = x3 + x8`" calculates into `x6` the address of `x[i]`. Synthetic nodes have their ID in **bold** typeface, while fresh variables are still in red, and compensation code in green. We omitted blocks 14 and 1 in the optimized BTL code, as they are identical.

```
void init_slice(long *x, long n, long i) {
  long l = 10;
  for(; i < n; i++) x[i] = l * i;
}
```

Figure 10.10: Two Candidates for LSR.

After moving the left shift instruction from block 9 to block 12, the old destination (here `x8`) is replaced in the instructions following the original position of the candidate (in block 9) with the newly allocated variable (`x9`)[19]. This enables then to also strength-reduce the addition originally assigning to `x6` in block 9 (it is moved to block 12 as the assignment to `x10`). Note that the substitution of `x8` by `x9` is fundamental here: if we had simply inserted a move directly in place of the shift instruction, the data-flow analysis over the addition would have been blocked because of the write access to one of the arguments within the block. The multiplication "`x1 * x4`" originally in block 9 is moved out as in the standard way of Knoop, Rüthing, and Steffen [85].

Of course, it is necessary to update the registers of all these anticipated computations as the `x1` argument is incremented inside the loop. To handle this, we seek into the map from registers to affine forms which is updated during the candidates' detection phase (cf. §10.3.3). For example, the left shift operation associates `x9` (formerly `x8`) to affine form "`8 · x1`" (knowing that $x \ll n = 2^n \cdot x$). When the subsequent addition is selected as a candidate, a new affine form for `x10` (formerly `x6`) is created, and by substitution of existing affine forms, its value is "`x3 + 8 · x1`". The normalization of affine forms in the oracle follows the theory given in §7.6.2.1.

Finally, every affine form "injured" within the loop needs to be incremented (respectively decremented) by the product of the constant factor—within the form—of the concerned variable by its increment (resp. decrement) step in the "injuring" operation (e.g. the loop induction variable). In

[18] At this point, we also update the tables of affine values and constants by copying the previous mapping (if existing) to a new one bound to the auxiliary variable of the candidate.

[19] The original variable being dead in the loop, the compensation move (from `x9` to `x8`) was eliminated (cf. §10.2.4.2).

```
// Variables: x1 = i;
// x2 = n; x3 = x; x4 = l

14: x4 = 10L
    goto 12

12: goto 11

11: if (x1 >= x2)
      goto 1
    goto 9

9: x8 = x1 << 3
   x6 = x3 + x8
   x7 = x1 * x4
   int64[x6+0] = x7
   x1 = x1 + 1
   goto 3

3: goto 11

1: return
```

```
ℋ: ([x4 ≔ 10], {x4})
𝒢: ([], {x1,x2,x3,x4})
12: x9 = x1 << 3
    x10 = x3 + x9
    x11 = x1 * x4
    goto 11

ℋ: ([x4 ≔ 10], {x4})
𝒢: ([x9 ≔ x1 << 3; x10 ≔ x3 + x9;
    x11 ≔ x1 * x4], {x1,x2,x3,x10,x11})
11: if (x1 >= x2) goto 1
    goto 9

ℋ & 𝒢: see block 11
9: int64[x10+0] = x11
   x10 = x10 + 8
   x11 = x11 + 10
   x1 = x1 + 1
   goto 3

ℋ & 𝒢: see block 11
3: goto 11
```

Figure 10.11: Original (left) and Reduced (right) BTL Pseudocode.

this specific example, incrementing `x1` by one corresponds to increment the affine forms of `x9` and `x10` by 8. Thus, the oracle inserts assignments "`x9 = x9 + 8`" and "`x10 = x10 + 8`" in the loop, before the injuring operation. The exact same method applies to the affine form "`10 · x1`" associated to `x11` (formerly `x7`).

This updating phase of our LCT oracle does not need to track whether the `x9` variable is read afterward (either in the current block or in a successor), since DCE will remove it. In this example, the update "`x9 = x9 + 8`" is safely removed.

10.4 INFERRING INVARIANTS FROM ANALYSES

Once the main loop of LCT—consisting of the four steps described in §10.2.5—terminates, each possible LCT optimization was applied on the CFG. Before generating invariants annotations, we perform first a liveness analysis and then a dead code elimination (validated along with LCT thanks to our target liveness symbolic execution). Dead moves or update assignments generated by LCT are thus always eliminated.

Invariants annotations are inferred from both the liveness and the LCT analyses. This process is done for each pair $(\texttt{ckey}, \texttt{cand})$ in the list L of candidates (the same list as in §10.2.5) with a *defined* auxiliary variable (i.e. not `None`) in `cand.vaux` (of Figure 10.3). In other words, we iterate over the set C of candidates defined on line 2 of Algorithm 3. In fact, a defined auxiliary variable means that the candidate was moved or strength-reduced and so the validator will need invariants to ensure the transformation's correctness. The gluemap `gm` in Algorithm 3 contains both the history and gluing compact sequences of assignments of symbolic values (cf. Definition 6.2.5). The gluing invariants mapping in `gm` is already initialized with the above-mentioned liveness analysis results (so the "alive" sets at each node are filled).

Below, §10.4.1 explains how we infer preservation points for GIs; and §10.4.2 details our method for placing HIs in order to "save"—in a shared execution past—constants.

10.4.1 *Preservation Points for Gluing Invariants*

Given a pair $(\texttt{ckey}, \texttt{cand})$, the vector of block IDs where a gluing invariant about the candidate must be preserved is named G (line 5 in Algorithm 3). Preservation points depend on four predicates for

Algorithm 3 Generation of Invariants Annotations [◇].

```
1:  function BUILD_INVARIANTS(s : pc, L : (ckey_t, cand_t) list, gm : gm, constants : reg ↦ (pc, op))
2:      C ← {(ckey, cand) ∈ L | cand.vaux ≠ None}
3:      for (ckey, cand) ∈ C do
4:          st ← cand.state
5:          G ← ((st.⌈REPLACE⌉ ∧ st.⌈INSERT⌉) ∨ (st.⌊REPLACE⌋ ∧ st.⌈INSERT⌉)) ∨ (st.ISOL ∧ st.DELAY)
6:          if IS_TRAPPING(ckey) then
7:              G ← G ∧ st.⌈U-SAFE⌉
8:          const_prod ← IS_CONSTANT_PRODUCT(ckey)
9:          ok_reduced ← cand.was_reduced
10:         for pc ∈ CFG do
11:             if pc ≠ s ∧ const_prod ∧ ok_reduced then
12:                 r_c ← CONSTANT_REG(ckey)
13:                 c_op, c_pc ← constants[r_c]
14:                 ok_fresh ← ¬IS_FRESH_VAR(r_c)
15:                 A ← LIVE_INPUTS(gm, pc)
16:                 if pc < c_pc ∧ ok_fresh ∧ (G(pc) ∨ r_c ∈ A) then
17:                     gm ← ADD_HI(gm, pc, r_c, c_op)
18:             if G(pc) then
19:                 op ← GET_CKEY_OP(ckey)
20:                 gm ← ADD_GI(gm, pc, cand.vaux, op)
21:     return gm
```

non-trapping candidates, and five otherwise. G is efficiently calculated with bitwise operations on predicates.

There are two types of nodes in which we must insert a GI: one for each alternative of line 5. The *first* (in blue) groups blocks where `cand` was replaced, but not inserted. In this case, the target simulation must retrieve the candidate's value from the input gluing invariant (recall the *target* simulation: the input GI is applied *before* executing the target block). For instance, in Figure 10.11, a gluing invariant assignment of the multiplication "`x1 * x4`" in `x11` is needed for block 9, where the candidate is replaced (in entry) but not inserted. Conversely, a counter-example (where the alternative is false), arises in block 12: actually, an input gluing invariant would be *wrong* to remember the multiplication in `x11`, since it is not yet executed on the target side. However, this first alternative is not sufficient because the candidate's value must also be preserved in the gluing invariant if the auxiliary variable is live after (e.g. across loops). Thus, we define a *second* alternative (in orange) to insert an input gluing invariant on every node which is neither isolated nor delayed. Indeed, an isolated candidate is by definition (of `INSERT`) never used for insertion (`ISOL`(n) is true if an insertion at n would be only used at n itself). Moreover, it must not be delayed: if `cand` is delayed at node n, we know that its potential insertion can only happen after n (further in the CFG). Still in the example of Figure 10.11, the loop block 11 satisfies these conditions for the multiplicative candidate (neither isolated nor delayed); thus, in the source side simulation of block 11, the multiplication "`x1 * x4`" will be defined when executing the output gluing invariant, as expected (recall the *source* simulation: the output GI is applied *after* executing the source block).

The disjunction encoded by G suffices to obtain preservation points for non-trapping candidates, but is not strong enough for trapping ones (e.g. loads). Hence, we restrict G (line 7 of Algorithm 3) by conjunction with the entry up-safety predicate (as for condition (ii) of potential replacement points in §10.2.6.1). In Figure 10.7, this stronger version of G holds on the "`load[x1+0]`" candidate for blocks 8 to 19, thus allowing to insert the necessary invariants for `x9`.

A GI assignment of the candidate's operation (in `ckey`) into the auxiliary register `cand.vaux` is therefore inserted for every block `pc` such as `pc` ∈ G (lines 18-20 of Algorithm 3).

10.4.2 *Saving Constants With History Invariants*

Code motion transformations made by our LCT never require history invariants. The only use of history invariants is to remember the value of constants in strength-reduction multiplicative

candidates. In the example of §10.3.5, this use appears with the multiplication "`x1 * x4`": the `x4` argument contains the constant `10`, which is saved by an HI. On the other hand, for the shift "`x1 << 3`" (still in the example of Figure 10.11), the constant is directly encoded as an immediate *inside* the BTL operation, so no HI is needed.

The IS_CONSTANT_PRODUCT(`ckey`) function (line 8 in Algorithm 3) returns true if the candidate is of type `SRmul`, and if its constant is in a register (by seeking in the constants' table of §10.3.1.1). Inserting an history invariant is relevant only if the multiplicative candidate was effectively strength-reduced: this is indicated by the `cand.was_reduced` Boolean (defined in Figure 10.3, and read at line 9 in Algorithm 3). Furthermore, as the CFG entry must only include pure liveness invariants (cf. §6.3.3.4); the condition of line 11 checks that the current block is not the entry point[20], along with the two conditions defined above. Nevertheless, some additional checks are required before inserting an history invariant: (i) the constant must be defined in a previous block (if it is defined in the current block, no need for an HI); (ii) the constant must not be in an auxiliary variable (otherwise it will be handled by GIs); and (iii) either the current block must appear in G, or the constant's register live in the block (if these two conditions are false, there is no need to propagate the constant's value). The algorithm first gathers the constant register (line 12), and the constant operation and block of appearance (line 13) from the constant table. The comparison $pc < c_{pc}$ then checks condition (i) above; and the negation of function IS_FRESH_VAR(r_c) (line 14) ensures (ii). The set A of live variables in the block (line 15) was already computed before the DCE pass; here, we simply retrieve this information from gm. Finally, line 16 (the disjunction corresponds to condition (iii)) verifies that the three requirements are satisfied. If so, the algorithm inserts an history invariant assignment of the constant operation c_{op} into its associated variable r_c, and add r_c in the "alive" set of history invariant at block pc (line 17).

Notice that since the alive set for GIs was already filled by the liveness analysis, we only add information to the alive set for history invariants here. In the end, the oracle returns both the new BTL code and the "gluemap" to our certified validator.

10.5 CONCLUSION

We suggested LCT, an enhanced version of the LCM & LSR of [84, 85] within CHAMOIS-COMPCERT, validated by our formally verified general purpose framework. Our adaptation of the original data-flow analyses facilitates the defensive validation by symbolic execution of those algorithms. Overall, the LCT implementation represents about 2000 significant lines of OCaml code.

10.5.1 *Algorithm Control Options*

In order to let the user of CHAMOIS-COMPCERT have some control over our implementation, the LCT pass proposed in this chapter recognizes a few command line options.

A "`-flct`" option to activate or not (with "`-fno-lct`") the CM part; a "`-flct-trap`" option to activate or not the CM of trapping instructions in particular; and a "`-flct-sr`" option to activate or not the SR part. The second and third options can be set only when the first is also set.

Moreover, two ways of bounding the LCT algorithm are proposed. The first option is by limiting the maximum number of candidates to a threshold $n \in \mathbb{N}^*$. When activated, candidates are sorted after the detection phase and only the n candidates with the highest potential performance gain are kept. This procedure prioritizes SR candidates by sorting them in their topological order; while CM candidates are sorted by decreasing predicted latency w.r.t. to the pipeline model of the target core, if applicable. The final lists of candidates are then concatenated, with first SR ones (from earliest to latest) and second CM ones (from slowest to fastest). By default, in CHAMOIS-COMPCERT, this threshold is set to 64.

Our second option is to limit the LCT execution time to a threshold s, in seconds. When set, our algorithm stops its analysis when the running time reaches s, thus abandoning the optimization and retuning the original version of the code. This option is more aggressive, since its either all or nothing (there is no notion of partial LCT application with this method). That is why we do not activate it by default. Of course, both limitations can be combined.

[20] For GIs, this was implicitly ensured by the formula of G.

Finally, I added a sixth option that decides if a tunneling pass should be performed after the LCT pass. This tunneling pass eliminates any unnecessary branches, if any, and is applied on RTL using a port of the LTL tunneling specific to CHAMOIS-COMPCERT (it does not exist in the mainline version).

10.5.2 *Limitations of Our Formally Verified SR*

For now, the SR part of our LCT algorithm is restricted to affine arithmetic on `long` for RISC-V 64 bits architectures.

In addition to the inherent limitations resulting from the data-flow approach (see §10.5.3 below), our LCT is also subject to other restrictions because of the surrounding verification environment.

For example, as detailed in §7.6.2.1, we do not fully support the standard affine arithmetic. It seems that we could recover more powerful equations by considering a multi-sorted equational theory. But, BTL, inspired by RTL, is an untyped language which makes this way difficult. Let us now discuss other limitations of our SR.

INTEGER SIZE I mentioned in §7.6.2 that our affine normalization was only focused on long integers. This is because in addition to its architecture dependent aspect, the affine normalization is also type dependent. Indeed, due to the COMPCERT's encoding of values (recall §3.3.1), the entire normalization process would have to be proved again to handle `int` values. For sure, developing this proof would not be very hard, as it should resemble a lot to the existing one for `long`.

Nonetheless, on 64-bit architectures, this would only work for `int` multiplications unrelated to addressing calculations. This is because the compiler would need to insert a cast of the 32-bit index to 64-bit before scaling and adding it to the base address. Due to overflows, it can be wrong to strength-reduce the scaling. We thus opted for an alternative, more general solution: rather than applying SR on `int`, we prefer promoting them to `long` with a prior, dedicated pass.

Not only does this save us from having to prove normalization a second time, it also saves cycles on 64-bit architectures! In particular, on KVX, and even on certain AArch64 processors (e.g. the ARM Cortex-A53), the sign extension used to cast the value to a `long` makes us lose one cycle in latency. When this additional computation is inserted inside a loop, this may dramatically hinder code performance.

In fact, solving this problem with LICM is not that trivial, because the other loop instructions will probably block the lifting process if they expect a 32-bit argument. A promotion pass is therefore of great interest in that situation to pre-perform the sign extension *and* convert dependent operations to their 64-bit equivalent. Promoting is of course not always possible, and requires results from an interval analysis to be proven correct. See §11.2.

TARGETING 32-BIT ARCHITECTURES As just explained, the reduction of array addressing is currently limited to 64-bit architectures: either by directly applying SR on `long` indexes, or relying on an prior promotion for `int` ones. Porting it to a 32-bit architecture seems rather straightforward. Actually, combining 32-bit and 64-bit arithmetic on a 32-bit architecture seems easier than the opposite because truncation commutes with most `long` operations (in modular arithmetic). It would only require a little generalization of the syntax of our history invariants for allowing the source registers to be defined as symbolic expressions of *target* registers. But, this generalization does not seem difficult because our semantics of invariants already enables it.

DYNAMIC CONSTANTS AND PARAMETERS Currently, our LCT is unable to strength-reduce any dynamic constant (i.e. whose value is unknown, like function's parameters). This limitation is due to two reasons. First, the oracle would need to insert an additional multiplication for any non-literal constant multiplied by an induction variable whose increment is greater than one. Second, validating the result implies a non-trivial extension of our affine rewriting engine to integrate support of multiplications between two arbitrary symbolic values.

ELIMINATING/REDUCING LOOP COUNTERS (LINEAR-FUNCTION TEST REPLACEMENT, LFTR) Let us consider the code generated by "`gcc -O1`" for RISC-V 64 bits on the source in Figure 10.10. It is quite similar to the reduced code generated by CHAMOIS-COMPCERT, represented at Figure 10.11 except

that the loop is rotated (cf. Figure 4.1); and the increment "`x1 = x1 + 1`" is eliminated from the loop. GCC compensates for this elimination by replacing the loop condition "`x1 >= x2`" by condition "`x10 == x12`" where `x12` is a fresh variable initialized by "`x12 = x2 << 3; x12 = x3 + x12`", before the loop. In other words, GCC replaces the source condition "`i < n`" by "`x + i != x + n`".

We cannot prove such a transformation with our validator. Indeed, such a transformation seems difficult to verify in CompCert. First, note that the replacement of condition "`i < n`" by "`x + i < x + n`" would be incorrect because of possible overflows (but, in modular arithmetic, "`i != n`" implies "`x + i != x + n`"). Second, justifying the replacement of condition "`i < n`" by "`i != n`" requires inferring the loop invariant "`i <= n`": proving such an invariant, and allowing to rewrite—under this invariant—the condition "`i < n`" into "`i != n`" require non-trivial extensions of our validator. Last, in the CompCert memory model (as in the C standard), the comparison "`x + i != x + n`" is only well-defined if "`x + i`" and "`x + n`" are valid and point within the same allocated block (or just after the end of the block). Hence, it is highly non-trivial to prove that if the source program has no undefined behavior then "`x + i != x + n`" is also well-defined. See the discussion on BTL expressivity limitations in §9.3.

This example illustrates that some seemingly simple optimizations of "`gcc -O1`" are still difficult to formally justify within CompCert[21].

10.5.3 *Related and Future Work*

See §2.5 for general related work on translation validation and §9.4 for those concerning BTL.

HISTORY OF LOOP STRENGTH-REDUCTION As Cooper, Simpson, and Vick [36] rightly explained, strength-reduction of loops has several positive effects: it minimizes the number of instructions in loops, can minimize the number of needed registers, replaces multiplications with additions, and gives the scheduler more freedom when there are multiple arithmetic-logic units (ALUs)[22]. The authors even explain that for those reasons, they *"expect that strength reduction will remain a useful transformation, even if the costs of addition and multiplication become identical."*

There are two main ways of strength-reducing loop induction variables [36, §2]: methods working "a single loop at a time", seeking for loop induction variables [34] (e.g. SSA based techniques of GCC and LLVM, see below); and data-flow approaches [111] (e.g. LSR), which do not require control-flow analyses, and are mostly inspired by code motion and partial redundancy elimination. The absence of control-flow analysis makes data-flow methods simpler, but they only detect literal constants.

Refer to §10.2 for information about the LCT complexity.

See this explanation from [36]: *"This forces them [the data-flow methods] to use a much simpler notion of region constant—they detect only simple literal constants. Thus, they miss some opportunities that the ACK-style [for Allen, Cocke, and Kennedy—i.e. the first approach] methods discover, such as reducing* $i \times j$ *where* i *is the induction variable of the innermost loop containing the instruction and* j *is an induction variable of an outer loop. These algorithms must be repeated to handle second-order effects. Their placement techniques avoid lengthening execution paths; algorithms in the ACK family, including our own [their Operator Strength-Reduction (OSR) algorithm], cannot make the same claim."*

Although the above quote does not argue in favor of data-flow algorithms, we still chose this method because it fits well with our validation framework. Indeed, as we demonstrated it, data-flow equations can help in generating invariants. Moreover, the OSR algorithm of Cooper, Simpson, and Vick [36] is only operating on SSA graphs, which we do not yet support in BTL (cf. §9.4).

IN TODAY COMPILERS Strength-reduction designates various transformations, from replacing single instructions to linear-function test replacement (LFTR). The only form of SR in mainline CompCert is a form of peephole[23], divided among instruction selection and constant propagation. Modern, untrusted compilers rather implement straight-line SR (SLSR), a more powerful transformation targeting code sequences with arithmetic statements[24], that simplifies complex sequences unhandled by loop SR algorithms. The expansion mechanism for RISC-V of §7.6.1 could be, as a future work,

[21] This limitation of CompCert 's memory model may seem overly stringent, but is difficult to relax while preserving the many necessary properties of the memory model.

[22] Because it is common to have more add than multiply units, so additions are often less constraining to parallelize.

[23] Replacing an instruction sequence by a more efficient pattern.

[24] In LLVM: https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/StraightLineStrengthReduce.cpp; In GCC: https://github.com/gcc-mirror/gcc/blob/master/gcc/gimple-ssa-strength-reduction.cc.

extended to perform SLSR as well. See this online review on the LLVM website for a reference and motivating example: https://reviews.llvm.org/D7310.

The loop nests SR algorithms in GCC & LLVM are SSA based[25], and might be very difficult to adapt in a formally verified context. They can reduce induction variables, but also perform LFTR (i.e. completely eliminate the original induction variable). Furthermore, GCC and LLVM use a scalar evolution (SCEV) analysis, an efficient technique to find induction variables in specific code regions (e.g. loops). Proving correct such an analysis would nonetheless be interesting, knowing that it is subject to implementation bugs [147, §3.7, LLVM Bug #4].

Two important earlier works on CompCert are comparable to our LCM, and are examined below.

COMPARISON WITH THE VERIFIED LCM OF TRISTAN AND LEROY Tristan and Leroy [143] proposed a Coq-verified translation validator for LCM, based on two formally verified data-flow analyses, *availability* and *anticipability*. These analyses have quite high algorithmic complexity (cubic for availability). In contrast, our validator does not use them. The availability analysis is replaced by our gluing invariants which are themselves provided "for free" by the oracle: we hence avoid replaying a data-flow analysis already performed by the oracle. Hence, from the analysis of §9.3 over the case of our LCM (with a bounded number of candidates in invariants, without rewriting rules and working on basic blocks—i.e. with a bounded number of block exits), our validator is quasi-linear in practice: its worst-case complexity is $\mathcal{O}(n \times l)$ where n is the size of the code and l the number of maximal simultaneously live registers.

Moreover, we combine LCM with CFG restructuring, which validates some PRE of trapping instructions (a feature that they did not provide). Our CFG restructurings also partly compensate the lack of anticipability checking that is necessary to validate FRE of trapping instructions. In future works, our symbolic simulations might check the anticipability of trapping instructions, with a dedicated notion of *prophecy* (cf. item (iii) in §9.3).

Another difference between our setting for loop optimizations and theirs is that we operate at the level of large blocks while they operate at instruction granularity. Technically, their validator expects that any anticipated instruction is assigned with a fresh auxiliary register. When comparing the source and the target, a target instruction assigning a register that is dead for the source code can be skipped. But a source instruction assigning a register r can only be changed for a move instruction to r (from an appropriate auxiliary register). Thus, their validator seems less general purpose than ours. In particular, it would neither support the reduction of instruction sequences (contribution (iii) at the start of the chapter), nor instruction reordering modulo code compensation (that we explain in §11.3.1); while our validator does.

COMPARISON WITH THE CSE3 OF MONNIAUX AND SIX Monniaux and Six [109, 110] proposed three dedicated and formally verified passes to produce an efficient CSE optimization with LICM integrated into Chamois-CompCert. After loop unrolling (as we do), they run an untrusted analysis to collect inductive invariants in hash-consed sets, whose inductiveness is checked by a proven verifier, before eliminating redundant computations. This last phase actually consists of three sub-steps: replacing computations by move operations; and replacing moves from a variable to itself by "no-op"; then apply an existing DCE pass. On the one hand, their decomposition simplifies the formal proofs of each single pass. On the other hand, it can only validate some redundancy elimination. In contrast, our approach aims at validating a wider class of transformations, e.g. including scheduling and strength-reduction. However, their optimization includes an elimination of redundant conditions, a feature we leave to future work.

[25] LoopSR in LLVM: https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/LoopStrengthReduce.cpp; IVOPTS in GCC: https://github.com/gcc-mirror/gcc/blob/master/gcc/tree-ssa-loop-ivopts.cc.

11

INTEGRATION OF OTHER BTL OPTIMIZATIONS

Beside my work on the LCT algorithm, several other BTL optimizations have been developed *in collaboration with three of my colleagues* at Verimag: Alexandre Bérard, Benjamin Bonneau, and David Monniaux.

Those developments are available on the Verimag repository[1].

Some transformations presented in this chapter* are included in the experimental evaluation of Chapter 12. Note that their detailed specification is out-of-scope of this document.

Section 11.1 explains how to adapt BTL to support memory optimizations, and Section 11.2 introduces an abstract interpretation framework for BTL and a related promotion pass. Then, I sequentially present three applications validated using—for some of them, a generalized version of—our defensive simulation technique: an enhanced superblock scheduling in Section 11.3; the graph factorization in Section 11.4; an enhanced LCT in Section 11.5; and store motion in Section 11.6. Finally, Section 11.7 details how we set up our BTL passes in the existing CompCert pipeline.

11.1 VERY SUCCINCT OVERVIEW OF BTL GENERALIZATIONS

Among the new optimizations of this chapter, some are using memory access analyses to simplify the code. This required several generalizations of our SE framework, and notably of its memory model. All of them have been implemented by Benjamin Bonneau (during a six months internship for the ENS-PSL school):

MEMORY INVARIANTS: are a list of abstract stores added to the "gluemap" invariant mapping (Definition 6.2.5). These stores are encoded with a special type similar to ival (Definition 6.2.3). Memory invariants are evaluated in the target initial state of the simulation, after history and gluing ones. The resulting symbolic memory is then replaced in the source initial state.

SYMBOLIC CLAUSES: Bonneau also extended the inductive type of symbolic values with a mutual type of clauses representing propositions. These clauses can be evaluated as a Boolean, and are used instead of the symbolic state precondition (Definition 6.4.6). They are represented as propositions in the theory, and as a list of clauses in the implementation. This approach is very modular, since a new kind of clause can be defined if needed to encode some specific information. For now, clauses are used, among other things, for propagating information about live variables, promotable operations and conditions, and valid memory accesses.

MEMORY REWRITES: the symbolic memory of our framework has kept its representation based on a stack of symbolic stores (Definition 6.2.1). For the simulation test on final states to succeed, symbolic memories must be rewritten during the (symbolic) execution. This is managed by a specific set of normalization rules applied with the rewriting engine of our framework (§7.6).

11.2 PORTING STATIC ANALYSES FROM RTL TO BTL

Mainline releases of CompCert run multiple analyses over the RTL intermediate representation. Since BTL has almost the same vision of the execution state (pseudo-registers and memory) as RTL, the same abstract transfer functions can be used.

Leveraging this similarity, Benjamin Bonneau and David Monniaux defined an interface (as a Coq module type) that provides the abstract states and transfer functions of an analysis (as well as proofs of their correctness). Given an implementation of this interface, one obtains a proven abstract analysis that can be run *both* on RTL and BTL.

[1] https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert

*Many parts and examples of this chapter are adapted from our article about BTL [65]† (including appendices).

Then, Bonneau made BTL annotatable with results of verified static analyses instantiated in this new interface. These analyses are directly written and proven correct in Coq (i.e. without any kind of translation validation), and are generalizations of existing analyses of CompCert. Hence, annotations produced are formally guaranteed and usable both by the oracle and the simulation test. The BTL semantics take them into account as assertions (cf. the clauses of §11.1) utilized to justify some rewrites. For their part, oracles try to exploit annotations to optimize more efficiently.

VALUE & ALIAS ANALYSES Bonneau and Monniaux ported into this framework the existing value analysis[2] of Robert and Leroy [127], which includes an alias analysis. This analysis abstracts values within an (infinite) lattice of finite height, and also abstracts the contents of memory blocks. One important difference is that the original abstract execution of a single instruction in RTL produces one single abstract state (made of an abstract register state and an abstract memory state), which is propagated to all successors of the instruction (only branching instructions have multiple successors), whereas the framework of Bonneau and Monniaux allows branching instructions to provide a different abstract state for each successor. They took this opportunity to slightly improve on the approach implemented in the original analysis, which did not implement transfer functions for conditions (thus, for instance, it did not track that "`i = 0`" after a branch with condition "`i = 0`" is taken). They added transfer functions to conditions (only, so far, for equality tests), and they also added propagation of the value of the branching variable through jump table instructions.

INTERVAL ANALYSIS & PROMOTION Bonneau used the same framework to implement the interval analysis needed for the integer promotion pass (cf. §10.5.2).

Thanks to their work, we overcame the LCT limitation on `long` by combining two steps: (i) their formally verified interval analysis to justify the absence of overflow—for instance, under a loop condition of the form "`i < n`" for some `n`, then we know that the increment of `i` cannot overflow[3]; (ii) the use of the intervals found by this static analysis within an oracle (also realized by Bonneau) in order to validate the promotion of `int` variables as `long`: if there are no overflows, sign extension (or zero extension) and addition commute. The transformation performed by this oracle is itself validated by the generalized, annotatable symbolic simulation.

11.3 IMPROVED SUPERBLOCK SCHEDULING

During my thesis, I ported the superblock scheduler of Six et al. [135]† from RTLpath to BTL. Based on the model of §4.1.4.2, it attempts to minimize the running times of the execution path covering the whole superblock, even if it may increase running times of early exiting paths. The schedule is provided by an oracle and validated by our BTL simulation test. Among the two enhancements presented below, the first one (§11.3.1) is directly supported by the simulation test of Chapter 6, while the second one (§11.3.2) requires the generalizations of §11.1.

11.3.1 *If-Lifting*

Lifting conditional branches, especially when combined with code duplications, is a way to perform a weak form of software pipelining [89]. However, due to the *inflexible* block structure of RTLpath (cf. §4.4.5), applying if-lifting was difficult [135, Footnote 10]†. Alexandre Bérard [4] adapted and integrated the if-lifting of Justus Fasse [75] in BTL.

The ARM Cortex-A53 is a dual-issue in-order processor with two ALUs (already introduced in Chapter 4).

Let me show the effectiveness of if-lifting in parallelizing loop calculations on the Cortex-A53 (AArch64) core. In Figure 11.1, we optimize the top source C code by parallelizing computations between two successive iterations of the loop body. The left-hand side represents the BTL code of the loop body, after a loop-rotate and unroll-body (cf. Figure 4.1). Hence, the "`Loop`" superblock (which is a loop body containing two iterations of the original loop) is scheduled as the BTL block represented on the right-hand side: the two floating-point computations of the first iteration (in

[2] https://compcert.org/doc/html/compcert.backend.ValueAnalysis.html

[3] In C, overflow has undefined behavior in signed arithmetic, so if the loop index is signed (`int`), as it often is, we could simply assume overflow does not occur. Signedness information is however not available at that stage in CompCert (overcoming this limitation would require propagating type information from the frontend to RTL, a non-trivial work).

violet color, annotated as A and B) have been moved below the intermediate exit in order to be interleaved with those of the second iteration. The scheduler predicts[4] that the target loop body will run in at most 22 cycles instead of 32 cycles for the original one (e.g. more than 30% gain of running time). However, preserving the semantics requires *register renaming* with fresh registers (whose first occurrence in a branch is denoted in red color) and insertion of *compensation code* at the intermediate exit. Compensation instructions and fresh variable substitutions are colored green. Because of this insertion, the target block becomes an extended block.

This whole transformation is validated seamlessly by our simulation test, with GIs reduced to a conjunction of identity liveness assignments for all registers of the live sets given on the right-hand side (in orange color).

The oracle implementation of Alexandre Bérard [4] is also quite simple. First, a preliminary pass performs a backward register renaming (avoiding the forward renamings of [135, §5.3]† because they tend to pollute the superblock under scheduling with compensation renamings). Then, the scheduling solver is invoked on a *fake superblock*, with an empty live set on intermediate side exits. Last, the necessary compensation code is inserted in the side exits, following Justus Fasse [75]'s heuristic. Bérard's heuristic then compares the makespan (total estimated time) of this scheduling to the standard one (computed on the *original superblock* with correct liveness and without compensation code). If the ratio of the size of the compensation code over the makespan gain is greater than a given threshold, the standard one is retained instead. Hence, compiler users may control this scheduling heuristic by tuning the threshold on the command line.

```
double sumsq(double *x, unsigned long len) {
  double s = 0.0;
  for (unsigned long i=0; i<len; i++) s+=x[i]*x[i];
  return s;
}
```

```
Loop:
  x7 = float64[x2+x3<<3]
  x6 = x7 * x7 // A
  x4 = x4 + x6 // B
  x3 = x3 + 1
  if (x3 >= x1)
    goto Exit
  // start second iteration
  x7 = float64[x2+x3<<3]
  x6 = x7 * x7
  x4 = x4 + x6
  x3 = x3 + 1
  if (x3 >= x1)
    goto Exit
  goto Loop

Exit:
  return x4
```

```
Loop: // live: x1, x2, x3, x4
  x11 = float64[x2+x3<<3]
  x8  = x3 + 1
  if (x8 >= x1) {
    x10 = x11 * x11 // A
    x9 = x4 + x10 // B
    x4 = x9
    goto Exit // live: x4
  }
  x3 = x8 + 1
  x7 = float64[x2+x8<<3]
  x10 = x11 * x11 // A
  x6 = x7 * x7
  x9  = x4 + x10 // B
  x4 = x9 + x6
  if (x3 >= x1)
    goto Exit // live: x4
  goto Loop
```

Figure 11.1: Interleaving of Unrolled Loop-Bodies on AArch64 (pseudocode).
The case where "`len = 0`" is omitted for simplicity.

11.3.2 *Alias Aware Superblock Scheduling*

Benjamin Bonneau and David Monniaux investigated the benefits of alias analyses for the superblock scheduler, and they refined it with reordering of non-interfering load or store w.r.t. store.

[4]This estimation occurs at an abstract level and thus cannot be precise. First, the subsequent register allocation could introduce unexpected spills. Second, this estimation assumes that there is no cache miss. Third, the pipeline model is inexact.

```
extern void foo(int *u);

void bar(int *t) {
  int u[3];
  u[0] = t[0]+1;
  u[1] = t[1]+1;
  u[2] = t[2]+1;
  foo(u);
}
```

```
ldr  w2, [x0, #0]
add  w2, w2, #1
str  w2, [sp, #16]
ldr  w4, [x0, #4]
add  w6, w4, #1
str  w6, [sp, #20]
ldr  w3, [x0, #8]
add  w5, w3, #1
str  w5, [sp, #24]
```

```
ldr  w2, [x0, #0]
ldp  w3, w4, [x0, #4]
add  w6, w2, #1
add  w5, w3, #1
add  w4, w4, #1
stp  w6, w5, [sp, #16]
str  w4, [sp, #24]
```

Figure 11.2: AArch64 Scheduling With Robert and Leroy [127] Analysis.

```
void incr3(int *x) {
  x[0] ++;
  x[1] ++;
  x[2] ++;
}
```

```
ldr  w1, [x0, #0]
add  w5, w1, #1
str  w5, [x0, #0]
ldr  w4, [x0, #4]
add  w1, w4, #1
str  w1, [x0, #4]
ldr  w3, [x0, #8]
add  w2, w3, #1
str  w2, [x0, #8]
```

```
ldp  w1, w5, [x0, #0]
ldr  w3, [x0, #8]
add  w4, w1, #1
add  w1, w5, #1
stp  w4, w1, [x0, #0]
add  w2, w3, #1
str  w2, [x0, #8]
```

Figure 11.3: AArch64 Scheduling With Relative Addressing Analysis.

Alias analyses allow swapping stores with other non overlapping memory accesses. To achieve this, Bonneau and Monniaux implemented another version of the system of constraints for the scheduling problem. They run some alias analyses on memory accesses. A dependency is inserted between a read and a subsequent write (write-after-read dependency), a write and a subsequent write (write-after-write), a write and a subsequent read (read-after-write) only if according to analyses, they *may* interfere. It is sufficient that one analysis proves noninterference for the dependency not to be inserted. Bonneau and Monniaux used two alias analyses.

The first alias analysis runs the per-function value analysis discussed in §11.2, and uses the noninterference predicate provided by the value domain in mainline CompCert releases. For instance, if a pointer is proved to always point inside some global variable, and another pointer to always point inside some other global variable, then they cannot interfere—recall that attempting to move, through pointer arithmetic, between different variables has undefined behavior in C, and this is reflected by CompCert's memory model that each variable lives in a distinct memory block. This is a direct port of an existing analysis in CompCert [127].

In practice, the most useful noninterference case seems to be between contents of the current stack frame (`Stack` in the value domain), and anything outside the current stack frame (`Nonstack` in the value domain), such as anything pointed to by function parameters—indeed, a parameter pointer cannot point into the current stack frame, because the block of the current stack frame does not exist yet when the pointer is created. Consider the source code in Figure 11.2 left frame. Without alias analysis, the AArch64 code produced appears in the Figure 11.2 middle frame. The three memory assignments are not reordered by the scheduler (neither in prepass nor in postpass) because of a potential interference. They are carefully preserved in sequence, each as load (`ldr`), addition (`add`), store (`str`). This sequence will result in pipeline stalls, since every load takes multiple cycles even if available in the first level cache. With alias analysis, it is known that `t[.]` and `u[.]` cannot alias, because the former is outside the current stack frame and the latter is inside. Thus, the scheduler can first perform the three loads, then the three additions, then the three stores.

The assembly code in Figure 11.2 right frame is obtained in two steps. Firstly, the alias aware prepass—which operates on BTL—groups loads and stores together as just described. And secondly, noninterfering loads and stores to consecutive addresses are fused (into `ldp` and `stp`) by the postpass instruction rewriter—which operates on assembly code—of §4.3.3.2.

The second alias analysis addresses the cases where noninterference can be established because two pointers point to non-overlapping data chunks within the same object, for instance different

fields inside the same structure. It performs a local abstract interpretation within the superblock. Abstract values for pointers are of the form "$v_i + o$", where i is an integer index, v_i designates a "symbolic value", and o is a constant integer offset; we also have abstract values consisting only of a constant integer. When a value comes from the starting point of the superblock, or is computed by an operation inside the superblock that is not handled by the abstraction (e.g. multiplication), that value is abstracted by "$v_i + 0$" where v_i is fresh (the index i has never been used so far). When a value is computed by adding a pointer abstracted by "$v_i + o$" to an integer constant c, the result is abstracted by "$v_i + (o + c)$". Chunks of size s_1 and s_2 pointed to by pointers abstracted by "$v_i + o_1$" and "$v_i + o_2$" (note the same base pointer v_i) respectively are deemed not to interfere if the intervals $[o_1, o_1 + s_1)$ and $[o_2, o_2 + s_2)$ do not overlap.

Consider the source program of Figure 11.3 left frame. Without alias analysis—see Figure 11.3 middle frame—the scheduler is faced with the same issue as in the previous example: the three memory increments are kept in sequence, and the pipeline stalls. With alias analysis, like in the previous example, the scheduler can swap and group loads and stores. See Figure 11.3 right frame. The only difference comes from the criteria to ensure nonaliasing. In Figure 11.3, we consider offset relatively to the same base, whereas in Figure 11.2, we consider the allocation class of the pointers.

While both analyses are appealing, and indeed improve code on examples such as the above, experiments showed that, often, the improvement is not noticeable even on examples where the schedule is altered by activating these analyses. Performance is improved markedly only in specific benchmarks. Bonneau's generalized version of our BTL symbolic validator (cf. §11.1) fully supports the first analysis but only a restricted version of the second analysis. Since the latter is only used by the scheduler and for marginal gains, we postponed its complete integration in the checker.

11.4 FACTORIZATION

Many of our optimizations rely on code duplications to improve their efficiency or even circumvent certain limitations, as the anticipation of trapping instructions in my lazy code transformations algorithm (cf. §10.2.1.2). Leaving such a high amount of unrolled code in the generated assembly is nonetheless not desirable, as code size matters (especially on memory limited systems).

The factorization pass—written by Alexandre Bérard [4]—is performed by an oracle running on RTL code. This oracle post-processes the BTL to RTL oracle. Its result is checked by the morphism validator of §8.2.2 (the oracle must therefore calculate the CFG morphism corresponding to its transformation). Instead of targeting a given subset of loops (that could have been tagged as "to reroll" in previous passes), this solution tries to apply a variant of the standard DFA minimization by identifying equivalence classes on the CFG.

The obvious advantage in this approach is its independence from the other compiler pass, as the minimization does not need any information to process. Nevertheless, this genericity comes at a price: the generic factorization is easily hampered by small changes in the CFG. For instance, it cannot be applied along with if-lifting, because renaming alters the code syntax. The same phenomenon appears with scheduling which produces a reordered code harder to factorize. We consider this optimization to be still in the prototype stage.

In contrast, the factorization effect is great for pinpoint optimizations like LCT, since the code is less refactored and its structure untouched. As in the example of §10.2.7, where factorizing allows us to undo the first iteration unrolling whose only purpose was to circumvent the anticipability limitation of LCT.

11.5 MAKING LCT ALIAS AWARE

The (absolute) alias analysis ported to BTL and the associated rewrites in the symbolic simulation engine can also be used to improve LCT. I modified (code available in the Verimag repository[1]) the code motion part of our LCT to leverage noninterference information.

Taking account aliases was very straightforward: it suffices to slightly adapt the *transparency* local analysis so that load candidates stay transparent w.r.t. to noninterfering stores. This modification represents only a few lines of OCaml to check that both abstract addresses, which are now available as annotations in a special field of loads and stores, are disjoint.

```
#define n 100

void bar(int *x);

int b;
void foo() {
  int x[n];
  int i = 0;
  int s = b;
  while (i < n) {
    x[i] = b;
    s += b;
    i++;
  }
  b = s;
  bar(x);
}
```

```
x2 = 0
x1 = int32["b" + 0]
goto Loop

Loop:
  if (x2 >=s 100)
    goto Exit
  x3 = stack(0) // Allocate x[.]
  x5 = int32["b" + 0]
  int32[x3 + sext(x2) << 2] = x5
  x1 = x1 + x5
  x2 = x2 + 1
  goto Loop

Exit:
  int32["b" + 0] = x1
  x4 = stack(0)
  bar(x4)
```

```
x2 = 0
x6 = int32["b" + 0]
x1 = x6
x7 = stack(0)
goto Loop

Loop:
  if (x2 >=s 100)
    goto Exit
  int32[x7 + sext(x2) << 2] = x6
  x1 = x1 + x6
  x2 = x2 + 1
  goto Loop

Exit:
  int32["b" + 0] = x1
  x4 = x7
  bar(x4)
```

Figure 11.4: Code motion With Alias Aware LCT on AArch64 (pseudocode).

Figure 11.4 is a simple example—here on AArch64, but it also works on other platforms—involving the motion of a load *above* a disjoint store. The source code in the left frame of Figure 11.4 already contains a load of variable `b` before the loop, so that we do not need any unrolling to anticipate it. Then, Figure 11.4 middle frame shows the RTL pseudocode obtained after the "classical" LCT without alias analysis, and Figure 11.4 right frame contains the result of the alias aware LCT. Instructions to eliminate are in violet, fresh variables in red, and compensation code or substitutions in green. Concluding noninterference is trivial here, as `b` is global while `x[.]` is allocated on the stack. Exploiting this information, LCT eliminated the load of `b` from the loop body by introducing a fresh variable `x6` to remember the equivalent load of the header (originally in `x1`). It also factorized the stack access "`stack(0)`" in a fresh variable `x7`.

11.6 STORE MOTION

We recently integrated a store motion optimization into the Verimag CHAMOIS-COMPCERT fork, developed by Benjamin Bonneau. This optimization involves moving down stores located in loop bodies, when applicable, to reduce the number of memory accesses. Specifically, store motion can permute a store with loads or other stores and eliminate redundant stores.

An example combining sequentially LCT and store motion (in this order) on the C source code of Figure 11.5 is provided. The example involves a pointer `*p`, which accumulates each value of the `src` array, initialized by the `init` external function. This is reflected in Figure 11.6 left frame, which displays the RTL pseudocode after unrolling the first loop iteration. By means of the promotion pass of §11.2, the loop induction variable `i` and the addressing calculation are of type `long` (as visible in instruction "`x7 = 1L`"). Figure 11.6 middle frame illustrates how LCT leverages this promotion to strength-reduce the shift and the addition needed to calculate the array address. Lastly, the impact of store motion is illustrated in Figure 11.6 right frame, where the store is removed from the loop body by moving it across the load.

```
extern void init(int *src, int n);
void foo(int* p) {
  int src[100];
  init(src, 100);
  *p = 0;
  for (int i = 0; i < 100; ++i)
    *p += src[i];
}
```

Figure 11.5: Source code for Figure 11.6.

The color code in the Figure remains consistent, with fresh variables in red and compensation code in green. Note that store motion in this example *requires* prior loop unrolling for the same anticipation reason as in LCT: to avoid adding any potential trap, the store to eliminate must have been encountered earlier in the source to ensure it is a valid pointer. Memory stores that were

```
x2 = stack(0)
x3 = 100
init(x2,x3)
x4 = 0
int32[x1 + 0] = x4
x5 = int32[stack(0)]
x6 = x4 + x5
int32[x1 + 0] = x6
x7 = 1L
goto Loop

Loop:
  x8 = x7 << 2
  x9 = x2 + x8
  x5 = int32[x9 + 0]
  x6 = x6 + x5
  int32[x1 + 0] = x6
  x7 = x7 + 1
  if (x7 >= 100)
    goto Exit
  goto Loop

Exit:
  ret
```

```
x2 = stack(0)
x3 = 100
init(x2,x3)
x4 = 0
int32[x1 + 0] = x4
x5 = int32[stack(0)]
x6 = x4 + x5
int32[x1 + 0] = x6
x7 = 1L
x10 = x7 << 2
x11 = x2 + x10
goto Loop

Loop:
  x5 = int32[x11 + 0]
  x6 = x6 + x5
  int32[x1 + 0] = x6
  x11 = x11 + 4
  x7 = x7 + 1
  if (x7 >= 100)
    goto Exit
  goto Loop

Exit:
  ret
```

```
x2 = stack(0)
x3 = 100
init(x2,x3)
x4 = 0
x12 = x4
x5 = int32[stack(0)]
x6 = x4 + x5
x12 = x6
x7 = 1L
x10 = x7 << 2
x11 = x2 + x10
goto Loop

Loop:
  x5 = int32[x9 + 0]
  x6 = x6 + x5
  x12 = x6
  x11 = x11 + 4
  x7 = x7 + 1
  if (x10 >= 100)
    goto Exit
  goto Loop

Exit:
  int32[x1 + 0] = x12
  ret
```

Figure 11.6: Promotion, LCT, and Store Motion Using Load-Store Alias Analysis on RISC-V (pseudocode).

previously before and inside the loop are replaced with assignments in a fresh variable `x12`, and the memory write is delayed until the `Exit` label with a compensation store.

Overall, this example effectively demonstrates the impact of combining oracles. Not only have unrolling and store motion simplified memory accesses, but the *integer promotion and LCT* have also reduced the array addressing calculation. *All* these optimizations are validated using our defensive simulation.

Practically, the store motion oracle can be considered the dual of LCT: instead of lifting computations to achieve computational optimality while minimizing the live range, store motion pulls down memory writes only if it reduces the overall number of stores on a path. Bonneau noted that this duality is recognized in the literature [99]. Furthermore, like LCT, store motion also operates on basic blocks and requires a CFG free of critical edges[5] (recall §10.2.1.1).

The store motion validation in BTL required reordering stores inside symbolic memories to ensure they appear in the same order on both the source and target sides. This reordering is justified thanks to memory rewrites, and the effect of stores is propagated through memory invariants (cf. §11.1).

11.7 PLACEMENT OF BTL PASSES IN THE COMPCERT PIPELINE

Information given here is for illustrative purpose. The actual order of passes changes regularly as development progresses.

I implemented a Coq module to manage BTL passes and abstract them from the rest of the compiler pipeline: from the high level point of view of RTL passes, a sequence of BTL passes is a single opaque pass from RTL to itself. The internal content of these sequences of passes is controlled in a very modular way, so that we can change it without modifying proofs. See our module in the Verimag repository here [◇] (my manuscript repository contains a lighter variant of this module).

Table 11.1 compares the list of RTL passes of mainline CompCert and of the Verimag Chamois-CompCert (including the opaque sequences of BTL passes). Passes that are specific to Chamois-CompCert are colored red.

[5] Store motion depends on unidirectional data-flow fixed points inspired by those computed for LCT.

(1)	Tail calls recognition	(19)	Integers promotion
(2)	Tail recursion optimization	(20)	Renumbering pre CSE
(3)	Inlining	(21)	Common Subexpression Elimination
(4)	Profiling data insertion	(22)	Common Subexpression Elimination 2
(5)	Profiling data use	(23)	Common Subexpression Elimination 3
(6)	Renumbering pre CSE	(24)	Kill useless moves after CSE3
(7)	Common Subexpression Elimination	(25)	Forwarding moves
(8)	Static Prediction + inverting conditions	(26)	Dead Code Elimination
(9)	Loop peeling (unrolling one iteration)	(27)	RTL Branch Tunneling
(10)	Renumbering pre tail duplication	(28)	Set loads as non-trapping
(11)	Performing tail duplication	(29)	Unused globals
(12)	Renumbering pre unroll body	(30)	Renumbering pre BTL (BB)
(13)	Unrolling loop body	(31)	BTL BBpasses
(14)	Renumbering pre rotate	(32)	RTL Tunneling post LCT
(15)	Loop Rotate	(33)	Renumbering pre BTL (SB)
(16)	Renumbering pre constant prop.	(34)	BTL SBpasses
(17)	Constant propagation	(35)	Final renumbering
(18)	Interval propagation	(36)	Register allocation

Table 11.1: Comparison of RTL Passes Between Mainline CompCert and Chamois-CompCert (Verimag).

We have many more RTL passes. However, not all of them are activated by default. For instance, duplications passes such as loop peeling, loop body unrolling, tail duplication and loop rotation must be manually enabled with a command line option. In the table, numbers (31) and (34) correspond to sequences of BTL passes. The former operates on basic blocks and the latter on superblocks. Usually, the BTL basic block pass applies lazy code transformations, the expansion of macro-instructions, and store motion; while the superblock one is dedicated to our improved scheduling (including if-lifting).

This quick overview reveals *how difficult it is* to find an efficient order of passes. First, a given pass may undo the work of previous ones (e.g. LCT inserts moves while move forwarding removes them). Second, a pass may be less efficient or even impossible to apply if not preceded by some other pass (e.g. even if LCT adds moves, it benefits a lot from being placed after move forwarding). Third, many passes alter the CFG structure and thus require applying multiple time restructuring tools (e.g. the renumbering of RTL nodes which can be applied nine times if everything is activated).

Of course, we do not claim that the suggestion of Table 11.1 is optimal (it certainly is not). Nonetheless, considering our experiments, this order seems quite effective in improving code performance at runtime (as suggested by our experimental evaluation of Chapter 12).

Part IV

EVALUATION & CONCLUSION

This (final) part comprises two chapters.

Chapter 12 introduces our evaluation framework and methodology, and reviews the compile time and runtime measurements we conducted.

Chapter 13 summarizes our work and concludes.

12

TESTING AND EVALUATING A FORMALLY VERIFIED COMPILER†

We explain in this chapter our testing process. Section 12.1 motivates the need for testing, and reports on our methodology. We then perform multiple experiments to evaluate both compilation time and runtime performance in Sections 12.2 and 12.3, respectively. Finally, Section 12.4 takes a step back on this experiment and concludes.

12.1 GENERAL CONSIDERATIONS*

12.1.1 *What Are the Purposes of Testing?*

Testing is a very important part of the development, especially concerning verified compilers. It has at least three main objectives:

DETECTING COMPILE-TIME ERRORS CompCert 's formal proof ensures partial correctness: if compilation succeeds, then it is correct, but there is no formal proof that it succeeds. Remember §3.5: a compiler that always fails would trivially satisfy the partial correctness (formal) property. Hence, the first role of testing is to find and understand compile-time errors.

In particular, there is *no formal guarantee that our checkers will succeed* in validating our untrusted oracles (we prove their correctness, but not their completeness). Testing is therefore even more important considering the formally verified defensive programming approach: we want our oracles to be extensively tested on many kinds of source programs.

ACHIEVING REASONABLE COMPILATION TIMES Of course, we can consider normal that a fully verified compiler such as CompCert is a little slower than its unverified rivals. Nonetheless, the overall *compilation time must remain acceptable* and never explode even on huge source programs. Testing is needed to ensure this, notably because it allows to finely observe which passes are slowing down compilation.

MEASURING RUNTIME PERFORMANCE Finally, experimentally evaluating the effect of our optimizations is also a form of testing. It is necessary for us to get an idea of the performance of our generated code and to *compare ourselves* with mainstream compilers. Industrials using CompCert are mostly working with safety-critical systems, limited in their dynamic optimization power. Proposing a certified, yet optimizing compiler is thus highly desirable in many applications.

12.1.2 *Test Suites & Methodology*

We distinguish two types of tests: functional correctness oriented tests and representative, performance oriented tests.

12.1.2.1 *For Compiler Correctness*

AbsInt markets a version of CompCert suitable for qualification for safety-critical applications, e.g. nuclear power plants and avionics [88]. To our knowledge, this involves a large test suite, including the standard compliance suite SuperTest[1]. This test suite is not publically available.

The mainline, public releases of CompCert (on GitHub) also feature a test suite[2], composed of: (i) a few applications (a ray-tracer, the first-order prover Spass...); (ii) handwritten regression tests

*Some text in this section is adapted from our "Tests And Proofs" paper [108]†.

[1] See https://www.absint.com/ and https://solidsands.com/products/supertest.

[2] See directory "test" of https://github.com/AbsInt/CompCert/.

checking certain features (in the case of programs that are supposed to be able to be compiled and executed, the expected results are provided and compared with the results obtained); (iii) a program generator for testing application binary interface compatibility of data structure layout. We extended it with tests produced by off-the-shelf random generators, a form of *compiler fuzzing* [104], as well as the GCC "torture test" suite. For each program generator i (items 1. and 2. below), N_i programs are generated by varying the random seed of the generator from 0 to $N_i - 1$, ensuring reproducibility. In summary, we added the following functional tests:

1. Csmith 2.3.0 & YarpGen 1.1[3]. The produced code—which is supposed to be compilable and devoid of undefined behaviors—is compiled with both CompCert and GCC and run on the target processor or an instruction set simulator (e.g. QEMU). The results are then compared (*differential testing*). Yet this code may fail to terminate, thus a timeout is used; the test is considered valid if both programs yield the same value, or fail to terminate within the timeout. The timeout value is large enough to avoid cases where only one program, better optimized, terminates while the other does not, but would with more time.
2. CCG[4]. Its programs are not expected to run correctly, so we simply test that they compile correctly.
3. GCC 12.2.0 tests. Finally, we added GCC's C torture tests, both for compilation only and for compilation + execution, except those that relied on GCC-specific extensions (such as SIMD[5] vectors), GCC-specific behaviors on undefined or unspecified cases, and those that tested the limits of the compiler (i.e. very large number of declarations).

Each newly added generator or suite triggered new bugs (in our own extensions, or in upstream recent extensions not yet covered by AbsInt's tests). The full test suite, including the three items above, is launched in our *continuous integration* for a variety of targets[6].

See our TAP'23 paper (from which this section is inspired) for more on these topics [108]†.

REDUCING BUGS Generally, test cases that triggered bugs had to be reduced before the bug could be investigated. Random and application test cases are often too large for the compiler developer to identify bugs. Finding a reduced test case that exhibits the same bug is the first step for understanding what went wrong (as recommended by the GCC bug reporting guidelines). Reducing cases by hand is tedious and error-prone; we thus automated this task using C-Reduce[7].

By detecting compilation failures, testing gives us greater confidence in the ability of our validators to accept transformations of our oracles. In addition, tests helped to discover miscompilations and intolerable compilation running-times.

12.1.2.2 *For Performance*

Developing a new pass requires ensuring that its results are not only *beneficial* on every target, but also that it does not *interfere* with existing optimizations [42]. Since most instruction set simulators are not capable of counting cycles, we measure and compare the code performance of various CompCert configurations directly on the target core.

Firstly, such measures are often subject to many *subtle biases* [113], among which are the runtime environment, the size of the benchmarks, as well as decisions by the operating system kernel: frequency scaling, migration between cores, etc. We address this by running multiple execution of each test, and by forcing the process to remain on the same core (e.g. with `taskset`), under the same

[3] https://github.com/csmith-project/csmith [147] with packed structures (GCC extension) disabled. https://github.com/intel/yarpgen [98] (One random seed value is excluded because on ARM it leads to register allocation causing out of memory. Large auto-generated programs causing resource exhaustion in the compiler is not considered a bug [91].)

[4] https://github.com/Mrktn/ccg. We disabled the generation of ternary conditional operators with omitted middle operand, a GCC extension not supported by CompCert.

[5] Single Instruction Multiple Data.

[6] x86, x86-64, AArch64, ARMv7 with software and hardware floating-point, 32-bit PowerPC, 64-bit RISC-V, KVX. This even led us to find bugs in QEMU for PowerPC.

[7] C-Reduce [122] is a software able to reduce a source program given an interestingness predicate (e.g. a shell script) that indicates if the pruned source still contains the bug.

shell environment. Then, we average out the different executions to filter them when the *relative standard deviation* (*RSD*) exceeds a certain threshold (*noise elimination*), so that too small or unreliable tests are removed.

Secondly, for the comparison to GCC or Clang to be *fair*, we have to compare compilers on a common basis of applicable transformations. Notably, we disable options that would not be correct in the CompCert semantics—e.g. "fast-math", or replacing "$a \times b + c$" by a fused multiply-add[8], and instruction set extensions that CompCert cannot use—e.g. vector (SIMD[5]) instructions. In contrast, options that enable finely-dependent processor optimizations, e.g. "`-march=armv8-a+nosimd -mtune=cortex-a53`" for the AArch64 Cortex-A53 core or "`-march=rv64imafdc -mcpu=sifive-u74 -mtune=sifive-7-series`" for the RISC-V SiFive U740 core, are left active.

For the measure to be *representative* and to avoid concluding on an overfitted subset of benchmarks, we combine five benchmarks suites: (i) a subset of the widely diversified LLVMtest suite[9]; (ii) a subset of the MiBench [70] embedded system oriented suite; (iii) the full polyhedral, computational oriented PolyBench [120] suite; (iv) the full TACLeBench [53] WCET specialized suite; and (v) our own test suite at Verimag featuring various concrete applications.

We developed a performance measuring toolkit[10] (this is a joint work with Olivier Lebeltel—a research engineer at Verimag—and myself), based on a JSON configuration that details, for each compiler to measure, sets of options to compare. Shell scripts then automatically (i) build; (ii) copy to the target machine (e.g. via `rsync`); (iii) run N times on a fixed core; and (iv) gather tests results as CSV files. Finally, a Python/Pandas script filters and analyses CSVs to yield (in text or as a plot) the observed gains w.r.t. a reference compiler with its set of options.

12.2 COMPILATION TIME (ON RISC-V)

In this section, I focus on measuring the compilation time of CompCert on various tests and for various optimizations. I only present the experiment for RISC-V, as the results we obtained on other architectures are similar.

12.2.1 *BTL Translation Validation Time of LCT*

To ensure that our validator scales well even on large applications, we instrumented the OCaml code generated from Coq to time both the LCT oracle and the symbolic execution engine. The validator is run 10 times for every function of every source program, so that we average timings to get more accurate results. For this experiment, we activated the full LCT (including its SR component), with a fixed maximum of candidates (to 64). Loop peeling and integer promotion were also enabled.

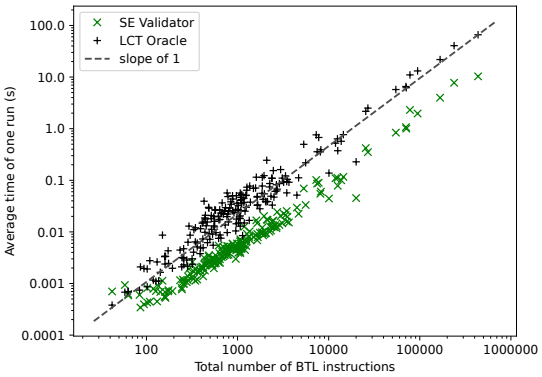


Figure 12.1: LCT Oracle and Validator Times w.r.t. the Number of Instructions (logarithmic scale).

We tested this setting over every performance benchmark from our five suites. Figure 12.1 graphically represents those timings measures w.r.t. the *total number of BTL instructions*, including assignments in history and gluing invariants. Each point in the figure correspond to a single benchmark whose (average) timings and number of instructions were summed for all its BTL functions.

The worst validation time was of approximately 10 seconds. This timing occurred on the GLPK benchmark (a run of the GNU Linear Programming Kit with a large input); LCT took 66 second to optimize this benchmark (i.e. much longer that the time required to validate its result). The

[8] An "fma" rounds differently from a multiplication followed by an addition. Replacing the latter by the former thus is a semantic change, which runs afoul of CompCert's soundness criteria.

[9] Accessible at https://github.com/lac-dcc/Benchmarks.

[10] Our benchmarks and toolkit: https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/chamois-benchs.

second-longest validation time was on Spass and took 8 seconds (40 seconds for LCT). The two following ones were LLVM benchmarks: MAFFT (Multiple Alignment using Fast Fourier Transform, a bioinformatics program) and SMG2000 (a semi-coarsing multigrid solver for linear systems, known as "ASCI purple") whose validation took 4 and 2 seconds, respectively (21 and 11 seconds for LCT, respectively). All the remaining 201 benchmarks were validated in less than 2 seconds, and took less than 10 seconds to optimize with LCT (this mainly indicates that the four aforementioned benchmarks are outliers[11]).

We observe an almost perfect linear correlation between the validator's and the LCT oracle's execution times, near 99%. Put another way, with the threshold on the number of LCT candidates, the oracle seems linear in the number of instructions per BTL function. On average, the validator seems *a bit faster than the oracle* for a given benchmark size.

12.2.2 *Time of Other Passes*

CompCert already include an option (`-timings`) to measure the execution time of each compiler pass. In this section, we compare the overhead induced by BTL passes w.r.t. other passes (cf. Table 11.1).

I manually selected four benchmarks among those with the longest compilation time, with the following passes enabled: the CSE3 of Monniaux and Six [109], the RISC-V macro-expansions of §7.6.1, the promotion of Benjamin Bonneau of §11.2, the alias aware superblock scheduling (without if-lifting) of §11.3.2, and my LCT optimization of Chapter 10.

Colored blocks in Figure 12.2 are ordered by decreasing execution times of the first column (bench ASCI purple).

The result is plotted as a stacked histogram in Figure 12.2. For the sake of readability, I only represented the 10 slowest passes. Each colored block in the histogram corresponds to the execution time of a specific pass during compilation. Colored blocks of passes that took more than ten seconds to execute have their total execution time annotated in white, in the middle of the block (i.e. which is equal to the block's height). For example, the original CompCert CSE (in pink color) took eighteen seconds to execute on benchmark GLPK (from our custom suite at Verimag), while it took at most ten seconds on MAFFT (from LLVMtest). The total height of a histogram column thus corresponds to the total compilation time of the corresponding benchmark—e.g. approximately one hundred seconds for ASCI purple (LLVMtest again).

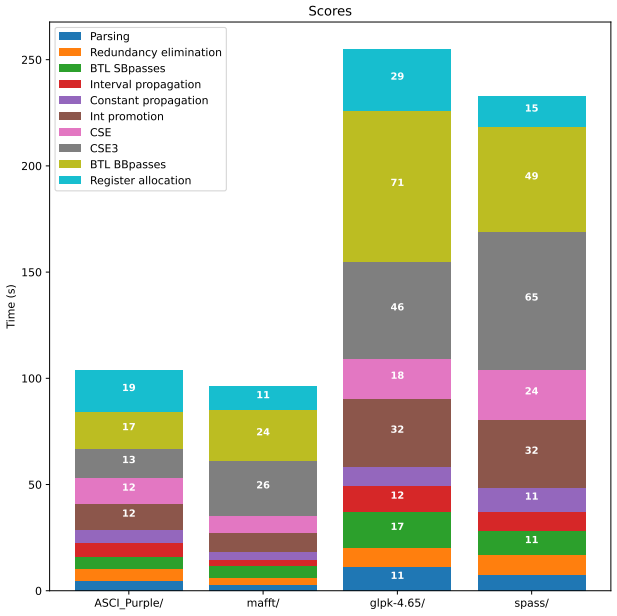


Figure 12.2: Top Ten Slowest Chamois-CompCert Passes Across Four Selected Benchmarks.

It appears that each time, four passes are taking more than half of the compilation time. Our sequence of BTL passes in basic blocks comes out on top for the GLPK benchmark, and CSE3 comes first on both MAFFT and Spass (Verimag's suite). The situation is a bit different on ASCI purple, where register allocation is the slowest pass. The original CompCert's CSE and the integer promotion passes are also among the slowest, and are even taking more time than register allocation on Spass.

Globally, we are fairly satisfied of these results, because they demonstrate that our basic block passes (comprising two translations, macro-expansions, LCT, and SE validation) operate in a reasonable amount of time w.r.t. other passes. Seeing compilation times between one and five minutes for these four benchmarks may seem excessively long. This is true when compared to GCC, which takes around eight times less time to compile them. Nevertheless, it should be borne in mind that CompCert is written in Coq and OCaml (i.e. languages less optimized than C++), and that the

[11] The compilation time analysis proposed in our OOPSLA'23 and ICOOOLPS'23 papers [64, 65]† did not include the Verimag custom benchmark suite. That is why their worst-case validation time is only 4 seconds (and their worst-case LCT optimization time is approximately 10 seconds).

four programs chosen are very heavy (e.g. ~75k significant lines of code for GLPK). This slow compile time could also be attributed to the use of inefficient data structures in the oracles, as it the case, for instance, in some parts of the CSE3's implementation. Since most embedded programs are significantly smaller, compilation time is rarely an issue in practice.

There are several ways to improve LCT execution time. For instance, we could implement a parallelized version of the data-flow analyses; set a lower threshold for the maximum number of candidates; use more efficient data structures; or work on reducing the algorithmic complexity of certain parts.

It is important to remember that our goal when implementing LCT was mainly to make a proof of concept of our validation technique: we do not paid much attention to its execution time. Now that we are convinced of the applicability of our solution, it could be interesting to research more efficient solutions in terms of both execution time and optimization power.

12.3 RUNTIME PERFORMANCE

And, when using LCT, we bound the number of candidates to 64.

Following the methodology sketched in §12.1.2.2, we run each benchmark 10 times, setting the relative standard deviation threshold to 2%. Typically, when executing benchmarks on small cores, the deviation remains relatively minor, leading to the exclusion of only a few benchmarks or none at all. However, this threshold proves invaluable for larger cores, where some benchmarks finish almost instantaneously, leading to less precise measurements.

All our results reflect the runtime performance gain, expressed as a percentage w.r.t. a reference version. A lower number indicates slower execution, and vice versa. The term runtime performance gain is used to compare the *outcomes* of a specific configuration against a reference version. These outcomes are execution times measured in number of *CPU clock cycles* for a given benchmark. Let C represent the result from the specific configuration and R denote the outcome from the reference version (both in cycles). The gain is then calculated using the formula: $\mathit{gain}(C) = ((R - C)/C) \times 100$. The latter gives the evolution rate (in percentage) relatively to R. In essence, it computes how much C deviates from R as a percentage of C. For instance, if $R = 1000$ and $C = 500$ cycles—so C is twice faster—then $\mathit{gain}(C) = 100\%$; in contrast, if $R = 500$ and $C = 1000$—so C is twice slower—we have $\mathit{gain}(C) = -50\%$.

The gain for a complete series of benchmarks is derived from either the mean or the median cycle count of each benchmark.

HARDWARE PLATFORMS We conducted our performance evaluation over three architectures. For AArch64, a Cortex-A53 in-order dual-issue core (Raspberry Pi 3 Model B+ Rev 1.3), and a Cortex-A72 out-of-order three-issue core (Raspberry Pi 4 Model B Rev 1.1). Both cores are superscalar, but the Cortex-A72 is less dependent on compiler optimizations thanks to its OoO pipeline and its speculative execution capability. And for RISC-V, a SiFive in-order dual-issue U740 core (HiFive Unmatched).

SOFTWARE VERSIONS Throughout this section, we consistently use specific versions of three distinct compilers for comparison. "Mainline" CompCert designates the mainline CompCert version (`3.12`) from the AbsInt GitHub repository[12]. GCC, for both AArch64 or RISC-V, is in version `11.4.0`. And Chamois-CompCert is the version of the Verimag repository[13], including the optional extensions of Chapter 11. Our experiments always juxtapose these three compilers. The aim is to discern the improvements already realized over the mainline CompCert and to gauge the distance we need to cover to match the performance of GCC.

12.3.1 *Lazy Code Transformations*

We focus here in evaluating the LCT algorithm on both AArch64 and RISC-V.

[12] Commit hash `35feefd229792e6b560ccf156465a6e309bc1d98`.
[13] Same URL as in Chapter 11: `https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert`.

Setup (versus "Base1")	Cortex-A53	Cortex-A72
GCC −O1	+54.7%	+35.5%
GCC −O2	+69%	+99.5%
Mainline CompCert	−18.5%	−1.9%
Base1+Prepass	+4%	+0.2%
Base1+LCM	+10.4%	+10.2%
Base1+Prepass+LCM	+14.4%	+10.2%
Base1+Prepass+LCT (CM & SR)	+17%	+12.7%

Table 12.1: Mean LCT Gains on PolyBench for Two AArch64 Cores.

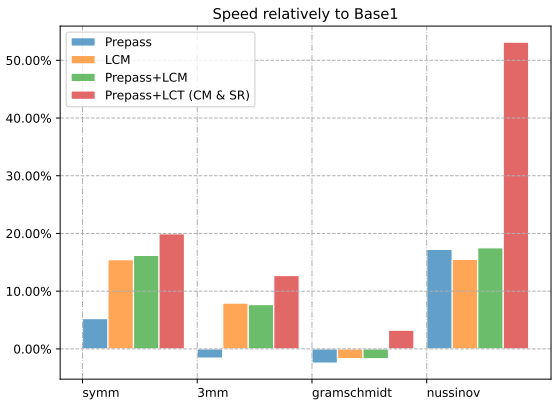


Figure 12.3: Chamois-CompCert Configurations for Four Selected PolyBench on Cortex-A53.

12.3.1.1 *On AArch64*

The SR backend of LCT was ported to AArch64 by Hugo Strappazzon, a Verimag intern I co-supervised. This work consisted firstly in writing the oracle's module allowing to detect and reduce SR candidates, and secondly in formalizing and proving the associated affine normalization rules for AArch64 instructions. It took much less time to realize this port than the first implementation on RISC-V, as most of the affine normalization engine is defined in a polymorphic way.

Unfortunately, we have not yet ported the integer promotion of Benjamin Bonneau to this architecture, meaning that the new SR backend can only operate on variables defined as `long` in the C source. Because of this restricted scope, the performance measurements on our benchmarks suites do not show any significant gain. In order to still observe the impact of this SR port, I adapted the PolyBench suite by replacing (in a very ugly way) `int` variables with `long`. This rather dirty method is just to show that the optimization works and is useful.

Recall that gains are the average over 10 executions, after RSD filtering.

Results are listed in Table 12.1, for both the Cortex-A53 and the Cortex-A72 cores. The reference configuration is called "Base1" and corresponds to Chamois-CompCert without CSE3, prepass, or LCT, but with the postpass scheduler of §4.3. This notably explains why the performance of mainline CompCert is much slower than "Base1" on the Cortex-A53 (which is in-order). In contrast, the postpass being almost ineffective on OoO cores[14], mainline CompCert and Chamois-CompCert produce comparable results on the Cortex-A72 (1.9% difference).

This behavior does not apply to lazy code motion, which speeds up the runtime performance of approximately 10% on both cores. LSR, for its part, produces a less important improvement on average, but which is still encouraging considering its embryonic stage.

Prepass scheduling in this experiment is the one with alias analyses from §11.3.2.

Furthermore, if we look at the results in details, it appears that albeit LSR is not effective on every benchmark (hence the small difference in average), its effect is significant when actually applied. Figure 12.3 demonstrates this claim on four individually selected benchmarks for the Cortex-A53. To make this plot easier to read, only Chamois-CompCert configurations are displayed (compared relatively to "Base1"). For example, the performance gain on benchmark Nussinov (kernel computation for molecule origami) jumps from 17.5% to 53.2% by applying strength-reduction.

12.3.1.2 *On RISC-V*

Since the RISC-V backend benefits from the integer promotion pass, we were able to benchmark the LCT's strength-reduction on a larger set of benchmarks, and without needing to manually modify sources.

[14] In practice, we still observe a small gain (usually around 2% difference). As said in margin of §4.1, this may be due to a small reordering buffer in the physical core. In addition to the postpass, the Chamois-CompCert improvement in Table 12.1 may also be due to move-forwarding. For that same reason of dynamic reordering, we can see that prepass scheduling is only significant on the Cortex-A53 (without postpass, the small scheduling improvement would have been gained by the prepass).

Setup (versus "Base2")	GCC −O1	Mainline CompCert	C_1 =Base2 +Prepass	C_2 = C_1+LCM	C_3 = C_2+ Promotion+ LCT (CM & SR)	C_3+CSE3
LLVMtest/CrystalMk	+50.8%	+0.3%	+12.9%	+15.5%	+27.3%	+29%
LLVMtest/fl-dhrystone	+86.8%	−5%	+6.8%	+12.3%	+12.7%	+2.2%
LLVMtest/fp-convert	+24.2%	+0%	+7.9%	+13.4%	+17.1%	+17.1%
LLVMtest/nbench-num	−9.2%	−9.3%	+6.3%	+5.6%	+5.8%	−9.8%
LLVMtest/nbench-str	+17.7%	−7.8%	+1.9%	+7.5%	+9.5%	+10.2%
LLVMtest/nbench-bit	+75.1%	+0%	+12%	+33.5%	+33.5%	+33.7%
LLVMtest/nbench-id	+14.7%	−2.9%	+0%	−0.1%	+9.2%	+12.4%
PolyBench/*	+64.9%	−1.2%	+21.7%	+25.7%	+32.7%	+41.7%
MiBench/stringsearch	+134.6%	+0.1%	−0.5%	+15.9%	+38.3%	+39.3%
MiBench/blowfish	+11.4%	−1.3%	+7%	+8.9%	+9.6%	+12.2%
MiBench/sha	+93.4%	+0.3%	+36.6%	+35%	+38.8%	+53%
TACLeBench/*	+37.4%	−3%	+12%	+16%	+20.2%	+21.3%

Table 12.2: Comparing LCT, Promotion, CSE3, and Prepass With GCC and Mainline CompCert on Both Individual Benchmarks and Complete Suites, on the RISC-V U740 Core.

Nevertheless, our SR is still inapplicable to many benchmarks. The primary obstacle is the limitation of our promotion pass, which cannot access signedness information in RTL (cf. §11.2). Under these conditions, observing the SR effect on a full benchmark suite can obscure its true impact in the average. Considering this difficulty, we opted, when relevant, to investigate results on individual benchmarks where the optimization could be applied. Our results in Table 12.2 list some individual benchmarks, except for PolyBench and TACLeBench. We retained the entire suites for these as they appear to be very well-supported by our LCT and promotion passes.

To prevent any misattribution, we added optimizations incrementally: first prepass scheduling, then CM, followed by promotion and SR, and finally CSE3. The reference version, named "Base2", is thus defined as "Base1" from §12.3.1.1 but without postpass, since it does not yet exist for RISC-V. We intentionally group integer promotion and SR together because applying them separately does not always yield benefits. Specifically, promoting variables without strength-reducing them can improve results in some cases, but it can also significantly hinder performance. Conversely, applying our 64-bit restricted SR without promotion also yields varied outcomes. We generally achieve the best performance by combining both optimizations.

From the measures in Table 12.2, we draw the following conclusions: (i) the "Base2" version is nearly equivalent to mainline CompCert; (ii) the effect of optimizations varies considerably depending on the benchmark; (iii) overall, both prepass scheduling and LCT are generally effective and significantly enhance runtime performance; (iv) CSE3, which is applied *before* BTL, is generally effective, but there are exceptions, such as in LLVMtest/fl-dhrystone or LLVMtest/nbench-num; (v) In most cases, we are still notably behind `gcc -O1`.

12.3.1.3 *Comparative Analysis and Challenges of Our LCT Implementation*

Another interesting result to consider is the comparative impact of our best configurations w.r.t. the mainline CompCert version. To this end, we plotted the runtime gains of four incremental setups in Figure 12.4. First, we consider the combination of prepass, CSE3, and loop peeling; next, we introduce LCM, followed by integer promotion; finally, we add LSR. For this evaluation, I handpicked ten pertinent benchmarks.

My goal with this comparison is to highlight that our lazy code transformations algorithm is not always beneficial, and preemptively determining its potential inefficacy is complex. Notably, there are instances where LCT has negligible effect, primarily due to its reliance on promoted integers; and there are also instances where it can adversely impact runtime performance. Such a predicament manifests in "stringsearch" benchmark from MiBench, revealing two distinct challenges:

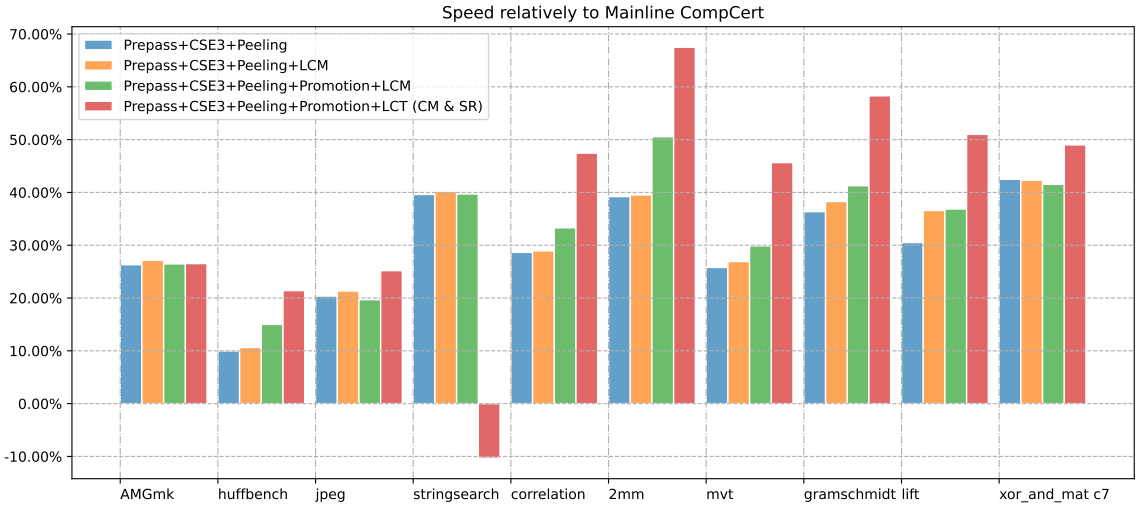


Figure 12.4: CHAMOIS-COMPCERT Configurations for Ten Selected Benchmarks vs. Mainline COMPCERT on U740.

1. **A scope problem:** this benchmark searches for substrings within strings. For each string in the input dataset, it starts by initializing an array structure. This procedure is executed by a function invoked for each string. At theoretical level, the optimization applied by LCT appears advantageous: it removes a left shift and an addition instruction from a loop, substituting the sequence with a single immediate addition as compensation code. Moreover, given that we can determine the number of iterations in advance, which is capped by `UCHAR_MAX` (255 on 64-bit RISC-V), and considering the shift's one cycle latency, a gain of 255 cycles seems plausible. However, several unforeseen inefficiencies emerge. The removed addition existed between a load and a store, which were three instructions apart. The subsequent store depends on the preceding load. Due to the addition elimination, the gap between the two memory accesses is reduced to two instructions, each bearing one cycle latency. Loads on the U740 exhibit a three cycles latency, and this gap, left unaddressed by scheduling, nullifies the cycle saved, causing the core to stall. Additionally, the compensation code, incorporated into the unrolled first loop iteration (recall the application of loop peeling here), also loses two cycles due to suboptimal scheduling. This, combined with the introduction of two new instructions each costing one cycle, results in the function shedding at least four cycles in the loop header. Combined again with the loop body new stall, and coupled with slightly more than two thousands, six hundreds invocations of the initialization function, the outcome drops to the poor result we observed.

   Intriguingly, this complication arises solely when loop peeling is active (though SR remains functional without it), and the cycle reduction is not discernible on all RISC-V cores. We only observed this on the U740. In contrast, when I assessed the same optimized code on a RISC-V Rocket chip (Berkeley design, emulated on FPGA[15]), it did not exhibit a similar performance loss. This discrepancy is likely attributed to the Rocket core's single issue pipeline, which renders scheduling less impactful for its performance.

   Ultimately, this suboptimal result can be imputed to flawed optimization hypotheses. Our heuristic postulates that: (i) the scheduler will accommodate the voids left by partial redundancy elimination; (ii) positioning compensation code before the loop, even if it leads to four additional cycles, would be inconsequential relative to potential gains in the loop body. Ineluctably, our oracle remained oblivious to the vast number of invocations.

   In prospective endeavors, if we achieve the implementation of inter-procedural transformations, specific addressing calculations in this benchmark might be wholly extricated from the initialization function, significantly enhancing its runtime performance.

[15]Field-Programmable Gate Arrays are versatile integrated circuits predominantly employed for prototyping.

Setup (versus Mainline CompCert)	Cortex-A53 (AArch64)	Cortex-A72 (AArch64)	SiFive U740 (RISC-V)
GCC −O1	+50.8%	+35.1%	+47.4%
GCC −O2	+73%	+51.5%	+65.9%
C_1 =Prepass[+Postpass for AArch64]	+17.9%	+1.1%	+18.5%
C_2 = C_1+Unroll body	+26.8%	+9.8%	+24.4%
C_3 = C_2+Loop rotate	+29%	+10.8%	+25.4%
C_4 = C_3+Register renaming	+33.9%	+11.6%	+28.1%
C_5 = C_4+If-lifting	+34.6%	+11.1%	+30.6%
C_6 = C_5+LCM[+LSR+Promotion for RISC-V]	+37.4%	+12.6%	+35.4%
C_7 = C_6+CSE3	+39.3%	+14.7%	+37.2%

Table 12.3: Mean Gain of Incrementally Adding Scheduling, Duplications, Renaming, If-Lifting, LCT, and CSE3 on All Cores, for All Benchmark Suites, and Comparing With GCC.

2. **A register pressure problem:** the "stringsearch" benchmark (and this observation extends to other benchmarks) grapples with elevated register pressure, exacerbated by loop peeling. This primarily arises due to the static initialization of the dataset within the main function. This challenge can be addressed in two ways. First, gauging the dataset initialization, which is executed only once, is arguably superfluous (this differs from the array initialization mentioned in the preceding point). Secondly, this complication is easily mitigated by bounding the number of LCT candidate to a threshold, so that it does not increase register pressure over the critical point.

   In the future, we believe that it would nonetheless be beneficial to create a register pressure sensitive LCT, as Nicolas Nardino [117] did for superblock scheduling.

Apart from these situations, which are admittedly difficult to anticipate, the strength-reduction applied by LCT is proving highly effective in the selection of Figure 12.4. The gain of CM is less pronounced in this handpicked set than it was in the results of Table 12.2 (and of Table 12.1 for AArch64); but this is explained by its application post-CSE3, which, when coupled with loop peeling, already addresses most redundancy eliminations.

12.3.2 *If-Lifting*

In this section, we evaluate the impact of register renaming and if-lifting of [4]. In conjunction with loop body unrolling, these three transformations emulate the weak form of software pipelining detailed in §11.3.1. We also add loop rotation, as it is known to help in hoisting conditionals.

Acts of unrolling loop bodies and rotating loops are expected to amplify the opportunities for LCT, scheduling, and CSE3. We incrementally integrated them into our set of configurations, which means our comparison, while primarily focused on, is not solely confined to if-lifting.

To make easier the comparison between the AArch64 and RISC-V backends, all performance gains assessed in this section reference the mainline CompCert as the baseline. We conducted measurements on our three target processors: Cortex-A53, Cortex-A72, and U740. I have collated the results from our five benchmark series by calculating the mean gain for each configuration, as showcased in Table 12.3.

At a glance, both GCC optimization levels yield significant performance improvements across all processors, with the Cortex-A72 exhibiting a notably lesser gain compared to the others. This is expected considering its inherent hardware capabilities for speculation and reordering. Beginning with configuration C_1, which incorporates scheduling, we observe a marked performance uplift on both in-order cores. The layering of subsequent configurations highlights a steady increase in performance for all cores. Interestingly, naive code duplications like C_2 and C_3 in the table suffice by themselves to produce an improvement, thanks to the opportunities they offer. The introduction of the LCM along with the LSR and promotion (only for RISC-V) in C_6 and the final incorporation of CSE3 in C_7 intensifies these gains further.

Setup of	LLVMtest		MiBench		PolyBench		TACLeBench		Verimag	
Tbl. 12.3	A53	U740	A53	U740	A53	U740	A53	U740	A53	U740
C_4	+25.8%	+23.4%	+19.6%	+15.7%	+62.9%	+35.2%	+32.5%	+17.5%	+33.7%	+44.6%
C_5	+27%	+24.5%	+23.1%	+20.9%	+64.6%	+37.3%	+37.6%	+21.1%	+34.2%	+48.3%

Table 12.4: Comparing Configuration C_4 and C_5 From Table 12.3 for Each Benchmark Suite.

Setup (versus "Base3")	LLVMtest	MiBench	PolyBench	TACLeBench	Verimag
GCC −O1	+50.2%	+13.1%	+107.8%	+38.3%	+12%
GCC −O2	+65.4%	+42%	+130.7%	+88.4%	+47.4%
Mainline CompCert	+1%	+0.3%	−0.4%	−0.1%	−1%
C_1 =Postpass+Peephole	+7.2%	+8.7%	+29.6%	+4.2%	+10.4%
C_2 =Prepass	+11.8%	+12.6%	+34.6%	+12.2%	+13.5%
$C_3 = C_1 + C_2$	+13.1%	+13.7%	+37.4%	+14.1%	+15.5%
$C_4 = C_3$+Loop peeling	+14.3%	+14.9%	+38.7%	+16.6%	+16.8%
C_4+CSE3+LCM	+18.7%	+24%	+65%	+23.8%	+18%

Table 12.5: Mean Gain From Schedulers, Unrolling, and Redundancy Elimination Algorithms on Cortex-A53, With GCC and Mainline CompCert.

To delve deeper into the influence of if-lifting (C_5), we sharpened our analysis by examining its impact on individual benchmark suites. Table 12.4 offers such a breakdown, built upon the same data as Table 12.3. This analysis omits other optimizations to strictly spotlight the distinction between configurations C_4 and C_5. We have sidestepped the Cortex-A72 platforms for this particular comparison, because it brings nothing more than the global one—i.e. almost no effect or even small losses on every suite.

From this granular perspective, it is evident that if-lifting is indeed beneficial for in-order cores, albeit with some variations across benchmark suites. The most pronounced effect appear in the MiBench and TACLeBench suites, with an increase of approximately 4%.

Finally, each incremental enhancement contributes mostly positively to the performance across all target processors, reinforcing the potency of the evaluated transformations.

12.3.3 *Prepass, Postpass, and Peephole on AArch64*

We tested our postpass scheduling and peephole optimizer on AArch64. Since the Cortex-A72 has an OoO pipeline, the effect of any scheduling, whether in prepass or in postpass, is negligible[14]. This section thus only presents results on the Cortex-A53 core.

The experiment was carried out on all five benchmark series, with a "Base3" reference configuration corresponding to Chamois-CompCert without CSE3, prepass, postpass, or LCT. In addition to comparing both schedulers, we observe the effect of combining loop peeling with LCM and CSE3.

Our measurements are given in Table 12.5, grouped by benchmarks series. Mainline CompCert is, as expected, approximately equivalent to our Chamois-CompCert fork when every important pass is disabled.

SIDE-NOTE ON THE PEEPHOLE (OF §4.3.3.2) Note that here, I grouped the application of the postpass scheduling with its peephole optimizer pairing loads and stores. I did so because the Cortex-A53, although dual-issue, has only a single load-store unit. The peephole pairing has, therefore, no significant effect on performance. In some rare cases, it is still able to slightly increase performance by offering new opportunities after pairing. Another interest of this peephole is its capacity to reduce code length: such considerations are sometimes important in embedded systems.

Anyway, we did not design the peephole only for loads and stores pairing but also to use a larger part of the ISA, and as a foundation for later similar peepholes. For instance, one would like to implement the ANDS or BICS instructions, which are not supported in RTL because of the language limitations (recall §3.4.2, and especially Footnote 7: the peephole can help us circumvent the RTL unicity limitation on destination registers).

SIDE-NOTE ON THE ALIAS AWARE PREPASS (OF §11.3.2) As for all results of this chapter, the prepass scheduling we use includes the relative and absolute alias analyses. While I do not present the gains resulting from these analyses compared to the "naive" prepass, we generally observe an improvement on the order of approximately 2%.

Coming back to our results, the postpass+peephole configuration seems less efficient than the prepass alone. Indeed, the prepass being applied on superblocks, and being aware of aliases, this result is not surprising. Despite this, Table 12.5 demonstrates a complementarity between these two schedulers: in fact, this is certainly due to the postpass ability in reordering spills. We then incrementally add loop peeling, resulting in a small gain on every suite; and both CSE3 & LCM, resulting this time in a much more considerable gain. The latter increase is made possible by the former loop peeling, enabling CSE3 & LCM to perform loop-invariant code motion of potentially trapping instructions.

Overall, our best CHAMOIS-COMPCERT configuration (last line of Table 12.5) outperforms "`gcc -O1`" on two benchmark suites. However, we still lag behind GCC's performance on other suites, and the gap widens when compared to "`gcc -O2`"; a lot of work thus remains to be done.

12.4 DISCUSSION

We saw that our optimization framework based on block transfer language was able to validate many optimizations in a reasonable amount of time. Our optimizations target mainly AArch64 and RISC-V platforms, with specific attention given to in-order processors. Nonetheless, the lazy code motion component of our LCT pass, as well as scheduling related optimizations, are also applicable for other architectures that are used in embedded systems (e.g. ARMv7, PowerPC, etc.) As demonstrated in Tables 12.1 and 12.3, some of our optimizations even benefit out-of-order cores like the Cortex-A72.

In summary, we observed important performance gains (close to 40%) compared to mainline COMPCERT on both the AArch64 and RISC-V in-order cores. Albeit our optimizations are in general less effective for OoO processors, we still succeeded to obtain significant increases thanks to duplications, CSE3, and especially code motion. We largely reduced the gap with the first GCC's optimization level, and our optimizations even outperformed it on some benchmarks. Our formally verified SR, albeit only targeting 64-bit integers or requiring a prior promotion, already yields very promising results encouraging us to extend its application.

Continuing this work would thus imply adding more complex optimizations, including finer alias analyses, linear-function test replacement, and more accurate pipeline models. In addition, we believe that extending the ISA support is also desirable. On AArch64, some features are not yet exploited by CHAMOIS-COMPCERT (e.g. pre and post addressing modes, arithmetic-comparisons) and on RISC-V, we could implement ISA extensions designed for SCS applications.

Finally, our experiments show that the BTL validation mechanism can help in producing more efficient code for multiple architectures, as long as we succeed to implement and validate the relevant transformations for the architecture.

In the context of our OOPSLA'23 paper [65]†, we published an artifact [66]† allowing anyone to reproduce these results. The artifact contains both a version of our CHAMOIS-COMPCERT fork and our benchmarking toolkit. It provides a step-by-step explanation on how to reproduce the effects of our optimizations on specific handcrafted examples, and on how to benchmark CHAMOIS-COMPCERT on specific hardware (AArch64 & RISC-V).

13

CONCLUSION

This chapter summarizes my work, our insights on the translation validation by symbolic execution topic, and opens the door to future works.

13.1 SHORT SUMMARY

Proving compiler optimizations correct is a challenging task. In the well-established research area of translation validation, the mainline approach initiated by Pnueli, Zuck et al. uses verification condition generators with theorem proving. A less-investigated alternative, pioneered by Necula, only combined static analyses with symbolic execution and normalized rewriting. Yet, recent advances in applying this technique within industrial scale certified compilers inspired further exploration, with the ambitious aim of formally validating intra-procedural optimizations.

Our work realized this goal, demonstrating that SE is indeed a viable strategy to improve the optimization capabilities of formally verified compilers. We selected the CompCert formally verified compiler by Xavier Leroy as our object of study, which is recognized as one of the most successful projects in this field.

Benefiting from the substantial research embodied in this project, we developed and integrated a formally verified SE framework. This tool leverages a defensive simulation test, modulo invariants provided by optimization oracles. Through a co-design development of this validator and oracles, we successfully validated code motion and strength-reduction, two optimizations that were missing in CompCert. Moreover, thanks to the work of other researchers, notably my colleagues at Verimag, we had the opportunity to test our validation tool over several more optimizations. These include alias aware scheduling, elimination of redundant stores, code factorization, and even a weak form of software pipelining.

Considering rising architectures in safety-critical systems, such as AArch64 and RISC-V, we tailored our work towards in-order, predictable processors.

The results indicated significant runtime performance enhancements on these platforms. Our version of CompCert, named Chamois-CompCert, now rivals the first optimization level of GCC.

The designs we proposed throughout this thesis seem applicable beyond CompCert.

Although we are admittedly far from closing the gap with the most powerful optimization levels of mainstream compilers, our approach has enabled us to learn important lessons about the way forward. I summarize those insights in the following section.

A compact overview of the development size in this thesis, in number of significant lines of codes (excluding blank lines and comments), per project, is given in the table below:

Project	**Ocaml**	**Coq**
BTL oracles & framework	3332	10 932
AArch64 scheduling & peephole	1157	11 171
Total	4489	22 103

For BTL, the items included in the "oracles" count are LCT, expansions (of macro-instructions), translators (from and to RTL, including factorization), liveness analysis, renumbering algorithm, and shadow fields type definitions (which are all my contributions).

13.2 INSIGHTS*

With this thesis, we argue that advanced compiler optimizations can be implemented with a formal proof of correctness by:

*Those insights are adapted from our OOPSLA'23 paper Gourdin et al. [65]†.

- splitting them into individual phases with well-defined, easy to understand functionality and independent correctness proofs (e.g. one phase that reorganizes code followed by one phase that leverages this reorganization to perform simplifications);
- splitting complex phases into:
  - an untrusted oracle, which computes the transformed code, or at least some mapping between the original and transformed code, and possibly some extra annotations such as invariants;
  - a formally verified interpreter, which uses and checks these mappings and annotations to establish simulation between the original and transformed code (the oracle must be designed so that everything it does is understood by the interpreter).

By construction, such an interpreter must redo some computations originally done by the oracle. "Only do *simple* computations in the formally verified interpreter" is our motto. In particular, complex computations should be avoided by appropriate hints (a.k.a. certificates) from oracles. It is indeed much easier and more efficient to directly extract correctness arguments from the existing computations of the oracle, than to rediscover them a posteriori from scratch in the interpreter. The lazy code transformations oracle of Chapter 10 is a perfect example. In summary, this motto leads us to design interpreters that are efficient while keeping a manageable proof of correctness. Moreover, such simple designs often lead to interpreters that are appliable to various kinds of transformations, and not just a single one.

In order to formally prove the transformations, the semantic arguments used for correctness must be precisely identified. They will be turned into invariants and "match" relations used for simulation proofs. This remains nontrivial work. Some unexpected semantic issues may arise about seemingly trivial matters. For instance, one may think that "$(a + b) - b = a$"; but this is incorrect in pointer arithmetic, because if "$a + b$" exceeds the bounds of the block to which a points, then "$a + b$" is undefined, and thus "$(a + b) - b = a$" too is undefined; what is true is that if the left-hand side of this equation has defined value, then it is equal to the right-hand side. This makes seemingly trivial computations on linear expressions actually tricky. One could argue that such complexity is unneeded since anyway all pointers are integers at the machine level, but this may be wrong in some contexts (pointers with capabilities) and anyway involves changing the semantics to reflect this, as in CompCertS [19].

Certain optimizations may be impossible inside a verified compiler that cannot change one well-defined value into another. For instance, the C standard [74, §6.5.8] allows "`x * y + z`" to be replaced by `fma(x, y, z)`, despite the two expressions possibly yielding different results (the former computes "$r(r(x \times y) + z)$", the latter "$r(x \times y + z)$" where r is the current floating-point rounding function). This has to be taken into account when benchmarking.

Observe how this contradicts with the tendency described in conclusion of §1.1.1: this time, it is the verified compiler that makes usage the norm.

Certain assumptions made by compilers may have unsuspected importance. For instance, GCC by default assumes, as the C standard allows it to do, that signed integer arithmetic does not overflow. This, in turn, allows it to easily promote 32-bit integers to 64 bits. CompCert does not make this assumption: its designers preferred to err on the side of caution, because many industrial embedded programs contain old code. Some programmers used to assume that the compiler would not take advantage of signed arithmetic having undefined behavior, and that integer arithmetic was thus just modular arithmetic. Our version of CompCert partially compensates this by running a static analysis for ranges, but there are cases in which it cannot perform optimizations that GCC does, because they would be incorrect in its semantic model.

Information needed for optimizations must be formally available in the semantics of the intermediate representation. Unlike an unverified compiler, we cannot assume, while at a specific optimization pass, that certain things cannot happen because they are ruled out by the way the preceding optimization passes work. (We can, however, run a procedure to check that the code fed to the pass satisfies certain properties and refuse to run the optimization if it does not.)

13.3 ONGOING AND FUTURE WORKS

Some feasible future improvements would involve extending the application of optimizations presented throughout this thesis. Specifically, Antoine Combet, an undergraduate intern at Verimag,

started to port the postpass scheduler of Chapter 4 to the RISC-V CompCert backend. At the time of writing, this port is not yet finished. Nonetheless, having such an optimization on RISC-V would be useful to reorder spills possibly introduced by prepass scheduling and LCT.

Similarly, it would surely be beneficial to port the SR backend of LCT to other architectures, such as ARMv7, as it was done by Hugo Strappazzon for AArch64. Notably, Alexandre Bérard, who realized the if-lifting pipelining optimization of §11.3.1, recently started a thesis at Verimag, aiming to improve CompCert's performance on the ARMv7 architecture. This thesis, in collaboration with Framatome company[1], is expected to lead to further developments, with potentially an industrial use of Chamois-CompCert.

Benjamin Bonneau, who generalized certain BTL constructs to allow for the validation of alias aware optimizations and store motion (see Chapter 11), also started a thesis at Verimag[2], supervised by David Monniaux. Although less connected to CompCert, this thesis aims to propose a dedicated framework able to use guaranteed properties on source programs for optimized compilation. Bonneau is especially interested in memory separation properties, provided either by language semantics, static alias analyses, deductive verification (e.g. in separation logic), or even user annotations. Typically, the Verified Software Toolchain (VST), which I mentioned in §2.5.3, includes program analysis tools capable of delivering such guaranteed properties. It offers a completely verified chain—built upon CompCert and integrated with Coq—from the formal proof of C program in separation logic to its assembly code. Like VST, the FRESCO project[3] seeks to build a Coq-integrated environment on top of CompCert for formally verified programming, but for a domain specific language tailored to efficient mathematical computations. In particular, this DSL would restrict aliasing. This would indeed not only facilitate optimized compilation, but also reduce the proof effort. The experience with Why3[4] suggests that limiting aliasing decreases the annotation effort in proofs for imperative programs.

Concerning RISC-V, a lot of work remains to be done, from exploring the integration of new finely dependent optimizations to extending the instruction set architecture coverage by implementing ISA official or community extensions. Given the limited availability of RISC-V chips in the market (in Europe as of 2023), testing these new instruction sets can be challenging. However, a possible solution would be to rely on hardware simulators such as gem5[5] to assess the positive impact of optimizations on various kinds of core pipelines.

Another interesting idea, quickly mentioned in introduction, is to use our formally verified defensive validator to prove correct the hardening of programs with countermeasures against software or hardware fault injections. In essence, this would entail using our validator to verify the insertion of redundancies (i.e. defensive monitoring) rather than their elimination. From October 2023, I am starting a research contract[6] centered on this topic. We mainly want to focus on two related objectives.

First, the automatic insertion of such countermeasures, proving (using the BTL SE engine) that they preserve program behaviors in the absence of attacks. A previous work by Torrini and Boulmé [139] proposed a CompCert backend for the IntrinSec architecture which extends RISC-V with hardware support to protect programs and their data[7]. Their backend uses this hardware support to protect programs against control-flow attacks. They conclude that their "manual" proof of semantic preservation was really painful. Alternatively, translation validation could really remove the need of such "manual" proofs.

Second, we would like to obtain security guarantees that some countermeasures protect programs against certain abstract attack models (e.g. single fault attack on a sensitive pseudo-register). In the latter approach, attacker models could be encoded as a program transformation of the hardened

[1] https://www.framatome.com/en/
[2] https://www.theses.fr/s366351
[3] https://fresco.gitlabpages.inria.fr/
[4] https://why3.lri.fr/
[5] https://www.gem5.org/
[6] In the context of the Arsene project, see https://www.pepr-cyber-arsene.fr/.
[7] This CompCert extension is publicly available at https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-intrinsec.

programs. With little generalizations, our BTL symbolic simulation test could validate a simulation between the original (unprotected) program and the attacked, hardened one.

More generally, many security properties (e.g. noninterference[8]) are actually hyperproperties of programs, similar to liveness, which the validator proposed in this document can already handle. Recently, research on relational SE has emerged as an effective technique for exposing these hyperproperties. By symbolically executing two different programs with different inputs (as we do), Farina, Chong, and Gaboardi [54] aim to generate verification conditions that feed a satisfiability modulo theories solver. Thus, instead of proving program simulation (our primary goal for validating optimizations), they prove (and also interactively refute) relational properties. Since they do not yet employ invariant synthesis, their framework requires users to annotate programs (e.g. with loop invariants). Because of the known lack of reliability of SMT solving, they sometimes experience failure in discharging verification queries (but this seems to remain rare according to their experimental evaluation [54, §8]). Essentially, given specifications in the form of Hoare triples, their paper highlights the potential of relational SE in validating various kinds of hyperproperties.

Using such relational SE on top of the Binsec binary analysis tool, Daniel, Bardin, and Rezk [44] proposed a compiler agnostic verification tool for constant time. Note that formally verified constant time compilation has been also established on a CompCert variant [12].

Our simulation test also being relational thanks to our invariants, we wonder whether only a small amount of generalizations—that we already started to investigate, as mentioned in Chapter 11 and in [65]†—would be needed to achieve similar goals.

The long-term goal would be to build a *secure compiler*, i.e. according to the PriSC workshop[9], "*secure compilation aims to protect high-level language abstractions in compiled code, even against adversarial low-level contexts, and to allow sound reasoning about security in the source language.*"

[8] Proving that secret variables do not interfere with public ones depending on inputs.

[9] Principles of Secure Compilation, see https://popl19.sigplan.org/track/prisc-2019.

BIBLIOGRAPHY

[1] *AArch64 Procedure Call Standard ABI for the ARM 64-bit Architecture*. Online. Arm, 2022. URL: https://github.com/ARM-software/abi-aa/blob/2022Q3-release/aapcs64/aapcs64.rst.

[2] Martín Abadi and Leslie Lamport. "The existence of refinement mappings." In: *Theoretical Computer Science* 82.2 (1991), pp. 253–284. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(91)90224-P. URL: https://www.sciencedirect.com/science/article/pii/030439759190224P.

[3] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996. ISBN: 978-0-521-02175-3. DOI: 10.1017/CBO9780511624162.

[4] Alexandre Bérard. "Moving side-exits despite code duplications for better Superblock scheduling in CompCert." Master Intership Report. Université Grenoble Alpes, 2022. URL: https://www-verimag.imag.fr/~boulme/CompCert_reports/Berard_Alexandre_M1report_2022.pdf.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. "Jasmin: High-Assurance and High-Speed Cryptography." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1807–1823. ISBN: 9781450349468. DOI: 10.1145/3133956.3134078. URL: https://doi.org/10.1145/3133956.3134078.

[6] Jan Andersson. "Development of a NOEL-V RISC-V SoC Targeting Space Applications." In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2020, pp. 66–67. DOI: 10.1109/DSN-W50199.2020.00020.

[7] Andrew W. Appel. "SSA is Functional Programming." In: *SIGPLAN Not.* 33.4 (Apr. 1998), pp. 17–20. ISSN: 0362-1340. DOI: 10.1145/278283.278285. URL: https://doi.org/10.1145/278283.278285.

[8] Andrew Appel and Xavier Leroy. "Efficient Extensional Binary Tries." In: *Journal of Automated Reasoning* 67 (Jan. 2023). DOI: 10.1007/s10817-022-09655-x. URL: https://hal.inria.fr/hal-03372247v2/file/extensional.pdf.

[9] *Arm A64 Instruction Set for A- profile architecture*. en. 2012. URL: https://developer.arm.com/documentation/ddi0602/2022-12?lang=en.

[10] *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*. version G.a. ARM. Jan. 2021. URL: https://developer.arm.com/documentation/ddi0487/ga/?lang=en (visited on 03/22/2021).

[11] Michael 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: *Proceedings of the First International Conference on Certified Programs and Proofs*. CPP'11. Kenting, Taiwan: Springer-Verlag, 2011, pp. 135–150. ISBN: 9783642253782. DOI: 10.1007/978-3-642-25379-9_12. URL: https://doi.org/10.1007/978-3-642-25379-9_12.

[12] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. "Formal verification of a constant-time preserving C compiler." In: *Proceedings of the ACM on Programming Languages* 4.POPL (Jan. 2020), pp. 1–30. DOI: 10.1145/3371075. URL: https://hal.univ-lorraine.fr/hal-02975012.

[13] Gilles Barthe, Delphine Demange, and David Pichardie. "A Formally Verified SSA-Based Middle-End." In: *Programming Languages and Systems*. Ed. by Helmut Seidl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 47–66. ISBN: 978-3-642-28869-2. URL: https://inria.hal.science/hal-01110783.

[14] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. "EasyCrypt: A Tutorial." In: *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*. Ed. by Alessandro Aldini, Javier Lopez, and Fabio Martinelli. Cham: Springer International Publishing, 2014, pp. 146–166. ISBN: 978-3-319-10082-1. DOI: 10.1007/978-3-319-10082-1_6. URL: https://doi.org/10.1007/978-3-319-10082-1_6.

[15] Gilles Barthe and César Kunz. "An Abstract Model of Certificate Translation." In: *ACM Trans. Program. Lang. Syst.* 33.4 (2011). ISSN: 0164-0925. DOI: 10.1145/1985342.1985344. URL: https://dl.acm.org/doi/10.1145/1985342.1985344.

[16] 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*. Toulouse, France: AAAF, SEE, Feb. 2012. URL: https://hal.inria.fr/hal-00653367 (visited on 09/19/2020).

[17] Nick Benton. "Simple Relational Correctness Proofs for Static Analyses and Program Transformations." In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: Association for Computing Machinery, 2004, pp. 14–25. ISBN: 158113729X. DOI: 10.1145/964001.964003. URL: https://doi.org/10.1145/964001.964003.

[18] 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. DOI: 10.1007/978-3-319-22102-1_5.

[19] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "CompCertS: A Memory-Aware Verified C Compiler using Pointer as Integer Semantics." In: *ITP 2017 - 8th International Conference on Interactive Theorem Proving*. Vol. 10499. ITP 2017: Interactive Theorem Proving. Brasilia, Brazil: Springer, Sept. 2017, pp. 81–97. DOI: 10.1007/978-3-319-66107-0_6. URL: https://inria.hal.science/hal-01656875.

[20] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Verified CompCert Front-End for a Memory Model Supporting Pointer Arithmetic and Uninitialised Data." In: *J. Autom. Reason.* 62.4 (2019), pp. 433–480. ISSN: 0168-7433. DOI: 10.1007/s10817-017-9439-z. URL: https://people.rennes.inria.fr/Frederic.Besson/compcert-front-end.pdf.

[21] 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*. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, pp. 460–475. DOI: 10.1007/11813040_31. URL: https://doi.org/10.1007/11813040_31.

[22] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. "Complete Removal of Redundant Expressions." In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: Association for Computing Machinery, 1998. ISBN: 0897919874. DOI: 10.1145/277650.277653. URL: https://doi.org/10.1145/277650.277653.

[23] Sylvain Boulmé. "Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)." See also http://www-verimag.imag.fr/~boulme/hdr.html. Habilitation à diriger des recherches. Université Grenoble-Alpes, Sept. 2021. URL: https://hal.science/tel-03356701.

[24] Sylvain Boulmé and Thomas Vandendorpe. "Embedding Untrusted Imperative ML Oracles into Coq Verified Code." This preprint has been largely rewritten and integrated into Sylvain Boulmé's Habilitation in 2021, See http://www-verimag.imag.fr/~boulme/hdr.html. July 2019. URL: https://hal.science/hal-02062288.

[25] Timothy Bourke, Lélio Brun, and Marc Pouzet. "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset." In: *Proc. ACM Program. Lang.* 4.POPL (2019). DOI: 10.1145/3371112. URL: https://doi.org/10.1145/3371112.

[26] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. "Implementing and Reasoning About Hash-consed Data Structures in Coq." In: *J. Autom. Reasoning* 53.3 (2014), pp. 271–304. DOI: 10.1007/s10817-014-9306-0.

[27] Lélio Brun. "Mechanized semantics and verified compilation for a dataflow synchronous language with reset." Theses. Université Paris sciences et lettres, July 2020. URL: https://theses.hal.science/tel-03068862.

[28] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[29] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. "LUSTRE: A Declarative Language for Real-Time Programming." In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, pp. 178–188. ISBN: 0897912152. DOI: 10.1145/41625.41641. URL: https://doi.org/10.1145/41625.41641.

[30] Manuel M.T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. "A Functional Perspective on SSA Optimisation Algorithms." In: *Electronic Notes in Theoretical Computer Science* 82.2 (Apr. 2024), pp. 347–361. ISSN: 15710661. DOI: 10.1016/S1571-0661(05)82596-4. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571066105825964 (visited on 03/08/2021).

[31] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. "Semantic Program Alignment for Equivalence Checking." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1027–1040. ISBN: 9781450367127. DOI: 10.1145/3314221.3314596. URL: https://doi.org/10.1145/3314221.3314596.

[32] Basile Clément. "Translation Validation of Tensor Compilers." Theses. École Normale Supérieure (Paris), Sept. 2022. URL: https://theses.hal.science/tel-03903895.

[33] Basile Clément and Albert Cohen. "End-to-End Translation Validation for the Halide Language." In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022). DOI: 10.1145/3527328. URL: https://doi.org/10.1145/3527328.

[34] John Cocke and Ken Kennedy. "An algorithm for reduction of operator strength." en. In: *Communications of the ACM* 20.11 (Nov. 1977), pp. 850–856. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359863.359888. URL: https://dl.acm.org/doi/10.1145/359863.359888 (visited on 08/25/2021).

[35] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. "Using Symbolic Execution for Verifying Safety-Critical Systems." In: *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-9. Vienna, Austria: Association for Computing Machinery, 2001, pp. 142–151. ISBN: 1581133901. DOI: 10.1145/503209.503230. URL: https://doi.org/10.1145/503209.503230.

[36] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. "Operator strength reduction." In: *ACM Trans. Program. Lang. Syst.* 23 (2001), pp. 603–625.

[37] Thierry Coquand and Gérard Huet. "Constructions: A higher order proof system for mechanizing mathematics." en. In: *EUROCAL '85*. Ed. by G. Goos et al. Vol. 203. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1985, pp. 151–184. ISBN: 978-3-540-15983-4 978-3-540-39684-0. DOI: 10.1007/3-540-15983-5_13. URL: http://link.springer.com/10.1007/3-540-15983-5_13 (visited on 01/23/2023).

[38] Thierry Coquand and Gérard Huet. "The calculus of constructions." In: *Information and Computation* 76.2 (1988), pp. 95–120. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(88)90005-3. URL: https://www.sciencedirect.com/science/article/pii/0890540188900053.

[39] Nathanaël Courant and Xavier Leroy. "Verified code generation for the polyhedral model." In: *Proc. ACM Program. Lang.* 5.POPL (2021), 40:1–40:24. DOI: 10.1145/3434321.

[40] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTREÉ Analyzer." en. In: *Programming Languages and Systems*. Ed. by David Hutchison et al. Vol. 3444. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 21–30. ISBN: 978-3-540-25435-5 978-3-540-31987-0. DOI: 10.1007/978-3-540-31987-0_3. URL: http://link.springer.com/10.1007/978-3-540-31987-0_3 (visited on 01/11/2023).

[41] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. "Predictability Considerations in the Design of Multi-Core Embedded Systems." en. In: *Proceedings of Embedded Real Time Software and Systems*. 2010. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=94c507ff47f765a932a7208cd8558733d327d660.

[42] Charlie Curtsinger and Emery D Berger. "STABILIZER: Statistically Sound Performance Evaluation." In: *ASPLOS'2013*. ACM, 2013, pp. 219–228. DOI: 10.1145/2451116.2451141.

[43] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[44] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. "Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level." In: *2020 IEEE Symposium on Security and Privacy (SP)* (2019), pp. 1021–1038. URL: https://leslyann-daniel.fr/ressources/papers/2020_SP_binsecrel.pdf.

[45] Delphine Demange. "Semantic Foundations of Intermediate Program Representations." EAPLS Best PhD Dissertation Award 2012. Gilles Kahn PhD Thesis Award 2013. PhD thesis. École Normale Supérieure de Cachan, France, Oct. 2012. URL: http://people.irisa.fr/Delphine.Demange/papers/DemangePhD.pdf.

[46] Delphine Demange, David Pichardie, and Léo Stefanesco. "Verifying Fast and Sparse SSA-based Optimizations in Coq." In: *24th International Conference on Compiler Construction, CC 2015*. London, United Kingdom, 2015. DOI: 10.1007/978-3-662-46663-6_12.

[47] Delphine Demange and Yon Fernandez de Retana. "Mechanizing conventional SSA for a verified destruction with coalescing." In: *25th International Conference on Compiler Construction*. Barcelona, Spain, Mar. 2016. DOI: 10.1145/2892208.2892222. URL: https://hal.archives-ouvertes.fr/hal-01378393.

[48] David Detlefs, Greg Nelson, and James B. Saxe. "Simplify: A Theorem Prover for Program Checking." In: *J. ACM* 52.3 (May 2005), pp. 365–473. ISSN: 0004-5411. DOI: 10.1145/1066100.1066102. URL: https://doi.org/10.1145/1066100.1066102.

[49] D. M. Dhamdhere. "Practical Adaption of the Global Optimization Algorithm of Morel and Renvoise." In: *ACM Trans. Program. Lang. Syst.* 13.2 (1991), pp. 291–294. ISSN: 0164-0925. DOI: 10.1145/103135.214520. URL: https://doi.org/10.1145/103135.214520.

[50] D. M. Dhamdhere and Harish Patil. "An Elimination Algorithm for Bidirectional Data Flow Problems Using Edge Placement." In: *ACM Trans. Program. Lang. Syst.* 15.2 (1993), pp. 312–336. ISSN: 0164-0925. DOI: 10.1145/169701.169684. URL: https://doi.org/10.1145/169701.169684.

[51] Dhananjay M. Dhamdhere and Uday P. Khedker. "Complexity of Bi-Directional Data Flow Analysis." In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. Charleston, South Carolina, USA: Association for Computing Machinery, 1993, pp. 397–408. ISBN: 0897915607. DOI: 10.1145/158511.158696. URL: https://doi.org/10.1145/158511.158696.

[52] Stefano Di Mascio, Alessandra Menicucci, Eberhard Gill, Gianluca Furano, and Claudio Monteleone. "Leveraging the Openness and Modularity of RISC-V in Space." In: *Journal of Aerospace Information Systems* 16 (2019), pp. 1–19. DOI: 10.2514/1.I010735.

[53] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research." In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Ed. by Martin Schoeberl. Vol. 55. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, 2:1–2:10.

[54] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. "Relational Symbolic Execution." In: *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. PPDP '19. Porto, Portugal: Association for Computing Machinery, 2019. ISBN: 9781450372497. DOI: 10.1145/3354166.3354175. URL: https://doi.org/10.1145/3354166.3354175.

[55] Paul Feautrier. "Dataflow Analysis of Array and Scalar References." In: *International Journal of Parallel Programming* 20 (Aug. 1996). DOI: 10.1007/BF01407931.

[56] Jean-Christophe Filliatre and Sylvain Conchon. "Type-Safe Modular Hash-Consing." en. In: ACM Press, 2006. DOI: 10.1145/1159876.1159880.

[57] J. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction." In: *IEEE Transactions on Computers* 30.07 (1981), pp. 478–490. ISSN: 0018-9340. DOI: 10.1109/TC.1981.1675827.

[58] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Electronics & Electrical. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 9780080477541. URL: https://books.google.fr/books?id=R5UXl6Jo0XYC.

[59] Alexis Fouilhe and Sylvain Boulmé. "[FouilheB14] A Certifying Frontend for (Sub)polyhedral Abstract Domains." In: *Verified Software: Theories, Tools and Experiments*. Ed. by Dimitra Giannakopoulou and Daniel Kroening. Vol. 8471. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 200–215. ISBN: 978-3-319-12153-6 978-3-319-12154-3. DOI: 10.1007/978-3-319-12154-3_13. URL: http://link.springer.com/10.1007/978-3-319-12154-3_13 (visited on 05/25/2020).

[60] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. "Towards Formally Verified Optimizing Compilation in Flight Control Software." In: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, March 18, 2011, Grenoble, France*. Ed. by Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm. Vol. 18. OASICS. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011, pp. 59–68. DOI: 10.4230/OASIcs.PPES.2011.59.

[61] Sabine Glesner. "Using Program Checking to Ensure the Correctness of Compiler Implementations." In: *JUCS - Journal of Universal Computer Science* 9.3 (2003), pp. 191–222. ISSN: 0948-695X. DOI: 10.3217/jucs-009-03-0191. URL: https://doi.org/10.3217/jucs-009-03-0191.

[62] Jonathan S. Golan. "Semimodules over Semirings." en. In: *Semirings and their Applications*. Dordrecht: Springer Netherlands, 1999, pp. 149–161. ISBN: 978-90-481-5252-0 978-94-015-9333-5. DOI: 10.1007/978-94-015-9333-5_14. URL: http://link.springer.com/10.1007/978-94-015-9333-5_14 (visited on 11/09/2022).

[63] Léo Gourdin. "Formally verified postpass scheduling with peephole optimization for AArch64." In: *20èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL 2021*. June 2021. URL: https://www.lirmm.fr/afadl2021/papers/afadl2021_paper_9.pdf.

[64] Léo Gourdin. "Lazy Code Transformations in a Formally Verified Compiler." In: *Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems*. ICOOOLPS 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 3–14. ISBN: 9798400702488. DOI: 10.1145/3605158.3605848. URL: https://hal.science/hal-04108775.

[65] Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. "Formally Verifying Optimizations with Block Simulations." In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023). DOI: https://doi.org/10.1145/3622799. URL: https://hal.science/hal-04102940.

[66] Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. "Artifact of the "Formally Verifying Optimizations with Block Simulations" OOPSLA'23 paper." In: *Proceedings of the ACM on Programming Languages* (Sept. 2023). DOI: 10.5281/zenodo.8314677. URL: https://doi.org/10.5281/zenodo.8314677.

[67] Léo Gourdin and Sylvain Boulmé. *Certifying assembly optimizations in Coq by symbolic execution with hash-consing*. en. Online. Coq Workshop, June 2021. URL: https://coq-workshop.gitlab.io/2021/abstracts/Coq2021-04-01-certifying-optimizations-hash-consing.pdf.

[68] Alban Gruin, Thomas Carle, Christine Rochange, Hugues Cassé, and Pascal Sainrat. "MINOTAuR: a Timing Predictable RISC-V Core Featuring Speculative Execution." In: *IEEE Transactions on Computers* 72.1 (2023), pp. 183–195. DOI: 10.1109/TC.2022.3200000. URL: https://ut3-toulouseinp.hal.science/hal-03773263.

[69] Alban Gruin, Thomas Carle, Christine Rochange, and Pascal Sainrat. "Enabling timing predictability in the presence of store buffers." In: *31st International Conference on Real-Time Networks and Systems (RTNS 2023)*. Dortmund, Germany: ACM, June 2023, pp. 1–10. DOI: 10.1145/3575757.3593653. URL: https://hal.science/hal-04082519.

[70] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. "MiBench: A free, commercially representative embedded benchmark suite." en. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. Austin, TX, USA: IEEE, 2001, pp. 3–14. ISBN: 978-0-7803-7315-0. DOI: 10.1109/WWC.2001.990739. URL: http://ieeexplore.ieee.org/document/990739/ (visited on 04/19/2021).

[71] Sebastian Hahn. "On Static Execution-Time Analysis—Compositionality, Pipeline Abstraction, and Predictable Hardware." PhD thesis. Universität des Saarlandes, 2019. URL: https://d-nb.info/1187241180/34.

[72] Wen-mei Hwu et al. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation." In: *The Journal of Supercomputing* 7 (May 1993), pp. 229–248. DOI: 10.1007/BF01205185.

[73] ISO. *C11 Standard*. ISO/IEC 9899:2011. 2011. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[74] *International standard—Programming languages—C*. Tech. rep. ISO/IEC, 9899:1999.

[75] Justus Fasse. "Code Transformations to Increase Prepass Scheduling Opportunities in CompCert." Master Thesis of Science. Université Grenoble Alpes, Aug. 2021. URL: https://www-verimag.imag.fr/~boulme/CPP_2022/FASSE-Justus-MSc-Thesis_2021.pdf.

[76] A. Kanade, A. Sanyal, and U. Khedker. "A PVS Based Framework for Validating Compiler Optimizations." In: *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*. 2006, pp. 108–117. DOI: 10.1109/SEFM.2006.4.

[77] Jeehoon Kang et al. "Crellvm: Verified Credible Compilation for LLVM." In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 631–645. ISBN: 9781450356985. DOI: 10.1145/3192366.3192377. URL: https://sf.snu.ac.kr/publications/crellvm.pdf.

[78] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. "Language-Parametric Compiler Validation with Application to LLVM." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 1004–1019. ISBN: 9781450383172. DOI: 10.1145/3445814.3446751. URL: https://doi.org/10.1145/3445814.3446751.

[79] Richard A. Kelsey. "A Correspondence between Continuation Passing Style and Static Single Assignment Form." In: *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. IR '95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 13–22. ISBN: 0897917545. DOI: 10.1145/202529.202532. URL: https://doi.org/10.1145/202529.202532.

[80] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. "Generalized Symbolic Execution for Model Checking and Testing." In: *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'03. Warsaw, Poland: Springer-Verlag, 2003, pp. 553–568. ISBN: 3540008985.

[81] James C. King. "Symbolic Execution and Program Testing." In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[82] Claude Kirchner and Hélène Kirchner. "Equational logic and rewriting." In: *Handbook of the History of Logic*. Ed. by Dov M. Gabbay, Jörg H. Siekmann, and John Woods. Vol. 9. History of Logic and Computation in the 20th Century Chap.8. Elsevier, Mar. 2014. URL: https://hal.inria.fr/hal-01183817.

[83] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. "Optimal Code Motion: Theory and Practice." In: *ACM Transactions on Programming Languages and Systems* 16 (Sept. 1995). DOI: 10.1145/183432.183443.

[84] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. "Lazy code motion." en. In: *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation - PLDI '92*. Most Influential PLDI Paper 2002 Award, https://www.sigplan.org/Awards/PLDI/. San Francisco, California, United States: ACM Press, 1992, pp. 224–234. ISBN: 978-0-89791-475-8. DOI: 10.1145/143095.143136. URL: http://portal.acm.org/citation.cfm?doid=143095.143136 (visited on 09/21/2021).

[85] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. "Lazy Strength Reduction." In: *Journal of Programming Languages* 1 (1993), pp. 71–91.

[86] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 179–191. ISBN: 9781450325448. DOI: 10.1145/2535838.2535841. URL: https://doi.org/10.1145/2535838.2535841.

[87] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. "Proving Optimizations Correct Using Parameterized Program Equivalence." In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 327–337. ISBN: 9781605583921. DOI: 10.1145/1542476.1542513. URL: https://cseweb.ucsd.edu/~lerner/papers/pldi09-pec.pdf.

[88] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. "CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler." In: *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*. 3AF, SEE, SIE. Toulouse, France, Jan. 2018, pp. 1–9. URL: https://hal.inria.fr/hal-01643290.

[89] M. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines." In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pp. 318–328. ISBN: 0897912691. DOI: 10.1145/53990.54022.

[90] M. Lee, P. Tirumalai, and T. Ngai. "Software pipelining and superblock scheduling: compilation techniques for VLIW machines." In: [*1993*] *Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences*. Vol. i. 1993, 202–213 vol.1. DOI: 10.1109/HICSS.1993.270744.

[91] Xavier Leroy. Answer to CompCert bug #137. URL: https://github.com/AbsInt/CompCert/issues/137#issuecomment-243353529.

[92] Xavier Leroy. "A formally verified compiler back-end." In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4. URL: http://xavierleroy.org/publi/compcert-backend.pdf.

[93] Xavier Leroy. "Formal verification of a realistic compiler." In: *Communications of the ACM* 52.7 (2009). DOI: 10.1145/1538788.1538814.

[94] Xavier Leroy. "Verified squared: does critical software deserve verified tools?" In: *POPL'11*. Austin, TX, USA: ACM, Jan. 2011, pp. 1–2. DOI: 10.1145/1926385.1926387. URL: https://xavierleroy.org/publi/popl11-invited-talk.pdf.

[95] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA, June 2012, p. 26.

[96] Xavier Leroy and Sandrine Blazy. "Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations." In: *J. Autom. Reason.* 41.1 (2008), pp. 1–31. DOI: 10.1007/s10817-008-9099-0.

[97] Pierre Letouzey. "Extraction in Coq: An Overview." en. In: *Logic and Theory of Algorithms*. Ed. by Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe. Vol. 5028. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 359–369. ISBN: 978-3-540-69405-2 978-3-540-69407-6. DOI: 10.1007/978-3-540-69407-6_39. URL: http://link.springer.com/10.1007/978-3-540-69407-6_39 (visited on 01/23/2023).

[98] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. "Random Testing for C and C++ Compilers with YARPGen." In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020). DOI: 10.1145/3428264. URL: https://doi.org/10.1145/3428264.

[99] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. "Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores." In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: Association for Computing Machinery, 1998, pp. 26–37. ISBN: 0897919874. DOI: 10.1145/277650.277659. URL: https://doi.org/10.1145/277650.277659.

[100] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. "Alive2: Bounded Translation Validation for LLVM." In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 65–79. ISBN: 9781450383912. DOI: 10.1145/3453483.3454030. URL: https://doi.org/10.1145/3453483.3454030.

[101] P Geoffrey Lowney, Stefan M Freudenberger, Thomas J Karzes, W D Lichtenstein, Robert P Nix, John S O'Donnell, and John C Ruttenberg. "The Multiflow Trace Scheduling Compiler." en. In: (1992), p. 83.

[102] Tao Lu. *A Survey on RISC-V Security: Hardware and Architecture*. en. arXiv:2107.04175 [cs]. July 2021. URL: http://arxiv.org/abs/2107.04175 (visited on 09/26/2022).

[103] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. "A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems." In: *ACM Comput. Surv.* 52.3 (June 2019). ISSN: 0360-0300. DOI: 10.1145/3323212. URL: https://www-users.york.ac.uk/~rd17/papers/ACMCSUR2019SurveyTimingVerification.pdf.

[104] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. "Compiler Fuzzing: How Much Does It Matter?" In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019). DOI: 10.1145/3360581. URL: https://doi.org/10.1145/3360581.

[105] W. M. McKeeman. "Peephole optimization." en. In: *Communications of the ACM* 8.7 (July 1965), pp. 443–444. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/364995.365000. URL: https://dl.acm.org/doi/10.1145/364995.365000 (visited on 12/09/2022).

[106] David Monniaux and Sylvain Boulmé. "The Trusted Computing Base of the CompCert Verified Compiler." In: *Programming Languages and Systems (ESOP 2022)*. Ed. by Ilya Sergey. Vol. 13240. Munich, Germany: Springer, Apr. 2022, pp. 204–233. DOI: 10.1007/978-3-030-99336-8_8. URL: https://hal.archives-ouvertes.fr/hal-03541595.

[107] David Monniaux, Sylvain Boulmé, and Léo Gourdin. *Formally Verified Advanced Optimizations for RISC-V*. Barcelona, Spain, 2023.

[108] David Monniaux, Léo Gourdin, Sylvain Boulmé, and Olivier Lebeltel. "Testing a Formally Verified Compiler." In: *Tests and Proofs*. Ed. by Virgile Prevosto and Cristina Seceleanu. Cham: Springer Nature Switzerland, 2023, pp. 40–48. ISBN: 978-3-031-38828-6. URL: https://hal.science/hal-04096390/.

[109] David Monniaux and Cyril Six. "Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion." In: *LCTES '21: 22nd ACM SIGPLAN / SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, Virtual Event, Canada, 22 June, 2021*. Ed. by Jörg Henkel and Xu Liu. ACM, 2021, pp. 85–96. DOI: 10.1145/3461648.3463850.

[110] David Monniaux and Cyril Six. "Formally Verified Loop-Invariant Code Motion and Assorted Optimizations." In: *ACM Trans. Embed. Comput. Syst.* (Mar. 2022). ISSN: 1539-9087. DOI: 10.1145/3529507. URL: https://amu.hal.science/IMAG/hal-03628646v1.

[111] E. Morel and C. Renvoise. "Global optimization by suppression of partial redundancies." en. In: *Communications of the ACM* 22.2 (Feb. 1979), pp. 96–103. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359060.359069. URL: https://dl.acm.org/doi/10.1145/359060.359069 (visited on 08/25/2021).

[112] Eric Mullen, Zachary Tatlock, Daryl Zuniga, and Dan Grossman. "Verified Peephole Optimizations for CompCert." en. In: (June 2016), p. 15.

[113] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. "Producing Wrong Data Without Doing Anything Obviously Wrong!" en. In: *ASPLOS'2009*. ACM, 2009, pp. 265–276. DOI: 10.1145/1508244.1508275.

[114] Kedar S. Namjoshi and Lenore D. Zuck. "Witnessing Program Transformations." In: *Static Analysis*. Ed. by Francesco Logozzo and Manuel Fähndrich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 304–323. ISBN: 978-3-642-38856-9. URL: https://kedar-namjoshi.github.io/papers/Namjoshi-Zuck-SAS-2013.pdf.

[115] George C. Necula. "Proof-Carrying Code." In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: Association for Computing Machinery, 1997, pp. 106–119. ISBN: 0897918533. DOI: 10.1145/263699.263712. URL: https://doi.org/10.1145/263699.263712.

[116] George C. Necula. "Translation Validation for an Optimizing Compiler." In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 83–94. ISBN: 1581131992. DOI: 10.1145/349299.349314.

[117] Nicolas Nardino. "Register-Pressure-Aware Prepass-Scheduling for CompCert." Bachelor Thesis of Science. ENS de Lyon, Aug. 2021. URL: https://www-verimag.imag.fr/~boulme/CPP_2022/NARDINO-Nicolas-BSc-Thesis_2021.pdf.

[118] Christine Paulin-Mohring. "Introduction to the Calculus of Inductive Constructions." In: *All about Proofs, Proofs for All*. Ed. by Bruno Woltzenlogel Paleo and David Delahaye. Vol. 55. Studies in Logic (Mathematical logic and foundations). College Publications, Jan. 2015, p. 14. URL: https://inria.hal.science/hal-01094195.

[119] A. Pnueli, M. Siegel, and E. Singerman. "Translation validation." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernhard Steffen. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1384. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 151–166. ISBN: 978-3-540-64356-2 978-3-540-69753-4. DOI: 10.1007/BFb0054170. URL: http://link.springer.com/10.1007/BFb0054170 (visited on 08/18/2021).

[120] Louis-Noël Pouchet. *the Polyhedral Benchmark suite*. 2012. URL: http://web.cs.ucla.edu/~pouchet/software/polybench/ (visited on 05/12/2020).

[121] B. Ramakrishna Rau, Christopher D. Glaeser, and Raymond L. Picard. "Efficient code generation for horizontal architectures: Compiler techniques and architectural support." en. In: *ACM SIGARCH Computer Architecture News* 10.3 (Apr. 1982), pp. 131–139. ISSN: 0163-5964. DOI: 10.1145/1067649.801721. URL: https://dl.acm.org/doi/10.1145/1067649.801721 (visited on 12/07/2022).

[122] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. "Test-case reduction for C compiler bugs." 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. 335–346. DOI: 10.1145/2254064.2254104.

[123] Silvain Rideau and Xavier Leroy. "Validating register allocation and spilling." In: *Compiler Construction (CC 2010)*. Vol. 6011. Springer, 2010, pp. 224–243. URL: http://gallium.inria.fr/~xleroy/publi/validation-regalloc.pdf.

[124] Martin C. Rinard and Darko Marino. "Credible Compilation with Pointers." In: *Proceedings of the FLoC Workshop on Run-Time Result Verification*. 1999. URL: https://people.csail.mit.edu/rinard/paper/credibleCompilation.html.

[125] Xavier Rival. "Symbolic Transfer Function-Based Approaches to Certified Compilation." In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: Association for Computing Machinery, 2004, pp. 1–13. ISBN: 158113729X. DOI: 10.1145/964001.964002. URL: https://doi.org/10.1145/964001.964002.

[126] Xavier Rival and Laurent Mauborgne. "The trace partitioning abstract domain." en. In: *ACM Transactions on Programming Languages and Systems* 29.5 (Aug. 2007), p. 26. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/1275497.1275501. URL: https://dl.acm.org/doi/10.1145/1275497.1275501 (visited on 12/21/2022).

[127] 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. DOI: 10.1007/978-3-642-35308-6_5. URL: https://doi.org/10.1007/978-3-642-35308-6_5.

[128] Oliver Rüthing. "Code motion in the presence of critical edges without bidirectional data flow analysis." In: *Science of Computer Programming* 39.1 (2001). Static Program Analysis (SAS'98), pp. 3–29. ISSN: 0167-6423. DOI: https://doi.org/10.1016/S0167-6423(00)00016-2. URL: https://www.sciencedirect.com/science/article/pii/S0167642300000162.

[129] Hanan Samet. "Automatically proving the correctness of translations involving optimized code - research sponsored by Advanced Research Projects Agency, ARPA order no. 2494." PhD thesis. Stanford University, 1975. URL: http://www.cs.umd.edu/~hjs/pubs/compilers/CS-TR-75-498.pdf.

[130] Hanan Samet. "Compiler testing via symbolic interpretation." In: *Proceedings of the 1976 Annual Conference, Houston, Texas, USA, October 20-22, 1976*. Ed. by John A. Gosden and Olin G. Johnson. ACM, 1976, pp. 492–497. DOI: 10.1145/800191.805648.

[131] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "Translation validation for a verified OS kernel." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 471–482. DOI: 10.1145/2491956.2462183.

[132] Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. "Preallocation Instruction Scheduling with Register Pressure Minimization Using a Combinatorial Optimization Approach." In: *ACM Trans. Archit. Code Optim.* 10.3 (Sept. 2013). ISSN: 1544-3566. DOI: 10.1145/2512432. URL: https://doi-org.ins2i.bib.cnrs.fr/10.1145/2512432.

[133] Cyril Six. "Optimized and formally-verified compilation for a VLIW processor." PhD thesis. Université Grenoble Alpes, July 2021. URL: https://hal.archives-ouvertes.fr/tel-03326923.

[134] Cyril Six, Sylvain Boulmé, and David Monniaux. "Certified and efficient instruction scheduling: application to interlocked VLIW processors." In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 129:1–129:29. URL: https://hal.archives-ouvertes.fr/hal-02185883.

[135] Cyril Six, Léo Gourdin, Sylvain Boulmé, David Monniaux, Justus Fasse, and Nicolas Nardino. "Formally Verified Superblock Scheduling." In: *Certified Programs and Proofs (CPP '22)*. Philadelphia, United States, Jan. 2022. DOI: 10.1145/3497775.3503679. URL: https://hal.archives-ouvertes.fr/hal-03200774.

[136] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. "Toward understanding compiler bugs in GCC and LLVM." en. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. Saarbrücken Germany: ACM, July 2016, pp. 294–305. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931074. URL: https://dl.acm.org/doi/10.1145/2931037.2931074 (visited on 06/17/2022).

[137] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality Saturation: A New Approach to Optimization." In: *Log. Methods Comput. Sci.* 7.1 (2011). DOI: 10.2168/LMCS-7(1:10)2011. URL: https://doi.org/10.2168/LMCS-7(1:10)2011.

[138] Zachary Tatlock and Sorin Lerner. "Bringing Extensibility to Verified Compilers." In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, Ontario, Canada: Association for Computing Machinery, 2010, pp. 111–121. ISBN: 9781450300193. DOI: 10.1145/1806596.1806611. URL: https://doi.org/10.1145/1806596.1806611.

[139] Paolo Torrini and Sylvain Boulmé. "A CompCert Backend with Symbolic Encryption." In: *Sixth workshop on Principles of Secure Compilation (PriSC'22), part of the 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2022)*. Philadelphia, Pennsylvania, United States, Jan. 2022. URL: https://hal.science/hal-03555551.

[140] Jean-Baptiste Tristan. "Formal verification of translation validators." PhD thesis. Université Paris 7 Diderot, Nov. 2009.

[141] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. "Evaluating Value-Graph Translation Validation for LLVM." In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 295–305. ISBN: 9781450306638. DOI: 10.1145/1993498.1993533. URL: https://doi.org/10.1145/1993498.1993533.

[142] Jean-Baptiste Tristan and Xavier Leroy. "Formal verification of translation validators: A case study on instruction scheduling optimizations." In: *35th ACM Symposium on Principles of Programming Languages (POPL 2008)*. ACM. San Francisco, United States: ACM Press, Jan. 2008, pp. 17–27. DOI: 10.1145/1328438.1328444. URL: https://hal.inria.fr/inria-00289540.

[143] Jean-Baptiste Tristan and Xavier Leroy. "Verified Validation of Lazy Code Motion." In: 2009, pp. 316–326. URL: http://gallium.inria.fr/~xleroy/publi/validation-LCM.pdf.

[144] Jean-Baptiste Tristan and Xavier Leroy. "A simple, verified validator for software pipelining." In: 2010, pp. 83–92. URL: http://gallium.inria.fr/~xleroy/publi/validation-softpipe.pdf.

[145] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. "The RISC-V instruction set manual." In: *Volume I: User-Level ISA', version* 2 (2014).

[146] Thom Wiggers. "Energy-Efficient ARM64 Cluster with Cryptanalytic Applications: 80 Cores That Do Not Cost You an ARM and a Leg." In: July 2019, pp. 175–188. ISBN: 978-3-030-25282-3. DOI: 10.1007/978-3-030-25283-0_10.

[147] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers." In: 2011, pp. 283–294. DOI: 10.1145/1993498.1993532.

[148] Anna Zaks and Amir Pnueli. "CoVaC: Compiler Validation by Program Analysis of the Cross-Product." In: *FM 2008: Formal Methods*. Ed. by Jorge Cuellar, Tom Maibaum, and Kaisa Sere. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 35–51. ISBN: 978-3-540-68237-0.

[149] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. "Formalizing the LLVM Intermediate Representation for Verified Program Transformations." In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: Association for Computing Machinery, 2012, pp. 427–440. ISBN: 9781450310833. DOI: 10.1145/2103656.2103709. URL: https://doi.org/10.1145/2103656.2103709.

[150] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. "An empirical study of optimization bugs in GCC and LLVM." en. In: *Journal of Systems and Software* 174 (Apr. 2021), p. 110884. ISSN: 01641212. DOI: 10.1016/j.jss.2020.110884. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121220302740 (visited on 06/17/2022).

[151] L Zuck, A Pnueli, and R Leviathan. *Validation of Optimizing Compilers*. Tech. rep. Computer Science Department, NYU, 2001.