COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

école normale supérieure rennes

THÈSE DE DOCTORAT DE

L'ÉCOLE NORMALE SUPÉRIEURE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Aurèle BARRIÈRE

Formal Verification of Just-in-Time Compilation

Thèse présentée et soutenue à l'IRISA, le 19 Décembre 2022
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Andrew APPEL Professor – Princeton University
Xavier LEROY Professeur – Collège de France

Composition du Jury :

Président :		
Examinateurs :	Andrew APPEL	Professor – Princeton University
	Xavier LEROY	Professeur – Collège de France
	Stephan MERZ	Directeur de Recherche – INRIA
	Magnus MYREEN	Associate Professor – Chalmers University of Technology
	Alan SCHMITT	Directeur de Recherche – INRIA
	Manuel SERRANO	Directeur de Recherche – INRIA
Dir. de thèse :	Sandrine BLAZY	Professeur des Universités – Université de Rennes 1
Co-encadrant de thèse :	David PICHARDIE	Chercheur – Meta

Résumé en Français

This dissertation begins with a summary in French. The rest of this document is written in English, and starts at page 8.

La compilation à la volée (ou compilation *just-in-time*) est une technique pour exécuter des programmes à la popularité grandissante. Les compilateurs à la volée (ou JITs) existent depuis les années 1960 [Aycock 2003], mais leur usage a été particulièrement démocratisé pour les langages *dynamiques* comme Python [PyPy 2022], JavaScript [V8 2022], Julia [Julia 2022], R [R 2022], Lua [LuaJIT 2022] ou Matlab [MathWorks 2022]. Aujourd'hui, des JITs sont aussi utilisés pour des langages statiques comme Java [HotSpot 2022] ou encore le langage eBPF dans le noyau Linux [eBPF 2022]. Les JITs offrent une performance remarquable pour les langages dynamiques, et pour cette raison la plupart des navigateurs web modernes utilisent des JITs pour exécuter les programmes trouvés sur les pages web visitées. Firefox utilise SpiderMonkey [Firefox 2022b], Google Chrome et Chromium utilisent V8 [V8 2022], et le moteur WebKit de Safari utilise JavaScriptCore [WebKit 2022]. Tous ces exemples utilisent de la compilation à la volée pour exécuter des programmes JavaScript.

Ces navigateurs web cumulent des milliards d'utilisateurs. Mais à cause de leur complexité, leurs JITs sont exposés à de nombreux bugs et vulnérabilités. De telles vulnérabilités sont trop nombreuses pour être listées exhaustivement ici [Firefox 2022a]. Si un attaquant connaît une faille dans un JIT, il peut créer et distribuer un programme JavaScript qui l'exploite. Par exemple, l'équipe Project Zero a trouvé des programmes exploitant des bugs dans JavaScriptCore pour faire exécuter du code malveillant par Safari [Project Zero 2019; 2020]. Cette équipe a également trouvé des exemples de programmes exploitant des vulnérabilités de Google Chrome avant qu'elles n'aient pu être corrigées [Project Zero 2021a]. Ces attaques peuvent avoir de graves conséquences, comme l'attaque visant l'entreprise CoinBase [Coinbase 2019] qui exploitait un bug de SpiderMonkey pour récupérer des données confidentielles. Pourtant, les navigateurs web sont incontournables aujourd'hui, y compris pour des utilisations sensibles qui ne peuvent tolérer de bugs.

Les méthodes formelles pour garantir l'absence de bugs Les navigateurs web et leurs JITs ne sont pas les seuls exemples de programmes sensibles, et de nombreuses techniques ont été conçues pour trouver ou empêcher des erreurs logicielles aux conséquences graves. La méthode la plus intuitive consiste à tester un programme sur un ensemble d'entrées généré ou choisi aléatoirement. Le test peut trouver de nombreuses erreurs, mais ne peut pas garantir l'absence de bugs sans tester toutes les entrées possibles du programme, ce qui s'avère souvent impossible.

Pour des garanties plus fortes, les *méthodes formelles* visent à prouver des propriétés de programmes. Avec des abstractions ou des modèles appropriés, il est possible de raisonner mathématiquement sur un programme sans l'exécuter. Par exemple, l'analyseur statique Astrée a été utilisé pour garantir l'absence d'une catégorie de bugs dans le logiciel de commande de vol électrique primaire de deux types d'avion Airbus [Delmas and Souyris 2007]. Les méthodes formelles incluent de nombreuses techniques, comme l'analyse statique, le model-checking ou la vérification déductive de programme.

La vérification déductive consiste à représenter des définitions et des raisonnements mathématiques dans un *assistant de preuve*. Les assistants de preuve sont des programmes pour énoncer et vérifier des preuves avec une rigueur mathématique. Ils sont soit automatiques comme Why3 [Filliâtre 2012] et F* [Swamy et al. 2013], soit interactifs comme Coq [Coq 2022], Isabelle/HOL [Nipkow et al. 2002] et ACL2 [Kaufmann et al. 2000]. Contrairement aux preuves faites à la main, il est impossible d'oublier un cas dans une preuve écrite avec un assistant. Si on fait confiance à un assistant de preuve, l'utiliser pour écrire et vérifier une preuve apporte alors une grande assurance et élimine le besoin de la vérifier à la main.

Pour raisonner mathématiquement sur des exécutions de programmes en utilisant des méthodes formelles comme la vérification déductive de programme, il faut définir des *sémantiques formelles*, une description rigoureuse des comportements d'un programme. Des sémantiques formelles ont été définies pour plusieurs langages de programmation, comme C [Norrish 1998, Blazy and Leroy 2009, Krebbers and Wiedijk 2011], JavaScript [Bodin et al. 2014] ou WebAssembly [Watt et al. 2021]. Pour prouver des propriétés de programme, on peut utiliser la sémantique formelle de ce programme et des *logiques de programmes*. Par exemple, une primitive cryptographique écrite en C a été prouvée correcte en Coq à l'aide de la sémantique formelle de C et une logique de programme appelée *logique de séparation* [Appel 2015].

Compilation formellement vérifiée Pour faire confiance à un programme qui s'exécute sur un ordinateur, il faut non seulement faire confiance au code source de ce programme, mais

aussi à son mécanisme d'exécution. Par exemple, de nombreux langages de programmations sont compilés *statiquement*. Compiler statiquement un programme consiste à le transformer avant de l'exécuter en *code natif*, le code qu'un processeur peut exécuter. Dans le cas de la compilation statique, il faut s'assurer que le compilateur n'a pas introduit de bugs dans les programmes qu'il transforme. Pour vérifier cette propriété, il est possible d'utiliser des méthodes formelles pour prouver que cette transformation préserve la sémantique formelle d'un programme compilé.

Il s'agit alors de compilation formellement vérifiée. CompCert [Leroy 2009a], CakeML [Kumar et al. 2014] et VeLLVM [Zhao et al. 2012] sont des exemples de compilateurs formellement vérifiés. Par exemple, CompCert compile des programmes écrits en C vers de l'assembleur pour plusieurs architectures cibles, comme x86, ARM, PowerPC et RISC-V. CompCert est écrit en Coq, et le code Coq qui transforme le programme peut être extrait en un programme exécutable en OCaml. CompCert contient également une preuve Coq mécanisée qui relie la sémantique formelle du programme compilé à celle du programme d'entrée du compilateur. Les compilateurs sont des programmes complexes et leur vérification formelle est particulièrement ardue. Cependant, cette vérification est la manière la plus sûre de garantir que la compilation n'a pas introduit de bugs.

La plupart des compilateurs ne sont pas vérifiés, et en conséquence se privent de cette garantie. Des tests seuls ne suffisent alors pas à trouver tous les bugs de compilation, comme le montre une étude dédiée à la détection de bugs dans les compilateurs C en utilisant des programmes d'entrée générés aléatoirement [Yang et al. 2011]. Des centaines de bugs ont été ainsi trouvés dans GCC et LLVM, deux principaux compilateurs C, mais aucun bug n'a été trouvé dans la partie formellement vérifiée de CompCert. Ces résultats suggèrent que tester un compilateur n'est pas suffisant pour éviter les bugs de compilation, et que leur vérification formelle apportent une garantie inégalée.

Compilation à la volée Traditionnellement, les langages de programmation ont longtemps été soit compilés, soit interprétés. Pour intérpréter un programme, on utilise un *interprète*, un autre programme qui lit, traduit puis exécute les instructions du programme d'entrée une à une jusqu'à la fin de son exécution. L'interprétation est usuellement plus lente que l'exécution de programmes compilés, mais permet de commencer l'exécution instantanément sans traduire l'intégralité du programme à exécuter.

La compilation à la volée est un autre mécanisme d'exécution qui mélange compilation et interprétation. Sa particularité est de mélanger exécution et optimisation du programme exé-

cuté. Par exemple, certains JITs commencent par interpréter le programme, puis des parties du programme sont graduellement compilées vers du code natif pendant l'exécution. L'exécution dand un JIT est donc un mélange d'interprétation et d'exécution de code natif dynamiquement généré.

Les JITs sont des logiciels particulièrement complexes utilisant des techniques spécifiques modernes. Premièrement, les JITs contiennent souvent des interprètes et des compilateurs statiques entiers (parfois même plusieurs compilateurs statiques, comme dans le cas de WebKit [WebKit 2014]) et des mécanismes supplémentaires pour orchestrer leur interaction. De plus, les JITs utilisent des optimisations et techniques qui leur sont propres pour des temps d'exécution encore plus rapides. Par exemple, de nombreux JITs modernes utilisent des *optimisations spéculatives* pour spécialiser le code compilé dynamiquement. Cette spéculation consiste à prédire le comportement futur du programme, et utiliser cette prédiction pour produire du code spécialisé plus rapide.

Malgré leurs vulnérabilités et leur utilisation grandissante, les JITs ont fait l'objet de peu de travaux de vérification formelle par rapport aux compilateurs statiques traditionnels. Ce manque de formalisation s'explique par plusieurs raisons. Premièrement, les JITs sont plus récents et plus complexes que les compilateurs statiques. Deuxièmement, il n'existe pas de manière standard de concevoir un JIT, et chaque implémentation se distingue par ses composants (tous les JITs ne contiennent pas d'interprète, comme les premières versions de V8 qui compilaient tout code exécuté), ses optimisations ou son architecture. Enfin, les techniques spécifiques aux JITs comme l'optimisation spéculative sont souvent reléguées au rang de détails d'implémentation, sans documentation ni abstraction suffisante à leur formalisation.

Contenu de cette thèse La rapidité d'exécution ne doit pas être synonyme de bugs. De tels bugs dans des JITs pourraient avoir de graves conséquences. Dans cette thèse, nous démontrons que des méthodes formelles peuvent être appliqués aux JITs pour garantir leur correction. Comme les JITs réutilisent des techniques issues de la compilation statique, nous réutilisons des techniques de preuves de compilation statique formellement vérifiée pour prouver la correction de JITs. Dans nos travaux, nous développons des prototypes de JITs vérifiés et exécutables.

Nous apportons les contributions suivantes:

Formalisation des JITs: Nous concevons une architecture pour JITs modernes, contenant chaque élément clé et présentant une spécification claire de chaque composant. Notre architecture contient un mélange d'interprétation et de compilation dynamique avec

des optimisations spéculatives. Nous proposons une sémantique formelle pour les programmes manipulés par des JITs contenant de la spéculation, un mécanisme rarement documenté.

Nouveaux défis de vérification Nous identifions quatre nouveaux défis dans la vérification formelle des JITs. Premièrement, les optimisations dans un JIT sont dynamiques, ce qui n'est pas le cas des compilateurs traditionnels. Nous adaptons la technique de simulation utilisée dans CompCert pour permettre ces optimisations. Deuxièmement, les optimisations spéculatives sont une particularité des JITs. Nous proposons donc des preuves de correction de plusieurs optimisations spéculatives en utilisant la sémantique formelle définie plus tôt. Troisièmement, certains composants d'un JIT moderne (comme l'appel de code natif) ne peuvent pas être entièrement écrits dans le langage de programmation pur d'un assistant de preuve comme Coq. Nous concevons un encodage spécifique pour représenter et raisonner sur un JIT impur en Coq. Enfin, les JITs doivent générer du code natif, et se reposent souvent sur des compilateurs statiques. Nous montrons qu'il est possible de réutiliser le code et la preuve de correction de CompCert dans un JIT.

Nouvelles techniques de preuve: Pour chacun de ces défis, nous proposons des preuves de correction. Toutes nos preuves ont été mécanisées dans l'assistant de preuve Coq. Nous prouvons un théorème qui garantit que l'exécution d'un programme par un JIT est conforme à la sémantique formelle du programme exécuté.

Un JIT vérifié et exécutable: Notre développement de JIT en Coq peut être extrait en OCaml et exécuté. Nous avons ainsi développé un prototype de JIT qui exécute des programmes en possédant toutes les caractéristiques des JITs modernes, tout en étant accompagné d'une preuve formelle mécanisée de correction.

Notre travail démystifie les techniques complexes utilisées par les JITs modernes. Notre développement nous permet d'affirmer que, bien qu'aussi intimidante que pour les compilateurs statiques, la vérification formelle de JITs est réalisable.

Acknowledgment

First and foremost, my biggest thanks go to my advisors, Sandrine Blazy and David Pichardie. You have provided me with your unwavering support and countless opportunities. I have learned so much and really enjoyed working with you. For your guidance, your help, your support, you have my deepest gratitude.

I also want to extend my sincere thanks to my PhD jury, as I am honored to have such an expert panel. I am particularly indebted to my reviewers, Andrew Appel and Xavier Leroy. Our discussions, your comments, questions and reports have helped shape this thesis dissertation.

I have been privileged to collaborate with Olivier Flückiger and Jan Vitek. This opportunity has been incredibly decisive for my PhD, and a fantastic introduction to JITs. My research owes a lot to our work together, thank you. I also thank Roméo La Spina for your work on our JIT. Finally, I would like to heartfully thank the entire Celtique/Épicure team for these memorable years. I will miss working alongside you.

Table of Contents

Table of Figures

Table of Simulations

Chapter 1

Introduction

1.1 Motivation

Just-in-Time compilation is a technique to execute programs which has seen its popularity rise in recent years. *Just-in-Time compilers* (or JITs) have existed since the 1960s [Aycock 2003], but their use has grown greatly for *dynamic languages* such as Python [PyPy 2022], JavaScript [V8 2022], Julia [Julia 2022], R [R 2022], Lua [LuaJIT 2022] or Matlab [MathWorks 2022]. JITs are also used for static languages such as Java [HotSpot 2022], and other use cases such as eBPF in the Linux kernel [eBPF 2022]. The performance offered by JITs for dynamic languages explains that today, most modern web browsers use JITs to execute the programs they find on the Web. Firefox uses SpiderMonkey [Firefox 2022b], Google Chrome and Chromium use V8 [V8 2022] and the WebKit engine of Safari uses JavaScriptCore [WebKit 2022], and all of them use Just-in-Time compilation to execute JavaScript programs.

Together, these web browsers have billions of users, but being complex software, their JITs are exposed to many bugs and vulnerabilities. Such vulnerabilities are too numerous to list exhaustively [Firefox 2022a]. For instance, the Project Zero team has found malware using JIT bugs in JavaScriptCore to execute malicious code on Safari [Project Zero 2019; 2020]. The team has also reported on Chrome vulnerabilities that were exploited before the bugs were reported and fixed [Project Zero 2021a]. The nature of web browsers exposes them to an adversarial setting, as their JITs may execute any input program found on the Web. Attackers knowing a JIT vulnerability can design a JavaScript program that targets it. For instance, bugs in SpiderMonkey have been exploited in a phishing attack against the Coinbase company [Coinbase 2019]. Emails were sent to company workers containing a link that, when opened with Firefox, would execute a program exploiting the JIT bugs to install malware on the company computers. Yet, web browsers are used greatly in our everyday lives and sometimes for sensitive applications that cannot suffer from bugs.

Formal Methods to Guarantee the Absence of Bugs Web browsers and JITs are not the only examples of sensitive software, and many techniques have been developed to find or prevent software bugs that could have dramatic consequences. For instance, one can simply test any program on a set of manually picked or randomly generated inputs. Testing can catch many errors, but cannot guarantee the absence of bugs without checking all possible program inputs, a task often impossible.

For stronger guarantees, *formal methods* have emerged to prove properties about programs. In essence, with carefully chosen abstractions and models, one can mathematically reason about a program without executing it for a particular input. For instance, the static analyzer Astrée has been used to guarantee the absence of a particular type of bugs in the flight control software of Airbus A340 planes [Delmas and Souyris 2007]. Formal methods include various techniques, such as static analysis, model-checking or deductive program verification.

Deductive verification is the act of mechanizing mathematical definitions and reasoning in a *proof assistant*. Proof assistants are either automatic like Why3 [Filliâtre 2012] or F* [Swamy et al. 2013], or interactive such as Coq [Coq 2022], Isabelle/HOL [Nipkow et al. 2002] or ACL2 [Kaufmann et al. 2000]. These are software designed to conduct and check mathematical proofs with mathematical rigor. Unlike in pen-and-paper proofs, this eliminates the possibility of forgetting edge cases. Using a proof assistant confers a lot of confidence in a proof, as one only needs to trust the proof assistant to be convinced of the rigor of the proof and not manually check it.

To mathematically reason about program executions, formal methods like deductive program verification define and reason on *formal semantics*, a mathematical and rigorous description of program behaviors. The task of defining formal semantics has been done for many programming languages, including C [Norrish 1998, Blazy and Leroy 2009, Krebbers and Wiedijk 2011], JavaScript [Bodin et al. 2014] or WebAssembly [Watt et al. 2021]. Given a program and its formal semantics, one can reason about the program behaviors using *program logics*. For instance, a cryptographic primitive written in C has been proved correct in Coq using the C formal semantics and a program logic called *separation logic* [Appel 2015].

Formally Verified Compilation However, to trust a program being executed on a computer, one must not only trust the source code of the program, but also the mechanism that executes it. For instance, many programming languages are compiled *ahead of time*, meaning that they are transformed before their execution into *native code*, the code that a computer processor executes. In that case, when executing a compiled program, one has to trust that the

compiler did not introduce any bugs in the program. Otherwise any analysis or verification performed on the source program semantics would be rendered useless by a compiler that changes the behavior of the program it produces. Compilers transform a program from a language to another and one can use formal methods to prove that this transformation preserves the behavior of the input program.

This is called *formally verified compilation*. Examples of formally verified compilers include CompCert [Leroy 2009a], CakeML [Kumar et al. 2014] and VeLLVM [Zhao et al. 2012]. For instance, CompCert compiles C programs to assembly for several target architectures, including x86, ARM, PowerPC and RISC-V. CompCert is written in Coq, and the Coq code that constitutes the compiler can be extracted to an equivalent OCaml program and run independently. CompCert also comes with a mechanized Coq proof that relates the formal semantics of the compiled program to the formal semantics of the input program of the compiler. Compilers are complex and large software and while their formal verification is a daunting task, it provides a way to guarantee compilation correctness with the highest assurance.

Most compilers are not verified, but cannot then guarantee that the compiled program behaves as specified by the input program. And despite all the testing ahead of time compilers go through, compilation bugs still find their way. For instance, Yang et al. [2011] conducted a study dedicated to finding bugs in various C compilers using randomly generated programs. Hundreds of bugs were found in GCC and LLVM, two of the most widely used C compilers, while no bug was found in the formally verified part of CompCert. These results confirm that testing is not enough to avoid compilation bugs and that formal methods and formal verification bring substantial guarantees to compilation.

Just-in-Time Compilation Traditionally, programming languages were either compiled or interpreted. With interpretation, one uses an *interpreter*, another program that reads, translates and executes instructions of the input program until execution finishes. Interpretation typically results in slower execution times than compilation, but faster start-up as there is no need to transform the entire program before executing it.

Just-in-Time compilation is yet another mechanism to execute a program, reusing techniques from both compilation and interpretation. In essence, Just-in-Time compilers interleave execution and optimization of the program to execute. [1] For instance, many JITs start with interpretation, but the parts of the program that are run often are gradually compiled

1. Others sometimes use "Just-in-Time compilers" to refer only to the optimization part of that process. In this work, a JIT means the entire engine, including both execution and optimization.

to native code during execution. The execution part of the JIT then interleaves interpretation and execution of dynamically generated native code.

JITs are complex software relying on cutting-edge techniques. First, they often contain entire interpreters and compilers (sometimes several of them, like the four-tier compilation design used in WebKit [WebKit 2014]) and a precise interplay between all their components. Second, they also use JIT-specific optimizations and features for faster execution times. For instance, many modern JITs use *speculative optimizations* to specialize the code they dynamically compile. This feature consists in trying to predict (or speculate on) the future behavior of the program, and use these predictions to produce faster code.

Despite their vulnerabilities and their growing use, JITs have been scarcely formalized and the topic of very few formal verification works, compared to standard ahead of time compilation. This lack of formalization can be explained by several reasons. First, JITs are more recent, more complex and less understood than traditional compilers. Second, the design space of modern JIT compilers is particularly large and ungoverned by any standard. There is no common way to do speculative optimizations, no common set of components (earlier V8 versions did not use interpretation for instance, but compiled everything), no common architecture across all modern JITs. Finally, JIT-specific features like speculation are often relegated to implementation details that are neither clearly abstracted nor documented.

1.2 Thesis and Contributions

Fast program executions should not come with bugs and vulnerabilities. With so many program executions relying on Just-in-Time compilation, we believe that formal methods should be used to guarantee the correctness of JITs. Bugs in JITs can be dangerous and more efforts should be dedicated to prevent them. JITs are also in dire need of demystification if one ever wants strong guarantees on their execution. In that perspective, we investigate in this work the formal verification of Just-in-Time compilation.

Because modern JITs are huge pieces of software, we believe that formal verification efforts should first focus on proving the correctness of model JITs that capture the essence of JIT compilation, instead of the insurmountable task of fully verifying a rapidly evolving modern JIT used in production. As JITs reuse standard compilation techniques to generate native code, we also argue that formally verified JITs should reuse formally verified ahead of time compiler proofs and techniques to alleviate the proof burden of such complex software. Just like the Coq code of CompCert can be extracted to OCaml and executed, formally verified JITs should also

be executable to run programs.

This document presents our solutions to develop and prove the correctness of JITs. Specifically, we present the following contributions:

JIT formalization: We introduce an architecture design for modern JITs, that clearly specifies the role of each component and their interplay. It includes dynamic compilation and interpretation, but also JIT-specific features like speculative optimizations. We also give precise program semantics to pieces of code containing speculation, a JIT feature that has been mostly undocumented.

New Verification Challenges: We identify four main JIT-specific verification challenges, listed in Section 1.2.1. We argue that these capture the essence of what separates the formal verification of JITs from the formal verification of standard ahead of time compilers.

New Proof Techniques: We develop new proof techniques for proving correct each of these new verification challenges.

Verified and Executable JIT: Every proof presented in this work has been mechanized in Coq. This results in an executable and formally verified JIT. It comes with a correctness theorem that states that executing a program with the JIT produces a behavior that corresponds to the program formal semantics.

Our formal work demystifies the complex mechanisms involved in JIT compilation, and our developments show that formally verified JIT compilation is feasible.

1.2.1 Four JIT-Specific Verification Challenges

The four JIT-specific verification challenges that we tackle in this work are the following:

Dynamic Optimizations JITs optimize and compile parts of their programs dynamically, during their executions. The program being executed at some point in time depends on what dynamic optimizations have been made before and cannot be known before execution.

Speculation Modern JITs speculate on the future behavior of the code they compile. They produce specialized code relying on assumptions that are checked during execution. This can lead to significant speed-ups, but requires complex mechanisms like *deoptimization* when the predictions are wrong. Deoptimization consists in manipulating the execution stack to switch from the execution of optimized code back to the execution of the original code.

Effectful JITs Some JIT components simply cannot be written in the pure programming languages of proof assistants like Coq. This includes calling native code that has been generated dynamically, or the use of global shared data-structures when execution is split between interpretation and native executions. For instance, a JIT execution stack and its heap are manipulated by both its interpreter and its dynamically generated native code.

Proof Reuse for Native Code Generation JITs reuse techniques from standard compilation when generating native code dynamically. Formally verified JITs should reuse proofs and proof techniques of formally verified ahead of time compilers. For instance, we reuse the simulation methodology of CompCert to modularly prove compiler transformations correct, as well as its proved algorithms to generate native code.

These challenges are key to the feasible formal verification of modern JITs that speculate and generate native code. We design new proofs and proof techniques to solve each of these verification challenges.

1.2.2 Collaborations and Publications

Unless listed here, everything presented in this document and the artefacts is the work of the author of this thesis, advised by Sandrine Blazy and David Pichardie.

Our work on speculative optimizations was done in collaboration with Olivier Flückiger and Jan Vitek. In particular, the CoreIR language (Section 5.1) was designed conjointly with them. The evaluation of the pure JIT presented in Section 8.3.3 was conducted by Olivier Flückiger. This includes the unverified Lua Lite frontend, the unverified LLVM backend and the performance comparison of Figure 8.4.

Two JIT upgrades were implemented in collaboration with Roméo La Spina, as part of his Master's project. The first one is the liveness analysis mentioned in Section 5.2.1. The second one allowed the inlining pass presented in Section 5.2.6 to trigger more often than in our original version. All other Coq proofs were developed by the author of this thesis.

Parts of the work described in this thesis have been presented at the following conferences:

CoqPL 2020 Our project was first presented at the CoqPL workshop:
Towards Formally Verified Just-in-Time Compilation [Barrière et al. 2020].

POPL 2021 A first version of our formally verified JIT has been the subject of an article at a peer-reviewed conference. The objective of this first work is to understand and formalize speculative optimizations in JITs. It includes our solutions to solve the first two

challenges of Section 1.2.1: this JIT optimizes dynamically and performs speculation. Chapter 5, Sections 4.5 and 8.3.3 borrow from this publication.
Formally Verified Speculation and Deoptimization in a JIT Compiler [Barrière et al. 2021].
Its development is available as an artefact:
https://github.com/Aurele-Barriere/CoreJIT.

POPL 2023 Next, we have extended our work to solve the two remaining challenges. We present a methodology to develop an effectful JIT that reuses CompCert and its proofs to dynamically generate native code. Chapters 6 and 7 borrow from this publication.
Formally Verified Native Code Generation in an Effectful JIT
or: *Turning the CompCert backend into a formally verified JIT compiler* [Barrière et al. 2023].
Its development is available as an artefact:
https://github.com/Aurele-Barriere/FM-JIT.

1.3 Outline

This document is organized as follows: Chapter 2 presents Just-in-Time compilation. In particular, we present the speculation technique used in many modern JITs, and previous work on JIT formalization and formal verification. Chapter 3 presents the formally verified C compiler CompCert. In particular, we present its simulation framework that we reuse to prove the correctness of formally verified JITs. Chapter 4 presents our approach to developing formally verified JITs. This approach is designed to allow the reuse of proof techniques from CompCert. In this chapter, we also introduce our solution to the first JIT-specific verification challenge, dynamic optimizations. In Chapter 5, we present CoreIR, the intermediate language that our JIT is executing, optimizing and compiling. This language is designed to support speculative optimizations. We then present our solution to the second JIT-specific verification challenge by implementing and verifying speculative optimizations in a JIT. Chapter 6 discusses the third JIT-specific verification challenge and introduces our solution, using a free-monadic encoding of effectful JITs. Chapter 7 then shows how we can reuse the CompCert backend and its proof to tackle the last JIT-specific verification challenge. This results in formally verified JITs that dynamically generate native code using CompCert. Finally, Chapter 8 shows that the various proof techniques we introduced are composable, and discusses our formally verified JIT implementations. Chapter 9 concludes and presents possible future work.

After the conclusion, there is an appendix on page 153 containing, for each definition presented in this document, a link to the corresponding definition in our developments.

Chapter 2

Background on Just-in-Time Compilation

Using a standard compiler is useful to produce fast programs. One first writes one's program in a high-level language, then trusts the compiler to produce semantically equivalent code in the assembly language of the target architecture. While transforming the program, most standard compilers are *optimizing compilers*, in the sense that they include optimization passes, code transformations that make the resulting program execute faster. In that setting, every piece of the program is transformed before the execution even starts. For that reason, these standard compilers can be referred to as *static compilers* or *ahead of time compilers*.

On the other hand, using an interpreter is useful to start executing a program quickly, as there is no need to translate the entire program. While interpretation is slower, interpreters typically have a faster startup. Moreover, as an interpreter has access to dynamic information while still working on its source code (and not a compiled version), interpretation leads to better diagnostics of run-time errors in general. Interpreters are often used for dynamic languages, whose ease-of-use and conciseness are popular among developers.

Just-in-Time compilers try to offer the best of both worlds. In a JIT, programs start by being interpreted, but code that might be run frequently (loop bodies or hot functions) may be compiled and optimized dynamically. When encountering such code, the JIT can then simply execute the compiled version, then go back to the interpreter for the rest of the execution. As a result, a JIT interleaves interpretation, compilation and execution of compiled code. This allows JITs to be much faster than interpreters alone, while still benefiting from low startup times. As compilation happens during program execution in JITs, fast compilation times are desired. Compiling only the frequently run code avoids spending compilation time on the entire program. At its inception, the main advantage of using JITs was to save memory space, as one did not need to compile the entire program and save the entire output of a compiler [Aycock 2003]. Today, by mixing various techniques, JITs bring flexibility and trade-offs. For instance, some JITs like JavaScriptCore [WebKit 2014] have multiple compilers, some performing ag-

```
for (int i=0; i<length; i++) {
  sum[i]     = a + array[i];
  product[i] = a * array[i];
}
```

Figure 2.1 – A simple loop where types stay the same

gressive optimizations, and others that are less optimizing but quicker to execute, and can use one or the other depending on how much they think some code is going to be run later.

2.1 Speculation in Just-in-Time Compilers

Another advantage of Just-in-Time compilation is the possibility of *speculative optimizations*. By speculating on the behavior of the compiled code, this technique can cause significant speedups. As optimizations happen dynamically in a JIT, the parts of the program that are compiled are usually compiled after they have been run enough times to be considered as *hot* code, code that is likely to be executed a lot. In these first executions of the code, JITs typically use *profiling* to record information about the program execution. That information can be leveraged to speculate on the future program behavior at compile time. Profilers are an essential component of modern JITs. Their role is to observe the code being interpreted. Using their observations, they determine what the hot code is. In JITs with speculation, they also suggest likely invariants of the code that the JIT optimizers should speculate on.

Speculation in JITs can be reminiscent of *profile-guided* optimizations in ahead of time compilers [Pettis and Hansen 1990]. To use profile-guided optimizations, the program is compiled a first time, called *instrumentation* compilation. Then, test runs of the compiled program are executed and profiling information is gathered about these test executions. Finally, the program is compiled a second time, using this profiling information to generate code that is particularly optimized for the expected behavior. In contrast, the profiling information used by JITs is gathered dynamically, during the beginning of the program execution. The dynamicity of optimizations in JITs eliminates the need for both the instrumentation compilation and the test executions.

2.1.1 Speculative Optimization Example

For instance, consider the code Figure 2.1. If the language is dynamically typed, the interpreted code corresponding to an iteration might need to check the type of `a` and `array`, then

```
if (a is int && array[i] is int) {
  sum[i] = int_add(a, array[i]); }
 else if (a is float && array[i] is float) {
   sum[i] = float_add(a, array[i]); }
 else if (a is string && array[i] is string) {
   sum[i] = string_add(a, array[i]); }
 else { error "Wrong addition"; }

if (a is int && array[i] is int) {
  product[i] = int_mult(a, array[i]); }
 else if (a is float && array[i] is float) {
   product[i] = float_mult(a, array[i]); }
 else if (a is string && array[i] is string) {
   product[i] = string_mult(a, array[i]); }
 else { error "Wrong multiplication"; }

i = i+1;
```

Figure 2.2 – The interpreted instructions of Figure 2.1

use the correct addition and product primitives. Simple programs end up having complex control flows. This might lead to the instructions of Figure 2.2 for each iteration. Executing a program containing such code with a JIT, one starts by interpreting the loop. But if the loop is executed often, a profiler might suggest to optimize and compile it. When the JIT compiles the loop, it has already been executed some number of times. If during the previous executions, the profiler has detected that `a` and `array[i]` were always integers, then we can reasonably speculate that this will be the case for the next executions as well. One way to speed up future executions is to produce and compile the code on Figure 2.3. In that code, `a` and `array[i]` are assumed to be integers, which greatly simplifies the next instructions.

This simplified example captures what many modern JITs do. They produce code that resembles the one on Figure 2.3. In the next execution of the loop, the new code is executed. The assumptions are checked at run-time, and if they hold, execution can proceed in this new optimized version. Of course, the next execution may execute the loop for other types. In that case, the JIT should go back to interpreting the original version as in Figure 2.2, slower but correct for any type. As a result, the `assume` instructions of Figure 2.3 are more than simple assertions. The process of going back to the original version is typically called *deoptimization* and is described in section 2.1.2.

Speculation is used a lot in JITs, especially for dynamic languages. JITs can speculate on various aspects of an execution, including types and values. For instance, in JavaScript without speculation, all arithmetic results are *boxed*. This boxing consists in wrapping any primitive

```
assume (a is int);
assume (array[i] is int);

ai = array[i];
sum[i] = int_add(a, ai);
product[i] = int_mult(a, ai);
i = i+1;
```

Figure 2.3 – The optimized iteration of the loop of Figure 2.1, with speculative optimization

value in an Object. This has the advantage of giving developers the opportunity to use methods on primitive values, for instance converting an integer to a string using the corresponding method. However, when such methods are not used, this boxing leads to many unnecessary instructions to box, typecheck and unbox every value during arithmetic operations. With type speculation however, most boxing and typechecking can be eliminated, as shown in the SPUR JIT [Bebenita et al. 2010]. JavaScriptCore is another example of a JIT using type speculation for better performance [WebKit 2020]. Some JITs speculate on other kinds of behaviors. For instance, the HotSpot compiler can make assumptions about the class hierarchy before compiling [Paleczny et al. 2001]. However, without formal verification, the complexity associated with speculation can lead to vulnerabilities. For instance, V8 speculates on some variable types to best decide how to represent them in a compact way (representing small unboxed integer values on 31 bits for instance). But the incorrect removal of an assumption check can lead to dangerous exploits, even allowing attackers to execute arbitrary code [Project Zero 2021b].

2.1.2 Deoptimization and On-Stack Replacement

While speculation leads to speed-ups if the profiler correctly predicted the future behavior of the program, there is no guarantee that future calls to the optimized code will behave as predicted. For instance, maybe the next call to the loop of Figure 2.3 will be with a string value for `a`. In that case, when executing the specialized code, the first assumption check will fail. The JIT must then find a way to return to the interpreter.

This is called *deoptimization*, as one goes from an optimized, specialized version of some code to a more generic and less optimized version. In a setting where assumptions can be inserted anywhere in optimized code, this requires a technique known as *on-stack replacement*. One needs to remove the current stackframe corresponding to the execution of the specialized code (often machine code for modern JITs), and replace it with a stackframe for the interpreter executing the original version.

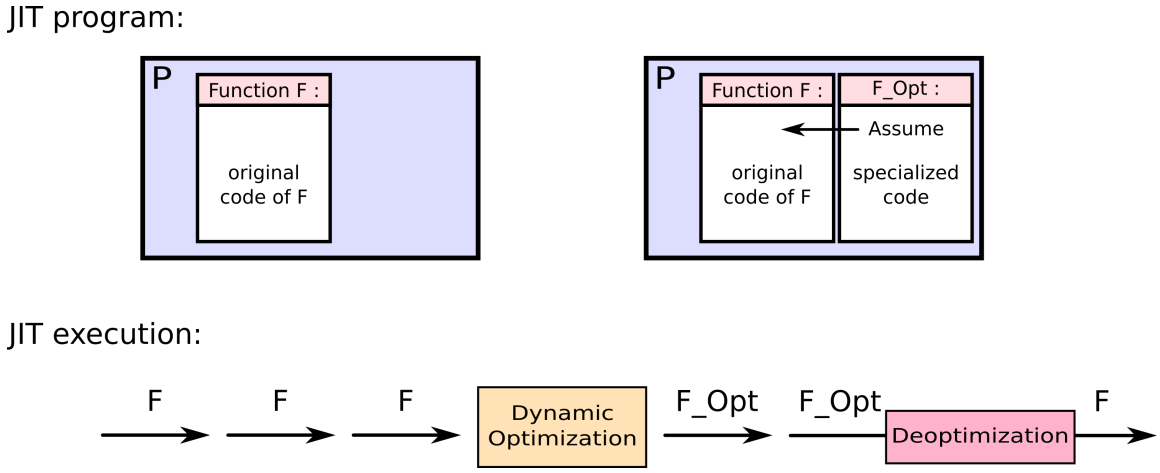

Figure 2.4 – Timeline of a JIT execution

Inlining is a standard compiler optimization where one substitutes a function call with the code of the function itself. While it requires more space as it copies instructions from a function to another, it can speed up some executions as function calls can sometimes hinder performance. It also enables other optimizations to perform better, when they use *intraprocedural* analyses that do not inspect the code of called functions. Deoptimization can be made even more difficult when combining *inlining* and speculation: if an assumption was inlined in a function and fails, one must reconstruct an extra interpreter stackframe for the original version of the caller function. Examples of on-stack replacement with inlining are given in section 5.2.6.

Implementing on-stack replacement is a difficult and language-specific task, where one needs to manipulate the execution stack. One way to do so is to have failing `assume` instructions jump to some arbitrary low-level *compensation* code outside of the function that will reconstruct the missing stackframe. Another way consists in augmenting the speculative instruction `assume` with *deoptimization metadata*, a representation of the deoptimization information needed to reconstruct the interpreter environment. With such an approach, used for instance in Graal [Duboscq et al. 2013] or Sourir [Flückiger et al. 2018], the synchronization between optimized and nonoptimized code is made explicit in the function. We find the latter approach more elegant to define formal semantics of speculation and deoptimization in Chapter 5.

Imagine a program that consists of five invocations of the function `F`, where the first four calls have the same arguments and the last one differs. Figure 2.4 shows a possible execution with a JIT that generates an optimized version of `F` after three invocations and then deopti-

mizes in the last call. After the first three calls, the JIT decides to optimize function `F`. It adds in the JIT program a new version of `F` called `F_Opt`, specialized for the value of the arguments that have been seen so far. This new version starts with an `assume`, and then contains specialized code. On the fourth call, the JIT can simply execute this new version. The dynamic check holds, as the fourth call also executes `F` with the same arguments. For the fifth call however, the dynamic check fails because the arguments have changed, and deoptimization is performed to go back to executing the original version of `F`.

Note that on-stack replacement has another optional use in JITs that can lead to speedups. For deoptimization, one uses on-stack replacement as an exit from compiled code, to go from more optimized code to less optimized code. But some function-based JITs also use on-stack replacement as an entry into compiled code. For instance, if a function has not been optimized yet, but is stuck in a loop for a long time, some JITs compile the function before the loop terminates, and use on-stack replacement to jump into the optimized code and finish the function execution there. This use was introduced in SELF [Hölzle and Ungar 1994], but is now also used in HotSpot, in the Jikes RVM [Fink and Qian 2003], in JavaScriptCore [WebKit 2014] or V8 [V8 2021b]. While on-stack replacement as an exit is required for the correctness of deoptimization, on-stack replacement as an entry is an optional JIT-specific optimization. A JIT without on-stack replacement as an entry can simply wait for the next function call to jump into compiled code. Our work focuses on on-stack replacement as an exit for deoptimizations.

2.2 Various JIT Designs

There is no standard way to build a JIT. Some JITs have interpreters, while some do not, like earlier versions of V8 that compiled everything. Most modern JITs use speculations, but some may not and compile hot code without specializing it. Some modern JITs have several tiers of compilation. For instance, V8 and SpiderMonkey first use a baseline compiler to generate bytecode, then use an optimizing compiler. Many WebAssembly interpreters like Wasm3 or those used in JITs rewrite the code before interpreting it, but others do not. With such a clear absence of standards, JITs are hard to compare and formalize. The development of our formally verified JIT is guided by some design choices that we present here.

Intermediate Representation Just like standard compilers, when transforming code, JITs do not usually go directly from the source language to the target language. They instead go through various intermediate languages, including bytecodes, called intermediate representa-

tions or IRs. This allows more modular transformations. IRs in JIT are sometimes represented as simple control-flow graphs (CFGs), where instructions correspond to nodes of the graph and point to their successors (like in the DFG compiler of JavaScriptCore). Some JITs use other representations that extend CFGs, like SSA [Rosen et al. 1988] (used in CraneLift for instance) or Sea-of-Node [Click and Cooper 1995] (used in HotSpot or the TurboFan compiler of V8). In this work, we focus on simple CFG representations. This design choice is not specific to JITs, and also appears in static compilers. Our use of simple CFGs allows us to stay close to the program semantics of CompCert RTL, introduced in Chapter 3.

Granularity JITs can also be distinguished by the *granularity* of the code they decide to compile. For instance, *function-based* JITs compile entire functions (like HotSpot or SELF), some compile only basic blocks (like QEMU). Others, called *trace-based* JITs, compile execution traces (like TraceMonkey), where a trace usually refers to a simple sequence of program instructions. Similarly, some *region-based* JITs like HHVM [Ottoni 2018] compile *regions*, generalizing traces to potentially interprocedural pieces of code that can contain branches. In this work, we focus on nontracing function-based JITs, compiling entire functions. This choice makes the reuse of formally verified standard compilers easier.

Speculation Points The use of speculation in JITs can also be classified in two categories. Some insert explicit speculation points in the optimized code, like the `assume` instruction of the example on Figure 2.3. This allows to speculate anywhere in a function, but requires complex deoptimization techniques. Another approach called *contextual dispatch* [Flückiger et al. 2020], consists in generating several versions of a function with different assumptions. For instance, one could have one version of a polymorphic function specialized for integers, and another version specialized for strings. This restrains where speculation can happen (only at the beginning of functions), but avoids the need for deoptimization, as the JIT always dispatch to a version that corresponds to its current context. This approach has been used by Julia, dispatching functions specialized with the types of their arguments [Bezanson et al. 2018]. In this work, we choose to investigate the verification of the speculation point approach (using instructions like `assume` in Figure 2.3), to demystify and formalize the complex mechanisms of deoptimization and on-stack replacement in modern JITs.

Profiling Heuristics Profiling and hot code detection heuristics occur in various degrees of complexity. Many function-based JITs simply use thresholds to determine when a function is

hot. Thresholds are used for instance in HotSpot [HotSpot 2022], where after a given number of calls, a function is considered hot. Others have tried more complex approaches. For instance, taking into account not only the number of time a function is called, but also how many loop iterations have happened during these calls, or an estimate of the time and space the function would require to compile. This has been investigated in the PAYJIT policy [Brock et al. 2018], which scales thresholds with function sizes. V8 also has varying thresholds depending on the function size and other analyses. In JavaScriptCore [WebKit 2020], the threshold for compiling a function is scaled by a complex formula, $(0.825914 + 0.061504 \times \sqrt{S + 1.02406}) \times 2^R \times \frac{M}{M-U}$ where S represents the size of the function, R the number of times it has already been compiled [1], M represents an amount of executable memory left and U an estimate of the total memory used after compiling the function. Such heuristics come from experiments on benchmarks rather than formal correctness arguments. In this work, the profiler heuristics we implemented were simple unscaled thresholds. However, as explained in Section 4.3, our proofs could support any other kind of heuristics.

2.3 Related Work on JIT formalization

JITs have been scarcely formalized so far. Yet, some other contributions have started investigating the issue of formalizing some parts of modern JITs.

For instance, focusing on the range analysis Javascript JITs perform, a DSL to write and prove the correctness of range analysis in JITs has been developed, called VeRA [Brown et al. 2020]. This tool uses a SMT solver to automatically prove that some range analysis is correct. VeRA has been used to write several range analyses used in Firefox and prove them correct. This also allowed to find a bug in an existing analysis of Firefox, that was consequently fixed. Compared to using interactive proof assistants, using SMT solvers allows the automatic proving of some properties. However, some properties are either difficult to express in the logics used in SMT solvers, or too difficult to be automatically proved. While a SMT solver is adapted to prove the correctness of a range analysis, the formal verification of entire JITs would benefit from the expressive specification language of interactive proof assistants like Coq.

For trace-based JITs, Guo and Palsberg [2011] focus on the soundness of trace optimizations. Sound trace optimizations may differ from standard ones. Indeed, when optimizing a trace, the rest of the program is not known to the optimizer. As such, optimizations such as

1. A function can get re-compiled from scratch if its speculations triggered deoptimization too many times. This comes with other complex heuristics to decide when to throw away an optimized version.

dead store elimination are unsound when optimizing a trace: a store might seem useless in the trace itself, but actually impacts the semantics of the rest of the program. Similarly, new optimizations can be done in this context: free variables of the trace can be considered constant for the entire trace. To characterize sound and unsound optimizations, the authors define formal semantics for trace recording and bail-out mechanism for speculative optimization. Finally, they define a bisimulation-based criterion for sound optimization. Checking for soundness then reduces to checking that the criterion is satisfied. Then, a theorem proves that using such sound optimizations, program semantics are observationally equivalent whether tracing is performed or not. While the formal framework introduced in this work successfully expresses and proves some of the correctness properties that a trace-based JIT should satisfy, this is not enough to develop formally verified executable JITs. The framework is not implemented and proofs are not mechanized.

Myreen [2010] presents another work that targets the formal verification of an entire JIT, and not just one of its components. It contains a JIT for a stack-based bytecode, and in particular targets the challenge of dynamically generating x86 code. Proofs are mechanized with HOL4, and the result is an executable JIT which dynamically generates native code for each called function. To that end, this work defines semantics for self-modifying x86 code, and a program logic to reason about such code. When x86 code is generated, it contains jumps to a code generator that may modify the x86 code that called it. This work successfully addresses the first JIT-specific challenge of Section 1.2.1. In particular, the JIT can compile code to x86 dynamically and this process is interleaved with execution. The last JIT-specific verification challenge, reusing existing proofs of native code generation, is also handled successfully. This JIT reuses a previous proof-producing (but nonoptimizing) compiler [Myreen et al. 2009] to generate x86 code equivalent to a bytecode instruction. However, some JIT-specific features could be difficult to implement and prove using this approach, such as speculation and on-stack replacement or interleaving multiple tiers of execution (interpretation and native code execution). Self-modifying code is one way to implement JITs, but JITs can also generate several versions of the same function instead of modifying existing code. We believe that our work proposes a design more typical of modern JITs, with JIT-specific features like speculation and enabling reasoning about the precise interoperability between interpretation and execution of machine code.

Others have explored the formal verification of BPF JITs. The BPF (Berkeley Packet Filter) bytecode language provides an interface for filtering packets in a kernel. For instance, an application can request a packet by providing a BPF filter, a program in a restricted lan-

guage that the Linux kernel then executes with a BPF engine and decides if the packets gets downloaded and executed or not. This architecture can also be used to filter system calls that can be accepted or denied by the kernel. BPF engines can interpret the bytecode, but can also decide to compile it to native code instead for efficiency. This is the case in the eBPF implementation [eBPF 2022] that extends the BPF architecture. Such BPF engines can be called BPF JITs, as the kernel uses the BPF engine to compile the code dynamically, as it receives it. First, Jitk [Wang et al. 2014] is a formally verified BPF JIT, which consists in several components. A new input language, SCPL, is used by applications to specify system call policies. This input is transformed into the BPF bytecode language. Then, the BPF code is transformed to Cminor, an intermediate language of CompCert (see Chapter 3), where it can be compiled to native code. These transformations are formally verified and mechanized in Coq and Jitk can be extracted to OCaml. Later, work on Jitterbug [Nelson et al. 2020] also presents a formally verified BPF JIT, where proofs are automated using SMT solvers. Jitterbug is written in a subset of C that can more easily be integrated in the Linux kernel. However, both of these work only formally verify the compilation component of a JIT, and not an entire JIT architecture, including the kernel code that calls the compiler and receives its result. Their correctness theorems resemble the one of a static compiler. BPF is also a restricted language and for instance, both of these work only compile terminating programs. There is no speculation, and the compiled code is not interleaved with other execution engines: it only interacts with the rest of the kernel when returning. While these restrictions make sense for a BPF JIT, in this work we aim at formally verifying not only the compilation part of a JIT, but an entire architecture that calls this compiler and may use speculation or interleave the native code execution with interpretation.

Finally, focusing on speculations, semantics preservation proofs for speculative optimizations were first studied in the Sourir intermediate representation [Flückiger et al. 2018]. Sourir introduced formal semantics for speculative instructions like the `assume` of Figure 2.3, and arguments for the correctness of inserting and manipulating them. This is an important step in demystifying speculation in JITs. Our own formalization of speculation in Chapter 5 uses speculative instructions inspired by this design and has been done in collaboration with two of the authors behind Sourir, Olivier Flückiger and Jan Vitek.

We believe that the formal verification of JITs should draw inspiration from the existing work on formally verified ahead of time compilers. In the next Chapter, we present the CompCert compiler, a C compiler developed and proved using the Coq proof assistant. The methodology for developing formally verified JITs that we present in this work is inspired by CompCert and reuses some of its proofs and techniques.

Chapter 3

Background on the CompCert Compiler

3.1 A Formally Verified C Compiler

CompCert [Leroy 2006; 2009a, Leroy et al. 2016] is the first commercially available optimizing static compiler that is formally verified. It compiles C programs into assembly programs, for the following target architectures: ARM, PowerPC, RISC-V and x86 (32 or 64 bits). CompCert includes several optimizations to produce fast executable programs. CompCert is an industrial-strength compiler: the performance of the compiled programs is similar to using GCC at optimization level 1 (with the `-O1` flag). CompCert supports almost every feature of the ISO C99 standard version of C. The supported subset is called CompCert C.

CompCert is a cornerstone of formally verified compilation. Due to the handling of a realistic language, C compilers are large and complex pieces of software, and their formal verification is challenging. For instance, CompCert C contains 22 different kinds of expressions, and 38 semantic rules to describe its behaviors. The x86 assembly language contains 177 different instructions. CompCert contains tens of thousands lines of both Coq code and proofs. Instead of verifying an existing compiler, CompCert follows a "software-proof" codesign, where the compiler is developed from scratch in Coq along with its proof. The compiler can be extracted to OCaml. This results in an OCaml program that can be compiled and run to compile realistic C programs.

Most of the compiler is written in the specification and programming language of Coq, Gallina. These parts of the compiler can be directly proved correct in Coq. Some transformations however, like register allocation, make use of a technique known as *translation validation*, where the transformation itself is written in OCaml, and not verified. A *validator* written and proven in Coq then checks *a posteriori* that the output of the OCaml transformation is correct. This can be useful when the validator is easier to prove correct than the transformation itself. However, the validator could reject the output of the transformation, in which case compila-

tion would fail. When using CompCert, the possibility of compile-time errors cannot be ruled out. In practice, the validator can be evaluated on a test suite to check experimentally that it does not reject correct transformations [Rideau and Leroy 2010].

CompCert Architecture CompCert can be decomposed into several parts. Its architecture can be seen on Figure 3.1. Each box represents an IR of the compiler. First, a *parser* converts a C program into the abstract syntax tree (AST) representation of CompCert C code. ASTs are used to represent programs in Coq in all of CompCert. Parsing is formally verified since CompCert 2.3 [Jourdan et al. 2012].

Second, a *compiler* transforms that CompCert C AST into an assembly AST. This is the most significant part of CompCert, and is entirely verified. This Coq transformation can be decomposed into a sequence of transformations, called *passes*. CompCert includes many such passes, decomposing the correctness of the compiler into modularly composable correctness arguments of each pass. There are two kinds of passes, *lowering* passes and *optimizing* passes. Lowering passes consist in translating a program in some intermediate representation to an equivalent one in the next IR. CompCert contains 11 IRs, from CompCert C to assembly. Different IRs have different properties. For instance, CompCert C allows side-effects inside expressions, but this is not the case in Clight. The lowering pass that generates Clight then consists in moving side-effects out of expressions. This allows all following IRs to avoid reasoning about expressions with side-effects.

On the other hand, optimizing passes stay within the same IR, but transform the code into a more efficient one. Most optimizing passes happen at the Register Transfer Language (RTL) level in CompCert. RTL is a standard compiler IR where values are stored in an unbounded number of pseudo-registers. The RTL code is represented with a simple CFG (control-flow graph), where each instruction is associated to a label. Register allocation, assigning pseudo-registers to machine registers, is done later in the lowering pass that generates LTL code from RTL. Optimizations done in CompCert are detailed in Section 3.1.1.

In this work, we distinguish the *frontend compiler*, that transforms CompCert C programs into RTL programs, from the *backend compiler*, that optimizes RTL and generate assembly programs. [1] In Chapter 7, we present how to reuse the backend part of the CompCert compiler in a JIT, from RTL to x86 assembly.

Finally, a third part consists in *assembling* and *linking* the generated assembly AST. Some

1. Note that in CompCert's parlance, the "backend" starts at the Cminor level. However, in this work, "backend" refers to the part that we reuse for native code generation in a JIT and starts at the RTL level.

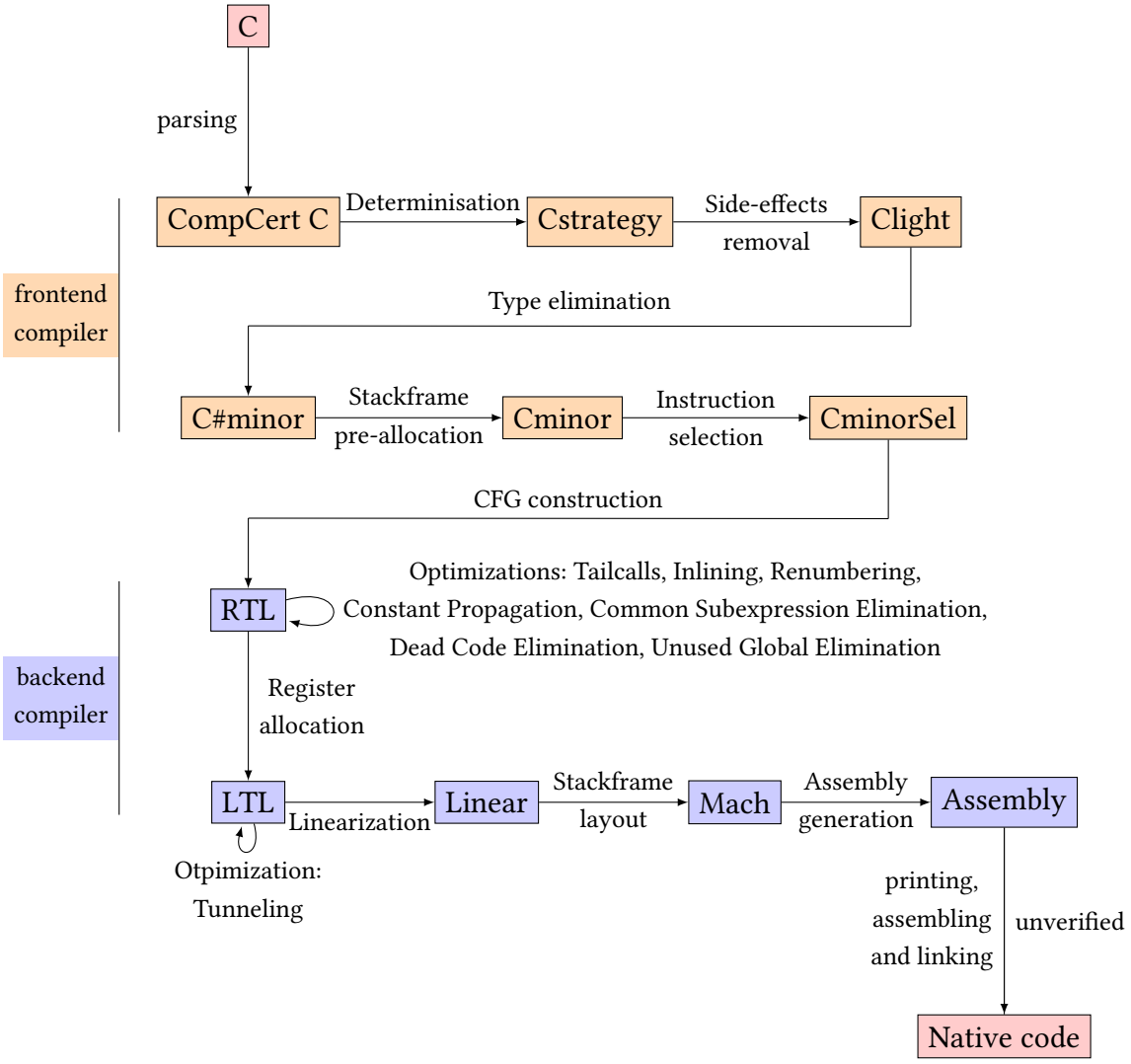

Figure 3.1 – CompCert Architecture

```
main() {
  x3 = 37
  x2 = 13
  x1 = x3 + x2
  x4 = x1
}
```

```
main() {
  x3 = 37
  x2 = 13
  x1 = 50
  x4 = 50
}
```

Figure 3.2 – A RTL program before and after the constant propagation optimization

unverified OCaml code prints the assembly AST into an assembly file. That file can be assembled (transformed into native code) and linked with other libraries, using the linker from `gcc` for instance. Note that these last steps are unverified, although the Valex tool has been developed to perform translation validation for PowerPC and ARM assembling and linking in the commercial release of CompCert [AbsInt 2015]. Also, the CompCert theorem is restricted to whole programs that have been entirely compiled at once with CompCert, while C compilers are sometimes used to compile incomplete programs that are then linked to libraries compiled independently.

3.1.1 CompCert Optimizations

In RTL, CompCert can perform several optimizations like *constant propagation*. In constant propagation, one performs static analysis to find out possible simplifications. For instance, registers that are known to be constant can be replaced by their value directly. An example is shown on Figure 3.2. In that program, static analysis can automatically find that at the time of the addition, `x3` contains value `37` and `x2` contains `13`. The constant propagation pass of CompCert performs that analysis, and then simplifies expressions using the known values. The result, on the right, has removed the addition entirely.

To perform the static analyses needed by optimizing passes such as Constant Propagation, Common Subexpression Elimination and Dead Code Elimination, and even some lowering passes like Linearization, CompCert includes a module implementing the Kildall algorithm [Kildall 1973]. This algorithm describes a fixpoint solver for dataflow inequations, and can be used to define various dataflow intraprocedural analyses, like a liveness analysis (for each program point, deciding what registers contain values that may be needed in the rest of the execution), or a value analysis (approximating for each program point the set of possible values a register can hold). This module, that we reuse in our JIT mechanization, comes with Coq correctness properties that help prove correct the code transformations using static analyses defined with it.

Another standard compiler optimization performed by CompCert is inlining. As described in Section 2.1.2, inlining consists in replacing some function calls with the code of the called function itself, copying the instructions.

3.1.2 The CompCert Theorem

CompCert comes with a Coq proof of correctness. This proof states that compilation did not introduce any bugs in the program. To do so, CompCert first defines semantics for CompCert C, and each of the target assembly languages. The theorem is a semantic preservation property that compares these semantics. To reason modularly about each pass, CompCert also defines the semantics of each intermediate language of Figure 3.1, and include theorems (as seen later in Figure 3.9 for instance) to compose the correctness arguments of each pass.

Small-Step Operational Semantics The semantics of languages in CompCert are defined with small-step operational semantics, also called transition semantics. Small-step semantics are described with a type representing semantic states, and containing the current state of execution (for instance the value assigned to each register, the state of the memory and the current instruction pointer), along with a small-step transition relation. This relation (noted $\longrightarrow$) represents elementary steps of the program execution. It relates every semantic state to its possible next semantic states, and each step is associated with a trace of observable I/O events (*e.g.*, calls to external library functions). CompCert also defines transitives closures star $\longrightarrow^*$ (any number of steps) and plus $\longrightarrow^+$ (a strictly positive number of steps) from the generic $\longrightarrow$.

CompCert includes a generic notion of small-step semantics shared across all its languages. A simplified version of it is shown in Figure 3.3, a Coq record containing everything to define the behaviors of a program. The type `state` represents semantic states, and `step` represents the small-step transition relation $\longrightarrow$, where `step s1 t s2` holds when $s_1 \xrightarrow{t} s_2$, meaning that there is an elementary step from the semantic state s_1 to semantic state s_2, associated with the trace t of observable events. To define when the execution of a program starts and stops, some semantic states are noted as initial or final according to the predicates `initial_state` and `final_state`. Additional omitted fields of this record are more specific to the semantics of the C language (like a *global environment* containing available function definitions), and are not relevant to this work.

Program Behaviors CompCert then associates a set of possible behaviors to a program given its language small-step semantics. Behaviors represent sequences of observable events

```
Definition Semantics {state: Type}
                     (step: state → trace → state → Prop)
                     (initial_state: state → Prop)
                     (final_state: state → int → Prop) :=
  {|  state := state;
      step := step;
      initial_state := initial_state;
      final_state := final_state
   |}.
```

Figure 3.3 – Simplified small-step semantics in CompCert

that can be emitted by a program while following the small-step transition relation. Behaviors exist in three possible forms: *termination*, *divergence* and *going-wrong*. A terminating behavior is a sequence of semantic steps that starts on an initial state and ends on a final state, associated to the finite list of events observed. Diverging behaviors are similar except that the sequence of steps is infinite and the list of observed events can be either finite or infinite. Going-wrong behaviors are sequences of steps ending on a semantic state that does not have a successor according to the small-step transition relation.

The type of observable events in CompCert is shared by all IRs. This means that the behaviors of an assembly program can be compared to the behaviors of a CompCert C program for instance. Note that multiple behaviors can be associated to a single program, as small-step semantics can be nondeterministic, meaning that there can exist multiple different steps from the same semantic state. C (and CompCert C) programs are notably nondeterministic as they do not specify the evaluation order of expressions.

The CompCert Theorem The final CompCert correctness theorem relates behaviors of a CompCert C program to the behaviors of the corresponding compiled assembly program. Figure 3.4 contains two versions of that theorem. Here, the definitions `Asm.semantics tp` and `Csem.Semantics p` are small-step semantics records as in Figure 3.3, where transitions follow the code of their respective programs `t` and `tp`.

The first, more generic theorem can be read as follows: for any program `p`, if compilation succeeds and produces assembly program `tp`, then any behavior `beh` of `tp` improves a behavior `beh'` of `p`, written `behavior_improves beh' beh`. Behavior improvement (a form of *behavior refinement*) means that either the behaviors are equal, or if `p` goes wrong, `tp` is allowed to avoid going wrong and can instead have any behavior after the point where `p` went wrong (while still having the same observable trace of events up to this point). C compilers can produce and

```
Theorem transf_c_program_preservation:
  ∀ p tp beh,
  transf_c_program p = OK tp →
  program_behaves (Asm.semantics tp) beh →
  ∃ beh', program_behaves (Csem.semantics p) beh' ∧ behavior_improves beh' beh.

Corollary transf_c_program_is_refinement:
  ∀ p tp,
  transf_c_program p = OK tp →
  (∀ beh, program_behaves (Csem.semantics p) beh → not_wrong beh) →
  (∀ beh, program_behaves (Asm.semantics tp) beh →
                                          program_behaves (Csem.semantics p) beh).
```

Figure 3.4 – CompCert Correctness Theorems

optimize code with the assumption that the source C code always has a behavior according to the C99 standard. Some realistic C programs however have an *undefined behavior* according to the standard, but programmers still expect them to be compiled in a certain way. With behavior improvement in its specification, CompCert can compile these programs as expected.

The second theorem is a corollary. Since behavior improvement allows different behaviors only when the source program `p` contains going-wrong behaviors, we deduce that if `p` has a *safe* behavior (not going-wrong) then any behavior `beh` of the compiled program is also a behavior of `p`. This is often the most interesting case, where we can be ensured that the assembly program behavior is exactly a behavior of the source program. One can use static analyzers like Verasco [Jourdan et al. 2015] to prove that a C program has safe behaviors.

Note that in both theorems, the fact that `p` has been compiled to `tp` is expressed with the hypothesis `transf_c_program p = OK tp`. This `transf_c_program` function could also return an error, instead of an assembly program. This means that the CompCert theorems allow the compiler to fail to produce a program, for instance if a validator rejects a translation. The theorems are vacuously true in that case. For ahead of time compilation, this is not an issue, as compilation and execution are typically independent.

3.2 The Simulation Framework of CompCert

In order to prove such theorems about program behaviors, CompCert uses a *simulation* methodology. Simulations define a relation between two programs (a *source* and a *target* program) and their small-step semantics. Simulations are used to prove the correctness of passes

that transform the source program into the target program. Simulations come in two forms in CompCert: *forward* and *backward* simulations. Intuitively, a backward simulation states that any target program behavior can be matched locally with a similar behavior in the source program. And a forward simulation states that any source program behavior can be matched locally with a similar behavior in the target program.

Backward Simulations The standard technique to prove the correctness of a code transformation in formally verified C compilation is to prove a backward simulation, as shown in Figure 3.5. Given a source program P_1 (in a language L_1) and its transformed target program P_2 (in a language L_2), each step in P_2 with a trace t must correspond to transitions in P_1 with the same trace t and preserve as an invariant a relation $\approx$ between semantic states of P_1 and P_2. As seen on the right of the diagram, it is possible to match a step of the target program P_2 with no step at all in the source program P_1, this is called a *stuttering step*. This is useful if the transformation increases the number of steps in the code, for instance by adding instructions, or splitting a high-level instruction into several low-level ones. However, a backward simulation should rule out the case of infinite stuttering, where infinitely many consecutive steps in P_2 are simulated by no step at all in P_1. To avoid this case, the theorem uses a measure over the states of L_2 that strictly decreases in cases where stuttering happens. [2] It is generically noted $m(\cdot)$ and is specific to each compiler pass. It can be defined using natural numbers, or any other type with a well-founded order. To prove a backward simulation, one must also prove that the transformation preserves *progress*, as seen on Figure 3.6. Progress preservation means that when two semantic states are matched with the invariant $s_1 \approx s_2$ and there exists a step in P_1 starting from s_1, then there also exists a step in P_2 starting from s_2. In CompCert's parlance, both the diagram and progress preservation together are denoted by `backward_simulation (sem1 P1) (sem2 P2)`, where `semi` defines the semantics of L_i. As illustrated by many published proofs conducted within CompCert, the gist of proving a simulation for an optimization or lowering pass relies on designing a suitable matching relation $\approx$ and measure $m(\cdot)$, specific to this pass, then proving the preservation of the relation $\approx$ for each possible semantic step.

Backward simulations are useful in CompCert because they imply behavior preservation. Looking at the diagram of Figure 3.5, one can see how locally, every behavior of the target program can be matched to a behavior of the source program. In fact, the correctness theorem of Figure 3.4 is obtained with a backward simulation between a CompCert C program and its

2. Technically in CompCert, the measure is not defined on semantic states but instead each matching relation rule comes with an index on which the measure is defined. The version presented here simplifies the presentation of our invariants.

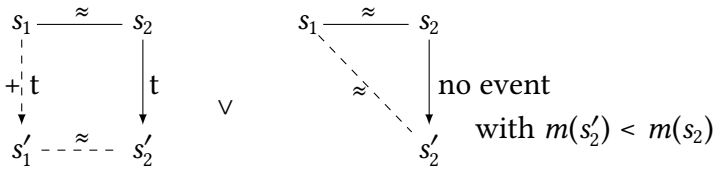

Figure 3.5 – A backward simulation diagram. Solid lines are hypotheses and dashed lines are conclusions. On the left of each diagram are the source program and its current state s_1; the target program and its current state s_2 are on the right. Horizontal lines represent the matching relation ≈, and vertical lines the semantic steps.

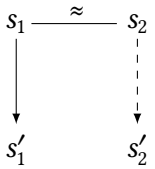

Figure 3.6 – Progress preservation in a backward simulation. Solid lines are hypotheses and dashed lines are conclusions.

compiled assembly program. A backward simulation can hold even if the code transformation removed some behaviors of the source program, as long as the remaining behaviors match the source ones, and that progress is preserved. Backward simulations can also be composed, and this allows to prove code transformations modularly.

Forward Simulations In general however, backward simulations are rather difficult to prove. One explanation is that one needs to reason in one direction to prove the diagram of Figure 3.5 (constructing steps of P_1 from P_2 steps), and in the other direction when proving the preservation of progress of Figure 3.6 (constructing steps of P_2 from P_1 steps).

For that reason, it is easier to prove a *forward simulation* between two semantics. To prove a forward simulation, it suffices to prove the diagram on Figure 3.7. Steps of the source program

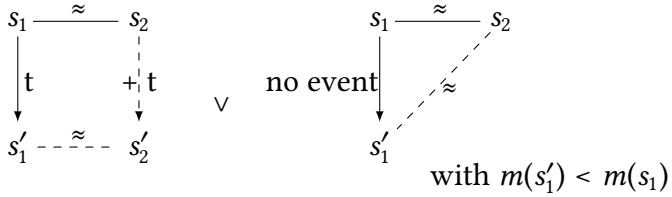

Figure 3.7 – A forward simulation diagram. Solid lines are hypotheses and dashed lines are conclusions. On the left of each diagram are the source program and its current state s_1; the target program and its current state s_2 are on the right. Horizontal lines represent the matching relation ≈, and vertical lines the semantic steps.

```
Lemma forward_to_backward_simulation:
  ∀ L1 L2,
  forward_simulation L1 L2 → receptive L1 → determinate L2 →
  backward_simulation L1 L2.
```

Figure 3.8 – CompCert forward-to-backward theorem

are matched with steps of the target program. Note that there is no need to prove progress preservation, as it is implied by the forward simulation diagram. This makes forward simulations easier to prove than backward ones.

However, a forward simulation on its own is not enough to imply a behavior preservation theorem like the one of CompCert. Just like a backward simulation can hold when removing behaviors from the source program, a forward simulation can hold when adding new behaviors to the target program. Even if a forward simulation holds, the target program could have additional behaviors that are not behaviors of the source program. However if the target program is deterministic, then either the source program is safe, in which case it is matched to the target behavior according to the forward simulation diagram because there exist a sequence of source steps; or the source program is not safe and has a going-wrong behavior (reaches a point from which there is no possible step), in which case the target program behavior improves the source one. Using this reasoning, CompCert proves and uses the *forward-to-backward* theorem of Figure 3.8. Given two small-step semantics `L1` and `L2`, if `L1` is *receptive* and `L2` is *determinate*, then a forward simulation implies a backward simulation between `L1` and `L2`. Receptiveness means that progress is independent of the external environment, which holds for the languages of CompCert. Determinacy is a weaker version of determinism. Small-step semantics are *determinate* when different semantic steps are possible from a semantic state only when they have different observable events. If L is determinate, then $s \xrightarrow{t} s_1 \wedge s \xrightarrow{t} s_2$ implies $s_1 = s_2$. The full determinacy definition used in CompCert also includes some additional properties, for instance that each small step emits at most one observable event, or the uniqueness of initial semantic states. These other properties hold for the semantics used in CompCert or the ones we define in this work.

As shown on Figure 3.9, forward simulations can be composed, and are then used extensively in CompCert. To avoid dealing with nondeterminism, CompCert starts with a backward simulation between CompCert C and Cstrategy, the same language where an evaluation strategy for expressions has been chosen. This can only be done with a backward simulation, as some source behaviors are removed. Next, all following passes are proved correct with forward

```
Lemma compose_forward_simulations:
  ∀ L1 L2 L3, forward_simulation L1 L2 → forward_simulation L2 L3 →
  forward_simulation L1 L3.
```

Figure 3.9 – Composing forward simulations

simulations. These forward simulations can be composed, and using the theorem of Figure 3.8, one can construct for free a backward simulation between Cstrategy and the compiled assembly program. The target assembly languages of CompCert are determinate. This backward simulation can be composed with the very first one, and the result is a backward simulation between CompCert C and assembly. This simulation implies the correctness theorems of Figure 3.4.

In conclusion, forward simulations are usually the simplest way of proving a compiler pass correct, as long as it can be used to construct a backward simulation using the theorem on Figure 3.8. Backward simulations are more generic and can hold even when removing behaviors of the source program, but require more proof efforts. In a language like C, nondeterminacy only arises in the order of evaluation of expressions, and CompCert uses Cstrategy to discard this issue as soon as possible.

In this work, we reuse multiple CompCert components to prove JITs correct in Coq. First, we reuse the simulation framework. This includes the small-step semantics definition of Figure 3.3, but also the backward and forward simulations presented in Figures 3.5 and 3.7, as well as some useful theorems related to them, such as the forward-to-backward theorem of Figure 3.8. For instance, we define small-step semantics for JITs in the following Chapter. Each code transformation done by our formally verified JIT is proved correct with a backward simulation. We also reuse the behavior library that associates behaviors to small-step semantics. In particular, this allows us to prove in Chapter 4 a behavior preservation property that resembles the correctness theorem of CompCert (Figure 3.4). We also reuse the Kildall module discussed in Section 3.1.1 to define various analyses in our JIT. In Chapter 5, we use it to define several dataflow analyses. In Section 5.2.3, we define a Constant Propagation pass that closely resembles the one of CompCert. Finally, the entire CompCert backend for the x86 architecture is reused in Chapter 7 to generate native code dynamically in a formally verified JIT. We produce RTL programs, and use the CompCert backend to generate equivalent x86 programs. We then reuse its proof scripts to prove the correctness of a JIT that generate native code.

Chapter 4

Designing Formally Verified JITs

In this Chapter, we present our approach to develop formally verified JITs. We first focus on building an architecture for JITs that clearly separates the role of each component. The architecture captures the essence of realistic JITs with speculation and native code generation. We then describe what the correctness theorem of a JIT should look like. Finally, we present our solution to the first JIT-specific verification challenge introduced in Section 1.2.1: the formal verification of a JIT with dynamic optimization. This results in a methodology to reduce the problem of verifying an entire JIT to proving the correctness of its optimizer, with a specification resembling those of static compilers.

4.1 A formalized JIT Architecture

The first step in formalizing and verifying JITs consists in identifying what constitutes a JIT. There is no standard architecture and no standard set of components across modern JITs, but there are similarities. Components of modern JITs fall into three main categories: *execution*, *optimization* and *monitoring*. Execution in a JIT can take multiple forms. First, many JITs use interpreters as a way to start executing code quickly. But as many JITs dynamically generate native code, another way to make progress in a JIT execution can be to call native code. Optimizations include speculative optimizations (see Section 2.1), standard optimizations (like constant propagation or inlining), or generating native code (which typically speeds up the next executions). Finally, JITs need mechanisms and components to orchestrate the interleaving of execution and optimization. This constitutes the monitoring category [1], from profilers that suggest optimizations and speculations, to the interfacing needed to interleave both levels of execution.

Our generic JIT architecture is shown in Figure 4.1. It contains all the previously mentioned categories. First, a monitor is in charge of gathering information and compilation hints about the program execution using a profiler, and regularly suggests functions to be optimized. This

1. Sometimes called *control*, for instance in JavaScriptCore [WebKit 2020].

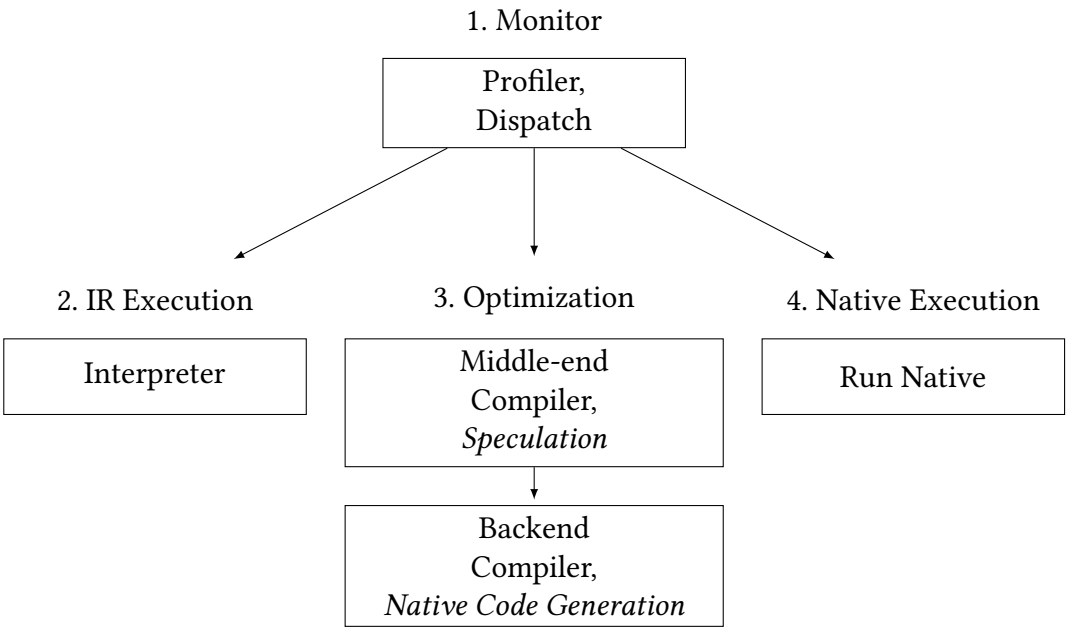

Figure 4.1 – Key components of our JIT design

monitor then calls the relevant JIT component among three possible ones: either optimization (3.), or dispatching execution to either the interpreter (2.) if the function to execute has not been compiled, or calling the corresponding native code (4.) if it has been. We follow a two-tiered optimizer design, where a first compiler called a *middle-end* compiler inserts speculation in the program and specializes the optimized function. The resulting optimized function (that may contain speculations) is then fed into a *backend* compiler, that generates native code. In our development, this backend compiler reuses the backend compiler of CompCert presented in Chapter 3.

To the best of our knowledge this resembles the architecture of modern JITs with a few deliberate exceptions, either design choices or simplifications, that we list here.

A simple common IR JITs typically have a common language that serves as the input of both interpretation and various compilation tiers. In our work, this IR is represented by a simple CFG representation, that can be directly interpreted. Some JITs rather interpret some bytecode, or even rewrite their code before interpreting it. This is the case for most WebAssembly interpreters like Wasm3 [Wasm3 2022] or the one in JavaScriptCore, as the structured control-flow of WebAssembly makes it difficult to interpret efficiently without rewriting [Titzer 2022]. Our IR, CoreIR, resembles CompCert's RTL and is fully presented in Chapter 5. Our design

```
Function Fun1 ():
  l1: i ← 1 l2
  l2: max ← 11 l3
  l3: b ← i < max l4
  l4: Cond b l5 l7
  l5: r ← Call Fun2 (i) l6
  l6: i ← i + 1 l3
  l7: Return r

Function Fun2 (x):
  l1: y ← x * x l2
  l2: Print y l3
  l3: Return y
```

Figure 4.2 – An example CoreIR program computing the 10 first squares

focuses on function-based JITs that compile entire functions of CoreIR. CoreIR is a simplified version of RTL (making easier the reuse of CompCert to generate native code in Chapter 7), augmented with speculative instructions. An example of a CoreIR program is given in Figure 4.2. Instructions are associated to labels and include the label of their successors. There are branches (`Cond`), function calls (`Call`) and output values (`Print`). The first function is a loop that calls `Fun2` with an argument ranging from 1 to 10, and `Fun2` prints the square of its argument. The observable behavior of a CoreIR program is defined by the sequence of values printed by the `Print` instructions, and that sequence of outputs must be preserved by the JIT. The IR used by modern JITs for realistic dynamic languages can be more complex. However, a simpler language allows us to focus on the JIT-specific verification challenges of Section 1.2.1, and could be extended with more features.

Two separate compilers In JITs with speculation, the separation between middle-end and backend is not always made as explicit as in our design. This design choice allows us to clearly separate the correctness arguments used for proving the correctness of speculation from the ones used for native code generation. This is reminiscent of how the Turbofan compiler of V8 starts compilation by specializing the code to compile using speculations [Meurer 2017]. As in Sourir [Flückiger et al. 2018], we believe that the logic involved in inserting and manipulating speculation should not be tied to implementation details and deserves its own formalization, especially with verification as a goal.

Our design only includes a single backend compiler to generate native code. Even though some JITs include several such compilers (see Section 2.2) to choose between more or less aggressive optimizations, we believe that this does not change the way JITs should be formally verified. The interaction between several execution engines is already showcased with both the interpreter and the execution of native code.

A monitor and its synchronization interface Some JITs may not have such a delimited notion of monitoring and dispatch, and instead directly implement them in the other components. For instance, the interpreter may contain code that directly calls the profiler and optimizer or directly jumps to native code on function calls. Backend compilers of JITs may generate native code that directly reconstructs the interpreter stackframe on deoptimization. In our design however, execution systematically returns to a monitor component at each *synchronization point*: function calls, function returns and deoptimization triggers. These points define a synchronization interface between the different JIT components, the moments in an execution where the JIT can switch between its various components. For JITs with on-stack replacement as an entry (as explained in Section 2.1.2), one would need to add other synchronization points in the middle of loops that may trigger it. Without this feature, a function-based JIT only needs to switch components when changing function (calls and returns), or when deoptimizing. We believe that this monitor component represents an abstraction layer that better separates the different mechanisms of a JIT execution and facilitates their formal verification. This allows to modularly verify the mechanism that chooses between components, independently from the verification of an interpreter, whose specification can simply consists in executing CoreIR code according to its semantics.

4.2 JITs as Coq State Machines

A JIT resembles an interaction loop in that it executes or optimizes the code until the program finishes. For that reason, JITs in Coq can be implemented as transitions on a state machine, to be looped until the end of program execution. Figure 4.3 shows the state machine that describes how the JIT alternates between its components, from IR execution, to optimization and native execution. To implement a JIT in Coq, we choose to define a function representing the transitions of this state machine and calling the relevant JIT component. For instance, one transition corresponds to the interpreter, and another transition corresponds to the optimizer. This transition function returns the next state of the state machine. From the state that corresponds to the monitor, this depends on what the profiler suggests for instance. Most transitions can be represented as terminating Gallina functions. Note however that some transitions called *impure* (like calling native code) cannot be represented in Gallina. Gallina is a pure functional language without mutable data-structures or the possibility to call arbitrary native code. This is discussed in Chapter 6, for now we consider that every transition is a terminating Gallina function.

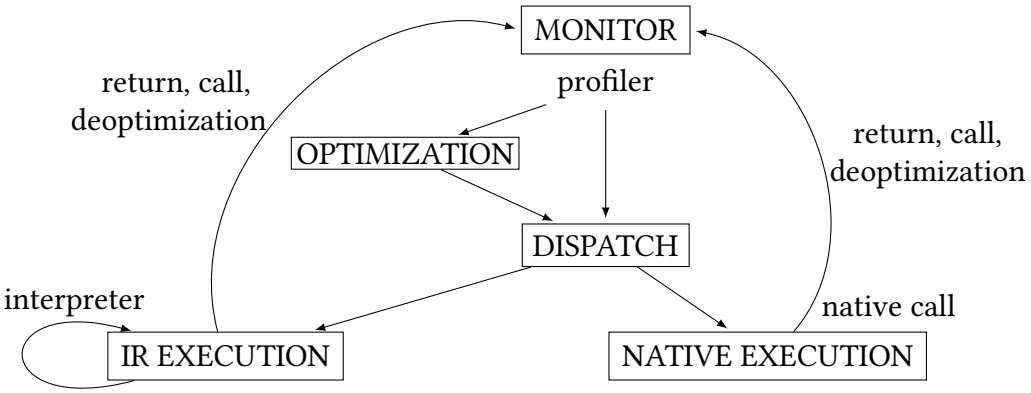

Figure 4.3 – A JIT architecture as a state machine

Defining a JIT as a Coq function encoding transitions of a state machine has several advantages. First, this is a simple way to allow the execution of possibly nonterminating programs. In Gallina, it can be difficult to write and reason on nonterminating functions. One must always prove the termination of recursive functions. While *corecursion* is a Coq feature allowing to write some nonterminating programs, reasoning on such definitions often implies writing coinductive proofs for which support is more limited than traditional structural induction [Hur et al. 2013]. This is not an issue for ahead of time compilers like CompCert: the entire compiler can be written as a terminating function, regardless of the compiled program's behaviors. However, JITs stay active during the entire program execution and may not terminate if the program they execute diverges. By breaking down the JIT behavior into a sequence of terminating steps (the transitions of the state machine), we can simply define the transition function in Coq, extract it to OCaml, and write a simple OCaml program that loops the transition until a final state is reached. This small piece of OCaml code diverges when the JIT execution diverges. The issue of nontermination when executing a diverging program with a JIT has been delegated to this small OCaml loop. While it is not verified in Coq, it is small and simple enough to manually audit.

For instance, one can define inductively in Coq a type `jit_state`, representing the different states of Figure 4.3, and also containing all the data used by the JIT (for instance, the CoreIR code). Defining JITs in Coq then amounts to defining a function representing its transitions, of type `jit_step: jit_state -> (trace * jit_state)`, where `trace` is the type of observable events emitted during a JIT step. The OCaml loop can then be written as on Figure 4.4, where `final_state` is another extracted function that returns `true` if the program execution has finished (when returning with an empty execution stack).

```
let rec jit_loop (js:jit_state) =
  if (final_state js) then
    printf "End of Execution"
  else
    let (t, next) = jit_step js in
    print_trace t;
    jit_loop next
```

Figure 4.4 – OCaml looping of extracted JIT transitions

Algorithm 1: The `jit_step` function

```
input : jit_state js = (jp, es, ps)
output: The output trace and the next jit_state
match es with
    case MONITOR synchro =>
        let newps = profiler ps synchro;
        match optim_policy newps with
            case OPT =>
              | return no_trace, (jp, OPTIMIZATION synchro, newps)
            case NO_OPT =>
              | return no_trace, (jp, DISPATCH synchro, newps)
        end
    case OPTIMIZATION synchro =>
        let newjp = optimizer ps jp;
        return no_trace, (newjp, DISPATCH synchro, ps)
    case DISPATCH synchro =>
        let is = create_interpreter_state synchro;
        return no_trace, (jp, IR_EXECUTION is, ps)
    case IR_EXECUTION is =>
        let (t, result) = interpreter is;
        match result with
            case synchro =>
              | return t, (jp, MONITOR synchro, ps)
            case new_is =>
              | return t, (jp, IR_EXECUTION new_is, ps)
        end
end
```

JIT step example For instance, Algorithm 1 shows a simplified version of such a `jit_step` function where there is no native code execution. In this version, JIT states are implemented with tuples `js = (jp, es, ps)` where `jp` is the current JIT program being executed and `es` is the current *execution state*. Execution states are a pair containing one state of the state machine of Figure 4.3, and either an interpreter state `is` when in the `IR_EXECUTION` state, or a synchronization point `synchro` otherwise (as defined earlier, a function call, a function return or a deoptimization). Finally, `ps` is the current profiler state, the data manipulated by the profiler.

While in the `MONITOR` state, the JIT calls the profiler, and decides to go to either the optimization or dispatch state depending on the suggestion of the profiler. The functions `profiler` and `optim_policy` are described in Section 4.3. Then, when in the `OPTIMIZATION` state, the JIT updates its current program by calling its optimizer. During the `DISPATCH` phase, the JIT constructs a state of the interpreter. If the current synchronization point `synchro` is a function call, the JIT creates an interpreter state corresponding to the entry of the called function. When this is a function return, the JIT looks up the top stackframe to know which function to return to. [2] When this synchronization point is a deoptimization, the JIT reconstructs an interpreter state corresponding to the execution of the original version of the function being executed. This reconstruction follows the rules described in the semantics of CoreIR, in Section 5.1.2. Note that on this simplified example, the dispatch always constructs an interpreter state as there is no native code execution. Native code execution requires a different kind of `jit_step` function that is described in Chapter 6. Finally, during interpretation, an `interpreter` function produces an observable trace and its next state. If a synchronization point has been reached, the JIT execution goes back to the monitor. Otherwise, the JIT stays in the `IR_EXECUTION` state.

JIT semantics Another advantage of defining JITs as transitions of a state machine is that it allows for a simple definition of small-step semantics for the JIT execution. Semantic states are exactly the type `jit_state`, and the step transition relation ($\longrightarrow$) corresponds to executing the `jit_step` function. Figure 4.5 shows a simple rule that shows how the `jit_step` function can be turned into a small-step transition relation with a single rule. In the rest of this chapter, we write `jit_sem p` for the small-step semantics defined by the small-step transition relation of Figure 4.5 and whose initial state contains the code of CoreIR program `p`.

2. In this simple example with no native code execution, the JIT execution stack can be part of the interpreter states and synchronization points. Later in Chapter 6, the stack is handled differently when it contains both interpreter and native code stackframes.

$$\texttt{jit\_sem}\ \frac{\texttt{jit\_step}\ js_1\ =\ (t, js_2)}{js_1 \xrightarrow{t} js_2}$$

Figure 4.5 – The JIT small-step semantic rule

4.3 Profiling as External Heuristics

To decide what and when to optimize, modern JITs use sometimes complex, empirical and language-dependent heuristics, as discussed in Section 2.2. Instead of trying to verify such mechanisms, we argue that what the profiler suggests should only impact the performance of the JIT, but not its correctness. In other words, a JIT execution should be correct even if the profiler suggests optimizing functions that will never be called, or suggests speculations that will never hold.

The task of verifying a JIT should not hinder itself with the verification of empirical heuristics. For this reason, we define the profiler heuristics in our Coq development as a collection of external parameters. This means that the JIT proof of correctness assumes nothing more than the existence of such parameters. During extraction, these external parameters can be implemented in OCaml. This allows us to easily change heuristics without even modifying the proofs.

```
Parameter profiler_state: Type.
Parameter initial_profiler_state: profiler_state.
Parameter profiler: profiler_state → synchro_point → profiler_state.
Parameter optim_policy: profiler_state → jit_status.
```

Figure 4.6 – External Parameters for Profiling

In Coq, our profiler is defined using a simple interface of parameters, shown in Figure 4.6. A parameter `profiler_state` type is defined. The `jit_state` type always holds the current profiler state. This profiler state contains the data used by the profiler, and can be freely implemented. To help define the initial state of the JIT, we require an initial profiler state. Every time execution comes back to the monitor on a synchronization point (call, return or deoptimization), the profiler updates its internal state with the `profiler` parameter. Finally, the profiler can suggest optimizations with `optim_policy`. Its return type, `jit_status`, either recommends no optimization, and the monitor then proceeds with execution, or contains a list of suggested optimizations. This includes what to speculate on, and the identifiers of the functions to optimize

and compile.

For instance, `optim_policy` may return `OPT [(F3,InsertAssume l5 (x1=x8)); (F3,Backend)]` if it wishes to optimize function `F3` by inserting an `Assume` instruction speculating that register `x1` is equal to `x8` at label `l5` (see Section 5.2.2), then calling the backend compiler to generate native code for that function.

Note that with this methodology, the JIT cannot always do what the profiler recommends. For instance, if the profiler always suggests optimizing and never executing, the JIT could get stuck in an optimization loop. The monitor is in charge of deciding when to follow the profiler recommendations. This is detailed in section 4.5. The benefits of clearly separating these parameters from the rest of our JIT development instead of implementing them in Coq is that our correctness results do not depend on the implementation of such heuristics, which should only impact performance, but not correctness.

4.4 A JIT Correctness Theorem

CompCert simulations state that any behavior of a compiled program matches at least one behavior of its source program. While the program of a JIT dynamically evolves during execution, one can still prove a similar correctness theorem if we compare the semantics of the input CoreIR program to the small-step semantics `jit_sem` describing the behavior of the JIT executing a program. Just like the theorem of Figure 3.4 is proved with a backward simulation in CompCert, we can reuse the CompCert simulation framework to prove JIT correctness.

Backward Simulation **Mechanized** in the POPL21 artefact

Backward simulation relating the behavior of the JIT executing a program `p` (including execution, profiling and dynamic optimizations) to the behavior of the CoreIR semantics of `p`.

```
Theorem jit_simulation:
  ∀ (p: program),
    backward_simulation (CoreIR_sem p) (jit_sem p).
```

Simulation 1 – JIT backward simulation (pure version)

We first prove the Backward Simulation 1, where `CoreIR_sem` corresponds to the small-step semantics of CoreIR. As defined in Figure 4.5, this `jit_sem` semantics contains every transition

of the JIT, including interpretation or optimization. Note that so far, we have assumed that every transition of the JIT was pure. This can be true for a JIT that does not generate and execute native code and instead only interprets CoreIR, like the one we first developed for our POPL21 article [Barrière et al. 2021]. However, for JITs with impure state machine transitions, we must modify the `jit_sem` definition and Simulation 1 accordingly. This is done later in Chapter 6, Simulation 11.

Every simulation described in our methodology for the formal verification of JITs is represented with such a box, with a short description and a simplified version of the Coq theorem present in our development. These Simulations are all listed on page 15. In the electronic version of this document, the **Mechanized** keyword in the top right of the box contains a link to the corresponding Coq proof. Like other definitions, these links are also available in the appendix, on page 153.

After proving the simulation, Figure 4.7 then corresponds to the final correctness theorem of our Coq JIT. This theorem is obtained using existing CompCert proofs that establish a behavior preservation theorem from a backward simulation, meaning that proving Simulation 1 constitutes our main verification work. This `jit_correctness` theorem strongly resembles the one from CompCert (Figure 3.4), only replacing the semantics of the compiled program with the semantics `jit_sem`. The definitions `program_behaves` and `behavior_improves` are directly imported from CompCert, as presented in Chapter 3.

Note that the JIT semantics are not allowed to go wrong if the program `p` does not go wrong. In particular, if the compilation of a function fails (analyses defined with the Kildall CompCert module are allowed to fail if the analysis does not converge for instance), then the JIT must cancel the optimization and keep interpreting safely. While the entire compiler is allowed to fail in the case of ahead of time compilation, failure to dynamically generate code in a JIT should not crash the entire execution.

Having a correctness theorem so close to CompCert's has several advantages. First, we avoid reproving existing results of formally verified compilation that apply to JITs. For instance, the entire simulation framework of CompCert can be reused, including the small-step semantics and simulation definitions, but also the proof that constructs a behavior preservation property from a backward simulation. Second, by reusing backward simulations, we facilitate the reuse of the CompCert backend as a backend compiler for our JIT in Chapter 7.

```
Theorem jit_correctness:
  ∀ p beh, program_behaves (jit_sem p) beh →
    ∃ beh', program_behaves (CoreIR_sem p) beh' ∧ behavior_improves beh' beh.
```

Figure 4.7 – JIT Correctness Theorem

4.5 Dynamic Optimizations

As established in the previous section, a good way to prove a JIT correct is to prove a backward simulation between the CoreIR small-step semantics of the program to execute, and the small-step semantics of the JIT executing that program. This ensures behavior preservation and thus that the JIT did not introduce any bugs while executing its program. However, this simulation cannot be built in the same way as in a static compiler like CompCert. In all the semantics defined by CompCert, the program being executed is a parameter that does not change during execution, while in a JIT, the program being executed changes at every optimization step. In the CompCert theorem of Figure 3.4, both `p`, the source program, and `tp`, the compiled program, are known. Before execution however, there is no way to predict what code will be executed in the JIT, as it is dependent on what optimizations are done and when.

In this section, we tackle the first JIT-specific verification challenge introduced in Section 1.2.1: proving the correctness of a JIT with dynamic optimizations. The main challenge is to define the invariant for Simulation 1, such that it defines an appropriate specification for the optimizer of the JIT.

4.5.1 The Nested Simulations Technique

In a JIT, the program being executed can be seen as some data manipulated by the optimizer. As such, our `jit_state` definition contains the current program of the JIT. When defining `jit_step`, the optimizer transition is the only one modifying that part of the JIT semantic states, for instance when creating a new optimized version of a function. For this section, we extend the JIT semantic state `jit_state` to be a tuple `js = (jp, es, n, ps)`, where `jp` is the current JIT program being executed. `es` is the current execution state, containing either a synchronization point (call, return or deoptimization), or the state of an execution engine (for instance, an interpreter state). States of execution engines contain the current mapping of the registers, the contents of the memory and the execution stack. Then, `n` is a bound on the number of optimizations. It decreases each time the JIT calls the optimizer. This allows to avoid the infinitely stuttering case where the profiler always suggests optimizing. When it reaches

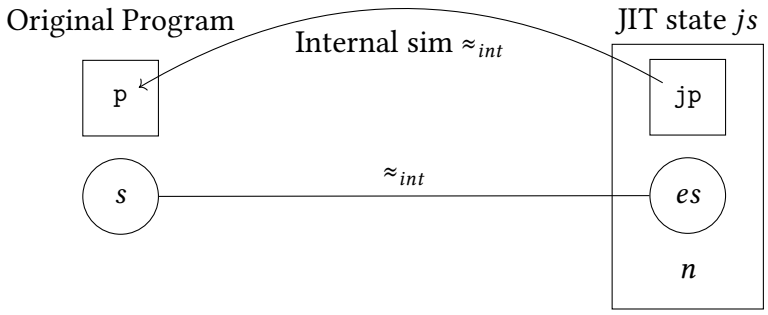

Figure 4.8 – External simulation relation $s \approx_{ext} js$ iff 1) there exists an (internal) backward simulation between `jp` and `p` using relation $\approx_{int}$ and 2) s is matched with es using $\approx_{int}$.

0, the monitor discards the profiler suggestions and simply stops calling the optimizer. Finally, `ps` is the current profiler state, and gets updated when calling the profiler. As this profiler state is not relevant to the proof, we omit it from most figures of this section.

To prove Simulation 1 (called *external* in this section), one needs to define a simulation invariant. Such an invariant must be adapted to our JIT setting, relating semantic states of CoreIR to JIT semantic states of type `jit_state`. Intuitively, this invariant should state that at any moment during the JIT execution, the optimizer, while having modified the program, did not change its behavior. Or in other words, that the current program of the JIT `jp` is in some way equivalent to the original program `p` fed to the JIT. Intuitively, this equivalence comes from the fact that `jp` has been obtained by gradually transforming `p` during the optimization steps of the JIT. To express this equivalence, we choose to use yet another simulation relation (called *internal*), between the current JIT program `jp` and the original program `p`. In short, we prove an external backward simulation whose invariant contains another backward simulation, which we call the internal simulation. As the JIT program gets optimized during JIT execution, the internal simulation changes to reflect that. This is our *nested simulations* technique to solve the JIT-specific issue of dynamic optimizations.

Figure 4.8 shows a graphical representation of this invariant, used later in Figure 4.10 to illustrate the proof. In the figure, (Internal sim $\approx_{int}$) means that there exists an internal backward simulation using internal invariant $\approx_{int}$ between the two programs pointed by the arrow.

We show a (simplified) definition of the invariant $\approx_{ext}$ of the external simulation on Figure 4.9.[3] The relation $\approx_{ext}$ is the simulation invariant of the JIT execution, relating CoreIR

3. The full mechanized version also includes some additional properties that we omit here for brevity. For instance, the measure should correspond to a well-founded order, and the invariant relation should match final semantic states. These properties are identical in the CompCert simulations and in our internal ones.

$$\text{internal-sim}\ \frac{\begin{array}{c}(1)\quad \forall s,\ \text{if } s \text{ is a function call synchronization state then } s \approx_{int} s \\ (2)\quad \textbf{Progress Preservation} \text{ (Figure 3.6):} \\ \forall s_1\ s_1'\ t\ s_2,\ s_1 \approx_{int} s_2 \wedge s_1 \xrightarrow[p]{t} s_1' \text{ implies} \\ \exists\ t'\ s_2',\ s_2 \xrightarrow[jp]{t'} s_2' \\ (3)\quad \textbf{Simulation Diagram} \text{ (Figure 3.5):} \\ \forall s_1\ s_2\ s_2'\ t,\ s_1 \approx_{int} s_2 \wedge s_2 \xrightarrow[jp]{t} s_2' \text{ implies} \\ (\exists\ s_1',\ s_1 \xrightarrow[p]{t}{}^{+} s_1' \wedge s_1' \approx_{int} s_2') \text{ or } (s_1 \approx_{int} s_2' \wedge m_{int}(s_2') < m_{int}(s_2) \wedge t = \emptyset)\end{array}}{\texttt{backward\_internal\_simulation} \approx_{int} m_{int}\ p\ jp}$$

$$\text{external}\ \frac{s \approx_{int} es \qquad \texttt{backward\_internal\_simulation} \approx_{int} m_{int}\ p\ jp}{s \approx_{ext} (jp, es, n, ps)}$$

Figure 4.9 – Defining the external simulation relation $\approx_{ext}$

semantics states s of the original program p to JIT states (jp, es, n, ps). According to the (external) rule, $s \approx_{ext} (jp, es, n, ps)$ when there exists an internal simulation between p and jp using some invariant $\approx_{int}$, and $s \approx_{int} es$. We use rule (internal-sim) to define our internal backward simulations. This rule has three preconditions, numbered (1) to (3). These simulations resemble the backward simulations of CompCert, with a small difference we explain below. With this rule, we define $\texttt{backward\_internal\_simulation} \approx_{int} m_{int}\ p\ jp$, meaning that programs p and jp are simulated using internal invariant $\approx_{int}$ and measure m_{int}. The invariant $\approx_{int}$ relates semantic states of p to semantic states of jp. It depends on what transformations have been done so far by the JIT optimizer. We use the notation $s_1 \xrightarrow[p]{t} s_1'$ to mean that there exists a step in the semantics of p between semantic states s_1 and s_1' with an observable trace t. Condition (2) corresponds to the progress preservation property of Figure 3.6; the existence of a semantic step in the p should imply the existence of a step in jp. Similarly, condition (3) corresponds to the simulation diagram of Figure 3.5; semantic steps in jp should either be stuttering steps with a decreasing measure, or should be matched with similar steps in p. In the case of a stuttering step in the simulation diagram, there should not be any observable events, and we use $\emptyset$ to denote the empty trace.

Note that the internal simulation relates the semantics of two known programs, p and jp. In that simulation, the semantics of jp are defined without any dynamic optimizations. If

jp contains only CoreIR code, then it is simply the CoreIR semantics. If the JIT has produced native code, we define mixed semantics in Chapter 7 that interleave semantics of CoreIR being interpreted with the semantics of the native code being executed.

The definition of backward internal simulations of Figure 4.9 is almost identical to the backward simulation definition used in CompCert. For instance, assumptions (2) and (3) are exactly the backward simulation diagram and progress preservation definitions presented in Section 3.2. The only change appears when initializing the simulation invariant, in condition (1). In the CompCert definition, the invariant is required to relate initial states of the simulated semantics. In a dynamic setting however, the JIT execution steps into its new program from the state where it called its optimizer. As a result, we require the invariant to relate any synchronization state where an optimization can happen. In our implementation of the monitor, optimizations are only possible at function calls synchronization states.

4.5.2 Proving the External Backward Simulation

As a result, if a JIT step consists in executing (for instance calling the interpreter) to update the current execution state *es*, we can use the internal simulation to deduce that this behavior matches some behavior from *s* of `p`, and that the new execution state es' is also matched with the same simulation relation $\approx_{int}$ to a new semantic state s' of `p`. This follows from the backward simulation diagram (condition (3) of Figure 4.9) of the internal simulation. This case is depicted on the top left of Figure 4.10, where the dashed lines are deduced from the invariant, and interpreter correctness is required to exhibit state s'. Note that, according to the backward simulation diagram, the step from *es* to es' could be matched with a stuttering step in the original program, as seen on the top right of the Figure. In that case, we use the decreasing measure of m_{int} of the internal simulation to build a decreasing measure for $\approx_{ext}$ (see the paragraph below).

If the JIT step is an optimizing one, calling the dynamic optimizer to update the current JIT program `jp`, then we prove that this corresponds to a stuttering step in our external backward simulation. Since in that step the JIT simply optimized without executing anything, it makes sense that this corresponds to no progress at all in the source semantics. Proving preservation of the $\approx_{ext}$ invariant amounts to proving that the new program `jp'` is also related to `p` with an internal backward simulation, as seen on the bottom of Figure 4.10. Since backward simulations compose, and `jp` is simulated with `p` from the invariant, it suffices to show that the dynamic optimizer will produce a program `jp'` that is itself simulated with `jp` with an internal backward simulation. We compose this new simulation with the previous one (using invari-

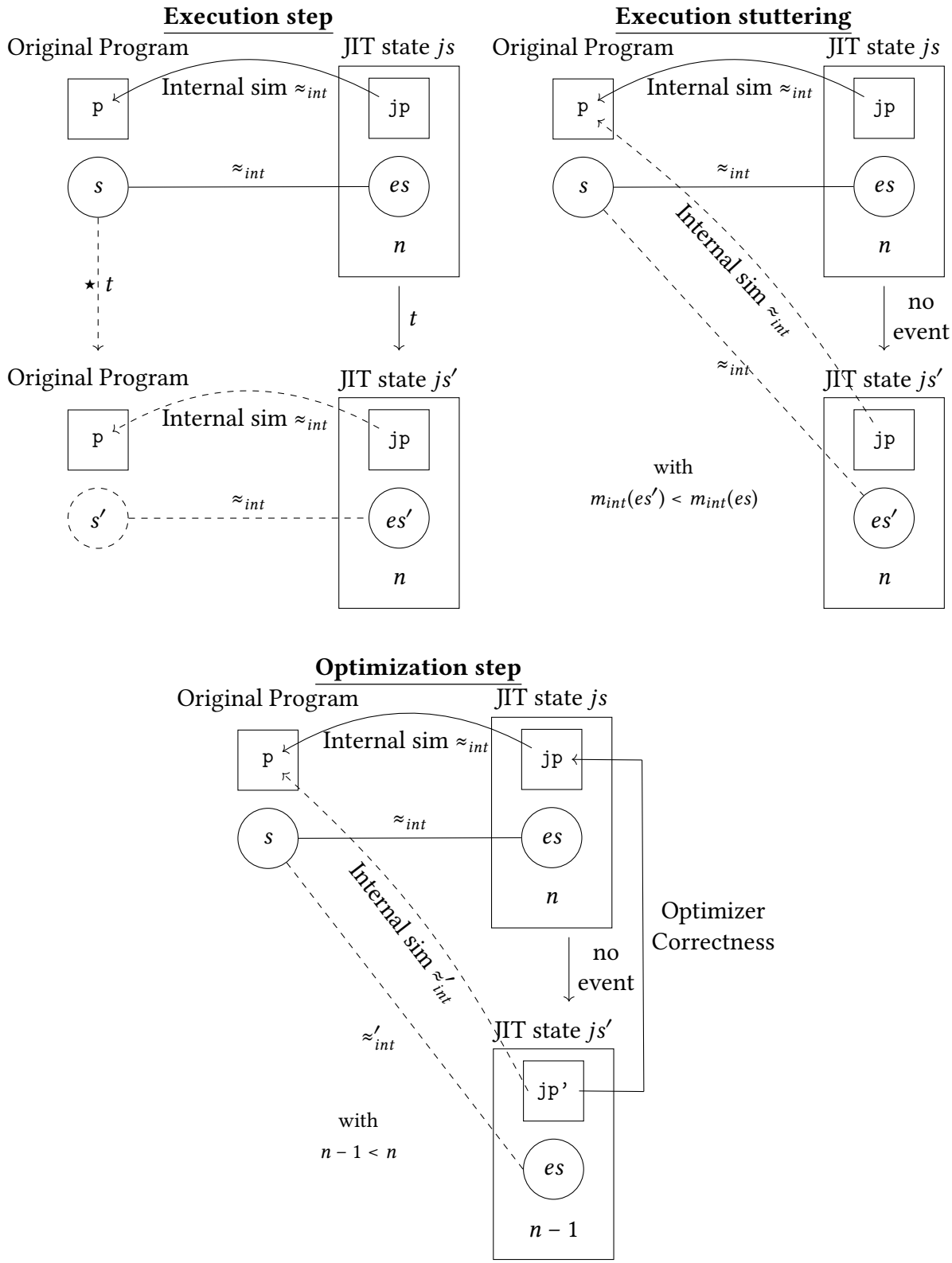

Figure 4.10 – The external simulation diagram. Solid lines are hypothesis and dashed lines are conclusions. Profiler states ps are omitted.

ant relation $\approx_{int}$) and get a new internal backward simulation (with an invariant relation $\approx'_{int}$) relating `jp'` to `p`. We use condition (1) of Figure 4.9 to prove that es is still matched with s with the new internal invariant $\approx'_{int}$.

Measures and Stuttering Steps Stuttering in our external simulation can happen in two cases: during optimization steps and when execution is matched with stuttering. Informally, we know that in the first case, the number n in our JIT states decreases, so we can use it to build a decreasing order. In the second case, we know that the current internal simulation comes with its own measure $m_{int}(\cdot)$ defined on execution states that decreases (according to the simulation diagram of Figure 3.5). We define here a measure $m_{ext}(\cdot)$ defined on JIT states for our simulation that strictly decreases on both cases of stuttering, ensuring that the JIT execution cannot get stuck. We define the following measure: $m_{ext}(jp, es, n, ps) = (n, m_{int}(es))$, and compare its elements with a lexicographic order. In our implementations, we must use dependent types to represent the invariant of the external simulation, as the type of $m_{int}(\cdot)$ may change as we compose internal simulations.

Progress Preservation As shown in Figure 3.6, part of proving a backward simulation consists in proving that the target semantics preserves execution progress. For execution steps, progress preservation directly comes from the progress preservation of the internal simulation (condition (2) of Figure 4.9). For optimizing steps however, we need to prove that calling the optimizer never fails. Some optimization passes in our compilers can fail (for instance, trying to insert a speculation at a label that does not exists, or trying to speculate on a unknown register value), but in these cases, the JIT detects the failure, cancels the optimization and keeps its current program.

4.5.3 A JIT Optimizer Specification

This nested simulations proof is at the core of the verification of our JIT design. This technique has several advantages. First, it successfully reuses the simulation framework of CompCert. Our backward internal simulations are almost identical to the ones used in CompCert, except for condition (1), which has been straightforward to prove for the code transformations we present in this work.

Second, it also helps us define clear specifications of JIT components. For instance, to reuse the internal simulation when using the interpreter, we must prove that any step of the interpreter corresponds to a step in the CoreIR semantics. More importantly, we see in the bottom

Backward Simulation **Mechanized**

Using the nested simulations technique, to prove the correctness of a JIT with dynamic optimizations, it suffices to show that the optimizer is proved correct with an internal backward simulation. This theorem relates the semantics of a program before and after optimization.

```
Theorem optimizer_correct:
    ∀ jp ps jp',
    optimizer ps jp = jp' ⟶
    backward_internal_simulation jp jp'.
```

Simulation 2 – Dynamic optimizer correctness in a JIT

of Figure 4.10 that proving our dynamic optimizer correct amounts to proving an internal backward simulation. This optimizer specification is shown on Simulation 2, where `ps` is any profiler state. This specification is designed to bear a striking resemblance with the CompCert correctness theorem, allowing us to reuse most of the methodology used to prove static optimization passes. In this theorem, the JIT-specific issue of dynamic optimization has been removed as the backward simulation is proved using semantics without optimizations. Since backward simulations compose, we can use this specification for both the middle-end and backend compilers of our JIT. In conclusion, to prove the final JIT correctness theorem of Figure 4.7, we prove Simulation 1. This simulation includes semantics that contain dynamic optimizations. Thanks to the nested simulation technique, we know that Simulation 2 implies Simulation 1, and Simulation 2 resembles the specification of a formally verified static compiler.

Chapter 5 shows how we prove such a backward simulation for a middle-end compiler with speculation, and Chapter 7 shows how we can prove such a backward simulation for a backend compiler reusing the CompCert backend. As explained in Section 4.4, the approach presented here has limitations: it requires that every JIT transition is a pure and terminating Gallina function. In Chapter 6, we extend the formalism presented in this section to handle impure and nonterminating JIT transitions.

Chapter 5

Formally Verified Speculative Optimizations

As seen in Chapter 2, modern JITs use speculation. This allows to focus the compilation efforts only on the most likely path that has been predicted. Despite being used in many JITs, speculation is often handled with ad hoc and language-specific implementations that can hide the already complex logic of specialization and deoptimization. We argue that designing JITs would benefit from a formal abstraction of speculation and deoptimization, and that such an abstraction is a necessary step towards the formal verification of JITs.

In this Chapter, we present a new high-level intermediate representation, CoreIR, that makes deoptimization and speculation explicit and separates the insertion of deoptimization points from the subsequent speculation checks. CoreIR is inspired by CompCert's RTL [Leroy 2009a] and Sourir [Flückiger et al. 2018], another IR with support for speculation.

Insertion and manipulation of speculation is then defined as CoreIR transformations, that constitute the core of the middle-end compiler of our JIT design. This approach to speculation follows the architecture of the Ř JIT [Flückiger et al. 2019], where R programs are JIT compiled by IR rewriting. In our work, such IR transformations can be proved correct with simulations, just like the passes of CompCert.

At its core, reasoning about speculation in a program requires reasoning about several behaviors: when the speculation holds and when it does not. As a result, nondeterministic semantics are particularly adapted to represent the behavior of speculation points. However, nondeterminism typically prevents the reuse of the forward-to-backward methodology of CompCert (Figure 3.8). In our proofs, we show how we can still reuse this technique for standard passes like constant propagation that preserve the nondeterminism of speculation.

5.1 CoreIR, an IR with Speculation

In order to formally verify the logic behind speculation used in modern JITs, we choose to use a language with *speculative instructions*: instructions that are designed to represent speculation. This representation allows us to insert speculation points anywhere in a function. Having dedicated instructions is reminiscent of the `speculate` instruction of JavaScriptCore [WebKit 2020], the `assume` instruction of Sourir [Flückiger et al. 2018], the `OSRPoint` instruction of JikesRVM [Soman and Krintz 2006], the *checkpoint*, *framestate* and *deoptimizeIf* nodes of V8 [V8 2022] or the *guard* and *FrameState* nodes of Graal [Duboscq et al. 2013].

In our design, we use two separate speculative instructions that represent different aspects of speculation and are presented in details below. We find that this separation makes the formal verification of speculation more modular. In short, one instruction `Assume`, contains the dynamic check that must be evaluated to see if a speculation holds, while the other, `Anchor`, expresses the synchronization to the version one must deoptimize to if the speculation fails. Some of the designs cited above use a third instruction to contain the *deoptimization metadata* (see section 2.1.2). As explained below, in our language the metadata is contained in both `Anchor` and `Assume` instructions.

In CoreIR, functions can have up to two versions: the original one, `Base`, and an optimized one, `Opt` where speculation may have been inserted. In our JIT design, the `Base` version always contains the original version of the function, and is never modified. When beginning the execution, each function only has this single `Base` version. But as the middle-end compiler gets called by the JIT, it adds new speculation in the function. It does so by creating, specializing and optimizing the `Opt` version.

Two CoreIR instructions are related to speculation, `Anchor` and `Assume`, and can only be inserted in the `Opt` versions. The `Anchor` instruction represents a potential deoptimization point, *i.e.*, a location in an `Opt` version where the correspondence with its `Base` version is known to the compiler and thus deoptimization can occur. For instance, in `Anchor F.l [r ← r+1]` the target `F.l` specifies the function (`F`) and label (`l`) to jump to, the mapping `[r ← r+1]`, called a *varmap*, describes how to create the state of the baseline version from the optimized state. Namely, how to reconstruct the value of each register that is live at the target instruction label. As we will see later, this mapping becomes more elaborate with inlining as multiple stackframes must be synthesized, resulting in complex proof invariants.

Anchors are inserted first in the optimization pipeline, when creating the `Opt` version and before any optimizations are done to this new version. Choosing where to insert them is impor-

tant as they determine where speculation *can* happen. Speculation itself is performed by inserting `Assume` instructions. To illustrate this second instruction, consider `Assume x=1 F.l1 [r ← r'+1]` which expresses the expectation that register `x` has value `1`. When the instruction is executed, if `x` has any other value, we say that the assumption *failed* and deoptimization is triggered. If this occurs, the currently executing function is discarded and control is transferred to the `Base` version of `F` at label `l1`, furthermore register `r` in that function is given the value `r'+1` where `r'` is evaluated in the `Opt` environment. Unlike anchors, assumes can be inserted at any time by the middle-end compiler once the `Opt` version has been created. To add an `Assume`, the profiler suggests what to speculate on and the location of an existing `Anchor`, and the middle-end compiler can insert the `Assume` next to it (possibly skipping some instructions, as shown in Section 5.2.5). If there is no anchor, then the assumption cannot be made and the insertion is canceled.

Illustrative Example All examples in this Chapter use a simplified concrete syntax for CoreIR. We omit labels when they refer to the next line and assume versions to start with the instruction on the first line. Moreover, we use the shorthand '`Anchor F.l1 [a,b]`' to represent the varmap '`Anchor F.l1 [a ← a, b ← b]`', and '`Assume ... ⚓l1`' to denote an `Assume` instruction with identical metadata as the `Anchor` instruction at label `l1`.

```
Function F(x,y,z):
Version Base:
      d ← 1
 l1:  a ← x*y
      Cond (z == 7) l2 l3
 l2:  b ← x*x
      c ← y*y
      Return b+c+d
 l3:  Return a
```

(a) Baseline

```
Function F(x,y,z):
Version Base:
      ...
Version Opt:
 l4:  Anchor F.l1 [x,y,z,d←1]
      c ← y*y
      Assume [z=7, x=75] ⚓l4
      Return 5626+c
```

(b) Optimized

Figure 5.1 – Example of speculation

Figure 5.1 shows an example of how a middle-end compiler could manipulate CoreIR. Assume that, for the program in Figure 5.1(a), a profiler detected that at label `l2` of function `F` registers `z` and `x` always have values 7 and 75. Function `F` can thus be specialized. Figure 5.1(b) adds an `Opt` version to `F` where an anchor has been added at `l4`. In order to deoptimize to the baseline, the anchor must capture all of the arguments of the function (`x`, `y`, `z`) as well as the local register `d`: these are all live registers needed to continue the execution of `Base` from `l1`.

The compiler could then be able to notice that `d` is not used anymore in the optimized version. It can remove it from `Opt`, but in that case the anchor must remember its value to reconstruct it when deoptimizing, hence the `d ← 1` in the deoptimization metadata.

The speculation is done by `Assume [z=7, x=75] ⚓l4` which specifies what is expected from the state of the program and the anchor above. The optimized version has eliminated dead code and propagated constants. If the speculation holds, then this version is equivalent to `Base`. In this new version, the irrelevant computation of `a` has been removed and `x*x` is speculatively constant folded. While in this particularly simple example, the overhead of checking validity of the speculation might negate the benefits of the optimization, a single speculation may simplify many operations at once in a bigger function. If the speculation fails, then the execution should return to `Base`, at label `l1`, and reconstruct the original environment. This involves for instance materializing the variable `d`. As we see here, `Assume` does not have to be placed right after an `Anchor` instruction. This will cause deoptimization to appear to jump back in time and some instructions (here, `c ← y * y`) will be executed twice. It is up to the compiler to ensure these re-executed instructions are idempotent. For instance, in section 5.3.5, we show how our formally verified middle-end can guarantee that inserting an `Assume` after a branch is correct.

Using Anchors The role of anchors is subtle. Maintaining the mapping between a `Base` and `Opt` versions is far from trivial as the optimized code gradually drifts away from its base, one transformation at a time. For the middle-end compiler, an anchor marks a point where it knows how to reconstruct the state needed by `Base` given the currently live registers in `Opt`. To be able to reconstruct that state, the anchor keeps portions of the state alive longer than needed in `Opt`, and, as the compiler optimizes code, the varmap is updated to track relevant changes in the code. Thus, the cost of an `Anchor` is that it acts as a barrier to some optimizations and may increase register pressure. Once the compiler has finished inserting assumes, all anchors can be deleted (see Section 5.2.4). They will not be used in the final version of the function sent to the backend compiler.

For our proofs, anchors have yet another role. They justify the insertion of `Assume` instructions (see Section 5.2.1). For this, we give the `Anchor` instruction a nondeterministic semantics, an anchor can randomly choose to deoptimize from `Opt` to `Base`. Crucially, deoptimization is always semantically correct at an anchor, nothing is lost by returning to the baseline code eagerly other than performance. An inserted `Assume` is thus correct if it follows an `Anchor` and the observable behavior of the program is unchanged regardless which instruction deoptimizes. One benefit of having anchors is that the assumes they dominate can be placed further down

the instruction stream. The compiler must make sure that the intervening instructions do not affect deoptimization. This separation is important in practice as it allows a single `Anchor` to justify speculation at a range of different program points. Initially the varmap of an `Assume` instruction will be identical to its dominating `Anchor`, but, as we will show shortly, this can change through subsequent program transformations. To sum up, in our middle-end compiler, the `Anchor` instruction is a helper for speculation and reasoning about correctness of speculation that is removed in the last step of the optimization pipeline.

5.1.1 CoreIR Syntax

CoreIR is inspired by CompCert's RTL [Leroy 2009b]. Its formal syntax is given in Figure 5.2. Code is represented by control-flow graphs with explicit program labels l. Each instruction i explicitly lists its successor(s). The instructions operate over an unbounded number of pseudo-registers, r, holding integer values v. Operations assign an expression to a register. A single operation instruction can contain several such assignments, which are meant to be evaluated in parallel. A program is a map from function identifiers to functions. Each function f has a default `Base` version V, its original version, and one optional optimized version `Opt`. This version may deoptimize back to its baseline if speculation fails. Versions contain code and an entry label. `Base` versions do not use the `Anchor` and `Assume` instructions. The deoptimization metadata in those operations consists of the function identifier and label where to jump to $(F.l)$ along with a varmap $(r_1 \leftarrow e_1, \ldots, r_n \leftarrow e_n)$ indicating that the value of target register r_i is that of expression e_i. The metadata can contain information to synthesize additional function frames as if they called the function to deoptimize to, in which case we also specify the register that will hold return value of the call. Examples of speculative instructions synthesizing such extra stackframes are shown in Section 5.2.6. The list of expressions in an `Assume` represents what the instruction is speculating on, and is called the *guard* of the `Assume`. Some simplifications have been made to avoid tying the development to any particular language; these include the addressing modes and the arithmetic operations. CoreIR programs can interact with the heap by storing and getting integer values with `Store` and `Load`. The arithmetic operations are generic and constitute a subset of the operations of CompCert RTL. CoreIR programs can output values with the `Print` instruction.

Expressions:

e	$::=$	$r + r \mid r - r \mid r * r \mid r \,\%\, r$	Binary Arithmetic
		$\mid r + v \mid r - v \mid r * v \mid -r$	Unary Arithmetic
		$\mid r < r \mid r = r \mid r = v$	Relational
		$\mid r \mid v$	Register or value

Instructions:

i	$::=$	**Nop** l	Noop
		$\mid (r_1 \leftarrow e, \ldots, r_n \leftarrow e_n)\ l$	Operations
		$\mid$ **Cond** $e\ l_t\ l_f$	Branch
		$\mid r \leftarrow$ **Call** $f\ (e_1, \ldots, e_n)\ l$	Call
		$\mid$ **Return** e	Return
		$\mid r \leftarrow$ **Load** $e\ l$	Memory load
		$\mid e \leftarrow$ **Store** $e\ l$	Memory store
		$\mid$ **Print** $e\ l$	Output value
		$\mid$ **Anchor** $deop\ l$	Deoptimization anchor
		$\mid$ **Assume** $(e_1, \ldots, e_n)\ deop\ l$	Speculation

Metadata:

vm	$::=$	$r_1 \leftarrow e_1, \ldots, r_n \leftarrow e_n$	Varmap
syn	$::=$	$f.l\ \ r\ \ vm$	Stackframe
$deop$	$::=$	$f.l\ vm\ (syn_1, \ldots, syn_n)$	Deopt metadata

Programs:

V	$::=$	$l \mapsto i$	Code
F	$::=$	$\{(r_1, \ldots, r_n), l, V, option\ V\}$	Function
P	$::=$	$f \mapsto F$	Program

Figure 5.2 – Syntax of CoreIR

5.1.2 CoreIR Semantics

The small-step semantics of a CoreIR program P is detailed in Figure 3.3. We define its judgment as $S\ V\ l\ R\ M \xrightarrow{t} S'\ V'\ l'\ R'\ M'$, where t is a trace of observable events, and a semantic state consists of a stack S, the current version V, the current label l, registers R and a memory heap M. A stack is a sequence of frames (r, V, l, R) containing the register r where the result will be stored, the caller V, the return label in the caller l, and the registers to restore R. Sequences of frames can be concatenated with the operation ++, which adds some frames to the execution stack. `[]` denotes the empty stack. States can also be final (v_{final}, M), in case the main function has returned, and then only contain a value and a memory. The memory heap

$$\text{Nop}\ \frac{V[l_{pc}] = \texttt{Nop}\ l_{next}}{S\ V\ l_{pc}\ R\ M \longrightarrow S\ V\ l_{next}\ R\ M}$$

$$\text{Op}\ \frac{V[l_{pc}] = r_1 \leftarrow e_1, \ldots, r_n \leftarrow e_n\ l_{next} \quad (e_1, R) \downarrow v_1 \ \ldots\ (e_n, R) \downarrow v_n}{S\ V\ l_{pc}\ R\ M \longrightarrow S\ V\ l_{next}\ R[r_1 \leftarrow v_1 \ldots r_n \leftarrow v_n]\ M}$$

$$\text{ConT}\ \frac{V[l_{pc}] = \texttt{Cond}\ e\ l_t\ l_f \quad (e, R) \downarrow true}{S\ V\ l_{pc}\ R\ M \longrightarrow S\ V\ l_t\ R\ M}$$

$$\text{ConF}\ \frac{V[l_{pc}] = \texttt{Cond}\ e\ l_t\ l_f \quad (e, R) \downarrow false}{S\ V\ l_{pc}\ R\ M \longrightarrow S\ V\ l_f\ R\ M}$$

$$\text{Call}\ \frac{\begin{array}{c}V[l_{pc}] = r \leftarrow \texttt{Call}\ f\ (e_1, \ldots, e_n)\ l_{next} \\ \texttt{current\_version}\ P\ f = V' \quad \texttt{init\_regs}\ (e_1, \ldots, e_n)\ R\ f = R'\end{array}}{S\ V\ l_{pc}\ R\ M \longrightarrow (r, f, l_{next}, R +\!\!+ S)\ V'\ entry\ (V')\ R'\ M}$$

$$\text{Ret}\ \frac{V[l_{pc}] = \texttt{Return}\ e \quad (e, R) \downarrow v}{(r, V', l_{next}, R' +\!\!+ S)\ V\ l_{pc}\ R\ M \longrightarrow S\ V'\ l_{next}\ R'[r \leftarrow v]\ M}$$

$$\text{RetFinal}\ \frac{V[l_{pc}] = \texttt{Return}\ e \quad (e, R) \downarrow v_{final}}{[\,]\ V\ l_{pc}\ R\ M \longrightarrow (v_{final}, M)}$$

$$\text{Print}\ \frac{V[l_{pc}] = \texttt{Print}\ e\ l_{next} \quad (e, R) \downarrow v}{S\ V\ l_{pc}\ R\ M \xrightarrow{v} S\ V\ l_{next}\ R\ M}$$

$$\text{Load}\ \frac{V[l_{pc}] = r \leftarrow \texttt{Load}\ e\ l_{next} \quad (e, R) \downarrow a \quad M[a] = v}{S\ V\ l_{pc}\ R\ M \longrightarrow S\ V\ l_{next}\ R[r \leftarrow v]\ M}$$

$$\text{Store}\ \frac{\begin{array}{c}V[l_{pc}] = e_1 \leftarrow \texttt{Store}\ e_2\ l_{next} \quad (e_2, R) \downarrow v \\ (e_1, R) \downarrow a \qquad M' = M[a \leftarrow v]\end{array}}{S\ V\ l_{pc}\ R\ M \longrightarrow S\ V\ l_{next}\ R\ M'}$$

$$\text{Ignore}\ \frac{\begin{array}{c}V[l_{pc}] = \texttt{Anchor}\ f.l\ vm\ (st_1 \ldots st_n)\ l_{next} \\ \texttt{deopt\_regmap}\ vm\ R = R' \quad \texttt{synthesize\_frame}\ R\ (st_1 \ldots st_n) = S'\end{array}}{S\ V\ l_{pc}\ R\ M \longrightarrow S\ V\ l_{next}\ R\ M}$$

$$\text{Deopt}\ \frac{\begin{array}{c}V[l_{pc}] = \texttt{Anchor}\ f.l\ vm\ (st_1 \ldots st_n)\ l_{next} \\ \texttt{deopt\_regmap}\ vm\ R = R' \quad \texttt{synthesize\_frame}\ R\ (st_1 \ldots st_n) = S'\end{array}}{S\ V\ l_{pc}\ R\ M \longrightarrow (S' +\!\!+ S)\ (\texttt{base\_version}\ f)\ l\ R'\ M}$$

$$\text{AssumePass}\ \frac{V[l_{pc}] = \texttt{Assume}\ (e_1 \ldots e_n)\ f.l\ vm\ (st_1 \ldots st_n)\ l_{next} \quad (e_1 \ldots e_n, R) \Downarrow true}{S\ V\ l_{pc}\ R\ M \longrightarrow S\ V\ l_{next}\ R\ M}$$

$$\text{AssumeFail}\ \frac{\begin{array}{c}V[l_{pc}] = \texttt{Assume}\ (e_1 \ldots e_n)\ f.l\ vm\ (st_1 \ldots st_n)\ l_{next} \quad (e_1 \ldots e_n, R) \Downarrow false \\ \texttt{deopt\_regmap}\ vm\ R = R' \qquad \texttt{synthesize\_frame}\ R\ (st_1 \ldots st_n) = S'\end{array}}{S\ V\ l_{pc}\ R\ M \longrightarrow (S' +\!\!+ S)\ (\texttt{base\_version}\ f)\ l\ R'\ M}$$

Figure 5.3 – CoreIR small-step operational semantics

is a fixed-size array of integer values, defined by a type `mem_state`, a constant representing the initial memory state and two operations over memory states provided as partial functions for accessing values ($M[a]$) and updating them ($M[a \leftarrow v]$). Option types are used to represent potential failures, like out-of-bounds accesses. When a rule does not emit any event, the empty trace is omitted. The only rule that emits an event corresponds to the execution of `Print`. The judgment for evaluating an expression e with the register map R is $(e, R) \downarrow v$. Similarly, lists of expressions can be evaluated as *true* or *false*, noted $(e_1 \ldots e_n, R) \Downarrow b$.

For readability, we use identifiers to name functions. In the semantics, a function has to be found in the program when called. $V[l_{pc}]$ denotes the instruction at label l_{pc} of version V, and $R[r \leftarrow v]$ is the update to register map R where r has value v. The function called `current_version` returns the current version of a function (the optimized version if it exists, the base otherwise), in the CoreIR program P. `entry` returns the entry label. To initialize registers of function f, we write `init_regs` $(e_1, \ldots, e_n)$ R f, where the expressions are evaluated in R. When deoptimizing, we need to reconstruct a register state with the mapping of the instructions. This is done with function `deopt_regmap`. The function `base_version` simply returns the `Base` version of a function, given its identifier. Finally, `synthesize_frame` creates the stackframes to synthesize during deoptimization.

If R contains value 17 for register r_1 and value 3 for r_3, then `deopt_regmap [r1, r2 <- r3 + 2]` R will create a new register mapping R' where r_1 has value 17, r_2 has value 5 and all other registers are undefined. To create additional frames, `synthesize_frame` R `[F.l retreg vm]` will create the stackframe (`retreg`, F, l, R') where R' is the result of `deopt_regmap vm`. At the next `Return` instruction, this will tell the execution to return to function F, at label l, with registers R' where *retreg* has been set to the return value of the function. Additionally, `synthesize_frame` can create multiple stackframes if given multiple *synth* metadata (see Section 5.2.6).

Most instructions have a standard semantics. `Nop` instructions represent a simple jump to another label. Operations modify the current register map and move to the next label. Multiple assignments are performed in parallel, evaluating every expression in the current register map before modifying it. The `Cond` branches evaluate their condition and jump to the corresponding label. To perform a call, one must create a new register map with the arguments. When returning from a call, one uses the data contained in the last stackframe. If the execution stack is empty, a final semantic state is reached. The (Print) rule is the only one with an observable event, corresponding to the value of its argument. `Store` and `Load` instructions are the only instructions using the current memory heap M, either inserting new values at a given address, or loading a value from the memory to a register.

`Anchor` is the only nondeterministic instruction of CoreIR. The semantics of an `Anchor` is such that execution proceeds either with the next label (Ignore), or with a transition (*i.e.*, a deoptimization) to unoptimized code (Deopt). This is a helper instruction used during optimizations such as the speculation insertion pass and all `Anchor` instructions are removed at the end of optimizations. Note that both rules share the same preconditions, even if the result of `deopt_regmap` is not needed in the (Ignore) rule; this ensures that both rules can always be used in the same conditions. Proving a backward simulation relating programs with anchors then implies behavior preservation, independently of the rules chosen when executing anchors.

Differences with Sourir The role of this new instruction is to capture what Sourir [Flückiger et al. 2018] referred to as *transparency invariant*: Given an `Anchor` instruction it is always correct to add more assumptions, since the `Anchor` ensures matching states at both ends of deoptimization for all executions. `Assume` however behaves deterministically, only deoptimizing if the guard fails (AssumeFail). These instructions are not removed by the middle-end compiler, and Chapter 7 shows how they can be compiled to native code that correctly implements deoptimization and on-stack replacement. Sourir did not feature anchors, instead `Assume` instructions had to be inserted first in the optimization pipeline, for the same reason that our anchors are inserted first: in general it is easier to justify a synchronization between `Opt` and `Base` when they are identical. Then, the compiler was able to insert new speculation by adding expressions to the guards of existing assumes. Instead, we first insert anchors when `Opt` and `Base` are identical, then allow the insertion of new assumes next to these anchors.

The separation in two speculative instructions that appear in CoreIR but was not in Sourir has two main advantages. First, it allows us to clearly separate the two roles of speculative instructions: maintaining a synchronization between `Opt` and `Base` versions, and checking the assumptions. Second, nondeterministic semantics are an easier way to define and mechanize the transparency invariant while reusing the CompCert simulation framework. And the `Anchor`/`Assume` separation helps us confine this nondeterminism to an instruction that gets removed at the end of middle-end compilation. Previously in Sourir [Flückiger et al. 2018], their transparency invariant required a bisimulation, using an invariant with a particular property: semantic states at an `Assume` had to be related to the deoptimization target state. This definition was incompatible with the simulation framework of CompCert and the simplicity of its forward-to-backward technique (Figure 3.8) that we reuse in Section 5.3.3. We believe that this separation has made the mechanization of speculative transformations easier, while in contrast Sourir only contained pen-and-paper proofs.

5.2 Manipulating Speculative Instructions

In this section, we present the different passes that compose our formally verified middle-end compiler. The transformations include inserting `Anchor` instructions, inserting `Assume` instructions, standard constant propagation, inlining and removing `Anchor` instructions. All these passes create or modify the `Opt` version of CoreIR functions. When suggesting optimizations, the profiler suggests a list of middle-end passes to perform. The profiler can suggest an optimization any number of times (constant propagation can sometimes find more simplifications after being run several times), including zero. They can be performed in any order, except for two exceptions. First, `Anchor` instructions can only be inserted when creating the `Opt` version, before any more optimizations are applied. Also, removing `Anchor` instructions always happens at the very end, so that when the middle-end compiler is finished, the JIT is left with a deterministic program, and the backend compiler presented in Chapter 7 does not have to compile anchors to native code.

5.2.1 Anchor Insertion

Anchor insertion must be run as the first pass in the optimization pipeline as it relies on a fresh copy of the base version, before any changes are made by other optimizations. No other pass presented in this work has this requirement. Intuitively, deoptimization points are easy to add when the `Opt` version is a simple copy of the `Base` one. Given a function and a list of labels, this pass creates a copy of the base version of the function, and inserts `Anchor` instructions at every label in the list. The `Anchor` instructions are a prerequisite for the later speculation pass, called assume insertion (as `Assume` instructions are inserted next to anchors). The profiler can suggest to insert `Anchor` instructions anywhere in a function. Selecting good locations for the insertion is the role of the profiler, and is out of scope of this work. Good locations may include the beginning of a function, to create a version specialized for some value of the arguments, or before a loop whose control-flow could be simplified with speculation. In any case, we allow the flexibility of inserting speculation points anywhere.

The following sections feature a running example with a `Function F` and its two versions, shown on Figure 5.4. The second version was obtained by inserting an `Anchor` instruction after the `Call` instruction. The two versions do not differ except for that instruction.

The deoptimization target of the `Anchor` points to the base version of `F`, at `l1`, where the instruction was inserted. The deoptimization metadata includes everything needed to reconstruct the original environment. This is done according to the varmap `[a,t]`, which is syntactic

```
Function F(a):
Version Base:
      t ← Call G(a,0)
  l1: Return a*t

Version Opt:
      t ← Call G(a,0)
  l2: Anchor F.l1 [a,t]
      Return a*t
```

Figure 5.4 – Running example, after Anchor insertion

sugar for [a ← a, t ← t] and denotes that expression a is evaluated in Opt's environment and bound to a in Base's environment; the same applies to t. Since the rest of the code is unchanged, it suffices to reconstruct the values of each defined register with their current values. Further optimization passes that would want to modify these values should also modify them in the deoptimization metadata to keep the synchronization between the two versions. The metadata is constructed using an analysis of live and defined registers in the current environment. In this example, the analysis sees that both a and t are live registers after the Call instruction as their values are used in the final Return statement. Both registers are also defined after the Call instruction (t has been assigned the return value of the function call, and a is a parameter to the function). As a result, the deoptimization metada of the new Anchor captures both registers.

The analyses used to find live and defined registers at a particular program point are implemented using the Kildall module of CompCert (as introduced in Section 3.1.1). Capturing in the deoptimization metadata a register that is not defined (meaning that it has not been assigned any value in that function) would be a mistake: the inserted Anchor would have no semantics since it evaluates the registers used in the metadata to reconstruct a new register mapping. That code transformation would not preserve progress (Figure 3.6) of the program it is applied to. Capturing nonlive registers (registers that are not used for the remaining of the Base version) would not be an issue for correctness, but deoptimization would reconstruct useless register values. In a first version, we simply captured the defined registers, but later implemented the liveness analysis. In our experiments, the cost of performing the analysis has been outweighed by the benefits of reduced register pressure.

5.2.2 Assume Insertion

Using these Anchor instructions, the middle-end compiler is able to insert Assume instructions. Unlike Anchor Insertion, this pass can happen at any time during the optimization pipeline, as long as anchors have been inserted already. The profiler may provide the label of an Anchor, and a guard expression, and the optimizer will try to insert an Assume with that guard right after that Anchor instruction. Another version of this pass, where the Assume is not directly after the Anchor, is presented in Section 5.2.5. In our example a request that can be satisfied would be to add the speculation t=0 at label l2, as depicted on Figure 5.5.

```
Version Opt:
      t ← Call G(a,0)
  l2: Anchor F.l1 [a,t]
      Assume t=0  ⚓l2
      Return a*t
```

Figure 5.5 – Running example after Assume insertion

The required metadata is copied from the Anchor instruction. As syntactic sugar to avoid repetition we just refer to the Anchor by ⚓l2, to denote that the metadata is identical. Since the profiler can suggest any guard to insert in an Assume instruction, the middle-end compiler must ensure that this guard may not introduce bugs in the program. In practice, this means that the guard must evaluate successfully, without errors. The Assume insertion pass thus includes an analysis to make sure that the guard will evaluate to some boolean value. In this case, checking that register t has a value is enough. We reuse the same defined analysis as used in the previous Anchor insertion pass. The semantics of the Anchor justifies that the execution will be equivalent to the original one whether the Assume deoptimizes or succeeds.

5.2.3 Constant Propagation

This next optimization pass is a standard dataflow analysis seen in many compilers. It closely resembles the constant propagation pass from CompCert presented in Chapter 3, and reuses its Kildall fixpoint solver library. If a register can be statically known to contain a value, it will get replaced. Expressions and instructions can be simplified. Other similar passes from CompCert with similar dataflow analyses (like Dead Code Elimination) could be reused, using the same verification technique shown in Section 5.3.3. We chose this particular compiler pass because of the added benefit of its interaction with the Assume instruction.

```
Version Opt:
      t ← Call G(a,0)
  l2: Anchor F.l1 [a,t]
      Assume t=0   ⚓l2
      Return 0
```

Figure 5.6 – Running example after constant propagation

Its interaction with the `Assume` instruction turns it into a speculative optimization. When constant propagation analyses an optimized version containing `Assume` instructions, it knows that the control flow will remain in that version only if the guard holds. Our previous example can thus be further optimized by transforming the return expression using the speculation, as in Figure 5.6. This also showcases how the `Assume` instruction is not handled like a simple branch. One limitation of the constant propagation pass of CompCert is that it cannot use the condition of a branching instruction to update its branches differently. For instance, a program containing the branch `if (t == 0)` will not replace `t` by 0 inside the *true* branch. However `Assume` is not a branching instruction for an intraprocedural analysis like the one used in constant propagation. Upon seeing `Assume` (`t = 0`), we define our analysis to set `t` to 0 for the next instructions. These instructions will only be reached if the `Assume` did not deoptimize.

Another feature of constant propagation in CoreIR is its ability to simplify deoptimization metadata. For instance, an instruction `Anchor F.l [x]` could be simplified to `Anchor F.l [x ← 3]` if the analysis shows that register `x` will hold value 3 at that point.

5.2.4 Removing Anchors

`Anchor` instructions are used to justify `Assume` insertion, but should not be executed. After all optimization wishes have been treated by the middle-end compiler, this pass removes all `Anchor` instructions, and replaces them with `Nop` instructions. In the resulting function, shown in Figure 5.7, the original version has been kept intact, and a new optimized version has been inserted.

5.2.5 Delayed Assume Insertion

So far we have not used the full flexibility of `Assume` insertion provided by `Anchor` instructions. For that we slightly modify our running example and define a new function in the left of Figure 5.8. In this variant there is an additional condition on register `a`, influencing the result

```
Function F(a):
Version Base:
      t ← Call G(a,0)
  l1: Return a*t

Version Opt:
      t ← Call G(a,0)
      Assume t=0  F.l1 [a,t]
      Return 0
```

Figure 5.7 – Running example after removing the Anchor

value. We would like the profiler to have as much freedom as possible for inserting assumptions. This means for example being able to insert assumptions at `l1`, `l2`, and `l3`. As seen in section 5.2.2 this could be achieved by preprocessing this version to include an `Anchor` at each of those labels. But, such an approach would have downsides, since additional deoptimization points constrain optimizations and unnecessarily bloat the code. Instead it is sufficient to place an `Anchor` at `l1`, and show that we can use it to justify `Assume` instructions at `l2` and `l3`. As a concrete example, Figure 5.8 shows a valid use of the `Anchor` by a delayed `Assume`.

```
Version Base:
      t ← Call G(a,0)
  l1: Cond a=0 l3 l2
  l2: Return a*t
  l3: Return 1
```

```
Version Opt:
      t ← Call G(a,0)
      Anchor F.l1 [a,t]
      Cond a=0 l3 l2
  l2: Assume t=0  F.l1 [a ← 0,t]
      Return 0
  l3: Return 1
```

Figure 5.8 – Example of delayed Assume insertion

In case this assumption fails, the execution travels back in time to `F.l1` executing the condition at `l1` a second time. Since the condition is a silent operation and does not alter any registers referenced later, the behavior is preserved. Therefore a single `Anchor` instruction can serve all possible locations for `Assume` instructions in this example. The delayed `Assume` insertion pass allows the profiler to suggest arbitrary locations for an assumption. It features a verification step that rejects all requests where the in-between instructions cannot be proved to be noninterfering. For now, this checker only accepts conditions, but could be extended with other instructions, like operations that do not modify the registers used in the deoptimization metadata. To explore the handling of other instructions, one could draw inspiration from the sufficient conditions of Sourir [Flückiger et al. 2018] to move an instruction over an

`Assume`. They require instructions that do not modify the deoptimization metadata, contain no side-effects or function calls, and have the `Assume` as only predecessor instruction.

5.2.6 Inlining with Speculations

Our middle-end compiler also features an inliner, a standard compiler optimization. We chose this optimization as it shows how speculation can make some standard transformations and their formal verification more complex. In particular, inlining `Assume` and `Anchor` has to be done with caution. Consider the program on Figure 5.9, where inlining of `G` in the `Opt` version of `F` simply replaces the `Call` instruction with the code of `G`. If the `Assume` instruction in `G` fails and deoptimizes, it returns to the original version of `G`. But if the `Assume` of `G` inlined into `F` fails, the execution would return to the original version of `G`, which upon returning skips the rest of the execution of `F`, and thus does not behave as the original program.

```
Function F(a,b):
Version Base:
      a ← a*3
      a ← Call G(a,b)
      ...
  l1: Return a

Version Opt:
      a ← a*3
      a ← Call G(a,b)
      Anchor F.l1 [a,b]
      ...
      Return a
```

```
Function G(c,d):
Version Base:
  l2: Return 2*c

Version Opt:
      Assume c=32  G.l2 [c]
      Return 64
```

Figure 5.9 – CoreIR program where the call to `G` should be inlined in `F`

The solution is to use the `Anchor` instruction right after the call in `Opt`. This is where the deoptimization metadata of CoreIR really shines: we can copy the metadata from the `Anchor` to synthesize a stackframe for the original version of `F`. We first initialize the parameters of the inlined call (`c` and `d`). The `Return` instruction of `G` is now replaced by a simple assignment to the return register of the `Call`, `a`. If the `Assume` fails, we deoptimize to the original version of `G`. The result is shown on Figure 5.10. The interesting part is the rest of the metadata `F.l1 a [a,b]` allowing deoptimization to reconstruct an additional stackframe that returns to `F` at label `l1`, return register `a`, and using a varmap `[a,b]`. For our inlining pass to be correct, it needs the `Call` instruction to be followed by an `Anchor`. The middle-end compiler checks that this is the

case, and cancels the optimization if there is no such instruction.

```
Function F(a,b):
Version Opt:
      a ← a*3
      c ← a
      d ← b
      Assume c=32  G.l2 [c], F.l1 a [a,b]
      a ← 64
      Anchor F.l1 [a,b]
      ...
      Return a
```

Figure 5.10 – Synthesizing an extra stackframe for a correct speculative inlining

5.3 Formal Verification of Speculation Manipulation

While the proofs of internal simulations closely resemble those of a static compiler, some complexity is added by the dynamicity of optimizations. In the static case, optimizing the entire program ahead of time means that in the optimized execution, any call to any function `f` will be replaced by a call to a function `f_opt`. In the dynamic case, function `f` and its original version may be part part of the execution stack before their optimization. Optimizations change the program, but not the stack. In that case, executing `f` in the source program can be both related to executing `f_opt` (when doing a new call), or `f` (when returning from the stack) in the new program. In practice, this means adding a single case to our matching inductive relations $\approx$ in the correctness proofs of our transformations.

Internal backward simulations can also be composed. Our optimization passes can thus be proved independently, each with a backward simulation. This allows for modular correctness arguments: inserting an `Anchor` is correct for other reasons than inserting an `Assume`. The next sections present the simulation proofs for the six optimization passes of our middle-end compiler. In terms of complexity, these simulation proofs resemble the ones that can be found in CompCert. The main novel difficulties come from the handling of speculative instructions and nondeterminism.

Backward Simulation **Mechanized**
Relates the semantics of a source program `p` to the semantics of the program `newp`, where function `fid` has been augmented with a new `Opt` version with `Anchor` instructions inserted at each of the labels contained in `lbllist`.

```
Theorem anchor_insertion_correct:
  ∀ p fid (lbllist:list label) newp,
    insert_anchors fid lbllist p = OK newp →
    backward_internal_simulation p newp.
```

Simulation 3 – Correctness of Anchor insertion

5.3.1 Correctness of Anchor Insertion

This pass (see section 5.2.1) is proved with an internal backward simulation, shown in Simulation 3. Since the pass adds new instructions, an optimized program execution will take additional steps. These additional steps are stuttering steps of the simulation, meaning that the simulation invariant ≈ sometimes relates two consecutive states of the target execution (both the `Anchor` and its next instructions).

We describe the simulation invariant that can be used to prove correct the insertion of `Anchor` instructions. Let V_{base} be the version where `Anchor` instructions are inserted, and V_{opt} the version after the insertion. In V_{opt}, some instructions have been replaced by an `Anchor`, and moved to a fresh (*i.e.* unused) label. We write $V\#l_{pc}$ to denote the instruction at label l_{pc} in version V. We write $\emptyset$ when there is no such instruction. The invariant ≈ comes in four different shapes and is defined in Figure 5.11. It relates semantic states of CoreIR before and after the transformation. The (refl) case states that outside of the modified version, the execution should be identical, except for differences in the stack explained below. As our JIT transformations always target a single version of a function, all our simulation invariants include a similar case. The (opt) invariant relates semantic states when executing the version with the new `Anchor` instructions. After passing an `Anchor` without deoptimizing, we use the (fresh) case to relate the label where the instruction has been moved to its original one. After deoptimizing at a newly inserted `Anchor`, semantic states are matched with the (deopt) case. By "agreeing", we mean having the exact same value on the set of live registers. During deoptimization, non-live registers are not captured by the `Anchor` and execution may end up in a state where less registers are defined than in the source execution. Note that in the Coq mechanization of the

$$\text{refl}\frac{S_1 \approx_{stk} S_2 \qquad V \neq V_{opt}}{(S_1\ V\ l_{pc}\ R\ M) \approx (S_2\ V\ l_{pc}\ R\ M)}$$

$$\text{opt}\frac{S_1 \approx_{stk} S_2}{(S_1\ V_{base}\ l_{pc}\ R\ M) \approx (S_2\ V_{opt}\ l_{pc}\ R\ M)}$$

$$\text{fresh}\frac{S_1 \approx_{stk} S_2 \qquad V_{base}\#l_{fresh} = \emptyset \qquad V_{base}\#l_{pc} = V_{opt}\#l_{fresh}}{(S_1\ V_{base}\ l_{pc}\ R\ M) \approx (S_2\ V_{opt}\ l_{fresh}\ R\ M)}$$

$$\text{deopt}\frac{S_1 \approx_{stk} S_2 \qquad R_1 \text{ and } R_2 \text{ agree on the live registers of } V_{base} \text{ at label } l_{pc}}{(S_1\ V_{base}\ l_{pc}\ R_1\ M) \approx (S_2\ V_{base}\ l_{pc}\ R_2\ M)}$$

Figure 5.11 – Invariant relation $\approx$ for `Anchor` insertion, transforming V_{base} into V_{opt}

$$\text{frame-same}\frac{V \neq V_{opt}}{(r, V, l_{pc}, R) \approx_{sf} (r, V, l_{pc}, R)}$$

$$\text{frame-opt}\frac{}{(r, V_{base}, l_{pc}, R) \approx_{sf} (r, V_{opt}, l_{pc}, R)}$$

$$\text{frame-deopt}\frac{R_1 \text{ and } R_2 \text{ agree on the live registers of } V_{base} \text{ at label } l_{pc}}{(r, V_{base}, l_{pc}, R_1) \approx_{sf} (r, V_{base}, l_{pc}, R_2)}$$

$$\text{stack-nil}\frac{}{[] \approx_{stk} []} \qquad \text{stack-cons}\frac{f_1 \approx_{sf} f_2 \qquad s_1 \approx_{stk} s_2}{f_1{+}{+}s_1 \approx_{stk} f_2{+}{+}s_2}$$

Figure 5.12 – Stack invariant for `Anchor` insertion

proof, the invariant is slightly more complex, due to how dataflow analyses like liveness are implemented in CompCert using its Kildall algorithm implementation. The (opt) case mention the analysis transfer function, so that we can infer that R_1 and R_2 agree on the live registers after deoptimization.

Since a version has been modified, the execution stack vary in both executions. This is described by the stack invariant $\approx_{stk}$, defined in Figure 5.12. This definition consists in first defining an equivalence relation on CoreIR stackframes, written $\approx_{sf}$. We then define $\approx_{stk}$ using the (stack-nil) and (stack-cons) rules, stating that matching execution stacks are sequences of matching stackframes according to $\approx_{sf}$. As seen in the (frame-opt) rule, after the insertion some stackframes of V_{base} may be replaced with stackframes of V_{opt}. This happens when the transformed version contained `Call` instructions. For function calls outside of the transformed version however, stackframes should be identical (frame-same). If we encounter a `Call` after having deoptimized from a newly inserted `Anchor`, the register mapping in the stackframes may vary (frame-deopt), but should agree on the live registers captured by the deoptimization.

Since there is a stuttering step at each inserted `Anchor`, we define the following measure on the semantic states of the transformed program:

$$m(S\ V_{opt}\ l_{pc}\ R\ M) = 1 \quad \text{if } V_{opt}\#l_{pc} \text{ is an inserted } \texttt{Anchor}$$
$$m(S\ V\ l_{pc}\ R\ M) = 0 \quad \text{otherwise.}$$

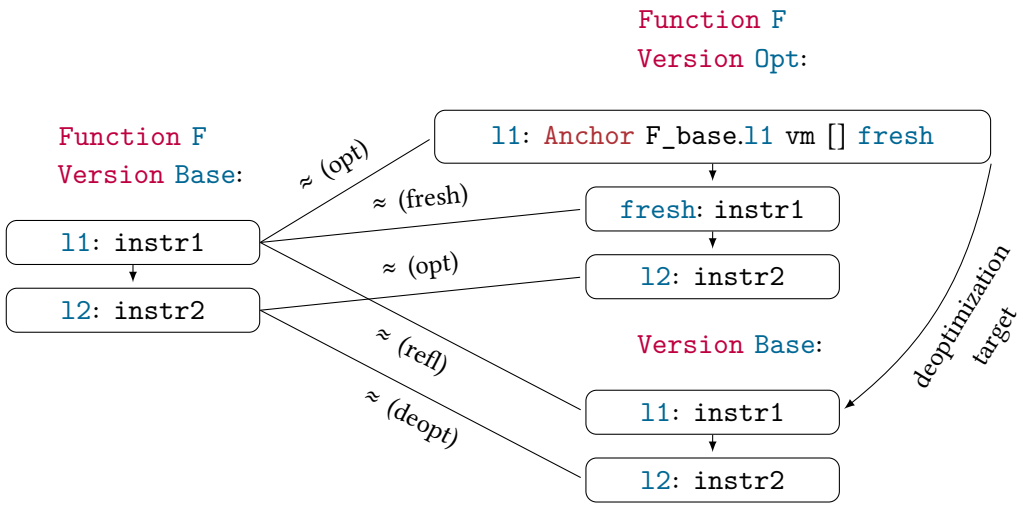

Figure 5.13 – Example of ≈ relation for `Anchor` insertion

An example of this invariant ≈ is depicted in Figure 5.13, where the optimization inserted an `Anchor` at label `l1`. The original program on the left has a single version for function `F`, while the program after the `Anchor` insertion pass now has an optimized version `F_opt` of `F`.

The new `Anchor` instruction can either go on or deoptimize, and in both cases, the invariant is preserved. The instruction `instr1` has been moved to a `fresh` (*i.e.* unused) label. Moreover, the deoptimization metadata includes the varmap `vm`, that assigns to each live and defined register its current value. To show preservation of the invariant, we prove that deoptimizing using this metadata preserves the contents of the live registers, as expected.

With this invariant, this simulation relates multiples behaviors (deoptimization or ignoring the `Anchor`) in the target program to a single behavior in the source one. This shows that combining the backward simulations from CompCert and the nondeterministic semantics of anchors is sufficient to capture the *possibility* of deoptimization expressed by the anchor.

5.3.2 Correctness of Assume Insertion

This pass (see section 5.2.2) is also proved with an internal backward simulation, shown in Simulation 4. The profiler suggests a guard (a list of expressions speculated to be true) and the location of an `Anchor` instruction. As the middle-end compiler should not have to trust the profiler, it checks that the suggested label actually corresponds to an `Anchor`, and runs our analysis to ensure that the guard will evaluate without errors. If the analysis succeeds, the `Assume` is inserted right after the `Anchor`. Inserting an `Assume` instruction next to an `Anchor` means adding a speculation check in the optimized execution, also represented by a stuttering step in the proof.

Backward Simulation **Mechanized**

Relates the semantics of source program `p` to the semantics of program `newp`, where the `Opt` version of `fid` has had an `Assume` inserted immediately after the `Anchor` located at label `lbl`. The `Assume` speculates on the expression `guard`.

```
Theorem assume_insertion_correct:
  ∀ p fid guard lbl newp,
    insert_assume fid guard lbl p = OK newp →
    backward_internal_simulation p newp.
```

Simulation 4 – Correctness of immediate Assume insertion

We define in Figure 5.14 our invariant relation $\approx$, when inserting an `Assume` in a version V_{src}. We name V_{opt} the version obtained after insertion. The new instruction is inserted at an unused label l_{fresh}, after an `Anchor` at label l_{anc}. The `Anchor` is modified to point to that fresh

$$\text{refl}\frac{S_1 \approx_{stk} S_2 \qquad V \neq V_{opt}}{(S_1\ V\ l_{pc}\ R\ M) \approx (S_2\ V\ l_{pc}\ R\ M)}$$

$$\text{opt}\frac{S_1 \approx_{stk} S_2 \qquad l_{pc} \neq l_{fresh}}{(S_1\ V_{src}\ l_{pc}\ R\ M) \approx (S_2\ V_{opt}\ l_{pc}\ R\ M)}$$

$$\text{spec}\frac{\begin{array}{c} S_1 \approx_{stk} S_2 \qquad V_{src}\#l_{anc} = \texttt{Anchor}\ deop\ l_{next} \\ V_{opt}\#l_{anc} = \texttt{Anchor}\ deop\ l_{fresh} \qquad V_{opt}\#l_{fresh} = \texttt{Assume}\ guard\ deop\ l_{fresh} \end{array}}{(S_1\ V_{src}\ l_{anc}\ R\ M) \approx (S_2\ V_{opt}\ l_{fresh}\ R\ M)}$$

Figure 5.14 – Invariant relation $\approx$ for `Assume` insertion, transforming V_{src} into V_{opt}

$$\text{frame-same}\frac{V \neq V_{opt}}{(r, V, l_{pc}, R) \approx_{sf} (r, V, l_{pc}, R)}$$

$$\text{frame-opt}\frac{}{(r, V_{src}, l_{pc}, R) \approx_{sf} (r, V_{opt}, l_{pc}, R)}$$

$$\text{stack-nil}\frac{}{[] \approx_{stk} []} \qquad \text{stack-cons}\frac{f_1 \approx_{sf} f_2 \qquad s_1 \approx_{stk} s_2}{f_1{+}{+}s_1 \approx_{stk} f_2{+}{+}s_2}$$

Figure 5.15 – Stack invariant for `Assume` insertion

label in the case where it does not deoptimize. The new `Assume` speculates on the expression $guard$ and only uses registers that are defined at l_{anc} according to an analysis.

The (refl) case serves the same purpose as in the previous invariant: outside of the optimized version, execution should encounter the same semantic states. Inside the optimized version, the same semantic states should be matched (opt) until the `Anchor` used for the insertion is reached. At which point, both the `Anchor` and the `Assume` of the optimized version are matched with the `Anchor` of the source version, with (opt) and (spec) respectively. Just like in the invariant of Section 5.3.1, execution stacks S_1 and S_2 can differ by substituting stackframes of V_{src} with stackframes of V_{opt}. The corresponding stack invariant is shown on Figure 5.15.

The stuttering is handled with the following measure:

$m(S\ V\ l_{pc}\ R\ M) = 1$ if $V = V_{opt}$ and $l_{pc} = l_{anc}$

$m(S\ V\ l_{pc}\ R\ M) = 0$ otherwise.

The simulation proof uses the nondeterminism of the `Anchor` insertion as a way to guarantee behavior equivalence regardless of the validity of the guard. For instance, if the speculation check fails in the optimized execution and the inserted `Assume` deoptimizes, then this behavior is related to the source behavior where the `Anchor` deoptimizes. Since the inserted `Assume` and the `Anchor` used for the insertion share the same deoptimization metadata, they deoptimize to the same semantic state.

5.3.3 Correctness of Constant Propagation

This pass (see section 5.2.3) closely resembles CompCert's constant propagation, but with some added interaction with the speculative instructions. For this proof, we successfully reuse the *forward-to-backward* methodology of Figure 3.8 that is used extensively in CompCert and allows simpler simulation proofs. However, the CoreIR semantics (see Figure 5.3) is not determinate (as defined on Chapter 3): the `Anchor` instruction can take two different steps that both produce a silent trace (outputting no event). For many optimization passes, it is important for the `Anchor` steps to be silent, as the behavior of `Anchor` should not be observable. For instance, a pass such as `Anchor` insertion would not be correct otherwise: it inserts new `Anchor` steps into the optimized execution but should not change the observable behavior of the program.

To circumvent this issue, we define a new temporary semantics called *loud* semantics for the `Anchor` instructions. The `Anchor` rules are simply augmented with a visible distinct event, as shown in Figure 5.16. Intuitively, an optimization pass such as constant propagation does not change the behavior of `Anchor` instructions: any behavior of an `Anchor` in the source program will behave just as the same `Anchor` in the optimized program. Apart from this observable

$$\frac{V[l_{pc}] = \texttt{Anchor}\ f.l\ vm\ st_1 \ldots st_n\ l_{next} \quad \texttt{deopt\_regmap}\ vm\ R = R' \quad \texttt{synthesize\_frame}\ R\ st_1 \ldots st_n = S'}{S\ V\ l_{pc}\ R\ M \xrightarrow{\textbf{GoOn}} S\ V\ l_{next}\ R\ M}$$

$$\frac{V[l_{pc}] = \texttt{Anchor}\ f.l\ vm\ st_1 \ldots st_n\ l_{next} \quad \texttt{deopt\_regmap}\ vm\ R = R' \quad \texttt{synthesize\_frame}\ R\ st_1 \ldots st_n = S'}{S\ V\ l_{pc}\ R\ M \xrightarrow{\textbf{Deopt}} (S' +\!\!+ S)\ (\texttt{base\_version}_P\ f)\ l\ R'\ M}$$

Figure 5.16 – Loud semantic rules for `Anchor` instructions

event, the rules are identical to those in Figure 5.3. Making these behaviors explicit in the semantics allows us to comply with determinacy and still preserve a program behavior. On these new semantics, we can use the forward-to-backward methodology. Finally, we show that a backward simulation on loud semantics implies a backward simulation on silent semantics. In the end, we prove the theorem of Figure 5.17.

```
Theorem fwd_loud_bwd:
    ∀ psrc popt,
    forward_internal_loud_simulation psrc popt →
    backward_internal_simulation psrc popt.
```

Figure 5.17 – From forward loud simulations to backward simulations

Using this methodology, the correctness proof for constant propagation then consists in proving a forward simulation, as shown in Simulation 5. This simulation closely resembles the constant propagation proof in CompCert. Its invariant relation is similar to the one used in CompCert, with an additional case resembling the (refl) case of Sections 5.3.1 and 5.3.2 to relate functions that have not been targeted by the transformation. Our other optimization passes have to be proved with backward simulations as they do not preserve the new observable events of Figure 5.16. However, this proof suggests that most standard nonspeculative optimization passes could be implemented in our middle-end compiler and proved just as in a static compiler. Note that this simulation, as well as the ones used for the correctness of delayed assume insertion Section 5.3.5 and inlining 5.3.6, have only been mechanized in the first pure version of our JIT [Barrière et al. 2021]. As future work, we could adapt our proof scripts to our current impure JIT implementation. This is discussed later in Chapter 8.

Forward Simulation **Mechanized** in the POPL21 artefact
Relates the *loud* semantics of source program `p` with target program `newp`, where the `Opt` version of function `fid` has been optimized with constant propagation.

```
Theorem constprop_correct_loud:
  ∀ p fid newp,
    constant_propagation fid p = OK newp →
    forward_internal_loud_simulation p newp.
```

Simulation 5 – Correctness of constant propagation

Comparison with Mixed Simulations Simulations on loud semantics can be reminiscent of the mixed simulations introduced in Pilsner [Neis et al. 2015]. Such simulations have also been implemented and used in CompCertM [Song et al. 2020], a variant of CompCert. Mixed simulations are another way to deal with nondeterminacy while still proving simulations in a forward manner. In essence, mixed simulations are simulations in which the invariant relation can be preserved either in a forward manner (see Figure 3.7) or in a backward manner (see Figures 3.5 and 3.6) depending on the current semantic state. The choice between a forward or a backward reasoning is done locally instead of choosing one direction for the entire proof. Then, one can use a theorem similar to the forward to backward theorem of Figure 3.8 to prove that a mixed simulation also implies the existence of a backward simulation. To do so, one needs to prove determinacy of the target program semantics only at the semantic states where forward reasoning is applied. When nondeterminacy is confined to a few instructions, this allows to prove most of the simulation in a forward manner.

The main difference with our approach is that a backward reasoning must still be applied at the nondeterminate steps. Using a forward simulation on loud semantics, every step is proved correct in a forward manner. Proving every step in the same direction reduces the number of intermediate lemmas; we tried both approaches for this proof and the loud semantics resulted in a simpler and more concise proof. In contrast, mixed simulations are more general (as general as backward simulations) and can be used to prove correct more transformations than forward simulations on loud semantics. For instance, the correctness of Anchor Insertion in Section 5.3.1 could be proved using mixed simulations (instead of the backward we wrote) using a backward reasoning only for the `Anchor` steps. However, this proof cannot be done using loud semantics as new observable events are inserted into the program.

5.3.4 Correctness of Removing Anchors

This pass (see section 5.2.4) is also proved with an internal backward simulation (Simulation 6). As it simply replaces `Anchor` instructions with `Nop` instructions, its correctness comes from the fact that the behavior of one such `Nop` instruction matches one of the possible behaviors of the `Anchor`: the (Ignore) rule of Figure 5.3, the one that goes on in the execution of the optimized version.

Backward Simulation **Mechanized**
Relates the semantic of `p` and `newp`, where `newp` is `p` where every `Anchor` instruction has been replaced with a `Nop` instruction.

```
Theorem remove_anchors_correct:
  ∀ p newp,
    remove_anchors p = newp ⟶
    backward_internal_simulation p newp.
```

Simulation 6 – Correctness of removing Anchors

5.3.5 Correctness of Delayed Assume Insertion

This pass (see section 5.2.5) is similar to the previous `Assume` insertion pass in its proof. However, an execution of the optimized program where the inserted `Assume` deoptimizes should be related to an execution of the source where the `Anchor` deoptimized earlier. To this end, it is essential that no side-effect occurs between an `Assume` and its `Anchor`. Our development, for instance, proves that it is safe to have a branch between the two speculative instructions. This allows to specialize different branches in different manners, and showcases that some instructions can remain between anchors and assumes. This pass is also proved with a backward simulation shown in Simulation 7.

In our simulation invariant $\approx$ presented in Figure 5.18, the optimized function stays matched to the `Anchor` until the guard of the `Assume` is evaluated. But to be able to catch up if the guard of the `Assume` succeeds, we also include in our invariant that for any step taken between the `Anchor` and the `Assume` in the optimized version, there exists a corresponding step in the source version that ends up in a matching state. In the example in Figure 5.18, the `Cond` instruction of the optimized program is matched to both `Anchor` and `Cond` instructions of the source program.

Backward Simulation **Mechanized** in the POPL21 artefact
Relates the semantics of source program `p` to the semantics of program `newp`, where the `Opt` version of `fid` has had an `Assume` inserted in the *true* branch of the `Cond` instruction following the `Anchor` located at label `lbl`. The `Assume` speculates on the expression `guard`.

```
Theorem assume_delay_correct:
  ∀ p fid guard lbl newp,
    insert_assume_delay fid guard lbl p = OK newp ⟶
    backward_internal_simulation p newp.
```

Simulation 7 – Correctness of delayed Assume insertion

In the in-between states, we are then proving two simulation diagrams at once: each of these states is related both to the `Anchor` (in dashed lines), and to a corresponding state that has executed the same instructions. That way, if the `Assume` fails, then we deoptimize to the exact same semantic state in both the source and optimized versions, and the in-between instructions did not emit any observable event. And if the `Assume` succeeds, then the preservation of the invariant has been constructed in a way to catch up in the source version.

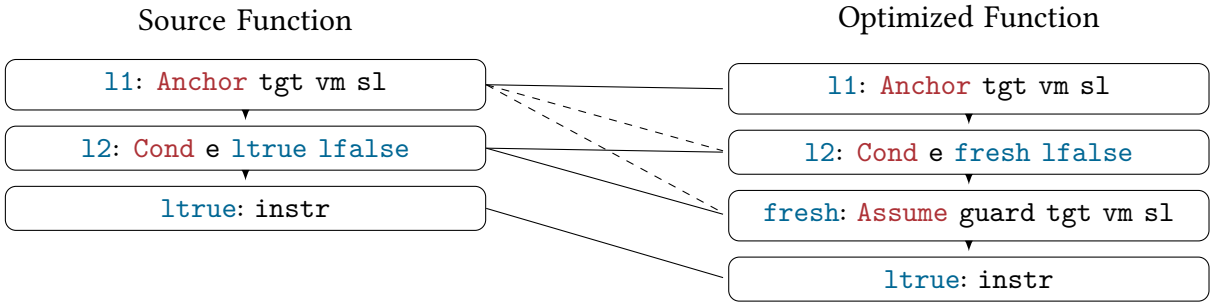

Figure 5.18 – The ≈ relation for delayed `Assume` insertion

In addition, the middle-end compiler checks that the `Cond` instruction used for the delay is strictly dominated by the `Anchor`, so that every execution going through the new `Assume` has seen the `Anchor`. To show that deoptimizing from the new `Assume` instruction goes to the same state as if the execution deoptimized at the `Anchor` instruction, we show that the `Cond` instruction does not have any observable behavior, and does not change the evaluation of the deoptimization metadata. This part would need to be adapted if we were to extend this optimization pass to other instructions (delaying the `Assume` after an `Op` instruction for

instance). As in the previous proof, we also check that the inserted guard in the `Assume` will evaluate without errors.

5.3.6 Correctness of Inlining with Speculations

Backward Simulation **Mechanized** in the POPL21 artefact
Relates the semantics of `p` and `newp`, where in `newp` the `Call` instruction at label `call_lbl` in function `fid` has been inlined. The inlined code is the code from the `Opt` version of the called function, possibly containing speculative instructions.

```
Theorem inlining_correct:
  ∀ p fid call_lbl newp,
    optimize_inline fid call_lbl p = OK newp →
    backward_internal_simulation p newp.
```

Simulation 8 – Correctness of inlining with speculative instructions

This pass (see section 5.2.6) must check that the inlined call is followed by an `Anchor` instruction, and use its metadata to synthesize an additional stackframe in the optimized program. Such a manipulation of the speculative instructions requires complex invariants. This pass is proved with an internal backward simulation (Simulation 8).

Our ≈ invariant comes in three shapes. The semantic states might represent the execution of the same version (*refl*). The target state might be in the new inlined version while the source one is in the caller function (*caller*). And the target state might be in the new version while the source one is in the function that we inlined (*callee*). In the last two cases, we design invariants to represent that the environment of the new version with inlining holds the environments of both the caller and the callee functions.

Consider Figure 5.19, where we show on a simplified example how this invariant can be preserved. We abbreviate `Assign args` the instruction that assigns each parameter of `G` to the corresponding expression of the list `args`. On the left, the source execution starts in the caller function `F`. As the execution progresses in `F`, we use the *caller* invariant. When the source calls into the callee function `G`, we show that the *callee* invariant holds: the new `Assign args` instruction successfully updates the optimized environment, which now contains the register mappings of both `F` and `G`. If a speculative instruction deoptimizes in the inlined code, it also deoptimizes in `G`. The optimized execution now deoptimizes back to the original version

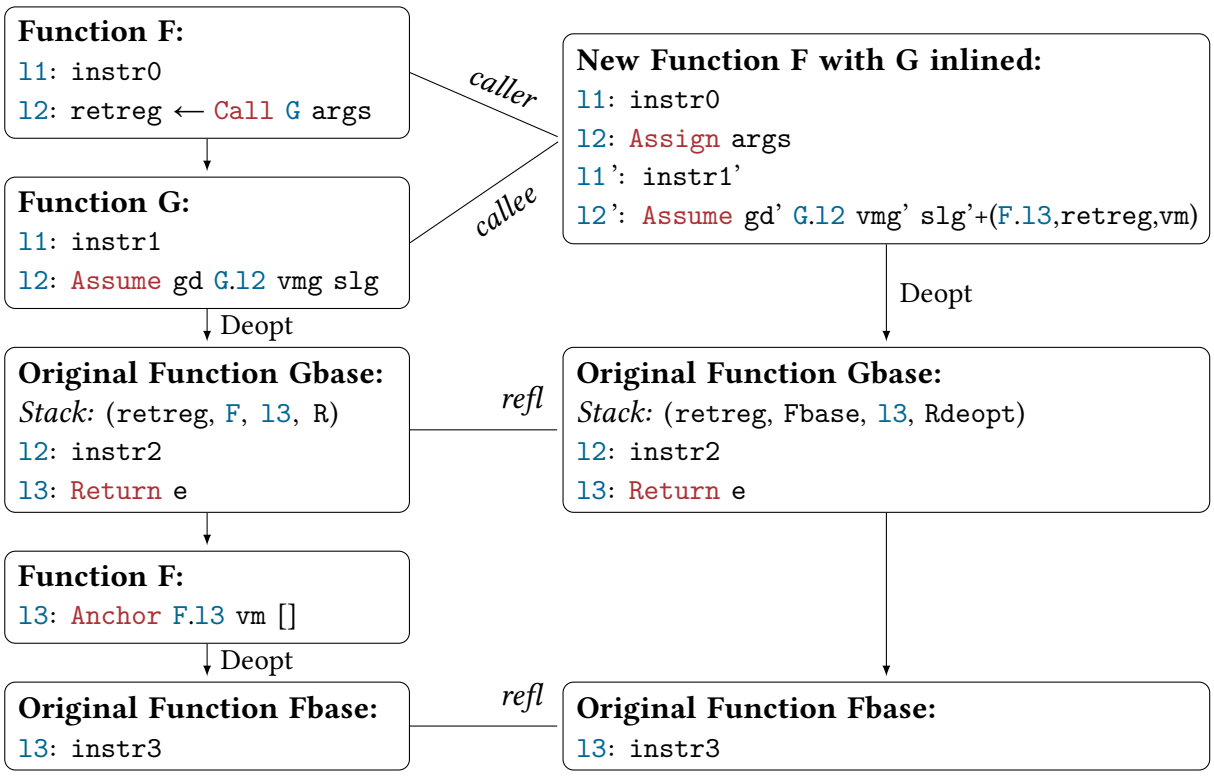

Figure 5.19 – Matching a deoptimization in the inlined code

`Gbase`. The new metadata that has been inserted synthesizes a new stackframe, pointing to `Fbase`. When the execution of `Gbase` finishes, we prove that returning into that synthesized stackframe matches the behavior of the source where we return to function `F`, and use the `Anchor` after the call to deoptimize once more into `Fbase`. This example only focuses on deoptimization, but many other behaviors must be matched. For instance if the `Assume` succeeded, then one needs to prove that stepping out of the inlined code ensures a new *caller* invariant. Or if another `Call` happens during the inlined part, the optimized version now has one less stackframe than the source in its stack. Expressing and preserving an invariant where stackframes can be matched in multiple ways makes this correctness proof the most complex one of our middle-end compiler. However, this optimization allows us not only to inline `Assume` speculations, but also to inline `Anchor` instructions and use them later to insert new speculations.

5.4 A Formally Verified JIT Middle-end Compiler

Finally, we prove the correctness of the entire middle-end compiler which can use the transformations presented earlier. Its correctness theorem is shown in Simulation 9, where `ps` is any profiler state (providing the list of optimization wishes). Being specified with a backward internal simulation, it matches the specification presented in Simulation 2, allowing its reuse in nested simulations so that it can be called dynamically. As we can see later in Chapter 7, this simulation can be composed with the similar Simulation 16 specifying the backend compiler, to get the correctness of the entire optimization step of our JIT.

Backward Simulation **Mechanized**

Given a profiler state `ps` that can be used to produce a list of optimizations, this theorem relates the semantics of `p`, to the semantics of `newp` where all the suggested optimizations have been performed.

```
Lemma middle_end_correct:
  ∀ p ps newp,
    middle_end ps p = newp ⟶
    backward_internal_simulation p newp.
```

Simulation 9 – JIT middle-end correctness

Figure 5.20 shows how the simulations presented in this Chapter can be composed together to obtain Simulation 9. The boxes p_1 to p_7 on the Figure represent the current program of the JIT as it gets transformed by the middle-end compiler. The exact passes that are performed by the middle-end compiler and their order can vary, and depend on what the profiler suggests at the time of the optimization. There are two exceptions however: the Anchor insertion and Anchor removal passes must be performed respectively at the beginning and the end of middle-end compilation. In the Figure, we depict an arbitrary order of middle-end passes, showcasing all of them. Performing these middle-end passes is the only moment where our JITs use the `Anchor` instructions. The nondeterminism of anchors explains why Simulations 3, 4, 7, 8 and 6 are proved with backward simulations. For constant propagation however (and any standard optimization not using the nondeterminism of Anchors), we have shown in Section 5.3.3 that we can prove a forward simulation on the *loud* semantics of p_5 and p_6, and construct a backward simulation using the theorem on Figure 5.17.

Speculation is a difficult yet essential feature of modern and efficient JITs. Our designs

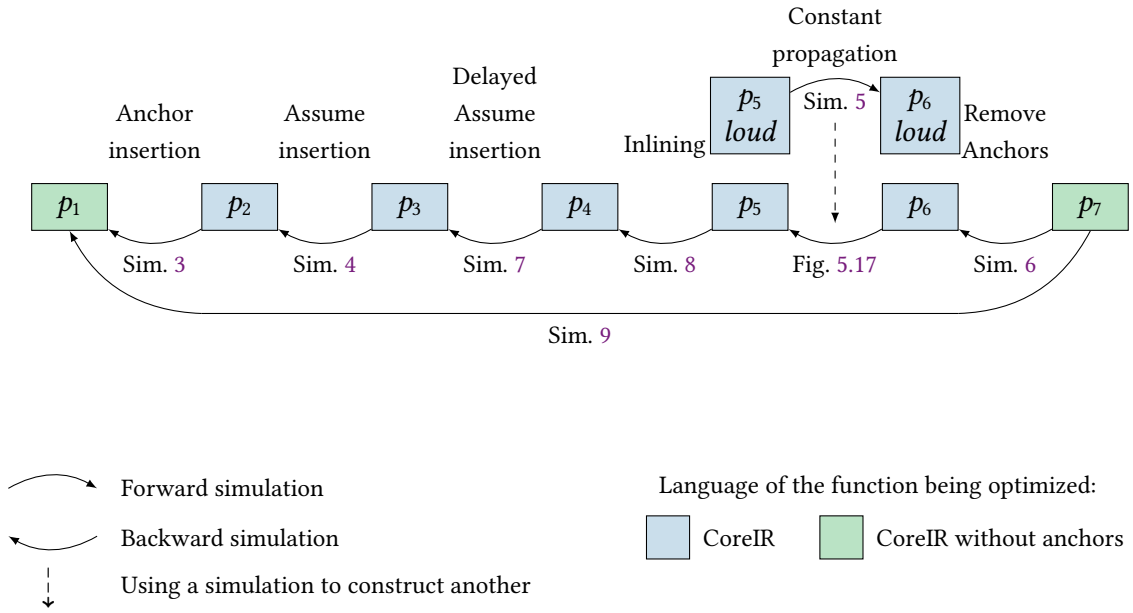

Figure 5.20 – Middle-end Correctness

for CoreIR and the middle-end show that it is possible to formally reason about speculation and deoptimization, the second JIT-specific verification challenge introduced in Section 1.2.1. CoreIR was designed to not be tied to any particular language, and the main ideas behind the speculative instructions semantics and the correctness arguments exposed in our proofs of section 5.3 could be used with any other mechanized intermediate language. Its similarities with CompCert RTL allow us in Chapter 7 to connect the middle end to the backend of CompCert.

In particular, we showed that nondeterministic semantics are a natural and simulation-compatible way to express what compilers expect when inserting speculation. While nondeterminism is often seen as an burden for formally verified compilation (as seen in section 3.1, the very first pass of CompCert removes it), the loud simulation methodology of section 5.3.3 shows that for standard passes that preserve the amount of speculative nondeterminism, we can reuse the forward-to-backward methodology used in CompCert.

Chapter 6

Formal Verification of Impure Coq JITs

The design introduced in Chapter 4, where every transition of the JIT is a pure and terminating Coq function, is good enough for simple JITs with only interpretation and without native code generation. However, it fails to address two challenges that modern JITs typically rely on for speedups: generating native code and interleaving native code execution with interpretation. Having multiple ways to execute code (running native code and interpretation) typically help JITs in finding the right balance between execution time, startup time and memory consumption.

JITs that generate and execute native code contain components that cannot be written in the pure functional programming language of Coq, Gallina. First, they must install and run the native code that has been generated. By *code installation*, we mean converting to machine code the assembly code that a backend compiler has produced (using an *assembler*), and writing it in a portion of the memory that is then given executable permissions. To the best of our knowledge, neither of these operations can be programmed in Gallina, the pure functional programming and specification language of Coq. Extending CompCert to include a formally verified assembler has been investigated before in CompCertELF [Wang et al. 2020]. This ahead of time compiler produces machine code, but does not write it in memory nor execute it. Second, a JIT with native code generation interleaves the execution of two languages, the IR and the native code. Both share some data structures (*i.e.*, an execution stack and a heap). Stack manipulation is made particularly difficult with on-stack replacement as one needs to be able to synthesize an interpreter stackframe. There is little hope of defining mutable, global Coq data-structures that once extracted can also be accessed and modified by native code. In Coq every object is immutable, and Coq proofs rely on the implicit assumption of that immutability. As a result, Coq proofs cannot guarantee anything about the execution of a Coq function extracted to OCaml where we modify the machine representation of extracted data-structures. In the rest of this work, all such components that cannot be written in Coq, whether it is global and

shared data-structures manipulation, native code installation or running native functions, are called *impure* components.

In this Chapter, we tackle the third JIT-specific verification challenge introduced in Section 1.2.1: designing and proving correct in Coq an impure JIT. Any JIT generating native code requires such impure components. This challenge is thus key to develop formally verified but realistic JITs in Coq with native code generation and execution. In Chapter 7, we will show how using the solution presented in this Chapter, we can reuse the CompCert proof to develop and prove a formally verified JIT backend compiler. Here, we propose a methodology that extends the design of Chapter 4 to delimit, specify and reason on the impure effects of a JIT.

6.1 A Monadic Encoding for Impure JITs

6.1.1 The Need for a New Extraction Methodology

A static compiler like CompCert is written as a pure and terminating Coq program. Hence, it is a prime candidate for a traditional extraction workflow: one can write a compiler as a Coq function, then one can extract that function to an equivalent executable OCaml function. This OCaml code can then be compiled and executed independently. However, JITs are impure and effectful programs. Yet, these impure parts (*e.g.*, calling native code or interacting with global data-structures) are not the only things a JIT does. A JIT must also translate code (with its backend compiler), interpret code, and orchestrate its components (with its monitor). All these remaining parts, making up for most of a JIT's work, can be written in a pure functional language and can be formally verified. In this section, we are tasked with the challenge of verifying in Coq a program that can only be *partially* written in Coq.

In Figure 6.1, we revisit the design presented in Figure 4.1, while identifying the impure parts of a JIT with native code generation. In a pure JIT with only interpretation, the execution stack and the heap could be defined as Coq immutable data-structures and their modification could be considered as pure manipulations. However, when interleaving interpretation with native code execution, such data-structures become shared across the two execution engines. JIT memory manipulation is then considered impure in our design. As a result, while most of the monitor can be written in Coq, it still needs some impure stack manipulation, for instance to inspect the top stackframe and decide whether to call the interpreter or native code on a function return. The interpreter also needs to manipulate the stack or the heap, for in-

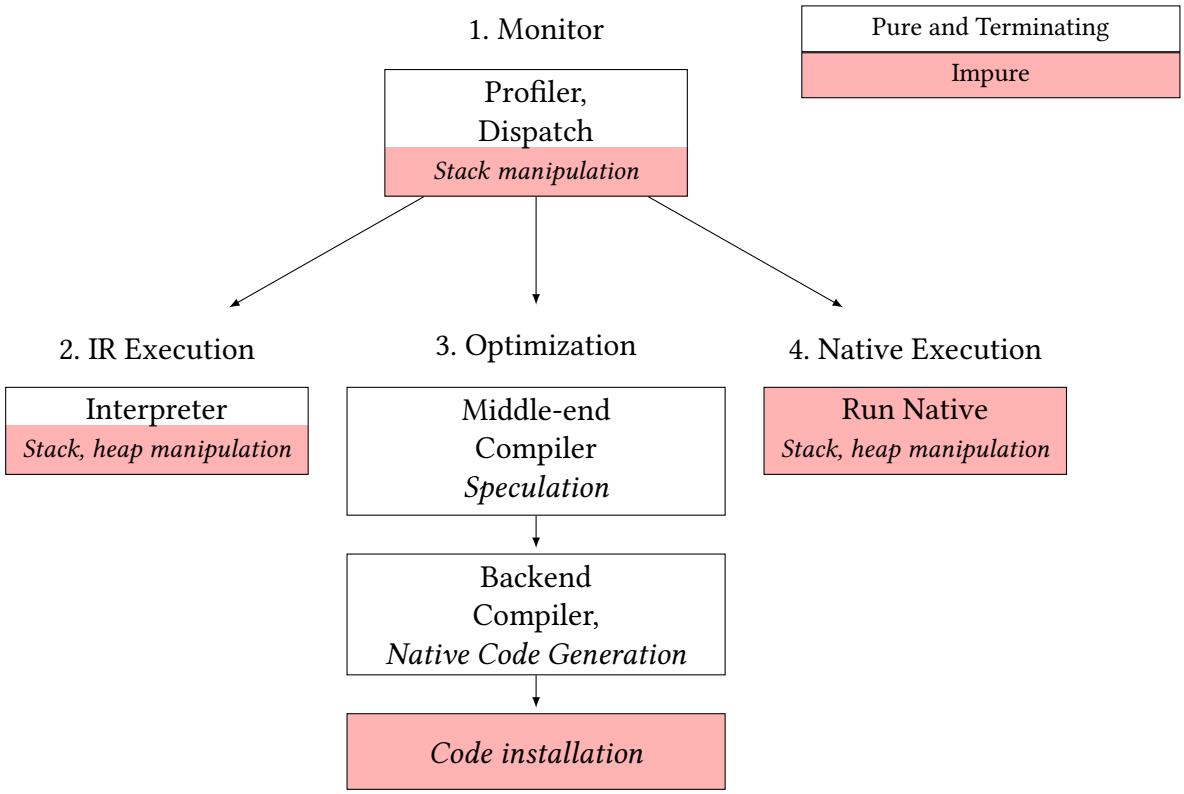

Figure 6.1 – Pure and impure components of a JIT with native code generation

stance when interpreting a **Store** instruction, which modifies the heap. Both the middle-end and the backend compilers can be considered as entirely pure, as they simply transform AST representations of code. The output of the backend compiler is an assembly AST, just like the output of the CompCert backend. However, writing and assembling that AST in the memory and making it executable is an impure component, just like running the installed code.

6.1.2 JITs as Incomplete Coq Programs

In order to develop impure JITs with native code generation, we must accept that these impure components cannot be written in Coq and extracted to OCaml. Our solution is to define a small set of primitives for everything that cannot be written in Coq in a JIT (*i.e.*, the pink boxes of Figure 6.1): manipulation of shared data-structures, native code installation and native code execution. These simple primitives are not implemented in Coq, but can be specified using Coq functions on an abstract model of the JIT memory. We need a formalism to encapsulate and delimit these impure effects such that our Coq proofs can reason on these abstract specifications, but our extracted JIT can use other implementations of the primitives. We use a variation of free monads, a pure functional encoding of effectful programs [Swierstra 2008], to represent in Coq a program such as the JIT with unimplemented primitives. Free monads have been used for years in pure functional languages to represent incomplete programs. The free monad definition used to write an impure JIT is defined in Section 6.3. In short, our encoding contains all the pure parts of a JIT, and its calls to some unimplemented impure primitives. Primitives can be called from both native code and components defined in Coq, allowing the interoperability that JITs intrinsically need. For instance, when a CoreIR function calls a function that we dynamically compile, we can generate native code that uses such a primitive to push the return value to the stack before returning to the JIT monitor. The monitor may now pop that value using another primitive, then call the interpreter again on the CoreIR function with that return value.

There are two ways to supplement such incomplete Coq programs: Coq *specifications* and *impure implementations*. First, primitive specifications allow us to reason on the behavior of an effectful JIT with multiple languages and shared data-structures, as if the JIT was entirely programmed in Coq. Specifically, using an abstract model of the JIT memory and an abstract specification of the primitives, we can redefine the JIT semantics of Figure 4.5, and reprove the behavior preservation theorem of Figure 4.7, even in the presence of impure components. This abstract model of the JIT memory contains an abstract model of the three disjoint portions of the memory the JIT uses: its execution stack, its heap and the executable memory where the

native code are installed. In Section 6.2, we present state and errors monads, which provide a good formalism to encode in a pure functional language all of our primitive specifications. For the JIT, we can write pure Coq monadic functions working on an abstract model of the JIT memory to specify each primitive. Next, Section 6.4 shows how we can fill the holes of an incomplete program with such primitive specifications.

Second, to get an impure executable JIT from such an incomplete program, we must move away from Coq. We can extract the incomplete JIT to an incomplete OCaml program. In OCaml, where effects are possible and other languages can even be called, we can define impure implementations of each primitive, working on actual shared data-structures and really calling native code. For instance, we call native code and C implementations of the global mutable data-structures from OCaml. In Section 6.5, we show that both the incomplete program and the primitive implementations can be composed together to get an executable impure program.

Our methodology allows to develop Coq JITs that come with a correctness proof, but can also be extracted and completed with primitive implementations to be executable. Using specifications, we can reason about impure programs in Coq that cannot be entirely written in Coq. If one trusts these primitive implementations to match their specification, the Coq correctness property extends to the executable JIT. Using free monads not only has the advantage of clearly identifying and specifying the various effectful components of a JIT, it also allows us to switch between different specifications of our data-structures. This is the *refinement* methodology of Section 6.6, facilitating the proofs of a multi-language program such as our JIT. The meta-theory needed to reason about our free monad implementation is very lightweight (a few hundred Coq lines). In particular, we only ever use free monads to write terminating computations, which eliminates the need for coinductive reasoning. We show in Section 6.7 that our monads can be used in conjunction with the small-step semantics framework of CompCert to reason about possibly infinite behaviors, and possibly diverging transitions of the JIT state machine of Section 4.2.

There are two main drives behind our chosen formalism. First, the meta-theory used to reason about effectful programs should be as simple and lightweight as possible, to not hinder already difficult proofs. Second, our formalism should be entirely compatible with CompCert so that we can reuse its simulation framework, which already handles proof composition and diverging executions. In the end, this free monad formalism could be used for any verification work trying to extend the CompCert simulation framework to effectful programs. We compare our methodology to other existing monadic encodings for impure program verification

in Section 6.8.

6.1.3 A Minimal Interface of JIT Impure Primitives

Since their implementation has to be trusted, we keep the smallest possible interface of impure primitives that a JIT with native code generation requires. Our impure JIT uses the following impure primitives:

- `HeapGet x` to get the heap value at address `x`.
- `HeapSet x y` to set `x` in the heap at address `y`.
- `Push x`, to push a value `x` on the stack.
- `Pop`, to pop a value from the top of the stack.
- `Push_IRSF sf` to add an interpreter stackframe `sf` to the stack.
- `OpenSF` to get the top stackframe of the stack.
- `Install_Code asm` to install some native code `asm` generated by a JIT backend compiler.
- `Load_Code fun_id` to load some installed native code for function `fun_id`.
- `Check_installed fun_id` to check if some function `fun_id` has been compiled.
- `Print x` to print some value `x`.

In static compilation, one simply generates code that modifies the memory, while in a JIT the memory is a data-structure directly modified during execution. `HeapSet` and `HeapGet` are used to modify and access the heap. These primitives are called by the IR interpreter when interpreting **MemSet** and **MemGet** instructions. Also, when compiling a function that contained such instructions, we generate native code that calls `HeapSet` and `HeapGet` so that executing a compiled function has the same effect on the JIT heap as interpreting its original version.

The JIT execution stack contains both interpreter stackframes and stackframes for the native code. Stackframes for the native code consist in 64-bit integers pushed on top of the stack, while interpreter stackframes are Coq records (containing values for the CoreIR registers for instance). The native code we generate uses the `Pop` and `Push` primitives to add or get integers from the stack, for instance to get function arguments, or push return values. When deoptimizing, the native code also pushes its deoptimization metadata to the stack using `Push`, so that the monitor can reconstruct the corresponding interpreter stackframe. The monitor uses both primitives when calling a native function (pushing arguments), or when returning from a native call (popping the return value), or after native-code deoptimization (popping the deoptimization metadata before creating the corresponding stackframe). But the stack also contains interpreter stackframes, that can be pushed by the interpreter with `Push_IRSF` when

an interpreted function calls another. Technically, we could design our interpreter such that it uses stackframes made of 64-bit integers as well, and then use the `Push` primitive for both native and interpreter stackframes. Such an interpreter would be more difficult to write and prove correct than our current version using these Coq records, which closely follows the CoreIR semantics. Having a dedicated primitive also shows that our design is not tied to a particular interpreter implementation. After a function return, the monitor uses `OpenSF` to get the top stackframe, and dispatches execution accordingly.

Finally, our JIT with native code generation must also install the generated native code. The optimizer uses `Install_Code` to allocate some space in the memory, write machine bytes and make that memory executable. The implementation of this primitive is discussed in Section 8.3.2. The addresses of these allocations are stored, and `Load_Code` can then be used when running native code to return the address of the machine code corresponding to a function identifier. `Check_Installed` is used by the monitor to decide if it should call the interpreter or the native code for function calls.

As JITs differ from static compilers in their need to do effectful impure computations, this list already sheds some light on the way a formally verified JIT should be designed. The impure effects of our Coq JIT are restrained to that small list of impure primitives. Everything else done by the JIT can be written directly in the pure programming language of Coq.

6.2 An Existing Solution for Specifying Effects: State and Error Monads

A standard way to encode effects in pure functional languages is to use *monads*. For effects that consist in computations modifying and accessing a global state, one can use the *state monad* [Wadler 1992]. In our JIT, we use a variation of the state monad that also includes errors: our primitives may modify a global state, but also fail (for instance, when trying to pop an empty stack). That state and error monad definition can also be found in CompCert. The CompCert definition of state monads that we reuse is on the left of Figure 6.2. Intuitively, a state monad of type `smon A` represents computations that either return a value of type `A` and possibly change the global state, or fail. The `smon` type is parameterized by a type `state` representing global states, which contain a model of the stack, heap, and executable codes. The type `sres` represents the possible return values of the monads. Such state monads are executable Coq functions, taking as argument an initial global state, and returning its return value as well as the new global state (or an error).

```
Definition smon (state A:Type) : Type :=
  state → sres state A.

Inductive sres (state A:Type) : Type :=
| SError : errmsg → sres state A
| SOK :  A → state → sres state A.
```

```
Definition sret state A (x:A) :  smon state A :=
fun (s:state) ⇒ SOK x s.

Definition sbind state A B (f: smon state A)
  (g:A → smon state B) : smon state B :=
  fun (s:state) ⇒ match (f s) with
    | SError msg ⇒ SError msg
    | SOK a s' ⇒ g a s'
    end.
```

Figure 6.2 – Defining the Coq state and error monad

Like all monads, state monads come with helpful constructors `sret` and `sbind` to build complex monadic computations, as defined on the right of Figure 6.2. The constructor `sbind` sequentially chains together monadic computations, constructing entire programs that may have global effects. For instance, executing `sbind f g` in some global state `s` first executes `f` on `s`. If that computation succeeded and returned a value `a` and modified the global state to `s'`, then we execute `g` on `a` and `s'`.

At first, one could think about writing a JIT as a state monadic computation whose state contains a model of the stack, the heap and the generated native codes. However, extracting such a JIT to OCaml would only produce a pure OCaml program, because Coq extraction only targets a pure subset of OCaml. With such an approach, there is little hope of getting an executable JIT that installs and calls actual native code. In order to develop executable impure JITs, we can instead use the free monad formalism presented in the next Section. To sum up, state monads are a great formalism to specify primitives that modify a global state, but this is not enough to implement an effectful executable JIT.

6.3 A Solution to Write Impure JITs in Coq: Free Monads

Since the primitives cannot be written in Coq, an impure Coq JIT is an incomplete Coq program. Free monads are a convenient formalism to write incomplete programs, namely programs with some holes left to represent the primitives that have not yet been implemented. Free monads provide a DSL to write such programs given a list of primitives that they may use. Such incomplete programs will either be completed with specifications of the primitives (Section 6.4), or with impure implementations of the primitives (Section 6.5). First, we inductively define the list of primitives our free monad definition uses as on the left of Figure 6.3. We see that one possible primitive is `Prim_Push`, taking an `int` as argument and not returning

```
Inductive primitive: Type → Type :=
| Prim_Push: int → primitive unit
| Prim_Pop: primitive int
| Prim_HeapSet: int → int →
primitive unit
| Prim_HeapGet: int → primitive int
...
```

```
Inductive free (T :Type) : Type :=
| pure (x : T) : free T
| impure R (prim : primitive R)
    (cont : R → free T) : free T
| ferror (e : errmsg) : free T.
```

Figure 6.3 – Definition in Coq of free monads

```
Fixpoint fbind X Y
  (f: free X) (g: X → free Y) : free Y :=
 match f with
 | pure x ⇒ g x
 | impure R prim cont ⇒
   impure prim (fun x ⇒ fbind (cont x) g)
 | ferror e ⇒ ferror e
 end.

Definition fret X (x:X) : free X := pure x.

Definition fprim R (p:primitive R) : free R :=
 impure p fret.
```

```
Notation "'do' X ← A ; B" :=
 (fbind A (fun X ⇒ B)).

Definition backend_and_install (f:function) :
   free unit :=
 (* generating RTL *)
 do f_rtl ← fret (IRtoRTL f_spec);
 (* using the CompCert backend *)
 do f_x86 ← fret (backend f_rtl);
 (* impure computation *)
 fprim (Prim_Install_Code f_x86).
```

Figure 6.4 – Free monadic constructors and using them to write an effectful JIT

any value.

Effectful computations are defined on the right of Figure 6.3. A term of type `free T` is called a free computation; it encodes a function that computes some value of type `T`, possibly using some primitives. The computation is either a pure computation, not using any primitive, an error, or an impure computation. Such an impure computation calls an unimplemented primitive `prim`, and then a continuation `cont` encodes the rest of that computation (possibly using other primitives), given the return value of the primitive. The CoreIR program that the JIT executes may have a going-wrong behavior, and errors help us define such going-wrong behaviors of the JIT, for instance if the program tries to call a function that is not defined. This simple type is quite easy to manipulate. For instance, one can write proof tactics (using the Coq proof tactics language LTac) that automatically check that some free computation only uses a subset of primitives and this helps us prove that the interpreter does not install any new native code.

Free monads are monads, and as such come with the monadic constructors on the left of Figure 6.4. Binding impure free computations entails adding primitives to the continuation.

These constructors allow us to define complex free computations, such as all the components of our impure JIT. For instance, the backend computation is detailed on the right of Figure 6.4. This code compiles and installs some function. First we call `IRtoRTL`, a Coq function we wrote which generates RTL code (see Section 7.1). This is a pure computation, using no primitive (hence the `fret`). We then call the CompCert backend, a pure Coq function that generates some equivalent x86 code. Finally, we call primitive `Prim_Install_Code`, that installs the generated code in an executable portion of the memory. While this looks like a simple program, `backend` is a pure function containing all of the CompCert backend passes, from RTL to x86. Here, the free monad encoding is used to orchestrate complex pure transformations with calls to the impure interface. To be executable, such a computation requires an implementation for `Prim_Install_Code`, discussed in Section 8.3.2.

The entire JIT can be represented this way. As explained in Chapter 4, we can define a Coq function representing the transitions of the JIT state machine of Figure 4.3. However, to describe the impure effects of the transitions, we change the `jit_step` signature to be `jit_step: jit_state -> free (trace * jit_state)`, allowing JIT transitions to call JIT primitives. Since we changed that signature, the JIT small-step semantics definition of Figure 4.5 has to be updated, this is done in the next Section, Figure 6.7.

6.4 Monadic Specifications and Semantics of Free Monads

Our impure JIT is an incomplete Coq program, written using free monads and lacking some implementation for the primitives it uses. In order to derive its semantics `jit_sem`, we specify each primitive using a state and error monadic specification (with the definitions of Section 6.2). Since our primitives work on global data-structures and may fail, the state and error monad is adequate for specifying them. We define in Figure 6.5 a *monadic specification* as a record containing an initial global state and a state-monadic computation for each primitive, called `Prim_X` for each primitive `X` of the list given Section 6.1.3. This record is parameterized by the type `mstate` of global states, containing a pure model of the stack, the heap, and the executable installed code of the JIT. The initial state is a global state that models the initial contents of the JIT memory: an empty stack, an empty heap, and no native code has been installed yet. This is used to define the initial state of the JIT small-step semantics. Moreover, we define `get_prim` to access the corresponding specification of a primitive with its arguments. An example of specification is given in Figure 6.8 for the `heap_get` primitive.

Furthermore, to fill the holes of incomplete programs, a function called `free_to_state` in

```
Record monad_spec (mstate:Type): Type :=
mk_mon_spec {
  init_state : mstate;
  prim_push: int → smon mstate unit;
  prim_pop: smon mstate int;
  prim_heapset: int → int → smon mstate unit;
  prim_heapget: int → smon mstate int
  ... }.
```

```
Definition get_prim R S (p:primitive R)
  (i:monad_spec S): smon S R :=
match p with
| Prim_Push x ⇒ (prim_push i) x
| Prim_Pop ⇒ (prim_pop i)
| Prim_HeapSet x y ⇒ (prim_heapset i) x y
| Prim_HeapGet x ⇒ (prim_heapget i) x
...
```

Figure 6.5 – Coq monadic specifications of representative primitives

```
Fixpoint free_to_state (A S:Type) (f:free A) (i:monad_spec S): smon S A := match f with
  | pure a ⇒ sret a
  | ferror e ⇒ fun (s ⇒ SError e)
  | impure R prim cont ⇒ sbind (get_prim prim i) (fun r:R ⇒ free_to_state (cont r) i)
end.
```

Figure 6.6 – Turning free computations into state and error computations

Figure 6.6 transforms any free monad computation `f` into a state monad computation. It simply replaces recursively any call to a primitive `prim` by its specification. It uses `get_prim` to get the primitive specification, and the continuation of an impure computation is bound to the result using the state-monad bind.

Now that free computations can be completed with primitive specifications, we can define the small-step semantics of the entire JIT. State and error monadic computations are executable Coq functions, so one can execute the one corresponding to the JIT transitions. The semantics states of the JIT execution contain both a `jit_state` (js_1), the data that can be written and manipulated in Coq (a state of Figure 4.3 also including the CoreIR functions), and a state of the state-monadic specification `mstate` ($ms1$), a model of the data structures that we cannot write in Coq. Note that the contents of the memory (stack, heap and native codes) are no longer represented in the `jit_state` as in Section 4.5.1, but they are now represented by the `mstate` type. One simply goes from one of such state to another according to the execution of the state-monad computation given by completing the JIT free transitions `jit_step`. This is defined by a single small-step semantic rule shown in Figure 6.7, where i is a monadic specification, and t is the observed trace. In the figure, we omit the types `A` and `S`, implemented respectively with `jit_state * trace` and the `mstate` type of i. These new small-step JIT semantics replace the ones defined in Figure 4.5, allowing impure JIT effects.

$$\texttt{jit\_sem}\ \frac{\texttt{free\_to\_state}\ (\texttt{jit\_step}\ js_1)\ i\ ms_1\ =\ \texttt{SOK}\ (js_2, t)\ ms_2}{(js_1, ms_1) \xrightarrow{t} (js_2, ms_2)}$$

Figure 6.7 – The JIT small-step semantic rule

```
int64_t heap_get (int64_t x){
  assert (x < HEAP_SIZE);
  int64_t val = jit_heap[x];
  return val;
}
```

```
Definition heap_get (x:int) : smon int :=
  fun s ⇒
    if (Int.lt x heap_size) then
      SOK (PMap.get (pos_of_int x) (heap s)) s
    else SError ("MemGet out of memory range").
```

Figure 6.8 – A C primitive implementation and its Coq monadic specification, that reuses CompCert libraries `Int` (to compare 64-bit integers) and `PMap` (associating values to positive addresses).

6.5 An Impure Implementation for an Effectful JIT

Using the free monad encoding, impure JITs can be completed with monadic specifications, but we can also extract impure JITs to OCaml and complete them with impure and effectful implementations of the primitives. The result is effectful OCaml JITs that can be executed. We write C functions for interacting with our global data-structures. For instance, Figure 6.8 shows the C implementation for the heap access primitive. On the right is its monadic specification used in the JIT semantics. While the C function accesses some global array `jit_heap`, the specification accesses a map contained in its monadic state `s`. Both the global array and the monadic state `s` are unchanged and the primitive fails for out-of-bound accesses.

Next, Figure 6.9 shows an interpreter of free monad computations in OCaml, where the function `exec_prim prim` executes the C function corresponding to the primitive `prim`. This free interpreter directly interprets the incomplete program itself, without resorting to the state

```
let rec free_interpreter (f: A free) : A = match f with
  | Coq_pure (a) ⟶ a
  | Coq_ferror (e) ⟶
    print_error e; failwith "Free monad error"
  | Coq_impure (prim, cont) ⟶
    let x = exec_prim prim in
    free_interpreter (cont x)
  end.
```

Figure 6.9 – Executing free computations in OCaml

monadic specifications and `free_to_state`. Running our free computation `jit_step` through this `free_interpreter` until the program execution finishes, we get an executable JIT in OCaml that calls effectful primitives.

6.6 Facilitating the Correctness Proofs with Refinement

We now have a methodology to write an impure JIT, specify its impure components, and extract it to an executable program. The next task consists in proving the JIT correct. In this Section, we develop a methodology to make the proof easier.

In our approach there is a small list of JIT-specific functions to manually audit: our primitives implementations must match their specifications. One could then try to write specifications that closely match what the impure implementation is doing, but in practice that monadic specification can be hard to reason with in the JIT correctness proof. In particular, for a fast access to the stack using only `Pop` and `Push` in the generated native code, one would like the execution stack to be a simple array of integers. However, when writing our proof invariants, it would be simpler if these integers were structured in several lists, each list corresponding to a specific stackframe. Moreover, in the final executable implementation, the execution stack is split in two parts: one that holds the interpreter stackframes, and one that holds the integer stackframes. This splitting makes the C implementation of the `Pop` and `Push` primitives simpler. Some simulation invariants (*e.g.*, in Simulations 13 or 15) are however easier to write if there is a single stack containing both interpreter and native stackframes. Then, proving the compilation of a function correct simply requires substituting its future stackframes instead of moving them from one structure to the other.

We argue that an advantage of using free monads is the ability to switch between different monadic specifications. One can then define a monadic specification that is close to the actual primitive implementations (called `prim_spec`), and another *reference* monadic specification (or `ref_spec`), with which proofs are easier to conduct. In this section, we show that once we prove that `prim_spec` *refines* `ref_spec`, then the correctness theorems about the JIT semantics using the reference specification can be propagated to the JIT semantics using `prim_spec`. We use this technique to switch between two specifications of the execution stack: an unstructured stack of integers, close to the C implementation, and a reference one where the stack is structured for easier stack invariants. Figure 6.10 shows the (simplified) types representing the monad state of both specifications. On the left, for the reference specification, the stack is a list whose elements contain either interpreter or structured native stackframes. On the right however,

```
Definition ASM_stackframe : Type := list int.

Record ref_state: Type := mkrefs {
  ref_stack:
    list (IR_stackframe + ASM_stackframe);
  ref_heap : heap;
  ref_codes: asm_codes;
}.

Definition ref_spec:
  monad_spec ref_state := ...
```

```
Record prim_state: Type := mkprims {
  prim_stack: list int
  prim_irstk: list IR_stackframe;
  prim_heap : heap;
  prim_codes: asm_codes;
}.

Definition prim_spec:
  monad_spec prim_state := ...
```

Figure 6.10 – Monadic states of the two primitive specifications

the state of `prim_spec` has split the stack. One list `prim_irstk` contains the interpreter stackframe, and the second `prim_stack` is a list of integers containing the concatenation of every native stackframe. This refinement methodology takes advantage of free monads to modularly separate the correctness arguments for stack manipulation and stack representation.

The benefit is a more modular reasoning: first we prove correct the JIT using `ref_spec`, then we prove the refinement. The full proof architecture used in our JIT is later shown in Figure 8.1. So, as composing CompCert simulations facilitates modular proofs, instead of redeveloping new proof techniques for modularity, we define our refinement relation so that we can reuse the simulation composition technique of CompCert. More precisely, a monadic specification i *refines* another j if there exists a relation $\simeq$ between monad states of i and monad states of j (written $s_i \simeq s_j$) such that for each primitive p,

$$\begin{aligned} \forall s_i\ s_j\ s_i'\ args\ r, \quad s_i \simeq s_j \quad \wedge \quad p_i\ args\ s_i\ =\ \texttt{SOK}\ r\ s_i' \longrightarrow \\ \exists s_j', \quad s_i' \simeq s_j' \quad \wedge \quad p_j\ args\ s_j\ =\ \texttt{SOK}\ r\ s_j' \end{aligned}$$

where p_k is the state monad for p in the monadic specification k, s_i and s_i' are monad states of i, s_j and s_j' are monad sates of j and $args$ are arguments for the primitive p (for instance, an `int` for `Push`). When such a relation exists, we say that i refines j and write $i \simeq j$. This definition purposely resembles the forward simulation definition of CompCert, but relating primitive executions instead of small-step semantics. We then prove the refinement theorem of Simulation 10, using the refinement relation to build the simulation invariant of a forward simulation.

As the JIT behavior is deterministic, that forward simulation is used to construct a back-

Forward Simulation **Mechanized**
Given two monadic specifications `i` and `j` and a program `p`, relates the semantics of the JIT executing `p` using `i` with the semantics of executing `p` with `j`. `jit_sem` is the JIT small-step semantics defined on Figure 6.7, using a monadic specification.

```
Theorem refinement:
  ∀ (prog_state istate jstate:Type) (p:prog)
    (i:monad_spec istate) (j:monad_spec jstate),
  refines i j →
  forward_simulation (jit_sem p i) (jit_sem p j).
```

Simulation 10 – Refinement theorem for switching specifications

ward simulation. Using that theorem on a Coq JIT and the two monadic specifications discussed above, we prove the main correctness theorem about `jit_sem` with the reference specification, and then propagate that correctness theorem to the JIT semantics using `prim_spec`, without additional proof effort. Proving the refinement (`prim_spec` ≃ `ref_spec`) is straightforward. In practice, while proving that an unstructured stack (`prim_stack` and `prim_irstk` in Figure 6.10) corresponds to a structured stack (`ref_stack`) is not a difficult thing, its justification should not hinder the proofs of every program transformation.

6.7 Nonatomicity of Transitions: Small-Step Semantics of x86 to the Rescue

As in the `backend_and_install` example of Figure 6.4, almost every transition of the JIT state-machine of Figure 4.3 is written as a terminating function of type `free T` for some return type `T`. Such transitions are called *atomic*: they describe an elementary computation done by the JIT. However, more than effectful transitions, a JIT compiler may also include nonterminating transitions. This only happens when calling the native code corresponding to a compiled CoreIR function: the JIT may have compiled and executed a nonterminating function. These transitions are called *nonatomic*, as they represent a possibly infinite sequence of elementary transitions of the generated native code.

This is not an issue with the IR interpreter which can be defined with some arbitrary fuel (*i.e.*, an integer limiting its maximum number of steps) and return regularly to the monitor

```
Inductive nasm_transition (State Trace:Type): Type :=
| Atomic: free (Trace * State) ⟶ nasm_transition
| LoadAndRun: nasm_transition.
```

Figure 6.11 – Nonatomic State Machine definition

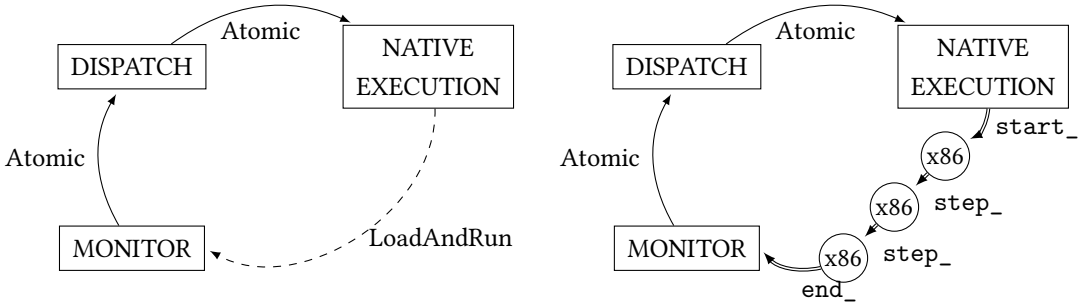

Figure 6.12 – Giving small-step semantics to nonatomic transitions in a NASM

only to be called again. But the dynamically generated native code may be stuck in a loop without any return, call or deoptimization, and the fuel technique cannot be used to represent such an infinite execution. This possibly diverging transition cannot be specified like our other primitives in Section 6.4, with a terminating free computation. In this section, we modify one last time the way transitions are represented to allow nonterminating and effectful transitions without fuel. Then, we modify the JIT semantics `jit_sem` of Figure 6.7 accordingly.

One solution could be to extend our free monad definition to a coinductive structure to represent and reason about possibly nonterminating effectful programs, like Interaction Trees [Xia et al. 2020]. However, this would require coinductive reasoning on such transitions. On the other hand, the CompCert simulations are already equipped to reason about nonterminating executions, as long as such executions are defined by small-step semantics. Moreover, x86 semantics in CompCert are already specified with executable small-step semantics, looping a function that computes the next semantic state. [1] As a result, the nonatomic transitions of the JIT can be decomposed into several atomic small steps of CompCert x86 semantics. This has two advantages: not only can we avoid writing coinductive proofs in Coq and instead reuse those of CompCert, but this also facilitates in Chapter 7 the reuse of the CompCert correctness theorem, expressed in terms of these x86 semantics, without modifying it.

To specify such native calls, we first define a JIT as a *NonAtomic State Machine* (or NASM),

1. More precisely, most of x86 semantics are defined with a function in CompCert. External calls are specified with an inductive nonexecutable parameter. However, in the subset of x86 that our JIT generates, the only such calls are calls to our JIT primitives that we can specify as usual with Coq state-monadic functions.

$$\text{jit-step}\ \frac{\texttt{jit\_step}\ (js_1)\ =\ \texttt{Atomic transition} \qquad \texttt{free\_to\_state transition}\ i\ ms_1\ =\ \texttt{SOK}\ (t,\ js_2)\ ms_2}{(js_1,\ ms_1) \xrightarrow{t} (js_2,\ ms_2)}$$

$$\text{start}\ \frac{\texttt{jit\_step}\ (js)\ =\ \texttt{LoadAndRun} \qquad \texttt{free\_to\_state start\_}\ i\ ms_1\ =\ \texttt{SOK}\ (xs)\ ms_2}{(js,\ ms_1) \longrightarrow (xs,\ ms_2,\ js)}$$

$$\text{x86-step}\ \frac{\texttt{free\_to\_state}\ (\texttt{step\_}\ xs_1)\ i\ ms_1\ =\ \texttt{SOK}\ (t,\ xs_2)\ ms_2}{(xs_1,\ ms_1,\ js) \xrightarrow{t} (xs_2,\ ms_2,\ js)}$$

$$\text{end}\ \frac{xs\ =\ Final\ (r) \qquad \texttt{free\_to\_state}\ (\texttt{end\_}\ js_1\ r)\ i\ ms_1\ =\ \texttt{SOK}\ (js_2)\ ms_2}{(xs,\ ms_1,\ js_1) \longrightarrow (js_2,\ ms_2)}$$

Figure 6.13 – The small-step semantic rules for the NASM JIT with primitive specification i

whose transitions, defined on Figure 6.11, are either atomic steps or a possibly infinite sequence of native code steps. In that definition, `Trace` is the type of observable events, and `State` is the type of state-machine states. Now, every transition of the JIT state machine can be represented with the type `nasm_transition jit_state trace`, meaning that any transition of the JIT is either an atomic transition represented by its free-monadic computation, or a nonatomic transition. Finally, the JIT state machine itself can now be represented with a Coq function `jit_step` of type `jit_step: jit_state -> nasm_transition jit_state trace`, replacing the previous signature introduced in Section 6.3 which only allowed atomic transitions.

Next, to give small-step semantics to a NASM, we specify the call to native code with three free-monadic computations `step_`, `start_` and `end_` as pictured on Figure 6.12. Moreover, we extend the JIT small-step semantics definition of Figure 6.7 with rules that execute `start_` when seeing a `LoadAndRun` transition (building an initial x86 semantic state), then loop `step_` until a final x86 semantic state is reached, and execute `end_` to get back to the next JIT state. This new small-step semantics definition for the JIT as a NASM is presented on Figure 6.13. Semantic states of the new semantics come in two shapes. When in a square state of Figure 6.12, semantic states are pairs (js, ms) containing a `jit_state` and a state of the monadic specification i (with a model of the shared data-structures). When in a round state of Figure 6.12, semantic states are (xs, ms, js) where xs is a x86 semantic state, ms is a state of i, and js is the `jit_state` that was reached before calling the native code. Rule (jit-step) of Figure 6.13 is the same rule as

previously defined on Figure 6.7. When a JIT transition is atomic, we simply use the monadic specification i to get a state monadic function corresponding to the transition, and execute it to obtain an observable trace t and the new semantic state. When a JIT transition consists in calling native code (`LoadAndRun`), we begin with rule (start), using the `start_` monadic function to construct an initial x86 semantic state xs of the called native program. `start_` is a monadic function using the primitive `Load_Code`, and returning the initial state of the x86 program according to its semantics defined in CompCert. Then, we follow the x86 semantics using rule (x86-step). The `step_` function is the small-step transition of x86, as defined in CompCert. We simply extend it so that every call to one of our primitives is specified with its monadic effect. Note that while executing x86, js is unchanged. This means that we trust the generated native code not to modify the current JIT data. For instance, we trust that our generated native code is not going to modify the current CoreIR functions, or even the current profiler state. Finally, rule (end) is used when x86 execution has reached a final state. We use the function `end_` and the return value of the native code to construct the next JIT state. In our implementation, this is always a `MONITOR` state. In the case of a diverging native execution, no final x86 semantic states is ever reached, and these new JIT semantics will remain in x86 semantic states. In that case, the entire JIT behavior of the small-step semantics shown in Figure 6.12 is diverging. Such behaviors are intended. The behavior of a JIT that compiles and executes a diverging function should be diverging.

Finally, we extend the OCaml `free_interpreter` of Figure 6.9 so that it loads and calls native code when seeing a `LoadAndRun` transition. As discussed in Section 8.2, we trust that our three monadic specifications `start_`, `step_` and `end_` correspond to what our modified free interpreter is doing when calling native code.

With these simple definitions unfolding native calls according to their CompCert x86 semantics, one can define small-step semantics for the entire JIT even in the presence of effectful and nonatomic transitions. The final JIT correctness theorem of Figure 4.7 is then modified to feature these new small-step semantics. The result is shown on Figure 6.14, where this version of `jit_sem` is the version shown in Figure 6.13 and includes both effectful and nonterminating transitions. To prove this new theorem, we once again prove the backward simulation shown on Simulation 11, replacing Simulation 1.

Backward Simulation **Mechanized**
Replaces the Simulation 1 for effectful JITs. Backward simulation relating the behavior of the JIT executing a program `p` to the behavior of the CoreIR semantics of `p`. `jit_sem` represents the JIT small-step semantics, including optimizations and nonatomic transitions. `prim_spec` is the monadic specification of the primitives.

```
Theorem jit_correctness_simulation:
∀ p,
  backward_simulation (CoreIR_sem p) (jit_sem p prim_spec).
```

Simulation 11 – JIT backward simulation (impure version)

```
Theorem jit_correctness:
  ∀ p beh, program_behaves (jit_sem p prim_spec) beh →
    ∃ beh', program_behaves (CoreIR_sem p) beh' ∧ behavior_improves beh' beh.
```

Figure 6.14 – Impure JIT correctness theorem, where `prim_spec` is a primitive specification.

6.8 Related Work on Impure Program Verification

Our work is not the first Coq mechanization about effectful programs. There are various ways to go around the limitations of Gallina and Coq extraction to pure OCaml. For instance, other Coq developments [Pit-Claudel et al. 2020; 2022] entirely avoid the extraction from Coq to OCaml to produce effectful verified programs. They directly transform a functional specification into a fully linked assembly program represented as a Coq term. This approach successfully removes Coq extraction from the trusted code base of a verified effectful program, but comes with slower compilation times and more restrictions on the input language. Modifying Coq extraction so that it can produce effectful OCaml programs has also been investigated for programs using mutable arrays [Sakaguchi 2018]. Programs are written using a state monad encoding, and the improved extraction produces efficient OCaml code using mutable data-structures. In contrast, our formalism allows us to use extraction to OCaml without modifications, but only translating the pure parts of the program. The Impure library [Boulmé 2021] has also provided a way to monadically encode nondeterministic functions in Coq as parameters that can be implemented with nondetermistic OCaml functions after extraction. It defines a calculus of weakest-preconditions to reason about such nondeterministic programs.

One way to reason about monadic encodings of effectful programs consists in defining

appropriate program logics. The Ynot project [Nanevski et al. 2008] also defines a Coq extension to reason about impure programs. Ynot has been used to formally verify a database management system for instance [Malecha et al. 2010]. Impure effects are represented using a monadic encoding and axiomatized with Hoare and separation logic predicates. Proofs can be conducted in these program logics. In several F* developments, the Dijkstra monad has been used to encode several kinds of effects and corresponding program logics. Effects can be specified with an ad hoc specification monad relating their postconditions and preconditions [Maillard et al. 2019]. In Coq, recent work [Nigron and Dagand 2021] has explored defining a separation logic to reason about freshness in monadic programs. In particular, this has been used to revisit an existing proof of CompCert in the `SimplExpr` module. In this experiment, defining the separation logic introduces a more complex framework (for instance, they include the entire MoSel framework [Krebbers et al. 2018]), but this increased complexity can then lead to a more concise proof than the existing one in CompCert. In our developments we do not define program logics or use preconditions and postconditions relations, and our primitives are instead specified with executable state-monadic functions, which facilitates the reuse of the CompCert simulation framework for our JIT small-step semantics.

Other verification work using monads has also been conducted in other proof assistants. For instance, a monadic approach is followed in Isabelle/HOL for the verified seL4 microkernel [Cock et al. 2008], in order to refine monadic functional specifications into a C implementation. Impure effects are also modeled with state-monadic computations. The main difference with our work lies in the way seL4 obtains verified executable programs. Instead of using program extraction, the approach followed by CompCert and our work, they write a specification of the program in a subset of Haskell that can be translated into Isabelle/HOL. Then, they rewrite the program in C and prove equivalence between the C implementation and the Isabelle/HOL specification [Klein et al. 2014]. Using program extraction allows us to avoid rewriting the pure parts of the JIT, which are automatically converted to OCaml, but requires trusting the extraction mechanism.

Other contributions have explored using variations of free monads in Coq before us, but our version was designed to be lightweight and compatible with CompCert. FreeSpec [Letan and Régis-Gianas 2020] uses a free monadic definition similar to ours to encode programs with effects, and uses this formalism to verify a minimal web server. Their definition is more general in the sense that one can compose several interfaces of primitives, each interface describing the interaction with a data-structure. We could imagine having several interfaces as well, such as one for the heap, one for the stack and one for the executable codes, instead we used a single

interface containing all three data-structures. Having multiple interfaces makes the free monad definition more complex but produces interesting lemmas for free, for instance automatically proving that using a stack primitive has no effect on the heap. Our monadic definitions mainly differ in the way primitives are specified. Our use of state monads allows to simply define small-step operational semantics *à la* CompCert.

Interaction Trees [Xia et al. 2020] are a coinductive variant of free monads. They have been used to define the semantics of a subset of LLVM [Zakowski et al. 2021], or to reason about *transactional objects*, a type of concurrent data-structures [Lesani et al. 2022]. Like in our approach, computations using interaction trees can be extracted to OCaml and executed using effectful event handlers, much like our `free_interpreter`, where events include the calls to effectful primitives. Their use of a coinductive structure allows interaction trees to represent diverging computations with some added monadic constructors, at the only cost of developing a library for coinductive reasoning. Coinductive reasoning can be difficult to work with in Coq. We entirely avoid this issue by breaking down what a JIT does into small, atomic computations (the transitions of Figure 4.3). Even possibly diverging transitions can be broken down themselves into small steps (Figure 6.12). Finally, our lightweight monadic library does not have the slightest coinductive proof but we rather reuse one of CompCert, going from a simulation to the `jit_correctness` theorem of Figure 6.14. Interaction Trees are an impressive framework for the verification of effectful diverging programs, but we chose to rely on existing CompCert proof techniques to model diverging behaviors.

Our refinement methodology is also reminiscent of many other work using refinement to prove the correctness of a concrete implementation using a more convenient abstraction. For instance, Isabelle includes a refinement framework for programs encoded in a nondeterminism monad [Lammich 2012]. This framework has been used to refine abstract formalizations of algorithms to more concrete implementations using efficient data-structures, for instance to verify network flow algorithms [Lammich and Sefidgar 2019]. In contrast, our refinement methodology is designed specifically for the novel interaction between the free monad formalism and the CompCert simulation framework we presented in this Chapter. Our refinement definition purposely resembles a CompCert forward simulation so that it can be used to produce a simulation (Simulation 10).

Chapter 7

Formal Verification of a JIT Backend Compiler

The task of producing native code from a language such as RTL or CoreIR is not a simple one. One must for instance allocate pseudo-registers to either real machine registers or the memory (*spilling* the registers). To produce fast code, register allocation requires complex algorithms and complex verification techniques [Rideau and Leroy 2010]. One must also move away from the CFG representation, to build a code representation where instructions are ordered linearly and nonbranching instructions are expected to jump to their successor. As seen in Chapter 3, the CompCert backend not only contains optimizations, but also contains formally verified algorithms to produce native code from RTL code, for various target architectures.

JITs with native code generation need similar algorithms: for a given function, they must create some equivalent assembly code. Modern JITs have reused techniques from ahead of time compilation, and even entire ahead of time compilers. Many JITs have used LLVM as a native backend, and LLVM even provides a JIT infrastructure, MCJIT [LLVM 2022], explaining its popularity among JITs. Examples of JITs using LLVM include the WebKit JIT until 2016 [WebKit 2014], the Mono .NET framework [Mono 2022], MCVM for Matlab [Chevalier-Boisvert et al. 2010] or the ŘJIT for the R programming language [Flückiger et al. 2020]. OSRKit [D'Elia and Demetrescu 2016] even explores the encoding of on-stack-replacement in LLVM. Before we developed the formally verified JIT backend presented in this Chapter, the evaluation of our first pure Coq JIT (shown in Section 8.3.3) also included LLVM as backend to generate native code from CoreIR. However, this backend was not part of the formal Coq model and, as a result, was unverified.

In this Chapter, we tackle the final JIT-specific verification challenge introduced in Section 1.2.1: reusing the formally verified optimizations and native code generation of a static compiler in a formally verified JIT. In particular, we reuse the backend compiler of CompCert and its proof, from RTL to x86-64 assembly. As explained in Chapter 3, CompCert includes

backends for other architectures. For simplicity, we only reuse the one for x86-64, although we believe that reusing the others should not generate new technical difficulties. We show that by slightly transforming CoreIR to use custom calling conventions, we can reuse the proof of the CompCert backend, in order to prove the correctness of the native code generation in a verified JIT. Intuitively, one could think that we can directly use this backend dynamically to transform some parts of the JIT program, and then because it generated "equivalent" code, the JIT execution semantics should not change. However, this is not as straightforward.

First, the CompCert backend correctness theorem is established by relating the semantics of an x86 program to the semantics of a RTL one. This means that our dynamic backend compilation step should be split into two passes: first generating a piece of RTL for a given function, then using the CompCert backend to generate some x86 from that RTL function. To prove these passes modularly, we need to reason about the intermediate program, where some RTL has been generated but not yet compiled. To reason about our backend compilation step modularly, we define mixed semantics in Section 7.2 that include semantic states for CoreIR, x86 but also RTL. These semantics are used to specify each step of the backend compilation of our JIT.

Second, the correctness theorem of CompCert states that the observable behavior of a program is preserved. Nothing in this theorem is related to the effects on the memory, that are indeed not preserved by the CompCert backend. In fact, the backend goes from an abstract stack in RTL (a list of RTL stackframes), to an actual execution stack allocated in the memory handled by CompCert. This is an issue for our impure JIT. We want to be sure that every modification to the heap done by each RTL function will be compiled to some x86 code that also modifies the heap similarly, otherwise executing the rest of the program after that function may differ. To avoid that issue while not modifying the code of the CompCert backend, we make the generated code interact with the JIT stack and the JIT heap only through external calls to 5 of the JIT primitives of Section 6.1.3: `HeapGet`, `HeapSet`, `Push`, `Pop` and `Print`. This means not relying on CompCert to compile any CoreIR function call, but instead generating RTL code that uses our primitives. We then define custom calling conventions relying on our primitives that the generated code uses. For instance, even though CompCert only compiles programs with no arguments, our solution consists in generating RTL programs that start by popping their arguments off the stack. These conventions also ensure that the generated native code always returns to the monitor on function calls, which may then decide to either execute or optimize the called function. In the end, the native programs we generate may use the CompCert memory to spill registers, however we do not use the stack and heap handled

by CompCert, but rather the shared data-structures handled by the JIT and all its components, and manipulated by the free monads of Chapter 6.

This clear separation between the stack handled by CompCert and the stack handled by the JIT using the primitives may produce nonoptimal code at times. For instance, live registers that are spilled on the CompCert stack may have to be also copied and saved on the JIT stack by our custom calling conventions on a function call. However, this allows us to reuse the CompCert backend proof without modifying any of the backend code itself. In the future, we could imagine adapting CompCert register allocation so that it directly uses the JIT stack using the `Push` and `Pop` primitives, removing these redundant copies.

Our backend compilation process consists in two steps. For a given CoreIR function to compile, we first generate several RTL programs. Then we compile each of them using the CompCert backend. In this chapter, we describe this compilation process and how we prove it correct in our formally verified JIT. We describe how we generate RTL programs from CoreIR functions in Section 7.1. In particular, we need to change the calling conventions to produce code that interact with the stack using our JIT primitives. Then, we define the mixed semantics in Section 7.2. These describe the behaviors of the programs our JIT manipulates, possibly containing pieces of CoreIR, x86 and RTL. In our nested simulation methodology presented in Chapter 4, these semantics are used for the internal simulations of Section 4.5.1. We can then prove correct the two main passes of our backend compilation JIT step. Section 7.3 proves a backward simulation for our RTL generation pass, and Section 7.4 shows how we can reuse the backward simulation of CompCert to prove a backward simulation for the pass that generates native code.

7.1 Splitting RTL Programs to Directly Reuse CompCert Proofs

CompCert only allows the compilation of complete programs and uses its own calling conventions. By generating several pieces of code that interact with the stack only through our JIT primitives, we demonstrate that both CompCert and its theorem can be used for formally verified native code generation in our JIT.

To reuse the CompCert backend with custom calling conventions that use the JIT stack, we split our CoreIR functions at function calls when generating RTL. The only JIT primitives that need to be called from that RTL code are `HeapGet`, `HeapSet`, `Push`, `Pop` and `Print`. The first two are used to interact with the heap. Stack primitives `Pop` and `Push` are used to save and restore

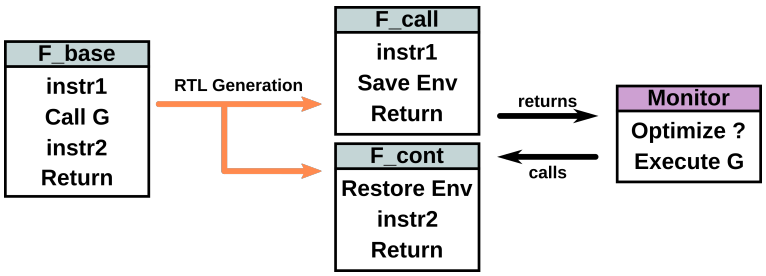

Figure 7.1 – Transforming CoreIR function `F_base` into two RTL functions `F_call` and `F_cont` using custom calling conventions to return to the monitor between calls.

the live environment, but also store return values or function arguments. The last one is used when compiling CoreIR functions with **Print** instructions so that the compiled code has the same observable behavior.

Figure 7.1 shows how to split each function `F` before compiling it. When our JIT decides to optimize `F`, it first splits its original version `F_base` in two functions: `F_call` corresponding to the beginning of the function, and `F_cont`, its continuation after the call. We add primitives to these functions that save and restore the environment (the live registers). Finally, `F_call` and `F_cont` can be compiled with a backend that preserves the primitive calls. Each one is defined as a whole RTL program. After optimization, the JIT will start by calling the compiled function `F_call`. When it encounters the call to function `G`, the generated native code returns to the JIT monitor, which may now optimize or simply execute `G`. When this call returns, the monitor calls the continuation function `F_cont`. This way of transforming CoreIR functions is reminiscent of writing code in a *trampolined* style [Ganz et al. 1999], where a function execution regularly comes back to a scheduler which then decides how to call its continuation. Trampolining has been used to interleave function executions, and here we use this transformation to interleave the execution of the generated native code with the execution of the other components of the JIT. At each function call, the monitor decides which JIT component to call (the optimizer, the interpreter or even another compiled function) before returning to the compiled function by executing a continuation program.

Figure 7.2 shows an example of a CoreIR function being compiled. The `Fun1` function does a computation, then calls another function `Fun7`, then does another computation and returns. When generating RTL, because there is only one call in `Fun1`, we split the function into two RTL programs. The first one (`$1`) starts by getting the function argument (`reg1`/`x8`) off the stack. After an instruction for the computation, it performs a call by first saving the live register `x8` on the stack. It then completes the current stackframe by pushing the identifier of the

```
Function Fun1 (reg1):
  reg2 ← reg1 + 4
  reg3 ← Call Fun7(reg2)
  reg3 ← reg1 + reg3
  Return reg3
```

(a) A CoreIR function

```
$1() {
// Getting call argument
  x8 = "Pop"()
  x9 = x8 + 4 (int)
// Saving live environment
  x1 = "Push" (x8)
// Completing the stackframe
  x1 = "Close"(1, 2)
// Pushing call argument
  x1 = "Push"(x9)
// Number of arguments
  x1 = "Push"(1)
// Called function identifier
  x1 = "Push"(7)
  x7 = RETCALL
  return x7
}

$2() {
// Getting return value
  x10 = "Pop"()
// Restoring live environment
  x8 = "Pop"()
  x10 = x8 + x10
// Pushing return value
  x1 = "Push"(x10)
  x7 = RETRET
  return x7
}
```

(b) Two Generated RTL programs corresponding to Fun1

```
# Generated by CompCert
$2:
leaq   32(%rsp), %rax
movq   %rax, 0(%rsp)
movq   %rbx, 8(%rsp)
call   _Pop
movq   %rax, %rbx
call   _Pop
leal   0(%eax,%ebx,1),%edi
call   _Push
movl   $RETRET, %eax
movq   8(%rsp), %rbx
addq   $24, %rsp
ret
```

(c) The x86 program for the continuation

Figure 7.2 – Compilation of a function `Fun1` by our JIT backend

current function and the label of the call (to identify the corresponding continuation function). `Close` is simply implemented by several calls to `Push`. After that, we push the call arguments, the number of arguments, and the identifier of the function we want to call (`Fun7`). Finally, we return with the constant `RETCALL`, returning to the monitor but indicating that the function wants to call another one. The monitor may now pop the function identifier and decide to optimize or execute `Fun7`. After the call to `Fun7`, its return value has been pushed to the stack. The execution then follows with the second program `$2`; it starts by getting the return value of `Fun7`, and restores the live register `x8` that was pushed earlier. Finally, it ends by returning another constant, `RETRET` to indicate to the monitor that the execution has finished. The CompCert backend then provably preserves the calls to JIT primitives, as seen on the x86 code produced for the second RTL program of Figure 7.2.

During the RTL generation pass, `Assume` instructions are compiled as branches. If the speculation holds, we proceed in the rest of the program. Else, we push the deoptimization metadata on the stack and return with another constant `RETDEOPT`. If deoptimization occurs, we return to the monitor which will read that data from the stack and reconstruct the corresponding interpreter state. Finally, we do not need to handle `Anchor` instructions when transforming CoreIR to RTL. These instructions were only used by the middle-end to insert speculation and are removed by the pass presented in Section 5.2.4 before calling the backend compiler.

7.2 Mixed Semantics: Interleaving Pieces of Executions Related to Multiple Languages

To prove the correctness of our two code generation passes of the JIT backend, we prove a simulation for each pass. As these passes temporarily generate RTL code, we need to define semantics for programs containing CoreIR, RTL, and x86 code. This allows us to reason about the intermediate state of the program between the two passes, where RTL has been generated but not yet compiled to x86. Note that such programs with pieces of RTL are never executed by the JIT, which waits until the backend compilation has entirely finished (if the second pass fails, the RTL is also discarded). The semantic states of these mixed semantics include the semantic states of each of these three languages semantics. To interface them, we also define three generic states forming a synchronization interface: CallState, ReturnState and DeoptState. Succinctly, each function call, return or deoptimization goes through shared synchronization states; this is shown on Figure 7.3. This closely mimics what the JIT state machine described in Chapter 4 does when it executes multiple languages. In essence, this mixed

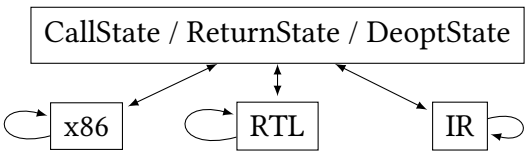

Figure 7.3 – Semantic states of the mixed semantics

semantics matches exactly the JIT behavior, except that it contains neither optimizing steps nor profiling but only execution. As shown in Section 4.5, using our nested simulation technique to prove the backend compiler correct using these semantics is enough to then prove correct the entire JIT with dynamic optimizations.

Figure 7.4 shows representative simplified rules of the mixed semantics. We omit it in the Figure, but all mentions of `free_to_state` use the reference specification `ref_spec` introduced in Section 6.6. The semantic states are pairs, of one mixed state of Figure 7.3 and one monadic state of the reference monadic specification (containing the JIT stack and the heap). By definition, the execution remains in a language according to its semantics until reaching a function call, a return or a deoptimization. For instance, in rule (step x86), we follow the execution of the x86 semantics. On any instruction that is not an external call, the monadic state is unchanged. There is a similar rule reusing RTL semantics, (step RTL). However, we extend both semantics with monadic rules when calling JIT primitives. For instance, in rule (push x86), if the current x86 instruction calls the primitive `Push`, we update the monadic state *ms* with the execution of the primitive specification. We move to the next x86 state `s'` with function `next_state`, moving to the return address and putting the return value in the right register. This definition resembles the external call semantics defined in CompCert. Along with (step x86), these rules form the modified x86 small-step semantics that we also use to specify the `step_` function of Section 6.7.

When the x86 or RTL semantics reach a final state, we move to the corresponding synchronization state. For instance in (x86 return), upon seeing the constant `RETRET`, we move to a ReturnState. In our x86 and RTL programs, the return value is pushed on the stack, so we move to a state that does not contain the return value itself, but rather some indication (`OnStack`) that it has to be popped. In rule (call x86), we see one way to go from a synchronization state to a x86 state. If we were about to call function `f` with some arguments `args`, and see that `f` had been compiled to some native program `p`, then we would push arguments to the stack and move to the initial semantic states of `p`, mimicking the behavior of the monitor and extending it to RTL executions. The free-monadic computation `push_args` simply corresponds to using the `Push` primitive on each argument of the list `args`. Just like `Push`, it returns an element of

$$\text{step x86}\ \frac{\texttt{s} \longrightarrow^{x86} \texttt{s'} \quad \text{not external call}}{\texttt{(s, ms)} \longrightarrow \texttt{(s', ms)}}$$

$$\text{step RTL}\ \frac{\texttt{s} \longrightarrow^{\text{RTL}} \texttt{s'} \quad \text{not external call}}{\texttt{(s, ms)} \longrightarrow \texttt{(s', ms)}}$$

$$\text{push x86}\ \frac{\begin{array}{c}\texttt{find\_instr(s) = Call Prim\_Push [v]} \\ \texttt{free\_to\_state (Prim\_Push v) ms = SOK retval ms'} \\ \texttt{next\_state (s, retval) = s'}\end{array}}{\texttt{(s, ms)} \longrightarrow \texttt{(s', ms')}}$$

$$\text{x86 return}\ \frac{\texttt{s} \longrightarrow^{x86} \texttt{Final RETRET}}{\texttt{(s, ms)} \longrightarrow \texttt{(ReturnState OnStack, ms)}}$$

$$\text{call x86}\ \frac{\begin{array}{c}\texttt{free\_to\_state (Prim\_Load\_Code f) ms = SOK p ms} \\ \texttt{free\_to\_state (push\_args args) ms = SOK tt ms'}\end{array}}{\texttt{(CallState f args, ms)} \longrightarrow \texttt{(initial\_state p, ms')}}$$

Figure 7.4 – Some representative rules of the mixed semantics

type unit `tt` but modifies the JIT stack contained in `ms`.

Mixed semantics include other similar rules for CoreIR and RTL. A final rule also steps from a ReturnState to a final semantic state if the stack is empty. While CallStates and ReturnStates step to any of the three languages, DeoptStates always reconstruct an interpreter state for CoreIR. Finally, all rules depicted on Figure 7.4 are silent and produce an empty trace. The only time an observable behavior is produced is when executing the primitive `Prim_Print`, either when calling it from x86 or RTL, or when interpreting a CoreIR **Print** instruction. These are the observable events preserved by the JIT execution.

Using this definition, we can define the small-step semantics of a program containing multiple languages. We write `mixed_sem (p, rtl, nc)` the small-step semantics, defined as in Figure 3.3 where the small-step transition relation is the mixed step partially presented in Figure 7.4. This corresponds to the semantics of a program which contains CoreIR functions in `p`, an optional set of RTL programs corresponding to a function in `rtl`, and installed x86 programs in `nc`.

Equivalence with CoreIR semantics These semantics define the behaviors of every program our JIT manipulates, including the temporary programs in-between the two backend compilation steps, where RTL has been generated but not yet compiled to x86 using the CompCert backend. The JIT correctness theorem however, as seen in Figure 6.14, uses the CoreIR

semantics (as defined on Figure 5.3) to specify the correctness of the JIT. These CoreIR semantics do not use monadic states or monadic computations, but are defined to be the simplest possible description of how a CoreIR program should behave. It is important to make sure that on simple CoreIR programs, the mixed semantics we defined here using monadic computations match the expected CoreIR semantics. To that end, we prove Simulation 12, showing that the behavior of a CoreIR program as defined by the mixed semantics matches its CoreIR semantics. Later, Section 8.1 shows how this proof helps us construct the final JIT correctness theorem of Figure 6.14.

Backward Simulation **Mechanized**

Relates the mixed semantics of a CoreIR program `p`, without any RTL (`None`) and without any x86 programs (`nocode`) to the CoreIR semantics of `p`.

```
Theorem input_mixed:
  ∀ p,
    backward_simulation (CoreIR_sem p) (mixed_sem (p, None, nocode)).
```

Simulation 12 – Equivalence of the CoreIR and mixed semantics on CoreIR programs

7.3 Correctness of RTL Generation

To generate native code from CoreIR, we proceed in two steps: first generating RTL, then transforming RTL programs into x86 ones. The first backend compilation pass generates several RTL programs for a given CoreIR function `F`: one program for the entry of `F`, and one continuation program for each function call in `F`. The generated code must contain the calls to JIT primitives that are going to be preserved by the CompCert backend and used by the native code. This means that this pass stops producing interpreter stackframes but instead uses native stackframes. Proving this pass correct then means proving correct the change of calling conventions in a function.

In practice, this first step is also conducted in two parts: we first generate RTLblock, a language we designed where labels of the CFG can be associated to basic blocks of RTL instructions (instead of single instructions like in RTL). Basic blocks in RTLblock contain a (possibly empty) list of nonbranching RTL instructions to be executed sequentially, then end on an *exit*

instruction. An exit instruction is either a simple jump (`Nop`) to the next basic block, a branch that points to two basic blocks, or a return. We then transform that RTLblock code into RTL code, unfolding the basic blocks. To that end, we extended the mixed semantics of Section 7.2 to also include the semantics of RTLblock. Using RTLblock as an intermediate language between CoreIR and RTL allows us to have simpler invariants, where every CoreIR instruction is matched with a single basic block of RTLblock. Both steps are proved correct with forward simulations that we can use to construct backward simulations (using Figure 3.8) since the mixed semantics of programs without `Anchor` instructions is deterministic.

Forward Simulation **Mechanized**

Proving the correctness of transforming a CoreIR function (with function identifier `fid`) into several RTLblock programs `rtlblock`. The CoreIR and x86 codes (`p` and `nc`) are unchanged by this transformation.

```
Theorem block_gen_correct:
  ∀ p nc fid rtlblock,
    rtlblock_gen fid p = OK rtlblock →
    forward_internal_simulation (p, None, nc) (p, Some rtlblock, nc).
```

Simulation 13 – Correctness of RTLblock generation

We first prove Simulation 13, relating the semantics of the current JIT program before and after adding new RTLblock programs corresponding to a CoreIR function. This theorem proves that splitting CoreIR functions and using our JIT primitives to interact with the JIT stack and heap preserves the behavior of the entire program. Building our invariant is crucial to proving the simulation. There are three main cases in the invariant for the transformation of a function `F` to RTLblock. First, because we are only compiling a single function, we need to relate identical semantic states when outside of that function (*refl*). However, even in that case, the execution stack can differ: some interpreter stackframes for `F` may have been replaced with equivalent native stackframes containing the live registers at the time of the call. Another possible case of the invariant (*rtlb*) happens when executing one of the new RTLblock programs (either the entry or one of the continuations). RTLblock semantic states are related to CoreIR semantic states. The two states must agree on live registers (not all registers, as only the live ones are restored after a call). Finally, the synchronization states (Callstate, Returnstate or Deoptstate) differ when reached from RTLblock or CoreIR. In RTLblock (but also in RTL and

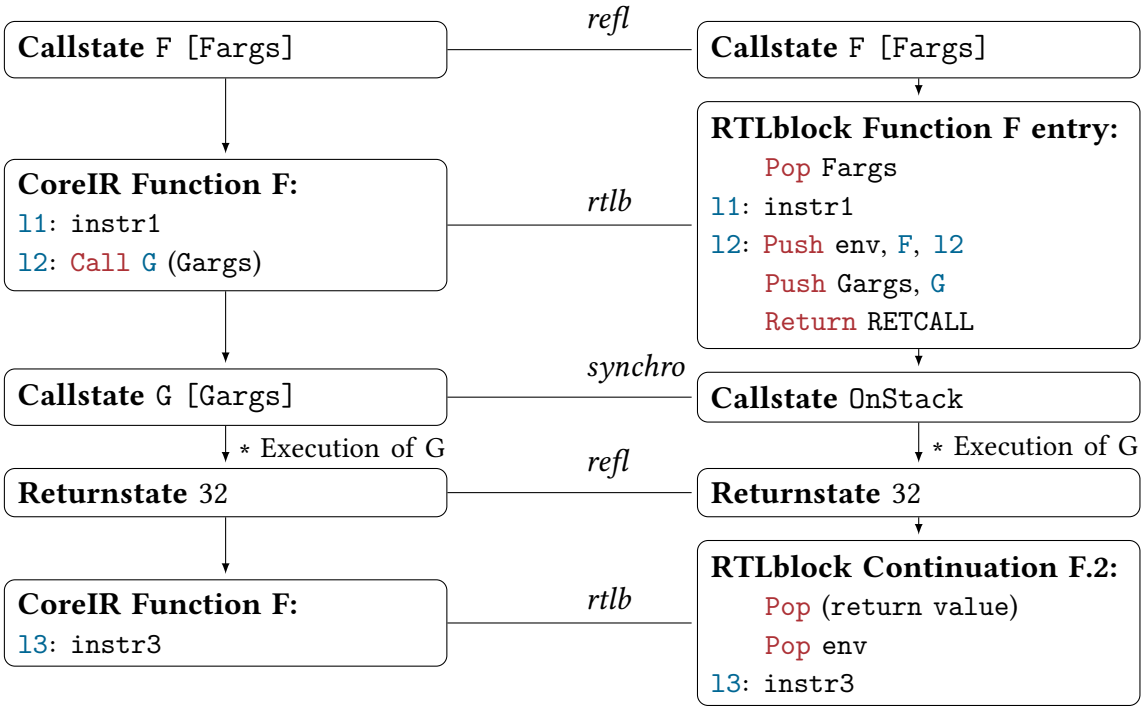

Figure 7.5 – Preservation of the invariant while transforming Function F to RTLblock

x86), the arguments of such synchronization states (like call arguments, the return value or the deoptimization metadata) have been pushed to the stack instead of directly given by the interpreter. A last invariant case *synchro* expresses that.

Figure 7.5 showcases an example of the invariant preservation in our simulation proof. The execution on the left corresponds to executing the program before F is transformed to RTLblock. Before calling the transformed function F, semantic states are related with the *refl* invariant. Then, we prove that the beginning of the execution of F in CoreIR matches its execution in RTLblock using the *rtlb* invariant, even though in RTLblock the arguments have to be popped first. As we see a function call to G (at l2), the CoreIR interpreter simply builds a Callstate containing the arguments. In RTLblock however, we need to push that to the stack and end on a different Callstate. Both are related with the *synchro* invariant. Execution of G then proceeds; when it returns after several steps, the source execution simply goes back in the middle of the CoreIR F. On the RTLblock side, we show that by going into the corresponding continuation program F.2 and popping the return value and environment off the stack, we get to semantic states matched with the *rtlb* invariant.

We then prove Simulation 14, replacing each RTLblock program with an RTL one. Since

Forward Simulation **Mechanized**
The `flatten` function transforms each RTLblock program into equivalent RTL programs.

```
Theorem flatten_correct:
  ∀ p nc rtlblock rtl,
  flatten rtlblock = OK rtl →
  forward_internal_simulation (p, Some rtlblock, nc) (p, Some rtl, nc).
```

Simulation 14 – Correctness of RTL generation from RTLblock

the language of RTLblock instructions is a strict subset of RTL instructions, the transformation is rather straightforward. The main complexity lies in transforming the CFG with fresh instructions labels for each instruction of the RTLblock basic blocks.

7.4 Correctness of Native Code Generation

Backward Simulation **Mechanized**
Relates the mixed semantics of `(p, Some rtl, codsrc)`, a program containing the RTL representation of a function, to the mixed semantics of `(p, None, codopt)`, the program where that RTL has been compiled to native and installed in the native codes. Monad states are pairs `(sh, codsrc)`, where `sh` is a model of the stack and heap, and `codsrc` and `codopt` model the executable memory.

```
Theorem native_gen_correct:
∀ p rtl asm sh codsrc codopt,
  rtl_backend rtl = OK asm →
  free_to_state (Prim_Install_Code asm) ref_spec (sh, codsrc) = SOK tt (sh, codopt) →
  backward_internal_simulation (p, Some rtl, codsrc) (p, None, codopt).
```

Simulation 15 – Correctness of native code generation

To prove correct the pass that uses the CompCert backend to transform RTL into x86, most of the effort lies in reusing CompCert simulations, relating x86 and RTL semantics, to construct a backward simulation on mixed semantics, as shown on Simulation 15. Note that this pass transforms at once each RTL program (one for the compiled function entry, and one

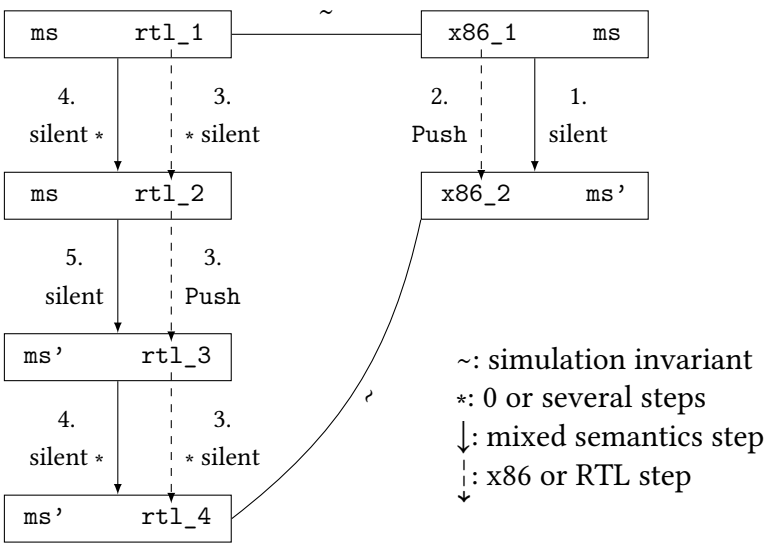

Figure 7.6 – Preserving a primitive call in the mixed semantics

for each continuation). The JIT then installs the resulting x86 programs using the `Install_Code` primitive, which modifies the monadic state to include the new executable programs.

We first define a simulation invariant that relates two states of the mixed semantics ($rtls$, ms) and ($x86s$, ms) where $rtls$ and $x86s$ are semantic states of respectively RTL and x86, related by a CompCert simulation. Since the programs have the same effects, the monadic state ms must be identical in both executions. Next, we prove a backward simulation: every semantic step of the mixed semantics after calling the CompCert backend is related to some steps of the mixed semantics before calling the backend.

For instance, Figure 7.6 illustrates one of the interesting cases of the proof: when the mixed step (1.) of the compiled program is calling the `Push` primitive (rule (push x86) of Figure 7.4). We prove that this corresponds to a step (2.) with an observable behavior in the x86 semantics. Then, using the CompCert backward simulation, we know that this step is matched by some steps in RTL semantics. These RTL steps emit the same behavior and can be split into three parts (3.) around the nonsilent step. We prove that silent RTL steps correspond to silent mixed steps (4.), and that the RTL call at state `rtl_2` corresponds to a silent mixed step (5.) where the monadic effect of `Push` has been applied, turning `ms` into `ms'` just like it did on the x86 side. Finally, we have proved that the single silent step on the right can be matched with steps on the left that have the same monadic effect and thus preserve the invariant. We then prove the entire backward simulation on mixed semantics. Outside of the compiled code, the invariant is simply reflexive.

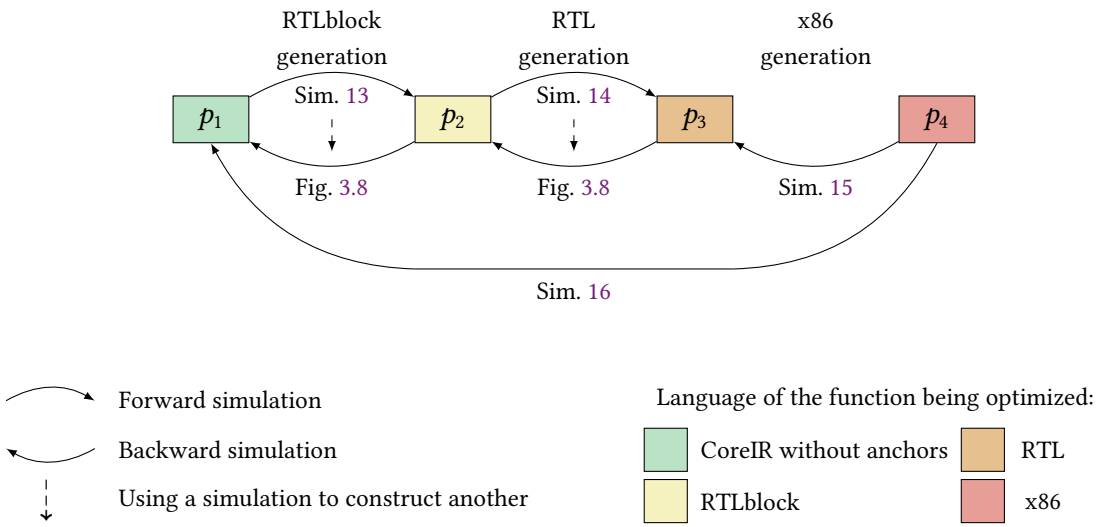

Figure 7.7 – Backend correctness

In summary, the CompCert backend specification provides us with a simulation stating that each x86 step is related to steps of its corresponding RTL program. In that simulation, external calls to JIT primitives like `Push` are part of the observable behavior, and are preserved by compilation. In contrast, only `Print` produces an observable event in the mixed semantics. This CompCert simulation is not exactly Simulation 15: what we want to build for the JIT backend to be correct is a simulation relating mixed semantics, the semantics of entire programs that not only contain the function being compiled, but also every other function of the current JIT program. We proved that we could construct this mixed semantics simulation by using the CompCert simulation exactly on the parts that were compiled.

7.5 A Formally Verified JIT Backend Compiler

Finally, the simulations we proved in this Chapter can be composed, as seen on Figure 7.7. The boxes p_1 through p_4 represent the different stages the current JIT program goes through during the backend compilation phase. The color of the boxes represent the language of the function being compiled. The other functions contained in the program can be in either CoreIR or x86 (these other functions may have been compiled by previous calls to the backend). Note that the two forward simulations used for proving the correctness of RTLblock and RTL generation can be used to construct backward simulations using the forward-to-backward theorem from CompCert (Figure 3.8), because the mixed semantics are determinate. The only nondeter-

minate instruction of CoreIR, `Anchor`, is removed by the middle-end compiler (see Section 5.2.4) before feeding a function to the backend compiler. The correctness of native code generation, Simulation 15, however reuses the backend simulation of the CompCert backend and is a backward simulation itself. The result of that simulation composition is the `backend_correct` theorem of Simulation 16, where `ps` is a profiler state that contains a compilation suggestion.

Backward Simulation **Mechanized**

Given a profiler state `ps` that can suggest functions to be compiled, this theorem relates the mixed semantics of a program before and after compiling some of its functions. The backend generates RTL temporarily, but the JIT program before and after calling the backend has no RTL (`None`) as it gets removed after code installation. `sh` models the JIT stack and heap, while `codsrc` and `codopt` model the executable memory and contain the x86 functions.

```
Theorem backend_correct:
  ∀ p ps sh codsrc codopt,
    free_to_state (jit_backend p ps) ref_spec (sh, codsrc) = SOK tt (sh, codopt) ⟶
    backward_internal_simulation (p, None, codsrc) (p, None, codopt).
```

Simulation 16 – JIT backend correctness

In that theorem, the free-monadic computation `jit_backend` uses its profiler state `ps` to get its suggestion, a function identifier. The corresponding CoreIR function is compiled as follows. `jit_backend` begins by calling `rtlblock_gen` to produce several RTLblock programs corresponding to that CoreIR function (proved correct with Simulation 13). Then it uses `flatten` to transform these programs into RTL programs (proved correct with Simulation 14). Then, it calls `rtl_backend` to use the CompCert backend and produce a x86 program for each of these programs (proved correct with Simulation 15). Finally, it uses the `Install_Code` primitive to install these x86 programs in the executable portion of the JIT memory. It does not return any value (its return value is `tt` of type `unit`), but that last step modifies the x86 codes of the JIT (`codsrc` becomes `codopt`). As a result, the `backward_internal_simulation` relates the mixed semantics of the CoreIR program `p` and its native codes before compilation, to the mixed semantics of the same program `p` where the new native codes have been installed. Note that on both sides of the simulation, the programs contain no RTL (`None`). As a result, the internal simulation used to specify the backend relates executions that only contain CoreIR and x86.

Additional Axioms to reuse CompCert The CompCert theorem only holds for complete programs, where every piece of code has been compiled as a whole. This is not the case of the functions compiled by our JIT with native code generation. We add three simple axioms in our Coq development to still reuse the CompCert theorem. The first one strengthens an existing axiom of CompCert, specifying that calls to our JIT primitives have a precise annotated behavior in CompCert semantics, leaving unchanged the memory handled by CompCert. We also assume that for each primitive and compiled function, there exists a place in CompCert memory where they have been allocated. In practice, primitives have been compiled outside of the memory modeled by CompCert, as part of the JIT C library. But the RTL semantics defined in CompCert, designed for traditional ahead of time compilation, do not handle this case. As a result, we axiomatize that primitives are allocated in the CompCert memory, but then edit the generated x86 code to jump to the actual addresses of the primitives. These axioms are not incomplete proofs, but consequences of the need of CompCert to model a view of the complete memory. This issue is reminiscent of separate compilation where several programs have different views of the memory, but is out-of-scope of our work. CompCert supports a kind of separate compilation using its `Linking` module. Currently we are not using this module to link our generated native code with the primitives but instead edit the generated code after compilation. Separate compilation is not a JIT-specific need, and its formal verification can involve new proof techniques [Stewart et al. 2015, Song et al. 2020] that we could investigate for our JIT as future work.

Last, we include a simple axiom that could be proved by unfolding CompCert code transformations: the CompCert backend does not generate any new built-in call that is not already in the RTL programs we create. We currently need this property because our `step_` function of Section 6.7 does not handle built-ins functions in the x86 semantics. This is due to the fact that such built-ins are the only part of the x86 semantics specified in a relational way in the CompCert definitions. We can either trust or prove this lemma, or modify our small-step semantics of Figure 6.12 to specify `step_` with a relation instead of a free monadic function. In our experiments, we have validated that the x86 code we generate does not contain any built-in function call.

Chapter 8

Assessment

In the previous Chapters, we have presented all our proof techniques for the formal verification of JITs. The result is a methodology for developing formally verified and executable effectful JITs with dynamic optimizations, speculation and native code generation. While the Coq JIT we have developed is minimal, we believe that our work establishes a foundation for the feasible verification of realistic JITs.

In this Chapter, we first present the modularity of our work in Section 8.1. In particular, we show how every proof and technique presented in this work can be composed together. Next, Section 8.2 presents the full *trusted code base* of our work, the code and assumptions that are not formally verified in our proofs and should either be trusted or verified using other methods. Finally, we discuss our Coq JIT implementation in Section 8.3.

8.1 Composing all Simulations

All our simulations and proof techniques have been designed in such a way that allows them to be combined. This is shown on Figure 8.1, where we see how all of our proofs can be combined for the formal verification of an impure JIT with dynamic optimizations, speculation and native code generation.

On the top of the Figure is represented the correctness of the JIT optimizer. An optimization step in our JIT design consists in calling the middle-end compiler, then calling the backend compiler. The boxes p_1 to p_{10} on the Figure represent the current program of the JIT as it gets transformed during such an optimization step of the JIT. Each transformation is proved correct with a backward internal simulation. JITs that generate native code interleave the execution of several languages, and as a result these backward simulations relate the mixed semantics (see Section 7.2) of the programs p_1 through p_{10}. Whenever possible, we reuse the forward-to-backward reasoning introduced in CompCert (Figure 3.8) to prove a forward simulation instead (which is easier to prove) and deduce a backward simulation for free. This can be done for any transformation that preserves every behavior of its source program and when the

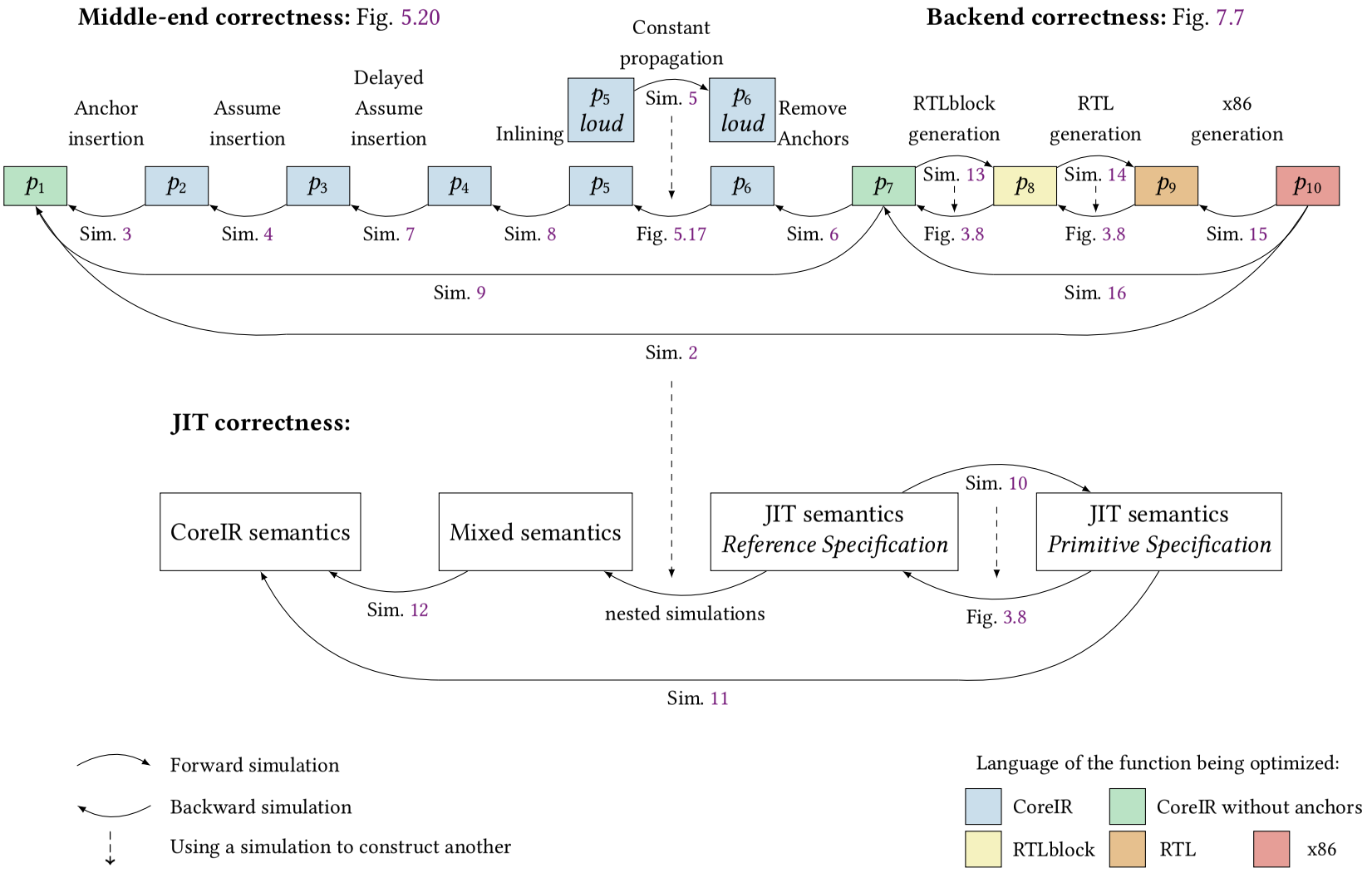

Figure 8.1 – Composing all our simulations for an effectful JIT with speculation and native code generation

target semantics are determinate.

The correctness of the middle-end compiler has been presented earlier in Figure 5.20. The order of the transformations can vary depending on what the profiler suggests, except for Anchor insertion and Anchor Removal. We depict here an arbitrary order showcasing all transformations of Chapter 5. Then, the correctness of the backend compiler has been shown earlier in Figure 7.7. Since Anchors have been removed when the backend compiler is called, the mixed semantics are deterministic which allows to prove Simulations 13 and 14 in a forward manner.

This results in multiple backward simulations that can be composed together. The proofs of Simulations 9, 16 and 2 simply consist in composing the backward simulations of their respective components. Then, using the nested simulation technique presented in Section 4.5.1, we can prove that there also exists a backward simulation between the mixed semantics of p (without dynamic optimizations) and the JIT semantics of p (adding dynamic optimizations). Because our JIT optimizer is proved correct with a backward simulation, this technique allows us to prove the correctness of a JIT that interleaves such optimizations with execution. All the previous optimizer proofs have used the reference specification of the primitives, which allows for simple proof invariants. Using our refinement methodology, we can prove Simulation 10, which relates the JIT semantics using the reference specification to the JIT semantics using the primitive specification, closer to the actual primitive implementation. Since the JIT has deterministic semantics, one can use the forward-to-backward methodology once again. As initially, the program executed by the JIT only contains CoreIR, we show using Simulation 12 that the CoreIR semantics of p match its mixed semantics. Doing so allows to relate the JIT semantics to CoreIR semantics which are simpler than the mixed semantics, making the final correctness theorem of the JIT, shown in Figure 6.14, easier to read and understand. The theorem does not mention mixed semantics, but simply states that behaviors of the JIT are related to behaviors of the CoreIR semantics of the program they execute.

The final simulation consists in composing the three bottom backward simulations of Figure 8.1. This is Simulation 11, a backward simulation between the JIT semantics executing a program p with the primitive specification, and the simple CoreIR semantics of p. Using a proof from CompCert, we can deduce the final behavior preservation theorem of the JIT, as seen on Figure 6.14. Our proof architecture leverages the power of the simulation framework of CompCert. Each optimizer transformation has been proved modularly, and our architecture could feasibly be extended with new optimizations or new features.

8.2 Trusted Code Base

While most of our JIT development is proved correct in Coq, there is still some code that has to be trusted in order to fully trust the OCaml executable JIT. Some of it is inherent to developments mechanized in Coq: for instance, one needs to trust the Coq kernel itself when it checks our proofs, and one needs to be convinced by the statement of our correctness theorem about behavior preservation (Figure 6.14). Similarly, some of it is inherent to running OCaml programs: one needs to trust the OCaml runtime and compiler. Below, we list the trusted code base that is specific to our development.

- The Coq to OCaml extraction of the JIT.
- The OCaml function that loops the JIT step (Figure 4.4 for instance) should correspond to the JIT semantics. This is a just a few lines of code repeatedly calling the free interpreter until reaching a final state.
- The free interpreter of Figure 6.9. This calls C functions from OCaml.
- The primitive implementations should correspond to their monadic specifications (Figure 6.8).
- The call to native code should correspond to the three monadic specifications `start_`, `step_` and `end_` of Section 6.7.

Our use of unsafe OCaml code for the free interpreter Our free interpreter definition, that needs to be trusted, uses the `Obj.magic` OCaml function. In Figure 6.9, this is used in the implementation of the `exec_prim` function. `Obj.magic` allows to cast any OCaml data to any type. It can be considered unsafe and dangerous because it allows to write OCaml code that entirely bypasses the type system and the guarantees it provides.

However, our current free-monadic approach requires bypassing the OCaml type system due to limitations of Coq extraction. The `primitive:Type -> Type` inductive type of Figure 6.3 defines all the primitives the JIT can use. Each primitive constructor conveniently includes the types that the primitive expects and the type that the primitive returns. For instance, `Prim_Push: int -> primitive unit` means that the primitives expects an `int` as argument and returns a value of type `unit`.

However, when this inductive type is extracted to OCaml, it results in an OCaml algebraic data type that does not contain the return type of each primitive. For instance, `Prim_Push` gets extracted to the type constructor `Prim_Push of Int.int`. This means that the `exec_prim` function that we mention in the free interpreter definition of Figure 6.9 cannot be typed correctly in OCaml without resorting to `Obj.magic`, because its return type should be exactly the return

type of the primitive it executes. This issue is not surprising as the inductive data types that can be defined in Coq are more expressive than the algebraic data types of OCaml. `Obj.magic` allows us to bypass the limitations of the OCaml type system, When auditing our free interpreter however, particular care must be given to checking that our primitive implementations return the correct types.

There are two possible ways to remove `Obj.magic` from our implementation. First, we could remove the `primitive` type from our Coq definitions. We would change the `free` type of Figure 6.3 so that there are as many variations of the type constructor `impure` as there are primitives. For instance, we would define `impure_Push (i:int) (cont: unit -> free T)`, to encode a free-monadic impure computation that starts by calling primitive `Push`. This solution extracts well to OCaml algebraic data types, but implies less elegant free monad Coq library and proofs.

But OCaml also includes GADTs (generalized algebraic data types) [Xi et al. 2003], that are more expressive than algebraic data types and expressive enough to represent the `primitive` type. If Coq extraction was able to generate such a GADT for the `primitive` type, we would have an implementation of the `free_interpreter` that does not use `Obj.magic`. We have validated this by manually modifying the extracted code to use a GADT. But until Coq extraction is able to automatically generate GADTs, this solution requires manual work every time we extract the JIT from Coq to OCaml.

Monadic specifications and primitive implementations The task of verifying that the primitive implementations comply with their monadic specifications is out of scope of our work. We have focused our efforts on proving correct exactly the parts of the JIT that are extracted to OCaml, given a specification of the rest. This orthogonal verification work could be done with the help of program logics. For instance, VST [Appel 2015] allows to prove that a C function implementation corresponds to a Coq specification, using Separation Logic. As future work, we could try using VST to prove correct our C primitives. We would first need to transform our state-monadic specifications into separation logic predicates.

Note that there are a few examples where the monadic specifications may not exactly match their implementations. One solution can be to use another primitive specification, even closer to the C implementation than `prim_spec`, then use the refinement methodology once again. For instance, consider the JIT stack of integers we use to communicate with the native code. We currently model this stack of integers with a Coq list, but we could refine that to an array and a counter which more closely resemble the C implementation. However, refinement cannot solve some issues that we discuss here. First, even in the `prim_spec` specification, the

stack is assumed to be infinite. In practice, we have implemented it as a finite array of 64-bit integers. This is not a JIT-specific issue, and CompCert also assumes to be working with an infinite memory. There exists a CompCert variant [Wang et al. 2019] that allows reasoning about bounded stack usage.

Second, in the formal model, the native code is represented by its x86 AST, just like in CompCert. In practice however, we call an assembler to generate machine bytes for the native code. These machine bytes are installed in the memory, not the x86 AST. This is not a JIT-specific issue, and we decided to go as far as CompCert goes in our formal model, at the cost of trusting the assembler.

Also, in its monadic specifications, the primitive that installs code never fails. In practice however, our installation could fail if we ran out of memory for the dynamically generated code. We could solve this issue by allowing the primitive specification to nondeterministically fail and in such cases cancel the optimization step. In our experiments so far, we have never encountered this issue.

Finally, we also need to trust that calling the native code is correctly specified with the three monads of Section 6.7, `start_`, `step_` and `end_`. The `step_` function simply reuses the CompCert x86 semantics, with an exception for primitives as seen on Figure 7.4. In our experiments, we did not find any bugs with the execution of native code linked with primitives.

8.3 Our Coq JIT implementation

The results presented in this work have been mechanized. In this Section, we present our implementation and its evaluation. As indicated in Chapter 4, we first developed a strictly pure JIT that featured dynamic optimizations and speculation in CoreIR. However, it contained no formally verified native code generation and used an interpreter to execute everything. Its correctness theorem is shown on Figure 4.7 and is obtained using Simulation 1.

This development was then extended with the support for impure computations and a formally verified backend, using the methods described in Chapter 6 and 7. Its correctness theorem is shown on Figure 6.14 and is obtained using Simulation 11. While developing this new version, some simplifications have been made. For instance, the proofs of the middle-end passes of Chapter 5 have to be adapted to the new mixed semantics. This adaptation does not require new correctness arguments and we can reuse the same simulation invariants, but it still requires some rewriting of the proof scripts. This was done for a minimal set of middle-end transformations to insert speculation: anchor insertion, assume insertion and anchor removal.

	Pure JIT	Impure JIT
Coq code	13 000	20 000
OCaml code	500	700
CompCert Coq code	9 500	16 000
CompCert OCaml code	0	46 000
CoreIR parser	400	400
C code	0	400

Figure 8.2 – Number of lines of codes of our JIT implementations

As future work, we could adapt the constant propagation, inlining and delayed assume insertion passes. The middle-end passes we already adapted to the impure setting show that it is possible to compose the correctness arguments of Chapter 5 to the ones of the following Chapters, as seen on Figure 8.1. Some simplifications to CoreIR have also been made in that new development to facilitate backend compilation. For instance, function calls now expect a list of CoreIR registers as arguments, instead of a list of expressions. This is closer to the RTL syntax. We have also removed the possibility of synthesizing extra stackframes in the deoptimization metadata, since we have not reimplemented inlining yet.

8.3.1 Implementation and Proof Reuse

Our JITs have been designed to be executable. To this end, we have not only developed Coq proofs of correctness, but also OCaml or C code that interacts with the extracted Coq code. Figure 8.2 presents the number of lines of code we wrote or reused in our JIT implementations. The Coq code category contains the code we wrote specifically for the JITs. This includes both functions and proofs. The OCaml code category consists in code that executes the extracted Coq code. This includes the `free_interpreter` of Figure 6.9 or our profiler.

Next, we have included a lot of Coq and OCaml code from CompCert. For the first pure JIT, we only included a few Coq libraries, for instance the simulation framework or the Kildall fixpoint solver. Our impure JIT however includes the entirety of CompCert 3.11 for the x86-64 target architecture. We believe that including the other CompCert target architectures would be possible. Some of that included code is not used by the JIT, like the frontend that generates RTL, but it was easier to integrate the entire project. Some of that superfluous code could however become useful when adding features to our JITs. For instance, we have also included the Flocq library for floating point arithmetic [Boldo and Melquiond 2011] that comes with CompCert 3.11, even though our JIT currently only works with integer values.

We have also used the Menhir OCaml parser generator library [Pottier and Régis-Gianas 2022] to write simple CoreIR parsers. The parser is the first thing used by our JITs, and it generates the AST representation of CoreIR that is used in our Coq development. Finally, we wrote some C code for our impure JIT, as discussed in Section 8.3.2.

8.3.2 Our C library of Impure Primitives

We wrote a C implementation for all the JIT primitives used by the impure JIT, that the OCaml free interpreter of Figure 6.9 can call. [1] Each of the stack and heap primitives uses a global array of 64-bit integers, and resembles its monadic specification (see Figure 6.8).

The primitives to install code are more involved and use system calls. The implementation of `Install_Code` allocates memory with `mmap`, writes into it after assembling the output of CompCert, then makes it executable but nonwritable with `mprotect`. JITs typically need this change of permissions to comply with the *write xor execute* security policy, expressing that in order to prevent some attacks memory pages cannot be both writable and executable. For instance, Firefox supports *write xor execute* protection for the code that is dynamically generated [de Mooij 2015].

We use the `elf` library [2] to get the binary code of the assembled file. Our C library of primitives is compiled with CompCert 3.11.

8.3.3 Evaluation

The JIT prototypes we presented have no ambition to be compared to industrial JITs used to execute real-world programs, but rather allow us to showcase our proof techniques to handle the four JIT-specific verification challenges of Section 1.2.1. Nevertheless, we can run our JITs on some example programs to validate our intuitions. In this section, we present some experiments to answer the following questions. Does the unverified code (see Section 8.2) behave as expected? Do our JITs trigger dynamic optimizations during execution? A JIT that would always call the interpreter could be correct, but we want to see our middle-end and backend compilers in action. Can we observe speedups due to the middle-end speculation? Can we observe speedups due to generating native code?

1. Available here: https://github.com/Aurele-Barriere/JIThm/tree/master/c_primitives.
2. Dcoumented here: https://man7.org/linux/man-pages/man5/elf.5.html.

```
local function fib(n)
  if n<2 then return n end
  return fib(n-1)+fib(n-2)
end
```

(a) Fibonacci in Lua Lite

```
Function fib(n_val, n_tag)
Version Opt:
    Assume (n_tag=3) F.1 [n_tag, n_val]
    Cond n_tag=3 l2 l1
l1: Call DynamicTypeError(...)
l2: ...
```

(b) Speculating that `n` is an integer in CoreIR

Figure 8.3 – Speculating on the type of Lua Lite variables

Evaluating the pure JIT One common usage of speculation in modern JITs is speculating on the type of the variables of the program they execute [WebKit 2020, Meurer 2017]. CoreIR is minimal by design and its registers only contain integer values, but it can be the target for a higher-level language with multiple types. To evaluate the practicality of CoreIR, we implemented an unverified frontend for Lua Lite, an ad hoc subset of Lua. This subset includes a subset of Lua values (nil, booleans, integers, tables with integer indices), no closures, no methods, and only programs where function call targets can be statically resolved. These choices are not fundamental. The frontend models Lua values as tuples of integers, where the first value holds a type tag and the second value the actual value. Speculating on the type of a Lua Lite variable then amounts to speculating on the value of the corresponding CoreIR register containing its type tag. In our experiments, we hand-craft profiler hints to speculate on these type tag registers.

An example is given on Figure 8.3, showing a Lua Lite function and the beginning of the corresponding CoreIR function after inserting speculation. The assumption `n_tag=3` (speculating that `n` is an integer) allows the subsequent constant propagation pass to remove all type checks from `n<2`, as well as `n-1` and `n-2`. We therefore observe a three-fold reduction of executed condition instructions in optimized code in this program.

Our pure JIT does not include a formally verified backend compiler, but the benefits of inserting speculative instructions are better observed in synergy with other optimizations. As a result, the final ingredient for our pure JIT evaluation is an unverified backend, which consists in a translation pass from CoreIR to LLVM IR. Equipped with the LLVM backend and its powerful optimizer, we were able to evaluate this extended JIT on two Lua Lite example programs. The first one is the Fibonacci example from Figure 8.3a and the second is an implementation of gnome sort. This implementation of the sorting algorithm is able to sort arrays where the contents are nil, booleans, or integers. In the benchmark we only pass arrays of integers and

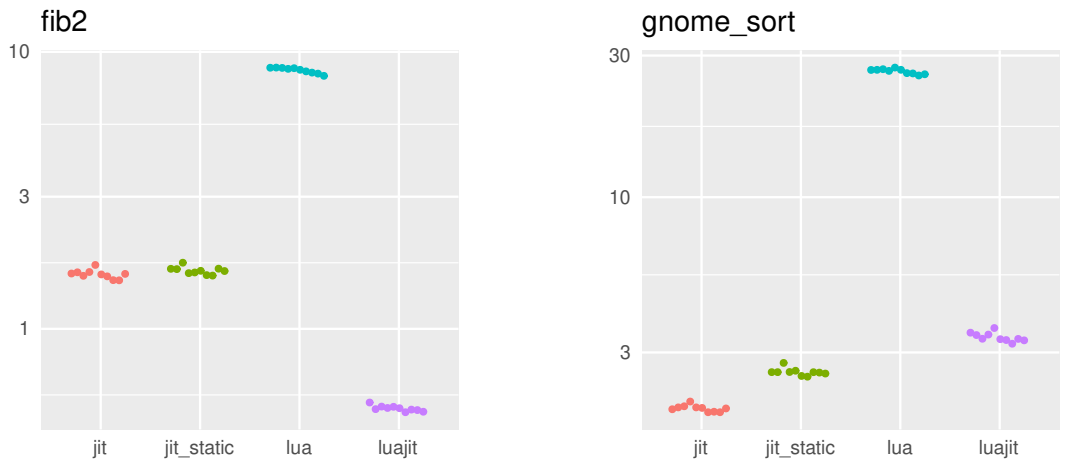

Figure 8.4 – Performance comparison of Lua Lite execution, runtime in seconds

we make the profiler speculate on this type.

The results are shown on Figure 8.4. We ran those benchmarks in the following configurations: (jit) the pure JIT with the unverified Lua frontend and the unverified native backend; (jit_static) the same as the former, but disabling speculative optimizations; (lua) the official Lua interpreter version 5.3.5; (luajit) the JIT compiler for the Lua language version 2.1.0. We ran the programs 10 times on a Laptop with a i7-7600U CPU at 2.80GHz, stepping 6, microcode version 0xc6 and 16 GB of RAM. The reported run times in seconds are measuring one execution of the whole process, including startup and compilation. In the case of `fib2` it consists of one call to `fib(39)`, in the case of gnome_sort sorting an array of length 46, 20000 times.

For Fibonacci we observe that the speculative optimizations do not yield large gains, despite removing all but a third of all type checks in the optimized version. The reason is that those type checks can be removed by common subexpression elimination as well, which in fact LLVM does, therefore the similarity in performance is as expected. For gnome sort, we observe a large gain by speculative optimizations. Here the speculation happens in a loop on values loaded from the heap and the data shows that a nonspeculating compiler such as our LLVM backend cannot optimize this code well without the speculation. In this light, we can see how our design for speculation in synergy with the LLVM optimizer can lead to speedups. If our pure JIT outperforms luajit in this example, this is expected since we do not have support for actual lua tables, but instead only for the subset that uses integer keys and can therefore be represented as arrays.

This evaluation of our pure JIT prototype shows that our approach to speculation can bring speedups to some program executions.

Evaluating the impure JIT While our impure JIT contains a formally verified backend compiler, it also contains unverified primitive implementations. Executing the impure JIT on some example programs helps us validate that these implementations behave as expected, and shows how native code generation can provide important speedups.

Modern JIT compilers generate native code dynamically to execute faster than simply using an interpreter and this is true of our impure JIT as well. For instance, we have used it to execute the program shown in Figure 8.5. This program prints the first prime numbers. Function `Fun1` calls `Fun2` for each value of `n` between 2 and 100000. Then, Function `Fun2` tests naively if `n` is prime and only prints its value if it is. The test is done with a loop checking that n is not divisible by any number i where $i \leq \sqrt{n}$. This program is the example program where we saw the most speedups from generating native code. Our impure JIT dynamically compiles `Fun2` after the first two calls, and the whole execution is 40 times faster than execution with only interpretation. Of course, these speedups are only as good as the profiler and real programs may require better heuristics. When compiling a function that is barely called after, like increasing the profiler threshold so that `Fun2` of Figure 8.5 is only compiled for its very last call, we can observe a small execution time overhead due to calling the optimizer for instance.

```
Function Fun1 ():
    n ← 2
l1: Cond (n < 100000) l2 l3
l2: x ← Call Fun2 (n)
    n ← (n + 1) l1
l3: Return n
```

```
Function Fun2 (n):
    i ← 2
l1: Cond ((i * i) < (n + 1)) l2 l4
l2: Cond (n % i) l3 l5
l3: i ← (i + 1) l1
l4: Print n
l5: Return 0
```

Figure 8.5 – Printing prime numbers in CoreIR

We wrote and executed other example programs to test every implementation of the primitives of Section 6.1.3. We also tested every possible case of our custom calling convention: an interpreted function calling a compiled function, a compiled function calling an interpreted function... We tried executing programs that already contain `Assume` instructions to show that our backend successfully handles deoptimizations, or that use the JIT heap. This evaluation of our impure JIT demonstrates that our implementation of the impure components matches its specification used in the proofs. Just like in modern JITs, generating native code in our Coq JIT can bring substantial speedups.

Chapter 9

Conclusion

9.1 Summary

Modern JITs are complex and large pieces of software, relying on advanced techniques. Such complexity comes with many opportunities for implementation or logic bugs. As a result, it is difficult to write a JIT, and even more difficult to write one correctly. Formal verification, while particularly time consuming, can provide the strong guarantee that a program does not contain bugs. In that perspective, the formal verification of Just-in-Time compilation appears essential.

There is however one key obstacle: if the formal verification of even simple programs can be challenging and time consuming, then the task of formally verifying a modern JIT like the ones used in web browsers seems insurmountable at first. They include many components and there is no existing formalization of their interplay. We believe that this work has taken valuable first steps towards the feasible verification of modern JITs. While our mechanized work simply consists in formally verified proof-of-concept prototypes, it provides key insights to make the formal verification of JITs look only as intimidating as the formal verification of ahead of time compilers.

In particular, we have taken special care to reuse both the proof scripts and the proof techniques of CompCert. This compatibility is important; there is a vast literature of formal verification work extending and reusing CompCert and some of them can be relevant to the optimizer part of a JIT. We have carefully selected JIT-specific features that separate JITs from ahead of time compilers, and provided new proof techniques to handle them. Our proofs are designed to be composed modularly, suggesting that our JIT design can reasonably be extended with new JIT features.

Formalizing JIT prototypes is insightful on its own. It defines clear specifications of their various components and demystifies the interplay between them. For instance, code transformations using speculative instructions are not trivial to write correctly. Even standard optimizations like inlining then require more complex invariants, such as the one shown on

Figure 5.19. Giving formal semantics to these speculative instructions and mechanizing some code transformations as in our formally verified middle-end compiler sheds some light on the way a JIT should handle speculation, deoptimization and on-stack replacement.

Chapter 4 has presented the architecture of our JIT design. This architecture orchestrates the various components that can be found in modern JITs. To the best of our knowledge, this is the first time a JIT formalization models all these components together. Our state machine representation is convenient to describe small-step semantics for our JIT, which then allow a JIT specification theorem to resemble the one of CompCert. Finally, Section 4.5 presents our solution to the first JIT-specific verification challenge: dynamic optimizations. This separates JITs from ahead of time compilers, because such static compilers theorems typically relate the behaviors of two known programs, while the program executed in a JIT changes along its execution. Intuitively, an invariant of the JIT execution should state that the current program executed by a JIT is equivalent in some sense to the original program. Our nested simulation technique expresses exactly that, while reusing the simulation framework of CompCert.

In Chapter 5, we have investigated the second JIT-specific challenge: speculative optimizations. Speculation has shown to be particularly impactful to execute dynamic languages faster, and many modern JITs speculate on some aspect of the execution. We have given formal semantics to speculative instructions resembling the ones used in modern JITs, and we have mechanized the correctness proofs of code transformations that insert and manipulate them. These proofs resemble the ones used in CompCert, and we can even reuse the time-saving forward-to-backward reasoning for some standard optimizations. While we could implement even more speculative optimizations in our middle-end, the transformations we selected (inserting anchor and assumes, constant propagation and inlining) are enough to generate speedups in some programs. Modern JITs need deoptimization when speculating, and to the best of our knowledge this work is the first mechanized implementation of this feature.

In Chapter 6, we presented our solution to the third JIT-specific challenge: proving the correctness of effectful JITs. Modern JITs require components that cannot be entirely written in the pure functional programming language of Coq. This includes installing and executing dynamically generated native code, or the manipulation of shared data-structures that must be modified by both the native code and the rest of the JIT. We want our JIT to perform these operations while still being formally verified in Coq. As a result, we need to represent our JIT in such a way that the correctness proof uses Coq specifications, but the JIT code can also be extracted to an OCaml program that calls effectful C implementations instead. We decided to use a methodology based on free monads that clearly delimit the impure parts of the JIT

from the rest that can be written in Coq. This led to small adjustments to the definitions and theorems presented in Chapter 4. Once again, our definitions are designed to be compatible with the CompCert simulation framework, and especially the refinement technique that we use to simplify proof invariants.

Chapter 7 then presented our approach to the final JIT-specific challenge: reusing already existing proofs of native code generation. Native code generation has already been mechanized in formally verified static compilers like CompCert. The task of formally verifying a JIT can be made substantially easier if one can reuse these existing proofs. Reusing the backend of CompCert had been an objective from the start, and motivated the use of its simulation framework in our solutions to the previous challenges. However, compiling functions of a JIT with the CompCert backend still required some work, as it is designed to compile entire programs. Our solution consists in splitting functions into several pieces of code that we can each feed to the CompCert backend. We have defined custom calling conventions that orchestrate these pieces of code with the rest of the JIT execution, and support deoptimization of native code. This results in a formally verified JIT reusing the CompCert backend to dynamically generate native code.

Finally, we have shown in Chapter 8 that all the techniques presented in this work can be composed together. One main objective of our work was to produce not only an abstract models of JITs, but also an executable JIT that can run actual programs. We presented our Coq JIT implementations that can execute programs written in a minimal language, but also come with a Coq proof of correctness. Our implementation captures essential features of modern JITs, but one can imagine even more features to add to our existing work. In Section 9.2, we present potential short and long-term improvements to our work.

9.2 Perspectives

While our methodology for formally verified JITs successfully handles the four JIT-specific verification challenges of Section 1.2.1, we could improve it in various ways. In this section, we present possible avenues for amelioration that we have not yet implemented or verified.

For instance, we could try to write better profiling heuristics. Currently, our prototype profilers simply record the number of each function call, the values of the functions arguments, and we provided in our pure JIT implementation a way to manually annotate CoreIR programs to tell the profiler what to record at a given program point. We used such annotations to speculate on the types of Lua Lite variables (see Section 8.3.3). Profiling in modern

JITs however is completely automatic and usually more closely tied to interpretation. For instance, in JavaScriptCore, the interpreter directly modifies some counters (called *case flags* or *case counts*) that the profiler uses to suggest speculation [WebKit 2020]. We could modify our design such that the interpreter itself calls external profiling parameters as it executes a CoreIR function and try to model the behaviors of modern JIT profilers. As explained in Section 4.3, this would only impact performance, and no modification to our proofs would be needed. A more realistic profiler would allow us to conduct a better performance evaluation of our design and implementation. We could then measure more precisely the benefits of different speculative optimizations, and their interactions with other optimizations contained in the backend compiler.

Our middle-end compiler could also be extended. For instance, we could avoid copying the deoptimization metadata from anchors to assumes by having a dedicated instruction to contain this metadata that other speculative instructions can refer to. This would correspond to the Framestate node used in Graal for instance [Duboscq et al. 2014], and we could insert it with anchors during the first middle-end pass (Section 5.2.1). Also, our delayed `Assume` insertion of Section 5.2.5 allows to insert speculations at different points while using a single `Anchor`. Keeping the number of anchors down is convenient, because as explained in Section 5.1 anchors may increase register pressure and reduce the efficiency of other optimizations. We could extend this delayed Assume insertion pass to allow other instructions than branches between assumes and anchors, which would allow a single anchor to be used for even more assumes. Combined with a standard dead code elimination pass, we could then recreate the motivating example of Figure 5.1. Other contributions have also investigated different ways to manipulate speculative instructions. For instance, reordering the speculation checks can bring execution speedups [Odaira and Hiraki 2005], which also corresponds to the predicate hoisting used in Sourir [Flückiger et al. 2018]. It would be valuable to provide formally verified implementations for such optimizations.

Our mechanized work could also benefit from some proof engineering improvements. In particular, we ended up having several versions of the CompCert small-step semantics and simulation libraries, identical except for a few differences. For instance, the original framework from CompCert is used to prove the backend compiler, but we also have another version for the internal simulations that we use to specify the correctness of the entire JIT optimizer step. The original version contains a few features that are tied to the C language semantics. In this version, small-step semantics for instance contain a *symbol environment* that we do not need when describing our JIT semantics. Also, the internal simulations differ from the ones in

CompCert when matching initial states, as discussed in Section 4.5.3. Instead of using a modified copy of the libraries, it would be better to write a more general library, of which both copies are instantiations. In particular, our work has shown that the small-step and simulation framework of CompCert can be used for other purposes than compiling the C language, and it could be valuable to have a generic library available for other verified program transformations.

9.2.1 Recompilation and Contextual Dispatch

Currently in our proof methodology, a JIT cannot remove the optimized version of a function. For instance, one cannot remove the native code that has been installed using the primitive `Install_Code`, and one cannot remove the current `Opt` version of a CoreIR function (the middle-end can modify it using its various passes, but not remove it). However, removing previously optimized version of a function can be helpful in JITs with speculative optimizations. If prior speculations turn out to be wrong too often, a JIT can recompile the function from scratch using new speculations [WebKit 2020].

This limitation of our prototypes comes from the fact that we prove correct our JIT optimizer step with a backward simulation on deterministic semantics. When calling functions our JIT has a deterministic behavior, it always calls the most optimized function. As a result, when the optimizer added a new native code version of a function, our nested simulation invariant simply guarantees that the new program, calling the new native codes, is simulated with the original program of the JIT. But our invariant says nothing about executing the same program that would however execute the original CoreIR version of the function instead of the native code. If we knew that this behavior was also simulated with the original program behavior, we could safely remove the native codes. According to our current invariant, the only correct way to return to the original version is by deoptimizing.

One simple way to extend our methodology would be to define the mixed semantics such that it chooses nondeterministically between the different versions of a function. As a result, removing an optimized version could be proved with a backward simulation, as it simply consists in removing one of the multiple possible behaviors of the mixed semantics. Just like speculation needed nondeterministic semantics to represent the possibility of deoptimization, we would use nondeterministic semantics to represent the possibility of choosing either version of a function. And we could reuse the *loud* semantics principle of Section 5.3.3 to enable the forward-to-backward reasoning, adding observable events to function calls. It would be interesting to generalize the loud semantics methodology to formally verify recompilation in

our JIT.

We also believe that nondeterministic semantics for function calls could be a simple way to implement and verify the *contextual dispatch* approach mentioned in Section 2.2, another approach to speculation (instead of speculation points like `Assume` and `Anchor` inside of the function code). One could extend our JIT design so that functions can hold multiple optimized versions, such that each version has been specialized for a particular call context (for instance, a given value of its arguments). Each version would be annotated with a CoreIR expression describing the call context for which it is specialized, just like the *context predicates* used in the Ř JIT for contextual dispatch [Flückiger et al. 2020]. Contextual dispatch requires simpler invariants from the JIT execution, it suffices to know that the JIT can call any version compatible with the current call context. To capture this, the mixed semantics would nondeterministically choose between all the versions that are allowed given the current call context. The proof of adding a new specialized version could then resemble the proof of inserting an `Anchor` instruction. On function calls, the monitor of the JIT would evaluate the expressions annotating each version to determine which versions are correct in the current call context, and choose the most specialized one. While contextual dispatch only allows specialization to happen at the beginning of a function (instead of anywhere in its code with the `Assume` instruction), it would allow different specializations to coexist in our JIT instead of the single optimized version we have currently.

In summary, we could further investigate the benefits of using nondeterminism to represent speculative JIT invariants. This would allow to verify both recompilation and contextual dispatch, speculative techniques that are used in modern JITs, but that our current work has not formalized yet.

9.2.2 Direct Calls and Builtins

One may wonder if our synchronization interface, going back and from the monitor at each function call, can be a bottleneck for execution. In our pure JIT experiments, we used an unverified optimization when compiling a call to another already compiled function: we asked the LLVM backend to generate a direct call to that function, without going back to the monitor. This means that going back to the monitor is only needed when going to the interpreter is needed, and execution can stay at the native level as long as possible.

We believe that a similar optimization with our verified backend could be possible to implement and verify. When producing RTL code, we could compile function calls as branches. We could ask the RTL code to use the primitive `Check_Installed` to see if the function to call

has been compiled already. If the function has not been compiled, we would return to the monitor as described in Section 7.1, pushing live registers to the JIT stack and returning the `RETCALL` value. If however the function is compiled, we could instead generate a RTL program that calls the native function at a specific address, as returned by primitive `Load_Code`.

This would require some modifications in our mixed semantics definition, as currently only calls to primitives are allowed in the RTL and x86 code we generate. In practice, this would mean using a mix of our custom calling convention when returning to the monitor, and CompCert calling conventions when doing such direct calls. As a result, for the correctness of RTL generation, we would need an invariant where part of the JIT execution stack is found in the RTL semantic state. While this is probably a nontrivial implementation and proof, we believe that this could be possible in our impure JIT, where we have already formalized and mechanized native code generation and execution.

Similarly, using external calls in the native code for each stack and heap interaction could be detrimental to execution times, although we have not yet evaluated this. Using these external calls allowed us to define a clear interface of the impure effects of the JIT. One can imagine possible optimizations, either inlining x86 implementations of the primitives as an additional step of the backend, or defining custom builtin functions at the RTL level that CompCert can also inline and compile. Having defined semantics for native code execution in a JIT compiler, such optimizations could be proved correct in our impure JIT.

In summary, interactions between the dynamically generated code in JITs and their other components represent a large area of design and optimizations for JITs. For instance, the V8 engine has recently changed the way native code calls builtins functions, to better exploit the branch prediction mechanism used by microprocessors [V8 2021a]. Our verified JIT methodology has presented a simple interface that could be modified or extended to model and reason about such low-level interactions in JITs.

9.2.3 Formally Verified JITs for Realistic Languages

CoreIR is a language that we designed to be simple enough to ease the burden of formal verification, but still allowing us to showcase interesting features of modern JITs. It comes with an interpreter, is close enough to CompCert RTL to make compilation easier, and has support for speculative optimizations with dedicated instructions. We believe that all the correctness arguments that we presented in our various proofs could also be adapted to a more realistic language. Dynamic languages like JavaScript or Python are typical examples of languages for which just-in-time compilation is used. While the JSCert project [Bodin et al. 2014] includes

a Coq interpreter for JavaScript, the lack of existing formally verified Coq compilers for these languages would slow down the task of reusing our work for these languages. But any language equipped with a Coq interpreter and a Coq compiler could use our work to develop a formally verified JIT.

For instance, recent work has formalized in Coq the semantics of WebAssembly [Watt et al. 2021]. This work describes an on-going work on a Coq interpreter for WebAssembly, and mentions that one could possibly link that mechanization of the semantics to a CompCert intermediate language, in order to have formally verified compilation. If these two tasks are completed, one could reuse the techniques we developed to obtain a formally verified JIT for WebAssembly. This would however require additional work. For instance, adapting the custom calling convention to save and restore the new WebAssembly interpreter stackframes. Similarly, a translation from full WebAssembly programs to full RTL programs (or any other intermediate language of CompCert) is not enough. In order to use the JIT calling conventions, one would also need to split the functions at function calls, like we did in Section 7.1. WebAssembly is a realistic language used extensively on the Web, and major browsers engines such as V8, JavaScriptCore, SpiderMonkey and ChakraCore include JITs not only for JavaScript but also for WebAssembly.

Appendix – Relating the Development to the Dissertation

All definitions and proofs presented in this document have been implemented and mechanized. The electronic version of this Chapter contains links to the relevant definitions. We have presented two JIT implementations, available here:
https://github.com/Aurele-Barriere/CoreJIT
https://github.com/Aurele-Barriere/JIThm.

The first one is the pure JIT containing the full middle-end optimizer, but not the formally verified backend. The second one is the impure JIT containing the formally verified backend optimizer. As explained in Section 8.3, it contains some but not all of the middle-end optimizer of CoreJIT. We provide links to both developments when applicable.

Chapter 4

Figure 4.4	Pure JIT OCaml loop	Pure JIT	
Algorithm 1	The `jit_step` function	Pure JIT	Impure JIT
Figure 4.5	Pure JIT small-step semantic rule	Pure JIT	
Figure 4.6	Profiling parameters	Pure JIT	Impure JIT
Section 4.3	Profiler implementation	Pure JIT	Impure JIT
Simulation 1	Pure JIT backward simulation	Pure JIT	
Figure 4.7	Pure JIT correctness theorem	Pure JIT	
Figure 4.9	Backward internal simulation	Pure JIT	Impure JIT
Figure 4.9	External simulation invariant	Pure JIT	Impure JIT
Section 4.5.2	Nested simulation measure	Pure JIT	Impure JIT
Simulation 2	Optimizer backward simulation	Pure JIT	Impure JIT

Chapter 5

Chapter 6

Figure 6.5	Monadic specification	Impure JIT
Figure 6.5	`get_prim`	Impure JIT
Figure 6.6	`free_to_state`	Impure JIT
Figure 6.7	Atomic small-step rule	Impure JIT
Figure 6.8	Heap access (C implementation)	Impure JIT
Figure 6.8	Heap access (monadic specification)	Impure JIT
Figure 6.9	Free interpreter	Impure JIT
Figure 6.9	`exec_prim`	Impure JIT
Section 6.6	Primitive specification `prim_spec`	Impure JIT
Section 6.6	Reference specification `ref_spec`	Impure JIT
Section 6.6	`prim_spec` ≃ `ref_spec`	Impure JIT
Figure 6.10	`ref_state`	Impure JIT
Figure 6.10	`prim_state`	Impure JIT
Section 6.6	Refinement definition	Impure JIT
Simulation 10	Refinement theorem	Impure JIT
Figure 6.11	NASM transitions	Impure JIT
Figure 6.13	NASM semantics	Impure JIT
Section 6.7	Impure `jit_step` function	Impure JIT
Section 6.7	Specifications `step_`, `start_` and `end_`	Impure JIT
Simulation 11	Impure JIT backward simulation	Impure JIT
Figure 6.14	Impure JIT correctness theorem	Impure JIT

Chapter 7

Simulation 12	Equivalence of CoreIR and mixed semantics	Impure JIT
Section 7.1	JIT primitives that are external calls in the native code	Impure JIT
Figure 7.3	Mixed semantics states	Impure JIT
Figure 7.4	Mixed semantics rules	Impure JIT
Section 7.3	RTLblock syntax	Impure JIT
Section 7.3	RTLblock semantics	Impure JIT
Section 7.3	Generating RTLblock	Impure JIT
Simulation 13	Correctness of RTLblock generation	Impure JIT
Figure 7.5	Invariant of RTLblock generation	Impure JIT
Section 7.3	Generating RTL from RTLblock	Impure JIT

Chapter 8

Bibliography

AbsInt (2015). CompCert release 15.10. https://www.absint.com/releasenotes/compcert/15.10/.

Appel, A. W. (2015). Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31. https://doi.org/10.1145/2701415.

Aycock, J. (2003). A Brief History of Just-in-Time. *ACM Comput. Surv.* http://doi.acm.org/10.1145/857076.857077.

Barrière, A., Blazy, S., Flückiger, O., Pichardie, D., and Vitek, J. (2021). Formally Verified Speculation and Deoptimization in a JIT Compiler. *Proc. ACM Program. Lang.*, (POPL). https://doi.org/10.1145/3434327.

Barrière, A., Blazy, S., and Pichardie, D. (2020). Towards Formally Verified Just-in-Time Compilation. CoqPL.

Barrière, A., Blazy, S., and Pichardie, D. (2023). Formally Verified Native Code Generation in an Effectful JIT or: Turning the CompCert Backend into a Formally Verified JIT Compiler. *Proc. ACM Program. Lang.*, (POPL). https://doi.org/10.1145/3571202.

Bebenita, M., Brandner, F., Fähndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Venter, H. (2010). SPUR: a Trace-based JIT Compiler for CIL. In Cook, W. R., Clarke, S., and Rinard, M. C., editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 708–725. ACM. https://doi.org/10.1145/1869459.1869517.

Bezanson, J., Chen, J., Chung, B., Karpinski, S., Shah, V. B., Vitek, J., and Zoubritzky, L. (2018). Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.*, 2(OOPSLA):120:1–120:23. https://doi.org/10.1145/3276490.

Blazy, S. and Leroy, X. (2009). Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reason.*, 43(3):263–288. https://doi.org/10.1007/s10817-009-9148-3.

Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., and Smith, G. (2014). A Trusted Mechanised JavaScript Specification. In Jagannathan, S. and Sewell, P., editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 87–100. ACM. https://doi.org/10.1145/2535838.2535876.

Boldo, S. and Melquiond, G. (2011). Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In Antelo, E., Hough, D., and Ienne, P., editors, *20th IEEE Symposium on Computer Arithmetic, ARITH 2011*, pages 243–252. IEEE Computer Society. https://doi.org/10.1109/ARITH.2011.40.

Boulmé, S. (2021). *Formally Verified Defensive Programming (Efficient Coq-Verified Computations from Untrusted ML Oracles)*.

Brock, J., Ding, C., Xu, X., and Zhang, Y. (2018). PAYJIT: Space-Optimal JIT Compilation and its Practical Implementation. In Dubach, C. and Xue, J., editors, *Proceedings of the 27th International Conference on Compiler Construction, CC 2018*, pages 71–81. ACM. https://doi.org/10.1145/3178372.3179523.

Brown, F., Renner, J., Nötzli, A., Lerner, S., Shacham, H., and Stefan, D. (2020). Towards a Verified Range Analysis for JavaScript JITs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*, pages 135–150. ACM. https://doi.org/10.1145/3385412.3385968.

Chevalier-Boisvert, M., Hendren, L. J., and Verbrugge, C. (2010). Optimizing Matlab through Just-In-Time Specialization. In Gupta, R., editor, *Compiler Construction, 19th International Conference, CC 2010*, volume 6011 of *Lecture Notes in Computer Science*, pages 46–65. Springer. https://doi.org/10.1007/978-3-642-11970-5_4.

Click, C. and Cooper, K. D. (1995). Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196. https://doi.org/10.1145/201059.201061.

Cock, D., Klein, G., and Sewell, T. (2008). Secure Microkernels, State Monads and Scalable Refinement. In *Proc. of TPHOLs 2008*, volume 5170 of *LNCS*, pages 167–182. Springer. https://dl.acm.org/doi/10.1007/978-3-540-71067-7_16.

Coinbase (2019). Responding to Firefox 0-days in the Wild. https://blog.coinbase.com/responding-to-firefox-0-days-in-the-wild-d9c85a57f15b.

Coq (2022). *The Coq Proof Assistant Reference Manual*. Inria. Version 8.12.1.

de Mooij, J. (2015). W xor X JIT-code enabled in Firefox. https://jandemooij.nl/blog/wx-jit-code-enabled-in-firefox/.

D'Elia, D. C. and Demetrescu, C. (2016). Flexible On-Stack Replacement in LLVM. In Franke, B., Wu, Y., and Rastello, F., editors, *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016*, pages 250–260. ACM. https://doi.org/10.1145/2854038.2854061.

Delmas, D. and Souyris, J. (2007). Astrée: From Research to Industry. In Nielson, H. R. and Filé, G., editors, *Static Analysis, 14th International Symposium, SAS 2007*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer. https://doi.org/10.1007/978-3-540-74061-2_27.

Duboscq, G., Würthinger, T., and Mössenböck, H. (2014). Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler. In Kolodziej, J. and Childers, B. R., editors, *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*, pages 187–193. ACM. https://doi.org/10.1145/2647508.2647521.

Duboscq, G., Würthinger, T., Stadler, L., Wimmer, C., Simon, D., and Mössenböck, H. (2013). An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In Bockisch, C., Haupt, M., Blackburn, S., Rajan, H., and Gil, J., editors, *VMIL@SPLASH '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 1–10. ACM. https://doi.org/10.1145/2542142.2542143.

eBPF (2022). eBPF. https://ebpf.io/.

Filliâtre, J. (2012). Verifying Two Lines of C with Why3: An Exercise in Program Verification. In Joshi, R., Müller, P., and Podelski, A., editors, *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012*, volume 7152 of *Lecture Notes in Computer Science*, pages 83–97. Springer. https://doi.org/10.1007/978-3-642-27705-4_8.

Fink, S. J. and Qian, F. (2003). Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In Johnson, R., Conte, T., and Hwu, W. W., editors, *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003)*, pages 241–252. IEEE Computer Society. https://doi.org/10.1109/CGO.2003.1191549.

Firefox (2022a). Security Advisories for Firefox. https://www.mozilla.org/en-US/security/known-vulnerabilities/firefox/.

Firefox (2022b). SpiderMonkey. https://firefox-source-docs.mozilla.org/js/index.html.

Flückiger, O., Chari, G., Jecmen, J., Yee, M., Hain, J., and Vitek, J. (2019). R Melts Brains: an IR for First-Class Environments and Lazy Effectful Arguments. In Marr, S. and Fumero, J., editors, *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2019*, pages 55–66. ACM. https://doi.org/10.1145/3359619.3359744.

Flückiger, O., Chari, G., Yee, M., Jecmen, J., Hain, J., and Vitek, J. (2020). Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.*, 4(OOPSLA):220:1–220:24. https://doi.org/10.1145/3428288.

Flückiger, O., Scherer, G., Yee, M., Goel, A., Ahmed, A., and Vitek, J. (2018). Correctness of Speculative Optimizations with Dynamic Deoptimization. (POPL). https://doi.org/10.1145/3158137.

Ganz, S. E., Friedman, D. P., and Wand, M. (1999). Trampolined Style. In Rémy, D. and Lee, P., editors, *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, pages 18–27. ACM. https://doi.org/10.1145/317636.317779.

Guo, S. and Palsberg, J. (2011). The Essence of Compiling with Traces. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*. https://doi.org/10.1145/1926385.1926450.

Hölzle, U. and Ungar, D. M. (1994). A Third-Generation SELF Implementation: Reconsiling Responsiveness with Performance. In McKenna, J., Moss, J. E. B., and Wexelblat, R. L., editors, *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1994*, pages 229–243. ACM. https://doi.org/10.1145/191080.191116.

HotSpot (2022). Java HotSpot Performance Engine. https://openjdk.org/groups/hotspot/.

Hur, C., Neis, G., Dreyer, D., and Vafeiadis, V. (2013). The Power of Parameterization in Coinductive Proof. In Giacobazzi, R. and Cousot, R., editors, *The 40th Annual ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, POPL '13, pages 193–206. ACM. https://doi.org/10.1145/2429069.2429093.

Jourdan, J., Laporte, V., Blazy, S., Leroy, X., and Pichardie, D. (2015). A Formally-Verified C Static Analyzer. In Rajamani, S. K. and Walker, D., editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, pages 247–259. ACM. https://doi.org/10.1145/2676726.2676966.

Jourdan, J., Pottier, F., and Leroy, X. (2012). Validating LR(1) Parsers. In Seidl, H., editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer. https://doi.org/10.1007/978-3-642-28869-2_20.

Julia (2022). The Julia Programming Language. https://julialang.org/.

Kaufmann, M., Manolios, P., and Moore, J. (2000). *Computer-Aided Reasoning: ACL2 Case Studies*, volume 4.

Kildall, G. A. (1973). A Unified Approach to Global Program Optimization. In Fischer, P. C. and Ullman, J. D., editors, *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM Press. https://doi.org/10.1145/512927.512945.

Klein, G., Andronick, J., Elphinstone, K., Murray, T. C., Sewell, T., Kolanski, R., and Heiser, G. (2014). Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70. https://doi.org/10.1145/2560537.

Krebbers, R., Jourdan, J., Jung, R., Tassarotti, J., Kaiser, J., Timany, A., Charguéraud, A., and Dreyer, D. (2018). MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–77:30. https://doi.org/10.1145/3236772.

Krebbers, R. and Wiedijk, F. (2011). A Formalization of the C99 Standard in HOL, Isabelle and Coq. In Davenport, J. H., Farmer, W. M., Urban, J., and Rabe, F., editors, *Intelligent Computer Mathematics - 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011*, volume 6824 of *Lecture Notes in Computer Science*, pages 301–303. Springer. https://doi.org/10.1007/978-3-642-22673-1_28.

Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). CakeML: a Verified Implementation of ML. In *Proceedings of POPL*. https://doi.org/10.1145/2535838.2535841.

Lammich, P. (2012). Refinement for Monadic Programs. *Arch. Formal Proofs*, 2012. https://www.isa-afp.org/entries/Refine_Monadic.shtml.

Lammich, P. and Sefidgar, S. R. (2019). Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. *J. Autom. Reason.*, 62(2):261–280. https://doi.org/10.1007/s10817-017-9442-4.

Leroy, X. (2006). Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Proceedings of POPL*. http://doi.acm.org/10.1145/1111037.1111042.

Leroy, X. (2009a). Formal Verification of a Realistic Compiler. *Communications of the ACM*. https://dl.acm.org/doi/10.1145/1538788.1538814.

Leroy, X. (2009b). A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, (4). https://dl.acm.org/doi/10.1007/s10817-009-9155-4.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. (2016). CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE. https://hal.inria.fr/hal-01238879.

Lesani, M., Xia, L., Kaseorg, A., Bell, C. J., Chlipala, A., Pierce, B. C., and Zdancewic, S. (2022). C4: Verified Transactional Objects. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–31. https://doi.org/10.1145/3527324.

Letan, T. and Régis-Gianas, Y. (2020). FreeSpec: Specifying, Verifying, and Executing Impure Computations in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP*. https://doi.org/10.1145/3372885.3373812.

LLVM (2022). MCJIT Design and Implementation. https://llvm.org/docs/MCJITDesignAndImplementation.html.

LuaJIT (2022). The LuaJIT Project. https://luajit.org/.

Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hritcu, C., Rivas, E., and Tanter, É. (2019). Dijkstra Monads for All. *Proc. ACM Program. Lang.*, 3(ICFP):104:1–104:29. https://doi.org/10.1145/3341708.

Malecha, J. G., Morrisett, G., Shinnar, A., and Wisnesky, R. (2010). Toward a Verified Relational Database Management System. In Hermenegildo, M. V. and Palsberg, J., editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 237–248. ACM. https://doi.org/10.1145/1706299.1706329.

MathWorks (2022). MATLAB Execution Engine. https://www.mathworks.com/products/matlab/matlab-execution-engine.html.

Meurer, B. (2017). An Introduction to Speculative Optimization in V8. https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8.

Mono (2022). Mono LLVM. https://www.mono-project.com/docs/advanced/mono-llvm/.

Myreen, M. O. (2010). Verified Just-in-Time Compiler on x86. In Hermenegildo, M. V. and Palsberg, J., editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 107–118. ACM. https://doi.org/10.1145/1706299.1706313.

Myreen, M. O., Slind, K., and Gordon, M. J. C. (2009). Extensible Proof-Producing Compilation. In de Moor, O. and Schwartzbach, M. I., editors, *Compiler Construction, 18th International Conference, CC 2009*, volume 5501 of *Lecture Notes in Computer Science*, pages 2–16. Springer. https://doi.org/10.1007/978-3-642-00722-4_2.

Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L. (2008). Ynot: Dependent Types for Imperative Programs. In Hook, J. and Thiemann, P., editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008*, pages 229–240. ACM. https://doi.org/10.1145/1411204.1411237.

Neis, G., Hur, C., Kaiser, J., McLaughlin, C., Dreyer, D., and Vafeiadis, V. (2015). Pilsner: a Compositionally Verified Compiler for a Higher-Order Imperative Language. In Fisher, K. and Reppy, J. H., editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 166–178. ACM. https://doi.org/10.1145/2784731.2784764.

Nelson, L., Geffen, J. V., Torlak, E., and Wang, X. (2020). Specification and Verification in the Field: Applying Formal Methods to BPF Just-in-Time Compilers in the Linux Kernel. In *14th*

USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, pages 41–61. USENIX Association. https://dl.acm.org/doi/abs/10.5555/3488766.3488769.

Nigron, P. and Dagand, P. (2021). Reaching for the Star: Tale of a Monad in Coq. In Cohen, L. and Kaliszyk, C., editors, *12th International Conference on Interactive Theorem Proving, ITP 2021*, volume 193 of *LIPIcs*, pages 29:1–29:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2021.29.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer.

Norrish, M. (1998). C Formalised in HOL.

Odaira, R. and Hiraki, K. (2005). Sentinel PRE: Hoisting beyond Exception Dependency with Dynamic Deoptimization. In *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005)*, pages 328–338. IEEE Computer Society. https://doi.org/10.1109/CGO.2005.32.

Ottoni, G. (2018). HHVM JIT: a Profile-Guided, Region-Based Compiler for PHP and Hack. In Foster, J. S. and Grossman, D., editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, pages 151–165. ACM. https://doi.org/10.1145/3192366.3192374.

Paleczny, M., Vick, C. A., and Click, C. (2001). The Java HotSpot Server Compiler. In Wold, S., editor, *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*. USENIX. https://dl.acm.org/doi/10.5555/1267847.1267848.

Pettis, K. and Hansen, R. C. (1990). Profile Guided Code Positioning. In Fischer, B. N., editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27. ACM. https://doi.org/10.1145/93542.93550.

Pit-Claudel, C., Philipoom, J., Jamner, D., Erbsen, A., and Chlipala, A. (2022). Relational Compilation for Performance-Critical Applications: Extensible Proof-Producing Translation of Functional Models into Low-Level Code. In Jhala, R. and Dillig, I., editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 918–933. ACM. https://doi.org/10.1145/3519939.3523706.

Pit-Claudel, C., Wang, P., Delaware, B., Gross, J., and Chlipala, A. (2020). Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and

Proofs. In *Proc. of IJCAR 2020*, volume 12167 of *LNCS*, pages 119–137. Springer. https://dl.acm.org/doi/10.1007/978-3-030-51054-1_7.

Pottier, F. and Régis-Gianas, Y. (2022). Menhir. https://pauillac.inria.fr/~fpottier/menhir/menhir.html.en.

Project Zero (2019). JSC Exploits. https://googleprojectzero.blogspot.com/2019/08/jsc-exploits.html.

Project Zero (2020). JITSploitation. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html.

Project Zero (2021a). Chrome Exploits. https://googleprojectzero.blogspot.com/2021/01/in-wild-series-chrome-exploits.html.

Project Zero (2021b). Chrome Infinity Bug. https://googleprojectzero.blogspot.com/2021/01/in-wild-series-chrome-infinity-bug.html.

PyPy (2022). PyPy Python Implementation. https://www.pypy.org/.

R (2022). The R Project. https://www.r-project.org.

Rideau, S. and Leroy, X. (2010). Validating Register Allocation and Spilling. In Gupta, R., editor, *Compiler Construction, 19th International Conference, CC 2010*, volume 6011 of *Lecture Notes in Computer Science*, pages 224–243. Springer. https://doi.org/10.1007/978-3-642-11970-5_13.

Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global Value Numbers and Redundant Computations. In Ferrante, J. and Mager, P., editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27. ACM Press. https://doi.org/10.1145/73560.73562.

Sakaguchi, K. (2018). Program Extraction for Mutable Arrays. In Gallagher, J. P. and Sulzmann, M., editors, *Functional and Logic Programming - 14th International Symposium, FLOPS 2018*, volume 10818 of *Lecture Notes in Computer Science*, pages 51–67. Springer. https://doi.org/10.1007/978-3-319-90686-7_4.

Soman, S. and Krintz, C. (2006). Efficient and General On-Stack Replacement for Aggressive Program Specialization. In Arabnia, H. R. and Reza, H., editors, *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006*, pages 925–932. CSREA Press.

Song, Y., Cho, M., Kim, D., Kim, Y., Kang, J., and Hur, C. (2020). CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.*, (POPL). https://doi.org/10.1145/3371091.

Stewart, G., Beringer, L., Cuellar, S., and Appel, A. W. (2015). Compositional CompCert. In Rajamani, S. K. and Walker, D., editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, pages 275–287. ACM. https://doi.org/10.1145/2676726.2676985.

Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., and Livshits, B. (2013). Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398. https://dl.acm.org/doi/10.1145/2491956.2491978.

Swierstra, W. (2008). Data types à la carte. *J. Funct. Program.* https://doi.org/10.1017/S0956796808006758.

Titzer, B. (2022). A Fast In-place Interpreter for WebAssembly. *Proc. ACM Program. Lang.*, (OOPSLA). https://dl.acm.org/doi/abs/10.1145/3563311.

V8 (2021a). Short Builtin Calls. https://v8.dev/blog/short-builtin-calls.

V8 (2021b). Sparkplug — a Non-Optimizing JavaScript Compiler. https://v8.dev/blog/sparkplug.

V8 (2022). V8 Javascript Engine. https://v8.dev/.

Wadler, P. (1992). The Essence of Functional Programming. In Sethi, R., editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM Press. https://doi.org/10.1145/143165.143169.

Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., and Tatlock, Z. (2014). Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In Flinn, J. and Levy, H., editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14*, pages 33–47. USENIX Association. https://dl.acm.org/doi/10.5555/2685048.2685052.

Wang, Y., Wilke, P., and Shao, Z. (2019). An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.*, 3(POPL):62:1–62:30. https://doi.org/10.1145/3290375.

Wang, Y., Xu, X., Wilke, P., and Shao, Z. (2020). CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files. *Proc. ACM Program. Lang.*, 4(OOPSLA):197:1–197:28. https://doi.org/10.1145/3428265.

Wasm3 (2022). Wasm3. https://github.com/wasm3/wasm3.

Watt, C., Rao, X., Pichon-Pharabod, J., Bodin, M., and Gardner, P. (2021). Two Mechanisations of WebAssembly 1.0. In Huisman, M., Pasareanu, C. S., and Zhan, N., editors, *Formal Methods - 24th International Symposium, FM 2021*, volume 13047 of *Lecture Notes in Computer Science*, pages 61–79. Springer. https://doi.org/10.1007/978-3-030-90870-6_4.

WebKit (2014). Introducing the WebKit FTL JIT. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/.

WebKit (2020). Speculation in JavaScriptCore. https://webkit.org/blog/10308/speculation-in-javascriptcore/.

WebKit (2022). The JavaScriptCore Framework. https://developer.apple.com/documentation/javascriptcore.

Xi, H., Chen, C., and Chen, G. (2003). Guarded Recursive Datatype Constructors. In Aiken, A. and Morrisett, G., editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235. ACM. https://doi.org/10.1145/604131.604150.

Xia, L., Zakowski, Y., He, P., Hur, C., Malecha, G., Pierce, B. C., and Zdancewic, S. (2020). Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.*, (POPL). https://doi.org/10.1145/3371119.

Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and Understanding Bugs in C Compilers. In Hall, M. W. and Padua, D. A., editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 283–294. ACM. https://doi.org/10.1145/1993498.1993532.

Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V., and Zdancewic, S. (2021). Modular, Compositional, and Executable Formal Semantics for LLVM IR. (ICFP). https://doi.org/10.1145/3473572.

Zhao, J., Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. (2012). Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*. https://doi.org/10.1145/2103656.2103709.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

ens rennes
école normale supérieure

Titre : Vérification Formelle de Compilation à la Volée

Mot clés : Vérification formelle, Compilation à la volée, Coq, CompCert

Résumé : La compilation à la volée est une technique pour exécuter des programmes, où l'exécution est mélangée à des optimisations. Les compilateurs à la volée se distinguent par leur efficacité, mais aussi par leur complexité. Par exemple, ils réutilisent des techniques variées : certains contiennent des interprètes pour exécuter leur programme, mais aussi des compilateurs traditionnels pour générer du code machine optimisé. Ils utilisent également des techniques qui leur sont propres comme la spéculation, qui consiste à prédire le comportement futur du programme et générer du code particulièrement rapide si cette prédiction s'avère vrai.

Cette grande complexité peut être à l'origine de bugs. Cette thèse s'attelle à leur vérification formelle, dans le but de développer des compilateurs à la volée dont on peut prouver formellement qu'ils se comportent comme spécifié par la sémantique du programme qu'ils exécutent. Nous présentons des preuves de correction des techniques principales qu'ils utilisent, comme la spéculation, les optimisations dynamiques ou la génération de code machine. Nous réutilisons des techniques de preuves issues de compilateurs traditionnels vérifiés comme CompCert. Nos preuves sont toutes vérifiées mécaniquement dans l'assistant de preuve Coq.

Title: Formal Verification of Just-in-Time Compilation

Keywords: Formal Verification, Just-in-Time Compilation, Coq, CompCert

Abstract: Just-in-Time compilation is a technique to execute programs, where execution is interleaved with optimizations. Just-in-Time compilers often produce fast executions, but are particularly complex. For instance, they reuse various existing techniques: some contain both interpreters to execute programs and traditional compilers to generate optimized machine code. They also use ad hoc techniques like speculation, which consists in predicting the future behavior of the program to generate specialized code that executes particularly fast if the prediction holds.

This great complexity can lead to bugs. This thesis tackles their formal verification, to develop Just-in-Time compilers in such a way that one can formally prove that they behave as prescribed by the semantics of the program they execute. We present correctness proofs for their main features, including speculation, dynamic optimizations and machine code generation. We reuse proof techniques from formally verified traditional compilers like CompCert. All our proofs have been mechanized in the Coq proof assistant.