# Techniques for Foundational End-to-End Verification of Systems Stacks

by

Samuel Gruetter

BSc., Ecole Polytech Fed De Lausanne, 2013
MSc., Ecole Polytech Fed De Lausanne, 2017

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2025



Authored by: Samuel Gruetter
Department of Electrical Engineering and Computer Science
September 12, 2024

Certified by: Adam Chlipala
Arthur J. Conner (1888) Professor of Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

# Techniques for Foundational End-to-End Verification of Systems Stacks

by

Samuel Gruetter

Submitted to the Department of Electrical Engineering and Computer Science
on September 12, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

**ABSTRACT**

Today's software is full of bugs and vulnerabilities. Formal verification provides a promising remedy through mathematical specifications and machine-checked proofs that the implementations conform to the specifications. However, there could still be bugs in the specifications or in the verification tools, which could lead to missed bugs in the software being verified. Therefore, this dissertation advocates for *foundational end-to-end verification*, a proof-based software development method that can mitigate both of these concerns:

It is *end-to-end* in the sense that the correctness proofs of individual components are used to discharge the assumptions of adjacent components throughout the whole stack, resulting in end-to-end theorems that only mention the top-most and bottom-most specifications, so that bugs in intermediate specifications cannot invalidate the soundness of the end-to-end statement anymore.

The method is *foundational* in the sense that the soundness of the proofs relies only on the foundations of mathematics and on the correctness of a small proof-checking kernel, but not on the correctness of other, domain-specific verification tools, because these tools are either proven correct once-and-for-all, or they output proofs that are checked by the kernel.

Ensuring that all the reasoning can be checked by the same small foundational kernel requires considerable effort, and the first part of this dissertation presents techniques to reduce this effort:

Omnisemantics, a new style of semantics that can be used instead of traditional small-step or big-step operational semantics, offer a smooth way of combining undefined behavior and nondeterminism, and enable forward-simulation compiler correctness proofs with nondeterministic languages, whereas previous approaches need to fall back to the much less convenient backward simulations if support for nondeterminism is needed.

Live Verification is proposed, a technique to turn an interactive proof assistant into a programming assistant that displays the symbolic state of the program as the user writes it and allows the user to tweak the symbolic state as long as the tweaks are provably sound. An additional convenience-improving feature is that instead of stating lengthy loop invariants, the user only needs to give the diff between the symbolic state before the loop and the desired loop invariant, resulting in shorter and more maintainable annotations. Finally, in order to make Live Verification practical, a number of additional proof techniques is presented.

The second part of the dissertation shows how these techniques were useful in three collaborative case studies: An embedded system running on a verified processor with an end-to-end proof where the software-hardware interface specification cancels out, a cryptographic server with an end-to-end proof going from high-level elliptic-curve math all the way down to machine code,

and a trap handler to catch unsupported-instruction exceptions whose correctness proof combines program-logic proofs about C-level functions, a compiler correctness proof, and proofs about hand-written assembly.

Thesis supervisor: Adam Chlipala
Title: Arthur J. Conner (1888) Professor of Computer Science

# Acknowledgments

First, I thank my advisor, Professor Adam Chlipala, for advising and supporting me, and for creating a unique environment where exciting multi-person multi-year projects combining software proofs and hardware proofs can be carried out, and for reassuring me that with a project of this size, it is absolutely fine to publish the first paper only after 4 years.

I also thank Professor Frans Kaashoek and Professor Armando Solar-Lezama for serving on my thesis committee and for providing me with interesting food for thought during our meetings.

Many parts of this thesis were collaborations, and I deeply thank my colleagues Andres Erbsen and Thomas Bourgeat for the inspiring collaborations and discussions and for everything they taught me. Many thanks also go to all my co-authors, without whom the work described in this dissertation would not have been possible: Arthur Charguéraud, Joonwon Choi, Ian Clester, Viktor Fukala, Dustin Jamner, Ashley Lin, Jade Philipoom, Clément Pit-Claudel, Pratap Singh, Clark Wood, and Andy Wright.

I would also like to thank Frédéric Besson from the Coq development team for his work on the solver for linear integer arithmetic. Over the course of several years, I reported many issues and limitations about it, and he fixed all of them, until I did not report any more issues – not because I got tired of it, but because the tactic had become so good that I did not run into issues anymore, despite heavy use over several years.

There are also many people who made my time here in Boston very enjoyable: people who are my friends, colleagues, roommates, outdoor buddies, some or all of these at same time. Thank you to Axel, Ben, Brett, Clément, Emma, Eric, Min Ho, Quan, Reyu, Sara, Stella, Thomas, and Twan.

A big thank you also to my "jolly mad" friends (you know who you are ;-)) in Switzerland, for keeping in touch with me so well despite me moving to the "wrong" side of the Atlantic.

And finally, I want to thank my parents and my brother for always being there for me.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

When building computer systems, one combines many components: hardware and many layers of software, usually consisting of some handwritten assembly code, more low-level code and performance-critical code in C or a C-like language, and higher-level application code, compiled and linked using different tools.

The individual components are usually created by many different teams of people, and making sure that a system behaves as expected is hard: On one hand, each individual component needs to be correct, but even if each component's team believes that their component is correct, there might still be bugs because different teams do not agree on what "correct" means for their components: One component might make some reasonable-looking assumptions about its usages, while another component using it makes slightly different and subtly incompatible assumptions that also look reasonable in isolation.

To give just one example, a C application might invoke `memcpy` with a pointer argument that might be `NULL` in the cases where the number of bytes to copy is 0, and its authors might argue that this is okay because no bytes are read or written, while the GCC authors exploit the fact that according to the C standard, programs that pass `NULL` pointers to `memcpy` (and other standard library functions) have undefined behavior, and therefore they optimize the programs under the assumption that all pointers passed to `memcpy` unconditionally are non-`NULL`, which can lead to the deletion of `NULL` checks [Langley, 2016].[1]

Formal methods, by means of formal specifications and machine-checked proofs, seem to provide a promising solution to both the problem of ensuring that each individual component is correct and the problem of ensuring that the assumptions that the components make about each other are compatible. But unfortunately, while formal methods have been applied extensively to the for-

[1] Thanks to my colleague Andres Erbsen for pointing me to this interesting blog post.

mer problem, much less work has applied them also to the latter. In particular, if one uses the correctness proof of one component (e.g. a program proven correct against some formal programming language semantics) to discharge the assumptions of an adjacent component's correctness theorem (e.g., a compiler assuming the same programming language semantics as its source language semantics), one can combine the correctness proofs of the two adjacent components into a new theorem whose statement does not need to mention the intermediate specification (in this example, the source language semantics) anymore. In principle (but not necessarily in practice, as we will discuss shortly in section 1.1), by applying this *modus ponens* rule to all correctness proofs of adjacent layers throughout a whole stack, one can obtain an end-to-end theorem where all the intermediate specifications cancel out, and only the specifications of the top-most and bottom-most layers remain part of the theorem statement, which can then serve as a concise description of the overall behavior of the system.

Such an end-to-end theorem can significantly reduce the auditing burden for people who wish to convince themselves that a system behaves as expected: While auditing a traditional, unverified system requires auditing all its code, and auditing a system whose individual components have been verified requires auditing each component's specification as well as checking that the specifications of adjacent components match, auditing a system with an end-to-end theorem only requires auditing this single theorem statement (and the definitions it references, relating to the top-most and bottom-most layers).

An auditor might also doubt whether the tools that were used to verify the components are correct, because bugs in a verification tool might lead to missed bugs in the programs being verified. To give just one concrete example, in the same way as one might forget a necessary arithmetic overflow check in a program, the implementers of a verification tool might also forget to perform a symbolic overflow check in the verification condition generator of a verification tool. But more interestingly, when combining different verification tools, and feeding them the "same" specs as inputs, there might still be bugs at the boundary between different layers that are checked by different verification tools: For example, let us consider the program snippet `arr[i mod len] = v`, where `arr` is an array, `len` is its length, and `i` and `v` are user-provided values. Suppose this snippet, equipped with suitable pre- and postconditions, passes verification by an application verification tool, and is compiled by a compiler which was verified using a compiler verification tool. Now, if the application verification tool's semantics for the modulo operator assume that $a \bmod b$ is always in the interval $[0, b[$, while the compiler verification tool's semantics and the compiler implementation use a definition of modulo that can return results in the interval $]-b, b[$, we can end up in a situation where both the application verification tool and the compiler verification tool

report successful verification, and yet, when the program is run, a malicious user who provides a negative value for `i` could overwrite some memory outside of the array with an arbitrary value `v`, opening the door for all kinds of remote code execution attacks – a blatant failure of the whole verification endeavor.

To address this concern of bugs in verification tools or incompatibilities between different verification tools, this thesis does *foundational* verification, as defined by Appel [2001]: "A foundational proof is one from just the foundations of mathematical logic," that is, in our case, the calculus of inductive constructions as implemented in the Coq proof assistant. In order to trust Coq proofs, one only needs to trust that Coq's (reasonably small) proof-checking kernel is correct, while all other tools are proven correct once-and-for-all or create proofs that are checked by Coq's kernel.

Ensuring that all the reasoning can be checked by the same small foundational kernel requires considerable effort. Therefore, Part I of this dissertation presents a number of proof techniques that facilitate the verification of individual components as well as the composition of such proofs. And in order to demonstrate that these techniques can indeed be composed into end-to-end theorems where the intermediate specifications cancel out, Part II of this dissertation describes three case studies using these techniques, thus validating their usefulness towards the goal of creating systems that are easy to audit thanks to their small trusted code bases.

## 1.1 Is it Just as Easy as "Apply *Modus Ponens*"?

Composing proofs about individual components into end-to-end-theorems might look as easy as just applying *modus ponens*, but, as a look at some state-of-the-art projects from related work shows, many of them ended up with specifications that seem hard to compose and therefore have not been combined into end-to-end theorems where the intermediate specifications cancel out, thus missing out on a prime opportunity to reduce the trusted code base (TCB) and foregoing one of the benefits of using general-purpose interactive proof assistants.

In the Verified Software Toolchain [Appel et al., 2014], the definition of a Hoare triple $\{P\}\ c\ \{Q\}$ is (ignoring step-indexing) phrased using continuations, roughly: "for all states $s_2$ satisfying $Q$ and for all continuations ($\approx$ C code snippets) $k$, if it is safe to run $k$ in $s_2$, then it is also safe to run $c$ followed by $k$ in any state $s_1$ satisfying $P$" [Appel and Blazy, 2007]. Even though VST's Hoare triples have been proven sound with respect to small-step C semantics of the CompCert verified C compiler [Leroy, 2009a], it is not clear how easy or hard it would be to compose this continuation-based definition, a concrete verified source program, and the CompCert correctness proof into one theorem where the C semantics cancel out, and we are not aware of any work that actually

composed a VST program logic proof with the CompCert compiler proof.[2]

CompCert's backward simulation is a somewhat intricate definition consisting of a `Prop` record of seven properties,[3] even though CompCert proves quite a nice specification preservation theorem[4] as a corollary of the main correctness theorem, which looks like it might be composable.

CertiKOS [Chen et al., 2018] also uses quite a bit of a "lock-in"-prone style: Its top-level correctness theorem roughly says that for all assembly programs, if you link it with the kernel assembly to get a low-level system and also link it with the high-level kernel spec to get a high-level system, then there is a backward simulation between the high-level and low-level system. On one hand this is a very strong correctness property, but on the other hand, it relies on some quite particular notions, namely a notion of linking CompCert assembly, a notion of linking assembly with a high-level spec, and a CertiKOS-specific notion of backward simulation based on the backward simulation of their modified CompCert. They have nice lemmas to compose such refinements. So as long as one stays in this setup, everything works, but breaking out of it, in order to compose with proofs that do not use this setup, looks hard.

Similarly, in the DeepSpec web server [Zhang et al., 2021], whose goal was to connect Coq developments from several research groups and to cancel out the intermediate specifications [Appel et al., 2017, sections 3)a)ii) and 3)c)i)], no theorem refers to both UPenn's and Yale's codebases, and the achieved level of integration is limited to using the same definition of interaction trees [Xia et al., 2019] in both codebases, but these interaction trees seem to still appear inside a framework-specific statement from which they were unable to escape.

## 1.2 Contributions

The contributions of this thesis are of three kinds. In increasing order of generality, they are: First, I contributed to three case studies that push the boundaries of what is possible in foundational end-to-end systems verification. Second, I developed building blocks that enabled these case studies and could be reused for further case studies. Third, and perhaps most importantly from a research point of view, I developed techniques to facilitate foundational systems verification in interactive proof assistants.

[2] Also confirmed in a discussion with William Mansky in January 2024 at POPL and in Lennart Beringer's presentation at AppelFest in May 2024, which listed a connection theorem between VST and assembly where Clight "drops out" as the "ultimate goal" of future work.

[3] https://github.com/AbsInt/CompCert/blob/v3.13.1/common/Smallstep.v#L1293-L1319

[4] https://github.com/AbsInt/CompCert/blob/v3.13.1/driver/Complements.v#L159-L171

### 1.2.1 Case Studies in Foundational End-to-End Systems Verification

**The IoT lightbulb: an embedded system with an end-to-end proof spanning software and hardware** I co-developed, with Andres Erbsen, Joonwon Choi, and Clark Wood, a bare-metal embedded system (chapter 8) with an end-to-end proof that spans both software and hardware. I co-developed the Bedrock2 source language with Andres Erbsen, made sure my compiler and RISC-V semantics are provably compatible with the layer above (i.e. the program logic) as well as with the layer below (the processor), wrote down most of the intermediate specifications and coordinated and negotiated assignment of the proof tasks required to fill all the unproven gaps.

The system is the first such system with unbounded reactive execution, as opposed to computing the result for one input and then terminating. I show that even if the source language semantics require termination for all functions, the required infinite top-level loop can be implemented in assembly (section 8.3.1) and combined with the compiler correctness proof in a way that leads to a concise end-to-end theorem.

This case study is also the first foundationally verified end-to-end software-hardware system featuring realistic I/O, memory-mapped I/O (MMIO) in our case, as opposed to storing its result somewhere in regular memory and relying on an unverified outside process for reading it out. I made sure MMIO can be represented at each layer of the stack at an appropriate level of abstraction and that the different address ranges used for MMIO, instruction memory, stack, and heap remain disjoint, even though in the source language semantics, the only existing memory is heap memory (section 8.3.4).

I also solved the corner case that arises when a pipelined processor writes to an address containing an instruction that was already fetched and is about to be executed, by means of a specification that only allows execution of addresses that were not written to previously (section 8.3.5).

**The garage door opener: An embedded system with cryptographic authentication, with an end–to-end theorem from high-level elliptic-curve math all the way down to machine code** My colleagues Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Clément Pit-Claudel and I developed an end-to-end verified cryptographic server that opens and closes a garage door in response to cryptographically authenticated requests. I did an initial feasibility test to demonstrate that my verified compiler (the Bedrock2 compiler) is able to compile, inside Coq, long functions generated by Fiat Cryptography [Erbsen et al., 2019] that compute modular arithmetic operations, I added features to the compiler (such as embedding data into the emitted code) that are required for the application, and I wrote the compiler's top-level always-eventually theorem whose structure is

mirrored by the garage door top-level theorem. All the code of the system, including code created by two high-level compilers as well as handwritten code, is compiled by my verified compiler.

**Verifying software emulation of an unsupported hardware instruction** I implemented, with advice from my colleague Thomas Bourgeat, a RISC-V trap handler for unsupported instruction exceptions, to demonstrate how my compiler and RISC-V semantics enable reasoning about code that many other verification projects would label as "unverified trusted glue code". The trap handler can be installed on RISC-V systems that do not support the multiplication instruction, and it runs an instruction decoder and multiplier implemented in software whenever an exception is thrown due to a multiplication instruction. The proof combines program-logic proofs about C-level (Bedrock2) functions, the correctness proof of my compiler, and proofs about hand-written assembly, to demonstrate that despite all the implementation complexity, a concise statement about the system's behavior can be proven: The system with multiplication implemented in software behaves as if multiplication were implemented in hardware.

### 1.2.2 Building Blocks

**The Bedrock2 compiler: A verified compiler all the way down to machine code providing correctness proofs about individual functions** I developed a verified compiler (chapter 3) from a C-like language (Bedrock2) to RISC-V. Contrary to CompCert [Leroy, 2009a], which compiles to a fairly high-level assembly language with infinite memory and builtin pseudo-instructions such as allocating and freeing stack frames, my compiler goes all the way down to bytes that represent machine code. It uses a phase-composition mechanism (section 3.5) which, contrary to other, state-of-the-art verified compilers such as CompCert or CakeML [Lööw et al., 2019], enables correctness theorems about individual compiled functions, rather than just about whole programs. Note that there is also a whole line of work around extending CompCert to support various forms of separate compilation, see e.g. [Jiang et al., 2019; Song et al., 2020] and papers cited there, but they all express compiler correctness with respect to a notion of linking that takes place both at the source and target level, resulting in theorems that still talk about execution of whole programs, and it is unclear how one would compose these correctness theorems with correctness theorems about source functions to obtain a recipe (expressed as preconditions on target machine state) on how to call individual functions.

**A formal specification of an industrial ISA usable in combined software-hardware proofs** Based on a specification of the RISC-V instruction set architecture (ISA) in Haskell, I developed a RISC-V specification in Coq, called riscv-coq (chapter 4), that can be used both to prove a compiler against it, as well as to prove a processor against it, in such a way that we can combine the two to obtain a proof where the ISA specification in the middle cancels out. It is the first project where this was done with an industrial ISA rather than a custom one.

**Formally specifying an ISA without using a DSL** A prominent approach to formally specifying ISAs in today's literature is Sail [Armstrong et al., 2019], a domain-specific language just for defining ISA semantics. One might wonder whether for each new domain in which we want to write specifications, a new DSL will be needed, with its own set of tools and maintenance burden. With riscv-coq I showed, together with the other use cases described by Bourgeat et al. [2023], that an ISA can also be specified using only existing languages and tools.

### 1.2.3 Techniques

**Omnisemantics: Combining undefined behavior and nondeterminism in interactive theorem proving** I co-developed, with my colleague Andres Erbsen, a new style of programming language semantics that we call *omnisemantics* and that is significantly easier to use than traditional small-step or big-step operational semantics when the language has both undefined behavior and nondeterminism (chapter 2). The key insight is to define inductive rules in such a way that one derivation talks about all possible nondeterministic executions (hence the name *omni*). Andres and I only worked with imperative languages, and while writing a paper about omnisemantics, we learned that Arthur Charguéraud was working with the same kind of semantics, but for functional languages, so we joined forces, and he contributed the parts about functional languages. I also discovered that a folklore trick to prove type safety in one single proof instead of two proofs (progress and preservation), which only works for deterministic languages when using traditional operational semantics, now also works for nondeterministic languages thanks to omnisemantics.

Schäfer et al. [2016] already used a similar style of semantics they called *axiomatic semantics*, but did not recognize that their approach would enable compiler phases with nondeterministic target languages, nor the application to type safety.

**Forward-simulation compiler correctness proofs with nondeterministic target languages** I developed the first compiler that uses omnisemantics with nondeterministic target languages and

showed that contrary to CompCert's forward simulations, which only work for deterministic languages, omnisemantics forward simulations also work for nondeterministic languages, which is a considerable advantage because forward simulations are much easier to prove than backward simulations.

**Live Verification in an interactive proof assistant** In chapter 5, I show that an interactive proof assistant with an extensible parser and a proof goal display (such as e.g. Coq) can be turned into a live verification tool, that is, a tool that enables programmers to verify their code as they write it in real-time. After each line of code that the programmer writes, the tool tells the programmer whether it was able to prove absence of undefined behavior so far, and it displays a concise representation of the symbolic state of the program right after the added line. The user can then either write the next line of code, or if needed or desired, write a specially marked comment that provides hints on how to solve side conditions or on how to represent the symbolic state more nicely. Once the programmer has finished writing the program, it is already verified with a mathematical correctness proof.

**Expressing a loop invariant as a diff from the symbolic state before the loop** Software verification requires loop invariants, and different tools approach them in different ways: Some attempt to infer loop invariants automatically, which can take a long time and does not always work; while others require the user to state loop invariants manually, which can represent a high annotation burden and require manual adaptation on unrelated source code changes. Based on the observation that a loop invariant often looks quite similar to the symbolic state before the loop, I propose a middle ground between these two extremes: My Live Verification tool still requires the user to come up with the necessary *insight* of the loop invariant, but enables the user to express this insight as a *diff* from the current symbolic state rather than having to spell out the whole loop invariant manually.

**Concepts to reduce the need for backtracking lead to better proof debuggability** Proof automation should be designed to optimize the user experience for failing proofs rather than for proofs where everything works, because the former is the default case in a proof developer's day-to-day work. To make it easier to debug failing proofs, it helps if the framework's automation is structured into many small proof steps that the user can run separately in sequence, and with this structure, it particularly helps to reduce the need for backtracking in proof search. Previous work as well as mine use a notion of *safe steps*, that is, proof steps that do not turn a provable goal into an un-

provable goal. In addition to some well-known safe steps, I present a number of separation-logic concepts that inform proof automation what the only reasonable next proof step can be, which leads to further reduction of the need for backtracking and thus easier debuggability.

**Automated splitting and merging of separation logic clauses** A common pattern in separation logic proofs is that a caller has a big separation logic clause (e.g. an array or a record) and needs to pass a subpart of it (e.g. one array element or a slice of the array or a record field) to an operation such as a memory load or store or a function call. Through specialized lemmas, VST [Cao et al., 2018] provides automation for this pattern in the case of memory loads and stores, but not for function calls. I show that if the right automation is chosen for this pattern, the same automation can be used for memory loads, stores as well as for function calls, and that for a big class of such call patterns, all the information needed to determine the required splits and merges is already contained in the source program and can therefore be completely automated without requiring backtracking (section 5.4.9).

**Simplification of expressions describing symbolic state** I developed a term simplification procedure that helps present the symbolic state in a more concise way, combining rewrite rules (e.g. lemmas about arithmetic or lists) and type-specific simplification procedures (e.g. ring simplification or operator-specific operator-push-down procedures), and I provide informal analytical as well as preliminary empirical evidence that one single bottom-up expression traversal is usually sufficient.

**Additional techniques to make Live Verification in an interactive proof assistant practical** To make Live Verification practical, I developed additional techniques, including a mechanism to extract pure facts from separation logic clauses and a $\mathbb{Z}$ification process that turns expressions over fixed-width integers into expressions over unbounded integers $\mathbb{Z}$ in such a way that the work is reusable among several side condition proofs.

## 1.3 Structure of the Dissertation

Part I describes techniques and building blocks: Omnisemantics (chapter 2), the Bedrock2 verified compiler (chapter 3), the riscv-coq formal ISA specification (chapter 4), Live Verification (chapter 5), and a term simplification procedure (chapter 6).

Part II then presents an overview (chapter 7) of three case studies that show that the techniques can be used to create end-to-end verified systems with a small trusted codebase (TCB) and thus low auditing burden, followed by a chapter for each case study: The IoT lightbulb (chapter 8) combines software and hardware proofs in such a way that the ISA specification is not part of the TCB anymore, the garage door opener (chapter 9) provides a theorem about a cryptographic server that goes all the way from high-level elliptic-curve math to RISC-V machine code, and the softmul case study (chapter 10) shows how my techniques can be used to reason about a RISC-V trap handler.

In chapter 11, I try to measure, using lines-of-code counts, whether the claim that foundational end-to-end proofs reduce the auditing burden can be backed up with numbers, and Part III, chapter 12 concludes.

# Part I

# Techniques and Building Blocks

# Chapter 2

# Omnisemantics[1]

This chapter presents omnisemantics, a style of semantics that I co-developed with Andres Erbsen and Arthur Charguéraud. Undefined behavior and nondeterminism are, as explained in section 2.1, two useful features for modeling programming language semantics. For example, to specify the low-level memory access behavior of C, undefined behavior is crucial, and to specify the semantics of stack allocation, a feature widely used in the case study in chapter 9, nondeterminism is needed to leave the concrete value of the returned address underspecified. However, when using traditional big-step or small-step operational semantics (surveyed in section 2.2) to combine undefined behavior and nondeterminism, some problems arise, as explained in section 2.3, and CompCert's solution to one of the problems (section 2.3.2.5) is not satisfactory. In section 2.4, *omnisemantics* are introduced and shown to solve these problems, and section 2.5 shows that omnisemantics not only work in big-step style, but also in small-step style. Then, section 2.6 presents seven different ways one could discover omnisemantics, providing additional evidence that it makes sense to consider this judgment, and section 2.7 discusses related work.

## 2.1 Undefined Behavior and Nondeterminism

Let us have a look at two features that are useful to specify the semantics of a programming language: Undefined behavior and nondeterminism.

**Undefined behavior** is useful to deal with programs whose execution goes wrong, by which we mean, for example, that execution encounters a function application $(f\ a)$ where $f$ is not a lambda (but some other value, e.g. an integer), or that execution encounters an out-of-bounds write to an

[1]This chapter of the dissertation contains text copied and adapted from the TOPLAS'23 paper I co-authored with Arthur Charguéraud, Adam Chlipala, and Andres Erbsen [Charguéraud et al., 2023].

array. Specifying precise guarantees of what happens in such cases can be undesirable, because it puts too much burden on language implementations: For instance, in order to guarantee that out-of-bounds writes have some well-defined behavior (such as e.g. throwing an exception like Java does or implicitly growing the array like JavaScript does) requires bounds checks on each access and also requires additional metadata (such as the array length) to be stored at runtime – an overhead that low-level, performance-minded languages like C want to avoid. Instead, we want to leave the semantics of such erroneous programs undefined, and allow language implementations to exhibit arbitrary behavior on programs with undefined behavior, including, e.g., to overwrite other data or execute arbitrary code. Of course, this is dangerous from a correctness and security point of view, and therefore, undefined behavior tends to have a bad reputation in programming languages circles, but, as we will see in the later chapters, we can use formal methods to prove absence of undefined behavior. So, to define programming language semantics, we use rules like e.g. "*if* the array index is in bounds, execution does ..." and simply say nothing about all other cases, i.e. leave them *undefined*. The alternative would be to say explicitly that the semantics of this program is "error", but this would require additional rules to propagate errors. For instance, if we use undefined behavior, the construct **`let`** $x = e_1$ **`in`** $e_2$ can be defined with just one rule: "If $e_1$ evaluates to some value $v_1$ and $e_2$ with $v_1$ substituted for $x$ evaluates to some value $v_2$, then the overall let expression evaluates to $v_2$." On the other hand, if we wanted to avoid undefined behavior in our definitions, we would have to add two extra rules for the error cases: One saying that if $e_1$ leads to "error", then the overall let expression also leads to "error", and a similar one for $e_2$. So, to summarize, undefined behavior is useful because it gives flexibility to language implementers and allows concise definitions of language semantics.

**Nondeterminism** is useful to leave certain details in a language definition *underspecified*. Note the subtle difference between *undefined*, meaning that any behavior is allowed, as described above, and *underspecified*, meaning that any behavior *from a well-defined set of alternatives* is allowed. An example for underspecification would be the specification of a pseudo random number generator: It can return any integer, but invoking the pseudo random number generator cannot arbitrarily overwrite unrelated data, like performing an out-of-bounds write could. Other examples of using nondeterminism for underspecification include the address returned by memory allocation, which could be any address different from previously allocated memory, or taking the address of a local variable, or reading input.

One way to avoid this kind of nondeterminism is to model it using oracles, that is, opaque deterministic functions whose implementations are left unspecified. However, to make sure such an oracle function can return a different value each time, it needs to take some opaque state as an

$$\frac{c_1/s \Downarrow s' \qquad c_2/s' \Downarrow s''}{(c_1; c_2)/s \Downarrow s''}$$

$$\frac{y \in \mathrm{dom}\ s \qquad z \in \mathrm{dom}\ s}{(x = y + z)/s \Downarrow s[x := s[y] + s[z]]}$$

$$\frac{s[b] \neq 0 \qquad (c; \textbf{while}(b)\{c\})/s \Downarrow s'}{(\textbf{while}(b)\{c\})/s \Downarrow s'}$$

$$\frac{s[b] = 0}{(\textbf{while}(b)\{c\})/s \Downarrow s}$$

$$\frac{0 \leq v < n}{(x = \mathrm{rand}(n))/s \Downarrow s[x := v]}$$

Whole-program execution: $c/s \Downarrow s'$

(a) Big-step

$$\frac{c_1/s \rightarrow c_1'/s'}{(c_1; c_2)/s \rightarrow (c_1'; c_2)/s'} \qquad \frac{}{(\textbf{skip}; c_2)/s \rightarrow c_2/s}$$

$$\frac{y \in \mathrm{dom}\ s \qquad z \in \mathrm{dom}\ s}{(x = y + z)/s \rightarrow \textbf{skip}/s[x := s[y] + s[z]]}$$

$$\frac{s[b] \neq 0}{(\textbf{while}(b)\{c\})/s \rightarrow (c; \textbf{while}(b)\{c\})/s}$$

$$\frac{s[b] = 0}{(\textbf{while}(b)\{c\})/s \rightarrow \textbf{skip}/s}$$

$$\frac{0 \leq v < n}{(x = \mathrm{rand}(n))/s \rightarrow \textbf{skip}/s[x := v]}$$

Whole-program execution: $c/s \rightarrow^* \textbf{skip}/s'$

(b) Small-step

Figure 2.1: Selected rules in traditional big-step and small-step operational semantics for a simple imperative language

argument, and return an updated state with each result. Modeling what state an oracle depends on accurately can be tricky: For instance, a pseudo random number generator might collect entropy from various sources that might be affected by I/O or by the state of other external functions modeled as oracles. And another drawback of oracles is that threading the state of each oracle through the definition of the semantics of a programming language can become undesirably verbose, especially if there are several oracles.

## 2.2 Background and Baseline: Big-Step and Small-Step Operational Semantics

Traditionally in the programming languages field, the semantics of programming languages are usually defined using big-step or small-step operational semantics. A few sample rules for a simple imperative language are given in Figure 2.1.

The big-step operational semantics judgment has the form $c/s \Downarrow s'$, where $c/s$ is the initial configuration, consisting of a command $c$ and an initial state $s$, and $s'$ is the final state reached after executing $c$. In this simple language, states $s$ are partial maps from variable names to integers.

While the big-step judgment completely evaluates its command, the small-step judgment, written $c/s \rightarrow c'/s'$, only performs one small step at a time, from an initial configuration $c/s$ to a next configuration $c'/s'$ with an updated command and state. To talk about the execution of a whole program, we therefore need to take the reflexive-transitive closure of the step relation, written $c/s \rightarrow^* \texttt{skip}/s'$, where `skip` is used to denote the empty program.

Small-step operational semantics are more suitable to reason about infinitely-running programs and about interleaved execution of several programs, but they tend to be a bit less convenient than big-step semantics for proofs by induction over the structure of a derivation.

### 2.2.1 Expressing Undefined Behavior and Nondeterminism

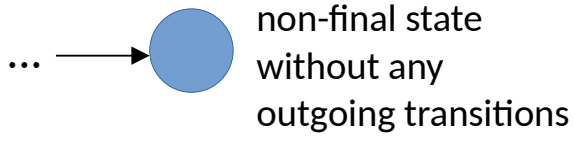
Let us see how undefined behavior and nondeterminism are expressed in traditional big-step and small-step operational semantics:

- Undefined behavior is expressed by *absence of applicable rules*: For instance, in the rules for the addition command $x = y + z$, if the premises requiring that $y$ and $z$ are in the domain of the variable map $s$ are not satisfied, there is no other applicable rule (and the configuration $c/s$ is called "stuck").
- Nondeterminism is expressed using *implicit top-level universal quantification*: All variables appearing in inference rules are presumed to be universally quantified, so in the rule for the pseudo random number generator, the user of the rule can pick any $v$ that satisfies the premise $0 \leq v < n$.

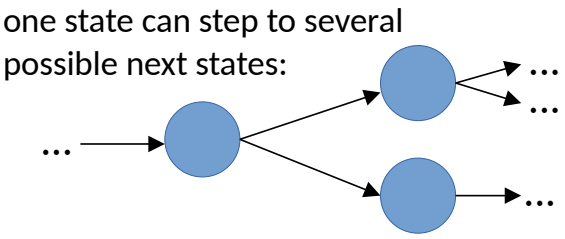
Alternatively, we can also represent them graphically in state transition diagrams, as depicted in Figure 2.2.

## 2.3 Problems Arising When Combining Undefined Behavior and Nondeterminism

When using both undefined behavior and nondeterminism at the same time, a few problems (or, at least, inconveniences) arise that are explained in the following subsections.




(a) States with undefined behavior (stuck states) are non-final states without any outgoing transitions.




(b) Nondeterminism is represented as multiple outgoing transitions from the same state.

Figure 2.2: Representing undefined behavior and nondeterminism in state transition diagrams

### 2.3.1 Problem 1: Expressing That Every Execution Safely Terminates in a State Satisfying Some Postcondition

In program and compiler verification, the statements we want to prove are often of the form

*Program c, when run in initial state s, safely terminates,*
*and all final states satisfy postcondition P.* (∗)

By "postcondition", we mean a predicate over states, and by "safely terminate", we mean that the program does not reach any non-final state where it is stuck, i.e., no state for which the behavior is undefined, and that it does not loop infinitely.

Expressing (∗) in a deterministic language is straightforward: $\exists s', c/s \Downarrow s' \wedge P\ s'$ in big-step, or $\exists s', c/s \rightarrow^* \texttt{skip}/s' \wedge P\ s'$ in small-step. However, as illustrated in Figure 2.3, as soon as we add nondeterminism to a language, proving that one path to a state satisfying the postcondition exists is not sufficient anymore, because there could be another path to a stuck state. Also, simply proving $\forall s', c/s \Downarrow s' \Rightarrow P\ s'$ (in big-step) or $\forall s', c/s \rightarrow^* \texttt{skip}/s' \Rightarrow P\ s'$ (in small-step) is not sufficient either, because these hypotheses include only final states, so we can still miss paths to stuck states.

#### 2.3.1.1 Unsatisfactory Existing Solutions to Problem 1

There are various fixes to support nondeterminism, but all of them are quite cumbersome:

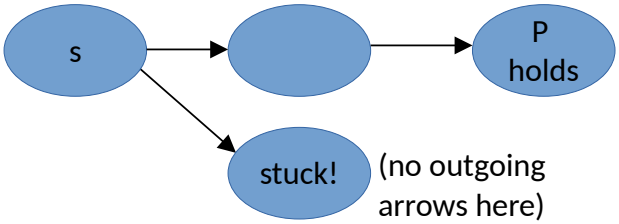



Figure 2.3: Proving that one path to a state satisfying the postcondition exists is not sufficient if there is nondeterminism, nor is proving that all final states satisfy the postcondition

**Long formula** One way is to say that for all reachable states $s'$ which do not step any further, the desired postcondition holds:

$$\forall s',\ (s\ \rightarrow^*\ s'\ \wedge\ \nexists s'',\ s'\ \rightarrow\ s'')\ \Rightarrow\ P\ s'$$

However, it is ill-suited for proofs by induction over execution steps.

**Separate safety judgment** Another fix would be to define a separate safety judgment $\mathsf{safe}(c/s)$ to assert that executing $c$ from state $s$ does not get stuck and terminates. For each construct of the language, we would then have to add a corresponding safety rule. For example, the safety rules for sequence and addition would look as follows:

$$\frac{\mathsf{safe}(c_1/s) \qquad \forall s',\ c_1/s \Downarrow s'\ \Rightarrow\ \mathsf{safe}(c_2/s')}{\mathsf{safe}((c_1; c_2)/s)} \qquad\qquad \frac{y \in \mathsf{dom}\ s \qquad z \in \mathsf{dom}\ s}{\mathsf{safe}((x = y + z)/s)}$$

Note how the rule for sequencing needs to universally quantify over all possible $s'$ *within* its premise. Having to duplicate the number of rules and carrying around a safety judgment everywhere might work, but is undesirable.

**Adding Explicit Errors** Yet another fix would be to introduce an explicit "error" state err to make the evaluation judgment total, in the sense that for all $c/s$, there exists at least one $s'$ such that $c/s \Downarrow s'$. However, this approach would increase the number of rules by a factor even greater than 2: On one hand, for each leaf rule (i.e. rule without recursive invocations of the judgment in its premises), we would have to add rules to specify how this statement could lead to an error. For

instance, the addition command would require two extra rules:

$$\frac{y \notin \text{dom } s}{(x = y + z)/s \Downarrow \text{err}} \qquad \frac{z \notin \text{dom } s}{(x = y + z)/s \Downarrow \text{err}}$$

And on the other hand, for each rule with recursive invocations of the judgment in its premises, we would have to add one extra rule per recursive invocation, to propagate the potential error of each recursive rule. For instance, these two extra rules would be needed for the sequence command:

$$\frac{c_1/s \Downarrow \text{err}}{(c_1; c_2)/s \Downarrow \text{err}} \qquad \frac{c_1/s \Downarrow s' \qquad c_2/s' \Downarrow \text{err}}{(c_1; c_2)/s \Downarrow \text{err}}$$

### 2.3.2 Problem 2: How to Use Forward Simulations in the Presence of Nondeterminism

#### 2.3.2.1 Expressing Compiler Correctness as a Backward Simulation

Traditionally, compiler correctness is expressed as a *backward*[2] *simulation*: For each possible behavior of the target program created by the compiler, there exists a corresponding behavior of the source program that justifies the target program behavior, which could be expressed as follows:

$$\forall s', C(c)/s \Downarrow s' \Rightarrow c/s \Downarrow s'$$

This statement only talks about terminating behaviors. If we use semantics that also allow talking about failing and diverging behaviors, we should add two similar implications saying that all failing and diverging behaviors of the target program are also justified by corresponding failing and diverging behaviors of the source program (this is, roughly, what CompCert does). In this thesis, however, we restrict ourselves to programs that are proven safe, that is, do not have undefined behavior and terminate. This choice is not really a restriction in our setting, because we prove functional correctness for all our source programs (which implies that they are safe), and for the very few deliberately infinitely-running programs, we use a separate mini-compiler as described in section 8.3.1.

So, we allow a compiler $C$ to assume that the source program is safe, and require, as part of compiler correctness proofs, a proof that the emitted target program is safe, which leads to the

[2]We follow CompCert's terminology, where "backward" means "from target language to source language", i.e. backward with respect to the direction of compilation, even though in earlier work, "backward" means "backward in time" [Lynch and Vaandrager, 1996].

following definition:

**Definition 2.1** (Backward-simulation-based compiler correctness).

$$\textsc{compile-correct-bw-sim}(C) := \forall c\ s,\ \mathsf{safe}(c/s)\ \Rightarrow\ (\mathsf{safe}(C(c)/s) \wedge (\forall s', C(c)/s \Downarrow s'\ \Rightarrow\ c/s \Downarrow s'))$$

This formulation still contains the simplification that the source and target state representation are assumed to be the same, which is not the case for many interesting compiler phases. Therefore, for full generality, we should use source language states $s_1$ and $s_1'$, target language states $s_2$ and $s_2'$, as well as a relation $R$ between source and target states, which leads to the following definition:

**Definition 2.2** (Backward-simulation-based compiler correctness, generalized).

$$\begin{aligned}&\textsc{compile-correct-bw-sim-generalized}(C) :=\\ &\quad\forall c\ s_1,\ \mathsf{safe}(c/s_1)\ \Rightarrow\ \big(\forall s_2,\ R\ s_1\ s_2\ \Rightarrow\ \mathsf{safe}(C(c)/s_2) \wedge\\ &\qquad(\forall s_2', C(c)/s_2 \Downarrow s_2'\ \Rightarrow\ \exists s_1', R\ s_1'\ s_2' \wedge c/s_1 \Downarrow s_1')\big)\end{aligned}$$

The structure and intuition behind this formula is still the same: If the source program is safe, the compiled program is safe as well, and each possible execution of the compiled program can be justified by a corresponding execution of the source program. However, as we can see, the formula is now quite verbose, so for the remainder of this chapter, we will use the simplified version where source and target language use the same state representation, but all the results also generalize to the case with different state representations with a relation $R$ between them.

#### 2.3.2.2 Forward Simulations

*Forward simulations* state that all source program executions have a corresponding target program execution:

**Definition 2.3** (Forward-simulation-based compiler correctness).

$$\textsc{compile-correct-fw-sim}(C) := \forall c\ s\ s',\ c/s \Downarrow s'\ \Rightarrow\ C(c)/s \Downarrow s'$$

For deterministic target languages, forward simulations can be turned into backward simulations:

**Theorem 2.4.** *If the target language is deterministic, then*

$$\textsc{compile-correct-fw-sim}(C)\quad\Rightarrow\quad\textsc{compile-correct-bw-sim}(C)$$

*Proof.* COMPILE-CORRECT-BW-SIM($C$) allows us to assume $\mathsf{safe}(c/s)$, which implies that there exists an $s'$ such that $c/s \Downarrow s'$, and using the implication from the forward simulation, we obtain $C(c)/s \Downarrow s'$. We need to show $\mathsf{safe}(C(c)/s)$, which holds because we have one execution starting at $C(c)/s$, and since the target language is deterministic, this one is the only possible execution, so all executions are safe, and we also need to show $\forall s'', C(c)/s \Downarrow s'' \implies c/s \Downarrow s''$, which holds because target language determinism allows us to conclude that $s''$ is the same as the $s'$ we already have, so we only need to show $c/s \Downarrow s'$, which we already did. □

Note that, contrary to the backward simulation, we do not need to add any safe assumption or conclusion, because for deterministic languages, safe is subsumed by $\Downarrow$.

A related variant of this forward-to-backward theorem (that also supports failing and diverging programs) is used by CompCert.

#### 2.3.2.3 Forward Simulations are More Convenient

The appeal of forward simulations is that they are easier to prove, because they allow writing a proof by induction over the source language execution, which involves a case analysis over all possible source language constructs, and given one source language construct, we can symbolically evaluate the compilation function on it to obtain a target language snippet, which we then can symbolically evaluate according to the target language semantics. Backward simulations, on the other hand, tend to be harder to prove, because typically, each target language construct could be the compilation result of several different source language constructs.

The claim that forward simulations are easier to prove is also supported by the fact that all CompCert phases are proven correct using forward simulations, except the first one, which does not actually change the program, but changes the semantics from nondeterministic expression evaluation order to one specific, deterministic expression evaluation order.

Note that this claim applies only to compiler correctness theorems that universally quantify over all possible programs. In a translation validation setting, where we reason about only one concrete source language program and its corresponding concrete target language program at a time, the difficulties do not appear.

#### 2.3.2.4 Forward Simulations do not Work With Nondeterminism

But unfortunately, if the target language has both nondeterminism and undefined behavior (which both are highly useful features to define programming language semantics, as we saw in section 2.1), forward simulations are meaningless: They merely say that for each source language

behavior, a corresponding target language behavior exists, but they do not show that there are no other (nondeterministic) target language executions that could lead to undefined behavior, as illustrated with !? in Figure 2.4b.

#### 2.3.2.5 CompCert's (Non-)Solution to Enable Use of Forward Simulations

CompCert's main correctness theorem is stated as a backward simulation, but the correctness proofs of almost all phases are stated as forward simulations, and (a variant of) Theorem 2.4 is used to turn them into backward simulations. However, this trick requires making the target language and intermediate languages deterministic, which comes at a considerable cost. Arguably, the most natural way to model memory allocation would be to use nondeterminism to state that the returned pointer could be any address (as long as it is disjoint from previously allocated memory). Now, to remove this nondeterminism, CompCert's memory model [Leroy et al., 2012] models pointers not just as 32-bit or 64-bit integers, but as a tuple of a block ID and an offset, both of which are unbounded integers, and the semantics state that if the most recent allocation returned block ID $n$, the next allocation will *deterministically* return block ID $n + 1$.

However, this design means that whenever a compilation phase introduces or removes allocations, the addresses in the target memory will differ from their corresponding addresses in the source memory, so to relate source states to target states, one has to use a so-called *memory injection*, which is difficult.[3]

### 2.3.3 Problem 3: How to Prove Progress & Preservation in One Linear-Size Proof

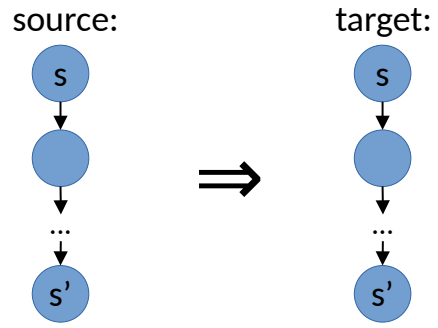
The traditional approach of proving type safety [Wright and Felleisen, 1994] for a typed programming language involves writing two proofs, progress and preservation:[4]

$$\begin{aligned} &\textsc{progress:} && \vdash t : T \implies \exists v,\ t \Downarrow v \\ &\textsc{preservation:} && (\Gamma \vdash t : T \ \wedge\ t \Downarrow v) \implies \Gamma \vdash v : T \end{aligned}$$

A problem with this approach is that both proofs have a fairly similar structure that requires case analysis over all typing rules, and each change to the language requires updating both proofs (unless they are highly automated, but in that case, failures of the automation are harder to pinpoint).

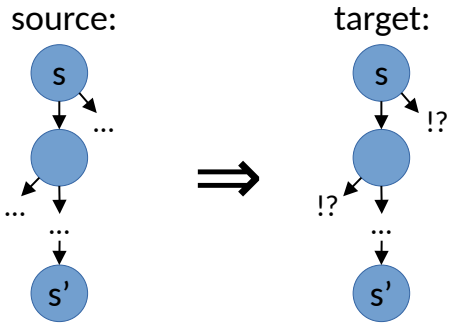
[3] Personal communication with several researchers who worked with CompCert

[4] For simplicity, we consider a functional language here, but the discussion also applies to imperative languages with a store $s$. Moreover, we only discuss the case for big-step semantics, but the same considerations also apply for small-step semantics.



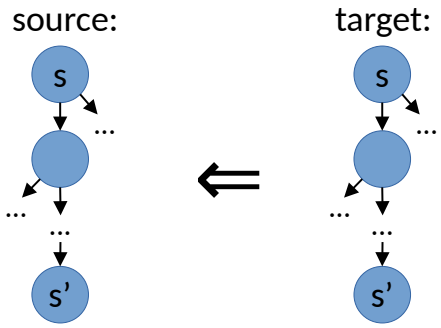

Meaningful: ✓ Convenient: ✓

(a) Traditional forward simulation for deterministic languages



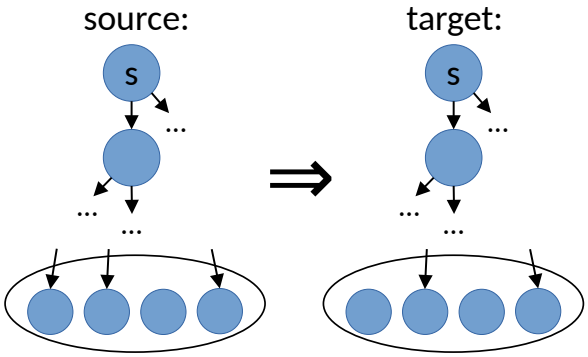

Meaningful: ✗ Convenient: ✓

(b) Traditional forward simulation for nondeterministic languages




Meaningful: ✓ Convenient: ✗

(c) Traditional backward simulation for nondeterministic languages




Meaningful: ✓ Convenient: ✓

(d) Omnisemantics forward simulation for nondeterministic languages

Figure 2.4: Different simulations and their assessment: Whether they are *meaningful*, i.e. prove something useful, and whether they are *convenient* to prove.

$$\frac{c_1/s \Downarrow Q \qquad \forall s',\, Q\, s' \implies c_2/s' \Downarrow P}{(c_1; c_2)/s \Downarrow P} \qquad \frac{y \in \mathsf{dom}\ s \qquad z \in \mathsf{dom}\ s \qquad P\, s[x := s[y] + s[z]]}{(x = y + z)/s \Downarrow P}$$

$$\frac{\begin{array}{c} s[b] \neq 0 \qquad c/s \Downarrow Q \\ \forall s', Q\, s' \implies (\mathbf{while}(b)\{c\})/s' \Downarrow P \end{array}}{(\mathbf{while}(b)\{c\})/s \Downarrow P} \qquad \frac{s[b] = 0 \qquad P\, s}{(\mathbf{while}(b)\{c\})/s \Downarrow P}$$

$$\frac{0 < n \qquad \forall v,\, 0 \leq v < n \implies P\, s[x := v]}{(x = \mathsf{rand}(n))/s \Downarrow P}$$

Figure 2.5: Selected big-step omnisemantics rules

Progress is usually proven by induction over the typing derivation, whereas preservation, which has both a typing derivation and an evaluation derivation as hypotheses, can go by induction over either of them. No matter which one is chosen in the preservation proof, in an interactive proof assistant, the first step of each case is then to do a case analysis on the other derivation to conclude that only a few cases are possible, namely those talking about the same language construct as the other hypothesis. Coq's inversion tactic automatically gets rid of the contradictory cases, but the proof term that it generates (and the kernel needs to check) still contains one case for each combination of a typing rule and an evaluation rule, which leads to a proof term whose size is quadratic in the size of the language definition, and for big languages, this blowup can be a problem.

Similarly to the two previous problems of section 2.3.1 and section 2.3.2, there exists a simple solution that unfortunately only works for deterministic languages: Instead of proving progress and preservation separately, we just prove the following combined type-safety statement:

$$\vdash t : T \implies \exists v,\, t \Downarrow v \ \wedge \ \vdash v : T$$

For deterministic languages, the existential quantification is unique, so it can be turned into the universal quantification required for preservation, but this trick stops working as soon as we introduce nondeterminism.

## 2.4 The Big-Step Omnisemantics Judgment

It turns out that all the problems described in section 2.3 vanish if we change the judgment so that one derivation talks about *all* (omni, hence the name omnisemantics) possible nondeterministic behaviors instead of just one. Concretely, we define a judgment of the form $c/s \Downarrow P$, where $P$ is a postcondition over states, whose meaning is that program $c$, when run in initial state $s$, safely terminates, and *all* possible final states satisfy $P$.

Sample rules are given in Figure 2.5. Note how, contrary to the traditional rules in Figure 2.1a, the rule for rand talks about *all* possible returned values of the rand function within *one* rule application. Deterministic rules such as the rule for addition just assert that the postcondition holds on the updated state, whereas the rule for sequencing requires the prover to pick a "mid-condition" $Q$ that holds after executing $c_1$, and to show that $c_2$ safely runs to a state satisfying $P$ for all states in that chosen $Q$. The first rule for while loops uses this same sequencing pattern to chain the first loop iteration with subsequent iterations, and the second rule for while loops just asserts that the postcondition needs to hold on the initial state in case the loop condition is false.

### 2.4.1 Relationship to Traditional Semantics

The big-step omnisemantics judgment is related to traditional big-step operational semantics in the following sense:

**Theorem 2.5** (Equivalence of big-step omnisemantics and traditional semantics).

$$c/s \Downarrow P \quad \Longleftrightarrow \quad \mathsf{safe}(c/s) \wedge \forall s',\, c/s \Downarrow s' \;\Rightarrow\; P\, s'$$

The proof is by induction on the derivations and is unsurprising.[5]

Moreover, the omnisemantics judgment has the exact same meaning as the weakest-precondition judgment, so if one defined an omnisemantics judgment for a language, one can use it to define $\mathsf{wp}(c, P)(s) := c/s \Downarrow P$.

However, there is still an important difference between the traditional way of defining a weakest-precondition judgment and omnisemantics: The weakest-precondition rule for loops requires providing a loop invariant, and one rule application covers the execution of the whole loop, whereas in the omnisemantics judgment, no loop invariant needs to be given, and the loop is unfolded iteration by iteration like in traditional big-step operational semantics, so proof trees contain one

[5] A Coq proof by Arthur Charguéraud is available in the artifact of our omnisemantics paper [Charguéraud et al., 2023] at https://samuelgruetter.net/assets/omnisemantics-artifact.zip in the file `lambda/Omnisemantics.v`.

rule application per loop iteration, instead of one rule application per loop construct. In other words, the structure of the proof tree of a weakest-precondition derivation mirrors the structure of the command, whereas the structure of the proof tree of an omnisemantics derivation mirrors the structure of the execution.

### 2.4.2 Solving Problem 1: It Works By Definition

To solve problem 1, it suffices to observe that our definition of $c/s \Downarrow P$ exactly matches the desired meaning given in $(*)$ in section 2.3.1.

### 2.4.3 Solving Problem 2: Omnisemantics Forward Simulations Just Work

Using omnisemantics, we can make the following definition:[6]

**Definition 2.6** (Omnisemantics forward simulation for compiler correctness).

$$\textsc{compile-correct-omni-fw-sim}(C) := \forall c\ s\ P,\ c/s \Downarrow P \implies C(c)/s \Downarrow P$$

Compared to COMPILE-CORRECT-FW-SIM from Theorem 2.3, the conclusion about target language execution is now much stronger, because it talks about *all* possible behaviors, so the risk of missing some bad executions is gone, as illustrated in Figure 2.4d.

#### 2.4.3.1 Omnisemantics Forward Simulation Implies Traditional Backward Simulation

The intuition that omnisemantics forward simulations mean what they should mean can also be made formal by proving that they imply traditional backward simulations:

**Theorem 2.7.** *For all compilation functions $C$,*

$$\textsc{compile-correct-omni-fw-sim}(C) \implies \textsc{compile-correct-bw-sim}(C)$$

*Proof.* Let us show a chain of implications, starting by unfolding COMPILE-CORRECT-OMNI-FW-SIM$(C)$:

$$\forall c\ s\ P,\ c/s \Downarrow P \implies C(c)/s \Downarrow P$$

[6] Note that, like in section 2.3.2, all definitions and proofs also extend to the case where the source- and target-language state representations differ, and are related by a relation $R$, but to make the presentation more readable, we stick to the simple case.

By applying Theorem 2.5 on both sides of the implication, we get

$$\forall c\ s\ P,\ (\mathsf{safe}(c/s) \wedge \forall s',\ c/s \Downarrow s' \Rightarrow P\ s') \Rightarrow (\mathsf{safe}(C(c)/s) \wedge \forall s',\ C(c)/s \Downarrow s' \Rightarrow P\ s')$$

and by instantiating $P$ with the strongest postcondition of $c/s$, i.e. with $(\lambda s'.\ c/s \Downarrow s')$, we get

$$\forall c\ s,\ (\mathsf{safe}(c/s) \wedge \forall s',\ c/s \Downarrow s' \Rightarrow c/s \Downarrow s') \Rightarrow (\mathsf{safe}(C(c)/s) \wedge \forall s',\ C(c)/s \Downarrow s' \Rightarrow c/s \Downarrow s')$$

and by removing a trivial implication, we get

$$\forall c\ s,\ \mathsf{safe}(c/s) \Rightarrow (\mathsf{safe}(C(c)/s) \wedge \forall s',\ C(c)/s \Downarrow s' \Rightarrow c/s \Downarrow s')$$

which is exactly the definition of COMPILE-CORRECT-BW-SIM($C$). □

### 2.4.4 Solving Problem 3: Progress and Preservation in One Go

Using omnisemantics, the trick described in section 2.3.3 also works for nondeterministic languages, that is, to prove type safety, one proves

$$\Gamma \vdash t : T \Rightarrow t \Downarrow (\lambda v.\ \Gamma \vdash v : T)$$

in one induction over the typing derivation. The interesting cases are rules for nondeterministic constructs like rand: Their evaluation rules have premises with universal quantifiers, so while proving type safety for these cases, we introduce the universally quantified variables *inside* the cases, whereas in the traditional approach, the universal quantification over all possible executions is on the very *outside* (top level), in the statement of preservation.

### 2.4.5 Overapproximation of the Set of Results

Note that the postconditions $P$ appearing in $c/s \Downarrow P$ are *overapproximations* of the set of possible outcomes. One might wonder why we do not use precise outcome sets that contain only the outcomes that can actually occur. For instance, the rule for sequencing would look as follows:

$$\frac{c_1/s \Downarrow Q \qquad \forall s', Q\ s' \Rightarrow c_2/s' \Downarrow P_{s'}}{(c_1; c_2) \Downarrow \bigcup_{s' \in Q} P_{s'}}$$

To get the precise outcome set, we rely on a family of outcomes $P_{s'}$ indexed by possible states $s'$ after executing $c_1$, and take the union of all these outcome sets, where we use the more intuitive set

$$\frac{c_1/s \;\to\; Q \qquad \forall c_1'\ s',\ Q(c_1'/s') \;\Rightarrow\; P((c_1';c_2)/s')}{(c_1;c_2)/s \;\to\; P} \qquad \frac{P(c_2/s)}{(\mathbf{skip};c_2)/s \;\to\; P}$$

$$\frac{y \in \mathrm{dom}\ s \qquad z \in \mathrm{dom}\ s \qquad P(\mathbf{skip}/s[x := s[y] + s[z]])}{(x = y + z)/s \;\to\; P} \qquad \frac{s[b] \neq 0 \qquad P((c;\mathbf{while}(b)\{c\})/s)}{(\mathbf{while}(b)\{c\})/s \;\to\; P}$$

$$\frac{s[b] = 0 \qquad P(\mathbf{skip}/s)}{(\mathbf{while}(b)\{c\})/s \;\to\; P} \qquad \frac{0 < n \qquad \forall v,\ 0 \le v < n \;\Rightarrow\; P(\mathbf{skip}/s[x := v])}{(x = \mathrm{rand}(n))/s \;\to\; P}$$

Figure 2.6: Selected small-step omnisemantics rules

notation $\bigcup_{s' \in Q} P_{s'}$ to denote $(\lambda s''.\exists s',\ Q\ s' \wedge P_{s'}\ s'')$. One can define precise rules for all constructs of the language, but we found them harder to use, because when applying rules, one cannot freely pick the postcondition, and the rule of consequence (weakening), a useful rule because it allows us to forget irrelevant details, does not hold anymore.

## 2.5 The Small-Step Omnisemantics Judgment

One can also define small-step rules in omnisemantics style, as illustrated in Figure 2.6. The judgment $c/s \;\to\; P$, where $P$ is a proposition over configurations, states that *all* immediate steps that configuration $c/s$ can take are safe and lead to a new configuration that satisfies $P$.

In order to lift a single small step to multiple steps, we use the *eventually* operator given in Figure 2.7b. Similarly to the transitive-reflexive closure operator given in Figure 2.7a, it has a base case and a recursive case for the chaining, and their difference is the same as the difference between the traditional and big-step rule for sequencing: The traditional one only considers one possible middle state, whereas the omnisemantics rule universally quantifies over all of them.

Finally, inspired by temporal logic, we can also define an *always* operator, as shown in in Figure 2.8. We can either use an inductive invariant $I$ as shown in Figure 2.8a, and prove three premises that *establish*, *preserve*, and *use* this invariant, or, if we are willing to use coinduction, we can use a mid-condition $Q$ that only needs to hold after the next step instead of always, as shown in Figure 2.8b, but in practice, one still needs to come up with an invariant, except that now it can be outside of the rule. The two rules can be shown to be equivalent in Coq.[7]

[7] See https://github.com/mit-plv/coqutil/blob/c1caa082052a/src/coqutil/Semantics/OmniSmallstepCombinators.v for the corresponding Coq code.

$$\frac{}{s \ \to^* \ s} \qquad \frac{s_1 \ \to \ s_2 \quad s_2 \ \to^* \ s_3}{s_1 \ \to^* \ s_3}$$

(a) Definition of the transitive-reflexive closure operator lifting a traditional small-step judgment to multiple steps

$$\frac{P \ s}{s \ \to^\Diamond \ P} \qquad \frac{s_1 \ \to \ Q \quad \forall s_2, \ Q \ s_2 \implies s_2 \ \to^\Diamond \ P}{s_1 \ \to^\Diamond \ P}$$

(b) Definition of the *eventually* operator lifting an omnisemantics small-step judgment to multiple steps

Figure 2.7: Lifting small-step judgments to multiple steps

$$\frac{I \ s_0 \qquad (\forall s, \ I \ s \implies s \ \to \ I) \qquad (\forall s, \ I \ s \implies P \ s)}{s_0 \ \to^\square \ P} \qquad \frac{P \ s_0 \qquad s_0 \ \to \ Q \qquad (\forall s, \ Q \ s \implies s \ \to^\square \ P)}{s_0 \ \to^\square \ P}$$

(a) Inductive definition using an invariant $I$

(b) Coinductive definition

Figure 2.8: Inductive and coinductive definition of the omnisemantics *always* judgment

## 2.6 All Roads Lead to Omnisemantics

We present seven different interpretations of the omnisemantics big-step judgment. Each of them represents a way one could discover it, and the fact that there are so many of them provides additional evidence that it makes sense to use this style of semantics. Moreover, these interpretations should also provide the reader with more intuition about what this judgment means.

**First interpretation: generalization from one result to a set of results** The standard big-step judgment $t/s \Downarrow v/s'$ relates one input configuration $t/s$ to one single result configuration $v/s'$.

The omnisemantics big-step judgment $t/s \ \Downarrow \ Q$ relates one input configuration $t/s$ to a set of results, described by $Q$. The omnisemantics big-step judgment thus appears as the immediate generalization of the standard big-step judgment to go from one result to a set of results.

**Second interpretation: a CPS version of the standard big-step judgment** Consider the view of the standard big-step judgment $t/s \ \Downarrow \ v/s'$ as a function that, given an input configuration $t/s$, returns (non-deterministically) a pair $v/s'$. Now, consider the continuation-passing style (CPS) version of that function. Instead of returning a $v/s'$, it takes an additional argument $Q$, the continuation, and passes $v/s'$ to $Q$. In CPS, we are free to choose the return type of the continuation. Here,

we choose $Q$ to have return type Prop, meaning that $Q$ is a predicate over final configurations.

**Third interpretation: a Hoare triple with a singleton precondition** Consider a Hoare logic with total correctness triples written $\{H\}\ t\ \{Q\}$, with a precondition $H$ of type $\mathsf{heap} \to \mathsf{Prop}$ and a postcondition $Q$ of type $\mathsf{val} \to \mathsf{heap} \to \mathsf{Prop}$ (again, this type is isomorphic to sets of final configurations). Such a triple asserts that, in any state $s$ satisfying the precondition $H$, any possible evaluation of $t/s$ reaches a final configuration satisfying the postcondition $Q$. The omnisemantics big-step judgment is thus very closely related to the Hoare triple judgment.

On the one hand, Hoare triples may be defined in terms of the omnisemantics big-step judgment, as follows.

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall s,\ H\,s \ \Rightarrow\ (t/s \Downarrow Q)$$

On the other hand, the omnisemantics big-step judgment is equivalent to a Hoare triple with a precondition that characterizes a single state. Let us write $(= s)$ as a shorthand for $\lambda s''.\ s'' = s$. The following equivalence holds.

$$\{(= s)\}\ t\ \{Q\} \quad \Longleftrightarrow \quad t/s \Downarrow Q$$

**Fourth interpretation: an inductively defined weakest precondition** The weakest-precondition judgment has the same meaning as the omnisemantics big-step judgment, i.e. $\mathsf{wp}\ t\ Q$ can be defined as $\lambda s.(t/s \Downarrow Q)$. So, if omnisemantics big-step semantics is just a weakest precondition, what is new about it? The novelty lies in the fact that the omnisemantics big-step judgment is directly defined through a set of inductive rules which *defines* the semantics of the language in big-step style. On the contrary, the weakest precondition judgment is typically defined as a derived judgment, expressed with respect to a small-step semantics (or with respect to a big-step semantics, but only in the particular case of deterministic semantics).

The reader may wonder whether it would be possible to define wp directly as an inductive judgment. For example, in the let-binding case, one would like to consider the reasoning rule:

$$\mathsf{wp}\ t_1\ (\lambda v'.\mathsf{wp}\ ([v'/x]\ t_2)\ Q) \ \vdash\ \mathsf{wp}\ (\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2)\ Q$$

as part of the *inductive definition* of the judgment $\mathsf{wp}\ t\ Q$. Yet, such a rule is not accepted by Coq as an inductive definition because the nested occurrence of wp in the premise is not a *positive occurrence*. This caveat is avoided in the omnisemantics big-step semantics through the use of the intermediate postcondition $Q_1$ in the rule MBIG-LET:

MBIG-LET

$$\frac{t_1/s \Downarrow Q_1 \qquad \big(\forall v'\ s',\ \ Q_1\, v'\, s' \ \Rightarrow\ ([v'/x]\, t_2)/s' \Downarrow Q\big)}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)/s \Downarrow Q}$$

**Fifth interpretation: a generalized typing judgment** Let us argue informally that the omnisemantics big-step judgment is a direct generalization of a typing judgment. To ease the discussion, let us assume a purely-functional language, that is, ignore all details related to the state. In that setting, the omnisemantics big-step judgment simplifies to $t \Downarrow Q$. As a first step towards a typing judgment, let us write this judgment instead in the form $\vdash\ t\ :\ Q$. For this judgment, the rule MBIG-LET becomes:

MBIG-LET (WITHOUT EFFECTS)

$$\frac{\vdash\ t_1\ :\ Q_1 \qquad \big(\forall v_1 \in Q_1,\ \ \vdash\ ([v_1/x]\, t_2)\ :\ Q\big)}{\vdash\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ :\ Q}$$

For the second step, let us assume an environment-based semantics as opposed to a substitution-based semantics—the two are equivalent. Concretely, the value $v_1$ produced by $t_1$ is no longer substituted for $x$ in $t_2$, but instead bound to $x$ in the environment $E$. We obtain the following rule.

MBIG-LET (WITHOUT EFFECTS / WITH SEMANTIC ENVIRONMENTS)

$$\frac{E\ \vdash\ t_1\ :\ Q_1 \qquad \big(\forall v_1 \in Q_1,\ \ (E, x \mapsto v_1)\ \vdash\ t_2\ :\ Q\big)}{E\ \vdash\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ :\ Q}$$

Finally, we can abstract *semantic environments* as *typing environments* in the following sense: rather than binding in the environment a variable $x$ to an arbitrary value $v_1$ that belongs in the set $Q_1$, we can directly bind the variable $x$ to $Q_1$. Essentially, this amounts to viewing the set $Q_1$ as a type. The result corresponds exactly to the standard typing rule for let-bindings.

MBIG-LET (WITHOUT EFFECTS / WITH TYPING ENVIRONMENTS)

$$\frac{E\ \vdash\ t_1\ :\ Q_1 \qquad (E, x : Q_1)\ \vdash\ t_2\ :\ Q}{E\ \vdash\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ :\ Q}$$

This discussion of the similarities between the omnisemantics big-step judgment and a typing judgment explains well, we believe, the role of the intermediate postcondition $Q_1$ that appears in the rule MBIG-LET: it plays exactly the same role as the type of $t_1$ in the typing rule for $\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$.

**Sixth interpretation: Avoiding existentials when proving deterministic programs correct** One can state that a deterministic program $t$, starting at state $s$, does not get stuck and that its final state will satisfy some postcondition $Q$ as follows:

$$\mathsf{runs\text{-}to\text{-}satisfying}(t, s, Q) := \exists\, v\ s',\ t/s \Downarrow v/s' \wedge Q(v, s')$$

However, as soon as one tries to prove properties about concrete or compiler-generated programs using proof goals of shape $\mathsf{runs\text{-}to\text{-}satisfying}(t, s, Q)$, it is inconvenient to deal with the existentials: If we unfold runs-to-satisfying, the first proof step we have to do is to instantiate the two existentials for the result value $v$ and the final state $s'$. Using Coq's *evar* feature (section 5.2.5) to delay choosing them does not help much, because at this point in the proof, the variables to be used to instantiate them eventually might not yet be in scope. It is therefore more convenient to have backward reasoning rules to step through the program $t$ one instruction at a time. Such backward reasoning rules can be proven as derived lemmas. For instance, for let expressions, one could prove the following lemma:

$$\frac{t_1/s \Downarrow v_1/s' \qquad \mathsf{runs\text{-}to\text{-}satisfying}([v_1/x]t_2, s', Q)}{\mathsf{runs\text{-}to\text{-}satisfying}(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2, s, Q)}\ \textsc{rt-step}$$

However, one can also define runs-to-satisfying inductively, by interpreting the above rule as a definition, and by adding the following rule for the base case:

$$\frac{Q(v, s)}{\mathsf{runs\text{-}to\text{-}satisfying}(v, s, Q)}\ \textsc{rt-done}$$

And finally, when one wonders whether the evaluation of $t_1$ in RT-STEP could also be expressed in terms of runs-to-satisfying, one stumbles on the rule MBIG-LET presented before, up to renaming of runs-to-satisfying into $\Downarrow$.

**Seventh interpretation: Adding a postcondition to a safety judgment** Wang et al. [2014] use a coinductively defined judgment to define when all nondeterministic executions of a program are safe (do not get stuck). For instance, using our notations, the rule for let expressions could be stated as follows:

$$\frac{\mathsf{safe}(t_1/s) \qquad (\forall v\ s',\ t_1/s \Downarrow v/s' \Longrightarrow \mathsf{safe}(([v/x]t_2)/s'))}{\mathsf{safe}(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2, s)}$$

Now, when proving statements of the form $\mathsf{safe}(t/s) \wedge \forall v\ s',\ t/s \Downarrow v/s' \implies Q(v, s')$, one might wonder whether the safety judgment could not also take care of asserting that $Q$ holds at the end. Indeed it can, and what one obtains is, again, omnisemantics big-step semantics.

## 2.7 Related work

Schäfer et al. [2016] already used a similar style of semantics they called *axiomatic semantics*, and used it to prove correctness of a compiler from a nondeterministic language to a deterministic language, but they did not recognize that their approach would enable compiler phases with nondeterministic target languages, nor the application to type safety.

While we can reason about a `random()` function as a function returning an arbitrary value, we do not attempt to do probabilistic reasoning, unlike Polaris [Tassarotti and Harper, 2019].

Iris [Jung et al., 2018] also uses weakest precondition forward reasoning. $\lambda$MC [Frumin et al., 2019] is a small C-like language with nondeterministic expression evaluation order, defined on top of Iris. It enables one to prove absence of undefined behavior in a given C program for any evaluation order, using a weakest precondition generator and separation logic.

Big-step omnisemantics are different from Hoare logic [Hoare, 1969] in two ways: First, they do not have preconditions, but only postconditions, and preconditions are obtained by outside hypotheses. Second, the rule for loops does not require an invariant, but instead an outcome set for one loop body iteration, and another derivation for the remainder of the execution. That is, the depth of a Hoare logic derivation corresponds to the depth of the abstract syntax tree, whereas the depth of an omnisemantics derivation corresponds to the number of loop iterations, the same way as a big-step derivation would.

One key question is how much of a program's internal nondeterminism should be reflected in its *execution trace*. At one extreme, one could include in the trace an event for every nondeterministic choice performed, as well as a *delay* event, a.k.a. a *tick*, to reflect in the trace each transition performed by the program, following the approaches of Danielsson [2012]. More recent work on interaction trees [Koh et al., 2019; Xia et al., 2019] maps each program to a coinductive structure featuring ticks in addition to I/O steps. Yet, these approaches come at the cost of reasoning "up to removal of a finite number of ticks."

A promising route to avoiding ticks is the *mixed inductive-coinductive* approach of Nakata and Uustalu [2010], for distinguishing between *reactive* programs that always eventually perform I/O operations and *silently diverging* programs that eventually continue executing forever without performing any I/O. Despite apparent benefits, this approach seems not to have gained popularity or

evaluation in the form of sizable case studies.

The refinement steps in Fiat [Delaware et al., 2015] can be seen as compilation passes that reduce internal nondeterminism.

Cito [Wang et al., 2014] integrates calls to axiomatically specified external functions into its big-step operational semantics, which talk about one execution at a time, and it uses a separate judgment talking about all executions at once to define which initial states are "safe" in the sense that no execution will fail. Omnisemantics can be seen as this safety judgment augmented with a postcondition, or alternatively, Cito's safety judgment can be seen as an omnisemantics judgment where the postcondition is $\lambda s_{final}.\mathsf{True}$.

The problem of having to duplicate each big-step operational semantics rule for failure can be addressed using pretty-big-step semantics [Charguéraud, 2013].

Lee et al. [2017] discuss undefined behavior and unspecified values in the context of LLVM.

Nondeterminism has also been studied using denotational semantics [Plotkin, 1976], and the relationship between nondeterminism, concurrency, and communication was already being studied in the 1970s [Francez et al., 1979].

The idea that nondeterminism can not only be used to choose between a finite number of statements to execute next, but also to axiomatically specify procedures, and that this requires *unbounded* nondeterminism, was first described in [Back, 1980].

Some authors distinguish between *demonic* nondeterminism (what we call simply "nondeterminism" here) and *angelic* nondeterminism (a form of nondeterminism where only one possible execution needs to satisfy the postcondition, rather than all of them), and study how these two forms of nondeterminism are related to each other. This line of work was summarized by Hesselink [2010].

# Chapter 3

# The Bedrock2 Verified Compiler

I joined my advisor's group because I was excited about the goal of proving an end-to-end theorem about a system that spans both software and hardware (see chapter 8 for the result), and he already had a PhD student (Joonwon Choi) working on the hardware side, and two students (Andres Erbsen and myself) excited about the software side were about to join for PhD. As a first step towards this goal, I developed a verified compiler (called "the Bedrock2 compiler") from a simple, C-like language (Bedrock2) to RISC-V [Waterman and Asanovic, 2019] machine code. Given that the CompCert verified C compiler [Leroy, 2009a,b] already existed, one might wonder why we did not reuse CompCert, and what the new research contributions of my compiler could be.

Regarding the first question, the original main motivation to start a verified compiler from scratch was to use a clean-slate source language design and memory model that should be as simple as possible, so that verifying programs against this source language would be easier than verifying them against the notoriously complex C semantics (which I had experienced through VST during my master's thesis [Gruetter, 2017]). But after writing my compiler and having made many design decisions informed by the goal to use it in end-to-end proofs, when comparing it to CompCert, it became apparent that CompCert lacks several features that are required (or, at least, helpful) to use it in bigger developments where the assumptions of the individual components' correctness proofs get discharged by the correctness theorems of the components' adjacent components, and I describe these below in section 3.1.

Regarding the second question, the main research contribution in my compiler is to show how omnisemantics enable forward simulation proofs, which is described in more detail in chapter 2, and, perhaps, some insight on how (not) to compose compiler phase correctness proofs (section 3.5). But, more important than the research within the compiler, it has enabled other research,

such as the three projects described in Part II, but it was also used in the Silver Oak project[1] at Google (unbeknownst to me until I joined the project in my internship), which proved correctness of drivers for hardware accelerators used in the OpenTitan[2] root-of-trust project, and there are two additional ongoing projects in my advisor's group at MIT based on my compiler, one on proving that the compiler does not introduce additional information leakage (e.g. through branching on potentially secret values or through memory accesses whose addresses might depend on secret values), and one on proving upper bounds on the running time of programs (measured in number of executed instructions).

## 3.1 Advantages of the Bedrock2 Compiler

While developing the Bedrock2 compiler, I made several design decisions that were motivated by the goal of including it in bigger end-to-end-verified software-hardware projects, and later, while studying CompCert in more detail, I discovered that I had deviated from CompCert's design decisions in many more ways than just choosing a simpler source language, and that using CompCert for our projects would have been quite difficult or even impossible. In the following, I will describe how my compiler differs from CompCert and why these differences matter.[3]

**Simple source language semantics.** The semantics of C are notoriously intricate. Trying to demonstrate that formal methods can capture all these intricacies is an interesting (academic) exercise,[4] but as the interface between verified programs and a verified compiler, one should choose a language that is as suitable for proofs as possible, which means that a language with simpler semantics is better.

**Simple, close-to-hardware memory model.** CompCert and the Bedrock2 compiler each pick one representation of memory and use that same representation for the source language, all intermediate languages, as well as the target language. However, the chosen representation differs between the two:

In CompCert, a pointer is a tuple of a block ID (which is increased sequentially with each allocation) and an offset, and memory is a map from such pointers to byte-sized fragments, but

[1] https://github.com/project-oak/silveroak

[2] https://opentitan.org/

[3] The comparison to CompCert in this chapter refers to CompCert version v3.14, which is available at https://github.com/AbsInt/CompCert/tree/v3.14.

[4] This exercise has been carried out in even more detail than in CompCert, by e.g. Norrish [1998], Krebbers [2015], Memarian et al. [2016] and others cited there.

the memory representation also records the primitive C types (int, long, float, single, or pointer) and values that each fragment is a part of, as well as read/write/deallocate permissions for each address. The block ID of a pointer is an unbounded positive number, and the offset is a unbounded integer number, so memories of arbitrary size (beyond $2^{64}$ bytes) are possible, and there is no direct correspondence between CompCert pointers and 32-bit or 64-bit addresses that are used on today's computers. CompCert's source language semantics allow casting a pointer to an integer, but the resulting integer remains tagged as coming from a pointer, and the only permissible operation on that integer is to cast it back to a pointer.

In contrast, the Bedrock2 memory representation is simply a partial map from 32-bit or 64-bit words to bytes. On one hand, this simple representation simplifies proofs of programs and of the compiler, but on the other hand, it is also much closer to the memory model used in proofs about hardware, which enables us to connect software proofs to hardware proofs, as described in chapter 8.

**All the way down to machine code.** The Bedrock2 compiler emits a list of bytes that corresponds to a sequence of RISC-V instructions, and its correctness proof is about the execution of these bytes, as specified by riscv-coq (chapter 4). In contrast, the verified part of CompCert emits idealized RISC-V[5] assembly.[6] It differs from actual RISC-V assembly by having pseudo-instructions to allocate and free memory on the stack, and by pretending that each arithmetic instruction has a 32-bit and a 64-bit variant (whereas in actual RISC-V, most instructions are shared between 32-bit and 64-bit architectures, and there are a handful of operations with a `W` suffix to operate on 32-bit data on 64-bit machines). This idealized assembly is then processed by trusted OCaml code that replaces the stack alloc/free pseudo-instructions by instructions that manipulate the stack pointer (register `X2`)[7] and then prints it to a text-based assembly file.[8] From there, linking, computing offsets of jumps, and emitting the machine code is performed by the (unverified) GCC RISC-V toolchain. Verified CompCert extensions that make it stack-aware and emit binary ELF files have been developed [Wang et al., 2019, 2020], but they are closed-source and therefore cannot be used by researchers outside the authors' research group.

[5] Contrary to the Bedrock2 compiler, CompCert also supports other instruction set architectures such as x86, ARM, and PowerPC, but all these backends end at a similarly high-level idealized assembly language.

[6] See file `riscV/Asm.v` in the CompCert sources

[7] Function `expand_instruction` in `riscV/Asmexpand.ml`

[8] Function `print_instruction` in `riscV/TargetPrinter.ml`

**Support for nondeterministic language specifications.** Nondeterminism is a useful feature for defining language semantics, in particular when *underspecification* is desired. For example, when specifying the operation of taking the address of a local variable in C, all we want to say is that this operation returns a pointer to memory that is disjoint from all other known memory, but we do not want to give the exact address, and we also do not want to introduce a deterministic (but opaque) oracle function for it, because we would have to say that this oracle depends on some opaque state and thread that state through all the semantics.

In Bedrock2, taking the address of a local variable is modeled using a stack allocation command of the form `let x := stackalloc(n) in c`, which assigns to `x` a *nondeterministically* chosen pointer to `n` bytes of memory that can be used within the command `c`.

Reasoning about code that takes the address of local variables is considered hard,[9] but I believe that it becomes considerably simpler if nondeterminism is available as a specification tool.

Contrary to the Bedrock2 compiler, all intermediate and target languages of CompCert are deterministic, and this is a design choice that was made to enable forward simulation proofs. However, as described in chapter 2, using omnisemantics, we can get the best of both worlds, that is, use nondeterminism and forward simulation proofs at the same time.

**Avoid using axioms for potentially dischargeable assumptions and parameters.** Coq provides an **`Axiom`** command that lets the user state a proposition that Coq will consider as true without requiring a proof for it. Using this command can threaten the soundness of a whole proof development, because an axiom could be false, or it could contradict other axioms in a way that might have been exploited in a proof. To make it easier to audit proofs, Coq provides a **`Print Assumptions`** *myTheorem* command, which traverses the proof of *myTheorem* and the proofs of all its transitive dependencies to collect and print all the axioms on which *myTheorem* depends. When auditing a Coq theorem, one therefore has to study not only the list of explicitly stated hypotheses of the theorem, but also the list of axioms returned by **`Print Assumptions`**. These two lists are part of the *trusted code base* of a development, and an important selling point of using Coq is that it enables exceptionally small trusted code bases – if the development is structured appropriately. In particular, whenever one component of a development makes an assumption that can be discharged by the correctness proof of another, one should perform this *modus ponens* step, so that in the combined end-to-end theorem, the assumption does not appear anymore, and the trusted code base remains small.

Metatheoretically, the choice of whether to make an assumption an axiom (top of Figure 3.1a)

[9] Personal communication with several members of the Verified Software Toolchain team

```
Axiom p_holds: P.
Theorem q_holds: Q.
Proof.
... proof using the axiom ...
Qed.

... later:

Theorem p_is_actually_provable: P.
Proof. ... Qed.

Theorem end_to_end: Q.
Proof.
  no way to combine q_holds and
  p_is_actually_provable :(
Abort.
```

(a) Theorem `q_holds` uses an axiom

```
Theorem q_holds: P → Q.
Proof.
... proof using the hypothesis ...
Qed.

... later:

Theorem p_is_actually_provable: P.
Proof. ... Qed.

Theorem end_to_end: Q.
Proof.
  apply q_holds.
  apply p_is_actually_provable.
Qed.
```

(b) Theorem `q_holds` uses a hypothesis

Figure 3.1: Making an assumption an axiom vs. making it a hypothesis

or a hypothesis (top of Figure 3.1b) does not matter, in the sense that the proof obligations that Coq asks the user to prove are the same. But when it comes to software engineering and easing the auditing of a proof development, the two choices are very different: Since Coq's **`Axiom`** command was originally intended only for axioms that cannot (or will never) be proven in Coq, there is no way in Coq to discharge an axiom (e.g. `p_holds`) assumed by a theorem (e.g. `q_holds`) to construct a new, axiom-free theorem (`end_to_end`): This *modus ponens* step can be performed only if the assumption was made a hypothesis, like in Figure 3.1b.

However, if many lemmas in a proof development all depend on some assumption `P`, it might appear cumbersome to make `P` an explicit assumption of all these lemmas, and one might be tempted to state it as an **`Axiom`** instead, so that the lemmas do not all need to have `P` as an explicit assumption.

Unfortunately, CompCert uses the **`Axiom`** command for this purpose, and therefore, its output of **`Print Assumptions`** on the top-level theorem[10] contains not only truly non-dischargeable axioms such as e.g. functional extensionality or the law of the excluded middle, but also dischargeable parameters such as compiler options and assumptions about the semantics of inline assembly and external functions. Discharging these assumptions in an end-to-end proof would require modifying CompCert's source code to hard-code the parameters and axioms to concrete values and proofs

[10] `transf_c_program_correct` in the file `driver/Compiler.v`

about them. So one could not use CompCert as an unmodified library, and if a development wanted to use two different instantiations of CompCert, it would have to use two copies of CompCert that live in different namespaces, which would require adapting the import statements at the top of all files – clearly an undesirable situation from a software engineering point of view.

In contrast, the Bedrock2 compiler uses hypotheses for all its parameters and assumptions so that they can be discharged later, and the output of **`Print Assumptions`** `compiler_correct`[11] only contains two non-dischargeable axioms from Coq's standard library: propositional and functional extensionality.

**Parameterization over behavior and compilation of external functions.** Instead of using axioms for the assumptions about the behavior of external functions (like CompCert does), the Bedrock2 compiler uses Coq's **`Section`** command and typeclasses to parameterize the development over them, and also over a compilation function for these external calls, which has to satisfy the same correctness theorem as the compiler itself. This mechanism is described in more detail in section 3.4, and was successfully used in the lightbulb (chapter 8) and garage door (chapter 9) case studies.

**Parameterization over bitwidth.** The CompCert and Bedrock2 compiler both support 32-bit as well as 64-bit architectures. However, the parameterization over the bitwidth is done by the build system in CompCert, whereas it is done inside Coq's logic for the Bedrock2 compiler. Therefore, CompCert needs to be compiled twice (separately for each bitwidth), and if a development wanted to use both the 32-bit and 64-bit version at the same time, it would have to use two copies of CompCert living in separate namespaces.

**Correctness theorems about individual compiled functions.** As explained in more detail in section 3.5, the correctness theorem of the Bedrock2 compiler can be used to reason about the behavior of individual compiled functions. This ability is crucial for the kind of reasoning done in Silver Oak[12] and in the Softmul (chapter 10) case study. In contrast, CompCert's correctness theorem only talks about whole programs. It does have a correctness theorem about separate compilation, but while this theorem allows separate compilation of individual functions, the correctness guarantee only applies to whole linked programs.

[11] This theorem is in the file `Pipeline.v`.
[12] https://github.com/project-oak/silveroak

**Running the compiler inside the proof assistant.** Being able to run the compiler inside the proof assistant has two advantages: On one hand, when proving a theorem about the behavior of a concrete program that was compiled with the compiler, we can discharge the assumption that compiling the source program resulted in the target program, leading to fewer *explicit* assumptions in our end-to-end theorems; and on the other hand, we do not need to assume that Coq's extraction mechanism for exporting Coq (Gallina) to OCaml is correct, which leads to fewer *implicit* assumptions (and a smaller trusted code base) in our end-to-end theorems.

The Bedrock2 compiler has been designed from the very start to be runnable inside Coq, which required avoiding the use of the `sumbool` type and other types that introduce proof terms into the terms that are supposed to be executable, because proof terms are often too big to be executed, and they also usually contain theorems that are marked as opaque and thus do not reduce. Some effort towards adopting this coding style was also made in CompCert [Leroy, 2015], but it seems that the changes described there were not merged, and the main recommended way of running CompCert is still via extraction to OCaml.

**License.** Finally, less relevant from a research point of view, but more relevant from an adoption point of view, the CompCert non-commercial license agreement is a non-free license that could be revoked at any time, which makes CompCert and any project depending on it unlikely to be adopted by any free-and-open-source project.

## 3.2 The Bedrock2 Source Language

The Bedrock2 source language is an untyped, simple imperative language. All variables are $n$-bit integers, where $n$ is the bitwidth of the processor (32 or 64). Whether a value is to be treated as an unsigned or signed integer or as a pointer is not derived from types but from the operator being applied. For instance, there are two different right-shift operators, one for signed and one for unsigned right-shifts.

The grammar of expressions is given in Figure 3.2a. Note that, contrary to C, expressions are side-effect-free and cannot contain function calls. This restriction simplifies the reasoning, and completely sidesteps any evaluation order questions, because all expression evaluation orders lead to the same result.

The `inlinetable`$N(t)(e)$ construct can be used to embed data (a list of bytes $t$) in source code. It is used in the garage door case study (chapter 9) to store a pre-computed list of constants required in the computation of IP checksums, as well as to embed the public key of the garage owner in the

| $e$ ::= | |
|---|---|
| $v$ | integer literal |
| $x$ | local variable |
| `load`$N(e)$ | load $N$ bytes from address $e$, $N$ = 1, 2, or 4 |
| `loadW`$(e)$ | load one word (i.e. 4 bytes on 32-bit machines, 8 on 64-bit) |
| $e_1$ $op$ $e_2$ | binary operation, $op$ = `+`, `-`, `*`, `&`, … |
| $e_1$ ? $e_2$ : $e_3$ | conditional |
| `inlinetable`$N(t)(e)$ | load $N$ bytes starting at the $e$-th byte of constant byte list $t$ |

(a) Grammar of expressions

| $c$ ::= | |
|---|---|
| `skip` | do nothing |
| $x = e$ | assignment to local variable |
| `store`$N(e_1, e_2)$ | store the lower $N$ bytes of $e_2$ at address $e_1$, $N$ = 1, 2, or 4 |
| `storeW`$(e_1, e_2)$ | store 4 or 8 bytes, depending on bitwidth, at address $e_1$ |
| **`if`** `(`$e$`)` $c_1$ **`else`** $c_2$ | if-then-else |
| $c_1$`;` $c_2$ | sequence |
| **`while`** `(`$e$`)` $c$ | while loop |
| $x_1, \ldots x_n = f(e_1, \ldots e_m)$ | call function $f$ and assign return values to $x_1, \ldots x_n$ |
| **`let`** $x$ := `stackalloc(`$n$`)` **`in`** $c$ | allocate $n$ bytes of memory on the stack |

(b) Grammar of commands

Figure 3.2: Grammar of the Bedrock2 source language

source code.

The main motivation for adding conditionals at the expression level is to use them to define syntactic sugar for lazy `&&` and `||`. There is also syntactic sugar for Boolean *not*.

The grammar of commands is given in Figure 3.2b. Unlike in C, functions can have multiple return values. Although not shown in Figure 3.2b, the grammar in Coq syntactically distinguishes calls to Bedrock2-defined functions and calls to external functions. For specification purposes, the latter are recorded in an interaction trace (which, as a purely specificational artifact, does not exist at runtime).

Local variables cannot be addressed. Instead, there is a stack allocation construct of the form `let x := stackalloc(n) in c`, which assigns to `x` a nondeterministically chosen pointer to `n` bytes of memory that can be used within the command `c`.

## 3.3 Compilation Phases

Figure 3.3 shows the compilation pipeline. The names next to the arrows are the names of the phases, and the names between the arrows are the names of the intermediate languages.

First, in **FlattenExpr**, the expressions in the source code are flattened, so that all operators assign their results to temporary variables. This language is called `FlatImp`, and it still uses the same type of variables as the source language, namely strings. Next, two optimizations (contributed by my undergraduate mentee Arthur Reiner de Belen) are run:

**UseImmediate** detects and marks binary operators where one argument is a constant, so that later, the FlatToRiscv phase can use RISC-V's *immediate* instructions that place the constant in a bitfield of the instruction. This optimization reduces the number of variables, because the constants do not need to be assigned to variables first anymore.

The next optimization phase, **DeadCodeElim**, eliminates all assignments to unused variables, including the ones that became unnecessary because of UseImmediate.

Then, the **RegAlloc** phase performs register allocation based on a linear scan, replacing the string variable names by variable indices of type $\mathbb{Z}$. After this phase, there can still be arbitrarily many variables, but variables smaller than 32 stand for registers,[13] and variables greater than or equal to 32 stand for stack slots, i.e. variables that will be spilled in the next phase. For simplicity, there is no SSA (static-single-assignment) transformation, and the register allocator assigns one live interval to each variable, which might be an over-approximation if the variable is not live

[13] The RISC-V ISA has 32 registers.

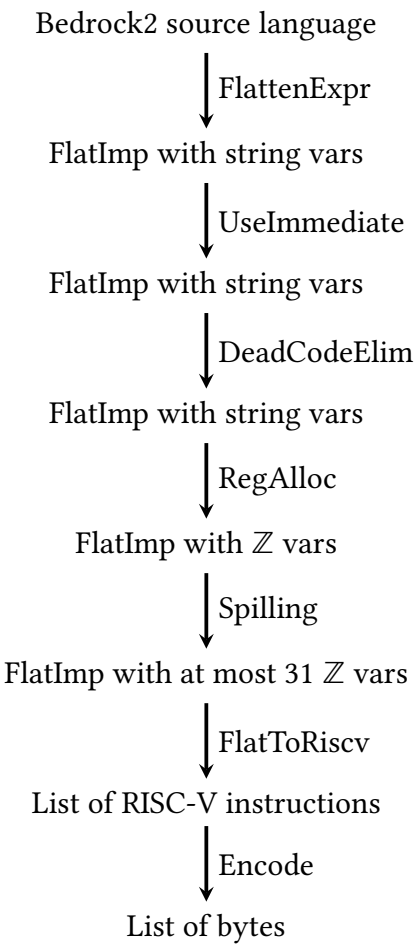



Figure 3.3: Phases (next to arrows) and intermediate languages between them

during some sub-intervals of its assigned interval. The live intervals are sorted by length, and one after the other, starting with the shortest one, is assigned to the lowest available $\mathbb{Z}$ variable. This sorting heuristic tends to put short-lived variables into registers ($\mathbb{Z}$ variables with lower indices), and if stack slots are needed, they will contain the longer-lived variables, which is beneficial under the assumption that the longer-lived variables are not accessed as frequently as the short-lived temporaries.

The structure of this register allocation algorithm does not at all follow the structure of the execution of the code, and therefore, proving correctness of the register allocation algorithm would be really hard. Instead, this phase is followed by a checker that traverses the source and target programs of the register allocation phase simultaneously, checking that they have the same structure and maintaining a mapping from source variables to target variables to check that each target program variable usage will see the correct value. This checker might, in theory, reject target programs emitted by the register allocator, but in practice, it did not reject any examples we tried. The correctness statement of this phase says that if the checker succeeds, the target program behaves like the source program, and is much easier to prove, because the checker follows the structure of the execution of the program quite closely.

Next, the **Spilling** phase uses the `stackalloc` command to create stack slots for all variables whose indices are greater or equal 32, and replaces each read of a spilled variable by a load from the corresponding stack slot to a temporary register, and also replaces each write of a spilled variable by a store to the corresponding stack slot, via a temporary register. After spilling, all used variable indices are less than 32.

From there, the **FlatToRiscv** phase compiles code directly to a list of position-independent RISC-V instructions, without using any intermediate assembly language with features such as labels or pseudo-instructions where one pseudo-instruction could result in a variable number of machine instructions. In order to emit the correct relative jumps for function calls, the compiler needs to know the position of the callee relative to the caller, which it obtains from a partial map from function names to relative positions. To compute this map, we need to know the size of each compiled function, so we run the compiler a first time with a dummy map and then a second time with the actual map.

Finally, the **Encode** phase turns the list of RISC-V instructions into a list of bytes as specified by RISC-V, using an encoding function implemented in Coq, which is proven to be the inverse of the decoding function specified in the RISC-V Coq specification.

## 3.4 Parameterization over the External-Calls Compiler

All phases above FlatToRiscv leave external calls unchanged, but once we want to emit RISC-V, we need to know how to turn external calls into actual RISC-V code. For instance, an external call might get replaced by a system call instruction or by a jump to a special address, or, on a processor with a custom instruction, by such a custom instruction, or, to perform memory-mapped I/O (MMIO), by a load or store at a special address. These examples just should serve to illustrate the intended generality of the external-calls feature of Bedrock2, but so far, the only instantiation that was actually used is MMIO.

The parameterization over different external-calls compilers works as follows: The specification of the compiler from `FlatImp` to RISC-V is expressed as a predicate

```
Definition compiles_FlatToRiscv_correctly
 (f: funname_env Z → Z → stmt → list Instruction)
 (s: stmt): Prop := ...
```

which takes in a compilation function `f` and a `FlatImp` statement `s`, where `f` takes in the map of relative function positions described above, the relative position of the statement being compiled, and a statement, and returns a list of RISC-V instructions.

`compiles_FlatToRiscv_correctly` asserts that if execution of `s` succeeds according to the `FlatImp` semantics and all executions end up in some set of states `P`, then a RISC-V machine whose memory contains the instructions emitted by `f` called on `s` runs into a state which corresponds to a `FlatImp` state in `P`.

The main compiler from `FlatImp` to RISC-V is parameterized over the implementation of the external-calls compiler, and its correctness proof assumes the same correctness statement about the external-calls compiler as the one it proves for the main compiler:

```
Lemma compile_stmt_correct:
  (forall resvars extcall argvars,
     compiles_FlatToRiscv_correctly compile_ext_call (resvars ← extcall(argvars))) →
  (forall s,
     compiles_FlatToRiscv_correctly compile_stmt s).
```

The final correctness theorem of the compiler that composes all phases is also parameterized over the external-calls compiler, its correctness assumption, and over the semantics of external calls. Each use case can then instantiate these differently. For instance, the lightbulb (chapter 8) and the garage door (chapter 9) case studies instantiate external calls to an MMIO read and write

function that get compiled to a RISC-V load and store instruction, whereas the Softmul (chapter 10) case study does not use any external calls and therefore just uses external-call semantics that always return False, i.e. that disallow external calls, and uses a dummy external-calls compiler that returns the empty list of instructions no matter what its input is.

## 3.5 How (not) to Compose Compiler Phase Correctness Proofs

Intuitively, it seems obvious that if we have correctness proofs for some compiler phases, and we run the phases in sequence, the composed compiler pipeline is also correct. However, formalizing this intuition in such a way that the resulting proof is both *usable* and *maintainable* is surprisingly hard, and I spent a considerable amount of engineering effort on this problem. And apparently I am not alone: In his PLDI'24 keynote about CakeML, Magnus Myreen, the head of the CakeML [Kiam Tan et al., 2019] project, gave a candid account of what he believes the CakeML team did well and not so well, and his presentation contained the following anecdote: Adding new optimization phases to the CakeML compiler can make for nice undergraduate projects, and since the undergrads prove the optimization phases correct, they do not risk breaking the compiler, even if they are not experienced compiler developers and do not understand the rest of the pipeline. But, as Myreen continued, this nice story also has a less-nice side: Whenever an undergrad finishes a new optimization phase and its correctness proof, the phase has to be integrated into the pipeline, and the proof of the pipeline needs to be updated, and this step is usually done by Myreen, and he estimates that it requires about the same amount of work as the optimization itself.

### 3.5.1 Approach 0: No Explicit Concept of Phase Composition

He did not go into further detail on why it takes so much work, but one reason might be that the CakeML compiler[14] does not introduce an explicit concept for the notion of composing compiler phases: Instead, the main compiler pipeline function[15] is just a series of plain let-in expressions, one for each phase. Each individual phase correctness lemma says that the semantics of the compiled program equal (or, depending on the phase, are a subset of) the semantics of the source program, where the semantics of a language are a set of behaviors, and a behavior[16] can be to `Terminate` (with a finite list of I/O events), to `Diverge` (with a potentially infinite stream of I/O events), or to

[14] This discussion refers to CakeML v2419, available at https://github.com/CakeML/cakeml/.
[15] See definition of `compile` in `compiler/backend/backendScript.sml`
[16] See `semantics/ffi/ffiScript.sml`

`Fail`. The correctness proof[17] of this pipeline does not seem to have a very principled structure, spans over 800 lines, and invokes the individual phase correctness lemmas, as well as tactics to discharge side conditions and glue everything together.

Originally, my compiler's pipeline correctness proof also did not use any formalized structure, and each time I added (or sometimes, changed) a phase, considerable work in the pipeline correctness proof was required.

### 3.5.2 Approach 1: Chaining Simulations and State Relations

To improve the situation, I tried using and composing simulations. I used *omnisemantics forward simulations* (chapter 2), but the issues discussed in this section also apply to traditional *backward simulations*.

To illustrate with simplified formulas, let us assume that we have a notion of simulation that takes three arguments, $exec_1$, $exec_2$, and $R$, where $exec_1$ refers to the source language semantics, $exec_2$ to the target language semantics, and $R$ is a relation between source language states and target language states, asserting that the target language state contains a program obtained by compiling the program found in the source language state, as well as data corresponding to the source language state's data, potentially in a different format in case the phase introduces state representation changes.

If using omnisemantics forward simulations, simulation would be defined as follows:

$$\mathsf{simulation}\ exec_1\ exec_2\ R := \forall s_1\ s_2\ P_1, R\ s_1\ s_2 \land exec_1\ s_1\ P_1 \implies exec_2\ s_2\ (\lambda s_2'.\ \exists s_1',\ P_1\ s_1' \land R\ s_1'\ s_2')$$

Whereas, if using traditional backward simulations, it would be defined as follows:

$$\mathsf{simulation}\ exec_1\ exec_2\ R := \forall s_1\ s_2\ s_2', R\ s_1\ s_2 \land exec_2\ s_2\ s_2' \implies \exists s_1', R\ s_1'\ s_2' \land exec_1\ s_1\ s_1'$$

We can then define composition of two state relations $R_{12}$ and $R_{23}$ the usual way:[18]

$$R_{12} \circ R_{23} := \lambda\, s_1\ s_3.\ \exists s_2,\ R_{12}\ s_1\ s_2 \land R_{23}\ s_2\ s_3$$

No matter which of the two definitions of simulation we use, the following simulation composition

[17] `compile_correct'` in `compiler/backend/proofs/backendProofScript.sml`

[18] For brevity, we omit types, but of course, each state type can be a different type, so the whole definition universally quantifies over state types $S_1$, $S_2$, $S_3$, and $R_{12}$ has type $S_1 \to S_2 \to \mathsf{Prop}$, and $R_{23}$ has type $S_2 \to S_3 \to \mathsf{Prop}$.

lemma holds[19] for all $exec_1$, $exec_2$, $exec_3$, $R_{12}$ and $R_{23}$:

$$\mathsf{simulation}\ exec_1\ exec_2\ R_{12} \implies \mathsf{simulation}\ exec_2\ exec_3\ R_{23} \implies \mathsf{simulation}\ exec_1\ exec_3\ (R_{12} \circ R_{23})$$

By applying this lemma repeatedly, we can obtain a compiler correctness lemma for a whole pipeline consisting of $n$ phases, resulting in $\mathsf{simulation}\ exec_1\ exec_n\ (R_{12} \circ R_{23} \circ \cdots \circ R_{n-1,n})$.

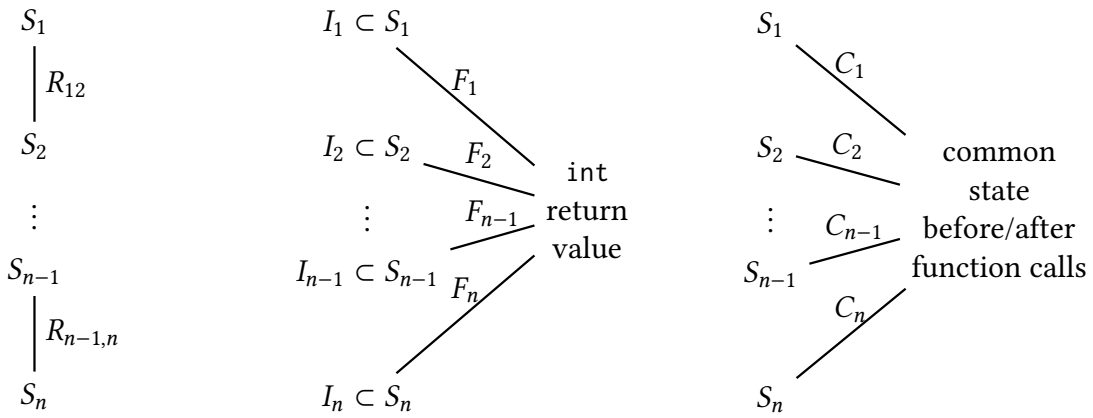
However, from a usability point of view, this solution is not great: If we want to use the compiler correctness theorem as a bring-up recipe that tells us how to initialize a system to ensure that the compiled program will run on it correctly, we need to make sure that $(R_{12} \circ \cdots \circ R_{n-1,n})\ s_1\ s_n$ holds for the initial target system state $s_n$ and for some source level state $s_1$, so we need to unfold the whole $(R_{12} \circ \cdots \circ R_{n-1,n})$, and adding new compilation phases can break usages of the compiler correctness theorem. Clearly, this is undesirable: The relations $R$ should but cannot be considered an opaque implementation detail of the compiler correctness proof.

Note that up to here, we assumed that each relation $R$ asserts that the source language state contains a source program, and the target language state contains the target program obtained from compiling the source program. Removing this condition from $R$ and mentioning it explicitly in the definition of simulation and in the simulation composition lemma improves the situation slightly, but still imposes the burden of having to plough through all of $R_{12} \circ \cdots \circ R_{n-1,n}$ in order to know how to set up a target system so that the compiler correctness theorem will hold.

### 3.5.3 Approach 2: Per-Language Initial-State and Final-State Predicates

CompCert solves the problem described in the previous section by making the relations $R$ between the states an opaque implementation detail of the compiler phase correctness proofs and by introducing, for each intermediate language, a unary predicate $I$ over states that says which states are initial states, and a predicate $F$ that relates final states to `int` exit codes. Figure 3.4b depicts this approach. The bring-up recipe for the target system is now very simple: Any initial state $s_n$ of the target system that satisfies $I_n$ is acceptable. However, this approach has two limitations: First, for each compilation phase from a language $L_1$ to a language $L_2$, it one must prove that if $I_1(s_1)$ and $I_2(s_2)$ hold, then the state relation $R_{12}\ s_1\ s_2$ holds. This condition makes it impossible to start with application-dependent initial states and essentially requires execution to start with an empty memory (or with a hard-coded, application-independent initial memory). And second, the com-

[19] Coq proofs of this lemma, for both definitions of simulation, are given in Appendix A and are straightforward.




Bring-up recipe: $s_n$ s.t. $(R_{12} \circ \cdots \circ R_{n-1,n})\ s_1\ s_n$

Bring-up recipe: $s_n$ s.t. $I_n(s_n)$

Bring-up recipe: contained in $C_n$

(a) Approach 1: Chaining simulations and relations

(b) Approach 2 (CompCert): Per-language initial-state and final-state predicate

(c) Approach 3 (Bedrock2 compiler): Per-language function-call spec based on common state

Figure 3.4: Chaining state relations vs. relating phase-specific states to a common state

posed compiler correctness theorem only talks about execution of whole programs.[20] However, being able to use the compiler correctness theorem to prove how individual compiled functions behave is crucial for the kind of reasoning done in Silver Oak[21] and in the Softmul (chapter 10) case study.

### 3.5.4 Approach 3: Per-Language Function-Call Specs

So, does that mean that if we want to get per-function correctness theorems, we need to go back to relation chaining (Approach 1 in section 3.5.2)? Not quite, as Figure 3.5 illustrates: If we can identify some state type $S_c$ that contains the common essence present in the state types of all languages, we can, in order to relate two state types $S_1$ and $S_2$, take a detour through that common state $S_c$, via two relations $C_1$ and $C_2$. Then, if we chain $n$ phases, as illustrated in Figure 3.4c, the complexity of the bring-up recipe for the lowest-level initial state does not grow proportionally

[20] There is also a correctness theorem for separate compilation, but it only says something about the behavior of a *whole program* obtained by linking *separately compiled* functions.

[21] https://github.com/project-oak/silveroak


Figure 3.5: Replacing a direct relation $R_{12}$ by a detour through a common state $S_c$

with $n$, but stays the same, namely just the lowest-level $C_n$.

In general, it might not always be possible to find a common state that can easily be related to the states of all intermediate languages, but in the case of the Bedrock2 compiler, it turns out to be possible. If we choose to look only at the state before and after function calls, the common state can be described as consisting of the following three components:

- The trace of I/O events that happened so far
- The high-level (Bedrock2 source language level) memory
- A list of fixed-size (32 or 64 bits) words that represents the function arguments before a function call or the function's return values[22] after the function call

Moreover, it turns out that for composing phases, instead of using an *exec* judgment that can execute arbitrary program snippets, it works better to use a *call* judgment that defines, for each language, how functions are called in that language. Conveniently, if we use the common state described above, *call* judgments can have the same generic type signature in all languages, namely

$$P \to \texttt{string} \to \texttt{trace} \to \texttt{mem} \to \texttt{list word} \to (\texttt{trace} \to \texttt{mem} \to \texttt{list word} \to \texttt{Prop}) \to \texttt{Prop}$$

where $P$ is the type of program ASTs and is the only aspect that differs across languages, the `string` represents the name of the function to be called, the `trace` and `mem` represent the state before the function call, the following `list word` represents the function arguments, and the last argument is a postcondition over the state and return values after the function call. So, for a *compile* function from a source language with a call judgment $call_1$ to a target language with a call judgment $call_2$, we can define its correctness as follows:

$$\mathsf{phase\_correct}\ \ compile\ \ call_1\ \ call_2 :=$$
$$\forall p_1\ p_2,\ compile\ p_1 = \mathsf{Some}\ p2 \implies \forall f\ t\ m\ args\ Q,\ call_1\ p_1\ f\ t\ m\ args\ Q \implies call_2\ p_2\ f\ t\ m\ args\ Q$$

As shown in the Coq code in Figure A.3 in Appendix A, it is trivial to compose this notion of

[22] Note that unlike in C, Bedrock2 functions can have multiple return values.

phase_correct to combine correctness proofs of individual phases into a correctness proof of a whole compiler pipeline.

To give two examples of *call* judgments, here is a (slightly simplified) call judgment for the Bedrock2 source language:

```
Definition call e t m argvals Q := exists argnames retnames fbody l,
  map.get e f = Some (argnames, retnames, fbody) ∧
  map.of_list_zip argnames argvals = Some l ∧
  exec e fbody t m l (fun t' m' l' ⇒ exists retvals,
    map.getmany_of_list l' retnames = Some retvals ∧ Q t' m' retvals).
```

At this level, programs are function environments $e$ that map function names to triples of their argument names, return names, and function bodies, and the `exec` judgment is the big-step omnisemantics judgment that gives the semantics of any statement, with respect to a state that consists of a trace `t`, a memory `m`, and a map of local variables `l`.

On the other hand, at the RISC-V level, the type of a program is a triple consisting of a list of instructions, a position map mapping each function name to the relative position of the function within the instruction list, and an integer computed by the compiler that indicates the required stack size[23] to run the program. The definition of $call\ (instrs, posmap, stacksize)\ f\ t\ m_H\ argvals\ Q$ at the RISC-V level is a bit lengthy[24] but can be summarized in English as follows: The function position map $posmap$ needs to contain the relative position $p_f$ of the function named $f$. Then, if one starts with a RISC-V machine that satisfies the following conditions:

- The argument registers contain $argvals$.
- Below[25] the address stored in the stack pointer register, there are at least $stacksize$ words of memory available.
- The instructions $instrs$ are in the machine memory at some address $p_{funcs}$.
- The program pointer points to $p_{funcs} + p_f$.
- The machine memory contains the source-level memory $m_H$.

Then, if one runs the machine according to riscv-coq semantics, it will eventually reach a state satisfying the following conditions:

- The argument registers contain return values accepted by the postcondition $Q$.

[23] Since we do not support recursive calls and only allow constant-sized stack allocation, this number is easy to calculate.

[24] https://github.com/mit-plv/bedrock2/blob/2223b2a2f7/compiler/src/compiler/LowerPipeline.v#L378

[25] We say "below" instead of "at" because the stack grows downwards.

- The stack pointer is the same as it was before.
- The instructions are still at the same location in the memory.
- The program pointer points to the address that was stored in the return address register before the function call.
- The machine memory contains updated source-level memory $m'_H$ that is accepted by the postcondition $Q$.
- The updated I/O event trace is accepted by the postcondition $Q$.
- All registers except the caller-saved registers are still the same as before the function call.

Note that no matter how many phases we stack on top of the last phase, the final correctness theorem will always be in terms of the RISC-V *call* judgment described above, which might seem a bit lengthy, but is precise enough to allow calling individual functions, and can still be simplified further when the full source program is known, as shown in the top-level theorems of each of the case studies in Part II.

### 3.5.5 Conclusion

Approach 3 so far has satisfied all our requirements: It results in a *usable* compiler correctness theorem, in particular, one that can serve as a bring-up recipe on how to initialize a RISC-V machine before running a single function (as opposed to being able to run only whole programs). Moreover, it is also *maintainable*, in the sense that inserting new phases into the pipeline does not break usages of the overall compiler correctness theorem, and in the sense that as a part of his undergraduate project, Arthur Reiner de Belen was able to integrate his two optimizations (UseImmediate and DeadCodeElim) into the pipeline on his own.

# Chapter 4

# Formal Semantics For an Industrial ISA[1]

A formal, executable specification of the RISC-V instruction-set architecture (ISA) was developed in Haskell by Ian Clester and Thomas Bourgeat.[2] As explained further in [Bourgeat et al., 2023], it was designed to support many different use cases, including simulation of RISC-V programs with different kinds of I/O, interactive theorem proving, model-checking of all possible program executions under weak memory, and compilation to hardware circuits.

During my PhD, I implemented and maintained the interactive theorem proving use case, called riscv-coq[3] for short. I started the project in February 2018, and at the time of writing, July 2024, it is still active, and I am mentoring an MEng student on extending the specification with support for the vector extension.

This chapter will give an overview of the organization of the RISC-V specification in section 4.1, show in section 4.2 how large parts of the Haskell specification can automatically be translated to Coq, and explain in section 4.3 how the parts that were deliberately left unspecified in the generic specification can be instantiated to obtain a specification that is suitable for interactive theorem proving.

I consider the main result of this chapter to be the software artifact: It is the first formal specification of an industrial ISA that has been used to prove a compiler as well as a processor against it in such a way that the ISA specification cancels out in an end-to-end theorem. And additional insight is how to (and how not to) bridge definitions in monadic style to weakest-precondition

[1] This chapter of the dissertation contains text copied and adapted from the ICFP'23 paper I co-authored with Thomas Bourgeat, Ian Clester, Andres Erbsen, Pratap Singh, Andy Wright, and Adam Chlipala [Bourgeat et al., 2023], as well as text copied and adapted from the PLDI'21 paper (and previous, unpublished longer versions of it) I co-authored with Andres Erbsen, Joonwon Choi, Clark Wood, and Adam Chlipala [Erbsen et al., 2021].

[2] The Haskell code is available at https://github.com/mit-plv/riscv-semantics.

[3] The Coq code and translation setup is available at https://github.com/mit-plv/riscv-coq.

style and omnisemantics style (section 4.3.5), and I am grateful to Andres Erbsen for showing me and convincing me of the easier (free-monads based) solution.

## 4.1 Abstracting Over Use Cases

In order to be reusable for different use cases, the semantics in Haskell deliberately only specify how each RISC-V instruction is defined in terms of a small number of *primitives* (listed in Figure 4.1) such as e.g. reading and writing registers or memory, but do not give semantics to these primitives. They also do not specify any data type representing the state of a RISC-V machine, instead only specifying how a RISC-V instruction is turned into a sequence of such primitives. For instance, the store-word instruction `Sw` is turned into a sequence of the four primitives `getRegister`, `translate` (performing virtual-to-physical address translation, which, up to now, is just the identity function in Coq), `getRegister`, and `storeWord`, as shown in Figure 4.2.

To abstract over the kinds of effects that the primitives can have, monads [Wadler, 1992] are used, and to make that abstraction explicit, a typeclass [Wadler and Blott, 1989] is used, as indicated by `` `{Monad M} `` in Figure 4.1.

Moreover, another typeclass `` `{MachineWidth t} `` is used to indicate that `t` is an integer type that has all the operations that a RISC-V ALU (arithmetic and logic unit) of a processor needs, while keeping its bitwidth unspecified, so that the specification can later be instantiated to either 32-bit or 64-bit integers.

## 4.2 Translating Haskell to Coq

The goal of the formal RISC-V specification in Haskell is to provide a machine-readable reference specification that can be shared by many different use cases, so that the specification does not need to be rewritten for each new use case. Therefore, we want to generate automatically as much of the Coq specification as possible from the Haskell specification.

Using hs-to-coq [Breitner et al., 2018], we can translate the Haskell specification to Coq, replacing designated Haskell library functions with corresponding Coq library functions. Since hs-to-coq was designed to model Haskell semantics in Coq as faithfully as possible, it ships with handwritten and auto-generated translations of Haskell's standard-library files, and by default they are referenced by the Coq files produced by hs-to-coq. However, for this project, we were not seeking a faithful reproduction of Haskell semantics in Coq but rather an idiomatic RISC-V

```
Inductive PrivMode: Set := User | Supervisor | Machine.

Inductive SourceType: Set := VirtualMemory | Fetch | Execute.

Class RiscvProgram{M}{t}`{Monad M}`{MachineWidth t} := mkRiscvProgram {
  getRegister: Register → M t;
  setRegister: Register → t → M unit;

  loadByte   : SourceType → t → M w8;
  loadHalf   : SourceType → t → M w16;
  loadWord   : SourceType → t → M w32;
  loadDouble : SourceType → t → M w64;

  storeByte   : SourceType → t → w8 → M unit;
  storeHalf   : SourceType → t → w16 → M unit;
  storeWord   : SourceType → t → w32 → M unit;
  storeDouble : SourceType → t → w64 → M unit;

  makeReservation  : t → M unit;
  clearReservation : t → M unit;
  checkReservation : t → M bool;

  getCSRField : CSRField → M MachineInt;
  setCSRField : CSRField → MachineInt → M unit;

  getPC: M t;
  setPC: t → M unit;
  getPrivMode: M PrivMode;
  setPrivMode: PrivMode → M unit;
  fence: MachineInt → MachineInt → M unit;

  endCycleNormal: M unit;
  endCycleEarly: forall A, M A;
}.
```

Figure 4.1: The primitives, hand-translated to Coq

```
match instr with
| Sw rs1 rs2 simm12 ⇒
    a ← getRegister rs1;
    addr ← translate Store 4 (add a simm12);
    x ← getRegister rs2;
    storeWord Execute addr (regToInt32 x)
| ...
end
```

Figure 4.2: Semantics of the store-word instruction

specification in Coq. Therefore, we used hs-to-coq's *edit files* feature, which allows one to provide renaming and rewriting patterns to be applied during the translation, so that we could map all Haskell standard-library references to reasonably close Coq equivalents and obtain an idiomatic, Haskell-independent Coq specification. We used hs-to-coq to translate the files specifying instruction execution, the instruction decoder, as well as the CSR-file specification, while we manually wrote remaining files like utility definitions, the definition of the `RiscvMachine` typeclass, and proof-specific files.

## 4.3 Typeclass Instances for Interactive Theorem Proving

In order to prove a compiler or processor correct against these semantics, we need to give precise semantics to the primitives, and we do so by fixing an explicit data type representing the state of a RISC-V machine, and implementing the primitives for this state type. We have both a deterministic implementation which allows us to run small RISC-V programs in Coq, as well as a nondeterministic implementation that can read nondeterministically chosen input through MMIO that we use for proofs. For instance, here is the implementation of the `getRegister` primitive:

```
getRegister reg :=
  if Z.eq_dec reg Register0 then Return (ZToReg 0) else
    if (0 <? reg) && (reg <? 32) then
      mach ← get;
      match map.get mach.(getRegs) reg with
      | Some v ⇒ Return v
      | None ⇒ Return (word.of_Z 0)
      end
    else fail_hard
```

Moreover, we define that there is no virtual memory, and that the physical memory is sequentially consistent. That is, it can be modeled as a partial map from addresses to bytes, and each write to a valid address is immediately visible to subsequent reads. However, since we want to allow pipelined processors, which could execute a store to an address containing an instruction that is already in the pipeline, we introduce the restriction that only addresses that have never been written to since the last fence instruction may be fetched.

In the following, we describe different possible instantiations of the typeclass in Figure 4.1.

### 4.3.1 Simulator in Coq

**State monad** In Coq, the simplest-possible instantiation of the monad is `p := State MachineState`, where `State` is the state monad defined as `State(S A: Type) := S → (A * S)`, and `MachineState` is a record containing the values of the processor's registers, the program counter, the memory, the CSR file, and the current privilege level. This instantiation can be used to obtain a deterministic RISC-V simulator.

**State monad with failure** An arguably simpler monad instantiation is `p := OState MachineState`, where `OState(S A: Type) := S → (option A) * S` uses a `None` answer to indicate that a failure occurred. Its `Bind` and `Return` operations are implemented as

```
Bind A B (m: OState S A) (f: A → OState S B) :=
  fun (s: S) ⇒ match m s with (Some a, s') ⇒ f a s' | (None, s') ⇒ (None, s') end;
Return A (a: A) := fun (s: S) ⇒ (Some a, s)
```

and an unrecoverable (hard) failure can be implemented as

```
fail_hard S A: OState S A := fun (s: S) ⇒ (None, s)
```

For compiler-correctness proofs, the always-failing function `fail_hard` can be used to indicate that a situation occurred that the compiler is supposed to avoid, e.g. memory access at an invalid address, and a compiler-correctness proof then states that all valid source programs are translated to RISC-V programs that never fail.

Moreover, if the compiler has been designed to emit code that does not use certain features, the RISC-V specification can be simplified by implementing the primitives of Figure 4.1 used by these features as just `fail_hard`. For instance, the Bedrock2 compiler (chapter 3) emits code that does not depend on the CSRs, does not use floating-point operations or atomics, and assumes that there is no virtual memory and that the code always runs at the `MachineMode` privilege level. Therefore, the

monad instantiation used to specify its correctness implements the primitives `makeReservation`, `checkReservation`, `clearReservation`, `getCSRField`, as well as `unsafeSetCSRField`, `getPrivMode`, `setPrivMode` as just `fail_hard` (while the TLB- and floating-point-related methods were omitted altogether in the translation from Haskell to Coq).

### 4.3.2 Adding Instruction Counters

In a separate project (unpublished at the time of writing) building on top of the compiler mentioned above, the `MachineState` record was extended to include counters for the number of executed instructions, number of memory accesses, and number of jumps, and proofs about how the compiler preserves these cost metrics were written, allowing one to calculate loose (but formally proven) upper bounds on the execution time of RISC-V programs.

### 4.3.3 Nondeterminism

The first way that we implemented to add nondeterminism is to use the nondeterministic option state monad, `OStateND S A := S → option (A * S) → Prop`, where the `option`'s `None` constructor is used to indicate failure, and `option (A * S) → Prop` can be thought of as the set of all possible outcomes. Its `Bind` and `Return` operations are implemented as

```
Bind A B (m: OStateND S A)(f : A → OStateND S B) := fun (s : S) (obs: option (B * S)) ⇒
  (m s None ∧ obs = None) ∨ (∃ a s', m s (Some (a, s')) ∧ f a s' obs);
Return A (a : A) := fun (s : S) (oas: option (A * S)) ⇒ oas = Some (a, s)
```

**Why not monad transformers?** We use monad transformers [Liang et al., 1995] to add logging or early returns in Haskell, but they do not work to add nondeterminism as an additional feature on top of an existing instance, because in order to obtain the right type for `OStateND`, one has to *start* the composition with the nondeterminism monad, rather than adding nondeterminism *at the end* of the monad-transformer composition chain, as would be required to reuse code written for `OState` in code for `OStateND`. Moreover, since this code serves as a specification, it should be easy to audit and understand, and we found that the definitions of `Bind` and `Return` above are much easier to digest than the composition of several monad transformers, where certain composition orders can result in unintended semantics.

### 4.3.4 Runtime Input

Once we have nondeterminism, we can use it to model memory-mapped I/O (MMIO). For instance, in the implementation of the `loadWord` primitive, if the address is not a physical memory address, we delegate to the following helper function:

```
mmio_load32 addr: OStateND S int32 := fun s oas ⇒
  (isMMIOAddr addr ∧ ∃ v: int32, oas = Some (v, (appendLog (mmioLoadEvent addr v) s))) ∨
  (~isMMIOAddr addr ∧ oas = None)
```

It can be read as a function that for each current state `s` returns a proposition that indicates whether an outcome `oas` (of type `option (int32 * MachineState)`) is in the set of possible outcomes, distinguishing two cases based on whether the address lies in the address range reserved for MMIO. We also augment `MachineState` with a log to which we append an MMIO event on each load and store that falls into the MMIO address range.

Proofs of a compiler targeting this specification have to show that all states in the outcome set given by `mmio_load32` satisfy the compiler's correctness guarantees (such as being related to a state of the source-language execution), so the body of `mmio_load32` will appear on the left-hand side of an implication, so the existentially quantified `v` becomes universally quantified, and as expected, the compiler proof must establish a guarantee for all possible read values `v`.

### 4.3.5 Nondeterminism by Means of Weakest Preconditions

The Bedrock2 compiler using our RISC-V specification requires RISC-V semantics that given an initial state `s`, a monadic computation `m` corresponding to the execution of a sequence of primitives from Figure 4.1, and a desired *postcondition*, returns the weakest precondition that must hold in order for the postcondition to hold. Therefore, it seems that we need the following bridge definition that tells when a monadic `OStateND` computation satisfies a postcondition:

```
Definition mcomp_sat{S A: Type}(m: OStateND S A)(s: S)(post: A → S → Prop): Prop :=
  forall (o: option (A * S)), m s o → exists a s', o = Some (a, s') ∧ post a s'.
```

For an example relating this definition to the previous subsection, `m` could be instantiated with `mmio_load32 addr`, and `post` could be instantiated with the claim that the final state is related to a state of the source-language execution.

When instantiating `m` with a monadic computation involving many `Binds`, unfolding `mcomp_sat` and all `Binds` quickly leads to huge formulas with one existential for each intermediate state and answer. For instance, each of the three semicolons in Figure 4.2 stands for a monadic `Bind` that unfolds

to a disjunction, two conjunctions, and an existential, as per the definition of `Bind` in section 4.3.3. We found these formulas to be larger than what human brains can deal with productively.

The solution was to treat `mcomp_sat` and `Bind` as opaque and to prove weakest-precondition-style rules for each primitive of Figure 4.1, using only these rules in the compiler-correctness proof, so that the large formulas were confined to just the proofs of these rules. However, when we proved a processor in the Coq-embedded hardware-description language Kami [Choi et al., 2017] against this RISC-V specification, the same formula-explosion problem struck again, but this time, on the other (left-hand) side of the implication. Inversion rules for `mcomp_sat` of primitives, dual to the weakest-precondition-style rules mentioned above, might have been a way to go, but, as my colleague Andres Erbsen pointed out, it is simpler (both for the compiler and the processor) to use a typeclass instantiation that does not use `OStateND` and is more suitable for weakest-precondition generation, namely a free monad. To fix the problem, it seems desirable to define `mcomp_sat` directly for each primitive (as opposed to first defining each primitive in terms of the cumbersome `OStateND`, and then using one generic definition to go from `OStateND` to postconditions). We can do so using a different instantiation of our `RiscvProgram` typeclass that materializes monadic computations into an **`Inductive`** with a constructor for each primitive from Figure 4.1, with an alternative definition of `mcomp_sat` that gives the weakest-precondition interpretation of this syntax. This monad is similar to freer monads [Kiselyov and Ishii, 2015] and interaction trees [Xia et al., 2019].

The crucial difference between `OStateND` and the freer-monad interpreter is that the former creates an existential for the intermediate state and answer of each `Bind`, whereas the latter works similarly to a continuation-passing-style interpreter and just passes updated states to the right-hand sides of the `Bind`s, leading to considerably simpler formulas. For comparison, the `mmio_load32` helper function now looks as follows:

```
mmio_load32 addr := fun s post ⇒
  isMMIOAddr addr ∧ ∀ v: int32, post v (appendLog (mmioLoadEvent addr v) s)
```

Note how, contrary to `OStateND`, no case for failure is needed, and the value `v` being read is already universally quantified, rather than existentially quantified on the left-hand side of the implication of `mcomp_sat`, and if more code follows after this snippet, it will be put into `post` and thus be invoked with the updated state `(appendLog (mmioLoadEvent addr v) s)`, with no intermediate existential.

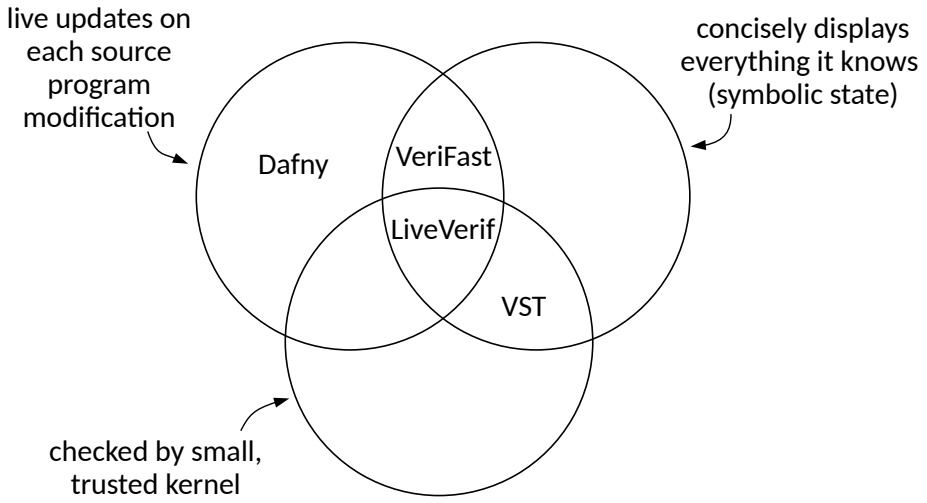
# Chapter 5

# Live Verification[1]

We show that an interactive proof assistant with an extensible parser and a proof goal display (such as e.g. Coq) can be turned into a live verification tool, that is, a tool that enables programmers to verify their code as they write it in real-time. After each line of code that the programmer writes, the tool tells the programmer whether it was able to prove absence of undefined behavior so far, and it displays a concise representation of the symbolic state of the program right after the added line. The user can then either write the next line of code, or if needed or desired, write a specially marked comment that provides hints on how to solve side conditions or on how to represent the symbolic state more nicely. Once the programmer has finished writing the program, it is already verified with a mathematical correctness proof.

Other tools providing real-time feedback already exist, but ours is the first to combine all the three benefits shown in the Venn diagram: Like Dafny [Leino and Wüstholz, 2014] and VeriFast [Jacobs et al., 2011], our tool does not require any recompilation or restarting on source code modifications like VST [Cao et al., 2018] does; and like in VST, the proofs are checked by a small, trusted kernel, so that the implementation of the tool (except for the kernel) need not be trusted. Moreover, our tool uses the proof assistant's goal display to show a concise summary of the current symbolic state, which is missing in Dafny and can only be achieved by querying the prover with user-stated assertions, to check which ones are provable.

[1] This chapter of the dissertation contains text copied and adapted from the PLDI'24 paper I co-authored with Viktor Fukala and Adam Chlipala [Gruetter et al., 2024b].




For the proof assistant, we use Coq, but it seems possible to do the same in other interactive proof assistants such as e.g. Lean or Isabelle as well; and as the source language, we use a subset of C (Bedrock2), but here too, the approach should be applicable to other source languages as well.

## 5.1 Introduction

Writing proofs about software can be a repetitive task, but fortunately, like many repetitive tasks, it can be automated by writing programs that perform it. But often, it is hard to find the right level of automation: One might think that the more automation, the better, but the more automated a prover is, the more it is at risk of going down a wrong route in its proof search and wasting time on proof steps that a human could easily recognize as useless. The reason is that, for a typical nontrivial program, the programmer has some (potentially domain-specific) insight about its correctness and about what strategies are promising to try, but the verification tool might not have this knowledge. An important question is therefore (a) how users can convey insight to the verifier. Equally important, but often neglected, is the opposite direction, i.e. (b) how the verifier can convey everything it knows to the user. This feature can be useful in two ways: If, while the user is writing a program, the verifier constantly provides a concise summary of everything it knows to be true at the current cursor position (also known as a symbolic state), this summary can help the user decide whether the program is correct up to that point, and it can hint at what the right next command in the program might be. And if the verifier fails to verify that an instruction is safe (for example, that an array access is within bounds), by looking at this summary of everything that the verifier knows, the user can guess more quickly why the verifier failed.

Our answer to (a) is to use Coq's tactic language Ltac both to implement the verifier and as the language in which users express their domain-specific insights, which leads to smooth cooperation between the two; and our answer to (b) is to use Coq's proof-goal display (which consists of a list of hypotheses that can be assumed and a conclusion that has to be proven) to display the current symbolic state of the program that the user is writing.

We start with some "clever tricks" to allow single source files to be accepted as legal code in both Coq and C, where interactive proof scripts appear amidst lines of normal C code. Then we add features that take advantage of the proof assistant, providing snapshots of "just right" complexity, describing what the framework inferred about all possible program states at particular code points. Along the way, we develop ideas that may mitigate some of the classic usability challenges of verification with Hoare logics, like the need to invent loop invariants out of whole cloth.

More specifically, we make the following contributions:

- We present a prototype of a framework that supports symbolic live debugging of (a subset of) C. It runs entirely within the Coq proof assistant and produces ASTs of the functions' source code as objects in Coq, with a correctness lemma for each function. Our tool's correctness need not be trusted, because it produces proofs that are verified by Coq's kernel. The correctness lemmas are expressed in terms of Bedrock2's source-language semantics [Erbsen et al., 2021], so our programs can be compiled with Bedrock2's verified RISC-V compiler.
- Most software-verification tools require users to provide loop invariants, which can become quite long and tedious to write down. We present a way to express a loop invariant as a diff from the inferred symbolic state at the beginning of the loop (section 5.3.1.7 and section 5.4.4). Using some tactics, users can generalize and/or strengthen the symbolic state, and our framework can then use this modified symbolic state as the loop invariant. So the user still needs to provide the insight that leads to a suitable loop invariant, but it is not necessary to spell out the whole loop invariant. This solution potentially leads to an easier, more intuitive, and more enjoyable user experience and to proofs that are more robust against code changes, because diffs (edits) tend to be smaller than whole invariants.
- We argue that proof automation should optimize the user experience for failing proofs (the default case in a proof developer's day-to-day work) rather than for proofs where everything works, and we describe three principles that emerge from this focus (section 5.4.8), including centering automation of side-condition solving around the notion of *safe steps* (section 5.4.8.3), i.e. proof steps that do not turn provable goals into unprovable goals. We provide users with means to register domain-specific proof steps, enabling proofs that rely on backtracking only very locally and thus are both automated and easy to debug at the same

time.

- If one is willing to trust our tool's notation-based parser, our polyglot Coq source files can also be viewed as C files and compiled with GCC, or if one is willing to trust Bedrock2's C pretty-printer, one can pretty-print the ASTs to C and compile with GCC (section 5.3.2).
- We developed and verified a small but promising set of functions (section 5.6) in our framework.

### 5.1.1 A First Glance At an Example

Figure 5.1 shows an example of a verified `memset` function. The file is a Coq file, but if we prefix it with an opening C comment `/*`, it becomes a C file. Lines 15 to 26 look like C code but are in fact just notations for proof tactics that gradually build the abstract syntax tree (AST) of the function, along with its correctness proof. The proof is completely automated, except for 5 lines of tactic code (lines 17-21, shown in Figure 5.4c) that express the desired loop invariant as a diff from the symbolic state before the loop. We will discuss this example in more detail in section 5.3.1, but we first provide some background in section 5.2.

## 5.2 Background

This section provides some background to make the text accessible to readers without prior knowledge of proof assistants, Coq, or program verification inside proof assistants. For each subsection, it should be safe to decide whether to skip it based on its title.

### 5.2.1 Weakest-Precondition Generators

A weakest-precondition generator wpgen takes a command $c$ and a postcondition $P$ (an assertion over the final program state) and returns the weakest precondition that the initial state must satisfy in order for the postcondition to hold after $c$ is executed. For example, the cases for the assignment and sequencing commands of a wpgen for a simple imperative language, where program states $s$ are just partial mappings from variable names to values, could be defined as follows:

$$\begin{aligned} \mathsf{wpgen}\ (x := e)\ P &:= \lambda s.\ \exists v.\ \mathsf{eval\_expr}\ s\ e\ v \wedge P\ s[x := v] \\ \mathsf{wpgen}\ (c_1; c_2)\ P &:= \mathsf{wpgen}\ c_1\ (\mathsf{wpgen}\ c_2\ P) \end{aligned}$$

The relation eval_expr $s$ $e$ $v$ (whose definition we omit) says that evaluating expression $e$ in state $s$ results in value $v$. It is a relation rather than a function because it is partial: If $e$ contains a

```
1  (* -*- eval: (load-file "../LiveVerif/live_verif_setup.el"); -*- *)
2  Require Import LiveVerif.LiveVerifLib.
3  Load LiveVerif.
4  #[export] Instance spec_of_memset: fnspec :=                                  .**/
5
6  void memset(uintptr_t a, uintptr_t b, uintptr_t n) /**#
7    ghost_args := bs (R: mem → Prop);
8    requires t m := <{ * array (uint 8) \[n] bs a
9                       * R }> m ∧
10                  \[b] < 2 ^ 8;
11   ensures t' m' := t' = t ∧
12        <{ * array (uint 8) \[n] (List.repeatz \[b] \[n]) a
13           * R }> m' #**/                                          /**.
14 Derive memset SuchThat (fun_correct! memset) As memset_ok.              .**/
15 {                                                                  /**. .**/
16   uintptr_t i = 0;                                                 /**.….**/
22   while (i < n) /* decreases (n ^- i) */ {                        /**. .**/
23     store8(a + i, b);                                              /**. .**/
24     i = i + 1;                                                     /**. .**/
25   }                                                                /**. .**/
26 }                                                                  /**. Qed.
27 End LiveVerif. Comments .**/ //.
```

Figure 5.1: `memset` example, as displayed in Emacs, with lines 5 of Ltac (lines 17-21) folded away into …

variable that is not in $s$, no $v$ can be found.

If we evaluate wpgen on a sample program and postcondition using call-by-value evaluation order, we effectively step through the program backwards, propagating an ever-growing postcondition through the program backwards:

$$
\begin{aligned}
& \mathsf{wpgen}\ (b := 2 * a; c := b/2)\ (\lambda s.\ \exists r.\ (c, r) \in s \wedge (a, r) \in s) \\
= \ & \mathsf{wpgen}\ (b := 2 * a)\ (\mathsf{wpgen}\ (c := b/2)(\lambda s.\ \exists r.\ (c, r) \in s \wedge (a, r) \in s)) \\
= \ & \mathsf{wpgen}\ (b := 2 * a)\ (\lambda s.\ \exists v.\ \mathsf{eval\_expr}\ s\ (b/2)\ v \wedge \exists r.\ (c, r) \in s[c := v] \wedge (a, r) \in s[c := v]) \\
= \ & \lambda s.\ \exists v_0.\ \mathsf{eval\_expr}\ s\ (2 * a)\ v_0 \wedge \exists v.\ \mathsf{eval\_expr}\ s[b := v_0]\ (b/2)\ v \wedge \\
& \exists r.\ (c, r) \in s[b := v_0][c := v] \wedge (a, r) \in s[b := v_0][c := v]
\end{aligned}
$$

At the end (and after also unfolding the definition of eval_expr that we omit here), this process results in a purely logical formula over the initial state that represents the weakest precondition that needs to hold in order for the postcondition $(\lambda s.\ \exists r.\ (c, r) \in s \wedge (a, r) \in s)$ to hold after executing the program snippet.

A simpler precondition for this program snippet that a user might write down as a specification would be $\lambda s.\ \exists x.\ s = \{(a, x)\}$, and using an automated or interactive prover, we can prove that it implies the computed weakest precondition, thus obtaining a proof that all executions of this snippet starting from a state satisfying the precondition will satisfy the postcondition.

### 5.2.2 *Forward* Symbolic Execution Using a *Weakest-Precondition* Generator

According to the viewpoint in section 5.2.1, weakest preconditions are backwards-oriented: One steps through the program backwards, gradually transforming the postcondition until the weakest precondition of the whole program is obtained.

However, this viewpoint implicitly assumes that the weakest-precondition generator is executed in standard call-by-value evaluation order.

So here is an insight that is considered folklore in communities around the Iris Proof Mode [Krebbers et al., 2017] and Bedrock2, but might be less well-known in other communities:

*By evaluating a weakest-precondition generator in normal-order evaluation (i.e. left-to-right), one can symbolically evaluate a program in forward direction.*

We will illustrate with the same example as in the previous section, but to make it look more intuitive, we change the argument order of wpgen from wpgen $c\ P\ s$ to wpgen $c\ s\ P$, so that the initial state $s$, which comes first in time before the postcondition $P$, also comes first in the argument order.

If we encounter eval_expr occurrences, we will simplify them on the fly, and we will also instan-

tiate all the existentials that we encounter, since they are all uniquely determined and only serve to express the partiality of map lookups. We mark existential-instantiation steps with a backwards implication $\Leftarrow$.

So, by simplifying a wpgen formula in left-to-right evaluation order, we can prove that a program satisfies a postcondition, and the proof steps correspond to a forward symbolic execution of the program:

$$
\begin{aligned}
& \mathsf{wpgen}\ \{(a,x)\}\ (b := 2*a; c := b/2)\ (\lambda s.\ \exists r.\ (c,r) \in s \wedge (a,r) \in s) \\
= \ & \mathsf{wpgen}\ \{(a,x)\}\ (b := 2*a)\ (\lambda s'.\mathsf{wpgen}\ (c := b/2)\ s'\ (\lambda s.\ \exists r.\ (c,r) \in s \wedge (a,r) \in s)) \\
= \ & \exists v.\ \mathsf{eval\_expr}\ \{(a,x)\}\ (2*a)\ v \\
& \quad \wedge \mathsf{wpgen}\ (c := b/2)\ (\{(a,x)\}[b := v])\ (\lambda s.\ \exists r.\ (c,r) \in s \wedge (a,r) \in s) \\
\Leftarrow \ & \mathsf{wpgen}\ (c := b/2)\ \{(a,x),(b,2*x)\}\ (\lambda s.\ \exists r.\ (c,r) \in s \wedge (a,r) \in s) \\
= \ & \exists v.\ \mathsf{eval\_expr}\ \{(a,x),(b,2*x)\}\ (b/2)\ v \\
& \quad \wedge (\lambda s.\ \exists r.\ (c,r) \in s \wedge (a,r) \in s)(\{(a,x),(b,2*x)\}[c := v/2]) \\
\Leftarrow \ & \exists r.\ (c,r) \in \{(a,x),(b,2*x),(c,2*x/2)\} \wedge (a,r) \in \{(a,x),(b,2*x),(c,2*x/2)\} \\
= \ & \exists r.\ r = 2*x/2 \wedge r = x \\
\Leftarrow \ & \top
\end{aligned}
$$

Note how the symbolic state, i.e. the map expression between $\{\}$, evolves from line to line exactly as it would in a forward symbolic evaluation of the program.

### 5.2.3 Using WP Rules instead of a WP Generator

Instead of using a weakest-precondition *generator* wpgen whose definition we selectively unfold to step through a program, we can also use an equivalent[2] weakest-precondition *judgment* wp (defined in terms of the operational semantics of the object language), prove rules for each object-language construct, and apply these rules in order to step through a program.

So, instead of using wpgen definitions like those in section 5.2.1, we can also use rules like the following:

$$
\begin{aligned}
\forall\ x\ e\ s\ P\ v.\ \mathsf{eval\_expr}\ s\ e\ v \wedge P\ s[x := v] \quad &\Rightarrow \quad \mathsf{wp}\ (x := e)\ s\ P \\
\forall\ c_1\ c_2\ s\ P.\ \mathsf{wp}\ c_1\ s\ (\lambda s'.\ \mathsf{wp}\ c_2\ s'\ P) \quad &\Rightarrow \quad \mathsf{wp}\ (c_1; c_2)\ s\ P
\end{aligned}
$$

[2]Note that we avoid some complications by considering only object languages without lambdas and by using a metalanguage (logic) that supports quantification over predicates. In particular, if program variables in the symbolic state could not only contain simple values like integers or Booleans but also lambdas, defining wpgen by structural recursion over the program syntax would not be directly possible anymore, and in the case of while loops, we rely on the ability to quantify existentially over an invariant:
wpgen $s$ (**while** $e$ **do** $c$) $P := \exists I.\ I\ s \wedge \forall s'.\ I\ s' \Rightarrow \exists b.\ \mathsf{eval\_expr}\ s'\ e\ b \wedge (b = \mathbf{true} \Rightarrow \mathsf{wpgen}\ s'\ c\ I) \wedge (b = \mathbf{false} \Rightarrow P\ s')$

Using such rules, we can perform the same kind of proofs as described in section 5.2.2 (with the minor difference that all = signs now become ⟸). From a logical point of view, selectively unfolding wpgen left-to-right is equivalent to repeatedly applying wp rules to the leftmost occurrence of wp, and the only relevant differences are Coq-specific proof ergonomics and performance considerations that we defer to section 5.5.8.3.

### 5.2.4 Editing Coq Proofs: Proof Goals and the Proof Cursor

The central notion for interactive proof development in Coq is that of a *proof goal*. On paper, we write proof goals as $\Gamma \vdash P$, where $\Gamma$ is a list of variables and hypotheses that can be assumed, and $P$ is the conclusion to be proven. In the actual Coq implementation, each variable and hypothesis is printed on a separate line, and the $\vdash$ is printed as a horizontal line (for example, see Figure 5.4a & b).

When a user wants to develop a proof of a statement $P$ in Coq, Coq starts by showing the proof goal $\vdash P$. The user then applies a series of tactics. Each tactic either completely solves a proof goal or transforms it into one or several new proof goals. There are tactics for modifying the conclusion (backwards reasoning) as well as tactics for modifying the hypotheses (forward reasoning), and all their operations are justified by previously proven lemmas. For instance, given a goal of the form $H : A \wedge B \vdash C \wedge D$, the tactic `destruct H` would transform it into $H_1 : A, H_2 : B \vdash C \wedge D$, whereas the `split` tactic would transform it into *two* new subgoals, $H : A \wedge B \vdash C$ and $H : A \wedge B \vdash D$.

Under the hood, this process generates a *proof term*. Once there are no more proof goals left, the user can conclude the proof with the **`Qed`** command, and Coq's small, trusted kernel checks that this proof term is indeed a proof of the original statement. Because of this rechecking by the kernel, none of the goal-management code or the tactics code needs to be trusted in order to trust the correctness of the proven statement, which leads to highly trustworthy proofs.

ProofGeneral is an Emacs extension for developing Coq proofs. For each Coq file being edited, it shows three windows: a window for the file itself, a window for the current proof goals, and a window to display error messages. In addition to the regular text-editing cursor, the file window also has a *proof cursor* that can be moved forward and backward using separate key bindings (or GUI buttons), and the proof-goal window always displays the proof goals that remain open at the current position of the proof cursor. If a proof contains an error, ProofGeneral ensures that the proof cursor can never be advanced past that error.

We will see in section 5.3.1 how we can repurpose the proof-goal window to serve as a debugger window displaying the symbolic values of all variables and memory, and how the proof cursor can

be seen as the indicator of a debugger pointing to the next instruction to be executed.

### 5.2.5 Evars in Coq: Lazily Instantiated Existential Variables

While writing proofs in Coq, it is sometimes desirable to delay choosing some term until a later point where the updated proof goal makes it more obvious what the right choice for that term is. For example, if we have the proof goal $a : \mathbb{Z},\ b : \mathbb{Z},\ c : \mathbb{Z},\ H_1 : a < b,\ H_2 : c > b \vdash a < c$ and want to apply the lemma `Z.lt_trans`, which says $\forall\ n\ m\ p.\ n < m \Rightarrow m < p \Rightarrow n < p$, Coq can infer (by unifying the lemma's conclusion with the goal's conclusion) that $n$ has to be instantiated to $a$ and $p$ to $c$, but it is not immediately clear what term $m$ should be instantiated with. So either the user has to provide it explicitly by running the tactic `apply Z.lt_trans with (m := b)`, which results in two subgoals with the same hypotheses as the original goal and conclusions $a < b$ and $b < c$ respectively; or the user can delay the choice of $m$ by running `eapply Z.lt_trans`. The `eapply` tactic is a variant of the `apply` tactic that creates so-called **evars** (short for existential variables) for terms that cannot be determined yet. On this example, it results in two subgoals with conclusions $a < ?m$ and $?m < c$, respectively, where the question mark is used to mark $m$ as an evar, i.e. as **some hole that will be filled in later**. Note that the two occurrences of $?m$ in the two subgoals are linked: As soon as $?m$ is instantiated to some term in one goal, it is also instantiated to the same term in the other goal. To continue the example, one could now run the `eassumption` tactic on the first goal, which applies any assumption from the list of hypotheses that matches the conclusion. The `e` at the beginning of the tactic's name means that it can instantiate evars in order to unify a hypothesis with the conclusion, so it will pick $H_1$ and instantiate $?m$ to $b$.

### 5.2.6 A Use Case of Evars: Deriving a Definition Based on its Proof

Coq's **`Derive`** command can be used to create a definition and a proof about it at the same time. For example, if we want to define a list `myList` such that it contains (at least) `1` and `2` as its elements, we can start with the the command

```
Derive myList SuchThat (In 1 myList ∧ In 2 myList) As my_property.
```

It starts the definition of a list named `myList`, along with a lemma named `my_property`. Note that the definition of `myList` is not yet given at this point and will only be filled in gradually while writing the proof. This command creates an evar `?myList` for the definition being made and opens the proof goal `⊢ In 1 ?myList ∧ In 2 ?myList`. Using the `split` tactic turns it into two goals, `⊢ In 1 ?myList` and `⊢ In 2 ?myList`. Given the lemma `in_eq` which says

```
∀ (A : Type) (a : A) (l : list A), In a (cons a l)
```

we can run `eapply in_eq` on the first subgoal, which unifies the conclusion of that lemma with `In 1 ?myList`. This step *partially instantiates* the evar `?myList`, namely to the term `(cons 1 ?l)`, which in turn contains a new evar `?l`, so the second subgoal now becomes `⊢ In 2 (cons 1 ?l)`. Then, the proof can be completed by applying `in_cons`, which says that

```
∀ (A : Type) (a b : A) (l : list A), In b l ⇒ In b (cons a l)
```

and leads to `⊢ In 2 ?l` and then applying `in_singleton`:

```
∀ (A : Type) (x : A), In x (cons x nil)
```

which instantiates the remaining evar `?l` to the singleton list containing just `2`.

So, through this series of proof steps, the list `myList` was defined to be `(cons 1 (cons 2 nil))` solely based on its proof, without ever having to spell out this term as a whole.

## 5.3 Overview: Writing and Compiling a Sample Program

### 5.3.1 Guided Tour Through the `memset` Example

This subsection gives an overview of our approach by means of a detailed discussion of the sample program in Figure 5.1. The sample program contains many notations, and in this section, we are not yet attempting to explain what exactly each notation unfolds to. Instead, we are just trying to give an intuitive understanding of their meanings. For reference, a listing of all notations can be found in section 5.9.

#### 5.3.1.1 Polyglot Source File Can be Read as C or Coq at the Same Time [Lines 1-27]

We use Emacs' `hideshow` minor mode to fold 5 lines of proof script into ···. The code in Figure 5.1 is a Coq file accepted by unmodified Coq 8.17.1 without requiring any plugins[3]. By (ab)using Coq's notation system, we can insert program snippets that look like C code. If the file is preceded by our framework-specific C header and an opening C comment `/*`, it becomes a C file that can be compiled with GCC.

[3]except some standard plugins that are distributed together with Coq, such as Ltac, Ltac2, and Lia

#### 5.3.1.2 Function Signature Using Only One Type [Line 6]

Line 6 contains the function signature in C syntax. Since we only support the subset of C that is also supported by Bedrock2, all variables have the same type, namely `uintptr_t` (defined in `stdint.h`). According to the standard, that is an unsigned integer type large enough to hold a pointer value, but we rely on the observation that in practice, compiler implementations define it as 32-bit and 64-bit unsigned int on 32-bit and 64-bit machines, respectively.

#### 5.3.1.3 Specifications Using Separation Logic and $\mathbb{Z}$ [Lines 7-13]

The C signature is followed by a function specification enclosed in a `/**# #**/` comment that lists ghost arguments, a precondition over the initial event trace `t` and the initial memory `m`, and a postcondition over the final event trace `t'` and final memory `m'`. The parts between `<{ }>` are separation-logic assertions. We use `*` symbols as bullet points for lists of separation-logic clauses to be joined by separating conjunction, so `*` can also be read as the traditional star operator from separation logic, just with the additional liberty of allowing a series of separating conjunctions to start with a superfluous initial `*`. The `array` predicate takes as arguments the predicate for its elements `(uint 8)`, followed by its number of elements, its list of elements, and its start address.

To make our specifications as trustworthy as possible, we need to avoid accidental integer overflows in the specifications, so we generally use unbounded integers ($\mathbb{Z}$) in our specifications rather than bounded integers (`word`), except in situations with many bitwise operations and where integer overflow is the desired outcome. Therefore, we often need to interpret bounded integers (values that were computed by our programs) as unbounded integers in order to mention them in specifications. To interpret a `word` value `x` as an unsigned $\mathbb{Z}$, we use the notation `\[x]` (which expands to the `word.unsigned` function), and there is also a `word.signed` function (for which we have not yet invented a notation because we use it less frequently). The reverse direction, coercing a $\mathbb{Z}$ into a `word`, does not need to distinguish between signed and unsigned integers, because in both cases, it simply takes the 32 least significant bits of the unbounded integer's binary representation in two's complement (and a negative number is considered to start with an infinite series of 1s on the left). We call this coercion `word.of_Z` and abbreviate it with `/[x]`, but since it drops the more significant bits, we try to use it as little as possible.

#### 5.3.1.4 The Initial Proof Goal [Line 14]

We use Coq's **`Derive`**[4] command (section 5.2.6) to start the correctness proof of a function that has not yet been defined but will be defined at the same time as we write the proof. Note that lines 15 to 26 are actually a proof script, even though they look like C code. The **`Derive`** command opens a proof goal which could be summarized, using the notation from section 5.2.4, as $\vdash P(t, s, m) \Rightarrow$ `wp` $(t, s, m)$ ?*body* $Q$, where $P$ stands for the precondition from lines 8-10, $Q$ stands for the postcondition from lines 11-13, and ?*body* is an evar (section 5.2.5) acting as a placeholder for the function body that is going to be defined. The state triple $(t, s, m)$ contains an event trace $t$, a partial mapping $s$ from variable names to values, and a memory $m$. The initial $s$ contains just the function arguments, so in this example, it equals `map.of_list [|("a", a); ("b", b); ("n", n)|]`.[5]

#### 5.3.1.5 C Snippets Acting As Proof-Script Steps [Lines 15-26]

Each C snippet is enclosed between a closing comment `.**/` and an opening comment `/**.` and is actually just a notation for a tactic. The first proof step, `.**/ { /**.`, introduces the precondition as hypotheses and performs some setup to start the proof. The datatype used to represent snippets is shown in Figure 5.2. Note that this datatype is only used by the tactics and does not appear in the function being defined: There, an AST with a conventional recursive structure is used.

#### 5.3.1.6 Applying Weakest-Precondition Rules [Lines 16-24]

The assignment on line 16 is a notation for a tactic that applies the `wp` rule for assignment shown on the in Figure 5.3. [6] It has a built-in sequence command, so applying it to a `wp` goal whose command is an evar instantiates that evar and leaves behind a new evar `?rest` for the subsequent commands.

The snippet on line 22 applies the WP-WHILE rule shown in Figure 5.3.[7] It requires an invariant *Inv*, a proof that *Inv* holds for the initial state; a proof that *Inv* implies that evaluating the condition $e$ is safe; a proof that if the condition is nonzero (true), running the loop body $c$ always leads to a

[4] A note for Haskell users: Unlike in Haskell, the **`Derive`** keyword in Coq is in no way related to the **`Instance`** keyword. The reason we mark specifications as type-class instances is explained in section 5.5.7.

[5] Note that for list literals, we use the notation `[|x; y; z|]` instead of Coq's standard notation `[x; y; z]`, because we want to use bracket notation to index into lists, so the term `f [b]` would become ambiguous: It could be the application of function `f` to the singleton list containing `b`, or it could be the `b`-th element of list `f`. We experimented with type-based operator overloading (section 5.5.8.2), but it did not seem worth the trouble.

[6] The rule that actually gets applied is specially tailored to work well with our proof automation, see section 5.5.2.

[7] For simplicity, we show a termination-insensitive variant, but the real lemma also requires a termination argument and is specially tailored for our proof automation, see section 5.5.2.

```
Inductive assignment_rhs :=
| RCall(fname: string)(args: list expr)
| RExpr(e: expr).

Inductive snippet :=
| SAssign(is_decl: bool)(x: string)(r: assignment_rhs) (* uintptr_t x = r;  |  x = r; *)
| SVoidCall(fname: string)(args: list expr)            (* fname(args);               *)
| SStore(sz: access_size)(addr val: expr)              (* store[|8|16|32](addr, val); *)
| SIf(cond: expr)(split: bool)                         (* if (cond) /* split */ {    *)
| SElse(startsWithClosingBrace: bool)                  (* } else {     |    else {   *)
| SWhile(c: expr){Measure: Type}(m: Measure)           (* while(c)/* decreases m */ { *)
| SStart                                               (* {                          *)
| SEnd                                                 (* }                          *)
| SRet(retexpr: expr)                                  (* return e;                  *)
| SEmpty.                                              (* /* for C comments */       *)

Notation "'while' ( e ) /* 'decreases' m */ {" :=
  (SWhile e m) (in custom snippet at level 0, e custom live_expr, m constr at level 0).

Notation "*/ s /*" := s (s custom snippet at level 100).

(* Standard usage:   .**/ snippet /**.    *)
Tactic Notation ".*" constr(s) "*" := next_snippet s; run_steps_internal_hook.
(* Debug mode (doesn't run verification steps):   .**/ snippet /*?.   *)
Tactic Notation ".*" constr(s) "?" := next_snippet s.
```

Figure 5.2: Datatype to represent C snippets and some of the notations for parsing them

WP-SET

$$\frac{\mathsf{eval\_expr}\ s\ m\ e\ v \qquad \mathsf{wp}\ (t, s[x := v], m)\ rest\ P}{\mathsf{wp}\ (t, s, m)\ (x := e; rest)\ P}$$

WP-WHILE

$$\frac{Inv\ \sigma \qquad \forall \sigma'.\ Inv\ \sigma' \Rightarrow \exists b.\ \mathsf{eval\_expr}\ \sigma'\ e\ b \wedge (b \neq 0 \Rightarrow \mathsf{wp}\ \sigma'\ c\ Inv) \wedge (b = 0 \Rightarrow \mathsf{wp}\ \sigma'\ rest\ P)}{\mathsf{wp}\ \sigma\ (\textbf{while}\ e\ \textbf{do}\ c; rest)\ P}$$

Figure 5.3: Some weakest-precondition rules

state that satisfies *Inv* again; and a proof that if the condition is zero (false), the code after the loop is correct.

Our framework contains rules for all language constructs, and they are all proven sound with respect to the semantics of Bedrock2 (expressed in omnisemantics [Charguéraud et al., 2023]).

#### 5.3.1.7 Expressing the Loop Invariant as a Diff from the Current Symbolic State [Lines 17-21 in Figure 5.4c]

The WP-WHILE lemma requires a loop invariant. Automatically inferring loop invariants is a hard problem, and we do not attempt to solve it. But spelling out loop invariants manually is also quite cumbersome. Therefore, we use an approach in-between these two extremes, based on the observation that the loop invariant often looks quite similar to the symbolic state just before the loop. Instead of requiring that the user spells out the *whole invariant*, we only require that the user expresses the *insight* needed to obtain the right invariant, expressed as a tactic script (Figure 5.4c) that transforms the symbolic state before the loop (Figure 5.4a) into a generalized and/or strengthened symbolic state (Figure 5.4b) which our framework then mechanically packages into a loop invariant (Figure 5.4d).

#### 5.3.1.8 Heapletwise Separation Logic [Background for Line 23]

It is useful to name each separation-logic clause and to make it available to Ltac's **`match`** command, which finds hypotheses matching a given pattern. Therefore, instead of using one big separation-logic clause `(P * Q * R) m`, we split it into one hypothesis per clause. This strategy requires explicitly splitting the memory `m` into a heaplet corresponding to each clause, which takes up some space in the display of the proof goal, but it can be handled completely automatically and therefore does not affect the user experience too negatively. This splitting then leads to three new heaplets `m0`, `m1`, `m2`; an equation saying that their disjoint union equals `m`, written as `m0 \*/ m1 \*/ m2 = m`; and three hypotheses `P m0`, `Q m1` and `R m2`. To make them more easily recognizable as memory hypotheses, we use the `mi |= Pi` notation, which just expands to `Pi mi`. See for example hypotheses `H0`, `H1`, and `D` in Figure 5.4a, and compare them to the precondition of the `memset` function on line 8 in Figure 5.1.

#### 5.3.1.9 Accessing Memory That Is Part of a Bigger Separation-Logic Clause [Line 23]

`store8(a + i, b)` stores the lowest 8 bits of `b` to the `i`-th element of the array at `a`. According to the loop invariant, we have the following separation-logic clause:

```
state : currently displaying
... 6 lines of section vars omitted ...
fs : list (string * func)
fs_ok : functions_correct fs ?Goal
Scope0 : ____ FunctionParams ____
a, b, n : word
bs : list Z
R : mem → Prop
Scope1 : ____ FunctionBody ____
t : trace
i : word
m, m0, m1 : mem
H0 : m0 |= array (uint 8) \[n] bs a
H1 : m1 |= R
D : m0 \*/ m1 = m
Hp1 : \[b] < 2 ^ 8
Def0 : i = /[0]
============================
ready
```

(a) Symbolic state (proof goal) after processing the first line of the function body in Figure 5.1

```
state : currently displaying
... 6 lines of section vars omitted ...
fs : list (string * func)
fs_ok : functions_correct fs ?Goal
Scope0 : ____ FunctionParams ____
a, b, n : word
bs : list Z
R : mem → Prop
Scope1 : ____ FunctionBody ____
t : trace
Scope2 : ____ LoopInvOrPreOrPost ____
i : word
m, m0, m1 : mem
H0 : m0 |= array (uint 8) \[n]
     (List.repeatz \[b] \[i] ++ bs[\[i]:]) a
H1 : m1 |= R
D : m0 \*/ m1 = m
Hp1 : \[b] < 2 ^ 8
============================
ready
```

(b) Symbolic state (proof goal) after processing the Ltac code in (c)

```
17 swap bs with
18   (List.repeatz \[b] \[i] ++ bs[\[i]:])
19   in #(array (uint 8)).
20 loop invariant above i.
21 delete #(i = ??).
```

(c) Snippet of Ltac code that was folded into ··· in Figure 5.1. The # notation is used to reference a hypothesis matching a pattern, instead of using its autogenerated (and thus subject-to-change) name.

```
fun (measure : word) (ti : trace)
    (mi : mem) (li : locals) ⇒
  exists i : word, ands [|
    measure = n ^- i; ti = t;
    li = map.of_list [|("a", a); ("b", b);
                       ("i", i); ("n", n)|];
    seps [|array (uint 8) \[n]
             (List.repeatz \[b] \[i]
              ++ bs[\[i]:]) a; R|] mi;
    \[b] < 2 ^ 8|]
```

(d) Loop invariant automatically built by packaging everything below `__LoopInvOrPreOrPost__` in (b)

Figure 5.4: Loop-invariant definition using a diff script (c) instead of explicitly spelling it out

```
H0 : m0 |= array (uint 8) \[n] (List.repeatz \[b] \[i] ++ bs[\[i]:]) a
```

However, the wp lemma for the store commands (omitted for space reasons) expects a separation-logic clause with just one `(uint 8)` element, so we need to split the array appropriately. Our tactics take care of this automatically, leading to the following three clauses:

```
H2 : m0 |= array (uint 8) \[i] (List.repeatz \[b] \[i]) a
H3 : m2 |= uint 8 bs[\[i]] (a ^+ i)
H7 : m4 |= array (uint 8) (\[n] - \[i] - 1) bs[\[i] + 1 :] (a ^+ i ^+ /[1])
```

The store then replaces `bs[\[i]]` with `b` in `H3`, and since the splitting tactic posed a hypothesis that acts as a reminder to merge the three clauses back together later, we end up with the following clause after the store:

```
H1 : m |= array (uint 8) \[n] (List.repeatz \[b] \[i] ++ [|\[b]|] ++ bs[\[i] + 1 :]) a
```

#### 5.3.1.10 Proving That the Current Symbolic State Satisfies Expectations [Lines 25 and 26]

The closing brace at the end of the loop body creates a proof that the symbolic state obtained by executing the loop body satisfies the loop invariant again, and the closing brace at the end of the function body creates a proof that the final symbolic state satisfies the postcondition given on lines 11-13. In this example, the proofs are found completely automatically, but in more complex examples, the automation might leave some goals open for the user to prove manually. This completes our tour of the `memset` example.

### 5.3.2 Tradeoffs Between Three Different Ways of Compiling

Finally, after proving our `memset` function correct, we might also want to compile and run our code. Table 5.1 compares three different ways of compiling code that was verified in our framework. The C-parsing notations of our framework expand to Bedrock2 ASTs, defined as a Coq inductive datatype, so the correctness proofs are statements about these ASTs. The verified Bedrock2 compiler consumes the same ASTs and is proven correct against the same semantics as used by our framework, so when it comes to minimizing the TCB, this is the preferred approach. For better performance and support of ISAs other than just RISC-V, one can choose to compromise on TCB minimality: If one trusts our notations to parse C as well as Coq's implementation of its notation system, one can feed our Coq files (which are also C files if preceded by our header defining loads and stores and an opening comment `/*`) to GCC (and likely also to other C compilers), or if one

| | Feed Coq file to GCC | Bedrock2's ugly-printer & GCC | Bedrock2 Compiler |
|---|---|---|---|
| Readability of exported code | OK (see e.g. Figure 5.1) | Decipherable (many casts & parentheses) | It is assembly |
| Instruction-set architecture support | Everything supported by GCC | Everything supported by GCC | Only RISC-V |
| Performance of compiled code | Good | Good | Bad |
| Additions to trusted code base | Notations to parse C into Bedrock2, GCC, load/store C header | Bedrock2's ugly-printer, GCC, load/store C header | Only the riscv-coq specification |

Table 5.1: Different Ways of Compiling

prefers to trust Bedrock2's pretty-printer (called ugly-printer by its author), one gets less readable C code but otherwise similar characteristics.

One might also wonder whether it would make sense to compile our programs with CompCert. In practice, this would probably work, but we do not have a compatibility proof between Bedrock2 semantics and CompCert C semantics, and such a statement would not be provable because of differences such as e.g. that comparisons between integers that were obtained by casting pointers are undefined behavior in CompCert C.

## 5.4 User Interface

### 5.4.1 New Separation-Logic Concepts

To better drive separation-logic proof automation and make some expressions more concise, we introduce a few properties of separation-logic predicates.

#### 5.4.1.1 Predicate Size

Often a separation-logic predicate `P` occupies some range of memory addresses, and we need to know the length in bytes of that range. Therefore, we define `PredicateSize P` to be an alias of $\mathbb{Z}$, mark it as a type class, and register a hint for each predicate, so that we can use type-class search to find the size of a predicate. The predicate `(array elemPred n xs a)` can then use an implicit, automatically inferred argument `elemSize` of type `(PredicateSize elemPred)`, to state

that at address `a`, we have an array of the `n` elements in list `xs`, where the `i`-th element of `xs` is asserted using `(elemPred xs[i] (a+i*elemSize))`.

#### 5.4.1.2 Support for Adjacent Sep Clauses: `sepapp` and `sepapps`

Often, we want to lay out several predicates adjacent to each other.[8] To avoid having to write out offsets explicitly, we introduce the notion of *separating append*, written `sepapp P1 P2 addr`. It takes two separation-logic predicates `P1` and `P2` of type `word → mem → Prop`, where the `word` stands for the address at which the predicate begins, and also takes an implicitly inserted argument `P1size` of type `PredicateSize P1` (which can be found by type-class search as explained above) and an address `addr`, and it is defined as the separating conjunction `P1 addr * P2 (addr ^+ /[P1size])`. We also define a `sepapps` predicate that takes a list of predicates, infers their sizes, and lays them out adjacently.

#### 5.4.1.3 *n*-Fillable Predicates

We call a predicate $P$ $n$-fillable if for any $n$-byte buffer at address $a$, there exists a value $v$ such that the predicate $P\ v\ a$ holds. This concept is useful to know whether we can cast the byte buffer returned by `malloc` into a predicate $P$.

### 5.4.2 Defining Record Predicates Using C Syntax

Our framework supports defining separation-logic predicates using C syntax. For example, given a Coq record type for nodes of singly linked lists, `Record node := { data: word; next: word }`, we can create a separation-logic predicate called `node_t` that asserts that at a given address, a representation of a given `node` record is found. Using `sepapps` and some custom notations, we can define a predicate that looks like a C struct definition (first definition in Figure 5.5). The two other definitions in that figure express the same predicate but using a notation for `sepapps` or `sepapps` directly, respectively.

### 5.4.3 IDE Extensions

Our framework can be used in any IDE for Coq. However, there are three very common operations for which we implemented keyboard shortcuts in 40 lines of Emacs Lisp:

[8] So far, we have only considered packed records, so we do not automatically insert spacing to respect alignment constraints.

```
Definition node_t(r: node):
  word → mem → Prop := .**/
typedef struct
__attribute__ ((__packed__)) {
  uintptr_t data;
  uintptr_t next;
} node_t;
/**.
```

```
Definition node_t(r: node): word → mem → Prop :=
  <{ + uintptr (data r)
     + uintptr (next r) }>.
Definition node_t(r: node): word → mem → Prop :=
  sepapps
    (cons (mk_sized_predicate (uintptr (data r)) 4)
    (cons (mk_sized_predicate (uintptr (next r)) 4)
    nil)).
```

Figure 5.5: Three equivalent definitions, using different notations

- Showing/hiding of the Ltac block under the cursor (i.e. folding tactics into ···)
- Inserting spaces until the end of line followed by a C-closing/Ltac-opening marker `/**.` and then processing the line
- Inserting and processing one `step` command (section 5.4.8.3)

### 5.4.4 Expressing a Loop Invariant as a Diff from the Current Symbolic State

Before each loop, the user of our framework must turn the current symbolic state into a shape that our framework can use as a loop invariant. The example in Figure 5.4 should be helpful to illustrate the general process that we are going to explain in detail now. All modifications are expressed in Ltac and are of two kinds:

The first is that the user needs to separate variables and hypotheses that remain unchanged during the loop from those that may change during the loop, by using the command `loop invariant above x`, where `x` is the name of a variable or hypothesis. This command adds a `LoopInvOrPreOrPost` marker above `x` to separate unchanged (above) from changing (below) variables and hypotheses. After adding this marker, one can use Coq's builtin Ltac commands `move x before y` and `move x after y` to move hypotheses and variables up and down, until each is on the correct side of the separating marker. The variables below the marker will turn into existentials in the loop invariant, and the hypotheses will turn into a big conjunction (expressed as `ands [|...|]`). The variables and hypotheses above the marker do not appear in the loop invariant, except that the local variables above the marker are asserted to keep their values throughout the loop, and the hypotheses naturally remain available during the proof of the loop body without requiring further intervention.

The second kind of modification is related to generalizing the state. For instance, a variable `i` that equals one particular value before the loop might need to be generalized to be within a range by `prove (0 <= i < n)`; and by `delete #(i = ??)`, which finds the first hypothesis of shape `i = ??`

and deletes it. Other common modifications of this kind include viewing a list of unprocessed items as the concatenation of an empty processed list and a remainder of unprocessed items, then forgetting that the unprocessed and processed list are the empty and whole list, respectively. A similar example is also in Figure 5.4c, where we replace the list `bs` of initial garbage data by the concatenation of repeating `\[b]` zero times (zero being the initial value of `i`) and the suffix of `bs` starting at `i`. And finally, it is sometimes also needed to introduce additional variables, so that a value that happens to be the same in two hypotheses can differ during the loop, which can be achieved using the `pose (a := b)` command, and `change b with a in H`, and finally, `clearbody a` to forget that `a` equals `b`.

### 5.4.5 Treating While Loops as Tail-Recursive Calls

Certain loops can be verified more easily by viewing them as tail-recursive functions with pre- and postconditions parameterized over ghost variables [Tuerk, 2010]. Before each loop iteration, the precondition must hold, and at the end of the loop body, one has to show that the current state implies the precondition with smaller ghost variables, and one also has to show that the postcondition with small ghost variables implies the postcondition with bigger ghost variables.

For instance, when iterating over a data structure, the ghost variables can include a representation of the data structure and a frame, and the former shrinks with each iteration, while the latter grows with each iteration, so that we can forget the parts of the memory that are not relevant anymore.

We implement support for while and do-while loops in this style, using the symbolic state before the loop, appropriately generalized and strengthened through a diff script by the user, as a precondition, and the function's postcondition as the postcondition of the tail-recursive view of the loop. Since we do not want users to spell out loop postconditions manually, we do not support yet this tail-recursive view for cases where the code after the loop still needs to access the memory that was "forgotten" (pushed into the frame) during the loop. In such cases, one would have to factor the code into two functions or resort to a traditional while loop with just one invariant.

As an example, proving correctness of a `strcmp` function with a traditional invariant-based loop would require an invariant like the following:

```
H2 : m2 |= array (uint 8) (len s1 + 1) (s1 ++ [|0|]) p1_pre
H1 : m1 |= array (uint 8) (len s2 + 1) (s2 ++ [|0|]) p2_pre
H3 : m3 |= R
H  : \[p1 ^- p1_pre] <= len s1
```

```
H0 : \[p2 ^- p2_pre] <= len s2
H6 : \[p1 ^- p1_pre] = \[p2 ^- p2_pre]
H7 : s1[:\[p1 ^- p1_pre]] = s2[:\[p2 ^- p2_pre]]
```

...where `p1` and `p2` are pointers pointing somewhere into the middle of the two strings `s1` and `s2` being compared, and `p1_pre` and `p2_pre` are the original values of `p1` and `p2` pointing to the beginnings of the strings. Each loop iteration compares the two characters at `p1` and `p2` and exits the loop if they are different.

On the other hand, if we view the same loop as if it were a tail-recursive function, its precondition can become much simpler:

```
H2 : m2 |= array (uint 8) (len s1 + 1) (s1 ++ [|0|]) p1
H5 : ~ List.In 0 s1
H1 : m1 |= array (uint 8) (len s2 + 1) (s2 ++ [|0|]) p2
H4 : ~ List.In 0 s2
H3 : m3 |= R
```

Each iteration compares the first character of `s1` and `s2`, and if a next iteration is needed, we forget the two compared characters by moving them from `s1` and `s2` into the frame `R`. As the postcondition, we can reuse the function's postcondition, generalizing it over the ghost variables `s1, s2, p1, p2, R`. Figure 5.6 shows the `strcmp` function proven in this style, with 32 lines of uninteresting proof folded into ···. Note that the user needs to provide the initial values of all ghost variables and somewhere in the omitted proof script also needs to specify the new values of the ghost variables for the tail-recursive call (i.e. the next iteration).

### 5.4.6 Variable-Naming Scheme

Our tactics make sure that a program variable named `"x"` always has its corresponding value bound to a Coq variable named `x`. When a variable gets reassigned, the old value is renamed into `x'`, and `x` is used for the new value. We use primes instead of Coq's standard, number-suffix-based fresh-name generation scheme because if we have program variables called `x1` and `x2` and ask Coq to generate a fresh name for `x1`, it will replace the suffix `1` by the lowest number that results in a fresh name, so it would become hard to tell whether a generated `x0` is a previous version of `x1` or of `x2`. Primes have the advantage that they cannot be used in C variable names, so by removing primes from the end, users can unambiguously infer what variable an old value comes from.

```
#[export] Instance strcmp_spec: fnspec :=                                    .**/

uintptr_t strcmp(uintptr_t p1, uintptr_t p2) /**#
  ghost_args := (s1 s2: list Z) (R: mem → Prop);
  requires t m := <{ * nt_str s1 p1
                     * nt_str s2 p2
                     * R }> m;
  ensures t' m' res := t' = t ∧
                       List.compare Z.compare s1 s2 = Z.compare (word.signed res) 0 ∧
                       <{ * nt_str s1 p1
                          * nt_str s2 p2
                          * R }> m' #**/                                  /**.
Derive strcmp SuchThat (fun_correct! strcmp) As strcmp_ok.                     .**/
{                                                                          /**. .**/
  uintptr_t c1 = 0;                                                        /**. .**/
  uintptr_t c2 = 0;                                                        /**.….**/
  do /* initial_ghosts(s1, s2, p1_pre, p2_pre, R); decreases (len s1) */ { /**. .**/
    c1 = deref8(p1);                                                       /**. .**/
    c2 = deref8(p2);                                                       /**. .**/
    p1 = p1 + 1;                                                           /**. .**/
    p2 = p2 + 1;                                                           /**. .**/
  } while (c1 == c2 && c1 != 0);                                           /**.….**/
  return c1 - c2;                                                          /**. .**/
}                                                                          /**.….**/ //.
```

Figure 5.6: Viewing a do-while loop as a tail-recursive function to simplify the correctness proof. Note that a total of 32 lines of proof has been folded into ….

WP-IF

$$\frac{\begin{array}{c}\mathsf{eval\_expr\_as\_bool}\ \sigma\ e\ b \\ (b = \mathbf{true} \implies \mathsf{wp}\ \sigma\ thn\ Q_1) \\ (b = \mathbf{false} \implies \mathsf{wp}\ \sigma\ els\ Q_2) \\ \forall \sigma'.\ (\mathbf{if}\ b\ \mathbf{then}\ Q_1\ \sigma'\ \mathbf{else}\ Q_2\ \sigma') \implies \mathsf{wp}\ \sigma'\ rest\ P\end{array}}{\mathsf{wp}\ \sigma\ (\mathbf{if}(e)\{thn\}\mathbf{else}\{els\};rest)\ P}$$

(a) Weakest-precondition lemma for if-then-else, presented as a simplified inference rule. Note that the **if** below the line belongs to the object language (Bedrock2), whereas the **if** above the line belongs to the metalanguage (Coq).

```
Definition after_if fs (b: bool) (Q1 Q2: trace → mem → locals → Prop) rest post :=
  ∀ t m l, (let c := b in if c then Q1 t m l else Q2 t m l) → wp_cmd fs rest t m l post.

Lemma wp_if_bool_dexpr fs c thn els rest t0 m0 l0 b Q1 Q2 post:
  dexpr_bool3 m0 l0 c b
    (then_branch_marker (wp_cmd fs thn t0 m0 l0 (package_context_marker Q1)))
    (else_branch_marker (wp_cmd fs els t0 m0 l0 (package_context_marker Q2)))
    (pop_scope_marker (after_if fs b Q1 Q2 rest post)) →
  wp_cmd fs (cmd.seq (cmd.cond c thn els) rest) t0 m0 l0 post.
```

(b) Tailored weakest-precondition lemma for if-then-else. The three definitions ending in `_marker` all are identity functions, but help the tactics keep track of where we are and what to do.

Figure 5.7: Weakest-precondition lemma for if-then-else

### 5.4.7 Context Packaging and Merging for if-then-else

The lemma that we use for if-then-else is given in Figure 5.7. Its essence is summarized as an inference rule in Figure 5.7a, and the lemma that we actually use is given in Figure 5.7b.

When WP-IF gets applied, evars are created for the result $b$ of evaluating the condition $e$, for the code snippets *thn*, *els*, and *rest*, as well as for the postconditions of the two branches, $Q_1$ and $Q_2$. The tactics first evaluate the condition $e$ into a Boolean $b$. Then, the user can provide more snippets that make up the code of the then-branch. When providing the snippet `.**/ } else { /**.`, the then-branch is closed, and the evar $?Q_1$ is instantiated by our tactics to a conjunction of all the hypotheses in the current context. When the user closes the else-branch, $?Q_2$ is instantiated in the same way, and before the first command after the if-then-else is processed, the two symbolic states (expressed by $Q_1$ and $Q_2$) are merged by pushing down the metalanguage **if** as far as possible by detecting parts in $Q_1$ and $Q_2$ that have the same structure. The tactics bind the value of $b$ to a fresh variable, so that we can mention it many times without becoming overly verbose. This merging

results in symbolic states containing hypotheses with many if-then-else expressions like e.g. in the following:

```
H1 : m0 |= uint 32 (if c' then in1 else if c then in2 else in0) a0
Def7 : w1 = (if c' then /[in0] else /[in1])
```

A `/* split */` option is available that can be inserted after the **`if`** condition if one prefers to continue the proof separately after the if-then-else rather than using a merged state. However, this option is only available if the if-then-else is at the end of a block with a concrete postcondition (i.e. a loop invariant or the function's postcondition), because splitting the proof of the code after the if-then-else into two separate proofs would require writing down all the code snippets (which drive the proof) twice, which would not result in the desired C code when treating the Coq file as a C file.

### 5.4.8 Optimize the User Experience for Failing Proofs Instead of Working Proofs

In the past, most frameworks for automated program proofs have focused on presenting automated proofs that work. However, we must recognize that the default case that users of program verification tools face is the case where the prover fails or seems to run forever, for one of the following reasons:

- The program or the specification contains a bug.
- A user-provided invariant is not strong enough.
- The prover lacks some domain-specific insight or hint that needs to be provided by the user.

We believe that debugging these situations, and being able to determine quickly which of the above is the case, is the primary usability criterion for a program-verification tool, much more important than the number of lines of proof script that users need to provide manually.

Therefore, we adhere to three principles described in the following subsections.

#### 5.4.8.1 Do Not Run "Forever" on Failing Proofs

We carefully designed our proof automation in such a way that it never runs for longer than a few seconds, and if it does, we consider it a bug. This is important because in order to figure out why a prover cannot prove a certain goal, one needs to try many different variations of the goal and see which ones the prover can and cannot prove. Now, if the prover runs "forever" (i.e., longer than it makes sense to wait) on failing goals (which, e.g., happened often in our experience with Dafny on programs that use the heap extensively), the user never quite knows whether waiting for a few

more seconds would have solved the goal, and the user's mental model of what the tool can and cannot solve becomes inconsistent.

#### 5.4.8.2 Actionable Error Messages

If the tool fails to prove a goal, it should provide the user with an error message containing information on what it tried and what (currently unprovable) conditions might enable it to make more progress.

As an example, let us look more closely at separation-logic cancellation, which is required e.g. before function calls, to match the caller's symbolic heap to the callee's symbolic heap. The strategy is repeatedly to delete separation clauses that appear both in the caller's heap and in the callee's heap. Since the clauses in the callee's heap typically contain evars for the callee's ghost arguments (because ghost arguments do not get mentioned in the source code), our procedure carefully only instantiates an evar if there is a unique choice. Sometimes, e.g. when a record field or a slice of an array is passed to the callee, cancellation needs to split a caller's clause before it can proceed. So, if the callee expects a chunk of memory covering the range starting at address $a$ of size $n$, we need to find a clause in the caller's heap covering a superrange of that range, starting at an address $a'$ and of a size $n'$ such that the subset relation on half-open intervals $[a, a + n[ \subseteq [a', a' + n'[$ holds, written in Coq as `subrange a n a' n'`.

Say we want to implement and verify a function with the following signature:

```
uintptr_t safeCopySlice(uintptr_t src, uintptr_t srcOfs, uintptr_t srcLen,
    uintptr_t unsafeN, uintptr_t dst, uintptr_t dstOfs, uintptr_t dstLen)
```

Its full specification spans 25 lines of code and is given in Figure 5.8 but is more easily expressed in English: Given a byte array of length `srcLen` at address `src` and a byte array of length `dstLen` at address `dst`, we want to copy `unsafeN` bytes starting at offset `srcOfs` in the source array to offset `dstOfs` in the destination array. We already know that `srcOfs` and `dstOfs` are within bounds, but `unsafeN` comes from an untrusted origin and needs to be checked. We might start with the following:

```
Derive safeCopySlice SuchThat (fun_correct! safeCopySlice) As safeCopySlice_ok.    .**/
{                                                                                  /**. .**/
  if (srcOfs + unsafeN <= srcLen && dstOfs + unsafeN <= dstLen) /*split*/ {        /**. .**/
    Memcpy(dst + dstOfs, src + srcOfs, unsafeN);                                   /**.
```

After processing the proof just until before the `Memcpy` call, our symbolic state contains 16 uninteresting lines listing variables that we elide, followed by what is shown in Figure 5.9.

```
#[export] Instance spec_of_safeCopySlice: fnspec :=                                .**/

uintptr_t safeCopySlice(
  uintptr_t src, uintptr_t srcOfs, uintptr_t srcLen,
  uintptr_t unsafeN,
  uintptr_t dst, uintptr_t dstOfs, uintptr_t dstLen
) /**#
  ghost_args := srcData dstData dstUninit (R: mem → Prop);
  requires t m :=
    <{ * array (uint 8) \[srcLen] srcData src
       * array (uint 8) \[dstLen] (dstData ++ dstUninit) dst
       * R }> m ∧
    \[srcOfs] <= \[srcLen] ∧
    len dstData = \[dstOfs];
  ensures t' m' r := t' = t ∧
   ((r = /[0] ∧
     <{ * array (uint 8) \[srcLen] srcData src
        * array (uint 8) \[dstLen] (dstData ++ dstUninit) dst
        * R }> m') ∨
    (r = /[1] ∧
     <{ * array (uint 8) \[srcLen] srcData src
        * array (uint 8) \[dstLen] (dstData ++
                                    srcData[\[srcOfs]:][:\[unsafeN]] ++
                                    dstUninit[\[unsafeN]:]) dst
        * R }> m')) #**/                                                    /**.
Derive safeCopySlice SuchThat (fun_correct! safeCopySlice) As safeCopySlice_ok.    .**/
{                                                                           /**. .**/
  if (unsafeN <= srcLen - srcOfs && unsafeN <= dstLen - dstOfs) /*split*/ {  /**. .**/
    Memcpy(dst + dstOfs, src + srcOfs, unsafeN);                            /**. .**/
    return 1;                                                               /**. .**/
  }                                                                         /**. .**/
  else {                                                                    /**. .**/
    return 0;                                                               /**. .**/
  }                                                                         /**. .**/
}                                                                           /**.
Qed.
```

Figure 5.8: The specification as well as the final, correct implementation of `safeCopySlice`. Note that all the proof steps are completely automated, including the splitting of the source and destination arrays before the `Memcpy` call, pasting them back together after the call, and matching that result to the desired postcondition.

```
H0 : m0 |= array (uint 8) \[srcLen] srcData src
H2 : m2 |= array (uint 8) \[dstLen] (dstData ++ dstUninit) dst
H3 : m3 |= R
D : m0 \*/ (m2 \*/ m3) = m
Hp1 : \[srcOfs] <= \[srcLen]
Hp2 : len dstData = \[dstOfs]
Scope2 : ____ IfCondition ____
H : \[srcOfs ^+ unsafeN] <= \[srcLen]
H1 : \[dstOfs ^+ unsafeN] <= \[dstLen]
Scope3 : ____ ThenBranch ____
============================
ready
```

Figure 5.9: Proof goal before `Memcpy`

At this point, the user could start wondering whether the word addition in hypothesis `H`, i.e. `\[srcOfs ^+ unsafeN]`, could also be expressed as an addition on $\mathbb{Z}$, i.e. as `\[srcOfs] + \[unsafeN]`, and why the tool did not do that, even though it usually does, or the user can also just carry on and move the proof cursor past the `Memcpy` call. Doing so changes the conclusion of the goal to `(find_hyp_for_range (dst ^+ dstOfs) (\[unsafeN] * 1) X)`, where $X$ is a bigger goal that we elided for presentation purposes. `find_hyp_for_range` is a Gallina definition of an identity function that takes two phantom arguments that it ignores, plus a third one, $X$, that it returns. It serves as a marker to inform the tactics as well as the user that the tool is performing cancellation and looking for a separation-logic clause in the caller's symbolic heap whose range starts at `(dst ^+ dstOfs)` and spans `(\[unsafeN] * 1)` bytes. The tool also emits the following error message: `"Exactly one of the following claims should hold:" [|subrange (dst ^+ dstOfs) (\[unsafeN] * 1) src (\[srcLen] * 1); inrange src (dst ^+ dstOfs) (\[unsafeN] * 1); subrange (dst ^+ dstOfs) (\[unsafeN] * 1) dst (\[dstLen] * 1); inrange dst (dst ^+ dstOfs) (\[unsafeN] * 1)|] !` This message might look quite unintelligible at first, but we will show how it is *actionable* in the sense that it points the user to something to try to prove that is unprovable and will make the user understand the bug. The message contains four semicolon-separated claims and says that exactly one of them should hold. We can ignore the two `inrange` claims, because they are only needed to split separation-logic clauses on the callee side, which we do not expect to happen here, so we are left with just two `subrange` claims, and the second one looks like it *should* be provable (whereas the first one tries to relate completely unrelated ranges, one in `dst`, the other in `src`). Since we are in Coq's proof mode, right after the call to `Memcpy`, we can insert the following Ltac snippet to try to prove the `subrange` claim that we think should hold:

```
assert (subrange (dst ^+ dstOfs) (\[unsafeN] * 1) dst (\[dstLen] * 1)). {
  unfold subrange. bottom_up_simpl_in_goal.
```

It leads to a new goal with conclusion `\[dstOfs] + \[unsafeN] <= \[dstLen]`, but the most closely matching hypothesis is `H1` (see Figure 5.9), which performs the addition on `word` instead of on $\mathbb{Z}$. So now, the user might complain about how limited the proof automation of this Live Verification tool is and attempt to prove the goal manually, by invoking Coq's standard `Search` command with the pattern `\[_ ^+ _]` as its argument, whose top two results are a lemma which shows that for all words `x` and `y`, `\[x ^+ y] = word.wrap (\[x] + \[y])`; and one which shows that if `\[x] + \[y] < 2 ^ width`, then `\[x ^+ y] = \[x] + \[y]`. By this point, it should have become clear that the program contains an overflow bug: If `unsafeN` is very big, the addition overflows and results in a small number that might satisfy the condition tested by the **`if`** on the first line of `safeCopySlice`, but the `Memcpy` will still copy `unsafeN` bytes and overwrite out-of-bounds data, which could be exploited by attackers for arbitrary code execution. In fact, this (seemingly simple) type of bug (overflow on unsigned integer addition that computes the required amount of memory) led to the stagefright bugs, which in 2015 exposed the majority of Android users to no-click remote code execution on mere reception of a malicious MMS message.[9] Replacing the test by the following, overflow-safe variant resolves the problem:

```
  if (unsafeN <= srcLen - srcOfs && unsafeN <= dstLen - dstOfs) /*split*/ {   /**. .**/
```

For reference, the corrected full program (which does not do anything more in addition to what was already shown except returning 1 or 0 depending on whether `unsafeN` was accepted) is given in Figure 5.8. We note that all its proof steps are completely automated, including the splitting of the source and destination arrays before the `Memcpy` call, pasting them back together after the call, and matching that result to the desired postcondition.

#### 5.4.8.3 Safe Steps – Avoiding Backtracking for Better Proof Debuggability

To make our proof automation more debuggable, we avoid backtracking as much as possible and instead use mechanisms that allow us to know whether a proof step is *safe*, i.e. will not turn a provable goal into an unprovable one. We expose a tactic called `step` to the user, and when a proof does not work, the user can disable the automatic invocation of side-condition solving by replacing the `/**.` after a snippet by `/*?.` and then manually invoke `step` many times and watch step-by-step what the prover does and how it affects the proof goal.

[9] https://en.wikipedia.org/wiki/Stagefright_(bug), https://nvd.nist.gov/vuln/detail/CVE-2015-3864, https://www.exploit-db.com/docs/39527

To give an example of safe and unsafe steps, if we have a goal `?xs ++ ?ys = vs1 ++ vs2`, i.e. two evars on the left and normal variables on the right, it would be tempting to just instantiate `?xs` to `vs1` and `?ys` to `vs2`. However, this choice might make another goal in which the two evars also appear unsolvable, because the correct choice for `?xs` might be `vs1 ++ vs2[:1]`, and the correct choice for `?ys` would then be `vs2[1:]`. On the other hand, on a very similar-looking goal, `vs1 ++ ?ys = vs1 ++ vs2`, it is safe to instantiate `?ys` to `vs2`, because that is the only possible choice.

We use a user-extensible hint database of judgments of the form `safe_implication P Q`, which is defined as `P` implies `Q`. The opposite direction usually also holds, but in some cases, `Q` does not quite imply `P`, yet the only reasonable way to prove `Q` is to reduce it to proving `P`, so we do not require the opposite implication direction. For examples like the above, our hint database of safe steps contains the following two rules:

- `safe_implication (ys1 = ys2) (xs ++ ys1 = xs ++ ys2)`
- `safe_implication (xs1 = xs2) (xs1 ++ ys = xs2 ++ ys)`

As an example of how safe steps can help debug failing proofs, consider the last proof step of the insert function of a binary search tree, in the case where a new leaf had to be allocated for the value to be added. Assume that the programmer correctly initialized all fields of the new leaf but forgot to link the leaf to the parent node. The return value of the function is specified to be 0 if the memory allocator failed and 1 if it succeeded. Figure 5.10 shows the postcondition that needs to be proven in this case.

To debug why this postcondition cannot be proven automatically, the user can insert and process[10] many invocations of `step` and see how they try to solve the goal step-by-step. Each step also prints a short description of what it did. Here, we just summarize the most interesting steps. The full log of all steps is given in Appendix B. One of the first steps gets rid of the trivial equality `t = t`, and a subsequent step notices that since `\[/[1]] = 0` can never hold, it is *safe* to attempt proving only the right-hand side of the disjunction. Further steps then start cancelling the separation-logic formula with clauses from the hypotheses (not shown here) and manage to prove everything except a remaining goal `(is_empty_set (fun x : Z ⇒ x = \[vAdd] ∨ s x))`, which asks us to prove that a set, expressed as a lambda returning a proposition, is empty, even though it clearly contains at least one element, namely `\[vAdd]`. Here we see a case where reducing unprovability to the smallest possible core is actually too much, so that it is not easily understandable anymore why the tool asks us to prove this contradictory goal. But, fortunately, one of the intermediate

[10] We provide an Emacs macro to do so efficiently (see section 5.4.3), but simple copy-paste works as well.

```
t = t ∧
(\[/[1]] = 0 ∧
 <{ * allocator_failed_below 12
    * (EX rootp : word, <{ * uintptr rootp p
                           * (EX sk : tree_skeleton, bst' sk s rootp) }>)
    * R }> m4 ∨
 \[/[1]] = 1 ∧
 <{ * allocator
    * (EX rootp : word,
       <{ * uintptr rootp p
          * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>)
    * R }> m4)
```

Figure 5.10: Postcondition that needs to be proven in one case at the end of a (buggy) binary-search-tree insert, where `bst' sk s a` asserts that at address `a` is found a binary search tree whose tree structure is `sk` and whose contents correspond to the set `s`, represented as a proposition over values.

goals that the user encounters while invoking `step` repeatedly is more enlightening: It asks the user to prove `bst' ?x (fun x : Z ⇒ x = \[vAdd] ∨ s x) /[0]`, i.e. that at address 0, there is a binary search tree containing `vAdd`, the value being inserted. However, what we expect to prove is that this binary search tree is at some nonzero address `p`, which points us directly to our bug, namely that the pointer that should point to our newly allocated leaf still is 0 instead of `p`.

So, to summarize, this example shows that sometimes (in fact, often, in our experience), neither the full initial unprovable goal nor its smallest unprovable core is very enlightening, but the most enlightening goal is somewhere in-between during the automated proof process, and giving the users a means of running this automated proof process step-by-step enables them to understand more quickly why a goal cannot be proven.

### 5.4.9 Automated Splitting and Merging of Separation Logic Clauses

A common pattern in separation logic proofs is that a caller has a big separation logic clause (e.g., an array or a record) and needs to pass a subpart of it (e.g. one array element or a slice of the array or a record field) to an operation such as a memory load or store or a function call. Through specialized lemmas, VST [Cao et al., 2018, Section 5.4.2] provides automation for this pattern in the case of memory loads and stores, but it comes with two limitations: The automation only works for memory loads and stores, but not for function calls, and it only works if the access path is fully

spelled out at the point in the source program where the memory access happens.

An *access path* describes how to access a subpart of a nested combination of arrays and records, as e.g. in `.myField[i].otherField[j][k]`. However, as soon as the access path is not fully spelled out at the point in the source program where the memory access happens, VST's automation does not work fully automatically any more. An example of such a situation would be that before a loop, a pointer `p` is initialized to `&(myRecord.myField[0])`, and inside the loop, a memory load at `p` is made and `p` is incremented.

Our separation logic automation lifts both of the above limitations: To support function calls that modify a subpart of a callee's separation logic clause, the same automation as for memory loads and stores is used. That is, the split-and-merge procedure described in section 5.3.1.9 is also used for cases like the `Memcpy` call in Figure 5.8. Moreover, in order not to rely on access paths, all the reasoning is performed in terms of addresses (bounded integers), and to test which record field an address belongs to, queries to Coq's linear integer arithmetic solver `lia` are made.

## 5.5 Implementation Notes

### 5.5.1 Parsing C in Coq

Using Coq's notation system , we can declaratively write a parser for a big enough subset of C. Our ASTs use strings to represent identifiers, but we do not want double quotes around these strings to appear in our C code. Unfortunately, there is no officially supported way of getting rid of these quotes in Coq. But using a somewhat sinister trick by Pit-Claudel and Bourgeat [2021, Section 3] that combines notations, tactics in terms, and Ltac2's low-level API, we can give Ltac2 access to an unbound identifier, before Coq attempts to check the term and complains that the identifier is unbound, and we inspect the identifier in Ltac2 and convert the underlying OCaml string into a Gallina string.

### 5.5.2 Tailored Weakest-Precondition Lemmas

Based on `wp` rules like the ones in Figure 5.3, we prove another layer of `wp` rules (two of which are shown in Figure 5.11) that is tailored to work well with the proof-automation tactics.

While WP-SET uses two separate hypotheses for the evaluation of the expression $e$ and the remainder of the program *rest*, `wp_set` uses a judgment called `dexpr1` whose last argument is a proposition that needs to hold after evaluating `e`, so that changes to the symbolic state (i.e. changes

```
Lemma wp_set: forall fs x e v t m l rest post,
  dexpr1 m l e v (update_locals [|x|] [|v|] l (fun l' => wp_cmd fs rest t m l' post)) →
  wp_cmd fs (cmd.seq (cmd.set x e) rest) t m l post.

Lemma wp_while {measure: Type} (v0: measure) (e: expr) (c: cmd) t (m: mem) l fs rest
  (Inv: measure → trace → mem → locals → Prop) {lt} {post: trace → mem → locals → Prop}:
  Inv v0 t m l →
  well_founded lt →
  (∀ v t m l, Inv v t m l →
    ∃ b, dexpr_bool3 m l e b
      (loop_body_marker (wp_cmd fs c t m l (fun t m l => ∃ v', Inv v' t m l /\ lt v' v)))
      (pop_scope_marker (after_loop fs rest t m l post))
      True) →
  wp_cmd fs (cmd.seq (cmd.while e c) rest) t m l post.
```

Figure 5.11: Tailored Weakest-Precondition Lemmas

to the hypotheses of the proof goal) that are made while evaluating `e` are also visible to the proof code for the rest of the program. For instance, if the evaluation of `e` contained some memory access that treats some byte buffer as a record, the proof for `e` will change the corresponding hypothesis from a byte-array predicate to a record predicate, and it is usually desirable to preserve this change for the rest of the program.

Lemma `wp_while` (Figure 5.11) is based on WP-WHILE (Figure 5.3) but adds a termination measure that needs to decrease at the end of each iteration according to a user-provided less-than predicate `lt` which needs to be `well_founded`. The lemma contains markers such as `loop_body_marker`, `pop_scope_marker`, and `after_loop` (an alias of `wp_cmd`) that inform the tactics what to do. It uses a judgment called `dexpr_bool3`, whose last three arguments are propositions that need to hold in case the Boolean `b` obtained from evaluating the expression `e` turns out to be true, false, or either, respectively. For example, a loop searching through a tree where null pointers are used for leaves might start with `while (p && load(p) != key)`, and during the evaluation of the condition, in the case where `p` is non-null, this fact allows us to turn the memory assertion which says that at `p`, we either have a leaf or a node into one that says we certainly have a node at `p`, and using `dexpr_bool3` instead of a simple conjunction that gets split into separate subgoals allows us to keep this modification visible to the evaluation of the loop body.

### 5.5.3 Extracting Pure Facts From Sep Clauses

A separation-logic formula often contains some pure (i.e. heap-independent) facts, either by explicitly asserting them or because its definition implies them. For example, a `(ring_buffer cap vs a)` judgment declaring a circular buffer of capacity `cap` at address `a` containing the elements in list `vs` might imply the pure fact `len vs <= cap`.

In order to make such pure facts available to our solver for arithmetic side conditions, we define the judgment `purify R P := ∀ (m: mem), R m → P`, and whenever we define a new separation-logic predicate `R`, we also prove a corresponding `purify` lemma and register it in a custom hint database. Before running side-condition solvers, our framework searches the hint database for a purification rule of the form `(purify R _)` for each separation-logic clause `R` and applies all the rules it finds.

### 5.5.4 Pattern-Based Selective Warning Suppression

If the framework encounters a separation-logic clause for which it cannot find a `purify` hint or a `PredicateSize`, it emits a warning, because often, this is the reason a proof does not go through. But some clauses do not contain pure facts or do not have constant sizes. For these, we want to suppress the warning selectively. To do so, we use a Coq hint database to which we add the patterns of all warnings that should be suppressed. Compared to most other warning-suppression mechanisms, which only allow warnings to be suppressed by their kinds, ours also allows suppressing them based on their arguments, without any additional implementation effort: We just piggy-back on Coq's very general building blocks and benefit from implementing the framework in the same language as the user-facing language.

### 5.5.5 Mixed Word/Integer Arithmetic Side Conditions

When reasoning about array accesses and simplifying symbolic expressions indexing into lists, many arithmetic side conditions need to be solved. Since our specifications are written in terms of $\mathbb{Z}$, but the programs operate on a bounded `word` type, we obtain side conditions that mix the two. We solve such a goal as follows: First, if it is an equality or inequality on words, we use an injectivity lemma to reduce it to an equality or inequality on $\mathbb{Z}$. Next, we push down all conversions from word to $\mathbb{Z}$ (written as `\[x]`), transforming them into modulos. For instance, `\[a ^+ b]` gets rewritten to `(\[a] + \[b]) mod 2 ^ 32`. Then, we eliminate modulos using the Euclidean equations, leading to terms like `\[a] + \[b] - 2 ^ 32 * k`, where `k` is `(\[a] + \[b]) / 2 ^ 32`. For efficiency, our

implementation merges these two steps into one. This push-down of `\[_ OP _]` into `\[_] OP \[_]` with modulos is applied recursively until only variables or uninterpreted functions are wrapped in `\[_]`. Bounds are then asserted, since interpreting a word as an unsigned $\mathbb{Z}$ always leads to a number between `0` and `2 ^ 32`. Finally, we invoke Coq's linear-arithmetic solver `lia`.

### 5.5.6 Undoable, Reusable $\mathbb{Z}$ification

We call the preprocessing described in the previous subsection $\mathbb{Z}$ification. Before solving arithmetic side conditions, it has to be applied to the conclusion, as well as to all arithmetic hypotheses. Our bottom-up term-simplification procedure needs to invoke arithmetic-side-condition solving hundreds of times in order to find which simplifications to apply. For instance, when encountering a list slice starting at `i` of a list append like `(xs ++ ys ++ zs)[i:]`, we need to test whether `i` is $\leq 0$, points somewhere into `xs`, `ys`, or `zs`, or whether it exceeds the whole length, which already amounts to 5 separate queries. $\mathbb{Z}$ifying all hypotheses from scratch for each arithmetic side condition would be unacceptably slow. Instead, we implement $\mathbb{Z}$ification in such a way that it does not modify any hypotheses but just makes a $\mathbb{Z}$ified copy of each arithmetic hypothesis. Each time the user adds a new C snippet, we run hypothesis $\mathbb{Z}$ification once and reuse the $\mathbb{Z}$ified hypotheses for many side conditions, and just as the last step before marking the goal as ready for the next C snippet, we clear all the $\mathbb{Z}$ified hypotheses, so that a clean and concise context is presented to the user.

### 5.5.7 On-Demand Addition of Callee-Correctness Hypotheses

The correctness statement of a function in our framework whose direct callees are $c_1$, $c_2$, $\ldots c_N$ roughly looks like $\mathit{spec\ of}\ c_1 \wedge \ldots \wedge \mathit{spec\ of}\ c_N \Rightarrow \forall\, t\ m\ s.\ \mathit{Pre}(t, m, s) \Rightarrow \mathsf{wp}\ \mathit{env}\ (t, m, s)\ \mathit{body}\ \mathit{Post}$. Listing the specifications of the callees allows our proofs to abstract over callee implementations: For each callee, all we require is that the function environment *env* contains a function satisfying some spec. But at the moment we start a function's correctness proof, we do not know yet its function body, so we cannot yet determine its list of callees. We resolve this seemingly circular dependency using an evar. Each function definition is augmented with its list of callee specs, which is instantiated to an evar at the beginning of the proof. The hypothesis of the proof is then just a fold using logical "and" over this (not-yet-instantiated) list of callee specs. Whenever we call a function whose spec is `s`, we instantiate this evar to `(cons s ?new_evar)`, and at the end of the function, we instantiate the remaining evar to the empty list.

### 5.5.8 Discussion

In the following, we discuss a few design alternatives that we decided not to pursue further.

#### 5.5.8.1 Why Not a Stand-Alone Tool?

Building our framework inside Coq required us to go through some contortions, especially to make tactic invocations look like C snippets – clearly, Coq was not designed to do this.

In order to build a software-verification tool that provides a live display of the current symbolic state, we could also have built a stand-alone tool from scratch, which might have saved us some trouble and, if implemented well, might also have been more performant because it could be more specialized to our task, thus not having to pay the cost of being run inside a tool as generic as Coq.

However, Coq still has several advantages that made us choose it: Coq provides many term-manipulation facilities, including concise term matching, and its foundational proofs, i.e. proofs that are checked by its small proof-checking kernel, guarantee soundness, so that bugs in our framework cannot lead to wrong proofs, which allows us to modify the tool more freely and confidently, without worrying about soundness at each modification. Finally, working with Coq paves the way for connecting to the many other interesting verified artifacts in the Coq ecosystem.

#### 5.5.8.2 Limiting the Number of Conversions and Avoiding Operator Overloading

To avoid accidental overflows in our specifications, we write them using unbounded integers $\mathbb{Z}$, but the values treated by our programs are bounded 32-bit integers, and loading and storing 8-bit and 16-bit values is supported as well. Moreover, certain values in the specifications cannot be negative, so they would belong to $\mathbb{N}$. We tried using separate types for $\mathbb{N}$, $\mathbb{Z}$, 8-bit, 16-bit, and 32-bit words, but it led to two problems:

First, since Coq does not support subtyping natively, coercion functions are needed between different number types. Writing and displaying them explicitly becomes very verbose quickly, and relying on Coq's implicit-coercion feature did not work well. Coercions are inserted during type-checking, so patterns, which are untyped, do not have them inserted, which can lead to confusion. Coercions also make it harder to copy-paste a term from the goal into the proof script, because one might miss a coercion that only gets inserted because of the surrounding context.

The second problem was operator overloading: We would like to use some short infix notation for common operators like addition, subtraction, etc. Coq provides a mechanism called notation scopes that works well as long as no polymorphic functions are used, because when parsing the

arguments of a function, Coq relies on the signature of the function to determine in which notation scope (e.g. the notation scope for $\mathbb{N}$ or for $\mathbb{Z}$) to parse the arguments. Another popular mechanism for operator overloading is to use type classes. For instance, the infix notation `(a + b)` might be defined as `(TypeclassBasedAdd a b)`, where `TypeclassBasedAdd` takes an implicit argument that is a type-class instance implementing addition on the type of `a` and `b`. However, if we simplify terms or obtain terms from third-party libraries not using such a type class-based notation system, they contain the plain `(Nat.add a b)` instead of our type class-based one, so they will not be printed the same and will not match our terms syntactically. Similar problems occur with a related approach based on canonical structures. We also tried an operator-overloading approach using notations with tactics in terms that type-checks the operands and picks the right operator based on the type of the operands, resulting in plain terms like `(Nat.add a b)`, combined with ambiguous printing-only notations that use the same + symbol for addition on all types. It was a bit heavy-weight and did not work in patterns (because they are not type-checked), so we stopped using it.

Finally, we decided to restrict ourselves to just two number types: 32-bit words and $\mathbb{Z}$. This approach only requires three coercions: truncating a $\mathbb{Z}$ to a bounded integer (which we write as `/[x]`), interpreting a word as a signed integer (which we use less frequently and write as `word.signed x`), and interpreting a word as an unsigned integer (which we write as `\[x]`). It also only requires two sets of infix arithmetic operators, so we use the regular operators for $\mathbb{Z}$ and operators prefixed by `^` such as `^+`, `^-`, etc. for words.

#### 5.5.8.3 Selectively Unfolding wpgen vs. Repeatedly Applying WP Lemmas

Repeatedly unfolding the leftmost occurrence of a weakest-precondition *generator* (section 5.2.1) in the proof goal results in the same series of proof goals as repeatedly applying weakest-precondition *rules* (section 5.2.3).

The reason we prefer the latter approach is due to a design choice in Coq's implementation: While constructing proofs using tactics, each application of a lemma is recorded as a function application in the proof term being built, but unfolding of definitions and simplification of terms according to the evaluation rules of the Gallina language are not recorded in the proof term. Therefore, at the end of the proof, when the generated proof term is rechecked by Coq's small trusted kernel, the type checker will encounter situations where the inferred type of a proof term does not match the expected type of a proof term, and it needs to unify these two types by unfolding and evaluating some definitions appearing in the two types. To do so, it uses some heuristics based on lazy evaluation but does not have any way of seeing the simplifications that were done during the

construction of the proof. Therefore, relying on these unification heuristics too much can lead to very slow `Qed` processing times, and it is preferable to use explicit lemma applications that leave a trace of the kind of performed simplifications in the proof term.

#### 5.5.8.4 Implementation Language

Our framework is implemented using a mix of tactic scripts and lemmas and definitions in Gallina (Coq's specification language) that are specifically tailored to work well with our tactic scripts. For compatibility with other code from the Bedrock2 ecosystem, we refrain from modifying Coq itself (even though such modifications might have simplified certain parts); and for easy compatibility with new Coq versions, we refrain from writing any OCaml plugins, because Coq's OCaml API tends to change with each Coq version. The tactics are implemented in a mix of Ltac1 and Ltac2.

#### 5.5.8.5 Ltac1 vs. Ltac2: When to Prefer an Untyped Language With Undocumented Semantics

Ltac1 is an untyped language without clearly specified semantics. For instance, whether a variable refers to a binder declared in Ltac, to a binder declared in a Gallina term quoted inside Ltac code, or to the name of a hypothesis in the current proof context is decided at runtime, in a barely documented manner. It can also happen that a thunk being passed to a function and meant to be evaluated lazily can accidentally be evaluated eagerly. Another common source of surprises is that whether a tactic is a pure function returning a term or an imperative program modifying the current proof goal is also decided at runtime.

Ltac2 addresses these shortcomings by being a typed language with straightforward call-by-value semantics and unambiguous quotation mechanisms to make it clear what variables refers to. In addition, it offers some low-level APIs that Ltac1 does not have.

Given this situation, one might expect that the unambiguously preferred choice for the whole framework would be Ltac2. However, this is not the case in our experience:

First, even though Ltac2 has been developed over several years now, it still lacks support for many tasks that can be done in Ltac1 much more easily, so when writing Ltac2, a considerable amount of time is spent working around non-fundamental limitations related to not-yet-implemented features. And second, Ltac1 code is exceptionally concise, in a manner that really matters: In our experience, there seems to be a certain verbosity threshold below which a tactic programmer can read and understand tactic code very quickly and easily, and Ltac1 is the only programming language we know to be below this verbosity threshold. The reason for Ltac2 often being above it seems to be on one hand that it is typed and more principled, i.e. it requires being

explicit about many things that are implicit in Ltac1; requiring more explicit quotation, unquotations, thunking, and other requirements of being explicit about things that are implicit in Ltac1 (e.g. when to focus on one goal) and on the other hand, that less effort has been spent yet on defining concise notations for Ltac2. We are curious to see how future evolution of Ltac2 affects these considerations.

For now, we use a mix of Ltac1 and Ltac2, preferring Ltac2 for bigger, more complex functions, where the benefit of catching errors before tactic runtime is considerable, and for situations where low-level term APIs are needed.

## 5.6 Evaluation

### 5.6.1 Scope of Sample Programs

We used our tool to verify a few sample functions listed in Table 5.2, trying to cover an interesting set of low-level memory-handling patterns. It includes splitting a byte buffer into a linked list of free blocks in the init function of a simple malloc library with a fixed block allocation size, lookup and insertion functions for a binary search tree and a crit-bit tree [Bernstein, 2006] where we exploit the pre-/postcondition loop verification style by Tuerk [2010] to avoid the need for "tree-with-a-hole" predicates, passing record fields and subarrays to functions with automatic splitting of the callers' record or array predicates, and functions with up to three if-then-else constructs.

The crit-bit tree example shows that we can also support data structures with more involved validity constraints, at the expense of more manual proof lines, though we believe that a more automated proof style could reduce the number of lines of proof. This example also provides a data point on usability of our framework, because the example was developed by an undergraduate student, Viktor Fukala, who did not participate in the development of the framework and had started to learn Coq less than three months before completing this proof of crit-bit lookup and insert functions (`critbit v1` in Table 5.2). Later, he also added functions for deletion, obtaining min and max keys as well as the next key greater than a given key and support for iteration, leading to a development [Fukala, 2024] of 356 lines of C code spanning 23 functions, inside a Coq file with 5495 lines of code (`critbit v2` in Table 5.2).

| File | Funcs | Snippets | Lines | Time[s] | Loops |
|---|---|---|---|---|---|
| `onesize_malloc` | 3 | 24 | 345 | 20 | 1 + 0 |
| `tree_set` | 4 | 66 | 389 | 74 | 0 + 2 |
| `swap` | 2 | 10 | 44 | 4 | – |
| `swap_record_fields` | 2 | 6 | 83 | 4 | – |
| `fibonacci` | 1 | 17 | 83 | 9 | 1 + 0 |
| `memset` | 1 | 7 | 41 | 9 | 1 + 0 |
| `sort3` | 1 | 22 | 51 | 36 | – |
| `swap_subarrays` | 1 | 3 | 48 | 16 | – |
| `sort3_separate_args` | 1 | 22 | 58 | 23 | – |
| `linked_list` | 2 | 16 | 252 | 10 | 1 + 1 |
| `nt_uint8_string` | 1 | 11 | 299 | 61 | 0 + 1 |
| `min` | 3 | 32 | 71 | 7 | – |
| `critbit v1` | 8 | 122 | 1881 | 255 | 2 + 2 |
| `critbit v2` | 23 | 356 | 5495 | 455 | 3 + 9 |

Table 5.2: Statistics on our case studies. The two numbers in the Loops column indicate the number of invariant-based loop proofs and the number of of pre-/post-based loop proofs, respectively.

### 5.6.2 Qualitative Discussion of Loop-Invariants-as-Diff Approach

As illustrated by the example in section 5.3.1.7, in our framework, users express loop invariants as diffs from symbolic state before loops. Table 5.3 shows why we prefer this middle ground over the two extremes in the design space, manually spelled-out invariants or automatically inferred invariants.

By robustness, we mean how likely it is that after a small modification of the program, the proof still works. Manually spelled-out loop invariants are very likely to require some update after a program modification, whereas an invariant expressed as a diff that just encodes the insight and avoids mentioning irrelevant details is more likely to remain applicable. Automated invariant inference tends to be not so stable under modification of the proof context, because the presence of a new but unrelated term might send the invariant search down a wrong path, so that an invariant that was found within reasonable time before the program change might time out after the change.

By proof performance, we mean the running time it takes to produce and check the correctness proof. Executing the diff script corresponds to proving that the symbolic state before the loop implies the loop invariant, i.e. proof work that any framework needs to do, so we do not count it.

The expressivity of the manual and the diff approach is maximal, because any invariant express-

| | manual | by diff | automatic |
|---|---|---|---|
| verbosity | ★ | ★★ | ★★★ |
| robustness | ★ | ★★ | ★★ |
| proof performance | ★★★ | ★★★ | ★ |
| fully expressive | ✓ | ✓ | ✗ |
| can display state | ✓ | ✓ | ✗ |
| total | 5★ + 2✓ | 7★ + 2✓ | 6★ + 0✓ |

Table 5.3: Tradeoffs in the design space around loop-invariant automation. Conjectured ratings on a one-star (worst) to three-star (best) rating scale.

ible in the logic can be used, whereas in the automatic approach, only those that the heuristics find within a reasonable time limit can be used.

Another advantage of our approach (shared with the approach of manually providing loop invariants) is that we can display a symbolic state at any point inside the loop body even if the loop body has not been completely written yet or some parts of the proof fail because of a bug in the code or because of a missing hint or tweak. In contrast, the fully automatic approach only knows that it picked a reasonable loop invariant if the correctness proof of the whole loop worked out.

Currently, the star ratings in Table 5.3 are not based on measurements but on anecdotal evidence, so it is cautious to view them as conjectures. In the future, we hope to back them up with measurements, but currently, our framework is still in an early prototype phase where most new examples that we verify point us to some bugs and limitations in the framework that we fix on the fly, but for a meaningful evaluation, one should not make fixes to the framework while evaluating it.

### 5.6.3 Some Statistics

Some file-by-file statistics are shown in Table 5.2. The first column lists the number of functions in each file, and the second lists the number of snippets, which typically corresponds to the number of lines of C code. The total number of lines of each file (third column) is much bigger, because the files also contain specifications, definitions needed to state the specifications, helper lemmas, file-specific proof automation and hints, as well as proof code interspersed between the C snippets.

Table 5.2 also shows the total time Coq takes to verify each file. Typically, processing each

snippet takes just a couple of seconds, and in our experience, it is just right below the threshold of what is bearable for interactive development (and whenever it exceeded that perceived threshold, we spent more effort on speeding up the proof automation).

The final column shows the number of loops in each file, expressed as $x + y$, where $x$ is the number of loops proven with an invariant expressed as a diff script from the symbolic state before the loop, and $y$ is the number of loops proven with a family of pre/postcondition pairs (in the style popularized by Tuerk [2010]) by expressing the precondition as a diff script and automatically generalizing the function's postcondition to use it as the loop's postcondition.

So far, our experience seems to confirm our conjectures from Table 5.3. Once our framework has matured to a point where we do not anymore feel compelled to make framework improvements with every new sample program, we plan to evaluate our conjectures from Table 5.3 more rigorously.

We conjecture that describing loop invariants and loop pre/postconditions in terms of edits needed to obtain them from the current symbolic state rather than spelling them out completely in the proof script or as annotations in the source code has the following advantages:

1. It leads to an easier, more intuitive, and more enjoyable user experience.
2. The proofs are more robust against code changes, because diffs (edits) tend to be smaller than whole invariants.

Unfortunately, it is hard to evaluate these conjectures empirically, because our prototype implementation has not yet reached the level of usability and maturity that would be required in order to ask a statistically significant group of test users to solve some sample tasks.

## 5.7 Related Work

Dafny [Leino, 2013, 2017] is a high-level programming language with a specification language and SMT-based, highly automated proving of verification conditions. The development experience is very interactive, as the IDE continuously checks the verification conditions. Our framework is still far from reaching the level of automation of Dafny but does have a few advantages over Dafny:

- It allows to reason about (a subset of) C, which is more low-level and more efficient, and can reason about low-level operations like casting byte arrays to records.
- Users can extend the proof automation with domain-specific verification procedures.
- By repeatedly invoking our `step` tactic, users can watch how our system solves side conditions and can easily debug cases where our solver fails.

- The correctness of the tool itself need not be trusted, only Coq's kernel, which is much smaller than Coq's tactic system and our tool's tactics, and also much smaller than the Dafny tool and the SMT solver it uses.
- Finally, and perhaps most importantly, our tool can provide a concise representation of everything the prover knows, in the form of the proof context (list of hypotheses) of Coq's current proof goal. We believe that such a concise summary of all known facts is similar to what attentive programmers need to keep track of in their minds while programming, so displaying it on-screen can assist the programmer. In Dafny, there is no such representation, and the only way to find out whether the prover knows a given fact is to write it down as an assertion at the program point in question and see if Dafny can prove it.

VeriFast [Jacobs et al., 2011] is a separation-logic-based C verification tool. Its symbolic debugger can display the current symbolic state to the user at any program point, and users can affect the symbolic state by invoking lemma functions in ghost code (comments) in the source program. Thus, they have a way of modifying the symbolic state, and the tool ensures that these modification are sound, because the lemma functions need to be proven in VeriFast as well. VeriFast has a symbolic debugger and an IDE that allows developers to inspect the symbolic state at each program point (see e.g. Figure 6 in [Jacobs et al., 2010]). VeriFast is implemented in OCaml. As far as we know, there is no easy way to add domain-specific verification automation on a per-function or per-module basis, while our own approach provides various Ltac hooks and hint databases that users can extend and provides smooth integration between framework code and user code, because both are written in the same language (Ltac).

Boogie, the intermediate verification language powering Dafny, used to have a verification debugger [Le Goues et al., 2011] providing counterexamples for failed verification conditions. However, it appears that it was not popular enough to be maintained, and was eventually removed from the code base [Qadeer, 2020]. In the design space between automatically inferred and manually spelled-out loop invariants, Boogie chooses an interesting middle ground: It infers some simple loop invariants and combines them with those written explicitly by the user [Barnett et al., 2006].

The KeY project [Ahrendt et al., 2016] uses the Java Modeling Language (JML) to specify the behavior of Java functions and provides an Eclipse IDE plugin to verify Java programs. It features a Symbolic Execution Debugger [Hentschel et al., 2014].

Rupicola [Pit-Claudel et al., 2022] and its predecessor Fiat [Delaware et al., 2015] are extensible, user-configurable compilers from functional programs written in Coq to Bedrock2 and Bedrock1 code, respectively. The user specifies the compilation strategy and lets the framework derive the

code accordingly. The Isabelle Refinement Framework [Lammich, 2015, 2017] applies similar techniques in Isabelle/HOL. In contrast, our framework is designed for users who already have a clear idea of what low-level code they want and feel that configuring the compiler until it emits the desired low-level code would be more work than just writing down the code.

The Verified Software Toolchain (VST) [Cao et al., 2018] is a tool based on Hoare logic and separation logic, implemented in Coq, for proving correctness of C programs. It uses a similar style of stepping through a program line-by-line, using Coq's context of hypotheses to keep track of the symbolic state. Instead of using Hoare triples $\{P\}c\{Q\}$ like VST, we use wp judgments of the form $\forall s.\ P\ s \Rightarrow \mathsf{wp}\ c\ s\ Q$, so the precondition is already separated and can more easily be moved into Coq's context of hypotheses. In VST, one has to recompile the source program and reload the whole proof each time one wants to change the source program, and sometimes, it is hard to relate the positions in the proof script to positions in the source code.

Like Bedrock2 (which our Live Verification framework targets), CakeML can also be used to create end-to-end-verified software-hardware stacks [Lööw et al., 2019]. All their program verification happens at the ML level, whereas we believe that certain performance-critical pieces of software need to be written in more imperative and low-level languages, which is what our framework enables.

One goal of building our Live Verification framework was to make it easier to verify efficient source programs that are compatible with the Bedrock2 ecosystem, to create end-to-end-verified software-hardware stacks. A similar end-to-end-verified software-hardware stack was also built by the CakeML team [Lööw et al., 2019]. Instead of a low-level C-like language, they use ML as their source language, so they need to spend less work on the verification of the source programs, because they are already much closer to the desired functional high-level specification, but need to spend more effort on turning ML programs into efficient low-level programs in their CakeML compiler optimizations [Kiam Tan et al., 2019].

Why3 [Filliâtre and Paskevich, 2013; Bobot et al., 2015] is a tool for interactive development and verification of programs. It provides a programming and specification language called WhyML and can also be used as an intermediate language to verify C, Java, and Ada programs. It discharges its verification conditions to automated as well as interactive external theorem provers.

CAPS [Chaudhari and Damani, 2015, 2014], which stands for Calculational Style of Programming, uses a tactic-based approach to derive programs from specifications and uses Why3 as its backend. It uses Why3 as an automated theorem prover and maintains the invariant that as the user constructs more and more of the program, the program constructed so far is always proven correct.

## 5.8 Conclusion and Future Work

We have presented a tool for verifying low-level programs using the Coq proof assistant, in a way that continually provides a concise representation of the current symbolic state as the user writes the program. Additionally, our tool stands out by its support for diff-based loop invariants, its option to allow users to extend the proof automation with domain-specific procedures, its small trusted code base that does not include the tool itself, and its compatibility with the Bedrock2 ecosystem that enables end-to-end proofs, which also check that the assumptions that the different tools make about each other are compatible.

In terms of proof automation and usability, we feel that it is an improvement over the previous Bedrock2 program logic and other approaches in Coq, but compared to more mature tools like Why3 or Dafny, there is still a lot of room for improvement, which is to be expected, because Why3 has been under active development for over 20 years [Filliâtre, 2009] and Dafny for over 15 years [Barnett, 2009].

The eventual goal of our line of research is to turn software verification into an enabler of more efficient software development, considering the overall cost of development, testing, and debugging. In particular, we hope that the constantly updated display of a symbolic state can assist the programmer when writing programs. For tiny examples like computing the minimum of three numbers without flipping the direction of the comparisons by accident, we have experienced the feeling that indeed, the display of the symbolic state made us more productive at writing this (admittedly highly trivial) program. For more complex programs, the tool is certainly useful for proving correctness, but we believe that in addition, it can also be useful merely for knowing what snippet to write next, but in the current state of the tool, the frequently failing verification of side conditions constitutes a distraction that hampers this dimension of usefulness.

It seems to us that the size of the biggest case study in Bedrock2 [Erbsen et al., 2021] was mostly bottlenecked by the lack of automation and usability of the program logic. Similar limitations apply to other Coq-based C verification tools like e.g. VST [Cao et al., 2018] as well. With our live-verification framework, we hope to make a step towards more convenient verification of low-level code in Coq, eventually enabling bigger end-to-end verified stacks.

## 5.9 Listing of Notations

In this glossary, we give semi-formal definitions of all notations used in the code examples. For each notation (or group of notations), we point to a section that explains it and also indicate the Coq file where the precise definition can be found. Snippets in `typewriter` font are literal parts of the syntax of the notation, while names in *italics* are placeholders.

**Specification of a function** § 5.3.1.3, `LiveVerif/src/LiveVerif/LiveProgramLogic.v`

```
.**/ uintptr_t fname(uintptr_t arg_1, ..., uintptr_t arg_N) /**#
       ghost_args := ghost_1 ... ghost_M;
       requires t m := Pre;
       ensures t' m' r := Post #**/  /**.
```

The first line of this notation is chosen to be compatible with C function signatures. Note that since `uintptr_t` is currently the only supported type, it is a literal part of the notation and cannot be replaced by anything else. $arg_1, \ldots arg_N$ are the function argument names, and $ghost_1, \ldots ghost_M$ are ghost arguments (whose types are usually inferred by Coq, but can also be annotated explicitly like in $(ghost_i : MyType_i)$, where $MyType_i$ can be any Coq type). *Pre* is the precondition, and it can refer to the binders $t$ and $m$, which stand for the event trace and memory before the function call. *Post* is the postcondition, and in addition to $t$ and $m$, it can also refer to $t'$ and $m'$, which stand for the event trace and memory after the function call, and if the function has a return value, to a binder $r$ representing that return value. It roughly expands to the following formula, which is parameterized over the function environment $e$:

$$\lambda e.\ \forall arg_1 \ldots arg_N\ ghost_1 \ldots ghost_M\ t\ m,\ Pre \Rightarrow \mathsf{wp}\ e\ fname(arg_1, \ldots, arg_N)\ t\ m\ (\lambda\ t'\ m'\ r\ .\ Post)$$

There is also a variant of this notation with a `void` return type, where the `ensures` clause only takes $t'$ $m'$ instead of $t'$ $m'$ $r$.

**Providing the next C snippet** § 5.3.1.5, `LiveVerif/src/LiveVerif/LiveProgramLogic.v`

`.**/` *snippet* `/**.` — a tactic notation to provide the next snippet of C code. Typically, we write the `.**/` at the end of the preceding line, and fill in whitespace between *snippet* and the `/**`. The *snippet* is parsed using a custom grammar for C snippets, and passed as an argument to the tactic `next_snippet`, which applies the appropriate weakest-precondition rule, and then repeatedly applies the `step` tactic to solve the side conditions.

**Providing the next C snippet without automation** § 5.4.8.3, `LiveProgramLogic.v`,

`.**/` *snippet* `/*?.` — the same as `.**/` *snippet* `/**.`, but without running the `step` tactic

**Bullet-point list of separation-logic clauses** § 5.3.1.3, `bedrock2/src/bedrock2/SepBulletPoints.v`

`<{ *` $clause_1$
`...`
`*` $clause_{N-1}$
`*` $clause_N$ `}>` — stands for `(sep` $clause_1$ `...` `(sep` $clause_{N-1}$ $clause_N$ `) ...)`, where `sep` is the standard separating conjunction (usually written as $*$ in the literature)

**Bullet-point list of sepapp clauses** § 5.4.1.2, 5.4.2, `bedrock2/src/bedrock2/SepappBulletPoints.v`

`<{ +` $clause_1$
`...`
`+` $clause_{N-1}$
`+` $clause_N$ `}>` — stands roughly for `(sepapp` $clause_1$ `...` `(sepapp` $clause_{N-1}$ $clause_N$ `) ...)`, where `sepapp` means *separating append* (section 5.4.1.2)

**Marker hypotheses** § 5.4.4, `LiveVerif/src/LiveVerif/PackageContext.v`

`____` *scopekind* `____` — in a proof goal, marks the beginning of a scope of kind *scopekind*

**Referring to hypotheses by pattern** Figure 5.4c, `bedrock2/src/bedrock2/find_hyp.v`

`#(`*pattern*`)` — returns the name of the first hypothesis that matches *pattern*, which may contain `??` placeholders

**Heapletwise separation logic** § 5.3.1.8, `bedrock2/src/bedrock2/HeapletwiseHyps.v`

$m_1$ `\*/` $m_2$ — disjoint union between two heaplets (pieces of memory)

$m$ `|=` $P$ — heaplet $m$ satisfies predicate $P$, defined as $(P\ m)$

**Notations for bounded integers** § 5.3.1.3, 5.5.8.2, `bedrock2/src/bedrock2/WordNotations.v`

| | |
|---|---|
| `\[`$w$`]` | interprets a `word` $w$ as an unsigned integer, returning a $\mathbb{Z}$ |
| `/[`$z$`]` | coerces a $\mathbb{Z}$ into a `word` by dropping the more significant bits |
| `^+` `^-` `^*` | addition, subtraction, multiplication on `word` |

**List notations** § 5.3.1.4, `deps/coqutil/src/coqutil/Datatypes/ZList.v`

| | |
|---|---|
| $\ell$`[`$i$`]` | $i$-th element of list $\ell$ |
| $\ell$`[`$i$ `:=` $x$`]` | $\ell$ with its $i$-th element replaced by $x$ |
| $\ell$`[:`$i$`]` | the first $i$ elements of $\ell$ |
| $\ell$`[`$i$`:]` | $\ell$ with its first $i$ elements skipped |
| $\ell$`[`$i$`:`$j$`]` | the slice of $\ell$ from index $i$ (inclusive) to index $j$ (exclusive) |
| `[\|`$x_1$`;` $x_2$`; ...;` $x_N$`\|]` | list literal |
| $h$ `::` $t$ | list cons |
| $a$ `++` $b$ | list append |

**Potentially less-well-known standard Coq notations**

| | |
|---|---|
| $\sim P$ | negation of proposition $P$, defined as $P \to \bot$ |
| `?`$x$ | an evar (see section 5.2.5) |
| $a$ `^` $b$ | $a$ raised to the $b$-th power |

# Chapter 6

# Simplification of Expressions Describing Symbolic State

## 6.1 Problem

In automated program verification tools that never show their internal proof state to the user, it is not too much of a problem if symbolic expressions representing program state become somewhat big, as long as the prover can handle them. For interactive program verification tools, however, which display the current symbolic state to the user after each command of the source program, it is important that this state is kept concise and readable.

For example, in the `memset` function, which initializes each byte of a byte array at address `a` whose initial value is a list of bytes `bs` to a given constant `b`, the loop invariant says that before the `i`-th loop iteration, the contents of the byte array are[1]

```
List.repeatz \[b] \[i] ++ bs[\[i]:]
```

That is, the array is initialized to the desired value up to index `i`, but still contains the original values between index `i` and the end of the array. The loop body stores the value `b` at address `a + i`. The proof automation[2] applies a lemma whose conclusion says that the list now equals the previous list up to index $k$, followed by a singleton list containing `b`, followed by the previous list starting at $k + 1$, where the index $k$ equals `(storeAddress - baseAddress) / elementSize`, and `storeAddress = a ^+ i`, `baseAddress = a`, `elementSize = 1`.

[1] See section 5.9 for an explanation of the notations used in this example

[2] See also section 5.3.1.9

So, the updated list appearing in the symbolic state now looks as complicated as

```
(List.repeatz \[b] \[i] ++ bs[\[i]:])[:\[a ^+ i ^- a] / 1] ++
[|\[b]|] ++
(List.repeatz \[b] \[i] ++ bs[\[i]:])[\[a ^+ i ^- a] / 1 + 1:]
```

Clearly, such an expression should not be presented to the user. Instead, it should be simplified to

```
List.repeatz b i ++ [|b|] ++ bs[i + 1:]
```

and then further to

```
List.repeatz b (i + 1) ++ bs[i + 1:]
```

One might hope that by adding more specialized lemmas for commonly appearing special cases, one could get simpler resulting symbolic states. For instance, a lemma specialized to the case where the element size is 1 could get rid of the division, and a lemma specialized to the case where `storeAddress` matches the pattern `(baseAddress ^+ elementSize ^* _)` could simplify the expressions appearing in the list slicing expressions, but maintaining such a (potentially ever-growing) collection of common patterns and encountering undesirable behavior whenever a programmer writes a program that is not part of the anticipated common patterns seems undesirable.

Instead, let's see how a more principled and general term simplification mechanism for this kind of expressions can be implemented.

### 6.1.1 Going Beyond Rewrite Rules: The Need for Custom Procedures

One might hope that all required simplifications can be expressed as rewrite rules, which could then be fed into a generic rewrite engine. For the above example, this might be the case, but if we slightly generalize the simplifications required in the example, simple rewrite rules do not suffice anymore, as two subterms of the above examples illustrate:

To simplify `(a ^+ i ^- a)` into `i`, one might add a rewrite rule, but as soon as we encounter an example where the two expressions that cancel each other out are a bit further apart, such as e.g.

```
(((a + x1) + x2) + x3) - (y1 + (y2 + (y3 + a)))
```

the rule would not apply anymore. Similarly, if `(List.repeatz b i)` and `[|b|]` appear as the left and right argument of a `++`, a rewrite rule can merge them into `List.repeatz b (i + 1)`, but if they are buried deeper inside other concatenation expressions, such as e.g.

```
(xs1 ++ (xs2 ++ (xs3 ++ List.repeatz b i))) ++ ((([|b|] ++ ys1) ++ ys2) ++ ys3)
```

a simple rewrite rule does not work anymore. The situation could be slightly improved by first canonicalizing the associativity, but then, each rewrite rule that replaces an `(xs ++ ys)` by a `cs` would also need to be added in a second form that replaces all `(xs ++ (ys ++ zs))` by `cs ++ zs`.

One might argue that these more complex examples where simple rewrite rules do not work are artificial and do not appear typically appear during program verification, and one might even be able to support this claim by giving a series of verified programs that all are verifiable with simplification that only uses simple rewrite rules. But such reasoning misses an important point: Verification tools should work well not only on proofs that work, but also on proofs that do not work (yet), because that is the steady state of proof development. In particular, if one simplification that the user expected to trigger did not trigger because the user forgot to include some precondition that is required for the simplification, and further symbolic manipulation is applied to a not fully simplified expression, the expression can quickly grow quite complex, and in order to help the user figure out why the proof does not work, it is really important that the verification tool also can simplify as much as possible on the bigger expression – even if this expression will never occur anymore during verification of the final proof where everything works.

## 6.2 Related Work

Coq ships with two solutions for rewriting:

The first, `autorewrite`[3], performs rewrites based on hint databases that the user can populate with rewrite rules. The rewrite rules may have side conditions, and arbitrary tactics can be registered to solve these side conditions. However, it has a long-standing bug[4] that makes it basically unusable: If there are two rewrite opportunities for a given rewrite rule, but for the first one, the side condition prover fails, the rewrite rule is not tried for the second opportunity.

Coq's second rewriter is `rewrite_strat`[5], which provides a small DSL to describe the desired term traversal order, and also supports hint databases like `autorewrite`. It does not have the `autorewrite` bug, but instead a series of other bugs and limitations that make it undesirable to use.[6]

Gross et al. [2022] developed a performant rewriter in Coq based on reification, which was successfully used in Fiat Cryptography [Erbsen et al., 2019], but it only supports executable Boolean side conditions that only contain compile-time-known constants.

[3] https://coq.inria.fr/doc/V8.19.0/refman/proofs/automatic-tactics/auto.html#coq:tacn.autorewrite
[4] https://github.com/coq/coq/issues/7672
[5] https://coq.inria.fr/doc/V8.19.0/refman/addendum/generalized-rewriting.html#strategies-for-rewriting
[6] See e.g. Coq issues 10848, 10934, 12053, 13708, 13712, 15093, 15701, and 16471

All the three above Coq solutions have the additional limitation that they are purely based on rewrite rules and cannot be combined with custom rewrite procedures as section 6.1.1 wishes.

Though I have not used them myself, Isabelle simprocs [Urban, 2019, section 5.5] seem to satisfy the requirements described before: The simplifier (simp) proceeds through terms bottom-up [Nipkow et al., 2024, section 9.1.2] and applies rewrite rules from the user-specified rewrite databases as well as user-specified simplification procedures (simprocs). Its only drawbacks are that it is not available in Coq, and that the user-specified simplification procedures need to be implemented in ML, which is fairly low-level compared to Ltac or Ltac2 that we will use in our solution.

## 6.3 Attempt 1: Ad-Hoc Rewrites and Simplifications

The first attempt was to interleave invocations of Coq's basic `rewrite` tactic (which performs one rewrite at a time) with invocations of its `ring_simplify` tactic to simplify word expressions like `(a ^+ i ^- a)` into `i`.

However, this approach is too slow, because each rewrite and each procedure that gets applied to a subterm of the whole term needs to register in the proof term where it has been applied, and the way this is encoded in Coq requires copying the whole surrounding term.

To illustrate the problem, consider the following proof, where the repeated application of `f` stands for some big term, and the lemma `g_is_h` says **`forall`** `x, g x = h x`:

```
Lemma some_rewrite: forall x, f (f (f (f (g x)))) = f (f (f (f (h x)))).
Proof.
  intros.
  rewrite g_is_h.
  reflexivity.
Qed.
```

Its proof term is

```
eq_ind_r (fun n : nat ⇒ f (f (f (f n))) = f (f (f (f (h x))))) eq_refl (g_is_h x)
```

and uses the lemma `eq_ind_r` of type

```
forall [A : Type] [x : A] (P : A → Prop), P x → forall y : A, y = x → P y
```

As we can see, to indicate where to apply the `g_is_h` lemma, the `P` argument of `eq_ind_r` is instantiated with a lambda that repeats the whole surrounding term (the `f` applications), and if we were to run many rewrites in a row, each of them would repeat the whole context around the term being rewritten, leading to quite big proof terms and slow performance.

## 6.4 Attempt 2: E-Graphs

Another problem is that it is not always clear what rewrites will lead to the smallest and nicest possible simplified terms. Using e-graphs [Nelson, 1980] can help: We can start by representing the term to be simplified as an e-graph, and then repeatedly apply rewrites (which unifies e-classes) until saturation or a predefined limit is reached. Then, using a bottom-up dynamic-programming approach, we can extract the smallest representative of each e-class. As a size measure, we can either use the number of AST nodes, or we can also customize it to weigh definitions that we do not like higher, so that they are less likely to appear in the simplified term. With my colleague Thomas Bourgeat, I implemented this approach as a Coq plugin [Bourgeat, 2023, Chapter 7], based on the egg e-graph implementation by Willsey et al. [2021].

However, a significant limitation of the egg implementation is that it does not support theory combination. In particular, it was unclear how one would use a linear integer arithmetic solver to discharge side conditions of the rewrite rules in this setup.

## 6.5 Current Solution

To avoid the problem of big proof terms described in section 6.3, we implement a bottom-up term traversal procedure in Ltac2 that applies rewrite rules and custom simplification procedures at each node, and produces an equality proof term on-the-fly.[7] Each rewrite rule invocation in the proof term still repeats its arguments, but there are no more context terms needed for each rewrite. Specifically, the simplifier returns a value of type

```
Ltac2 Type res :=
  [ ResNop(constr) (* new and old term *)
  | ResConvertible(constr) (* new term *)
  | ResRewrite(constr, constr) (* new term, proof *) ].
```

where `ResNop` means that no simplifications were made (and the original term, a `constr`, is just returned for convenience and uniformity), `ResConvertible` means that the term can be simplified to a new term according to Coq's reduction rules, which does not require an explicit proof term, and `ResRewrite` means that rewrites were made that result in a new term stored in the first argument and that it is justified by a proof term stored in the second argument of the constructor.

[7] See https://github.com/mit-plv/bedrock2/blob/800a8a15c599/bedrock2/src/bedrock2/bottom_up_simpl.v for the source code

Using congruence lemmas like $\forall f\ g\ x\ y,\ f = g \Rightarrow x = y \Rightarrow f\ x = g\ y$ and a few specialized versions of it, the procedure combines results as it travels up the term.

At each node, it invokes a customizable *local simplifier*. At the moment, the following local simplifiers are invoked:

- `local_follow_eqs_until_const_val e` checks if the expression `e` is a variable, and if so, whether there is an equation of the form `e = rhs` in the context with `rhs` a constant or again a variable, and continues following equations until a constant is reached. If `rhs` is not a constant, it fails, because inlining all variables risks leading to big terms.
- `local_zlist_simpl e` pushes down list expressions `_[_:]`, `_[:_]`, `_[_]` into list concatenations, canonicalizes the associativity of concatenation, but also tries for each pair of list expressions appearing left and right of a concatenation whether they can be merged into one expression.
- `local_ring_simplify parent_kind e` applies Coq's `ring_simplify` procedure on `e`, but only if the `parent_kind` is different from `e`'s `kind`, where `kind` is one of `WordRingExpr`, `ZRingExpr`, `OtherExpr`, to make sure it is not uselessly applied on subterms whose parents it will get applied to later anyways.
- `local_ground_number_simpl e` simplifies `e` if it can be simplified to a constant number.
- `push_down_len e` pushes down list length expressions into other list expressions.
- `local_word_simpl e` pushes down `word.unsigned` (injection of bounded integers into $\mathbb{Z}$) into word operations.
- `local_nonring_nonground_Z_simpl e` runs a simple canceler for divisions.

## 6.6 Preliminary Evaluation

The simplification procedure described above runs after each addition of a C snippet in our Live Verification framework. During its development, we have applied the policy that processing one snippet should usually not take more than 1 second, and 3 seconds at the most, and so far, the simplification procedure has taken up a considerable share of that budget, but has rarely exceeded it, and thus appears usable for our Live Verification framework, whereas Attempt 1 (section 6.3) did not clear that performance bar, and Attempt 2 (section 6.4) was not powerful enough because it did not support side-condition solving adequately.

# Part II

# Case Studies

# Chapter 7

# Overview

The goal of this part of the dissertation is to support two claims:

- The techniques presented in Part I lead to proofs that can be composed into end-to-end theorems where the intermediate specifications cancel out.
- Systems with end-to-end theorems where the intermediate specifications cancel out are easier to audit because the number of lines of code that need to be audited is lower than it would be in traditional unverified systems or in systems where only some individual components were verified.

To support the first claim, I will present three systems that were built using techniques of Part I and explain the guarantees that their end-to-end theorems prove. The three systems are an internet-of-things lightbulb (chapter 8), a cryptographically authenticated garage door opener (chapter 9), and a RISC-V trap handler that emulates the multiplication instruction in software on processors that do not implement it in hardware (chapter 10).

To support the second claim, I will, in chapter 11, present LOC counts that compare the number of lines of code that one needs to read in order to audit our end-to-end verified systems (i.e., the top-level theorem and all definitions it references) to the number of lines that one would have to read in order to audit the systems if they were unverified (i.e., the number of lines of the implementations).

Interestingly, even though all our previous publications on these case studies include LOC counts, none of them made the comparison described above. The reason might be that in the rush before the submission deadlines, we considered that collecting all the spec code that the end-to-end theorem references would be too much work. But merely counting all the lines to be audited should not be harder than actually auditing them, and since we claim that our systems are easy to audit, one would expect that it should also be easy to count the number of lines. And indeed, we will see

in chapter 11 that this expectation can be confirmed, in the sense that I was able to produce TCB LOC counts with reasonable effort (though with some caveats such as e.g. that I excluded library code).

However, during this line-counting activity, many not-so-clear-cut decisions will have to be made, e.g. which lines exactly do we count, do we count very standard definitions from Coq's standard library such as e.g. addition on $\mathbb{Z}$ or list concatenation, how many lines of code would one have to audit in a system with cryptographic code not synthesized and proven correct by fiat-crypto, and finally, one needs also to take into account that the amount of effort that needs to be spent per line of code to understand and audit it can vary greatly depending on the kind of code.

So, instead of presenting just one number per case study, I will present several different numbers and discuss the caveats of their interpretation.

# Chapter 8

# IoT Lightbulb[1]

This chapter presents the first case study we built on top of the Bedrock2 framework: A bare-metal embedded system that reads network packets and turns on and off a lightbulb based on the network packets' contents. We chose this extremely simple application because we wanted to work in a bottom-up fashion and get a fully proven end-to-end theorem, spanning software as well as hardware, before venturing into more complicated applications, but, as we will see in the further chapters, the Bedrock2 ecosystem can also support more complicated applications. This case study uses omnisemantics (chapter 2), the Bedrock2 compiler (chapter 3), as well as riscv-coq (chapter 4).

Section 8.1 starts with a review of related work, setting the agenda on how we want to do better, section 8.2 gives an overview of the system and the end-to-end theorem we proved about it, followed by section 8.3 describing some implementation choices and a conclusion in section 8.4.

## 8.1 Related Work and Concepts

Formal verification of computer systems has already been done many times since the 1980s, so to motivate our work, we start by reviewing the qualities and shortcomings of prior work, and also use this discussion as an opportunity to define a few important concepts.

### 8.1.1 Verifying Implementations Against a Spec

A little more than ten years ago, formal verification suddenly "broke out" as visibly practical for computer systems, and quite a range of projects have demonstrated its use. The seL4 project [Klein

[1]This chapter of the dissertation contains text copied and adapted from the PLDI'21 paper (and previous, unpublished longer versions of it) I co-authored with Andres Erbsen, Joonwon Choi, Clark Wood, and Adam Chlipala [Erbsen et al., 2021].

et al., 2009] set the pace in the systems community, by proving (in the Isabelle/HOL proof assistant) that a microkernel implemented in C satisfies a logical specification, using a proof structured in layers. This is a great achievement, and yet, someone might ask "but what if the spec is wrong?", and use this argument to dismiss this project and formal verification as a whole. We believe that this is as wrong as claiming that a computer system has been "proven correct": all that formal verification does is to reduce the amount of code whose bugs could invalidate a crucial property from a large, hard to audit implementation, to a short, easier to audit specification, and this alone makes it a worthwhile endeavor.

But there are more questions to be asked: for instance, what if the compiler used to compile the seL4 microkernel has a bug?

### 8.1.2 Tool Verification

One way to address the above question would be to prove, once and for all, that the output of the compiler always satisfies a specification of compiler correctness. For instance, this was done for CompCert [Leroy, 2009a], a C compiler verified in Coq.

However, the seL4 authors wanted to use gcc, so they used an alternative approach to tool verification called *translation validation*: They translate the C code to a graph (and prove this translation correct in the Isabelle/HOL proof assistant), and they translate the binary produced by the gcc compiler to the same graph format (with a translation correctness proof in the HOL4 proof assistant), and then run an SMT solver to check that the two graphs are equivalent. This allows them to safely dismiss the question of compiler correctness, but a few questions remain: What if the SMT solver has a bug? What if the two proof assistants make subtly different assumptions about the semantics of these graphs? And on the application side, what if the processes running on seL4 make assumptions about the system call semantics that subtly differ from the assumptions in seL4's spec? To our knowledge, these questions were not addressed in seL4, and this leads us to the concept of *integration verification*.

### 8.1.3 Integration Verification

When integrating two components, we need to ensure that they make the same assumptions about the interface between them. Formal verification provides a prime solution: it enables us to write down the interface specification in a format that is both human- and machine-readable, and to verify that both components adhere to it. We believe that formal verification research should

not only focus on the verification of individual components, but also much more on this kind of *integration verification*.

Examples of work towards this direction include the Verified Software Toolchain (VST) [Appel et al., 2014], which verifies C programs against logical specifications in Coq, and compiles these programs using CompCert [Leroy, 2009a]. VST and CompCert use the same semantics of C written down in Coq, so the risk of undetected incompatibilities is greatly reduced.[2]

Similarly (but with a different, abstraction-layer-based proof approach), the CertiKOS [Gu et al., 2016] verified operating system implemented in C integrates in a verified way with a fork [Gu et al., 2015] of the CompCert compiler, and there is work [Mansky et al., 2020; Xia et al., 2019] trying to integrate it with VST.

### 8.1.4 Alternatives to Integration Verification

The term *integration verification* is derived from the term *integration testing*, and one might ask why the latter is not sufficient to ensure correct integration of a system's components. The fundamental challenge is to achieve not only full *line coverage* (that is, each line of code has been run during testing), but also to achieve full *input space coverage* (that is, the system behaves correctly in each possible scenario), which is infeasible due to combinatorial explosion (except for very small systems). Thorough testing with strategies geared towards detecting corner cases might reveal many bugs, but especially in a security setting, one must assume that a determined adversary will find the one scenario that tests happen to have missed.

### 8.1.5 Push-Button Integration Verification versus Modularity and Guaranteed Reusability

A promising technique for integration verification that is more automated than the projects surveyed in section 8.1.3 and ours is concolic testing [Godefroid et al., 2005], which combines testing with symbolic analysis of the conditions that drove execution down particular paths. The system can use solvers to invent new test inputs that exercise new control-flow paths, and this symbolic analysis can eventually certify that all relevant paths have been explored in full generality. An example of a verification tool built on top of this style of symbolic execution is Vigor [Zaostrovnykh et al., 2019], which uses the Klee symbolic-execution engine [Cadar et al., 2008]. Vigor verifies

[2]But unfortunately, we are not aware of any work that made use of this specification sharing to prove a combined theorem where the C semantics "cancel out" and the theorem only mentions application logic and assembly semantics.

all the software implementing particular network functions, even including the operating system, thus combining testing-style end-to-end thoroughness with verification-style coverage guarantees.

A similar style of symbolic execution is used in the Hyperkernel project [Nelson et al., 2017] and in the Nickel information-flow-checking tool [Sigurbjarnarson et al., 2018], and improved and generalized into the Serval verification platform [Nelson et al., 2019]. The most interesting design choice in this line of work is to rearchitect software systems to avoid unbounded loops or data structures, so that formal-verification tools can work without extra proof-oriented annotations that programmers would otherwise need to write. Serval performs symbolic evaluation on different assembly languages, so it successfully solves the integration verification problem between the source language and the compiler, because it directly verifies the output of the compiler rather than its input.

While these approaches offer a high degree of automation, they depend on SMT solvers, which are unverified tools with large code bases, and their verification is not *modular*: Whenever we change one part of the system, we need to rerun the whole verification, whereas the projects described in section 8.1.3 have the benefits of *modular verification*: each significant component has a purely local specification, mentioning its own inputs and outputs but not those of other components; and we can tinker with each component implementation and have a *guarantee* that the system will continue to work without needing to revisit the rest of the system, as long as the old specification continues to hold.

An additional interesting point in the design space is the Parfait project [Athalye, 2024], built on top of Knox [Athalye et al., 2022], which proves that a hardware security module (HSM) does not leak more information than its high-level specification already leaks. They use different tools for different layers, thus achieving improved modularity compared to the projects described before, but they still do not specify all intermediate layers: For instance, they only prove that the processor correctly executes the two concrete HSMs they consider, and verification of future systems might reveal bugs in the processor that were provably never triggered by their two HSMs. Moreover, they use several different unverified verification tools, so the soundness of their results also depends on the correctness of these tools, and on correct translation of the specifications into their respective input languages. For instance, CompCert specifies RISC-V assembly semantics in Coq, but they also need assembly semantics in Rosette, so they hand-translated them into Rosette, and one needs to trust or carefully audit both of these semantics in order to be able to trust their result.

### 8.1.6 Height of the Verified Stack

As every layer of a software stack could contain bugs, it is desirable that the verification effort spans a stack height as large as possible.

When it comes to starting the verification as high up as possible, a notable project is Everest [Bhargavan et al., 2017], which develops a TLS stack to span the assurance gap from high-level cryptographic security to low-level software correctness and security. For each layer, they choose the most suitable verification tool, and they do not invest into tool verification or integration verification, so at the end, in order to assure oneself that the claimed results hold, one has to audit a considerably large *trusted code base*.

And when it comes to ending the verification as low as possible in the stack, another project worth mentioning is CompCertMC [Wang et al., 2019], which extends CompCert [Leroy, 2009a] to compile to machine code running on a realistic machine model rather than just to an assembly language with pseudo instructions and a machine model with an unbounded stack. However, we have not yet seen this work being integrated with projects building on top of CompCert, such as VST and CertiKOS.

When thinking about extending the verified stack at the bottom, the interface between software and hardware is both important and subtle. The question is not just "what if the hardware contains bugs?" but crucially also "what if the software and the hardware make different assumptions about how the instructions should behave?"

### 8.1.7 Verified Software-Hardware Integration

We are aware of three projects having addressed the above questions. All of them achieve *integration verification* by connecting all components within one proof assistant, and verify all *tools* (except the proof assistant itself) as well, thus reducing the *trusted code base* required to audit to just their top-most and bottom-most specifications and the proof assistant.

In the late 1980s, the CLI stack [Bevier et al., 1989] connected a Pascal-like language to a 32-bit microprocessor design described in minimalistic register-transfer language. The purpose-built languages were modeled using interpreters and omitted input or output facilities. The processor implementation is described as a loop that executes one instruction per iteration and includes, for example, waiting for responses to memory requests [Hunt, 1989]. The verified software for this stack included arithmetic on large integers and a solver for the mathematical game Nim, and a successor of the processor was fabricated using gate-array technology.

The Verisoft project [Alkassar et al., 2008a], begun in the early 2000s, connects a program correctness framework for programs written in a language they call C0 to a compiler targeting their purpose-built VAMP processor architecture. To our knowledge, no complete physical demonstration system including input and output was ever built with this stack, and we also are not aware of any full-system proof against a short, natural application specification in terms of input and output by the hardware-software stack treated as a black box. The closest we are aware of [Daum et al., 2010] related a correctness proof of a small automotive-control C0 application to the correctness proof of an operating system, plugging into a proved stack including compiler and processor, but there is no discussion of a short full-system theorem, even though it seems to us that this should have been within close reach for them.

In work begun roughly 15 years after the Verisoft project started, the CakeML optimizing compiler [Kiam Tan et al., 2019] was extended with a backend to a new, purpose-built instruction set called Silver [Lööw et al., 2019]. This time the software stack does support input and output, but the complete stack still does not. Instead, external calls for file-system access and standard input/output are compiled into reads and writes of a memory buffer. The stack is run on a field-programmable gate array (FGPA), with a commodity microprocessor connected to the same memory to initialize input and collect output (in contrast to our experiments using a freestanding system). With this setup, several nontrivial programs are executed: examples include word count, sorting, and even compiling a "hello word" program using a cross-compiled copy of CakeML itself (in 4 hours).

Unfortunately, none of the three above projects performs what we call *realistic I/O*: CLI only provides proofs about values written into the main memory, while the other two do communicate to the external world, but not in a standard way, though Verisoft might have gotten close [Alkassar et al., 2007].

Moreover, all of them use their custom-built ISA instead of a *real-world ISA*, raising the question whether their techniques could be adopted in larger, more real systems.

### 8.1.8 Verified Hardware Optimizations

Once we have established that the software and the hardware make the same assumptions about the semantics of the ISA, and we have a proof methodology available at the hardware level, this enables us to replace the processor by an optimized processor, and as long as it satisfies the same ISA specification, this optimization did not increase the size of the *trusted code base*.

The projects described in the previous section make use of this advantage to various degrees: In

the Silver processor of CakeML, the most sophisticated optimization is of modest complexity and deduplicates circuitry for calculating the next program-counter value after different instructions. CLI uses a multicycle processor, while Verisoft's processor includes more interesting optimizations like out-of-order execution.

### 8.1.9 Contributions

In the previous sections, we have introduced various evaluation criteria for verified systems, and we summarize them in Table 8.1. It turns out that none of the existing prior work meets all of them in a way we find satisfactory. Therefore, we set out to build, one more time in the line of work of systems verification, an end-to-end verified systems stack, and we believe to be the first system achieving all the above criteria.

In particular, we build a simple verified systems stack whose *verified height* ranges from a simple high-level application I/O behavior specification all the way down to a hardware description language, using only *verified tools* (except for a small trusted proof kernel) and with *verified integration* at each layer boundary, resulting in an *end-to-end theorem* about the system's behavior which is a mathematical object that can be checked by Coq's trusted proof kernel.

The stack consists of a *modular* ecosystem of proved components, each of which can be modified individually, without having to revisit the rest of the system. It includes *hardware optimizations* such as a pipelined processor, and it is the first project in this space to use a *real-world ISA* (such as RISC-V) and *realistic I/O* (such as memory-mapped I/O) to communicate to the external world.

## 8.2 Overview

As shown in Figure 8.1, our physical system prototype consists of an FPGA that runs a verified RISC-V program on a verified processor and communicates with an unverified network interface card (NIC) over a serial peripheral interface (SPI), as well as with a power switch over two general-purpose I/O (GPIO) pins.

The software uses memory-mapped I/O (MMIO) to access these interfaces, and the correctness of the system is expressed in terms of an *event trace* of MMIO loads and stores.

| Key: | |
|---|---|
| ✓ | met |
| ∼ | partially met |
| ✗ | not met |
| – | not applicable |

| | seL4 [Klein et al., 2009] | VST+CertiKOS [Mansky et al., 2020] | CompCertMC [Wang et al., 2019] | Everest [Bhargavan et al., 2017] | Serval [Nelson et al., 2019] | Vigor [Zaostrovnykh et al., 2019] | CLI stack [Bevier et al., 1989] | Verisoft [Alkassar et al., 2008a] | CakeML+Silver [Lööw et al., 2019] | This case study |
|---|---|---|---|---|---|---|---|---|---|---|
| Applications | | | | | | | | | | |
| OS and/or drivers | | | | | | | | | | |
| Source language | | | | | | | | | | |
| Assembly | | | | | | | | | | |
| Machine code | | | | | | | | | | |
| HDL | | | | | | | | | | |
| Integration verification | ∼ | ∼ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| One proof assistant | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Modularity | ∼ | ✓ | ✓ | ∼ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Standardized ISA | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ∼ | ✗ | ✗ | ✓ |
| HW optimizations | – | – | – | – | – | – | ∼ | ✓ | ✗ | ✓ |
| Realistic I/O | ✓ | ∼ | ✗ | ✗ | ∼ | ✓ | ✗ | ∼ | ✗ | ✓ |

Table 8.1: Comparison of the height of the verified stack and other evaluation criteria. We acknowledge that a single symbol and a few explanatory sentences can never accurately summarize the actual verification achievements, and we refer to the references in the column headers for a more detailed discussion.

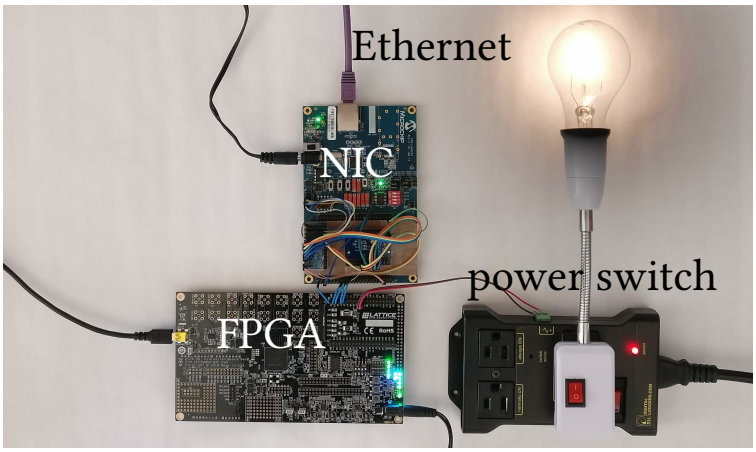



Figure 8.1: System demo

### 8.2.1 The End-to-End Theorem

Our end-to-end theorem looks as follows:[3]

```
Theorem end2end_lightbulb: forall mem0 t state,
  bytes_at (instrencode lightbulb_insts) 0 mem0 →
  Semantics.Behavior (p4mm mem0) state t →
  exists t': list (string * word * word),
    KamiRiscv.KamiLabelSeqR t t' ∧
    prefix_of t' goodHlTrace.
```

It proves that for all initial memories `mem0`, if `mem0` contains the encoded lightbulb instructions at address `0` (the first address the processor will execute), and we observe that the 4-stage pipelined processor `p4mm` connected to memory `mem0` steps to some `state` and produces the I/O trace `t` while doing so, then that trace can be mapped to an I/O trace `t'` consisting of triples of an action label of type `string` and an address and a value of type `word`, and that this trace `t'` is a prefix of a trace satisfying the high-level lightbulb-specific predicate `goodHlTrace`. Only the two action labels `"ld"` and `"st"` are allowed by `goodHlTrace`, and they stand for memory-mapped input (load) and output (store), respectively. We need `prefix_of` because `goodHlTrace` only allows traces observed immediately before the application (re-)enters the main event-handling loop, but the `Semantics.Behavior` hypothesis allows execution up to an arbitrary state, which might be in the middle of the execution

[3] See https://github.com/mit-plv/bedrock2/blob/2223b2a2f7/end2end/src/end2end/End2EndLightbulb.v for its Coq proof.

of the event-handling-loop body. In section 9.1, we will see a more elegant solution that does not require `prefix_of`, but the *always* and *eventually* operators.

Note that the value of `(instrencode lightbulb_insts)` can be computed and printed as a hex-dump inside Coq, which involves running the verified compiler inside Coq, while other compilers verified in Coq such as CompCert [Leroy, 2009a] and Œuf [Mullen et al., 2018] mix propositions and opaque proofs into the compile function, which makes it impossible to run them inside Coq and requires a less well-trusted proof erasure process outside the Coq proof kernel called *extraction* to translate the Coq code to OCaml, and there is no way for them to import back into Coq the fact that running the compiler returns a certain output.

In order to convince oneself that the `bytes_at` assumption holds in the real world, one has to audit the (unverified) Verilog code which creates the memory module and makes sure that its lower addresses are initialized with the machine code bytes output by Coq when computing `(instrencode lightbulb_insts)`.

And in order to convince oneself that `Semantics.Behavior` indeed describes the behavior of the processor running on the FPGA, one has to understand the semantics of the Kami HDL [Choi et al., 2017], and be convinced that the Kami-to-Bluespec transliterator, the Bluespec-to-Verilog compiler, and the synthesis tools work as expected.

And finally, in order to make sense of this theorem, one has to understand its conclusion, which consists of a straightforward mapping `KamiRiscv.KamiLabelSeqR` from the Kami trace format to the simpler trace format, and of the `goodHlTrace` predicate described in section 8.2.2.

We would like to emphasize that compared to other verification projects, only requiring the three items described above to be understood and trusted is very minimal. In particular, no source-language semantics need to be trusted in order to trust this theorem.

### 8.2.2 The Trace Predicate

An I/O trace is a list of triples, where `("ld", addr, value)` means that the system issued an MMIO-load request with address `addr` on the memory bus and got `value` as the reply, whereas `("st", addr, value)` means that the system issued an MMIO-store request with address `addr` and value `value`.

Our specifications stand for sets of legal I/O traces. For readability, we write them in the style of regular expressions (while, beneath the syntactic sugar, retaining the expressive power of higher-order logic), with notation `|||` for union, `+++` for concatenation, `^*` for Kleene closure, and `one` to lift a single I/O operation into a trace predicate (accepting only length-one traces with that operation).

Our top-level spec ("good high-level trace") is defined as

```
Definition goodHlTrace: list OP → Prop :=
  BootSeq +++ ((EX b: bool, Recv b +++ LightbulbCmd b) ||| RecvInvalid ||| PollNone) ^*.
```

Every trace accepted by `goodHlTrace` starts with a series of magic incantations `BootSeq` required to initialize the Ethernet card. After that, `goodHlTrace` requires that the trace only consists of one of three kinds of interactions: A command to turn on or off (as indicated by the Boolean `b`) the lightbulb was received, and an MMIO write to the general-purpose I/O (GPIO) pin connected to the power switch was issued; or an invalid packet was received and no further action was taken; or the network card was polled, but no new data was available. These high-level trace predicates, in turn, are defined as follows:

```
Definition Recv (cmd : bool) (t : list OP) : Prop :=
  exists (packet : list byte),
    lan9250_recv packet t ∧
    lightbulb_packet_rep cmd packet.

Definition LightbulbCmd (cmd : bool) : list OP → Prop := gpio_set 23 cmd.

Definition RecvInvalid : list OP → Prop :=
  (fun t ⇒ exists (packet : list byte),
       lan9250_recv packet t ∧
       ~ (exists (cmd : bool), lightbulb_packet_rep cmd packet)) |||
  (lan9250_recv_packet_too_long) |||
  (any+++spi_timeout).

Definition PollNone : list OP → Prop := lan9250_recv_no_packet.
```

To get an idea of how, after some more unfolding, all definitions eventually boil down to MMIO reads or writes, let us also look at the definition of `gpio_set`:

```
Definition GPIO_DATA_ADDR: word := word.of_Z (0x1001200c).
Definition gpio_set(i: Z)(value: bool): list OP → Prop :=
  EX v: word,
    one ("ld", GPIO_DATA_ADDR, v) +++
    one ("st", GPIO_DATA_ADDR, (
      let cleared := word.and v (word.of_Z (Z.clearbit (2^32-1) i)) in
      word.or cleared (word.slu (word.of_Z (Z.b2z value)) (word.of_Z i)))).
```

It states that setting the `i`-th GPIO pin requires first reading the current value of all 32 pins and then writing the same value again, but with the `i`-th bit cleared using an AND mask and then set to the desired value using an OR mask.

## 8.3 Implementation

### 8.3.1 An Infinite Loop Despite Using a Termination-Sensitive Program Logic

The semantics of the Bedrock2 source language are termination-sensitive. This is a boon and a bane at the same time: on one hand, it enables us to prove that programs terminate, which is an important correctness property. Compare this to Bedrock [Chlipala, 2013], where inserting an infinite loop is used as a backdoor to skip proving correctness in corner cases such as out-of-memory exceptions, and auditing the proofs requires going through the source code and collecting all potentially infinite loops. On the other hand, this design choice also means that we cannot reason about programs that enter infinite loops intentionally, which is the case in our lightbulb example: at the top level, this application consists of an infinite loop polling for network packets and processing one-by-one. However, it seems that all infinite loops we would care about are top-level event loops, so instead of changing the semantics of the Bedrock2 source language, we write a small event-loop compiler that expects input programs of the following form:

```
init();
while (true) loop();
```

where `init` and `loop` are two program-specific functions (like `main` in C programs). It emits RISC-V code that calls `init`, then `loop`, and then jumps back to calling `loop` again, forever.

Based on this, we can state an invariant `ll_inv` on a RISC-V machine that is independent of the semantics of the source language, and holds at the beginning of each loop iteration. The invariant asserts that the I/O trace produced by the machine so far is accepted by the `goodTrace` predicate, as well as several other conditions on the RISC-V machine state such as that the program counter points to the instruction calling `loop`, that the memory contains the right instructions, that the stack pointer points to the beginning of the stack, etc. However, to prove that `ll_inv` holds at the beginning of each loop iteration, we need an invariant holding after each RISC-V instruction, and we can obtain one by asserting that the machine eventually reaches a state satisfying `ll_inv`.

Instantiating this invariant proof with the concrete `goodTrace` specification for the lightbulb, and combining it with the refinement proofs between the RISC-V Coq semantics, the Kami speci-

fication processor and the pipelined Kami processor, we obtain an end-to-end theorem describing the behavior of the IoT lightbulb just in terms of the I/O trace produced by the pipelined Kami processor, independent of all the intermediate interfaces such as the source-language semantics, RISC-V Coq semantics, etc.

### 8.3.2 Interfacing Hardware and Software

The Bedrock2 compiler and the Kami processor are proven correct against very different RISC-V specifications: the compiler uses the software-friendly specification introduced in chapter 4, while the Kami processor is proven against a single-stage processor, where the abstract ISA is instantiated to the one for RISC-V RV32I. In order to combine the compiler and the Kami processor in an end-to-end proof, we need to prove that these two RISC-V specifications are compatible. Since both specifications have a notion of step to handle an instruction, filling the gap between the spec processor and the RISC-V specification can be regarded as proving the correctness of the instantiated ISA in hardware over the RISC-V software specification.

### 8.3.3 Bridging Two Different Styles of Semantics

Because the Bedrock2 compiler and the Kami processor started as two individual projects, not only do they use two different RISC-V specifications, but these two specifications also use different styles of semantics.

The compiler uses what we call *omnisemantics* (see chapter 2), that is, predicates `cstep1` and `csteps` for single and multiple steps of execution, respectively, of type

```
CState → (CState → Prop) → Prop
```

where `CState` is the machine state including the interaction trace produced so far, and `csteps s0 P` means that all (nondeterministic) executions starting in state `s0` do not crash and reach states satisfying `P`.

In Kami, however, there is no concept of crashing: a processor always keeps doing something (or nothing) in the next cycle, and the semantics are defined by predicates `kstep1` and `ksteps` of type

```
KState → KState → Prop
```

where `ksteps s0 s1` means that running the processor from a starting state `s0` can lead to the state `s1` (but potentially also to other states).

Now, if we wanted to do a traditional refinement proof between the Kami spec processor and the compiler's definition of RISC-V semantics, we would have to prove that for each Kami execution, there exists a corresponding execution in the compiler's RISC-V semantics. But this would be impossible, because the compiler's semantics talk about all executions at once, whereas the Kami semantics only give us one execution, which is not sufficient to prove something about all of them.

To close this gap, our proof bridging the Kami semantics and the compiler semantics uses a different structure: it starts by proving that if we have a `cstep1` from the compiler as well as a `kstep1` from Kami, then the state `ks2` in which Kami lands can be simulated in the compiler's RISC-V semantics by some `rs2`:[4]

```
Theorem kstep1_sound: forall ks1 ks2 rs1 P,
  related ks1 rs1 →
  kstep1 ks1 ks2 →
  cstep1 rs1 P →
  related ks2 rs1 ∨
  exists rs2, related ks2 rs2 ∧ P rs2.
```

Note that contrary to what one might expect, this proof assumes *both* a Kami execution and an execution in the compiler's RISC-V semantics.

In order to lift this theorem to multiple steps, we cannot use the compiler's `csteps` predicate, because if we assume a `csteps` and a `ksteps`, they might disagree on the number of steps, and while Kami's predicate could be modified to fix a number of steps, the compiler's cannot, because it talks about all executions at once, and each execution might perform a different number of steps depending on nondeterministic choices. However, by using an invariant on the compiler's view of the machine state, we can state (and prove) that the Kami spec processor only has behaviors which are also behaviors of the compiler's RISC-V semantics as follows:

```
Theorem ksteps_sound: forall (inv: CState → Prop),
  (forall rs, inv rs → cstep1 rs inv) →
  forall ks1 ks2 rs1,
  related ks1 rs1 →
  ksteps ks1 ks2 →
  inv rs1 →
  exists rs2, related ks2 rs2 ∧ inv rs2.
```

The `kstep1_sound` theorem requires a simulation relation (`related`) between the two different

[4]The left-hand side of the disjunction accounts for the fact that the Kami proofs do not guarantee liveness.

RISC-V machine states. Since both the spec processor and the RISC-V specification have the same granularity in terms of the number of semantic steps required to handle an instruction, here the relation can almost be equality, though we must translate between different types used in the two semantics.

### 8.3.4 I/O Throughout the Stack

So far we have mostly emphasized *vertical* modularity. For instance, we could swap the implementation of a layer such as the compiler or the processor for a different implementation, and the specifications at the layer boundaries guarantee that we need not revisit the other layers of the system. However, some dimensions of variation across systems are orthogonal to that decomposition. One natural example is which peripheral devices are available and how to interact with them. Every layer of our stack is parameterized by its relevant choices there, and we think of this parameterization as *horizontal* modularity.

For our lightbulb case study, the processor communicates with the network card and the lightbulb power switch through MMIO.

#### 8.3.4.1 I/O in Bedrock2

In Bedrock2 source code, we use a syntactically distinct construct for MMIO. To keep the language more general, we do not introduce a specific construct just for MMIO but rather a more-general construct we call *external calls*, which appear as special functions callable like any others. The semantics of the source language are parameterized over the behavior of these external calls. The concept of external calls is a strict generalization of MMIO, not a relaxation of semantics: the source-code-level verification condition for an MMIO external call still needs to restrict the address to be within MMIO range.

The Bedrock2 program logic uses a verification-condition generator with the following definition to treat external calls (simplified assuming one argument and one return value):

$$
\begin{aligned}
&\mathsf{vcgen}((x = f_{\mathsf{ext}}(e)), t, m, \ell, Q) := \\
&\quad \exists v.\ \mathsf{expr\_evaluates}(m, \ell, e, v) \wedge \\
&\qquad \mathsf{vcextern}\big(f_{\mathsf{ext}}, t, [v], \\
&\qquad\quad \lambda r.\ Q((f,[v],[r])::t, m, \ell[x := r])\big)
\end{aligned}
$$

The predicate vcextern is a parameter of the semantics – for the lightbulb, we instantiate it with a

characterization of MMIO load and store operations and allowed address ranges in our platform. Like vcgen, vcextern computes a precondition that is sufficient to guarantee that the postcondition $Q$ received as input to vcextern holds after the call. An important difference between vcgen and vcextern is that vcgen models deterministic steps, whereas vcextern needs to account for unknown runtime inputs, which are represented using a universal quantifier in the definition of vcextern. For example, an external call called `"arbitrary"` that requires exactly one nonzero argument $b$ and can return any number less than $b$ would have the specification

$$\begin{aligned}&\mathsf{vcextern}(\texttt{"arbitrary"}, t, \mathit{args}, Q) :=\\ &\quad \exists b.\ \mathit{args} = [b] \wedge 0 < b \wedge (\forall r.\ r < b \Rightarrow Q(r))\end{aligned}$$

Note that when proving the proof obligation returned by vcextern, the programmer has to prove $Q$ (i.e., verify the remainder of the program) for all possible $r$.

#### 8.3.4.2 I/O in the ISA Semantics

Our RISC-V specification is also parameterized over external interactions, implemented by giving special treatment to loads and stores that fall outside the memory owned by the code running on this processor. This special treatment records non-memory loads and stores in the I/O trace of all externally visible behavior of the system, for which the end-to-end theorem will assert that it satisfies the `goodHlTrace` property. In our instantiation of the ISA specification, the memory footprint remains unchanged throughout execution.

The parameter modeling external interactions caused by an `n`-byte non-memory load, `nonmem_load`, takes an address `a`, a machine state `s`, and (in the same style as `vcgen` and `vcextern`) a postcondition `Q`, returning the proof obligation the compiler has to prove to make sure that `Q` holds after executing the load instruction. Here is the instance for MMIO:

```
nonmem_load n a s Q :=
  isMMIOAddr a ∧ isMMIOAligned n a ∧
  ∀ v, Q v (withLogItem (@mmioLoadEvent a n v) s).
```

It requires the compiler to prove that `a` is in the MMIO range, it is `n`-byte aligned, and the desired postcondition holds for a machine state where the address and the unknown read value `v` have been added to the I/O log. The same interface is also powerful enough to model direct memory access (DMA), by recording memory-ownership changes in the I/O trace, but we do not make use of this feature in the lightbulb application.

#### 8.3.4.3 I/O in the Bedrock2 Compiler

Our compiler pipeline is parameterized over an *external-calls compiler*, which defines how to implement each call with machine code. In the lightbulb example, it simply translates `MMIOREAD` and `MMIOWRITE` calls to `lw` and `sw` instructions.

We prove our compiler correct for all possible implementations of external calls in a compositional manner, requiring the same `correct`ness of the external-calls compiler as we are proving about the whole compiler:

```
Lemma compiler_correct: ∀ compile_ext,
  (∀ x f_ext a, correct compile_ext (x = f_ext(a))) →
  (∀ program, correct (compile compile_ext) program).
```

Condition `correct comp p` says that feeding program `p` into the compilation function `comp` produces position-independent code that takes any machine state satisfying the *compiler invariant* to some machine state satisfying the compiler invariant and the postcondition of `p` (assuming no execution of `p` from the given starting state can trigger undefined behavior).

Note that the postcondition has to be translated as well, because a source-level postcondition $P$ takes an I/O trace $t$ and a source-level state as arguments, whereas a target-level postcondition takes a target-level state instead of a source-level state. We do so using a state-representation relation $R$ between source and target states, translating the source-level postcondition $P$ into the target-level postcondition $\lambda t\ s_{tgt}.\ \exists s_{src}.\ R(s_{src}, s_{tgt}) \wedge P(t, s_{src})$. If we wanted to support different trace formats for the source and target languages, we could simply include the trace in the representation relation and translate the source-level postcondition $P$ into $\lambda t_{tgt}\ s_{tgt}.\ \exists t_{src}\ s_{src}.\ R(t_{src}, s_{src}, t_{tgt}, s_{tgt}) \wedge P(t_{src}, s_{src})$.

**Verifying the External-Calls Compiler** The correctness proof of the external-calls compiler (i.e., the proof of the main hypothesis of the last lemma) relies both on the compiler invariant and on the source-level verification condition of external calls (`vcextern`). However, the main compiler as well as `correct` are too general to know anything about the concept of MMIO, but they still need to empower the correctness proof of the external-calls compiler to show that the loads and stores emitted for MMIO do not modify application data or code. Therefore, the compiler invariant includes not only administrative conditions regarding the stack and registers but also an *external invariant* that the code emitted by the external-calls compiler can rely on and which the proof of the main compiler takes as an abstract parameter. In our case study with MMIO only, it is sufficient to use an external invariant that requires MMIO addresses not to overlap with the physical memory,

and `vcextern` requires the application programmer to show that the addresses are indeed within the MMIO range (see section 8.3.4.1).

We must also provide a means to the main compiler to prove that it preserves the abstract external invariant, and we do so by imposing the condition on the abstract external invariant that it is preserved by all ordinary RISC-V instructions the main compiler uses (that is, in particular, excluding `lw` and `sw` outside the physical memory).

We found these details to be a particularly tricky exercise in parameterization and "threading" of invariants through a development. The solution we describe here relies on quantifying over predicates (`vcextern` and the external invariant) and their properties, an example of how use of higher-order logic enables modularity.

#### 8.3.4.4 I/O in Hardware

I/O is encoded in Kami as invoking methods on an unspecified external module, which the semantics tracks in a behavior trace. The processor itself does not distinguish ordinary memory operations from MMIO. When the memory module is attached, it handles the loads and stores to memory addresses but makes designated external method calls for the rest. This factoring appears both in the pipelined processor and in the spec processor, making for an easy correctness proof by modular refinement.

### 8.3.5 Pipelining and Instruction Memory Consistency

The RISC-V instruction-set specification does not require memory accesses for instruction fetching to be sequentially consistent with data loads and stores. Concretely, a stale (cached or pipelined) instruction might execute even though the corresponding memory location was overwritten by a recent instruction. This is essential to enable simple pipelined designs, or even just simple instruction caches (while more sophisticated CPU designs can detect and handle such hazards, this additional implementation complexity is almost always omitted in embedded processors).

The specification we use handles this by tracking a set of executable addresses `XAddrs` throughout the execution. Whenever an instruction is fetched, undefined behavior is triggered if the fetch address is not in `XAddrs`, and after each store, all written addresses are removed from the set of executable addresses. The correctness proof of our compiler includes showing that the program addresses remain executable throughout program execution. This in turn relies on external calls not modifying the set of executable addresses, which in turn only holds if the program is calling the external calls in accordance with their specification, which of course depend on the correctness

of execution of compiled code so far. This is yet another example of how correctness specifications of interfaces between embedded systems' components are intertwined in non-obvious ways that are important for practical performance but are easily lost in academic simplifications (we believe no other verified stack tackles this particular challenge).[5]

## 8.4 Conclusion

We presented another step toward more complete end-to-end mechanized proof of systems combining software and hardware, with small trusted code bases. Since our top-level theorem does not reference any of the intermediate specifications, we can rule out a large class of potential integration bugs.

In order to focus on this kind of integration verification, we chose rather simple designs for the individual components, so the work we presented here cannot yet fully answer the question how our technique would scale if we replaced the individual components by more complex designs. More complex designs would likely lead to more complex intermediate specifications, and since we have not yet encountered any friction points in our specification style, we believe that it is ready for use with more complex designs that need features such as direct memory access or, more generally, external calls that acquire and release logical ownership of memory. On the other hand, concurrent software execution (on multiple cores or in interrupt handlers) would require considerable changes to our current approach.

Compared to past work, we emphasize building a freestanding digital system that uses realistic I/O, an instruction set that is already widely used, and low-level coding patterns representative of embedded systems. New challenges were raised for modularity of both the vertical (layering) and horizontal (parameterization) kinds. We also found that tooling challenges with performant proof automation in Coq dominated our development time, feeding a wishlist of mundane-sounding Coq improvements.

It seems that these limitations must first be overcome to attain feasibility for any integration-verification case study large enough to benefit genuinely from a modular architecture. Still, we were able to complete the last conceptual ingredients in an end-to-end functional-correctness theorem that directly captures the I/O behavior of a very simple untethered embedded system.

[5]Further, the RISC-V instruction `FENCE.I` could be used to resynchronize the instruction and data paths (usually at considerable run-time cost), but like many embedded processors, our processor does not support this instruction – it is treated as a no-op, and the specification is modified accordingly.

# Chapter 9

# The Garage Door: Foundational Integration Verification of a Cryptographic Server[1]

This chapter reports on a case study on a bare-metal software stack that includes functional and imperative languages, four qualitatively different compilers, optimized implementations of elliptic-curve and addition-rotate-XOR cryptography, an Ethernet driver, UDP/IP networking, and utility libraries along with a proof-of-concept application dubbed "the garage door opener".

Our work is unique in achieving a machine-checked integrated correctness proof about *all software in a system* by building on completely different reasoning methods in different subdomains.

Our demonstration system is based on a commercial SiFive FE310 micro-controller that runs RISC-V (RV32IM) code in a bare-metal environment. This FE310 processor does not come with a formal proof of correctness, so in this case study, the verification ends at the RISC-V level, and in order to trust the system, one must trust that the FE310 micro-controller implements RISC-V as specified by riscv-coq (chapter 4). The system consists of a server that listens for UDP packets over Ethernet, responds to a session initiation with an X25519 elliptic-curve Diffie-Hellman key exchange, and accepts a different packet type to complete the handshake and authenticate the user whose authorized public key is specified in the system configuration. After successful authentication, a general-purpose digital output is driven based on the command from the received packet. In the physical demonstrations, this output is connected to a motor controller that opens or closes a toy garage door, which stands in for remotely managed real-world infrastructure such as a power plant or the gates of a dam.

[1]This chapter of the dissertation contains text copied and adapted from the PLDI'24 paper I co-authored with Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Clément Pit-Claudel, and Adam Chlipala [Erbsen et al., 2024].

The specification is simple: successfully authenticated commands should drive the actuator, while other input should be ignored. The required elliptic-curve mathematics are proven against affine-coordinate formulas that fit on the back of a napkin, and the network packet formats are specified by concatenating appropriate lists of bytes in a functional language. The bottom interface is RISC-V machine code. Someone who wishes to understand and audit the *statement* of our system theorem does *not* need to read definitions of the programming languages, APIs and ABIs, and resource accounting we use that ensures that the system will not run out of memory, overwrite code with data, or enter an infinite loop, because our *statement* does not include or reference any definitions related to these concerns. The *proof* of our statement has to deal with all these concerns, but the internal specifications all "cancel out" as we compose the subproofs.

Our implementation builds upon established Coq projects from each domain, reusing both methods and artifacts when practical while extending each project as required for verified integration:

- Underspecification and unconditional requirements in internal specifications are encoded using omnisemantics (chapter 2), and we use simple separation logic as an assertion language throughout the stack, even to specify compilers.
- We use the Bedrock2 compiler (chapter 3) and show that it can ingest hand-written code as well as code generated by other compilers, bigger in size than in the lightbulb case study (chapter 8).
- We use the Bedrock2 program logic [Erbsen et al., 2021] to verify the handwritten Bedrock2 code. We show that it can be applied to a significantly more complex specification, with network input and output, compared to the lightbulb case study. We also add support for reasoning about read-before-write aliasing.
- We adopt the Fiat Cryptography [Erbsen et al., 2019] framework, for generation of fast finite-field arithmetic from templates expressed and proven as higher-order functional programs. We extend it *above* with proofs of elliptic-curve-point representations structures and algorithms, as well as *below* with a new verified backend targeting Bedrock2.
- We adopt the Rupicola [Pit-Claudel et al., 2022] code generator, deriving bare-metal-ready Bedrock2 code from functional programs. We use it to build cryptographic code on top of finite-field arithmetic and also integrate code for IP checksums from past work on Rupicola.
- We experimented with an interactive-proof-driven variant of relational compilation for generating code for algorithmically straightforward routines with memory-access-related reasoning bottlenecks such as ChaCha20.

```
Definition initial_conditions mach :=
  0x20400000 = mach.(getPc) ∧
  [] = mach.(getLog) ∧
  mach.(getNextPc) = word.add mach.(getPc) (word.of_Z 4) ∧
  regs_initialized (getRegs mach) ∧
  (forall a : word32, code_start ml <= a < code_pastend ml → In a (getXAddrs mach)) ∧
  valid_machine mach ∧
  (imem (code_start ml) (code_pastend ml) garagedoor_insns *
   mem_available (heap_start ml) (heap_pastend ml) *
   mem_available (stack_start ml) (stack_pastend ml)) (getMem mach).

Theorem garagedoor_correct : forall mach : RiscvMachine, initial_conditions mach →
  always run1 (eventually run1 (fun mach' ⇒ io_spec mach'.(getLog))) mach.
```

Figure 9.1: Top-level correctness theorem

The code of this case study is available at

https://github.com/mit-plv/fiat-crypto/tree/GarageDoorPLDI24

## 9.1 The End-to-End Theorem

Figure 9.1 shows the top-level correctness theorem. It uses the *always* operator as defined in Figure 2.8a and the *eventually* operator as defined in Figure 2.7b. The predicate transformer `run1` defines how executing one RISC-V instruction affects the `RiscvMachine` state: Given an initial state and a desired postcondition, it returns a `Prop` that says what needs to be proven so that the desired postcondition holds.

Our system theorem covers the execution from the first programmable RISC-V instruction onward, but it relies on specific conditions on the initial state: the program counter must start out pointing to the address where the verified machine code begins, the ghost-state trace must start out empty, the required amount of memory must be available, and so on. In return, our specification guarantees liveness and crash-freedom: `io_spec` will always eventually hold (namely, at the end of each iteration of the top-level loop) without any carve-outs for potentially running out of memory or entering a silent infinite loop instead.

The `eventually` operator allows us to use a specification that only applies after having received an entire network packet and potentially responded to it, but not in the middle of the transaction.

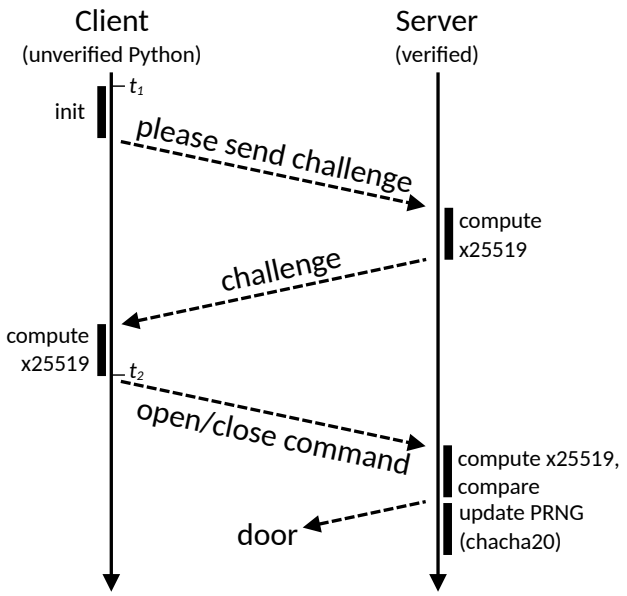



Figure 9.2: Client-server interaction. The bold black bars represent computation.

### 9.1.1 Network Protocol Specification

`io_spec` describes the network protocol and is depicted in Figure 9.2. It states that only traces are accepted that consist of a series of protocol steps, where one step can be any of the following:

- Polling the network card timed out, returned no packet, or returned an invalid packet; or
- A packet was received that asked for a challenge, and a matching reply was sent; or
- A correctly authenticated packet was received that asked to open or close the garage door, a corresponding MMIO request to the garage door was made, and a new seed for generating the next challenge was derived from the current seed.

The contents of the network packets are described using simple functional-program expressions. For example, here is our specification of the server's first packet of a Diffie-Hellman handshake featuring headers of network protocols and a fresh X25519 public key.

Ethernet header
```
mac_remote ++ mac_local ++ be2 ethertype ++
```
IP header
```
let ip_hdr checksum := ih_const ++ be2 ip_length ++
                       ip_idff ++ [ipproto] ++ le_split 2 checksum ++
                       ip_local ++ ip_remote in
ip_hdr (IPChecksum.Spec.ip_checksum (ip_hdr 0)) ++
```
UDP header
```
udp_local ++ udp_remote ++ be2 udp_length ++ be2 0 ++
```
Application data
```
garagedoor_header ++
  x25519_spec x25519_ephemeral_secret Curve25519.M.B
```

The elliptic-curve payload of the packet is described using `le_combine` and `le_split`, which convert between little-endian-ordered byte lists and $\mathbb{Z}$, as well as high-level `Curve25519` definitions.

### 9.1.2 RISC-V Machine Code for Memory-Mapped I/O and Infinite Loops

The lowest-level language considered in this study is machine code for the RISC-V instruction set. We follow the specification and reasoning strategy from chapter 8 but target the RV32IM instruction set of a commercial microcontroller instead of a verified processor implemented on an FPGA.

There are two kinds of operations that are not expressible in the Bedrock2 source language, and are therefore implemented directly as simple RISC-V assembly macros emitted by what we call a mini-compiler, in such a way that the proofs cover all the RISC-V code (as opposed to covering only the part that is expressible in the source language, and relying on "trusted glue code in assembly", like other verification projects often do).

The first problem solved using a mini-compiler is that our use of memory-mapped I/O to talk to devices is not directly expressible in the Bedrock2 source language. Instead, it is modeled by external calls, and the Bedrock2 compiler is parameterized over a mini-compiler for these external calls, which replaces each call by either a load or store assembly instruction. Thus, the proof of the assembly fragment for an external call must centrally establish the relationship between the low-level and high-level I/O-trace events, but it must also show that the MMIO store does not accidentally overwrite compiler data structures.

Moreover, the top-level loop is an infinite loop, but the Bedrock2 source-language semantics only accept terminating programs, so we use another verified assembly macro that takes the relative addresses of an init function and a loop-body function, emitting a program that first calls the init function and then repeatedly calls the loop-body function forever. In this case, the specification of the assembly fragment is parameterized over that of the loop body, and this specification states

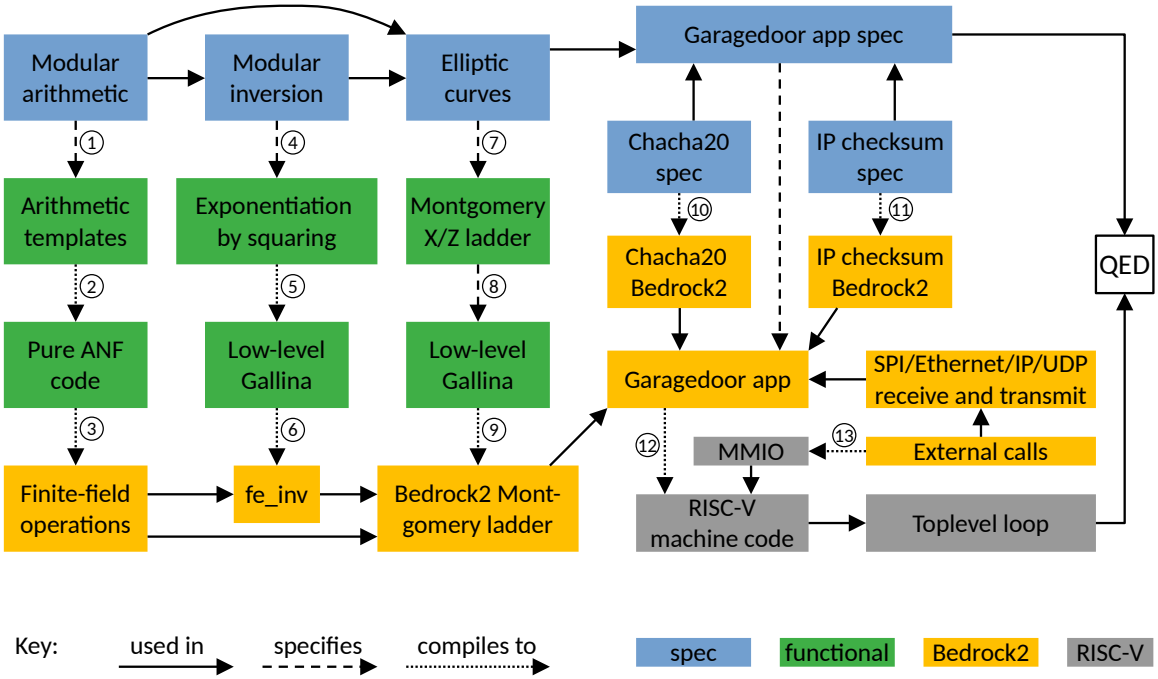



Figure 9.3: Overview of components and specifications

that any loop invariant is always eventually satisfied, leading to the top-level spec in Figure 9.1.

We reuse both mini-compilers from the lightbulb (section 8.3.1 and section 8.3.4.3). Their specifications are considerably longer than their implementations, and the corresponding proofs require correspondingly detailed ad-hoc reasoning, but there are no surprise obstacles or hard-to-bridge abstraction gaps. These proofs are made feasible by the use of language-independent separation-logic assertions and straightforward elementwise relations connecting Bedrock2 and RISC-V-level I/O traces. These techniques enable us to compose an end-to-end theorem that covers ad-hoc assembly code for functionality without language support, as opposed to relying on some unverified assembly glue code or extending the syntax and semantics of the languages.

## 9.2 Different Techniques Combined

This section explains how different techniques were combined to create the implementation and its end-to-end proof by means of a tour through Figure 9.3.

Its left-most column contains the Fiat Cryptography [Erbsen et al., 2019] pipeline (① and ②) that creates highly optimized code for modular arithmetic with constant modulus ($2^{255} - 19$ in the

present case study) in administrative normal form (ANF), i.e. straight-line code where each line of code assigns the result of one arithmetic operation to a fresh variable. The Fiat-Crypto-to-Bedrock2 compiler (③, [Erbsen et al., 2024, Section 3.6]), implemented as a functional program in Gallina, emits Bedrock2 code for it, as well as specifications using omnisemantics and separation logic that state that the outputs in memory correspond to the functional programs specifying the behavior of its input code.

The second column in Figure 9.3 implements modular inversion. Using Euler's theorem, it can be expressed as modular exponentiation, which can be shown (④) to be implemented by a functional square-and-multiply algorithm. Partially evaluating (⑤) this algorithm for the compile-time-known exponent using Coq's simplification tactics leads to functional code (labeled as low-level Gallina because it is close to the desired imperative code) which can be compiled (⑥) to Bedrock2 using the relational-compilation framework Rupicola [Pit-Claudel et al., 2022].

The main operation needed for the Diffie-Hellman key exchange is multiplication of a scalar times an elliptic-curve point, shown in the third column of Figure 9.3. A functional implementation of the Montgomery-ladder algorithm is proven (⑦) to conform to a high-level mathematical specification, and also proven (⑧) to be equivalent to a lower-level functional program that can then be compiled (⑨) to Bedrock2 using Rupicola. Rupicola is also used to generate (⑩ and ⑪) Bedrock2 code for the Chacha20 pseudo random number generator and the CRC32 checksum.

The main garage door app is handwritten in Bedrock2, references all the other, generated Bedrock2 code, and is compiled (⑫) to RISC-V machine code using the Bedrock2 compiler.

The interaction with the network interface card and the garage door actuator is implemented using memory-mapped I/O (MMIO), which is represented as external calls at the Bedrock2 source level, and a mini-compiler (⑬) that plugs into the main Bedrock2 compiler replaces these external calls by RISC-V load and store instructions. Another mini-compiler emits the top-level loop, which first calls an init function and then repeatedly and indefinitely calls a loop body function that handles one garage-door request.

Finally, the statement of the end-to-end QED only references the RISC-V code (and the RISC-V semantics) at the bottom and the high-level garage door spec (which references other high-level specs) at the top, resulting in a concise theorem statement that is easy to audit, as we will see in section 11.2.

To summarize what the combination of techniques described above means for the Bedrock2 compiler, consider Figure 9.4: It shows how the input of the Bedrock2 compiler in this case study consists of handwritten code as well as code generated by Rupicola and the Fiat-Crypto-to-Bedrock2 compiler, whose input in turn was generated by Fiat Cryptography.

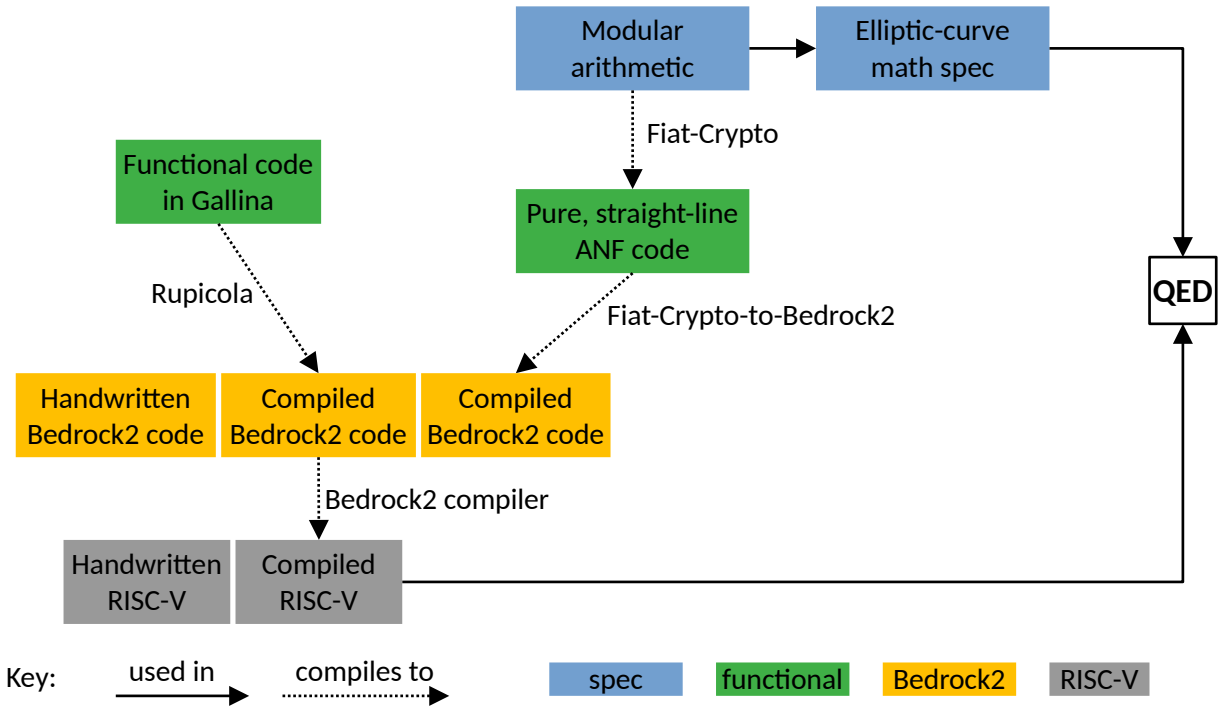



Figure 9.4: The Bedrock2 compiler compiles handwritten code as well as code generated by three higher-level compilers.

| Implementation | Time |
|---|---|
| Ours (compiled with Bedrock2) | 0.47s |
| Ours (compiled with GCC) | 0.12s |
| Ours (substituting BoringSSL) | 0.12s |
| Test client initialization only | 0.10s |

Table 9.1: Client-side performance measurements of different implementations

## 9.3 Evaluation

### 9.3.1 Performance

Instrumenting the server to measure its performance might be possible but a bit tricky. A much easier measure is the time the client takes to execute one open or close command. In Figure 9.2, this measure corresponds to $t_2 - t_1$, so it also includes the running time of the client program and the network latency and excludes the computation that the server performs to check the client's request as well as the computation to update the pseudo random number generator, but it does include computing one x25519 operation, so the measured time can serve as an upper bound for the ballpark of one significant part of the server's computation.

Table 9.1 shows the time it takes to execute the client for different implementations. The first line is for our verified system. In the second line, instead of using the Bedrock2 compiler, the Bedrock2 code was pretty-printed to C and compiled with GCC. As we can see, this leads to a significant performance increase, which is not surprising because no particular effort (beyond a simple register allocator) has been spent on compiler optimizations in the Bedrock2 compiler. On the other hand, swapping the elliptic-curve code generated by Fiat Cryptography with code from BoringSSL (a state-of-the-art crypto library used e.g. in the Chrome browser) leads to no measurable speed improvement, which suggests that the optimizations done in Fiat Cryptography are of similar quality as the ones in BoringSSL. And finally, for comparison, the last line measures how long it takes to start up the Python client and load its crypto library.

### 9.3.2 Effort and Project Size

To get a very rough estimate of the effort that was required to produce this case study, we give LOC counts of new code developed for this case study as well as LOC counts of all code involved.

| Component | Technique | LOC |
|---|---|---|
| Garage door app | Bedrock2 | 1142 |
| Transmit for LAN9250 | Bedrock2 | 134 |
| Bedrock2 library functions | Bedrock2 | 418 |
| ChaCha20 | Rupicola | 2591 |
| Montgomery ladder | Bedrock2 | 374 |
| Montgomery ladder | Rupicola | 1062 |
| Modular inversion | simpl, Rupicola | 467 |
| FC-to-Bedrock2 compiler | Generic Coq | 7110 |
| Invoking FC-to-Bedrock2 compiler | Generic Coq | 804 |
| Total | | 14102 |

(a) New lines of code of this case study

| Repository | LOC |
|---|---|
| Coq stdlib | 85495 |
| bedrock2 | 7402 |
| bedrock2Examples | 2759 |
| compiler | 14911 |
| Coqprime | 5650 |
| coqutil | 11424 |
| Rewriter | 30124 |
| riscv | 6077 |
| Rupicola | 8198 |
| Fiat Crypto | 83861 |
| Total | 255901 |

(b) `Required` lines of code

Table 9.2: New and total lines of code of the project

#### 9.3.2.1 Effort of This Case Study

Table 9.2a lists each component that was newly developed for this case study, which technique it used, and its line count.

#### 9.3.2.2 Size of the Code Base Including Dependencies

The source directory with our code and all dependencies contains over 3500 Coq files that amount to almost 400KLOC, but focusing on that measurement is misleading, because many of these files are from different research projects and not required for our project. To get a more representative count without resorting to manually classifying 3500 Coq files as on-topic or off-topic, we rely on the Coq command `Print Libraries` to list all files that the file containing our top-level theorem transitively depends on, and then we only count the lines of code of these, which leads to the numbers in Table 9.2b.

# Chapter 10

# Softmul: Verifying Software Emulation of an Unsupported Hardware Instruction[1]

Some processors, especially embedded ones, do not implement all instructions in hardware. Instead, if the processor encounters an unimplemented instruction, an unsupported-instruction exception is raised, and an exception handler is run which implements the missing instruction in software. Getting such a system to work correctly is tricky: The exception handler code must not destroy any state of the user program and must use the control and status registers (CSRs) of the processor correctly. Moreover, parts of the handler are typically implemented in assembly, while other parts are implemented in a language like C, and one must make sure that when jumping from the user program into the handler assembly, from the handler assembly into C, back to assembly and finally back to the user program, all the assumptions made by the different pieces of code, hardware, and the compiler are satisfied.

Despite all these tricky details, there is a concise and intuitive way of stating the correctness of such a system: User programs running on a system where some instructions are implemented in software behave the same as if they were running on a system where all instructions are implemented in hardware.

This chapter shows how to formalize and prove such a statement in the Coq proof assistant, for the case of a simple exception handler implementing the multiplication instruction on a RISC-V processor.

[1] This chapter of the dissertation contains text copied and adapted from a paper I co-authored with Thomas Bourgeat and Adam Chlipala [Gruetter et al., 2024a].

## 10.1 Introduction

Assembly language is frequently regarded as the lowest level of software abstraction in software-verification endeavors. However, the ISA (instruction-set architecture) semantics typically employed for software verification present an abstraction of the bare-metal ISA specifications, omitting machine-level aspects of the ISA, like the configuration registers that control the intricate interplay between the hardware's intrinsic capabilities and the meticulously crafted firmware (a piece of software) tasked with maintaining machine configurations and implementing high-privilege handlers in charge of emulating unsupported instructions, as well as managing other forms of low-level exceptions.

For example, in the RISC-V ISA, control and status registers (CSRs) shape the behavior and functionality of the machine. These registers serve as a mechanism for controlling various aspects of the processor's operation, ranging from enabling or disabling specific features to controlling where the machine jumps in case of interrupts and exceptions. These registers and the associated exception handlers exert fundamental control over machine behaviors, so their improper configuration can lead to undefined outcomes.

CSRs coupled with the handlers introduce an intriguing specification, implementation, and verification challenge: while they are essential to determining the machine's behavior, the CSRs are themselves set and manipulated by software, and the handlers are themselves software.

There is a bit of a chicken-and-egg problem: We want to provide a nice and simple ISA abstraction, but to implement this abstraction and prove it correct, we have to write a trap handler and want to compile parts of it with a compiler whose proof already relies on this abstraction that we are supposed to implement, so how can we break the circularity?

One might be tempted simply to augment software-verification efforts with more detailed and faithful ISA specifications. We eschew this approach. The simplified ISA abstractions commonly employed are far more practical and productive compared to their cumbersome and heavier bare-metal counterparts, and the intricate details of configurations and handlers should anyway remain irrelevant to software or compilers higher up the stack.

We endeavor to disentangle the problem by focusing on a *simplified-yet-illustrative* instance: the specification, implementation, and verification of a RISC-V machine with software-implemented multiply instructions.

Through this exploration, we aim to shed light on the interesting challenges posed by CSRs and handlers and pave the way for a more coherent understanding of hardware-software interactions.

We will show that for this simple case we can indeed provide (with proofs!) the desired ab-

stractions, and we can leverage tools that were built on top of those nice abstractions to provide the said abstractions without creating a circular conundrum. Our solution is to prove a helper lemma that ports assembly program-correctness proofs against the nice and simple ISA semantics to proofs against the detailed low-level ISA semantics. The helper lemma requires that the program does not contain any unsupported instruction that would trigger the trap handler, and this assumption gets discharged when we instantiate it with the concrete handler code produced by the compiler. However, there are also parts of the handler whose semantics cannot be expressed using the nice and simple ISA semantics, and we implement these manually in assembly and prove their correctness directly at the assembly level.

Specifically, this chapter makes the following contributions:

- We propose a pleasantly simple specification for a RISC-V system equipped with a software trap handler emulating unsupported instructions: User programs running on a system where some instructions are implemented in software in a trap handler should behave as if they were running on a system with hardware support for these instructions.
- We implement such a trap handler by combining code in a C-like language with handwritten assembly code, and we prove its correctness, in a mechanized and foundational way, down to the binary machine code of the handler, combining symbolic-evaluation proofs at the C level and assembly level with a compiler-correctness proof.

All the code is publicly available at https://github.com/mit-plv/softmul.

## 10.2 Overview

We want to show that a machine without hardware support for multiplication, but correctly configured with an exception handler that implements multiplication in software, behaves like a machine that supports multiplication in hardware. This theorem could then be used to simplify reasoning about programs running on a machine without hardware multiplication, because it saves the burden of reasoning about the trap handler and instead makes it as easy as reasoning about the specification with multiplication in hardware:

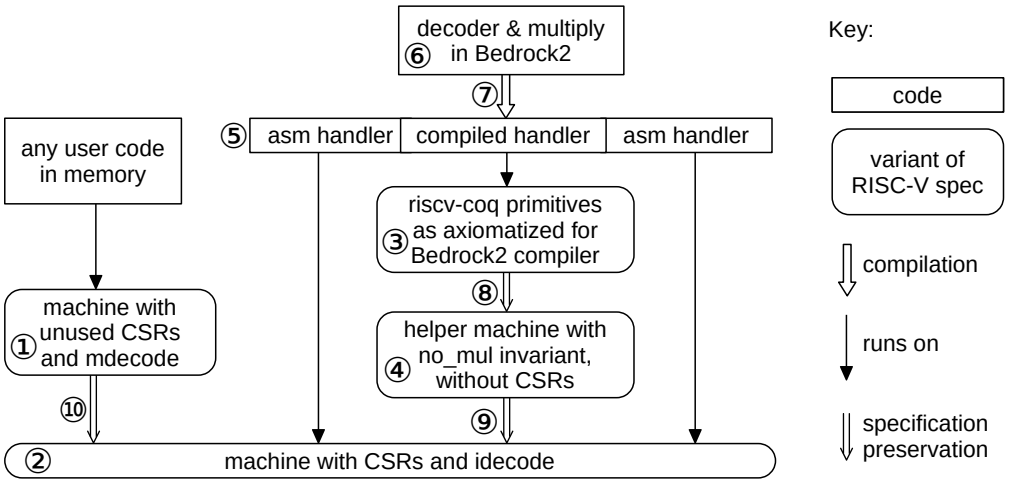



Figure 10.1: Overview diagram. The circled numbers are referenced in the text and do not stand for any meaningful order.

```
match inst with
| Mul rd rs1 rs2 ⇒
    x ← getRegister rs1;
    y ← getRegister rs2;
    setRegister rd (mul x y)
| ...
end
```

We use the RISC-V instruction-set architecture [Waterman and Asanovic, 2019; Waterman et al., 2021], as formalized in riscv-coq (chapter 4). RISC-V splits the instruction set into several extensions, each named with an uppercase letter. The base instruction set that every processor must support is called I, and multiplication, division and modulo operations are in a separate extension called M that small embedded processors may choose not to implement, or to implement in software by catching unsupported-instruction exceptions. In our proof-of-concept case study, we pretend that the M extension only contains one single instruction, namely the multiplication instruction, but we believe that support for the other instructions of the M extension could be added in the same way.

The riscv-coq specification defines a set of roughly a dozen *primitives* such as `getRegister`, `setRegister`, `loadByte`, `storeByte`, and then defines the semantics of each RISC-V instruction in terms of these primitives. The semantics of each primitive is deliberately left unspecified in riscv-coq, so that each application that needs a formal specification of RISC-V can instantiate these

primitives in a suitable domain-specific way.

Figure 10.1 presents an overview of our code (boxes ⑤ and ⑥) and specifications (the remaining boxes). Our theorem uses two instantiations of the riscv-coq specification: One that implements multiplication in hardware (box ①) and one (box ②) that implements it using a trap handler. Note that since the configurability of this specification is first-class, i.e. expressed in Coq itself rather than in some configuration files of the build process, there is no code duplication between the two instantiations.

Parts of the exception handler (box ⑥) are implemented in the Bedrock2 source language (see section 3.2) and compiled (⑦) using the Bedrock2 compiler, but the handler also needs some low-level operations that are not expressible in the Bedrock2 source language and are therefore implemented by-hand in assembly. That is, our handler (box ⑤) starts and ends in handwritten assembly and calls a compiled Bedrock2 function in the middle. Our proof combines a program-logic proof about the Bedrock2 handler function, the compiler-correctness proof, and a proof about the assembly instructions, guaranteeing that all these parts have been put together correctly, and the final statement only mentions RISC-V semantics. All the other interfaces have been canceled out by combining the proofs and thus are not part of the trusted code base anymore.

In addition to the two instantiations of the RISC-V semantics with and without hardware multiplication, our proof (but not the final statement) also uses a third instantiation (box ④) which does not have any CSRs (control and status registers, required by the exception mechanism). This third instantiation fails (with undefined behavior) on all CSR-related instructions. For the compiler, an axiomatization (box ③) of this instantiation was chosen to simplify the proof, because the compiler does not emit any instructions that depend on CSRs.

## 10.3 The Top-Level Theorem Statement

We can state the theorem (arrow ⑩ in Figure 10.1) as follows:

```
Theorem softmul_correct: forall (initialH initialL: MachineState) (post: State → Prop),
  runsTo (mcomp_sat (run1 mdecode)) initialH post →
  R initialH initialL →
  runsTo (mcomp_sat (run1 idecode)) initialL (fun finalL ⇒
            exists finalH, R finalH finalL ∧ post finalH).
```

It is phrased as a *specification-preservation*[2] statement: If a machine with hardware multipli-

[2]It can also be seen as a *small-step omnisemantics forward simulation* as defined in chapter 2.

cation runs from an initial state `initialH` to states satisfying a postcondition `post`, then every machine `initialL` with hardware multiplication, related to `initialH` by `R`, runs to a low-level state `finalL` which, when translated back to a high-level state `finalH`, satisfies the same postcondition.

The theorem uses `run1`, which defines how one single instruction is executed:

```
Definition run1(decoder: Z → Instruction): M unit :=
  pc ← getPC;
  inst ← Machine.loadWord Fetch pc;
  Execute.execute (decoder (LittleEndian.combine 4 inst));;
  endCycleNormal.
```

It is parameterized over the instruction decoder, which is instantiated with `mdecode` (a decoder that supports the multiplication instruction) in the hypothesis and with `idecode` (a decoder that returns `InvalidInstruction` for the multiplication instruction) in the conclusion. The `mcomp_sat` function, of type `M unit → State → (State → Prop) → Prop`, asserts that a monadic program (consisting of primitives used in riscv-coq such as `getRegister`, `setRegister`, `loadByte`, etc.), applied to some initial state, satisfies a postcondition, and `runsTo` lifts it to an arbitrary (but finite) number of steps.[3] The predicate `R` (Figure 10.2) is used to relate a high-level state (i.e. the state of a machine that supports multiplication in hardware) to a low-level state (i.e. the state of a machine that implements multiplication in software using a trap handler), and it also contains all the preconditions on how the low-level machine needs to be configured. That is, `R` asserts that the two states have the same values for the registers and the program counter, and that the memory (modeled as a partial map from 32-bit addresses to bytes) of the low-level machine contains all of the high-level memory, as well as the instructions of the exception handler and some scratch space that the exception handler can use as its stack (which must be available even if the main program has used up all of its stack). To define the addresses at which the handler and the scratch space are located, RISC-V defines some CSRs [Waterman et al., 2021] that our definition of `R` mentions:

- The CSR called `MTVecBase` is used to store the address of the trap handler (we use *direct* mode where all exceptions set the PC to the same address, but RISC-V also has a *vectored* mode where the PC is set to the base address in this register plus an offset corresponding to the cause of the exception).
- The CSR called `MScratch` is a read/write register dedicated for use by machine mode, and we use it to store the address of the *end* of the scratch space (we store the end address instead of the start address because it is used like a stack that grows downwards).

[3] `runsTo` is defined like the omnisemantics *eventually* operator (Figure 2.7b).

```
Definition R(r1 r2: MachineState): Prop :=
  r1.(regs) = r2.(regs) ∧
  r1.(pc) = r2.(pc) ∧
  r1.(nextPc) = r2.(nextPc) ∧
  r1.(csrs) = map.empty ∧
  basic_CSRFields_supported r2 ∧
  regs_initialized r2.(regs) ∧
  exists mtvec_base scratch_end,
    map.get r2.(csrs) CSRField.MTVecBase = Some mtvec_base ∧
    map.get r2.(csrs) CSRField.MScratch = Some scratch_end ∧
    <{ * eq r1.(mem)
       * mem_available (word.of_Z (scratch_end - 256)) (word.of_Z scratch_end)
       * ptsto_bytes (word.of_Z (mtvec_base * 4)) softmul_binary }> r2.(mem).
```

Figure 10.2: The predicate relating high-level states (multiplication implemented in hardware) to low-level states (multiplication implemented in software)

The memory (record fields `r1.(mem)` and `r2.(mem)` in Figure 10.2) is modeled as a finite map from 32-bit words to bytes. In the setup used in this case study, no primitive (nor other operation) changes the domain of that map. If an address outside of the domain of that map is accessed, the memory-access primitives cause undefined behavior, i.e. the `Prop` returned by `mcomp_sat` (and thus also the `Prop` returned by `runsTo`) becomes unprovable. This means that the `runsTo` hypothesis of the top-level theorem assumes a basic form of memory safety of the user program, namely that it does not access memory outside the domain of the memory. The separation-logic formula used in Figure 10.2 ensures that the memory the user program can write to (`r1.(mem)`) is disjoint from the scratch space and the handler code (second and third bullet points, respectively, in the separation-logic formula). To remove this memory-safety assumption, one could prove memory safety for the user program, i.e. that a `runsTo` holds for an arbitrary postcondition (the easiest choice would simply be `λs. True`). In our setting, user code and handler code both run in machine mode, but in more complex systems that feature both user mode and machine mode and also hardware-based memory-protection support (e.g. by segmentation or virtual memory), the requirement to assume or prove this basic memory safety for user programs could be lifted.

## 10.4 The Handler Code

The exception handler code is implemented partially in handwritten assembly and partially in the Bedrock2 source language (see section 3.2) and compiled to bytes by the Bedrock2 compiler.

In order to *prove* the `softmul_correct` theorem, we use the correctness theorem of the Bedrock2 compiler, but note that the *statement* of the `softmul_correct` theorem does not depend on the Bedrock2 language semantics or on anything related to the fact that we used the Bedrock2 compiler, so the auditing burden for someone (who trusts the Coq proof checker) auditing our handler is much smaller, because one does not need to worry about the compiler, its language semantics, and its interaction with the assembly code.

The handwritten assembly of the handler is shown in Figure 10.3. Since we want our software-emulated multiplication to behave as if it were implemented in hardware, we cannot make any assumptions about the remaining space on the user program's stack, nor about whether the stack pointer `sp` contains any meaningful value at all. Therefore, we reserve a separate scratch space in memory just for our handler, and we require that the CSR `MScratch` contains the address of that scratch space.

As its first action (in `handler_init`), the handler has to store all 32 registers of the user process by which it was triggered. It may only use registers that it has already saved, because otherwise it would destroy state of the user program. We therefore resort to tricks such as temporarily storing the user stack pointer in the `MScratch` CSR and then temporarily storing it in the return-address register. Such tricks are easy to get wrong (and we did; see section 10.8.2).

After `handler_init`, the registers 3 to 31 are saved to the scratch space as well, and then the Bedrock2-generated part is called by passing it the value of the CSR register `MTVal`, which contains the invalid instruction that caused the exception, and a pointer to the scratch space in which we saved the registers.

The Bedrock2 code (Figure 10.4) is written directly in Coq using the custom-notations feature, a C-like syntax, and operator precedence as suggested by whitespace. It extracts the three 5-bit fields of the instruction that indicate the two source registers (operands of the multiplication operation) and the destination register, respectively, and then calls another Bedrock2 function `rpmul` that implements multiplication in terms of addition, storing the result back into the scratch space. The `rpmul` function iterates over the bits of the second operand while repeatedly doubling the first operand, a technique sometimes called "Russian peasant multiplication." Both `softmul` and `rpmul` are verified using the Bedrock2 program logic. The spec of the former is given in Figure 10.5.

Its pre- and postcondition are expressed in terms of an (unused) I/O trace `t` and the memory `m`, for which we assert a list of two separation-logic clauses (a word array corresponding to the scratch space containing the register values, and a generic frame `R` for the rest of the memory).

After the Bedrock2 part, the handwritten snippet `inc_mepc` runs. It increases the CSR called `MEPC`, which stores the address of the instruction that caused the exception. This increment is

```
Definition handler_init := [[
  Csrrw sp sp MScratch;  (* swap sp and MScratch CSR *)
  Sw sp zero (-128);     (* save the 0 register (for uniformity) *)
  Sw sp ra (-124);       (* save ra *)
  Csrr ra MScratch;      (* use ra as a temporary register... *)
  Sw sp ra (-120);       (* ... to save the original sp *)
  Csrw sp MScratch;      (* restore the original value of MScratch *)
  Addi sp sp (-128)      (* remainder of code will be relative to updated sp *)
]].

Definition call_mul := [[
  Csrr a0 MTVal;  (* argument 0: value of invalid instruction *)
  Addi a1 sp 0;   (* argument 1: pointer to memory with register values before trap *)
  Jal ra (Z.of_nat (1 + List.length inc_mepc + 29 + List.length handler_final) * 4)
]].

Definition inc_mepc := [[
  Csrr t1 MEPC;
  Addi t1 t1 4;
  Csrw t1 MEPC
]].

Definition handler_final := [[
  Lw ra sp 4;
  Lw sp sp 8; (* Bug: used to be `Csrr sp MScratch`, which is wrong if Mul sets sp *)
  Mret
]].

Definition asm_handler_insts := handler_init ++ save_regs3to31 ++
  call_mul ++ inc_mepc ++ restore_regs3to31 ++ handler_final.
```

Figure 10.3: Assembly part of trap handler (embedded in Coq)

```
Definition softmul := func! (inst, a_regs) {
  a = a_regs + (inst>>15 & 31)<<2;
  b = a_regs + (inst>>20 & 31)<<2;
  d = a_regs + (inst>>07 & 31)<<2;
  unpack! c = rpmul(load(a), load(b));
  store(d, c)
}.

Definition rpmul := func! (x, e) ~> ret {
  ret = $0;
  while (e) {
    if (e & $1) { ret = ret + x };
    e = e >> $1;
    x = x + x
  }
}.
```

Figure 10.4: Bedrock2 part of trap handler (using custom Coq notations to make it resemble C)

```
Instance spec_of_softmul : spec_of "softmul" :=
  fnspec! "softmul" inst a_regs / rd rs1 rs2 regvals R,
  { requires t m :=
      mdecode (word.unsigned inst) = MInstruction (Mul rd rs1 rs2) ∧
      List.length regvals = 32 ∧
      seps [a_regs |→ word_array regvals; R] m;
    ensures t' m' := t = t' ∧
      seps [a_regs |→ word_array (List.upd regvals (Z.to_nat rd) (word.mul
               (List.nth (Z.to_nat rs1) regvals default)
               (List.nth (Z.to_nat rs2) regvals default))); R] m' }.
```

Figure 10.5: Specification of softmul function

needed because upon returning from the trap handler (by the `Mret` instruction), execution will jump to `MEPC`, so we have to set it to one instruction (i.e., 4 bytes) past the multiplication instruction.

And finally, in `restore_regs3to31` and `handler_final`, the values of the user program's registers are restored.

## 10.5 Combining the Program Logic Proofs and Compiler Correctness Proof

By combining the program-logic proofs about the two Bedrock2 functions with the compiler-correctness theorem, we can prove that if we run the compiler within Coq to obtain a list of instructions `mul_insts`, these instructions satisfy the specification shown in Figure 10.6, a verbose but unsurprising specification, laying out calling-convention details.

Lines 5 to 6 specify in which registers the arguments need to be placed, and line 14 requires that at address `a_regs`, there is an array of 32 words that store the values of the registers of the user program. Lines 18 to 20 state that after running `mul_insts`, the array at address `a_regs` storing the registers is updated at its `rd`'th index with the result of multiplying its `rs1`-th and `rs2`-th elements, and line 23 states that the new registers of the processor (not the ones saved in memory) only differ from the original registers on the callee-saved registers.

Note that the conclusion on line 27 refers to the same machine as the conclusion of the top-level theorem in section 10.3, namely the one described by `(mcomp_sat (run1 idecode))`, or box ② in Figure 10.1. However, to get there, two more proof steps (⑧ and ⑨) are needed: In order to keep the Bedrock2 compiler (somewhat) general, it was not proven against a specific instantiation of the riscv-coq semantics but against an axiomatization (box ③) of the primitives used in riscv-coq such as `getRegister`, `setRegister`, `loadByte`, etc. However, to keep the Bedrock2 compiler proof manageable, the RISC-V machine-state representation appearing in that axiomatization was hardcoded to a record type without CSRs (because compiler-emitted code never touches CSRs).

An additional problem requiring some proof effort to show compatibility is that the compiler correctness proof assumes a machine with hardware support for multiplication, but we want to run its code on one without. By inspecting the code that it generated, we can see that it did not output any multiplication instructions, but if it did, this would lead to a serious bug: If during the execution of the trap handler, a multiplication instruction were encountered, the trap handler would be recursively invoked again, infinitely many times.

We solve these two problems by introducing an intermediate helper machine (box ④) that uses

```
Lemma mul_correct: forall initial a_regs regvals invalidIInst R (post: State → Prop)
                           ret_addr stack_start stack_pastend rd rs1 rs2,
    word.unsigned initial.(pc) mod 4 = 0 →
    initial.(nextPc) = word.add initial.(pc) (word.of_Z 4) →
    map.get initial.(regs) RegisterNames.a0 = Some invalidIInst →
    map.get initial.(regs) RegisterNames.a1 = Some a_regs →
    map.get initial.(regs) RegisterNames.ra = Some ret_addr →
    map.get initial.(regs) RegisterNames.sp = Some stack_pastend →
    word.unsigned ret_addr mod 4 = 0 →
    word.unsigned (word.sub stack_pastend stack_start) mod 4 = 0 →
    regs_initialized initial.(regs) →
    mdecode (word.unsigned invalidIInst) = MInstruction (Mul rd rs1 rs2) →
    128 <= word.unsigned (word.sub stack_pastend stack_start) →
    seps [a_regs |→ with_len 32 word_array regvals;
          initial.(pc) |→ program idecode mul_insts;
          mem_available stack_start stack_pastend; R] initial.(MinimalCSRs.mem) ∧
    (forall newMem newRegs,
      seps [a_regs |→ with_len 32 word_array (List.upd regvals (Z.to_nat rd) (word.mul
                 (List.nth (Z.to_nat rs1) regvals default)
                 (List.nth (Z.to_nat rs2) regvals default)));
            initial.(pc) |→ program idecode mul_insts;
            mem_available stack_start stack_pastend; R] newMem →
      map.only_differ initial.(regs) reg_class.caller_saved newRegs →
      regs_initialized newRegs →
      post { initial with pc := ret_addr; nextPc := word.add ret_addr (word.of_Z 4);
             MinimalCSRs.mem := newMem; regs := newRegs }) →
    runsTo (mcomp_sat (run1 idecode)) initial post.
```

Figure 10.6: The correctness lemma of the compiler-generated part of the handler

the same state representation (without CSRs) as the compiler, and we prove an invariant `no_mul` saying that the memory region marked as executable (which only includes the compiled handler code in that instance) contains no multiplication instructions.

## 10.6 Correctness Proof of the Assembly Part

The assembly part of the handler is proven correct by induction over the `runsTo` hypothesis of `softmul_correct`. If the machine with hardware multiplication executes any instruction besides multiplication, we just need to show that after executing the same instruction on the machine with software multiplication, the `R` judgment is preserved, but we can do that once-and-for-all by inspecting each *primitive* of the riscv-coq spec (`getRegister`, `setRegister`, `loadByte`, etc.), instead of analyzing the much larger number of RISC-V *instructions*. The interesting case is when the machine with hardware multiplication encounters a multiplication instruction, and we have to show that

```
Definition raiseExceptionWithInfo{A: Type}(isInterrupt exceptionCode info: t): M A :=
  pc ← getPC;
  (* hardcoded simplification: we only support machine mode and no interrupts *)
  addr ← getCSRField MTVecBase;
  setCSRField MTVal (regToZ_unsigned info);;
  (* these two need to be set just so that Mret will succeed at restoring them *)
  setCSRField MPP (encodePrivMode Machine);;
  setCSRField MPIE 0;;
  setCSRField MEPC (regToZ_unsigned pc);;
  setCSRField MCauseCode (regToZ_unsigned exceptionCode);;
  setPC (ZToReg (addr * 4));;
  @endCycleEarly M t MM MW MP A.
```

Figure 10.7: Specification (in riscv-coq) of what hardware does in case of an exception

the machine with software multiplication steps to a related state. We do so by first symbolically executing the specification of what the *hardware* does in case of an exception (Figure 10.7), which boils down to setting some CSR fields and then setting the PC to the exception-handler address found in the `MTVecBase` CSR.

After that, we symbolically execute the handwritten assembly instructions, using Coq's proof context to keep track of all the facts that we know about the current state of the machine. For each assembly instruction, we encounter its specification in terms of the primitives of riscv-coq, and for each primitive, we have a helper lemma that updates our symbolic state. At the point where we reach the call to the Bedrock2-generated code, we apply the correctness lemma for the compiled trap handler. After that call, we step through more handwritten assembly instructions that restore the registers and then call the `Mret` instruction that jumps back to one instruction past the multiplication instruction that caused the exception. At that point, we need to prove that the symbolic state accumulated in the Coq proof context implies that the two machines are still related by `R`, which only works if there are no bugs in the handler code.

## 10.7 What If ...

To explain our specification from a different angle, we list a few potential bugs that an implementer could introduce, and we show how they make our specification unprovable. Note that these are not bugs that actually occurred in our own implementation. For those, we refer to section 10.8.2. To present each potential bug, we ask: What if ...

- ...the compiler used to compile the handler emitted a multiplication instruction, which would cause the handler to trigger itself recursively infinitely many times? When proving correctness of the handwritten assembly (section 10.6), when we get to the jump instruction that calls the code emitted by the Bedrock2 compiler, we need to apply the compiler-correctness theorem (instantiated with the Bedrock2 part of our handler), but that theorem talks about execution on a machine with multiplication support, whereas the theorem we are about to prove is about execution on a machine without multiplication support. To make the proof work, we need to introduce box ④ and steps ⑧ and ⑨ in Figure 10.1 as explained in section 10.5, which at some point requires us to go through the concrete list of instructions emitted by the compiler and to check that none of them is a multiplication instruction.
- ...the handler runs at a time when no stack exists or the stack does not have enough remaining space? The output of the Bedrock2 compiler contains a number that indicates the amount of stack space that the compiled code needs, and one hypothesis of the compiler-correctness theorem is that at least that much space is available below the current stack pointer. In order to make sure this hypothesis holds, our trap handler uses a separate reserved scratch pad in memory as its stack, and when the correctness theorem for the handwritten assembly applies the instantiated compiler-correctness theorem `mul_correct`, it has to prove that there are at least 128 bytes of space remaining in the scratch pad, as mandated by the hypothesis on line 13 in Figure 10.6.
- ...the assembly that calls compiled Bedrock2 code makes wrong assumptions about the calling conventions of the compiler, e.g. which registers are used to pass arguments, or whether they are passed on the stack, in which direction the stack grows, or which registers are caller-saved? All these conventions are also captured in the intermediate lemma `mul_correct` in Figure 10.6.
- ...the handler forgot to increase MEPC, the CSR storing the address to which the machine jumps when we return from the exception handler, which would cause the faulting multiplication instruction to be run again and trigger the handler again? At the end of the handler correctness proof, this bug would lead to a mismatch between the state of the machine with multiplication support (whose program counter gets advanced past the multiplication instruction) and the state of the machine without multiplication support (whose program counter would still point to the multiplication instruction).
- ...we ran a user program using compressed instructions (2-byte instructions) on our system? The riscv-coq specification only supports the uncompressed instruction format, where all instructions are 4 bytes long. There is no single location where the spec explicitly says

"compressed instructions are *not* supported" – it requires an attentive reader who notices that the whole spec never mentions compressed instructions. In this scenario, our trap handler would fail to decode the unsupported instruction, and arbitrary behavior would occur. If riscv-coq did support compressed instructions, and our handler correctly decoded them, that would still require it to decide correctly whether to increase the MEPC by 2 or 4, and like in the previous point, one would notice the mismatch during the proof.

## 10.8 Evaluation

We attempt to answer the following evaluation questions (and dedicate one subsection to each of them):

1. Does our verified trap handler run on a RISC-V system implemented by a third party?
2. Did our implementation contain bugs that our verification caught?
3. Did our implementation contain bugs that our verification failed to catch?
4. Was the effort required for verification lower than the effort for debugging would have been?

### 10.8.1 Running Our Handler

To validate that our verified handler actually runs on a system not implemented by ourselves, we first looked for small embedded RISC-V processors without multiplication support but could not find any product with enough documentation in English to make us want to try it out. Instead, we chose to test our code in the Spike[4] RISC-V ISA simulator, which offers fine-grained control over which RISC-V extensions are enabled.

We want to test that our handler behaves as expected on a system that runs a simple C program with multiplications, compiled by a third-party compiler. We wrote a simple program which computes the factorial of a hardcoded number and saves the result as well as a "done" flag to memory. We compiled it using the GNU RISC-V toolchain.

Our top-level theorem applies to a list of bytes called `softmul_binary` (mentioned in Figure 10.2 in the definition of the relation `R`), representing a piece of position-independent RISC-V machine code. However, Spike expects as input an ELF file. We relied on the GNU RISC-V toolchain to transform our binary into an ELF file, using a custom 25-line linker script.

For our theorem to be applicable, the conditions that the relation `R` (Figure 10.2) imposes on `r2` (the machine without support for multiplication) must hold on our Spike machine. The first six

[4] https://github.com/riscv-software-src/riscv-isa-sim

conditions above the **exists** are related to the formalization and do not require any special setup action. The two lines below the **exists** require that the `MTVecBase` and `MScratch` CSRs have suitable values, which we ensure by running an assembly script at the beginning that initializes these two CSRs with addresses defined in our linker script. The last three lines are a bullet-point separation-logic clause list describing the memory, saying that it must contain all of the specification machine's memory `r1.(mem)`, as well as 256 bytes of scratch memory at the address in the `MScratch` CSR and the `softmul_binary` at the address in the `MTVecBase` CSR. Our linker script, together with the memory-layout command-line argument we pass to Spike, ensures that these conditions hold.

Spike comes with its own small language of debugger commands, and we used it to run the system until the done flag in memory is 1, then print the value of the memory at the address where we expect the result, and we also print the value of the CSR `minstret`, the number of retired instructions, to see how many instructions were executed.

No matter whether we invoked Spike with or without multiplication enabled, we observed the same result for `factorial(5)`, namely 120. With multiplication enabled, the number of instructions was 87; and with multiplication disabled, the number of instructions increased to 787, which shows that our handler indeed ran. As an additional sanity check, we also confirmed that it stops working if we set the `MTVecBase` CSR to a different value.

Therefore, at least for this one simple example, we can answer question 1 with 'yes.'

### 10.8.2 Bugs Caught During Verification

At the end of the proof that steps through the handwritten handler assembly, we need to prove that the symbolic state accumulated in the Coq proof context implies that the two machines are still related by `R`, which only works if there are no bugs in the handler code (see end of section 10.6). At that point, we found two interesting bugs. The first one was that we forgot to reset the `MScratch` CSR, so one invocation of the exception handler works fine, but the next one will use a wrong address for its scratch space. The second bug was the corner case where the multiplication instruction stores its result into the stack pointer. In that case, we must not override the stack pointer with the original stack pointer that we swapped into the `MScratch` register at the beginning of the handler.

We also found two more obvious bugs related to when to set the stack pointer and what stack-pointer offsets to use.

So we can answer question 2 with 'yes.'

### 10.8.3 Bugs Encountered While Trying to Run It

We split the development of our experiment into two phases: First, we set up the linker script, with the trap handler already in place, but inactive, because we enabled the M extension. Once this experiment produced the expected output, we deactivated the M extension, so that our handler would run.

Getting phase 1 to work required some debugging. The most difficult part was to understand how to pass the linker-script-defined address of the heap memory to the C program, and it required reading the relevant page[5] of the GNU Linker's manual, which starts by saying that "accessing a linker script defined variable from source code is not intuitive," and further down explains that "when you are using a linker script defined symbol in source code you should always take the address of the symbol, and never attempt to use its value".

None of the code involved in phase 1 was verified, so it is not surprising that debugging was required. And to our delight, in phase 2, as soon as we disabled the M extension, our verified trap handler worked on the first try, and no debugging was needed at all.

So, to answer question 3, there were bugs in the unverified part, but not in the verified part.

In the future, it would be interesting also to verify ELF file generation, which we believe could have prevented the above bug.

### 10.8.4 Effort

For lack of better measures, we resort to lines-of-code counts as a very approximate measure of effort. Table 10.1 lists the counts of the different components.

It suggests that to produce 76 lines of verified code, a total of 3331 lines of code was necessary, which is more than a 40× blowup. This ratio looks not very appealing, but it still seems fair to say that for tricky code, large proofs are sometimes needed. We also have some (potentially alleviating) remarks for each row of the table:

- The RISC-V helper instance is not referenced by the top-level theorem statement but acts as a bridge between the RISC-V spec used by the Bedrock2 compiler (whose state does not contain any CSRs) and the one used in the top-level theorem (whose state does have CSRs). The helper instance maintains the invariant that no executable instructions are from the M extension, which is important during the execution of the trap handler, because if the trap handler contained a multiplication instruction, it would be invoked recursively over and

[5] https://sourceware.org/binutils/docs/ld/Source-Code-Reference.html

over again. The helper instance and its accompanying lemmas are mostly copied from the one used in the compiler, and careful refactoring to share the code with the compiler could considerably reduce this count, which also means that these lines were low-effort to produce.

- To verify multiplication and a simple instruction decoder in Bedrock2, we used the original Bedrock2 program logic [Erbsen et al., 2021], which only automates the application of weakest-precondition rules but does not provide any automation for side condition solving. Using a framework that provides more automation would have reduced this proof size.
- A large chunk of the proof lines (1454) is in the correctness proof of the trap-handler parts written in assembly. The reason for this verbosity might be that, to our knowledge, this project is the first within the Bedrock2 ecosystem to verify more than two or three lines of assembly at a time, so there was no assembly-specific framework available. About two thirds of the proof code could probably be factored out into a framework that would be reusable for other assembly programs as well. We also did not spend too much time on side-condition automation, which could further reduce the number of proof lines. We conjecture that in a more mature assembly-verification framework, the assembly part of the trap-handler proof might be as short as maybe 100 lines of code. Moreover, the code-to-proof ratio also looks bad because we count the number of lines of Coq code rather than the number of assembly instructions, which matters for `save_regs3to31` and `restore_regs3to31`: Each of these is just a two-line functional program but expands to 29 assembly instructions.
- The compiler compat & invocation code deals with the different RISC-V instances and decoders and also applies the Bedrock2 compiler's correctness theorem for the instruction decoder and multiplier implemented in Bedrock2. It consists of important but not particularly interesting bookkeeping that quickly adds up to many lines of proof.
- Finally, the top-level theorem puts everything together. It requires some helper lemmas that could probably be generalized and moved to a library, but the fact that these lemmas were not already present in any library used in the Bedrock2 ecosystem seems fairly representative of the general verification experience, so it seems fair to count these lines.

Finding the bugs described in section 10.8.2 through debugging (especially the first two) might have been quite hard but would probably still not have taken as long as our verification effort took, so the answer for question 4 is probably a 'no.'

However, we can imagine a promising world where the proof burden becomes lower than the debugging burden and verification becomes a part of most systems developers' toolboxes.

| | impl | spec | proof | total |
|---|---|---|---|---|
| RISC-V helper instance | 0 | 101 | 309 | 410 |
| Multiplication in Bedrock2 | 8 | 5 | 83 | 96 |
| Instruction decoder in Bedrock2 | 7 | 27 | 80 | 114 |
| Trap handler in assembly | 36 | 28 | 1454 | 1518 |
| Compiler compat & invocation | 14 | 47 | 716 | 777 |
| Top-level theorem | 11 | 18 | 147 | 176 |
| Excluded (imports & comments) | | | | 240 |
| Total | 76 | 226 | 2789 | 3331 |

Table 10.1: Lines-of-code counts, excluding the dependencies (coqutil, riscv-coq, Bedrock2, and the Bedrock2 compiler)

## 10.9 Related Work

A number of projects have attempted to verify the interaction between (some or all of) C code, its compilation, handwritten assembly code, and trap handlers.

In the context of the Verisoft project, Alkassar et al. [2008b] verified a virtual-memory system that can swap out virtual memory pages onto disk. If an address is accessed that currently is on disk, a page fault is triggered, and a verified page-fault handler runs. Their correctness statement says that a physical machine with the page-fault handler can simulate a virtual machine (by which they mean a machine that provides to a user process a linear memory covering the whole address space). Their handler is implemented in C0 (a subset of C) with some inline assembly, which is modeled as external calls that modify additional state that cannot be modified directly from C0. That is, they call assembly from C, whereas we chose the opposite direction, calling C (or the C-like language Bedrock2, in our case) from assembly. In their project, saving and restoring of registers before and after the handler are not implemented in assembly and verified like we do but are instead part of the semantics of the physical machine.

BabyVMM [Vaynberg and Shao, 2012] proves correctness of a simple virtual memory manager by showing that for all kernel implementations, linking the kernel with the virtual memory manager and running it on a machine with only physical memory ("hardware model" HW) behaves like running the kernel on a machine with an address space whose lower part is physical memory and whose upper part is virtual memory ("address space model" AS). It is implemented in a C-like language, and no compiler nor assembly code appears in the formalization. Instead, the theorem

is stated in terms of C semantics. It also does not mention any page-fault handlers.

The verified microkernel seL4 [Heiser, 2020] is implemented in C, but some small parts are handwritten assembly and are not verified [Klein et al., 2014, sections 4.4 and 4.8]. Contrary to our approach of using a verified compiler, they apply translation validation to the binary generated by GCC and certify using SMT solvers that it behaves like the C program.

CertiKOS [Gu et al., 2014, 2016; Chen et al., 2018] is a verified OS kernel. By means of certified abstraction layers, it fully captures the behavior of each component in a deep specification, so that from the outside, it does not matter whether the component is implemented in C or in assembly, thus achieving interoperability at the proof level between C and assembly. Its correctness is expressed as a contextual refinement, based on CompCert's [Leroy, 2009a] notion of a backward simulation, extended with a universal quantification over all possible surrounding programs (contexts): It states that for all assembly programs, all behaviors of that assembly program when linked with the low-level kernel can be simulated by the same program when linked with the high-level kernel specification. It relies on a notion of linking and uses CompCert's formalization of assembly, which is still fairly high-level compared to binary machine code, e.g. jumps use labels instead of offsets or addresses, and there are instructions that allocate and free a stack frame that do not correspond to any machine instructions. CompCert's assembly (which is used to model CertiKOS's lowest layer) also does not model CSRs, whereas riscv-coq, on which our project is based, does, so to model trap handlers at our level of detail, the assembly (or machine) model would have to be extended.

CompCertELF [Wang et al., 2020], a different project by the same group, extends CompCert to also cover machine-code generation and uses a more realistic memory model, without the stack-frame allocation/freeing instructions mentioned above. As far as we know, CompCertELF has not (yet) been integrated with CertiKOS and is not publicly available. If it were, and if we managed to make CompCertELF compatible with our project, it could have helped to prevent the bug (section 10.8.3) we encountered in our unverified usage of the GNU linker to turn our plain binary into an ELF file.

Goel et al. [2020] verify a subset of the instructions of an x86 processor which decodes x86 instructions and translates them into micro-operations before executing them. For the more complex instructions, the generated micro-operations contain a trap that causes a jump to microcode stored in a ROM. Similarly to our theorem, they prove that this processor behaves as if there were no micro-operations, traps or microcode, and instructions were executed according to a high-level x86 specification.

The CakeML compiler [Kiam Tan et al., 2019] targets multiple ISAs, and some instructions (e.g.

division) are not supported by all of them, so the compiler has to implement some unsupported instructions in software, but contrary to our work, the necessary in-software implementation is emitted directly by the compiler, and no trap handler comes into play.

## 10.10 Conclusion and Future Work

We have shown a pleasantly simple way of specifying the correctness of a trap handler that emulates unsupported instructions in software, and we proved that our implementation of such a trap handler combining handwritten assembly and compiler-generated code satisfies this specification by combining symbolic-evaluation proofs about assembly and Bedrock2 programs with the correctness proof of the Bedrock2 compiler, as well as by proving that the output of the Bedrock2 compiler, which assumes a machine without CSRs and with hardware support for multiplication, also runs correctly on a machine with CSRs but without hardware support for multiplication.

This style of proof relating multiple execution models constitutes a first step towards the more ambitious goal of thoroughly proving correctness of a virtual memory system, stated in a similar flavor by saying that user programs running on a system with virtual memory (implemented by a combination of hardware, assembly, and C) behave as if they were running on a machine where the user program can use the full physical address space.

# Chapter 11

# Analysis of the Auditing Burden in the Case Studies

In this chapter, I will try to answer the question whether our foundational end-to-end verification style has *measurable* benefits on the auditing burden, that is, whether predicting what a system does by reading code is easier for systems built using our approach than it is for traditional, unverified systems, or for systems where only individual parts have been verified without composing the proofs.

There are two key advantages that are expected to make our approach perform well at reducing the auditing burden:

- All intermediate specifications cancel out, so the top-level theorem does not depend on them, and they need not be audited.
- The tools used to generate and check the individual components have been proven correct (e.g. the compiler) or are proof-producing (e.g. the program logic), so their code need not be audited nor (blindly) trusted. Of course, the exception is Coq's kernel and the tools used to compile and run it, but since Coq is an exceptionally general-purpose verification tool, its auditing burden can be shared among all its many users.

To measure auditing burden, ideally, we would gather data on how much time it takes different developers to audit the implementations and specifications in question, and how many bugs they missed. Unfortunately, this kind of user study is beyond the scope of this thesis, so we make the assumption that the number of lines of code to be read, also known as trusted code base (TCB), can serve as a measure for auditing burden, even though, as described later, there are many complications that can invalidate this assumption.

In the following, we will look at each of the case studies from an auditing point of view (section 11.1, section 11.2 and section 11.3), discuss some related work (section 11.4), and conclude (section 11.5).

## 11.1 Lightbulb

### 11.1.1 Auditing the Theorem Statement

The top-level theorem of the lightbulb project (section 8.2.1) can serve as a concise description of the overall behavior of the system. It provides a guarantee (description of behavior) about the Kami 4-stage pipelined processor `p4mm` combined with the encoded RISC-V instructions of the lightbulb application (`instrencode lightbulb_insts`). In order to understand and audit that guarantee, one has to read all other definitions appearing in that statement, that is, `bytes_at`, `Kami.Semantics.Behavior`, `KamiRiscv.KamiLabelSeqR`, `prefix_of` and `goodHlTrace`, and while reading these, one also has to recursively follow all definitions referenced by these, which leads to a tree of definitions to be read. A summary of this tree is given in Table 11.1a. Each line contains a definition, or the name of a file in cases where a whole file needs to be read, and its number of lines of code.

The counts exclude definitions from the Coq standard library such as lists, strings, FMaps, and numbers, because these definitions are shared by most Coq projects, and the auditing burden can therefore be shared among them. The counts also exclude definitions from the coqutil word and map library and very generic, non-hardware-specific definitions from the Kami library about words, lists, maps, strings and structs. The choice to exclude this extra library code outside Coq's standard library might be controversial, but could be justified by saying that in a better future proof assistant, all these definitions (or similar ones that can be used for the same purposes) would be found in the proof assistant's standard library.

Table 11.1a only includes code that is (transitively) referenced by the top-level theorem statement. However, in order to trust that this statement accurately describes the behavior of the system, one also needs to trust the pipeline below the Kami hardware description language. It starts with some Coq code that defines a subset of the Bluespec language, which is then extracted to OCaml code (so we also need to trust Coq's OCaml extraction) together with the Coq definition of the processor, and then pretty-printed to Bluespec by some straightforward but lengthy (967 lines) OCaml code. LOC counts for the files related to this extraction process are given in Table 11.1b. The Bluespec code is compiled to Verilog by the Bluespec compiler, and combined with

| Component | LOC |
|---|---|
| `end2end_lightbulb` | 6 |
| `bytes_at` | 3 |
| `kami_mem_contains_bytes` | 3 |
| `get_kamiMemInit` | 5 |
| `kamiMemInit` | 1 |
| `Kami.Syntax` | 494 |
| `Kami.Semantics` | 470 |
| `KamiRiscv.KamiLabelSeqR` | 8 |
| `KamiLabelR` | 17 |
| `RqFromProc, RsToProc` | 19 |
| `prefix_of` | 2 |
| `lightbulb_spec up to goodHlTrace` | 186 |
| `TracePredicate (without stateful)` | 39 |
| `ReversedListNotations` | 5 |
| Total | 1258 |

(a) LOC counts of the TCB of the lightbulb top-level correctness statement (excluding libraries)

| Component | LOC |
|---|---|
| `Kami.Synthesize` | 115 |
| `Kami.Ext.BSyntax` | 213 |
| `Kami.Ext.Extraction` | 41 |
| `Ocaml/Main.ml` | 37 |
| `Ocaml/PP.ml` | 967 |
| Total | 1373 |

(b) LOC counts for extraction of Kami to Bluespec

Table 11.1: LOC counts of the TCB of the lightbulb case study

| Component | LOC |
|---|---|
| Lightbulb app | 176 |
| Bedrock2 compiler | 931 → 1628 |
| Kami 4-stage processor implementation | 1700 |
| Total | 2807 → 3504 |

Table 11.2: Implementation LOC of lightbulb case study. The numbers on the left of the arrows refer to the counts from 2020 as we reported in [Erbsen et al., 2021], whereas the numbers on the right of the arrows refer to the updated compiler as of 2024.

some Verilog[1] helper files provided by Bluespec (`FIFO2.v`, `RegFileLoad.v`, and `SizedFIFO.v`), as well as one hand-written project-specific Verilog file, `processor/integration/system.v`, spanning 223 lines, that provides memory supporting non-aligned access and also makes the RISC-V program available. More tools, including `yosys` and `nextpnr`, are then used to program the system onto an FPGA. This whole build process is orchestrated by Makefiles and runs on an operating system, so there is also some trust needed that these invoke the tools as expected.

### 11.1.2 Auditing the Implementation

Now, let us imagine that the system did not have any proofs, which means that in order to convince ourselves that it will behave as expected, we would have to read its implementation. Table 11.2 lists the number of lines that would have to be read.

The numbers for the lightbulb app and the first number for the Bedrock2 compiler are taken from Table 4 of our original publication about the lightbulb [Erbsen et al., 2021].

Since then, the Bedrock2 compiler has been extended considerably: From a four-phase compiler (just flattening, a simple register renaming phase, the RISC-V code generation phase and instruction encoding) to a seven-phase compiler (see Figure 3.3), and we kept building the lightbulb code with it on our continuous integration server. So, as of now (July 2024), the number of lines in the compiler implementation has grown to 1628, as the more detailed per-phase listing in Table C.1 in Appendix C shows.

To get an estimate of the size of the Kami 4-stage pipelined processor implementation, I read through its code and added up numbers as listed in Table C.2 in Appendix C. The code that ends up in the Kami processor implementation is spread over many files, and some code that ends up in the final 4-stage pipelined processor is actually reused from some intermediate refinement steps, e.g. from a 3-stage pipelined processor, so it is a bit tricky to get an accurate LOC count, so the 1700 LOC reported in Table 11.2 should be viewed as a rough estimate.

### 11.1.3 Comparison

Having LOC numbers for the definitions leading to the theorem statement as well as for the implementation, we can now compare them: For instance, one could say that theorem-based auditing would involve reading $1258 + 1373 = 2631$ LOC, whereas reading the implementation requires reading only $176 + 1700 = 1876$ LOC (excluding the compiler because it is a very generally reusable

[1] Note that, confusingly, the `.v` file extension is used for Coq files as well as for Verilog files.

component whose auditing burden can be shared among many projects and therefore does not need to be counted), so one could conclude that theorem-based auditing is actually **worse by a factor of 1.4**. However, one could also say that since the Kami-to-Bluespec extraction is part of the unverified pipeline, and every verification approach ends at some level, we only count the 1258 lines on which the top-level theorem statement depends, and since the Bedrock2 compiler *has* been verified, we would require an implementation auditor to also audit the compiler for a fair comparison, so auditing the implementation would require reading 3504 lines, so we could conclude that theorem-based auditing is **better by a factor of 2.8**.

However, there is also another hard-to-measure influence on the auditing burden: Code that serves as a specification is written with the sole purpose of making it as easy to understand as possible, whereas implementation code also needs to trade off ease of understanding against efficiency, so the auditing time spent per line of implementation code might be considerably higher than the auditing time spent per line of specification code.

## 11.2 Garage Door

Despite the big methodology-dependent variance in the possible results seen for the lightbulb, it is still interesting to apply the same kind of counting to the garage door project, because there, we decided to run it on a commercial RISC-V processor, so the bottom-most layer is now RISC-V instead of the Kami hardware description language.

### 11.2.1 Auditing the Theorem Statement

Table 11.3 lists LOC counts for the tree of definitions that one needs to read to audit the top-level theorem statement. The LOC count for the `run1` function of our RISC-V specification is not split up further because it comes from Table 4 of our publication about the lightbulb [Erbsen et al., 2021].

### 11.2.2 Auditing the Implementation

The size of the implementation can be measured by pretty-printing the Bedrock2 code to C and counting its lines, as reported in Table 11.4. These numbers do not actually correspond to the number of lines of code that we manually wrote as part of that project, because most code is generated either by Rupicola or by fiat-crypto. But the goal here is to estimate the LOC of a hypothetical, unverified system that does not use our tools, and the pretty-printed generated C code looks, in terms of code size, similar to what one would write manually in a traditional, unverified system.

| Component | LOC |
|---|---|
| garagedoor_correct | 2 |
| always | 6 |
| eventually | 9 |
| initial_conditions | 10 |
| regs_initialized | 2 |
| valid_machine | 3 |
| imem | 4 |
| ptsto_bytes | 1 |
| array | 8 |
| ptsto | 1 |
| mem_available | 4 |
| ex1 | 1 |
| emp | 1 |
| sep | 2 |
| MemoryLayout ml | 8 |
| Record RiscvMachine | 10 |
| run1 (RISC-V semantics) | 2053 |
| io_spec | 1 |
| only_mmio_satisfying | 2 |
| mmio_trace_abstraction_relation | 7 |
| boot_seq | 1 |
| BootSeq | 4 |
| OP | 1 |
| iocfg | 4 |
| LAN9250/SPI parts in lightbulb_spec | 140 |
| protocol_spec | 1 |
| labeled_transitions | 1 |
| protocol_step | 50 |
| garageowner_P | 9 |
| garageowner | 2 |
| RupicolaCrypto.Spec.chacha20_block | 30 |
| IPChecksum.Spec.ip_checksum | 6 |
| x25519_spec | 1 |
| Crypto.Spec.Curve25519.M | 21 |
| Crypto.Spec.MontgomeryCurve | 69 |
| Crypto.Spec.ModularArithmetic.F | 72 |
| TracePredicate | 48 |
| Total | 2595 |

Table 11.3: LOC counts of the TCB of the garage door top-level correctness statement (excluding libraries)

| Kind of code | LOC |
|---|---|
| Handwritten | 414 |
| Generated by Rupicola | 202 |
| Generated by fiat-crypto | 1852 |
| Total | 2468 |

Table 11.4: LOC counts of the garage door implementation

### 11.2.3 Comparison

If we do not count the implementation of the Bedrock2 compiler (arguing that compilers are very reusable), comparing the totals of Table 11.3 and Table 11.4, one might get the impression that theorem-based auditing is worse by a factor of 1.05, or if we include the 1628 LOC of the Bedrock2 compiler in the implementation, one could say that theorem-based auditing is better by a factor of 1.6. However, auditing the correctness of expert-written, highly optimized elliptic-curve code that is comparable in performance to the one created by fiat-crypto is really hard, so comparing implementation lines to specification lines 1:1 is not meaningful here. So, while I am happy to have computed some numbers, I do not believe that they carry any particularly valuable information.

## 11.3 Softmul

Gathering meaningful numbers on the auditing burden for the softmul trap handler seems even harder. The top-level theorem statement (section 10.3) and its dependencies are quite concise, except that they reference the RISC-V specification, which is quite long: 2053 LOC according to the count we did for the lightbulb project, but this count excludes the CSR-related code. Compared to this number, the LOC count of the implementation, as shown in Table 10.1, is tiny: The trap handler implementation consists of 8 lines of Bedrock2 code for the multiplication, 7 lines of Bedrock2 code for the instruction decoder, and 36 lines of assembly, resulting in a total of 51 lines of implementation.

But the interesting part that an auditor of the implementation would have to look at is not primarily in these 51 lines of implementation, but in the assumptions that this implementation implicitly makes about its environment, and to determine whether these assumptions hold, an auditor would have to read the RISC-V PDF specification, and probably also some compiler documentation and processor documentation, think really hard, and also perform lots of testing. Quantifying this effort with concrete numbers seems far out of scope for this thesis.

## 11.4 Related Work: Parfait

It is interesting to compare the numbers gathered in this chapter to numbers reported for the Parfait project (built on top of Knox [Athalye et al., 2022]), because they report numbers [Athalye, 2024] which, at first sight, look orders of magnitude better than ours: Their implementation consists of 2300 lines of software and 13500 lines of hardware (Verilog), i.e. 15800 LOC in total, while their

specification is only 40 LOC. If one did the same kind of division on these numbers as we did in the previous sections, it would seem that theorem-based auditing is better by a factor of almost 400 for their system, which would mean that their results are more than two orders of magnitude better than ours. Additionally, they prove not only functional correctness like we do, but also an information-preserving refinement, that is, they prove that the implementation does not leak more information than the specification.

So where does this big difference come from? On one hand, their statement depends on some more code than just the 40 LOC describing the specification-level system, namely also the code of the *driver* that turns the implementation-level interface into a specification-level interface and their Coq statement of what (information-preserving) refinement is. But, more importantly, they use a number of different verification tools (including F$^\star$ and Rosette, both of which rely on the Z3 SMT solver, as well as KaRaMeL to compile Low$^\star$ to C, and CompCert), and they have no automated way of verifying that these tools are compatible with each other: For instance, CompCert specifies RISC-V assembly semantics in Coq, but they also need assembly semantics in Rosette, so they hand-translated them into Rosette. A mistake there could lead to a missed bug, so in order to achieve the same level of assurance that our approach provides, one would also have to audit all language formalizations appearing in all used tools, as well as the code of the various verification tools themselves, the sum of which could easily dwarf the total implementation LOC of their system.

## 11.5 Conclusion

We have seen in this chapter that it is possible to gather LOC counts for the amount of code that needs to be audited in the theorem-based auditing approach and compare them to the number of LOC that need to be read in a traditional implementation-based auditing approach, and, depending on how one counts and on what the bottom-most specification is, the theorem-based auditing approach can be better by a factor of up to 2.8.

As we saw, the choice of how far down in the stack to end the verification affects the auditing burden, because different layers have different specification sizes. In particular, RISC-V tends to take more LOC to specify than the Kami hardware description language. In future work, it would be interesting to extend the verification pipeline further down. For instance, going all the way down to netlists might lead to an even smaller TCB, because the semantics of netlists should be quite simple.

| | No verification | Non-foundational | Foundational |
|---|---|---|---|
| Implementation | ✗ | | |
| Basic trust in build system & env | ✗ | ✗ | ✗ |
| Top-level application spec | | ✗ | ✗ |
| Spec of bottom-most layer | | ✗ | ✗ |
| Small proof-checking kernel | | | ✗ |
| Compiler for the proof-checking kernel | | | ✗ |
| Translations of intermediate specs | | ✗ | |
| Layer-specific verification tools | | ✗ | |
| SMT solver | | ✗ | |
| Compiler(s) for the implementation | | ✗ | |
| Compiler(s) for all verification tools | | ✗ | |

Table 11.5: Comparison of components in the TCB in different approaches. Generally, fewer ✗ is better, but one also needs to consider that certain components (e.g. specs or a small proof-checking kernel) require considerably less trust than others (e.g. the implementation of the code under study or verification tools).

However, we also saw that in this number-gathering process, one has to make many quite arbitrary decisions and encounters problems that are hard to account for:

- It is hard to decide which lines should be counted, especially given that different pieces of code have different potential for reusability and thus for sharing of auditing burden.
- It is hard to account for the fact that specification code is optimized for reading, while implementation code also needs to optimize for execution speed, and that therefore, specification code reading tends to be less work per LOC than implementation code reading.
- It is unclear how reliable code reading is, both for implementation code and specification code, i.e. we do not have directly usable data on how many bugs one would miss when reading code, and whether this missed-bug rate differs between specification code and implementation code.

Because of these limitations, I do not believe that the numbers obtained here are particularly meaningful, but, at least, I am happy to have gone through the exercise of trying to produce some numbers about the auditing burden.

On the other hand, in a qualitative analysis of the trusted code base (Table 11.5), our foundational approach (last column) looks much more favorable: Even though, compared to an approach without any verification (first column), it has more components in its TCB, they are of a different

nature that is easier to audit, because they were written with trustworthiness in mind rather than optimal performance.[2]

And compared to other verification projects that combine many layer-specific verification tools in a non-foundational way (middle column), we can see that there are many components with potentially intricate implementations that need to be trusted in that approach, but not in our foundational approach.

[2]The Coq proof-checking kernel, however, does contain performance-optimized code.

# Part III

# Conclusion

# Chapter 12

# Conclusion

**Exercising the different techniques** Let us see how the different techniques and building blocks presented in Part I have been useful in the three case studies presented in Part II:

- Omnisemantics (chapter 2) are the main form of language semantics used in all three case studies, and served us well in all of them. They enable use of forward simulations in the compiler in the presence of nondeterminism as well as a unified treatment of external nondeterminism (nondeterminism that gets recorded in the event trace, such as input) and internal nondeterminism (nondeterminism that does not get recorded in the trace, such as the addresses chosen for memory allocation). The type signature and meaning of omnisemantics judgments is the same as for WP judgments, so omnisemantics combine smoothly with WP-based program logics. There are only two places in the three case studies where omnisemantics were not used: In the semantics of the intermediate languages of Fiat Cryptography, because there, determinism is important and termination is obvious, so interpreters are the more natural choice for semantics there, and in the semantics of the Kami hardware description language, which uses a combination of traditional big-step operational semantics (to describe individual state transitions) and small-step operational semantics (to compose processor cycles), because Kami was designed before we started using omnisemantics. The end-to-end theorem of the lightbulb is expressed in terms of Kami's traditional semantics, which makes it more palatable and trustworthy for people who have not used omnisemantics before, while the two other case studies, which were published later, at a time when omnisemantics already seemed more accepted, use omnisemantics in their top-level statements.
- The Bedrock2 compiler (chapter 3) is used in all three case studies, and its stack allocation feature is used both in the lightbulb and in the garage door. The compiler can deal with hand-

written input programs as well as with input programs generated by higher-level compilers such as the Fiat-Crypto-to-Bedrock2 compiler or the Rupicola compiler, and the correctness theorems of its input programs (no matter whether hand-written or generated by higher-level compilers) compose with the Bedrock2 compiler correctness theorem in such a way that the semantics of the Bedrock2 language cancel out.

- The riscv-coq ISA specification (chapter 4) is also used in all three case studies. In the lightbulb, it has been exercised both from above (by proving the compiler against it) and from below (by proving the Kami processor against it) and thus cancels out in the end-to-end theorem. In contrast, the garage door opener runs on a commercial RISC-V processor that was not formally verified in Coq, so there, the riscv-coq ISA specification serves as the bottom-most specification layer in the stack and thus is part of the trusted code base of the end-to-end theorem. And finally, in the softmul case study, the riscv-coq ISA specification shows up both in the top-most specification layer (a processor specification where multiplication is implemented in hardware) and in the bottom-most layer (a processor where multiplication is implemented via a trap handler).
- The Live Verification techniques described in chapter 5 were used to verify a series of sample functions (Table 5.2), as well as a bigger development around a map data structure called crit-bit tree [Fukala, 2024].
- Automated splitting and merging of separation logic clauses (section 5.4.9) and the expression simplification techniques in chapter 6 are used extensively by the Live Verification framework.

As we can see, the techniques described in chapter 5, section 5.4.9, and chapter 6 have not yet been used in a bigger case study that combines several different verified components like the case studies in Part II do. However, the per-function correctness theorems that are proven with the Live Verification framework use the exact same semantics as the per-function correctness theorems produced with the Bedrock2 program logic as used in the three described case studies, so in future case studies, it should be possible to use Live Verification for program verification at the Bedrock2 source language level.

**Outlook** And indeed, at the time of writing, there are already two ongoing projects that will build on top of Live Verification:

The first one aims to develop a verified driver for a DMA-based network interface card (whereas the lightbulb and garage door case studies perform their I/O word-by-word through an MMIO-based SPI interface), and the plan is to use Live Verification for the driver functions.

The second project, called Fiat2, aims to create a high-level application programming language on top of Bedrock2 that can express business logic typically occurring in webapps as well as database queries within the same language, to allow for optimizations across the two. The language is simple and high-level enough that it can serve as the specification of a system's behavior, and human insight needed to optimize performance will be encoded as separate annotations that do not need to be read when auditing the code to understand the system's behavior. Compiling Fiat2 to Bedrock2 will require (among others) a map data structure, and it is planned to use the crit-bit tree structure developed in Live Verification there.

The work done in this thesis is just a small part in the bigger agenda of developing techniques and case studies to demonstrate that it is possible to build reliable systems whose behavior is easy to understand and which can be optimized without risking to change their behavior because each optimization comes with a proof that the behavior still corresponds to the behavior as specified by the high-level source program.

So, even though the amount of time spent and the amount of work published at this point might suggest that my PhD could and should come to an end here, the research started in this thesis should not come to an end, as there are still several ongoing projects and potential future projects based on this thesis' work that I would like to see happen and would like to help make happen.

# Part IV

# Appendix

# Appendix A

# Coq Code for Composing Simulations

```
Definition simulation{State1 State2: Type}
  (exec1: State1 → State1 → Prop)(exec2: State2 → State2 → Prop)
  (related: State1 → State2 → Prop): Prop :=
  forall s1 s2 s2', related s1 s2 → exec2 s2 s2' →
                    exists s1', related s1' s2' ∧ exec1 s1 s1'.

Definition compose_relation{State1 State2 State3: Type}
  (R12: State1 → State2 → Prop)(R23: State2 → State3 → Prop):
  State1 → State3 → Prop := fun s1 s3 ⇒ exists s2, R12 s1 s2 ∧ R23 s2 s3.

Lemma compose_sim{State1 State2 State3: Type}
  {R12: State1 → State2 → Prop}{R23: State2 → State3 → Prop}
  {exec1: State1 → State1 → Prop}
  {exec2: State2 → State2 → Prop}
  {exec3: State3 → State3 → Prop}:
  simulation exec1 exec2 R12 →
  simulation exec2 exec3 R23 →
  simulation exec1 exec3 (compose_relation R12 R23).
Proof.
  unfold simulation, compose_relation in *.
  intros S12 S23 s1 s3 s3' (s2 & HR12 & HR23) E3.
  specialize S23 with (1 := HR23) (2 := E3).
  destruct S23 as (s2' & HR23' & E2).
  specialize S12 with (1 := HR12) (2 := E2).
  destruct S12 as (s1' & HR12' & E1).
  eauto.
Qed.
```

Figure A.1: Standalone backward simulation composition proof in Coq

```
Definition simulation{State1 State2: Type}
           (exec1: State1 → (State1 → Prop) → Prop)
           (exec2: State2 → (State2 → Prop) → Prop)
           (related: State1 → State2 → Prop): Prop :=
  forall s1 s2 post1,
    related s1 s2 →
    exec1 s1 post1 →
    exec2 s2 (fun s2' ⇒ exists s1', related s1' s2' ∧ post1 s1').

Definition compose_relation{State1 State2 State3: Type}
  (R12: State1 → State2 → Prop)(R23: State2 → State3 → Prop):
  State1 → State3 → Prop := fun s1 s3 ⇒ exists s2, R12 s1 s2 ∧ R23 s2 s3.

Require Import Coq.Logic.PropExtensionality.
Require Import Coq.Logic.FunctionalExtensionality.

Lemma compose_sim{State1 State2 State3: Type}
      {exec1: State1 → (State1 → Prop) → Prop}
      {exec2: State2 → (State2 → Prop) → Prop}
      {exec3: State3 → (State3 → Prop) → Prop}
      {R12: State1 → State2 → Prop}
      {R23: State2 → State3 → Prop}:
  simulation exec1 exec2 R12 →
  simulation exec2 exec3 R23 →
  simulation exec1 exec3 (compose_relation R12 R23).
Proof.
  unfold simulation, compose_relation in *.
  intros S12 S23 s1 s3 post1 (s2 & HR12 & HR23) E1.
  specialize S12 with (1 := HR12) (2 := E1).
  specialize S23 with (1 := HR23) (2 := S12).
  simpl in S23.
  match goal with
  | H: exec3 s3 ?P |- exec3 s3 ?Q ⇒ replace Q with P; [exact H|]
  end.
  extensionality s2'.
  apply propositional_extensionality.
  firstorder idtac.
Qed.
```

Figure A.2: Standalone omnisemantics simulation composition proof in Coq

```
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.

Section WithParams.
  Context {mem: Type} {word: Type} {trace: Type}.

  Definition CallSpec(Program: Type): Type :=
    forall (p: Program) (funcname: string)
           (t: trace) (m: mem) (argvals: list word)
           (post: trace → mem → list word → Prop), Prop.

  Definition phase_correct{P1 P2: Type}
    (call1: CallSpec P1)(call2: CallSpec P2)
    (compile: P1 → option P2): Prop :=
    forall p1 p2,
      compile p1 = Some p2 →
      forall fname t m argvals post,
        call1 p1 fname t m argvals post →
        call2 p2 fname t m argvals post.

  Definition compose_phases{P1 P2 P3: Type}
    (phase12: P1 → option P2)(phase23: P2 → option P3):
    P1 → option P3 :=
    fun p1 ⇒ match phase12 p1 with
              | Some p2 ⇒ phase23 p2
              | None ⇒ None
              end.

  Lemma compose_phases_correct{P1 P2 P3: Type}
    {call1: CallSpec P1}{call2: CallSpec P2}{call3: CallSpec P3}
    {compile12: P1 → option P2}{compile23: P2 → option P3}:
    phase_correct call1 call2 compile12 →
    phase_correct call2 call3 compile23 →
    phase_correct call1 call3 (compose_phases compile12 compile23).
  Proof.
    unfold compose_phases, phase_correct. intros Ok12 Ok23 p1 p3 E23 * C1.
    destruct (compile12 p1) as [p2|] eqn:E12; [|discriminate].
    eapply Ok23. 1: exact E23.
    eapply Ok12. 1: exact E12.
    exact C1.
  Qed.
End WithParams.
```

Figure A.3: Standalone call spec composition proof in Coq

# Appendix B

# Sample Log of Running the step Tactic Repeatedly

This log was obtained by commenting out the line `store(p, res);` at the end of `bst_add`, replacing `/**.` by `/*?.` after the closing brace after the `return`, and running `step. step. step.` many times, where `step` has been redefined as follows:

```
Ltac step ::=
  assert_no_error;
  lazymatch goal with
  | |- ?g ⇒ idtac "Goal:"; idtac g; idtac "Step:"
  end;
  first [ step0 Logging.run_logger_thunk
        | idtac "No applicable tactic" ].
```

It leads to the following output:

```
Goal:
(t = t ∧
 (\[/[1]] = 0 ∧
  <{ * allocator_failed_below 12
     * (EX rootp : word, <{ * uintptr rootp p
                            * (EX sk : tree_skeleton, bst' sk s rootp) }>)
     * R }> m4 ∨
  \[/[1]] = 1 ∧
  <{ * allocator
     * (EX rootp : word,
```

```
          <{ * uintptr rootp p
             * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>)
     * R }> m4))
```

**Step**:

purify & zify

**Goal**:

```
(t = t ∧
 (\[/[1]] = 0 ∧
  <{ * allocator_failed_below 12
     * (EX rootp : word, <{ * uintptr rootp p
                            * (EX sk : tree_skeleton, bst' sk s rootp) }>)
     * R }> m4 ∨
  \[/[1]] = 1 ∧
  <{ * allocator
     * (EX rootp : word,
          <{ * uintptr rootp p
             * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>)
     * R }> m4))
```

**Step**:

step_hook

**Goal**:

```
(t = t ∧
 (\[/[1]] = 0 ∧
  <{ * allocator_failed_below 12
     * (EX rootp : word, <{ * uintptr rootp p
                            * (EX sk : tree_skeleton, bst' sk s rootp) }>)
     * R }> m4 ∨
  \[/[1]] = 1 ∧
  <{ * allocator
     * (EX rootp : word,
          <{ * uintptr rootp p
             * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>)
     * R }> m4))
```

**Step**:

cleanup_step

**Goal**:

```
(t = t ∧
 (\[/[1]] = 0 ∧
  <{ * allocator_failed_below 12
     * (EX rootp : word, <{ * uintptr rootp p
                            * (EX sk : tree_skeleton, bst' sk s rootp) }>)
     * R }> m4 ∨
  \[/[1]] = 1 ∧
  <{ * allocator
     * (EX rootp : word,
          <{ * uintptr rootp p
             * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>)
     * R }> m4))
```

**Step**:

split

**Goal**:

```
(t = t)
```

**Step**:

step_hook

**Goal**:

```
(\[/[1]] = 0 ∧
 <{ * allocator_failed_below 12
    * (EX rootp : word, <{ * uintptr rootp p
                           * (EX sk : tree_skeleton, bst' sk s rootp) }>)
    * R }> m4 ∨
 \[/[1]] = 1 ∧
 <{ * allocator
    * (EX rootp : word,
         <{ * uintptr rootp p
            * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>)
    * R }> m4)
```

**Step**:

step_hook

**Goal**:

```
(\[/[1]] = 1 ∧
 <{ * allocator
    * (EX rootp : word,
```

```
         <{ * uintptr rootp p
            * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>)
    * R }> m4)
```

**Step**:

```
split
```

**Goal**:

```
(\[/[1]] = 1)
```

**Step**:

```
zify_goal; xlia zchecker
```

**Goal**:

```
(<{ * allocator
    * (EX rootp : word,
         <{ * uintptr rootp p
            * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>)
    * R }> m4)
```

**Step**:

```
heapletwise_step
```

**Goal**:

```
(canceling
   [|allocator; EX rootp : word,
                 <{ * uintptr rootp p
                    * (EX sk : tree_skeleton,
                         bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>; R|]
   (m6 \*/ (m8 \*/ (m9 \*/ (m2 \*/ (m1 \*/ (m0 \*/ m5))))))) True)
```

**Step**:

```
heapletwise_step
```

**Goal**:

```
(canceling
   [|EX rootp : word,
       <{ * uintptr rootp p
          * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) rootp) }>; R|]
   (m6 \*/ (m8 \*/ (m9 \*/ (m1 \*/ (m0 \*/ m5))))) True)
```

**Step**:

```
heapletwise_step
```

**Goal**:

```
(canceling
```

```
   [|<{ * uintptr ?x p
        * (EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) ?x) }>; R|]
   (m6 \*/ (m8 \*/ (m9 \*/ (m1 \*/ (m0 \*/ m5))))) True)
Step:
heapletwise_step
Goal:
(canceling
   [|uintptr ?x p; EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) ?x; R|]
   (m6 \*/ (m8 \*/ (m9 \*/ (m1 \*/ (m0 \*/ m5))))) True)
Step:
heapletwise_step
Goal:
(canceling [|EX sk : tree_skeleton, bst' sk (fun x : Z ⇒ x = \[vAdd] ∨ s x) /[0]; R|]
   (m6 \*/ (m8 \*/ (m9 \*/ (m1 \*/ m5)))) True)
Step:
heapletwise_step
Goal:
(canceling [|bst' ?x (fun x : Z ⇒ x = \[vAdd] ∨ s x) /[0]; R|]
   (m6 \*/ (m8 \*/ (m9 \*/ (m1 \*/ m5)))) True)
Step:
step_hook
Goal:
(canceling [|bst' Leaf (fun x : Z ⇒ x = \[vAdd] ∨ s x) /[0]; R|]
   (m6 \*/ (m8 \*/ (m9 \*/ (m1 \*/ m5)))) True)
Step:
step_hook
Goal:
(canceling [|emp (/[0] = /[0] ∧ is_empty_set (fun x : Z ⇒ x = \[vAdd] ∨ s x)); R|]
   (m6 \*/ (m8 \*/ (m9 \*/ (m1 \*/ m5)))) True)
Step:
heapletwise_step
Goal:
(/[0] = /[0] ∧ is_empty_set (fun x : Z ⇒ x = \[vAdd] ∨ s x))
Step:
split
Goal:
```

(/[0] = /[0])

**Step**:

step_hook

**Goal**:

(is_empty_set (**fun** x : Z ⟹ x = \[vAdd] ∨ s x))

**Step**:

No applicable tactic

# Appendix C

# More LOC Counts

| Component | LOC |
|---|---|
| composed_compile | 9 |
| flatten_functions | 125 |
| NameGen | 4 |
| StringNameGen | 22 |
| useimmediate_functions | 55 |
| dce_functions | 155 |
| regalloc_functions | 453 |
| Spilling up to spill_functions | 259 |
| riscvPhase | 5 |
| FitsStack up to stack_usage | 55 |
| FlatToRiscvDef | 219 |
| compile_ext_call | 17 |
| instrencode | 2 |
| riscv.Utility.Encode up to encode | 248 |
| Total | 1628 |

Table C.1: LOC counts of the compiler implementation (excluding specifications and proofs)

| Component | LOC |
|---|---|
| PrimFifo | 123 |
| MemTypes | 21 |
| ProcFourStDec up to Definition p4st: | 125 |
| Kami.Ex.ProcFourStDec.ProcFDE.fetchDecode | 5 |
| Kami.Ex.ProcFDCorrect.fetchICacheDecode | 4 |
| Kami.Ex.ProcFCorrect.fetchICache | 3 |
| Kami.Ex.ProcFInl.fetchICache | 2 |
| Kami.Ex.ProcFetch.fetchICache (file up to this definition) | 160 |
| Kami.Ex.ProcFetchDecode.decoder (relevant parts in that file) | 250 |
| Kami.Ex.ProcThreeStage up to end of writeback, but without fetchDecode | 450 |
| IsaRv32.v | 461 |
| Plumbing in various files | 100 |
| Total | 1704 |

Table C.2: LOC counts of the Kami 4-stage processor (excluding specifications and proofs)

# Bibliography


Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification – The KeY Book*. Lecture Notes in Computer Science, Vol. 10001. Springer International Publishing, Cham. http://link.springer.com/10.1007/978-3-319-49812-6

Eyad Alkassar, Mark A. Hillebrand, Steffen Knapp, Rostislav Rusev, and Sergey Tverdyshev. 2007. Formal Device and Programming Model for a Serial Interface. In *Proceedings of 4th International Verification Workshop in Connection with CADE-21, Bremen, Germany, July 15-16, 2007 (CEUR Workshop Proceedings, Vol. 259)*, Bernhard Beckert (Ed.). CEUR-WS.org. http://ceur-ws.org/Vol-259/paper04.pdf

Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. 2008a. The Verisoft Approach to Systems Verification. In *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08) (LNCS, Vol. 5295)*, Natarajan Shankar and Jim Woodcock (Eds.). Springer, 209–224.

Eyad Alkassar, Norbert Schirmer, and Artem Starostin. 2008b. Formal Pervasive Verification of a Paging Mechanism. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Heidelberg, 109–123. https://doi.org/10.1007/978-3-540-78800-3_9

A.W. Appel. 2001. Foundational Proof-Carrying Code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 247–256. https://doi.org/10.1109/LICS.2001.932501

Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position Paper: The Science of Deep Specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (Oct. 2017), 20160331. https://doi.org/10.1098/rsta.2016.0331

Andrew W. Appel and Sandrine Blazy. 2007. Separation Logic for Small-Step Cminor. In *Theo-*

*rem Proving in Higher Order Logics*. Springer, 5–21. https://link.springer.com/chapter/10.1007/978-3-540-74591-4_3

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA.

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–31. https://doi.org/10.1145/3290384

Anish Athalye. 2024. *Formally Verifying Secure and Leakage-Free Systems: From Application Specification to Circuit-Level Implementation*. Ph. D. Dissertation. Massachusetts Institute of Technology. https://pdos.csail.mit.edu/papers/aathalye-thesis.pdf

Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with Information-Preserving Refinement. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 503–519. https://www.usenix.org/conference/osdi22/presentation/athalye

Ralph-Johan Back. 1980. Semantics of Unbounded Nondeterminism. In *Automata, Languages and Programming (Lecture Notes in Computer Science)*, Jaco de Bakker and Jan van Leeuwen (Eds.). Springer, Berlin, Heidelberg, 51–63. https://doi.org/10.1007/3-540-10003-2_59

Mike Barnett. 2009. Initial Commit in the Dafny Repo. https://github.com/dafny-lang/dafny/commit/b70a417a8cb3040aea90e5c85a0119a178f1de98

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*. Vol. 4111. Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

Daniel J. Bernstein. 2006. Crit-Bit Trees. https://cr.yp.to/critbit.html

William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. 1989. An Approach to Systems Verification. *Journal of Automated Reasoning* 5, 4 (Dec. 1989), 411–428. https://doi.org/10.1007/BF00243131

Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Mail-

lard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.1

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2015. Let's Verify This with Why3. *International Journal on Software Tools for Technology Transfer* 17, 6 (Nov. 2015), 709–727. https://doi.org/10.1007/s10009-014-0314-5

Thomas Bourgeat. 2023. *Specification and Verification of Sequential Machines in Rule-Based Hardware Languages.* Ph. D. Dissertation. MIT. http://adam.chlipala.net/theses/bthom.pdf

Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andy Wright, and Adam Chlipala. 2023. Flexible Instruction-Set Semantics via Abstract Monads (Experience Report). *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 192:108–192:124. https://doi.org/10.1145/3607833

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-Coq to Real-World Haskell Code (Experience Report). *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 89:1–89:16. https://doi.org/10.1145/3236784

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, USA, 209–224.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61, 1-4 (June 2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP'13)*. Springer-Verlag, Rome, Italy, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omniseman-

tics: Smooth Handling of Nondeterminism. *ACM Transactions on Programming Languages and Systems* 45, 1 (March 2023), 5:1–5:43. https://doi.org/10.1145/3579834

Dipak L. Chaudhari and Om Damani. 2014. Automated Theorem Prover Assisted Program Calculations. In *Integrated Formal Methods*, Elvira Albert and Emil Sekerinski (Eds.). Vol. 8739. Springer International Publishing, Cham, 205–220. https://link.springer.com/10.1007/978-3-319-10181-1_13

Deepak L. Chaudhari and Om P. Damani. 2015. CAPS, A Calculational Assistant for Programming from Specifications. https://www.cse.iitb.ac.in/~dipakc/CAPS/

Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2018. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. *Journal of Automated Reasoning* 61, 1 (June 2018), 141–189. https://doi.org/10.1007/s10817-017-9446-0

Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In *ICFP*. ACM Press, 391. https://doi.org/10.1145/2500365.2500592

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (Aug. 2017), 1–30. https://doi.org/10.1145/3110268

Nils Anders Danielsson. 2012. Operational Semantics Using the Partiality Monad. *ICFP'12* 47, 9 (Sept. 2012), 127–138. https://doi.org/10.1145/2398856.2364546

Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. 2010. From Operating-System Correctness to Pervasively Verified Applications. In *Proc. IFM*. Springer-Verlag, 105–120. https://hal.inria.fr/inria-00524575/document

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL 2015*. ACM, New York, NY, USA, 689–700. https://doi.org/10.1145/2676726.2677006

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 604–619. https://doi.org/10.1145/3453483.3454065

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-

Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1202–1219. https://doi.org/10.1109/SP.2019.00005

Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. 2024. Foundational Integration Verification of a Cryptographic Server. *PLDI'24* (June 2024). https://dl.acm.org/doi/10.1145/3656446

Jean-Christophe Filliâtre. 2009. Initial Commit in the Why3 Repo. https://gitlab.inria.fr/why3/why3/-/commit/44577e849e248adf957b65cfd662721d90fd09cb

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Nissim Francez, C. A. R. Hoare, Daniel J. Lehmann, and Willem P. De Roever. 1979. Semantics of Nondeterminism, Concurrency, and Communication. *J. Comput. System Sci.* 19, 3 (Dec. 1979), 290–308. https://doi.org/10.1016/0022-0000(79)90006-0

Dan Frumin, Léon Gondelman, and Robbert Krebbers. 2019. Semi-Automated Reasoning About Non-determinism in C Expressions. In *Programming Languages and Systems*, Luís Caires (Ed.). Vol. 11423. Springer International Publishing, Cham, 60–87. https://doi.org/10.1007/978-3-030-17184-1_3

Viktor Fukala. 2024. Formally Verified Low-Level C Implementation of Crit-Bit Trees in a Live Verification Tool. *PLDI'24 Student Research Competition* (2024). https://samuelgruetter.net/assets/CritBit_PLDI24_SRC.pdf

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

Shilpi Goel, Anna Slobodova, Rob Sumners, and Sol Swords. 2020. Verifying X86 Instruction Implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 47–60. https://doi.org/10.1145/3372885.3373811

Jason Gross, Andres Erbsen, Jade Philipoom, Miraya Poddar-Agrawal, and Adam Chlipala. 2022. Accelerating Verified-Compiler Development with a Verified Rewriting Engine. In *13th Interna-*

*tional Conference on Interactive Theorem Proving (ITP 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2022.17

Samuel Gruetter. 2017. *Improving the Coq Proof Automation Tactics of the Verified Software Toolchain, Based on a Case Study on Verifying a C Implementation of the AES Encryption Algorithm*. Technical Report. MSc thesis, EPFL/Princeton University. https://www.cs.princeton.edu/research/techreps/TR-999-17

Samuel Gruetter, Thomas Bourgeat, and Adam Chlipala. 2024a. Verifying Software Emulation of an Unsupported Hardware Instruction. *ITP'24* (2024). https://doi.org/10.4230/LIPIcs.ITP.2024.17

Samuel Gruetter, Viktor Fukala, and Adam Chlipala. 2024b. Live Verification in an Interactive Proof Assistant. *PLDI'24* (June 2024). https://dl.acm.org/doi/10.1145/3656439

Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2014. *Deep Specifications and Certified Abstraction Layers*. Technical Report Technical Report YALEU/DCS/TR-1500. Yale University.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. *ACM SIGPLAN Notices* POPL'15 (Jan. 2015). https://doi.org/10.1145/2775051.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 653–669.

Gernot Heiser. 2020. *The seL4 Microkernel – An Introduction*. Technical Report. https://sel4.systems/About/seL4-whitepaper.pdf

Martin Hentschel, Richard Bubel, and Reiner Hähnle. 2014. Symbolic Execution Debugger (SED). In *Runtime Verification*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.). Vol. 8734. Springer International Publishing, Cham, 255–262. http://link.springer.com/10.1007/978-3-319-11164-3_21

Wim H. Hesselink. 2010. Alternating States for Dual Nondeterminism in Imperative Programming. *Theoretical Computer Science* 411, 22 (May 2010), 2317–2330. https://doi.org/10.1016/j.tcs.2010.03.016

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Warren A. Hunt. 1989. Microprocessor Design Verification. http://www.cs.utexas.edu/users/boyer/ftp/cli-reports/048.pdf

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*. Vol. 6617. Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55. http://link.springer.com/10.1007/978-3-642-20398-5_4

Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A Quick Tour of the VeriFast Program Verifier. In *Programming Languages and Systems*, Kazunori Ueda (Ed.). APLAS 2010, Vol. 6461. Springer Berlin Heidelberg, Berlin, Heidelberg, 304–311. http://link.springer.com/10.1007/978-3-642-17164-2_21

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 111–125. https://doi.org/10.1145/3314221.3314595

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The Verified CakeML Compiler Backend. *Journal of Functional Programming* 29 (2019), e2. https://doi.org/10.1017/S0956796818000229

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. ACM, Vancouver BC Canada, 94–105. https://doi.org/10.1145/2804302.2804319

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70. https://doi.org/10.1145/2560537

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*. Association for Computing Machinery, Cascais, Portugal, 234–248. https://doi.org/10.1145/3293880.3294106

Robbert Krebbers. 2015. *The C Standard Formalized in Coq*. Ph. D. Dissertation. Radboud University, Nijmegen. https://robbertkrebbers.nl/thesis.html

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/3009837.3009855

Peter Lammich. 2015. The Isabelle Refinement Framework. *Kolloquium Programmiersprachen* (2015). https://www.complang.tuwien.ac.at/kps2015/proceedings/KPS_2015_submission_13.pdf

Peter Lammich. 2017. Refinement to Imperative HOL. *Journal of Automated Reasoning* 62, 4 (2017), 481–503. https://doi.org/10.1007/s10817-017-9437-1

Adam Langley. 2016. Memcpy (and Friends) with NULL Pointers. https://www.imperialviolet.org/2016/06/26/nonnull.html

Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. 2011. The Boogie Verification Debugger (Tool Paper). In *Software Engineering and Formal Methods (Lecture Notes in Computer Science)*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer, Berlin, Heidelberg, 407–414. https://doi.org/10.1007/978-3-642-24690-6_28

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. *PLDI'17* 52, 6 (June 2017), 633–647. https://doi.org/10.1145/3140587.3062343

K. Rustan M. Leino. 2013. Developing Verified Programs with Dafny. In *2013 35th International Conference on Software Engineering (ICSE)*. 1488–1490. https://doi.org/10.1109/ICSE.2013.6606754

K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 6 (Nov. 2017), 94–97. https://doi.org/10.1109/MS.2017.4121212

K. Rustan M. Leino and Valentin Wüstholz. 2014. The Dafny Integrated Development Environment. In *Proceedings 1st Workshop on Formal Integrated Development Environment, Grenoble, France, April 6, 2014 (Electronic Proceedings in Theoretical Computer Science, Vol. 149)*, Catherine

Dubois, Dimitra Giannakopoulou, and Dominique Méry (Eds.). Open Publishing Association, 3–15. https://doi.org/10.4204/EPTCS.149.2

Xavier Leroy. 2009a. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy. 2009b. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (Dec. 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Xavier Leroy. 2015. Using Coq's Evaluation Mechanisms in Anger. https://gallium.inria.fr/blog/coq-eval/

Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2.* Report. INRIA. 26 pages. https://hal.inria.fr/hal-00703441

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *In Proceedings of the 22nd ACM Symposium on Principles of Programming Languages.*

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *PLDI 2019.* Association for Computing Machinery, New York, NY, USA, 1041–1053. https://doi.org/10.1145/3314221.3314622

Nancy Lynch and Frits Vaandrager. 1996. Forward and Backward Simulations: II. Timing-Based Systems. *Information and Computation* (1996). https://doi.org/10.1006/inco.1996.0060

William Mansky, Wolf Honoré, and Andrew W. Appel. 2020. Connecting Higher-Order Separation Logic to a First-Order Outside World. In *Programming Languages and Systems (ESOP 2020)*, Peter Müller (Ed.). Springer International Publishing, Cham, 428–455. https://doi.org/10.1007/978-3-030-44914-8_16

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the de Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: Minimizing the Coq Extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018).* Association for Computing Machinery, New York, NY, USA, 172–185. https://doi.org/10.1145/3167089

Keiko Nakata and Tarmo Uustalu. 2010. Mixed Induction-Coinduction at Work for Coq. *2nd Work-*

*shop of Coq users, developers, and contributors* (2010). http://www.cs.ioc.ee/~keiko/papers/Coq2.pdf

Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford University, Stanford, CA, USA.

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19).* Association for Computing Machinery, New York, NY, USA, 225–242. https://doi.org/10.1145/3341301.3359641

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17).* Association for Computing Machinery, New York, NY, USA, 252–269. https://doi.org/10.1145/3132747.3132748

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2024. A Proof Assistant for Higher-Order Logic. https://isabelle.in.tum.de/dist/Isabelle2024/doc/tutorial.pdf

Michael Norrish. 1998. *C Formalised in HOL.* Technical Report UCAM-CL-TR-453. 156 pages.

Clément Pit-Claudel and Thomas Bourgeat. 2021. An Experience Report on Writing Usable DSLs in Coq. In *CoqPL'21: The Seventh International Workshop on Coq for PL*, Assia Mahboubi and Amin Timany (Eds.). https://pit-claudel.fr/clement/papers/koika-dsls-CoqPL21.pdf

Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational Compilation for Performance-Critical Applications: Extensible Proof-Producing Translation of Functional Models into Low-Level Code. In *PLDI 2022.* Association for Computing Machinery, New York, NY, USA, 918–933. https://doi.org/10.1145/3519939.3523706

G. D. Plotkin. 1976. A Powerdomain Construction. *Siam J. of Computing* (1976).

Shaz Qadeer. 2020. ModelViewer and BVD Projects. https://github.com/boogie-org/boogie/issues/293

Steven Schäfer, Sigurd Schneider, and Gert Smolka. 2016. Axiomatic Semantics for Compiler Verification. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs.* ACM, St. Petersburg FL USA, 188–196. https://doi.org/10.1145/2854065.2854083

Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Im-*

*plementation (OSDI'18)*. USENIX Association, USA, 287–305. https://unsat.cs.washington.edu/papers/sigurbjarnarson-nickel.pdf

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-assembly Linking and Lightweight Modular Verification. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–31. https://doi.org/10.1145/3371091

Joseph Tassarotti and Robert Harper. 2019. A Separation Logic for Concurrent Randomized Programs. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–30. https://doi.org/10.1145/3290377

Thomas Tuerk. 2010. Local Reasoning about While-Loops. *VSTTE 2010* (2010).

Christian Urban. 2019. The Isabelle Cookbook (Draft). https://cflmark.nms.kcl.ac.uk/hg/isabelle-cookbook/raw-file/883ce9c7b13b/progtutorial.pdf

Alexander Vaynberg and Zhong Shao. 2012. Compositional Verification of a Baby Virtual Memory Manager. In *Certified Programs and Proofs*, Chris Hawblitzel and Dale Miller (Eds.). Springer, Berlin, Heidelberg, 143–159. https://doi.org/10.1007/978-3-642-35308-6_13

Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

P. Wadler and S. Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler Verification Meets Cross-Language Linking via Data Abstraction. In *OOPSLA*. ACM Press, 675–690. https://doi.org/10.1145/2660193.2660201

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 62:1–62:30. https://doi.org/10.1145/3290375

Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 197:1–197:28. https://doi.org/10.1145/3428265

Andrew Waterman and Krste Asanovic (Eds.). 2019. The RISC-V Instruction Set Manual, Volume

I: User-Level ISA, Document Version 20191213. *RISC-V Foundation* (Dec. 2019). https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

Andrew Waterman, Krste Asanovic, and John Hauser (Eds.). 2021. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203. *RISC-V International* (Dec. 2021). https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 23:1–23:29. https://doi.org/10.1145/3434304

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 51:1–51:32. https://doi.org/10.1145/3371119

Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3341301.3359647

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. *ITP'21* (2021), 19 pages, 770230 bytes. https://doi.org/10.4230/LIPICS.ITP.2021.32