Foundational Integration Verification of Diverse Software and Hardware Components

by

Andres Erbsen

B.S., Massachusetts Institute of Technology (2017)
S.M., Massachusetts Institute of Technology (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 2023

Authored by:	Andres Erbsen Department of Electrical Engineering and Computer Science December 30, 2022
Certified by:	Adam Chlipala Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Foundational Integration Verification of Diverse Software and Hardware Components

by

Andres Erbsen

Submitted to the Department of Electrical Engineering and Computer Science
on December 30, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The engineering of computer systems is distinguished by a long-standing tradition of building on quicksand. Even the most venerable and critical systems have a history of serious bugs and security vulnerabilities. Human fallibility continues to prevail.

Computer-checked mathematical proofs of software correctness have emerged as a promising method to rule out large classes of bugs. However, the appropriate notion of correctness for a computer-systems component is exceedingly difficult to specify correctly in isolation, and unrelated verification of adjacent components does not rule out bugs due to their interactions. Therefore, I argue for (1) centering systems-verification efforts around interface specifications within a proof assistant, (2) proving both clients and implementations of an interface, and (3) using these results to prove an integrated-correctness theorem stated without referencing the internal interfaces.

I present a serious (several-year, several-person) exploration of what formally proven computer-systems development would look like if this practice were standard, culminating in precedent-setting case studies involving embedded implementations of networked software and elliptic-curve cryptography. Whole-system correctness theorems spanning from application behavior to hardware designs are proven by instantiating correctness proofs of compilers, translation validators, processor implementations, and mathematical theories. For example, RISC-V machine code for a public-key-authenticated Ethernet server is proven to always eventually satisfy a trace predicate.

Specifications of imperative languages within the system are modeled using an underappreciated technique that we call omnisemantics. Choosing an inductively defined weakest-precondition predicate transformer as the semantics of a language allows unspecified behavior to be encoded using rules with universally quantified premises, greatly simplifying compiler-correctness proofs and program-logic construction.

Thesis Supervisor: Adam Chlipala
Title: Professor of Electrical Engineering and Computer Science

Contents

Chapter 1

Introduction

Building computer systems that function as specified across a broad range of usage remains challenging despite substantial advances in programming tools. That this difficulty persists in spite of extensive efforts sets programming apart from mature engineering disciplines, which can achieve near-perfect reliability within design limits. The inherent nebulosity of desirable behavior in response to a vast space of possible inputs is only a part of the problem as evidenced by the obviousness of the failures in production software: simple user-facing specifications such as "doesn't freeze" are not reliably upheld either. Worse, even violations of subtle specifications internal to the system can drastically affect its overall behavior. Mundane confusion over which parts of memory store which information, or whether that information is an input to some code or itself a program to be executed, causes a strong majority of computer security vulnerabilities [Sum+22; Keh19; Mil19; Pro]. Even projects that consistently prioritize correctness and security are bound by human fallibility: for example, both of OpenBSD's "only two remote holes in the default install" were caused by failure to follow an established and sound programming discipline. The costs of computer unreliability are borne primarily by users, but development of computer systems themselves can grind to a halt when the proliferation of bugs spoils design foresight and necessitates trial-and-error programming (humorously, [Mic13]).

I started and led to completion a set of interconnected experiments in using computer-checked mathematical proofs of comprehensive application-level specifications as the primary means of quality assurance for computer systems. The approach was straightforward, almost naive: to understand the relevant design-review and code-auditing tasks and to encode that understanding in lemmas and mechanized proof procedures. I would pick a programming discipline I understand well and formalize a canonical example of it in the Coq proof assistant, relying on my experience about what considerations are important when reviewing this kind of code to choose what correctness theorem to prove. Then I would pick another qualitatively different component that interfaces with an existing example, repeat the exercise, and state and prove a com-

bined correctness theorem. Step-by-step, my collaborators and I expanded the verified stack inside-out while insisting on interface specifications that appropriately delineate the responsibilities between components and keep verification effort manageable. The resulting artifacts demonstrate *integrated* verification of realistic implementations of small systems, without formality gaps, across a wide range of programming environments, languages, and tooling, each verified using context-appropriate methods.

I chose to center this effort around cryptographic implementations and networked embedded systems. The primary reason for this choice is that reliability, correctness, and security are already recognized as important in these domains, opening up the opportunity for adoption of proven implementations. Another important feature of these domains is that they place substantial constraints (performance, resource usage, reliability, absence of information leaks) on the implementations, and these constraints encourage careful consideration of project scope, deterring frivolous additions of mechanisms and functionality. Following scoping decisions of leading cryptography and embedded-networking projects allowed the clean-slate effort I led to get to proving important properties of useful components and complete systems without taking a detour to reimplement (or axiomatically model) incidental developments just to remain realistic. While the focus on constrained domains worked out great for experimenting with integration verification and proof-based development in this setting, it is also a limitation as far as interpretation of the outcomes of these experiments is concerned: the trade-offs for specifying and verifying a tightly integrated part of an elaborate unverified system are expected to differ.

I strove to seek out, specify, and integrate challenging components whose functional correctness follows from delicate dedicated reasoning. In other words, adequacy and interoperability of proof methods is in the front and center of this work. I am pointing this goal out in contrast to trying to come up with one tool or methodology that could tackle all verification challenges within the application domain or trying to distill the essence that remains of a particular verification challenge once interface constraints are removed. Excellent engineering stands out for attention to context, fitting within the constraints of the environment and exploiting its flexibility, leaning on abstractions whenever applicable and looking through them to achieve better-than-generic results. I sought to demonstrate that building a computer system in a proof assistant can support these ambitions with grace. There is no specification that cannot be negotiated with, no red tape that says "abstraction barrier – do not cross!", but rather the engineering details of each interface are appropriately tackled by the (different!) proof methods used on each side of that interface. The developments I will review here are conclusive on this point: qualitatively, the Coq proof assistant is a great environment for proof of tricky integration of computer systems.

A pattern that emerged as successful for modeling internal interfaces describing the behavior of sequential imperative programs is the use of weakest-precondition predicate transformers. This encoding choice appears important precisely because the rules being described vary: underspecification (unspecified behavior) and absolute

requirements (avoiding undefined behavior) can be straightforwardly specified on a case-by-case basis within the same weakest-precondition definition. While predicate transformers are established for reasoning about concrete programs, I pioneered their application to state and prove correctness of practical compilers and instruction-set implementations. The canonical semantics of the C-like language at the center of this ecosystem are given by inductively defined weakest preconditions (omnisemantics); this definition elegantly serves the needs of a total-correctness program logic and certified and proof-generating compilers from and to this language. In particular, omnisemantics enable compiler-correctness proofs to exploit underspecification (e.g., of addresses and contents of freshly allocated memory) without resorting to backward-simulation arguments. For contrast, CompCert handles the same conceptual challenge using a technical deterministic memory model where printing pointers and comparing pointers to independently allocated memory locations are modeled as undefined behavior to avoid reasoning about underspecification [LB08].

1.1 Structure of This Thesis

Section 1.2 gives an overview of the components and integrations that were implemented and proven as a part of the collaborative effort covered in this thesis. The remainder of this chapter provides an elaborate explanation for why I pursued integrated, modular, and foundational verification. At first, the argument is driven by a number of examples of how subtleties and disagreements about interfaces of verified and unverified components alike jeopardize the correct operation of systems built from these components. Returning to desired outcomes from integration verification, I review some verification strategies and past efforts to tackle similar challenges.

Chapter 2 presents a detailed but primarily informal overview of the Bedrock2 programming language at the center of this effort. The focus is on the semantics of memory, addresses, allocation, and (sequential) interaction with adjacent components. I compare the design choices in Bedrock2 to different perspectives on the C programming language and consider their impacts on programming and compiling, program verification and compiler verification. Section 2.5 wraps this up with a general translation from Bedrock2 to C, intended for use with mainstream C compilers.

Chapter 3 covers the programmer-facing formalization of the Bedrock2 language. It starts out with some motivating program-proof examples. The presentation is structured around enabling desirable proof steps for each primitive command of Bedrock2, fleshing out cases of the definition of verification conditions along the way. The same chapter also includes discussion of proof procedures frequently used across Bedrock2 programs but does not aim for exhaustive coverage of this topic. I hope the techniques for precondition-driven quantifier instantiation during symbolic execution and for handing functions whose pointer arguments may alias are also applicable outside Bedrock2.

Chapter 4 explains how the Bedrock2 language and the RISC-V instruction set are modeled in compiler proofs and processor proofs, culminating in the "Lightbulb" case study [Erb+21]. A central innovation is the use of omnisemantics, which enable smooth modeling of programming languages with undefined behavior and nondeterminism. Only semantics-formalization design and integration-verification aspects are in-scope for this thesis: I am hoping Samuel Gruetter will write about his design and proof of the Bedrock2 compiler, and (with Arthur Charguéraud) we put forward an extensive, pedagogical presentation of omnisemantics using lambda calculi [Cha+23].

Chapter 5 first reviews Fiat Cryptography [Erb+19] and considers its verified integration with a combinatorial optimizer that targets x86 assembly and achieves competitive performance [Kue+22]. To connect field-arithmetic implementations from Fiat Cryptography to Bedrock2, a purpose-built certified compiler (by Jade Philipoom) is used to translate the straight-line code. The associated elliptic-curve operations, previously presented in my MEng thesis [Erb17], are translated to Bedrock2 by using proof-producing relational compilation (Rupicola [Pit+22]). The integration of these projects culminates in a proof that describes the behavior of the public-key-authenticated garage-door-opener demonstration in terms of RISC-V machine code, network packets, and standard definitions of elliptic curves.

1.2 Ecosystem Overview

Figure 1-1 shows the major components of the verified-computer-systems ecosystem. It contains formal specifications and compilers for five programming languages, with some nontrivial programs written in each. Functional programs and templates in Fiat Cryptography are proven against back-of-the-napkin specifications of elliptic-curve cryptography, appropriately instantiated, and compiled to Bedrock2 through dedicated pipelines for flat and structured code. They are then joined by similarly generated implementations of functionality such as IP checksums and handwritten drivers for GPIO, SPI, and Ethernet controllers. A certified compiler produces linked binary code for the RISC-V instruction set, which in turn can be executed on a verified processor proven against the same specification as the compiler. Numerous connections to unverified implementations of appropriate interfaces are available at all layers: for example, the same Fiat-Cryptography templates can be used to generate performance-competitive x86 assembly code through black-box optimization and certified translation validation.

I will now give some summary statistics to give a sense of the scale of development. My first commit in what became the main repository of this effort is from 7 years ago (September 2015, using Coq 8.4), and a slightly evolved formalization of the elliptic-curve-cryptography representation tricks from that time is a key part of a recently finished case study I describe here. Since then, 75 people have contributed code; 32 are attributed more than 100 lines and 16 more than 1000 lines currently

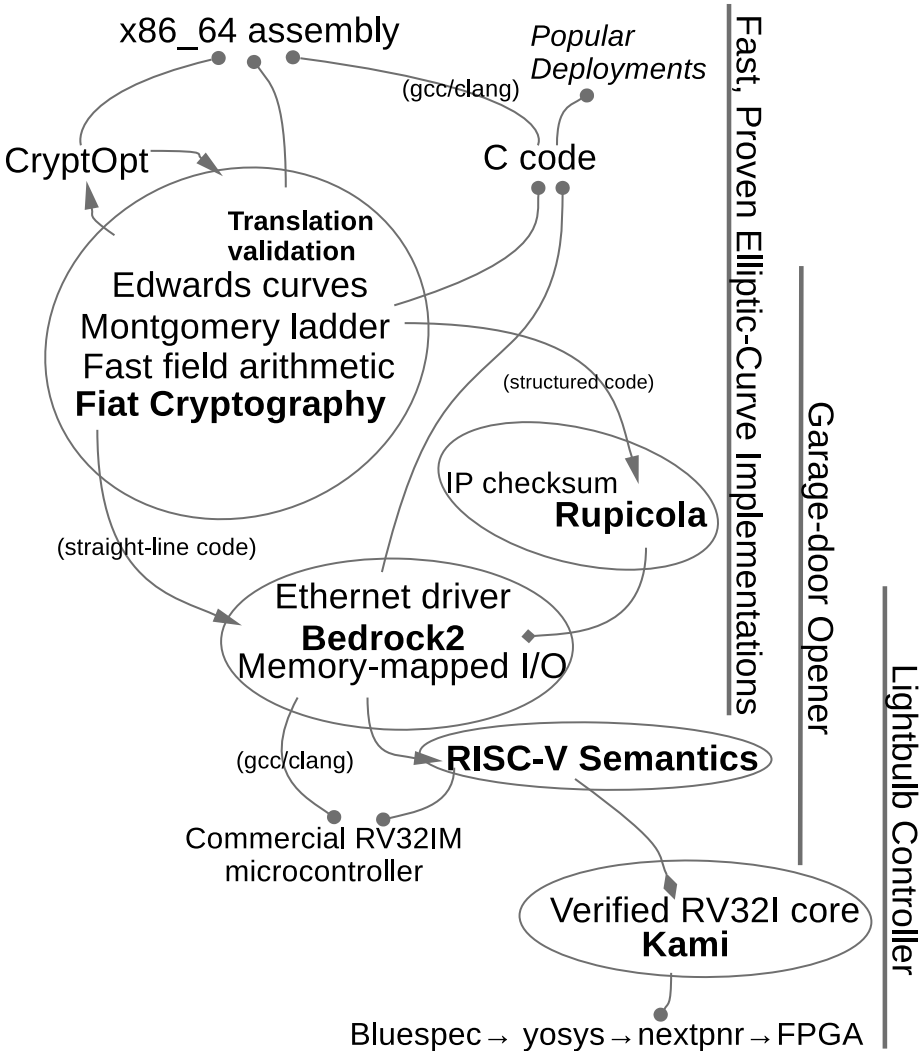

Figure 1-1: Diagram of components by implementation strategy (bubbles) and integration proofs (vertical spans). Non-boldface items inside component bubbles are examples of supported functionality. Arrows represent code transformations covered by foundational proofs; diamond-tipped arrows represent proofs without code translation, and circles represent unverified translations.

present in the codebase. The resulting ecosystem of proven and integrated systems components is now split across 7 repositories holding a total of 300k lines of Coq code from 18,848 commits (`cloc`). Each repository has some more-or-less independent use case outside the integration-oriented work described in this thesis; most notably, thousands of lines of Fiat-Cryptography-generated elliptic-curve implementations are used in web browsers, Linux, OpenBSD, MirageOS, Google and Apple products, and a dozen other contexts. A number of proof techniques mechanized as a part of this work are pushing the performance limits of the Coq proof assistant despite substantial optimization effort: Building a stripped-down version of our ecosystem is the most time-consuming task in the Coq continuous-integration benchmarking system, taking about a third (2h) of the total time and 5x longer than the four-color theorem. Components within this ecosystem have been described in 6 peer-reviewed publications, 2 nth-resubmission manuscripts, 1 PhD thesis, and 5 MEng theses. I am now writing about the common themes that make all this come together and tie each representative case study to one `Qed` against a satisfyingly thorough specification.

There are three case studies that I return to throughout this thesis: "lightbulb", "garage-door opener", and "CryptOpt". Please keep in mind that these case studies exist to show off the building blocks, not the other way around: while modeled after caricatures of real-world use cases, they might as well be Rube Goldberg machines demonstrating adequacy of and mastery over the pieces they are made of. With that out of the way, they do pretty much what they say on the tin:

- "Lightbulb controller" is an Ethernet server that turns a general-purpose output pin on and off in response to UDP packets. It is implemented in a Bedrock2, compiled to RISC-V using a certified compiler, and run on a proven-correct processor. Its theorem of (partial) correctness is stated in terms of the semantics of the hardware-description language and hypothetical logging of traces of memory-mapped I/O (MMIO) actions driving the network controller. Notably, the theorem statement does not depend on (the correct formalization of) the semantics of Bedrock2 or the RISC-V instruction set.

- "CryptOpt" is a search-based trial-and-error optimizer for straight-line x86 assembly, and Fiat Cryptography uses it as a backend. Certified translation validation is used to provide gap-free correctness theorems for CryptOpt-generated assembly code against specifications in terms of finite-field arithmetic, calling conventions, and semantics for x86 assembly.

- "Garage-door opener" is another Ethernet server, but it uses a Curve25519 Diffie-Hellman handshake to ensure that only the authorized user (identified by a public key) can control the actuator. Cryptography and networking routines are implemented and verified as functional programs but compiled to Bedrock2 code with manual memory management and executed on a commercial RISC-V microcontroller. Its correctness theorem is stated in terms of the semantics of

the RISC-V instruction set and hypothetical MMIO traces in a sense of always-eventually total correctness without internal carve-outs. This means that unlike what one would expect from generic implementations of functional languages, the compiler-produced binary is guaranteed not to run out of memory or stack space and will not otherwise abort or infinite-loop.

1.3 The Case for Integration Proofs

This section reviews the importance and pitfalls of specifying and modularizing computer systems in practice. In particular, the focus is on the possible consequences of subtle specification errors or misunderstandings. Most of the discussion isn't specific to formal verification, but the purpose of this investigation is to motivate studying and verifying integration of individually reasonable components. Similarly to how unit testing can miss serious bugs, proving correctness of single components or challenges distilled from them is not a guarantee against serious issues with the verified component. I chose the examples presented in this chapter to show the seriousness and pervasiveness of this consideration; not all of them are attractive targets for verification.

Integrated Correctness from the Perspective of Formal Methods Having a computer-checked proof about a system enables the proven properties to be reviewed based on the theorem statement alone, without auditing the implementation. The natural goal and expectation is that a verified program would only behave in ways described in its specification, implying that any observed misbehavior of the overall system must be the fault of some other component. For this principle to be applicable in practice, the specification of the verified component must cover *all* aspects of its behavior that the rest of the system relies on. This is a much higher standard than can be achieved by simply specifying desired behavior of the program in an idealized setting, and pursuing it rigorously can multiply the work required to verify a modest but highly interconnected component. Specifically, adhering to the requirements of the interfaces that a component uses can be as challenging, if not more challenging, than simply implementing the intended functionality, and proving a list of satisfied constraints does nothing to show that all requirements were listed.

An integration proof for some components conclusively resolves the question of adequacy of the specification of the interface between them: a theorem about the combined system can be stated without referencing the internal interface and remains valid even if the interface specification turns out to be inconsistent with some external standard. Verifying adjacent components in an integrated manner *decreases* the specification surface-area that needs to be audited and increases confidence in overall system correctness. Further, removing an interface from the trusted base un-

locks system-specification-preserving iteration on that interface, perhaps to enable optimized implementations of the components it delineates.

1.3.1 Systems Specification Challenges

While it is broadly useful to think of computer systems as collections of more-or-less independent components, this modularity is a product of careful design and engineering, not a given. Reliable operation of these components is interdependent, often inherently so, and some of the relationships are far from obvious. In particular, the correctness-dependency relationships of even the simplest systems are not stacks or trees, but rather the expected operation of lower-level components depends on higher-level components doing what is expected of them. For a concrete example, jumping to recently written memory without proper software interlocks would cause hard-to-predict behavior on any conventional multistage processor.

As a theme, internal interfaces of computer systems become vastly more complex than the user-facing functionality they support because implementers knowingly deviate from what an idealized specification would look like in order to simplify the implementation or to make it feasible at all. For each computer-systems interface contract there are numerous tempting, idealized versions of it that are not actually implemented. In tightly constrained contexts such as embedded, low-latency, high-throughput, or low-power devices, this kind of complexity is pervasive to the point that can be challenging to determine what a component interfaces with, let alone what obligations are associated with these interfaces. Yet failing to adhere to the actual requirements, even subtly, can result in arbitrary misbehavior, giving way to exceptionally nasty bugs and security vulnerabilities.

Example: Memory Allocation An ubiquitous and representative example of an idealized interface with context-dependent caveats is memory allocation. Programming languages above the assembly level provide built-in, almost or literally implicit mechanisms for finding an appropriate, unused memory location to store data. For example, C provides three: stack variables are assigned offsets within the function's stack frame during compilation, static variables are assigned addresses during linking, and `malloc` can be called to try to locate and set aside available memory during program execution. Details vary, but all three share the same important catch: each system only has a finite amount of memory. Yet nothing stops one from programming as if memory allocation always succeeds or proving properties of programs under this assumption. C with unlimited memory is convenient but unimplementable.

But what actually happens when available memory cannot be found? Without much more detailed knowledge about the rest of the system, it is hard to be sure. The specification allows `malloc` to return `NULL`, but some implementations terminate the program or return "overcommitted" virtual memory whose use may ter-

minate the program. Static allocation lacks the first option. Stack allocation is further severely performance-constrained and commonly returns some memory without checking whether it overlaps already allocated memory. The last scenario can result in arbitrary code execution through stack-pointer hijacking, necessarily violating any properties worth proving about the program. Accepting the intermediate options by reading "or it may crash" into every specification is deeply dissatisfying and unacceptable in safety- and security-critical applications.

Takeaways The correctness of the program and the memory-management system are interreliant: The program, verified or not, allocating more memory than the underlying system can provide will cause the system to fail to execute the program in a manner that could be specified in terms of the source language. The contract between them is intricate: memory fragmentation means that allocating `n` contiguous bytes may fail if more than `n` bytes are available, and all but the coarsest fragmentation models are allocator-specific. Ignoring the problem entirely is tempting, but the potential failure modes are not acceptable in general.

Designing an interface specification that appropriately delineates responsibilities between components is inherently challenging, and doing so without a reliable means to check against actual implementations on both sides of that interface is bound to miss details. For example, the Verified Software Toolchain (VST) [App+20] lets programs that stack-allocate more memory than fits in the address-space to be proven correct, and Bedrock `malloc` [Chl13] callers can satisfy *any* specification[1] in this scenario because `malloc` enters an infinite loop and the proof framework only guarantees partial correctness. On the other hand, the VST `malloc` implementation provides an interface with fragmentation-aware tracking of available memory [AN20, §4], opening the way to proofs of correctness under execution with bounded heap memory. The case studies in this thesis handle bounded stack memory.

A Note on Mitigations If terminating the program *is* acceptable, it is often possible to replace other failure modes with it, but doing so is not free. Common mitigations for stack-heap collisions include marking a last region of dedicated stack memory as inaccessible, explicitly checking for free memory before growing the stack, or checking the integrity of important stack and heap data structures before using them. Measures of this kind have been attributed considerable success in preventing attacks in practice. From a formal-methods perspective, they are components like any other and subject to the same challenges.

For example, SoftBound+CETS [Nag+10] is a particularly strong mitigation for

[1] Concretely, calling `malloc` to allocate 2^{31} words gives a postcondition promising to fit all of them in memory, but `base =?> (2*N.to_nat (Npow2 30)) ===> [|False|]`. To show this, rewrite `2*` to `+`, distribute `=?>` over `+`; the first cells of the two resulting arrays are at the same word address.

memory-access errors in C: operating as a compiler pass, it translates every program that potentially triggers undefined behavior only through out-of-bounds memory access into an ABI-compatible program that potentially crashes instead, at a ~2x performance overhead. A model of its implementation in LLVM 3.4 was proven correct in the Coq proof assistant against a model of a core subset of C. This combination of generality and confidence made a splash amongst computer-security enthusiasts and inspired a celebratory third-party presentation of the work at CCC 2013 under the title "bug class genocide" [Bog13]. After explaining the benefits of a computer-checked proof, the speaker left a particular snippet of the verified code on the slide for an awkwardly long pause and then commented "if you're not laughing, I don't trust your C code": the proof against an idealized model of C where addresses are natural numbers had missed a classic integer-overflow vulnerability in the mitigation's array-bounds checks.

1.3.2 Interface-Straddling Bugs in Unverified Systems

Insisting that the components on opposite sides of an interface agree on and abide by the rules for how to interact over that interface is not just about computer-checked proofs. Confusion or disagreement over the details of some systems-programming interface has been behind a number of serious bugs in real systems. This section will sample a representative set of examples from different interfaces in popular systems.

Intel `sysret` Operating-systems code written for AMD's 64-bit x86 processors by straightforwardly following the specification allows for privilege-escalation attacks on Intel's later 64-bit processors due to a subtle difference in how the `sysret` instruction behaves [Dun12]. It is believed that this difference is unintentional and that the instruction with encoding and mnemonic matching AMD's was also meant to match the behavior. Intel engineers dutifully wrote their own specification and made explicit the interaction with noncanonical addresses, a rather obscure feature. However, the behavior that is spelled out (and implemented) does not match AMD's.

As a tradeoff for page-table compactness, AMD defined virtual addresses on 64-bit x86 to be 48-bit signed integers, sign-extended to 64 bits. To keep the options open for later extension of these addresses, AMD specified that all instructions operating on virtual addresses must fault (redirect control flow to an exception handler) if the address does not satisfy this constraint. A typical kernel or hypervisor returning from a system call first sets the stack pointer and then calls `sysret`, which sets the instruction pointer. If the new instruction pointer is not canonical, `sysret` faults. However, AMD's version first drops from kernel mode to user mode and then faults, whereas Intel's version faults in kernel mode. The latter will result in a kernel-mode fault with the stack pointer controlled by the supposedly unprivileged process. The processor pushes data onto the stack when handling the fault, overwriting kernel memory at a user-controlled address and allowing code execution to be hijacked.

While Intel's `sysret` behavior is clearly undesirable, whether or not working as specified, it is also workable: the software can check the new instruction pointer and emulate AMD's behavior if necessary while still enjoying the performance benefits of a dedicated system-call return instruction in the common cases. The real tragedy is that this problem remained unknown to most operation-system maintainers for almost a decade, leaving Windows, Xen, FreeBSD, and NetBSD vulnerable to straightforward privilege-escalation attacks. Specifying something, even with unambiguous authority, is not enough to achieve reliable system integration, and having an appealing but subtly different specification already established unsurprisingly leads to the caveats being missed.

Stack Allocation and Processor Interrupts Another consequence of x86 kernel-mode interrupt handling is that data stored at stack addresses where the next few push operations would place it can be overwritten at any time if the stack pointer is not adjusted correspondingly. (Even if the interrupt handler returns to the interrupted code right away, the processor pushing interrupt information to the stack has already overwritten the data.) This is different from interrupt handling during user-mode execution which instead pushes exception state to a separate stack whose address is stored in task state segment, and the System V x86-64 ABI specification mandates that operating systems shall not modify the next 128 bytes to which stack could grow either.

A foreseeable consequence of this difference is that assembly code written for Linux userspace may misbehave arbitrarily when executed in the kernel (perhaps in an effort to benchmark it without context-switching noise). Less obviously, compiling source-code programs for x86-64 with the Linux calling convention does not mean that the compiled code is usable in the kernel. Compilers (e.g., GCC) can and do store leaf-function temporaries in the "red zone" past the stack pointer unless `-mno-red-zone` is specified, leading to hard-to-debug data corruption and crashes when the flag is omitted [Dzy14].

This caveat is particularly easy to miss because it violates an appealing but incorrect perspective that the application binary interface is a specification of how a component can interface with other code and does not influence its implementation internals. This perspective is otherwise workable. For example, the potential considerations listed on the Wikipedia page on ABIs[2] can each be categorized as being about instruction-set capabilities or data-layout conventions. While "stack organization" does appear, nothing calls out that it can be a relevant inside module boundaries.

The case of `memcpy(NULL, NULL, 0)` Taking interface specifications seriously, usage that does not follow the requirements of the specification would be considered

[2] `https://en.wikipedia.org/wiki/Application_binary_interface`

buggy regardless of whether the implementation of the component on the other side of the specification actually relies on the specific rule. This seemingly conservative stance can have drastic consequences when programming tools understand and rely on it. Common implementations of `memcpy` have no trouble copying zero bytes from any address to any address, but the C standard decrees that calling any function defined in it with a `NULL` pointer triggers undefined behavior. Relying on this rule, compilation with GCC removes branches checking for `NULL` value of pointers passed to `memcpy` – if the pointer could be `NULL`, the program would be undefined anyway.

There are no known benefits to this optimization in particular, and it is not clear whether `memcpy` specifically was considered when adding it. The risks are substantial: removing `NULL`-pointer checks for much less contrived reasons has resulted in serious security vulnerabilities [Cor09]. And even if the optimization had benefits, it is not obvious that it is desirable overall as it "adds a very subtle, exceptional case to several very common functions, burdening programmers" [Lan16]. Taking a step back, the quandary here is not just about the specific precondition but also about whether the call to `memcpy` has obligations to the compiler in addition to the callee. In principle, either answer is workable (by creating a friendlier `memcpy` *with a different name*[3]), but a broader conclusion appears more salient: Specification-aware tooling raises the stakes for getting specifications right.

Conclusion The challenges discussed in this section persist in spite of earnest efforts of engineers and designers. The complexity of computer-systems interfaces combined with the possibility that a component may critically rely on a seemingly minute specification detail can make implementation work tricky and unforgiving. Careful documentation and review of the precise requirements can go a long way, but this approach does not have an answer for how to get from from checking every known concern to having confidence that no caveats have been missed. Proof or verification of a component against a specification of that component's interfaces is not enough to catch these issues either, and dismissing this possibility as a specification bug misses the point: For delicate internal interfaces, the only relevant notion of specification correctness is that it should appropriately delineate the responsibilities of the relevant components.

Setting out to prove integrated correctness turns this consideration around. Proofs of tricky interactions are expected to be challenging, but once a theorem stated *without* referencing an internal specification is proven, uncertainty about expectations relating to that interface no longer threatens the overall result. Cases where adjacent components are relying on independently reasonable but subtly different assumptions about the interface between them will show up during attempted proof, and any consistent resolution should unblock the overall proof. Further, potential changes to an interface can be validated by reestablishing the same system-level result.

[3] `https://boringssl.googlesource.com/boringssl/+/17cf2cb1%5E!/`

1.4 Modularity and Proof Assistants

If interface specifications are so delicate and challenging, it makes sense to consider whether treating them as a cornerstone of the system is worth the trouble. Creating an implementation of common functionality (say, `memcpy`) that works for the cases where it is actually used may be reasonably considered easier than precisely describing when its use is valid. Further, not having a specification would avoid the temptation to transform code under the assumption that it conforms to that specification, avoiding the chance of introducing bugs due to differing expectations for that specification. Thus, it makes sense to consider avenues for all components of a system together could be verified against a top-level specification.

1.4.1 Automated Verification Using SMT Solvers?

In the last decade, there has been considerable progress in software verification methodologies that do not rely (or barely rely) on interface specifications. A commonality between the methods suitable for integrated verification of nontrivial implementations is the use of carefully-tuned symbolic execution for a common-denominator language (e.g., assembly) and SMT solvers. Possible control-flow paths from specified entry points are explored until program termination is reached, or until the SMT solver detects a conflict between the branch conditions required for the path to be taken. If all path are explored until the path condition is unsatisfiable, the overall specification can be checked on each completed execution to conclude correct behavior in all scenarios.

In addition to being able to think less about specifications, these approaches are appealing due to the high level of automation offered. Code changes that stay within the supported functionality of the symbolic execution engine can be tested out with no additional proof effort, and simple mistakes can often be caught quickly as the solver finds a case where the negation of the desired property is satisfiable. Efforts in this style include the Hyperkernel project [Nel+17], the Nickel information-flow-checking tool [Sig+18], Serval [Nel+19], and Vigor [Zao+19], which is specifically aimed at network-facing functionality like the main case studies in this thesis. Tools operating on assembly (or LLVIM IR or machine code) can potentially be used on code compiled from different languages in one verification unit, verifying absence of integration issues in the combined artifact.

Designing the tools around a specific automated verification procedure is also responsible for a number of limitations of these methodologies. The most pressing is that it can be challenging to tell why verification does not succeed in a reasonable time, to the point that a trial-and-error approach of creatively interpolating between the desired verification task and known feasible tasks is often recommended. This difficulty constrains the designs that can be verified automatically. For example, the Hyperkernel [Nel+17] authors write:

> In particular, we make adjustments to keep the kernel interface finite, by ensuring that the semantics of every trap handler is expressible as a set of traces of bounded length. To make verification scalable, these bounds should be small constants that are independent of system parameters (e.g., the maximum number of file descriptors or pages).

Humans reviewing operating-systems code are not thwarted by the possible number of file descriptors or pages because the reasoning about these resources is encapsulated in a modularly specified interface: the same rules apply to every file descriptor, regardless of its number, so their implementation can be checked only once. Being oblivious to interfaces, the fully automated tools are unable to benefit from them to keep verification effort (asymptotically) under control.

The only projects I know to have applied highly automated SMT-based verification to the integration of hardware and software are Notary [Ath+19] and later Knox [AKZ22]. In Notary, the verification goal was to show that the reset routine for returning a multi-app secure transaction approval to a clean state properly scrubs all information. The Knox framework was used to verify simple HSMs such as a PIN checker and a TOTP token against deterministic specifications under arbitrary input and the possibility of power interruption, showing functional correctness and absence of information leaks. Both projects use an extremely minimal RISC-V processor, PicoRV32 [Wol15], which frequently takes several cycles to execute an instruction. The Knox paper describes several optimizations that were required to get verification of the case studies to succeed, including deviating slightly from the fully automated verification paradigm by requiring proof hints to guide proof search. Lack of abstraction is again pointed out as reason for duplication of verification effort: "The relatively low performance of verifying PIN-protected backup is due to performing case analysis on the slot number, which causes many paths to be explored independently".

Verifying components whose implementations and specifications have high-level algorithmic differences seems to be out of reach for a purely automatic approach. Even projects that take pride in a high level of automation choose to rely on guided proofs against handwritten specifications to verify algorithmically challenging components. For example, Vigor [Zao+19] uses Hoare-logic-style proofs of important library routines to summarize them soundly in symbolic execution. However, using separate tooling to verify different components raises the question of how to ensure that the tools understand the interface between these components in a sound and consistent manner. The challenges for accurate specification discussed in Section 1.3 in the context of component implementation are equally applicable to verifications tools designed specifically for these components. To reap the benefits of integration verification, drawing an interface boundary through a system needs to be treated with the same rigor as an intermediate assertion during program verification: as a part of the proof, not the theorem.

1.4.2 Specifications Enable Proof Reuse

Even if highly automated verification methods were able to check sizable systems against fine-grained specifications that differ substantially from the implementation, the need to check every combination of components as a whole can be considered a significant limitation on its own.

Being able to develop and prove a module by itself is a significant flexibility benefit. A large fraction of the development effort of a computer system can be clearly attributed to a specific component and can be pursued effectively without revisiting other components or before finishing them in the first place. Further, changing out components at formalized boundaries is simplified: a comprehensively verified software package can be ported (with proof!) to a new processor proven to implement the same instruction set, and a software library can be updated without concern for which processor it was verified against. Troubleshooting is also simplified: a failed verification attempt focused on a single component cannot be due to a bug (or verification-evading correct code) in another component. The computational resources required for verification can be tracked and budgeted by component, without a concern that independently supportable changes in two components drastically increase the computation required for overall verification when combined.

This workflow difference is especially drastic from the perspective of developers of popular components with rich and flexible interfaces, for example programming languages, operating systems, and processors. Verifying each incremental modification to the system against every program that uses it is clearly infeasible for the same reasons current projects do not consider similarly wide testing a worthwhile goal. A diverse test suite can go a long way towards catching serious issues, but even thoroughly tested programming-language implementations such as SQLite are still regularly found to have bugs through other means. Issues that do escape pre-release testing but are caught independently by full-system verification by a downstream user are likely to be very difficult to diagnose for the user who is not an expert in programming-language implementation. Further, when a user of the compiler inevitably returns to the compiler maintainers with a complaint, the (failing) whole-system verification does nothing to confirm that the bug in fact lies in the compiler as opposed to the use case. All in all, while fully automatic automatic verification could give confidence in the correctness of a finished system, it does not help to build components for use in correct systems.

From the perspective of looking at a finished artifact, a proof relating a human-readable specification to each module can always be studied as the sum of its parts. Knowing the expectations that the rest of the system has for a component can be used to interpret its design choices: For example, a RISC-V processor that does not implement an instruction-fence operation can either always provide a coherent view of the memory between instruction and data accesses, not support self-modifying code, or require a nonstandard synchronization mechanism. Looking at the memory-related

precondition for fetching an instruction should make this difference clear, even if the implementation details are scattered throughout the subcomponents.

Finally, using an earnest model of an established interface in a modular integration proof can yield confidence in the formalization of that specification itself. Orthogonally from whether the specification matches existing informal descriptions and correctly handles specific test scenarios, it is valuable to know that the specification as written is sufficient to support satisfying proofs for both an implementation and a user of it.

1.4.3 Integration Verification in a Proof Assistant

Somewhat counterintuitively, general-purpose proof-assistant software that interprets definitions and theorem statements and checks proofs against them stands out as uniquely appropriate for proving deep properties about algorithmically nontrivial computer systems and components. The pattern behind this fit is that the same facilities used to support intricate mathematical constructions internally to proving simple-to-state theorems lend decently well to specifying intricate interfaces in a computer system. General lemmas about these interfaces can connect different perspectives on the same functionality, and software-specific proof techniques can be formalized similarly to mathematical-reasoning algorithms like polynomial manipulation.

I would like to stress that the utility of a foundational proof assistant for integration verification extends beyond defining a minimal interface for proofs against which verification attempts by domain-specific solvers can be checked. It is the very ability to model (and effectively reason about) programming languages and instruction sets internally to the system that sets proof assistants apart from domain-specific tools and allows them to be used to check that the modeled interfaces are well-defined and used consistently. Further, the understanding of definitions and their relationships already required for proof checking enables rigorous specification-auditing functionality such as finding all definitions that the statement of a theorem depends on.

The separation between proof finding and proof checking is also important for the way proof assistants are used in software verification: it enables creation and incremental improvement of domain-specific proof procedures, more or less similar to those available as standalone tools, without risking unsoundness. Extending functionality of proof procedures is common and encouraged when working with a proof assistant, while using nonfoundational tools it is a mark of merit to say that a program was verified using a standard tool that can also be audited by others. While proof assistants themselves are not immune to bugs either, the core features that are used throughout a wide variety of proving are substantially more stable than the domain-specific proof tooling they support.

A key benefit of having a common environment in which different proof procedures can hand verification tasks over to each other is that adjacent steps in the same proof can be dispatched using independently developed provers. The set of conventions covering givens (variables and assumptions), goals, and existential variables to be solved for during the proof is known as the interface of the proof engine, and (most) proof procedures are implemented in terms of incrementally checkable steps such as introducing a universal quantifier from the goal to the context or replacing a variable with its definition. If a proof procedure fails during an automated proof, the same goal can be retried using a different strategy or left over for manual inspection and proving later. As each step of the same automated proof strategy can be performed manually, it is straightforward (although sometimes tedious) to identify the step that did not work as planned and to update the automated prover or to finish the proof by hand.

This incremental interface responsible for the success of interactive theorem-proving comes with substantial costs. First and foremost, proof assistants are built for *interesting* proofs about elegant objects, be they about pure mathematics or programming-language theory. This means that performance throughout numerous "easy" proof steps on a large but monotonous program has not been a development priority, and it is not uncommon to run into cases where a seemingly simple proof takes minutes or even hours. Second, taking definitions, quantifiers, and scope seriously presents some genuine algorithmic challenges, situations where naive approaches to incremental verification do not achieve asymptotically desirable performance. However, unlike SMT solvers tackling undecidable or NP-complete problems, the performance challenges are amenable to classical algorithmic approaches. Much more detail on performance engineering in a proof assistant, including specifically for Fiat Cryptography, can be found in the dissertation of Jason Gross [Gro21].

Why Coq? As of the start of my work on integration verification, Coq was the only proof assistant with substantial precedent for verification of software written in other languages using sophisticated proof automation. Further, Coq uniquely provides (trusted) high-performance evaluation mechanisms for executing compilers and decision procedures inside the logic, which both Fiat Cryptography and Bedrock2 rely on. The same basic criteria hold true today with some qualifications, and a much larger ecosystem of engineering-oriented verification projects has taken off [App22], bringing with them general proof-assistant improvements to meet their needs. I did consider Lean2 and Lean3 and found a number of the design choices based on hindsight from Coq to be genuine improvements, but the focus on mathematics and lack of a fast evaluation mechanism dissuaded me from pursuing a project with them. I later heard that HOL Light is also a worthy candidate, but I never seriously evaluated it.

1.4.4 Possibilities for Mixing Tools

Dedicated stand-alone verification tools for specific domains such as C-like programs are currently associated with better proof-performance scaling expectations than similar foundational tools built on top of a general-purpose proof assistant. This perspective is consistent with the prevalence of surprising proof-performance challenges in the Coq projects I have followed, including every major component discussed in this thesis. For example, the verification-effort bottleneck that limited the scope of the final integration-verification case study in this thesis is closely related to Bedrock2's arithmetic-proof-performance issues: a faster proof procedure for array bounds checks seems necessary for guess-and-check automation of memory-hypothesis selection (see Section 3.4 and Section 3.5) To contrast, the VCC memory-model paper [Coh+09] describes verification of a 10000-line test suite and a 500-line library about doubly linked lists where no function took more than 20s to verify on a computer available in 2009. This example does not seem out if line: I am not aware of any proof-assistant-based verification tool that outperforms a similar stand-alone tool from a decade earlier. (I am also not aware of any principled explanation for why there would necessarily be a difference of performance between foundational and nonfoundational tools or of any systematic evaluation of the magnitude of this difference, but the perception stands.)

Thus, realizing an integration-verification methodology using appropriate stand-alone tools appears very attractive from the standpoint of proof-time performance using current tools. Achieving this goal without compromising confidence in the soundness of the approach seems challenging, but there are a number of possible tradeoffs that may be suitable depending on nontechnical circumstances. The primary reason to use a general-purpose proof assistant for integration verification is to ensure that specifications of interfaces are defined in a noncontradictory manner and used consistently from both sides. For evaluating alternative strategies, it is important to consider what they rely on to ensure the same guarantee.

Stand-Alone Verification-Condition Checking For example, one could consider writing a programming-language specification and a certified compiler in a proof assistant but implementing a dedicated verifier for programs written in this language. In this general scenario, a relatively rigorous approach would be to structure the verifier as a verification-condition generator that returns assertions in a relatively simple and established language such as SMT-LIB along with an assertion checker for this language. This way, the verification-condition generator can be proven sound against the semantics of the programming language and the assertion language. (However, it is reasonable to be concerned that this proof itself may run into proof-assistant performance challenges.) The Leapfrog project [Doe+22] is one example of a Coq project relying on an SMT-LIB prover. Use of an unverified prover could be considered analogous to Coq projects' ubiquitous use of the fast evaluation mechanisms based on compilation to bytecode or native code: both rely on an unverified procedure to ascertain a fact stated within the logic of the proof assistant.

Although the low-level implementations may be prone to bugs that do not plague a general-purpose proof-checker, one can argue that "reasonable" assertions generated from programs whose verification is desired would be unlikely to generate inputs that accidentally trigger these bugs. Note that even taking this distinction of "reasonable" inputs for granted, either strategy that relies on a proof tool with a history of soundness bugs crucially depends on a high level of trust towards the source of these inputs: an input that triggers a very unlikely bug could be engineered maliciously. Given this concern, a security-conscious project such as a popular cryptography library considering contributions from strangers would still benefit from increased confidence in the proof-checking process. One way to achieve this would be to have the SMT-LIB prover generate proof certificates that can be checked by the proof assistant, but current work in that direction does not appear ready for program-verification use[4] [Doe+22, §6.3]. The desire for this kind of checking highlights the assurance advantage of Coq's virtual-machine evaluator even if its implementation is not considered rock-solid: the fast evaluator is primarily used to run proof-certificate checkers that ship with Coq itself, and the same proofs can be checked with a simpler evaluator.

Separate Verification Across a Rich Interface The performance of SMT-based tools depends heavily on SMT-solver-friendly encoding of verification conditions, and the backends of tools such as VCC rely on an assortment of optimizations to achieve satisfying performance. Based on this, it is far from clear how easy it would be to port an efficient verification-condition generator to a proof assistant and to prove it correct. It seems reasonable to speculate that doing so would likely be easier than designing a general-purpose proof engine that can allow for incremental processing of individual proof steps inside a proof assistant, but I don't know how this prediction could be justified. Skipping the detour of proving the verification-condition-generator that just hands these verification conditions over to an unverified prover would allow the effort to be saved altogether. However, interfacing between the proof-assistant-checked development and the program-verification tool at the semantics of the programming language is more challenging to audit because this interface is much richer and more delicate than a pure expression language such as SMT-LIB.

For example, the VCC memory model [Coh+09] is described as follows:

> The main difference is the goal: we aim at a sound verifier for complex functional properties with whatever annotations are necessary [...] allow for arbitrary changes of type assignment at runtime, which is needed to prove correctness of components like memory allocator but also for something as simple as implementation of byte-copy of a struct.

This description seems to match Bedrock2's perspective but differs from a number of

[4] `https://github.com/smtcoq/smtcoq/issues/72`

formalizations and implementations of C discussed in Section 2.4. Compiling VCC-verified code with a certified compiler proven against a semantics that considers types of declared variables to be fixed and (as an "optimization") deletes code that accesses the same location using a different type would likely break the soundness guarantees expected from VCC.

Avoiding this kind of interface-mismatch bugs is the central goal of integration verification, and leaving an interface as complicated as a stateful low-level programming language at an unverified boundary between verification tools appears to substantially increase the risk of missing an issue. From the perspective of trust required in the development processes, the requirement in this case is much stronger than a lack of malice. Program-verification tools and optimizing compilers have drastically different engineering considerations and are evaluated against very different metrics: naively, perhaps annotation overhead and verification-time performance for the former and compile-time performance and run-time performance for the latter. Reaching, confirming, and maintaining precise agreement about the same interface throughout a healthy level of developer and management turnover requires reliably rejecting tempting optimizations that would deviate from the agreement even if issues from these deviations would not be immediately apparent. The very addition of type-based memory-access restrictions to C itself (see Subsection 2.4.4) is a great example of how challenging this technically trivial task can be in practice.

The work covered in this thesis establishes a new high-water mark for the complexity and realism of interfaces and components whose integrated correctness is proven within one proof assistant, without any formality gaps. Pushing this limit further (for example, to cover concurrency, latency, or side channels) seems valuable regardless of proof automation, but substantial improvements in the performance of proof tools may be required even just to support the examples that exercise these extensions, let alone industrial adoption. Simultaneously, a lot can be learned from development, optimization, and application of stand-alone verificaton tools and their use in soundness-conscious but potentially less technically integrated verification of multicomponent systems. However, it would be shortsighted to disregard either the integration of component-specific verification strategies or their performance as an engineering detail or to leave these considerations as exercises to the reader. In as much these two concerns correspond to what a verification tool guarantees and at what cost, one must always be evaluated in the context of the other.

1.4.5 Past Integration-Verification Projects

Three prior projects demonstrate integration verification across the software-hardware boundary. They all do so by connecting all components within one proof assistant, thus reducing the trusted audit-worthy code base to just their top-most and bottom-most specifications and the proof assistant.

In the late 1980s, the CLI stack [Bev+89] 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 [Hun89]. 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 [Alk+08], begun in the early 2000s, connects a 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 concise application specification in terms of input and output. The closest we are aware of related a correctness proof of a small automotive-control C0 application to the correctness proof of an operating system [DSS10], plugging into a proved stack including compiler and processor, but there is no discussion of a short full-system theorem, even though each interface individually seems to have been crossed for non-I/O code [Tve09].

In work begun roughly 15 years after the Verisoft project started, the CakeML optimizing compiler [Kum+14] was extended with a backend to a new, purpose-built instruction set called Silver [Löö+19]. This time the software stack did support input and output, but the complete stack still did not. Instead, external calls for file-system access and standard input/output were compiled into reads and writes of a memory buffer. The stack was run on an FPGA, 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 were executed: word count, sorting, and even compiling a "hello word" program using a cross-compiled copy of CakeML itself.

All three systems mark significant milestones in systems verification, but they rely on simplifications that clearly distinguish them from any deployed system. In an effort to bring gap-free correctness assurance to practice, I sought to build and verify plausibly deployable and useful systems and followed an established architectural choices at a high level. Additional functionality in my work includes interactive input and output, driving an off-the-shelf network controller, and cryptographic authentication. Behind the scenes, I am pleased to report that the components integrated are independently valuable in their own right, making for a showcase of modularity in action.

Chapter 2

The Bedrock2 Programming Language

A central interface in the integrated-verification ecosystem and the demonstrations is the Bedrock2 programming language. This chapter will give an informal overview of the language and associated engineering considerations. Support for Coq proofs about Bedrock2 programs is discussed in Chapter 3. The two are far from orthogonal: The design choices deliberated in this chapter, in particular the focus on simplicity and nonredundancy of the language's demands from the programmer, will become the foundation for a tolerable program-proof experience. Nevertheless, I will refrain from referring to particular proof methods in this chapter and argue for minimizing obligations placed on the programmer as an appealing goal in its own right.

Bedrock2 is modeled after C as understood by Linux and OpenBSD developers, and it draws from the integration-verification experience with original Bedrock [Chl11; Chl13; Chl15], Cito [WCC14], and Facade [Wan16; Pit+20]. Like Bedrock, Bedrock2 is intended to be used inside Coq, primarily for implementation of programs that will be verified and with program-proof flexibility as a central design priority. Unlike its in-Coq predecessors, Bedrock2 does not leave the world of unverified programming behind altogether. I made an effort to have Bedrock2 provide acceptable usability for unverified "portable-assembly-language" programming and, centrally, for creation of verified programs that call and are called by unverified programs. Fine-grained interoperability allows verified Bedrock2 programs to be included in existing C code bases, and being able to reuse existing low-level-programming interfaces whenever they make sense has allowed for componentwise prototyping of software eventually implemented wholly in Bedrock2.

The drives of integrated verification and simplifying systems programming appear more aligned than not in a number of concrete questions discussed in this chapter. However, the design of a programming language has to balance the needs and prefer-

ences for programming with the needs and preferences for compiler implementation. In particular, writing programs that manipulate memory at computed addresses is essential for efficient memory management, but a compiler cannot support arbitrary memory modifications that would overwrite data structures such as the control stack that are used to implement the language itself.

Precisely and satisfactorily specifying the conditions for accessing memory from a source-language program is challenging: in spite of the popularity of C programs and implementations of C, there is no consensus about the appropriate rules for memory access in a C program. Existing formal and informal specifications of C are written primarily based on the requirements of either the implementation or programming, with varying degrees of acknowledgement of the specified rules' impact on the other. To support modular but soundly integratable proofs for source programs and the compiler, Bedrock2 has to specify memory-access rules and other corner cases in a manner compatible with all code on both sides of the language interface. Thus this chapter will discuss fundamental design decisions such as what kind of state the execution of a program depends on and against which both the programs and the compiler are proven.

Consider the C function `memmove(void *dest, const void *src, size_t n)`: it copies `n` bytes from address `src` to `dest` and works even if the source range and destination range overlap. This function is a part of the C standard library for hosted implementations, and C compilers for freestanding implementations often require an implementation to be provided by the user. The implementation, usage, proof, and compilation of the corresponding Bedrock2 function rely on a number of aspects that are not commonly granted by compiler-derived C specifications.

- First, `memmove` compares `dest` and `src` to determine whether the input range potentially starts or ends within the output range. Evaluating this comparison without requiring that `dest` and `src` point to memory returned by the same allocation requires comparisons of all pointers to be allowed by the language.
- Then Bedrock2 `memmove` proceeds to copy the input to the output one byte at a time in an order that ensures that each address is read before it is written. This strategy relies on consistency between previous pointer comparison and subsequent loads and stores.
- The proof of `memmove` has to relate each byte written to `dest` to the corresponding byte read from `src` even if `src` points to a pointer. Any predicate that held about the input should hold about the output, so the language cannot make memory state depend on the type of the operation used to write that memory.
- Usage of `memmove` frequently features scenarios where the written memory is later read in units larger than a byte, for example initializing a dynamically allocated integer array from a network packet. For this to work, the language

implementation must not restrict access to the written memory to bytewise operations on the basis that its contents were copied from a byte array.

The description of Bedrock2 presented in this chapter represents one possible answer to these considerations. In short, the design is based on concrete and minimal state of the kind one might inspect using a debugger, but the evolution of this state during basic operations such as memory allocation is substantially underspecified to retain flexibility of compilation. The resulting model meets the needs of programs and compiler optimizations implemented for the Bedrock2 case studies described in this thesis but does not support all optimizations implemented by popular C compilers. Thus, the desire for fine-grained interoperability between Bedrock2 and C code is further in tension with the desire to use Bedrock2 for low-level programming including the implementation of its own basic library functions. Treating Bedrock2 as a subset of C would risk miscompilation of these functions, and restricting interoperable usage based on a best-effort model of the requirements of other compilers does not appear practical. Instead, I present a translation from Bedrock2 code to C that only relies on relatively uncontroversial aspects of the C language.

2.1 Syntax

At a glance, most Bedrock2 programs are completely unremarkable and ascetic. For example, a binary multiplication implementation appears below. (The return variable is named next to the arguments and implicitly returned at the end of the function.)

```
Definition rpmul := func! (x, e) ~> ret {
  ret = $0;
  while e {
    if (e & $1) { ret = ret + x };
    e = e >> $1;
    x = x + x
  }
}.
```

The concrete syntax above is verbatim from a Coq file, where it is handled by the relatively recent Coq feature for notations with independent grammars in custom entries. As an early adopter of this mechanism, I worked with Hugo Herbelin to identify, understand, and resolve limitations in its earlier versions. The corresponding abstract syntax treats variable names as strings; identifiers inside Bedrock2 programs are converted at elaboration time using Ltac2. Being able to define custom parsing and elaboration rules directly inside Coq obviates the need for a dedicated preprocessor or macro system and allows different Coq files to use different concrete notations for writing down Bedrock2 abstract syntax trees. The notation shown here is commonly

used, but the codebase includes some cases of nontrivial desugaring and expansive macros – `$` can be followed by an arbitrary Coq expression, not just a number.

In common usage, each Bedrock2 program is immediately accompanied by a human-readable and machine-readable specification of supported usage. For example, `rpmul` does not interact (`t`) with the external world, leaves memory (`m`) unchanged, and returns the product of its arguments.

```
Instance spec_of_rpmul : spec_of "rpmul" :=
  fnspec! "rpmul" x e ~> v,
  { requires t m :=
      True;
    ensures T M :=
      T=t ∧ M=m ∧ unsigned v = unsigned x * unsigned e mod 2^32 }.
```

The formalization of the meaning of these specifications is presented in Section 3.2; for now, it is just an example of expectations one might have of the executions of a Bedrock2 program. (This particular program was used in an exception handler to emulate the multiplication instruction on a plain RV32I processor, and the system was proven to correctly implement the specification of the encriched instruction set [Bou+21].)

Abstract Syntax The abstract syntax of Bedrock2 is minimal, stable, and strictly structured to simplify proofs of high-assurance implementations and reduce the risk of bugs when pretty-printing Bedrock2 as C. Most abstract-syntax constructs have direct analogues in C, but there are a number of deviations that warrant commentary.

```
Inductive bopname := add | sub | mul | mulhuu | divu | remu
                   | and | or | xor | sru | slu | srs | lts | ltu | eq.
Variant access_size := one | two | four | word.

Inductive expr :=
| literal (v : Z)
| var (x : string)
| load (s : access_size) (addr : expr)
| op (op : bopname) (e1 e2 : expr)
| inlinetable (s : access_size) (table : list byte) (index : expr)
| ite (c e1 e2 : expr). (* if-then-else expression ("ternary if") *)

Inductive cmd :=
| skip
| set (lhs : string) (rhs : expr)
| store (sz : access_size) (address : expr) (value : expr)
```

```
| cond (condition : expr) (nonzero_branch zero_branch : cmd)
| seq (s1 s2 : cmd)
| while (test : expr) (body : cmd)
| stackalloc (lhs : string) (nbytes : Z) (body : cmd)
| call (binds : list string) (function : string) (args: list expr)
| interact (binds : list string) (action : string) (args: list expr)
| unset (lhs : string).

Definition func : Type := list string * list string * cmd.
```

Bedrock2 has expressions and commands. Expressions are not allowed to change the state of the program, so assignments to variables (`set`), `store`s to memory, and function `call`s are commands, not expressions. Read-only memory is accessed using a special load command (`inlinetable`). There is no way to retrieve the address of the table itself.

Allocating memory on the stack follows a lexical stack discipline: `stackalloc 32 as k in c` makes 32 bytes available with `k` set to their address, but only for the duration of execution of command `c`; we actually write `;` instead of `in`.

Function calls can return zero or more machine words, for example `q, r := f(a, b)`. From the perspective of the callee, arguments and return values are handled symmetrically: a function declaration includes named lists of each. Primitive I/O is described as syntactically distinguished function calls (`interact`); access to memory-mapped I/O regions is one example of an `interact` action. There is currently no support for recursion or function pointers, but the reasons for these omissions no longer appear compelling to me.

For low-level convenience in proofs, lists of commands are represented using repeated `seq`uencing statements, `skip` does nothing, and `unset` makes a variable uninitialized.

2.2 Intended Semantics

2.2.1 Guiding Principles

The semantics presented in this section, and Bedrock2 design choices more generally, are aimed to concretize and delineate existing practices in correctness-oriented low-level programming. Simplicity of use, implementation, and proof is prioritized over that of refactoring, optimization, or debugging.

Performance is a goal but perhaps not quite in the same sense as C, C++ and Rust compiler maintainers see it. Current deprioritization of optimization implementation notwithstanding, the intent is that it should be possible to produce acceptably fast

code for almost all system-level requirements using readable Bedrock2 code and a compiler that implements predictable and tasteful optimizations. Having a compiler optimize naive or inefficient Bedrock2 code for maximum execution performance is explicitly not a goal, and no semantics concessions are made to help with it. Conversely, it is the compiler's job to perform optimizations not reasonably available at the source level, such as register allocation and spill placement, and the semantics must accommodate this.

Care is taken to ensure modularity beyond the integration case studies Bedrock2 was built for and to support embedding Bedrock2 code in other contexts than a bare-metal microcontroller environment. In particular, the I/O mechanism is specified in a parametrized fashion to allow instantiation with not just MMIO, but also DMA, dedicated I/O instructions, system calls, HTTP requests, or any first-order code implemented in another programming language.

2.2.2 Flat Model of Simple Memory

Load expressions and store statements in Bedrock2 act on *Bedrock2 memory*, a map from word addresses to bytes that is the main state variable in Bedrock2 semantics. Importantly, parts of the address space that may not behave like memory from the perspective of a Bedrock2 program are excluded from this map. While Bedrock2 programs can appropriately manipulate the control stack, locations for spilled local variables, memory-mapped input-output registers, and concurrently accessed (DMA) regions, doing so using normal loads and stores is prohibited. The importance of this distinction to sound compilation is discussed in the next subsection. For now, observe that it makes the semantics of memory access very simple: every load must return the last byte stored to the same address.

More central to a programmer (and program prover) is what the semantics does not distinguish: the memory does not contain objects, types, or compound values, just bytes. Every address in the memory can be accessed in any manner provided in the syntax, regardless of how it was last accessed or what other program state points to the same address (no "strict aliasing" restriction) or the inputs from which the address was computed (no "pointer provenance" tracking). Opting for simple memory-access semantics gives up popular shortcuts in compiler optimizations but does not pose an obstacle for simple implementations targeting any system with a single address space. The payoff is keeping compiler details away from memory-layout specifications and reasoning about programs that access memory: programs are specified in terms of which bytes they put in Bedrock2 memory, and the compiler is simply required to produce the same bytes at specified interfaces.

C's `volatile` In C, some address-space locations that do not behave like memory are allowed to be used with the usual memory-access operations provided that the *type*

of the expression is marked `volatile`. The C compiler is then required to replicate the same loads and stores verbatim in the compiled program without assuming anything about their effects. Associating the I/O modality with types instead of individual memory access operations works as expected for modeling a fixed listing of memory-mapped registers (or shared variables) and can be extended to register banks with computed addresses with careful use of casts. The adoption of direct-memory-access I/O and multicore parallelism do not fit this paradigm from C as neatly: numerous library functions are used to operate on memory that is currently private to the program but may at other times be shared; marking all associated variables as `volatile` would effectively disable optimization altogether[1]. Both considerations are handled in Bedrock2 by (different instantiations of) the external-calls mechanism, where each call is allowed to change the footprint of Bedrock2 memory.

Static Constants Read-only static data is not considered part of the Bedrock2 memory. Instead, load expressions accessing static tables are syntactically distinguished from normal memory access. The rationale for this is twofold. First, it seems desirable to avoid the need to account for read-only or read-write permissions in preconditions and postconditions about Bedrock2 state. Second, this distinction leaves open future extension of Bedrock2 to minimal embedded targets where read-only and read-write memory are in different address spaces. While most new embedded-systems platforms provide a single address space (at least as an option), the separation persists in tightly integrated systems as a reflection of completely different physical implementations and performance characteristics of the two parts, at the same time as other reasons for presenting multiple address spaces seem to be altogether deprecated.

Currently, the C backend of Bedrock2 emits C constants for Bedrock2 constants (leaving the placement up to the C toolchain), while the RISC-V backend places them next to machine code, accessed using normal load instructions. Note that an integrated proof targeting a system with read-only memory needs to model the distinction in the machine specification to be faithful regardless of whether read-only memory is in a separate address space from normal memory or whether the source language relies on a syntactic or semantic separation.

2.2.3 Undefined Behavior

Bedrock2 programs must adhere to strict requirements described in the Bedrock2 specification to be compiled correctly. Violations of these rules can result in the compiled program having behaviors that cannot be defined in terms of Bedrock2 concepts, and thus no guarantees are stated about them. In particular, it is not even guaranteed that a Bedrock2 program's violation of the requirements would result in the same sort of (mis)behavior as would be expected from the (machine) language to

[1] `https://www.kernel.org/doc/html/latest/process/volatile-considered-harmful.html`

which Bedrock2 code is compiled or the environment in which it is executed.

From the perspective of a compiler engineer, the specification assigning undefined behavior to some programs allows these programs to be miscompiled without warning. The burden of avoiding undefined behavior even in the most contrived edge cases lies heavy on the programmer. This section explains why and how the Bedrock2 specification nevertheless includes undefined behavior, covering some standard considerations and integration-verification-specific trade-offs.

Undefined Memory Access Consider a Bedrock2 function that stores the byte `0` to the address `0xba17c0de` and returns. Without knowing what is at that address, could we consistently define the behavior of this program? The specification of the Bedrock2 compiler includes the invariant that target-language memory separately contains the Bedrock2 memory, the control stack, and compiled code. While the former is used throughout Bedrock2 semantics for serving Bedrock2 stores and loads, the latter are only specified to produce the correct behavior when executed. The details of the stack-frame layout and code generation are omitted from the semantics to allow for compiler changes relating to these parts to be proven against the same specification and used with existing Bedrock2 programs with existing proofs without revisiting them. It is thus not feasible to specify in Bedrock2 semantics the behavior of a program after any given modification of the control stack or compiled code, or to specify which addresses belong there. Indeed, overwriting machine code or the return address can arbitrarily hijack the execution, completely departing from behaviors any Bedrock2 program could have.

Obstacles to Compromise Declaring that any store to an address outside designated Bedrock2 memory makes the entire program invalid is unforgiving, but there doesn't seem to be a workable alternative. Some stores to some addresses outside Bedrock2 memory result in arbitrary misbehavior, and precisely delineating which ones would require fully specifying the code generated by the compiler with no room for optimizations. A run-time check for whether a store to a particular address is allowed could be used to turn undefined behavior into a well-defined failure mode, but implementing it would require elaborate bookkeeping to track Bedrock2 memory and come with memory-usage and performance costs. A more permissive specification would be desirable for compiling unverified code, and ideally it would constrain the compiler from turning a lucky memory-safety violation (perhaps a write to unused memory) into an unlucky one where the wrong code gets executed.

However, even seemingly trivial expectations such as "loading the value from an address and then storing that value to the same address right away should be a no-op" can in principle be violated by reasonable compiler designs: one example is a pipeline-aware register allocator placing a spilling store between the two operations that are

right next to each other in the source code so that the latter potentially overwrites the value just spilled. Note that while the issue can be attributed to the compiler's assumption that memory accessible to source programs is disjoint from the spill area, it does not require the compiler to be granted any assumptions about disjointness of memory accessed in different ways by source code discussed in Subsection 2.4.3. The same example is also an obstacle to defining behavior after disallowed memory access even if considering a definition in terms of *target-language* semantics: while the daring store itself would have a clear specification, the compiler-defined internal invariants required to resume predictably executing compiled code may not be satisfied after the store is executed. Requiring the compiled code to have a well-defined entry point at every location where undefined behavior is possible (for example, in the middle of a function or a loop) appears unworkably restrictive. Thus, out-of-bounds memory access remains truly undefined.

I would like to reiterate that the reason for leaving some behaviors undefined in Bedrock2 is to allow for a self-contained specification to describe the behavior of predictable implementations. This tradeoff is acceptable here because of high assurance that the Bedrock2 programs we compile and use do not have undefined behavior: we prove them against the Bedrock2 semantics and functional-correctness specifications anyway. Other arguments have also been presented in defense of undefined behavior, including enforcing portability of code and simplifying compiler analyses for optimizations; these were not explicitly evaluated in the context of Bedrock2. Similarly, arguments against undefined-behavior-driven optimization on the basis that it amplifies impact of programming errors have little bearing here as the Bedrock2 compiler does not perform these optimizations, and we primarily use it to compile verified code.

Other Undefined Behavior in Bedrock2 With some stores already assigned undefined behavior, I decided that ergonomics of semantics and program-proof tools around undefined behavior are a priority. These challenges were tackled with considerable success (see Chapters 3 and 4). In that context, and again relying on the fact that finished case studies only compile code that we prove to have well-defined behavior, easy-to-rule-out undefined behavior was chosen to trigger "proof-time errors" in cases where a default behavior, unspecified behavior, run-time errors, or even compile-time errors may have sufficed: loads outside Bedrock2 memory, reading of uninitialized local variables, calling a function with the wrong number of arguments, etc.

Historical note: Some of these undefined behaviors were in fact present in Samuel Gruetter's very first prototypes of the Bedrock2 compiler, long before the investigation into whether Bedrock2 should have undefined behavior was raised by discussions of I/O semantics, but the reasoning stands as presented. It was convenient that e.g. initialization of local variables didn't need to be revisited because we found a satisfying

way to model undefined behavior, but that shortcut did not drive the decision-making.

All these cases share the premise that actually proving something about the behavior of the code requires ruling out the failure case regardless of its semantics. For the ease of programming, proof, and integration, Bedrock2 does not ascribe undefined behavior to cases where an obviously correct behavior does exist, such as loading 2 bytes from an address to which 4 bytes were previously stored (see Section 2.4) or performing arithmetic that reveals the reality that machine integers have finite width (see Subsection 2.5.2).

2.3 Interactive Sequential Programming

I/O actions in Bedrock2 behave like calls to functions whose behavior is specified axiomatically in terms of the arguments (resp. return values), trace of function calls so far, and Bedrock2 memory before and after the interaction. Informally, the semantics we want is to require programs to be proven correct for all input values and to constrain the compiler to produce the same output as the source program, given the input so far. The technical challenges of formalizing this notion while allowing unspecified addresses during stack allocation and ruling out undefined behavior and infinite loops are discussed in Chapter 4. This section will focus on programmer obligations associated with external calls and their importance for modular integration verification.

2.3.1 Preconditions (Safely Compiling MMIO Writes)

Independently of what is specified about the behavior of the source program, for some I/O operations, correct *compilation* requires preconditions on the arguments. The simplest example is validity of addresses used for memory-mapped I/O. As an MMIO write gets compiled to a normal store instruction, it is critical to ensure that Bedrock2 memory (or the stack, etc.) will not be overwritten when that store executes. Using an address that actually points to an MMIO register is already a sensible verification condition for the Bedrock2 program, and it is sufficient to guarantee disjointness from Bedrock2 memory and compiler data structures. The question is about how to make this guarantee available to the compiler correctness proof.

The example presented in the previous paragraph has an analogue that can be presented in the framework of external calls and compositional compilation, without discussing input or output specifically. What should the proof obligation for linking a program against a library look like, and when should it be proven? The standard answer is that a theorem about the behavior of a linked software package can be derived from a program-logic proof about the caller under some axiomatic specification of the callee and the proof that the callee satisfies that specification. This seemingly

simple requirement is further complicated by potential circular dependencies, but this possibility is not relevant to Bedrock2 I/O.

In Bedrock2, the compiler is also the linker, and it is parametrized over the compiler for I/O operations. Parametrizing the semantics in the same manner leads to a workable definition: invoking an I/O operation must satisfy an abstract precondition, otherwise undefined behavior ensues. The main compiler can be proven correct generically, without relying on the specific source-level specification. The I/O operation compiler (which in the running example just emits a single store) gets to assume the compiler memory-layout invariant and the concrete precondition of a specific MMIO operation. Its proof then proceeds by arguing that an address within the specified MMIO range cannot overlap any of the memory regions described in the compiler invariant, so the store preserves their contents and properties. The MMIO compiler is constrained from omitting the store by the same source-level specification requiring the I/O trace to be appropriately extended. Sound compilation of input and output operations is discussed in more detail in Section 4.3.

2.3.2 Relationship to Concurrency

Bedrock2 does not embrace concurrency as a first-class feature. This is not intended to exclude implementation or verification of programs intended for a concurrent environment but rather to leave the options open for how the concurrent composition of a Bedrock2 program and another program would be proven. More importantly, it is a goal to enable programs that interact with unproven external systems (e.g., peripheral devices) to be proven against putative specifications without cooperation from the maintainers of these systems. (As undesirable as it sounds, this scenario is very common in systems programming.)

This perspective is different from that of popular concurrency-centered verification frameworks, and it would be interesting to see how (or whether) it can be reconciled. For example, could we state and prove that a Bedrock2 program correctly implements the shared-memory client-server interface used by a program verified using Iris [Jun+18], perhaps in Refined C [Chr+21]? One appealing avenue would be to define the correctness conditions of Bedrock2 external interactions in terms of an Iris resource algebra element, but even stating that this is sound seems challenging.

2.3.3 Alternative Instantiations

The Bedrock2 external-interaction interface has been instantiated to model calls to functions implemented using different calling conventions in C and RISC-V assembly [Str20]. This work was done in the context of prototyping a verified first-stage bootloader and cryptographic attestation mechanism. The same project includes a verified driver for a concurrent DMA controller modeled using a very similar external-

interaction primitive but implemented in RISC-V assembly to allow access to the entire memory of the machine. Under the assumption that the concurrent DMA operation finishes in some unknown but bounded number of steps, total correctness is proven.

2.4 Other Memory Models for Sequential Code

Bedrock2 semantics reveal pointers as machine words and memory contents as bytes; this is sometimes called a concrete memory model. Addresses and contents of allocations are left unspecified (nondeterministic). This is made workable for compiler verification due to use of omnisemantics (Chapter 4). To my knowledge, this combination is novel in the context of integrated verification and represents a trade-off in favor of semantic simplicity over optimization simplicity while upholding modularity. This section will briefly review relevant history and alternative designs for semantics of pointers and memory from the perspective of the following question: What information can pointers and memory cells carry in a given state? For example, is a dangling pointer still an address, and does an uninitialized memory cell just hold an unknown byte, or can operations on the two behave in ways that no address or byte would?

Concrete modeling of addresses and memory has long served as the foundation of low-level software verification, for C and assembly. Famous tools like VCC [Coh+09] and VeriFast[2] and large-scale proof projects such as seL4 [TKN07; SMK13] rely on the assumption that pointers behave like addresses and values can be treated as bytes. For assembly-level languages such as the original Bedrock [Chl11], x86 with macros [JBK13], or bare machine code [Myr08], there isn't much of a choice: the machine has words, and the use of them as addresses is in the eye of the programmer. Informally, treating C programs with the same concrete semantics corresponds to the perspective of C as a portable assembly language. Programs that make use of address arithmetic and concrete in-memory representations can be tricky to verify, but the mere presence of this flexibility is not a detriment to effective specification and proving.

2.4.1 Abstraction for Determinism in Compiler-Correctness Proofs

Compiler-correctness proofs have preferred to avoid specifying pointers as machine words. Instead, the use of models that are just abstract enough to describe the semantics as deterministic appears to be the sweet spot within this domain, with addresses and contents of allocated memory being at the core of the main challenge. The Piton

[2] `https://github.com/verifast/verifast/blob/57fa89b1/soundness.md#c-programs`

language from the CLI verified stack only allows pointers to statically allocated scalars and arrays. Each pointer is represented in the semantics as a pair containing the name of the global variable and an offset within it [Moo88]. CompCert semantics apply a similar model to heap and stack allocations as well by generating sequential identifiers for dynamically allocated blocks on the fly. To maintain determinism, the pointers remain abstract throughout all supported usage, even when themselves stored into memory or cast to integers. As a consequence, address arithmetic is limited to keep offsets independent of addresses, and externalizing (e.g., through `printf`) any value derived from a dynamically allocated address is treated as triggering undefined behavior, to be compiled into anything whatsoever. A similar trick is used to operationalize interaction with freshly allocated memory in a deterministic manner: in addition to bytes, the type of values that can be stored in memory includes a special marker for uninitialized contents, which is propagated by loads and arithmetic operators.

Here is an example of how CompCert treats nondeterminism as undefined behavior:

```
#include<stdio.h>
int main() {
        int x = 0;
        printf("value of x: %d\n", x);
        printf("addr  of x: %p\n", (void*)&x);
}

./ccomp -interp printptr.c
value of x: 0
...
Stuck subexpression: printf(<ptr __stringlit_2>, <ptr x>)
ERROR: Undefined behavior
```

Why Determinism? Deterministic modeling greatly simplifies proofs of compiler correctness through execution-by-execution simulation arguments: if every program has at most one possible execution outcome, showing that every compiler-generated program has an execution that matches that of the corresponding source program implies that all of its executions do [Ler09; Ler06; LB08]. However, the machinery needed to maintain the technical determinism throughout the semantics can itself become a complication for defining a simulation relation. For example, consider a compiler pass that eliminates a redundant stack allocation. If allocated blocks are numbered sequentially, the optimization causes all blocks allocated in the future to be numbered one less than they would have been. As pointers to these blocks may themselves be stored in memory, the seemingly small optimization drastically affects that abstract representation of all future program states. Formal reasoning about this relationship between input-language and output-language memories and values is a key challenge for the compiler-correctness proof. CompCert proofs use purpose-built

mathematical objects called memory embeddings and memory injections to track the relationship between the precompilation and postcompilation memories, and an elaborate library of lemmas is proven about these constructions. Nevertheless, seemingly straightforward removing and coalescing of allocations remains the most difficult memory transformation in CompCert [LB08, §5].

Numerous projects have experimented with extending CompCert with more concrete memory models and nondeterministic allocation semantics, for example to handle concurrency [Šev+13] or integer-pointer casts [Kan+15]. These approaches give up using determinism to simplify simulation arguments of key compiler passes and instead directly prove that for every nondeterministic execution of the compiled program there is a matching execution of the source program. CompcertTSO authors in particular comment that the corresponding phases that change memory access "are the heart of our proof and the most challenging part," even though the same paper also presents nontrivial concurrency-specific optimizations such as fence elision.

2.4.2 Abstract Memory Models From a Source-Language Perspective

Later extensions of CompCert have managed to accommodate a subset of the low-level programming patterns validated by models with concrete pointers through more sophisticated manipulation of abstract pointers. For example, accessing pointers as bytes and reassembling them is supported through a dedicated form of intermediate semantic "values" that stand for "ith byte of this abstract pointer" [KLW14]. A proposed extension [BBW14; Wil16; BBW15; BBW17] allows more general pointer arithmetic to be modelled by letting partially evaluated expressions persist throughout the semantics and deterministically normalizing them to values whenever the expression would yield the same value in every concrete state compatible with the abstract pointers and memory. This proposal illustrates the key distinction between abstract and concrete semantics when it comes to determinism: even though the definition of the semantics includes a concrete model of pointers and memory, the state tracked throughout execution is a deterministic abstraction: there is no notion of a *current* concrete state. Even this mixed model is, however, insufficient to model code whose control flow is not determined by the symbolic rules: the current command (an abstraction of the process counter) cannot be a symbolic expression. One example of where this shows up this is a common implementation of `memmove` that chooses between left-to-right and right-to-left copying based on the ordering of the argument pointers: when called on two freshly allocated arguments, there is no deterministic answer to which is placed at a smaller address!

Software-verification projects based on CompCert semantics have handled the challenges raised by an abstract memory model in a number of ways. The Verified Software Toolchain embraces the abstract model and builds a separation logic for mem-

ories organized by identifier-offset pairs [App+20, §26]. After substantial efforts to encapsulate and automate the memory-reasoning framework, the overall proof experience is quite close to working with a concrete model, but some verification conditions such as pointer comparison requiring the pointers share the same allocation identifier need additional user-specified preconditions and invariants to be discharged. According to [Lep+22], operating-systems-focused projects such as CertiKOS and SeKVM work around this complexity by using a single up-front array "allocation" to have all pointers share same the same allocation identifier. Application-facing allocation is then implemented by carving out parts of that array. While not quite equivalent to treating pointers as plain addresses, this technique also allows arithmetic on the pointer offsets to satisfy many use cases usually handled through address arithmetic.

2.4.3 The Rumination of C Standards

There is also an entire body of research aimed at interpreting and clarifying the semantics described in ISO-standardized natural-language specifications of C. Formalization of semantics has played an important and influential role in this effort, but the priorities are distinct from projects aiming to create verified software or compilers. This work seeks to explain and resolve intricacies and ambiguities relating to existing (overwhelmingly unverified) compiler transformations and precisely delineate a set of rules that programs must follow to avoid miscompilation. Subtle edge cases in the C standard are numerous [KW12], but even the basic structure of state that expressions in a C program can access has been under constant revision essentially since Defect Report 28 in December 1992 [92], and perhaps continuously since the very start of standardization [Rit88].

It has been clear that the committee-approved standards do not specify a concrete model for values in memory or pointers to memory [04], but after 30 years of cumulative work, a faithful and formalizable description of the state of a C program seems just tantalizingly close. The remainder of this subsection will give a taste of state-of-the-art semantics based on the PhD thesis of Robbert Krebbers on the formalization of C11 [Kre15] and the recent proposal PNVI-ae-udi from [Gus+20]. The key modeling trick is that the state of a program is defined to contain *both* potentially nondeterministically generated concrete addresses and bytes and symbolic abstractions thereof. It is left as the programmer's responsibility, with very limited exceptions, ensure that each operation is valid with respect to both notions of state and that the two views remain consistent with each other.

Strict aliasing means that memory contains the C types and nested structure of objects in addition to the bytes that represent them. While most (initialized, non-padding) bytes have stable values which can be read, written, and copied using the `char` type, other means of reading and modifying the objects may be subject to restrictions based on effective types recorded in the abstract state. There is no way to query the effective type of an address, but access incompatible with it will still

trigger undefined behavior. For example, accessing an object declared as `uintptr_t` by dereferencing a pointer of type `void**` is undefined behavior even though both have the same size and alignment. Modifying or even just reading an object through a union type can change which variant of the union is considered active, forbidding accesses through pointers to alternative variants. This last rule was formalized from GCC documentation rather than the C11 standard, as the latter is unclear about these conditions [Kre13, p. 3]. Formalizing how modifications of dynamically allocated memory change the effective type of this memory is left for future work. It appears that doing so will require first reconciling the per-access effective-type checks and updates described in the C17 standard with the stricter resolution of Defect Report #236 which states that an example program would "invoke undefined behavior, by calling function f with pointers qi and qd that have different types but designate the same region of storage" [06]. I am not aware of any case studies verifying pointer-manipulating code against a model faithful to (a subset of) the strict aliasing rules.

Similarly, pointers have both word addresses, which are affected by pointer arithmetic, and abstract allocation-instance identifiers ("provenance") which just propagate, and it is the programmer's responsibility to make sure that every access is performed using an address that is within the bounds of the identified object. It is also forbidden for the address to leave the boundaries of the object even without dereferencing, except to point to the end of the object. Casting integers to pointers is allowed after arbitrary address arithmetic: the resulting pointer will have the provenance of some object which contains or ends at this address and whose address was previously converted to an integer, to be disambiguated by the next access through the freshly converted pointer. To show that it is possible to verify integer-pointer casts found in low-level code against this model, RefinedC-VIP [Lep+22] defines an abstraction where integer-to-pointer casts that are not trivial inverses of previous pointer-to-integer casts need to be annotated with other pointers whose provenenance must match the casted addresss.

2.4.4 C Standards and Systems Programming

Separately from the intellectual inquiry into how to disambiguate and formalize the rules underlying undefined-behavior-driven optimizations, there is the engineering question whether rules prescribing additional undefined behavior for the purpose of specific optimizations are acceptable, desirable, and worthwhile for practical programming. And as one should expect in any large human system, the interactions between the systems programmers, compiler maintainers, and standards committees about what the standard should specify are muddied with a question of power: who gets to make the rules? The current C standards appear to be written by compiler maintainers and semantics researchers, and even the effort to gather input from C programmers about the meaning of the C standard asked "Will that work in normal C compilers?" and "Do you know of real code that relies on it?", but not "Is it a burden that you cannot rely on it?" or "Is it important that compilers optimize other

code in ways that break this code?" [Mem+16].

In my experience of following and occasionally contributing to systems-programming projects, the abstract rules specified for post-K&R C are often considered a chore and sometimes outright problematic; little if any benefit is ever attributed to them. Several important and well-recognized projects (e.g, Linux[3], Chromium[4], Firefox[5], Clang[6], OpenSSH[7], reportedly[8] also FreeBSD and OpenBSD) rely on compiler flags such as `-fno-strict-aliasing` that restore more concrete behavior. Some explicitly prioritize clarity and maintainability of the code over conformance with abstract-state-based undefined-behavior rules. This preference is in stark contrast to the description of the same flag in the GCC documentation, which describes it as a "workaround" for "faulty legacy code"[9]. To be clear, my understanding of the situation is that systems programmers would absolutely prefer to have their code work based on standardized guarantees, just not at the cost abiding to the current standard. In the characteristic directness of Linus Torvalds, `-fno-strict-aliasing` and code simplifications that rely on it are here stay because he "tried to get a sane way a few years ago, and the gcc developers really didn't care about the real world in this area. [...] I'm not going to bother to fight it." This was in 2003, and not much appears to have changed since.

More recently, a number of pieces dedicated to expressing concern over the abstract-state-based undefined behavior in C and advocating for alternatives have been written and circulated, but they seem not to have reached semantics researchers I have discussed this issue with. I will thus direct a reader interested in systems programming in a C-like language to "The Strict Aliasing Situation is Pretty Bad" [Reg16], "What every compiler writer should know about programmers" [Ert15], and "How ISO C became unusable for operating systems development" [Yod21].

Verified systems programming is not exempt from these concerns either. The seL4 project chose to use translation validation of compiler-generated binaries against their (concrete) model of C semantics instead of connecting to a verified compiler. The rationale was based on concern over "cases where the standard is purposely violated to implement machine-dependent, low-level operating system (OS) functionality", explicitly noting that the "proof for seL4 did not specifically address the strict-aliasing condition. It is one of the standard violations that systems code must make at some point" [SMK13]. For a smaller example, the implementations of `malloc` and `free` proven using the Verified Software Toolchain [AN20] store a free-list pointer of type

[3] `https://lkml.org/lkml/2003/2/26/158`, `https://lkml.org/lkml/2009/1/12/369`

[4] `https://groups.google.com/a/chromium.org/g/chromium-dev/c/dUebWSEpAR8/m/xhsispuiNu4J`

[5] `https://bugzilla.mozilla.org/show_bug.cgi?id=413253`

[6] `https://github.com/llvm/llvm-project/blob/main/clang/CMakeLists.txt#L330`

[7] `https://github.com/openssh/openssh-portable/blob/614252b0/configure.ac#L192`

[8] `https://forum.nim-lang.org/t/3121#19651`

[9] `https://web.archive.org/web/20220412081358/https://gcc.gnu.org/bugs/`

`void *` to the address passed to `free`. It is, of course, the expected usage of these functions that the caller would store objects of a potentially different type at the same address. I believe this would not be allowed by Krebbers' formalization of strict aliasing rules, but it may be allowed by the standard's effective-type rules for memory with no declared type if not affected by the resolution of Defect Report #236. As VST is proven sound against the C semantics used by CompCert, there is no need to fear miscompilation when this program is compiled with CompCert. The paper does reassure the reader that the "malloc/free system, like any VST-verified program, can also be compiled with gcc or clang", and using `-fno-strict-aliasing` seems prudent in that case (it is the default but easily accidentally disabled by higher optimization levels). Here is a program that returns 0 with `-fno-strict-aliasing` but often 1 without it; VST can be used to prove that it returns 0:

```
long foo(long *p, void **q) {
  *p = 1;
  *q = 0;
  return *p;
}
long main() {
  int x;
  return foo(&x, &x);
}
```

The key to the VST proof is the following lemma:
`∀ p, data_at (tptr Tvoid) (Vint zero) p = data_at tint (Vint zero) p`
It allows the same memory location in a 32-bit C program to be interpreted as holding either `void*` or `int`, which is in-line with systems-programming needs but in direct contradiction to strict-aliasing rules.

2.5 Translating Bedrock2 To C

Regardless of how the pursuit of a clear and consistent C standard pans out, the purpose of Bedrock2 is to be a cross-platform assembly language, so Bedrock2 pointers are addresses, and memory maps addresses to bytes. As discussed, this view is far removed from that of the C-compiler maintainers, standards committee, and semantics researchers. At the same time, it would be be very appealing to use an off-the-shelf C toolchain to run Bedrock2 code. This would unlock platforms, debugging mechanisms, and optimizations not implemented in the Bedrock2 toolchain. Of course, the very optimizations whose soundness the complex C memory models were created to justify are not sound for C code that Bedrock2 was designed to support.

Somewhat surprisingly given the discussion in the previous section, it appears that the abstract-state semantics prescribed for C programs do not rule out a translation

that maps Bedrock2 programs that do not trigger undefined behavior to C programs that do not trigger undefined behavior. I implemented a quick-and-dirty Bedrock2-to-C translation very early on in the project to aid with prototyping of Bedrock2 code and its compiler, not expecting it to produce standards-compliant C. The C backend persisted with small fixes, and was incorporated into the testing infrastructure of Fiat Cryptography's Bedrock2 backend and used for performance evaluation of Rupicola. In parallel, the research into semantics of integer-pointer casts in C seems to have converged on PNVI-style models which are more permissive than purely abstract models, removing the main barrier to standards-compliant translation. The strategy is simple: the translated code will work with machine words and bytes to the extent possible, avoid C constructs that trigger undefined behavior conditionally based on the abstract state, and keep the concrete state of the C program in-sync with that of the Bedrock2 program. More specifically, the key invariant is that every byte in the Bedrock2 memory must have a corresponding concrete byte in the C memory at the same address.

Realizing a translator that for arbitrary well-defined Bedrock2 programs actually avoids all the ~200 ways undefined behavior can be triggered in C is still a delicate task to say the least. Yet I did write a translator, and this section will review how it dodges the kinds of undefined behavior I considered.

2.5.1 Memory Operations

Machine words (the only scalar type in Bedrock2) are translated to `uintptr_t`, and the translation of a complete program includes an automatic assertion checking that the C implementation uses a little-endian representation for this type. A Bedrock2 memory store takes a machine-word address and a machine-word value; there are four variants for writing 1, 2, or 4 bytes, or an entire machine word (4 or 8 bytes). The translation of a memory write to C uses a cast operation to convert the address to `void*`, takes the address of a C variable containing the value to be stored, and calls `memcpy` with the appropriate number of bytes.

```
static inline
void _br2_store(uintptr_t a, uintptr_t v, uintptr_t sz) {
  memcpy((void*)a, &v, sz);
}
```

For the `uintptr_t`-to-`void*` cast to be justified with respect to PNVI, the C program must have an object at the address so that provenance of the new pointer can be extracted from it. This condition is satisfied based on the assumption that the store in the Bedrock2 program succeeds and the Bedrock2 and C memories match. Under PNVI-ae and PNVI-ae-udi, there is an additional requirement that the concrete representation of the address of the pointed-to object must have been accessed earlier in the execution. The translation maintains the invariant that all objects the translated

program allocates have their address taken and cast to `uintptr_t` immediately, arguments are passed from C to Bedrock2 as `uintptr_t`, and translated Bedrock2 code only reads the concrete representation of the C memory. Thus PNVI-ae is equivalent to plain PNVI for programs translated from Bedrock2, and the translated code does not need the extra flexibility of PNVI-ae-udi.

From the perspective of strict-aliasing rules, `memcpy` is allowed regardless of the effective type of the object because it is specified as copying the requested number of *characters*, and accesses through the `char` type are allowed regardless of the object type recorded in abstract state. If the destination address points to memory that was dynamically allocated in C using the C standard `malloc` function (or if it otherwise lacks a declared type), copying a `uintptr_t` to that memory will change the effective type the C semantics consider to be stored at that address to `uintptr_t`. Translated Bedrock2 code will not be sensitive to this, but C code calling translated Bedrock2 code will no longer be allowed to access that address using an unrelated type.

Memory loads are similarly translated to `memcpy`, but with an optimization (contributed by Clément Pit-Claudel) to use a size-specific destination type to help compilers infer the range of the resulting value:

```
static inline
uintptr_t _br2_load(uintptr_t a, uintptr_t sz) {
  switch (sz) {
  case 1: { uint8_t  r = 0; memcpy(&r, (void*)a, 1); return r; }
  case 2: { uint16_t r = 0; memcpy(&r, (void*)a, 2); return r; }
  case 4: { uint32_t r = 0; memcpy(&r, (void*)a, 4); return r; }
  case 8: { uint64_t r = 0; memcpy(&r, (void*)a, 8); return r; }
  default: __builtin_unreachable();
  }
}
```

The considerations for integer-to-pointer casts and strict aliasing for loads implemented in this manner are a subset of those discussed in the context of stores.

Small dedicated experiments and the Rupicola evaluation benchmarks suggest that GCC and Clang generate good code for these calls to `memcpy`, compiling them equivalently to accesses through casted addresses (`*(uint64_t*)(p)`) when a sufficiently high optimization level is enabled. It is still mildly unfortunate that debugging and execution at lower optimization levels (or with simpler compilers) is cluttered with `memcpy`, but that is the price of complying with the C standards.

Alignment Replacing the bytewise `memcpy` with a multibyte access would not be possible in environments that do not implement unaligned memory access (either in

hardware or in an exception handler). Environments without support for unaligned access are not common and have been becoming less common, and semantics of Bedrock2 allow memory access of all supported sizes at all accessible addresses. C standards disallow creation of pointers with addresses that do not satisfy the alignment requirement of the target type of the pointer, but `memcpy` takes `void*`, which does not have an alignment requirement. In summary, the use of `memcpy` in this manner is portable but incurs a performance penalty on architectures where unaligned access is emulated by the compiler or an exception handler.

No special case for memory access at an offset A tempting optimization was proposed to use `_br2_load((char*)A+i)` instead of `_br2_load(A+i)`, which helps Clang 10 to find additional vectorization opportunities for simple loops mapping over an array of characters. (A vectorized implementation would be valid either way, but it is not discovered with the simpler translated version, likely because Clang relies on types instead of the context for some analysis.) However, this vectorization-friendly form is not valid in general. A first hint at the problem is that converting `A` instead of `i` to `char*` is an arbitrary choice as far as the semantics are concerned. Bedrock2 addition expressions are commutative, so code with a different argument order would satisfy the same Bedrock2 specification. PNVI rules for non-NULL integer-to-pointer casts in C require that an object exist at the casted address at the time of the cast; it is not acceptable to cast an out-of-bounds address to a pointer and then adjust it to point to a C object. Thus attempting to aid vectorization in this manner would cause the Bedrock2 program that stack-allocates 2 bytes at address `A` and then stores a zero byte to `1+A` to be translated to `_br2_load((char*)1+A)`, which triggers undefined behavior unless there is an object at address 1.

2.5.2 Arithmetic Operations

To recall, Bedrock2 arithmetic defines exception-free logical, bitwise, signed, and unsigned operations on the type of machine words. While some of these map trivially from Bedrock2 to C, there are several important considerations to keep in mind for a sound translation. This is the only part of the translation in which bugs were found by trying to use it, rather then during initial prototyping or code review.

Famously, arithmetic can trigger undefined behavior in C, and it cannot in Bedrock2, so the translator must ensure that the undefined cases are not used. Division and remainder operations are translated to function calls that check that the denominator is nonzero and return a fallback value otherwise. As all current Bedrock2 developments assume the RISC-V fallback values even though the semantics are technically parametrized over this choice, the C backend does the same. Bit shifts by more than the width of the argument are undefined behavior in C, so the translator uses a bitwise mask to keep the shift amount in the required range, again matching RISC-V.

In an effort to keep the generated C code readable for use as a debugging mechanism, Bedrock2 expressions are translated to C expressions in a manner that preserves nested operations. (Side effects are not allowed inside Bedrock2 expressions, so there is no concern about C sequence-point rules.) This means that inferred result types of C subexpressions can affect the meaning of outer operations, potentially causing incorrect outputs or triggering undefined behavior when they are not `uintptr_t` as expected. In particular, comparisons return `int` regardless of the types of the inputs, so `(uintptr_t)0==(uintptr_t)0` is an `int`, and shifting it left by one less than the width triggers undefined behavior due to signed overflow. To avoid this, the translator inserts `uintptr_t` casts around all comparisons.

The C standard provides no mechanism for `uintptr_t` literals, so the translator uses the `ULL` suffix for `unsigned long long` and casts the value to `uintptr_t`. Bedrock2 uses one type for both signed and unsigned arithmetic; signed comparison and right-shift are implemented by casting the operands to `intptr_t` and the output back to `uintptr_t`. There is no standard C construct to get the high half of a 64-bit-by-64-bit multiplication, but the translator implements this using `unsigned __int128` which is provided by GCC and compilers that mimic it. All in all, only bitwise operations, addition, subtraction, and multiplication enjoy trivial single-operator translations.

2.5.3 Functions, Allocations, and Variables

Bedrock2 assignment statements, conditionals, and while loops translate directly to corresponding C constructs. Functions, stack-memory allocation, and internal and external function calls require the introduction of new variables and are thus subject to the usual considerations of avoiding variable capture.

Bedrock2 functions can take arbitrarily many parameters (as in C) and can return arbitrarily many return values (unlike C, which can return structs instead). The function definition in Bedrock2 requires output parameters to be named next to the input parameters. To allow interoperability with idiomatic C code, the first output parameter is mapped to the return value of the C function. Additional output parameters are handled in destination-passing style: the translated C function takes a freshly-named `uintptr_t*` for each output beyond the first, and the local variables holding the output parameters are stored to corresponding pointers at the end of the function.

C requires local variables to be declared before use, so a declaration of all Bedrock2 local variables and output parameters is generated at the beginning of the function. Reading or returning an uninitialized local variable is undefined behavior in Bedrock2, so the generated C variables are not initialized either.

Bedrock2 features for allocating memory on the stack and accessing a table of constants are translated using `uint8_t`-array declarations. As the Bedrock2 program is

free to perform arbitrary arithmetic on the address of the space allocated on the stack, the address of the array is taken and cast to `uintptr_t` right away. It is tempting to use `alloca` instead of an array declaration because memory returned by `alloca` does not have a declared type, which may (unless DR #236 overrides C17, see Subsection 2.4.3) allow C code called from bedrock2 to access that memory with other types. However, memory allocated using `alloca` is only freed at the end of the function, so using it to implement Bedrock2's scoped stack-allocation primitive can lead to unbounded increase in stack usage (e.g., if called inside a loop). Hoisting all stack allocations to the beginning of the function would resolve both of these concerns at the cost of significantly complicating the unverified translation and potentially causing needless eager allocations, so I did not implement this transformation.

Critically, the stack-allocated C arrays are initialized even though Bedrock2 stack-allocation semantics leave the contents unspecified: in C, different accesses to the same uninitialized memory location can return different values[10]! Even though Bedrock2 semantics let the compiler nondeterministically choose the contents of the stack-allocated memory, it is chosen once at allocation and must remain fixed thereafter. The following Bedrock2 program is proven to return two equal machine words but would trigger undefined behavior if translated to C using an uninitialized array for `t`:

```
Definition stacknondet := func! () ~> (a, b) {
  stackalloc 4 as t;
  a = (load4(t) >> $8); (* extract btyes 1..3 *)
  store1(t, $42);       (* replace byte 0 with 42 *)
  b = (load4(t) >> $8)  (* extract bytes 1..3 again *)
}.
```

2.5.4 Program Units, Sanity-Checks, and Files

The recommended usage of Bedrock2-to-C translation is to translate an entire library or program at once. The generated C file is ready to be compiled: it includes `#include` directives for the required standard C headers and `static inline` functions for Bedrock2 arithmetic and memory operations. The continuous-integration tests of the Bedrock2 repository routinely extract, compile, and execute C versions of Bedrock2 programs defined and proven in Coq. If the Bedrock2 program contains calls to external functions (`cmd.interact`), the generated file is not usable alone; the current recommendation is to `#include` it in another file that also includes declarations or definitions of these functions.

The chosen implementations of essential features such as widening multiplication already rely on extensions to standard C, so I chose to include in the generated file an `__attribute__((constructor))` function that checks at startup that some

[10] `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1747.htm`

implementation-defined choices in C match those specified in Bedrock2 semantics and aborts otherwise. Concretely, `uintptr_t` must use a little-endian representation, and bitwise and arithmetic operations on `intptr_t` must be related through the two's-complement convention. This is intended to flag accidental unsupported use, but it is not an exhaustive list: for example, it is not clear how to check that the C implementation uses a single address space for all pointers.

Deviating from the long-established kludge of using Coq-to-OCaml extraction to print syntax trees generated inside Coq, Bedrock2 includes an Ltac2 implementation of printing a Coq list of bytes and Python script that calls it to print a Coq definition (in `coqtop`, with the artificial stack-usage limit lifted). This setup has three quality-of-life benefits over the previous status quo. First, the same syntax-tree stringification code can be used interactively in Coq to fill the proof feedback buffer with the exact output that would be redirected to a file by the build system, allowing it to be saved to an ad-hoc location or passed to another command from within the editor. Second, the possibility of running into bugs where Coq-to-OCaml extraction generates invalid OCaml code is avoided. Third, build rules are simplified by removing the step of compiling OCaml code (Bedrock2 can be used without an OCaml installation).

2.5.5 Reflections

Supporting use of Bedrock2 code with off-the-shelf C toolchains has been a great enabler for prototyping of Bedrock2 programs and the verified stack itself. If sufficient confidence is gained in the correctness of this translation with respect to behavior of mainstream C toolchains, it will provide a new pathway for contributing verified code to C projects. However, the combined experience of established efforts at C verification reviewed in the previous section and the amount of "small" ad-hoc fix-ups already identified as necessary for the conceptually simple translation presented here call for caution about correctness claims that can be made about this code. As the proofs say nothing about compatibility with any common C toolchain or some formalization of the C standard, code review will have to remain the primary means by which compatibility is ensured.

In principle, one could seek to prove more about this translation or about another translation to C. CompCert would be an appealing target for verified integration supporting more hardware platforms, but the CompCert C semantics are substantially incompatible with Bedrock2. It would not be enough to follow CertiKOS and require that all memory locations accessible to the same Bedrock2 program belong to the same allocation block in the CompCert memory model for modeling purposes. Addresses returned by Bedrock2 stack allocation would have to be instantiated with integer indices into a distinguished memory block acting as both the data stack and heap for Bedrock2 because CompCert does not allow arbitrary arithmetic on pointer addresses. Calling a Bedrock2 function soundly from CompCert-verified code would require copying input into that block and passing the index to the translated Bedrock2

function. While not as burdensome to use as implementations of functional languages, a translation compatible with CompCert proofs would be sufficiently dissatisfying from an usability perspective to discourage me from putting in the effort.

A proof against a faithful formalization of the C standard could be attempted for the current translation, but the current formalizations I am aware of are not complete enough to support it satisfyingly. CH2O does not allow integer-pointer casts, so all allocations, loads, and stores in the translated code would have undefined behavior. A proof against a formalization of any PNVI variant, such as the one used to prove soundness of VIP in [Lep+22] would be interesting, but it would not give confidence in any mainstream C implementation in particular or the strict aliasing rules in general. A partial translation based on an interprocedural type and alignment analysis could generate C code for Bedrock2 programs that don't "really need" integer-pointer casts or bytewise memory access and could likely be proven against an existing formal model of C with substantially more effort.

Chapter 3

Proving Bedrock2 Programs

Having settled on the rules that Bedrock2 programs must follow, it is now time to design how these rules will be encoded in the Coq proof assistant. Following what will be a theme in this thesis, I will first consider this question from the perspective of the user of the specification, in this case the programmer seeking to prove that a Bedrock2 program does not trigger undefined behavior and fulfills a programmer-chosen formal specification. Connection to a verified compiler for Bedrock2 will be discussed in Chapter 4.

After substantial experimentation, I settled on a program-proof-experience design centered around line-by-line symbolic execution of Bedrock2 code within the Coq proof engine and keeping track of the symbolic state of the program using normal Coq variables and hypotheses in the proof context. This approach has the key benefit that proofs of Bedrock2 programs can be composed incrementally from calls to existing Coq proof procedures, ad-hoc proof-automation scripts, and manual proof steps as appropriate. In other words, the API conventions for proof procedures called during the verification of a Bedrock2 program are the usual ones for Coq proof steps, not custom. This means that proofs of Bedrock2 programs are compatible with established workflows for prototyping and debugging proof scripts. In particular, steps of unsuccessful proof attempts can be processed incrementally, allowing for inspecting the symbolic state in proof context and fine-grained experimentation with alternative proof strategies.

While neither the Bedrock2 language nor the verification system is designed for a fixed set of proof procedures, support for proof automation is a priority. Proofs of most existing Bedrock2 programs rely on the Coq solver for linear integer arithmetic (`lia`) and the separation-logic-based proof procedure for memory access described in Section 3.3. In some cases, complete automation is achieved. More importantly, programs that are *almost* within the scope of automated proof still benefit from these solvers for all proof steps to which these solvers are applicable, with minimal tooling-

related overhead. The choice of proof method can be made at function, statement, or even subexpression granularity, and there is no restriction on control flow of the composition: solvers can be used to prove proof-script-generated assertions which in turn are used to discharge preconditions of a manually instantiated lemma, and all this can be preceded or followed by any proof style.

To keep the complexity of nontrivial combinations of proof methods manageable, the verification conditions and widely used proof scripts in Bedrock2 trade away deductive power for predictability. The philosophy is simple: it is better to sometimes stop and not verify the next command than to keep the programmer guessing whether the execution of some command would be verified automatically. A similar consideration applies to the symbolic state a proof step leaves behind, perhaps after processing another command in a Bedrock2 program: general-purpose Bedrock2 proof procedures only proceed in cases where they can leave behind a satisfying proof state, choosing to fail instead of proceed with a context that may be too weak to prove the rest of the program. An extreme form of this approach would be to only proceed with symbolic execution as long as the strongest postcondition of the program can be established and represented in the proof context. Proof automation for simpler commands also adheres to this stricter discipline, but processing of higher-level constructs such as loops and function calls allows for incompleteness due to user-specified abstraction. For example, the symbolic state of a program calling a function is updated based on its declared precondition and postcondition alone, without inspecting the implementation.

3.1 Program-Proof Examples

This section will describe the basic mechanics and user experience of specifying and proving Bedrock2 programs in Coq. The presentation is intended to be pragmatic and use-case-driven. General definitions that make the meaning of these proofs precise are deferred to Section 3.2. A reader adamant to start with the foundations can skip ahead first, but the approach to program proofs sketched in this section is central to why the later definitions are designed the way they are.

3.1.1 Basics, Memory Acces: `swap`

We will implement a function that swaps the values of two addresses it takes as arguments, specify its behavior using separation logic, and see how Bedrock2 proof tooling handles associated verification tasks. The program at hand is intentionally minimal, and Bedrock2 handles it in the same manner as existing separation-logic-based verification systems. Readers familiar with this background should feel free to glance at the code, pick up the notations of the day, and move on to a less trivial example in Subsection 3.1.2.

Implementation The following function takes two addresses as arguments, loads a machine word from each of them, and stores each value to the other address.

```
Definition swap := func! (a, b) {
  t = load(b);
  store(b, load(a));
  store(a, t)
}.
```

Note that execution of `swap` would trigger undefined behavior if `a` and `b` do not point to Bedrock2 memory. Thus the specification of `swap` should include a precondition that rules out this possibility. Further, we want the specification to summarize the effects `swap` has on the memory: albeit already minimal, the implementation includes a temporary variable `t` which is of no interest to the caller. The ability to elide implementation details from specifications will be important for more complex functions, and we will see that even for `swap` there is some consequential flexibility in what to specify about it.

Sepcification The usual specification of `swap` is expressed in Bedrock2 as follows:

```
Instance spec_of_swap : spec_of "swap" :=
  fnspec! "swap" a_addr b_addr / a b R,
  { requires t m := m =* scalar a_addr a * scalar b_addr b * R;
    ensures  T M := M =* scalar a_addr b * scalar b_addr a * R ∧ T=t}.
```

`Instance` and the redundant type declaration on the first line make this specification available for later automatic lookup by Bedrock2 proof automation processing callers of `swap`. Arguments listed after the function name are implicitly universally quantified, and additional universally quantified specification variables can be listed after `/`. While program variables are always machine words, the specification variables can be of any types: here `a` and `b` are also machine words, but `R` describes the remaining memory left untouched by this program[1]. The I/O trace `t` and memory `m` in the precondition (`requires` clause) are universally quantified with fixed types, and `T` and `M` after `ensures` bind the trace and memory at the end of the execution of the function. Note that all variables universally quantified by the specification notation including `t` and `m` are still in scope after `ensures`.

Here the precondition states that memory `M` separately contains a (little-endian representation of) the machine word `a` starting at address `a_addr`, `b` at `b_addr`, and `R`. The postcondition is similar, but with `a` and `b` swapped! Let's prove that `swap` works:

[1] `R`, `m`, and `M` are explicitly included in specifications by choice; the formal semantics of Bedrock2 also satisfy a frame-rule theorem that allows proofs of equivalent execution with additional memory to be derived in a black-box manner.

```
Lemma swap_ok : fnspec_goal_for! swap.
Proof. repeat straightline; eauto. Qed.
```

Proof This was simple, but perhaps deceptively so: proofs of more interesting programs are unlikely to be dispatched this easily, especially during development. And as code that works as desired tends to get minimal attention, it is exactly the feedback generated during unsuccessful proof attempts that really makes up the user experience of a program-proof tool! One can try to prove a broken variant of swap using the same script:

```
Definition bad_swap := func! (a, b) {
  store(b, load(a));
  store(a, load(b))
}.
```

The following goal is generated, and it makes the issue apparent. So far, so good.

```
H1 : (scalar a_addr a * (scalar b_addr a * R))%sep m1
Goal: (scalar a_addr b * (scalar b_addr a * R))%sep m1 ∧ t = t
```

For more bite-sized examples, consider the case of "swapping" the data at an address with itself. `spec_of_swap` implicitly rules this out by joining the memory contents at `a_addr` and `b_addr` with `*`, so current proof does not cover this case. The implementation and proof script also work with the following specification[2]:

```
{ requires t m := m =* scalar a_addr a * R ∧ b_addr = a_addr;
  ensures  T M := M =* scalar a_addr a * R ∧ T = t }.
```

However, the following alternative implementation only satisfies the first spec:

```
Definition swap := func! (a, b) {
  store(a, load(a)+load(b));
  store(b, load(a)-load(b));
  store(a, load(a)-load(b))
}.
```

Attempting to prove it against the second leads to a proof context where the memory assumption describes the value at `a_addr` using an unwieldy expression in terms of word operations and intermediate values computed by previous lines. The failure mode can be learned by manipulating the proof state as an experiment. The standard

[2]That the same implementation satisfies both specifications does not mean that the precondition is redundant: `swap(a,a+1)` does not leave the word from address `a` at `a+1` but instead overwrites all but the last byte of it, violating the postcondition even if it is amended not to assert separation.

Coq tactics `subst` and `ring_simplify` reveal the limitation:

```
H2 : (scalar a_addr (word.of_Z 0) * R)%sep m2
Goal: (scalar a_addr a * R)%sep m2 ∧ t = t
```

3.1.2 Automating Proof of Optimized Arithmetic

The principles underlying the design of Bedrock2 proof automation seen in the `swap` example are chosen for the benefit of more challenging verification and custom proof scripts, so they are needlessly conservative in the context of `swap` alone. For example, having to manually `subst` before `ring_simplify` is superfluous for programs that can be neatly represented without sharing of subexpressions, and a case can be made for on-by-default simplification of expressions. But consider the following implementation of 128-bit addition using RISC-V (RV32I) arithmetic operations. Each add-with-carry step requires up to 5 instructions, carefully scheduled for less horrible performance.

```
Definition uint128_add := func! (s, a, b) ~> c { (* read columnwise *)
    a0 = load4(a); a = a+$4;
    b0 = load4(b); b = b+$4;
    s0 = a0 + b0;
        a1 = load4(a); a = a+$4;
        b1 = load4(b); b = b+$4;
        c1 = s0 < a0;
        s1 = a1 + b1;
            a2 = load4(a);
            c2 = s1 < a1;
        s1 = c1 + s1;
                             a = a+$4;
            c2p = s1 < c1;
            b2 = load4(b); b = b+$4;
            c2 = c2 + c2p;
            s2 = a2 + b2;
                a3 = load4(a);
                b3 = load4(b);
                c3 = s2 < a2;
            s2 = c2 + s2;
    store4(s, s0); s = s+$4;
                c3p = s2 < c2;
                c3 = c3 + c3p;
                s3 = a3 + b3;
        store4(s, s1); s = s+$4;
                    c = s3 < a3;
                s3 = c3 + s3;
            store4(s, s2); s = s+$4;
```

```
                    c4p = s3 < c3;
                store4(s, s3);
                    c = c + c4p
}.
```

In this case, sharing of subexpressions is essential to effective reasoning. Inlining all subexpressions after symbolic execution generates more than 200 lines of expression tree, irrecoverably obscuring the purpose of the program from human inspection. The same complications also affect proof scripts and decision procedures: applying `ring_simplify` to the inlined subexpressions takes multiple seconds (and is useless anyway). Complete transformation is not the only way to hamper human understanding of the intermediate proof states arising from this program: just losing the correspondence between program variables and proof-context variables would be enough, and even dropping the original ordering of statements is an obstacle.

I will walk through how this program is proven against the following specification:

```
Definition eval : list word -> Z :=
  List.fold_right (fun a s => word.unsigned a + 2^32*s) 0.

Instance spec_of_uint128_add : spec_of "uint128_add" :=
  fnspec! "uint128_add" ps pa pb / a b s R ~> c,
  { requires t m :=
      m =*> array32 pa a     ∧ length a = 4 ∧
      m =*> array32 pb b     ∧ length b = 4 ∧
      m =*  array32 ps s * R ∧ length s = 4;
    ensures T M := ∃ y,
      M =*  array32 ps y * R ∧ length y = 4 ∧
      T = t ∧ 2^128*c + eval y = eval a + eval b }.
```

Here `array32 pa a` holds the 32-bit words from Coq list `a` at address `pa`. The memory `m` may satisfy the three preconditions in arbitrarily overlapping ways. Only `R` from the precondition associated with the output array appears in the postcondition. The other inputs are just required to be contained (`=*>`) in the memory.

The proof starts by taking one `straightline` step to process the preconditions:

```
Lemma uint128_add_ok : fnspec_goal_for! uint128_add.
Proof.
  straightline; repeat straightline_cleanup.
```

Then a custom proof script is used to break lists of a known length into individual elements, naming them after the list itself but with subscripts for indices:

```
  repeat match goal with
```

```
  | H : length ?l = _ |- _ =>
      let x := fresh l "_0" in destruct l as [(*nil*)|x l]; invert H
  end; unfold array in *.
  repeat straightline.
  exists [s0; s1; s2; s3]; ssplit; try ecancel_assumption; trivial.
```

The repeated calls to `straightline` complete the symbolic execution of the function, leaving only the postcondition to be tackled. The next line specifies the value of `y` required by the postcondition and dispatches all clauses except for `eval`. However, unlike swap, the specification of big-integer addition is substantially different from the implementation, so we are only halfway there still. But real progress has been made: arithmetic expressions in the proof context contain a purely functional description of the algorithm, compatible with normal Coq proof techniques. Variables like `a_0` were generated above and refer to members of input lists, variables like `s1'0` refer to past values of local variables of the Bedrock2 program, and the current values of program variables are presented with their names preserved:

```
s0 := word.add a_0 b_0
c1 := if word.ltu s0 a_0 then word.of_Z 1 else word.of_Z 0
s1'0 := word.add a_1 b_1
c2'1 := if word.ltu s1'0 a_1 then word.of_Z 1 else word.of_Z 0
s1 := word.add c1 s1'0
c2p := if word.ltu s1 c1 then word.of_Z 1 else word.of_Z 0
c2 := word.add c2'1 c2p
s2'0 := word.add a_2 b_2
c3'1 := if word.ltu s2'0 a_2 then word.of_Z 1 else word.of_Z 0
s2 := word.add c2 s2'0
s'0 := word.add ps (word.of_Z 4)
c3p := if word.ltu s2 c2 then word.of_Z 1 else word.of_Z 0
c3 := word.add c3'1 c3p
s3'0 := word.add a_3 b_3
s'1 := word.add s'0 (word.of_Z 4)
c'0 := if word.ltu s3'0 a_3 then word.of_Z 1 else word.of_Z 0
s3 := word.add c3 s3'0
s := word.add s'1 (word.of_Z 4)
c4p := if word.ltu s3 c3 then word.of_Z 1 else word.of_Z 0
c := word.add c'0 c4p
Goal:
 2^128*c+eval[s0;s1;s2;s3]=eval[a_0;a_1;a_2;a_3]+eval[b_0;b_1;b_2;b_3]
```

Individual manipulation of these equations still appears undesirable as a proof strategy. It would be convenient to use an off-the-shelf solver, but the hypotheses do not fall into any supported theory. In particular, addition is linear, but general comparison as an expression (thresholding) does not fall under linear arithmetic even though inequalities do. Case analysis on a carry bit could be used to to turn the goal with

a comparison into two goals with different inequalities as hypotheses. If each case would be proven automatically, solving 2^7 cases could be manageable in a purpose-built tool (it takes 48s in Coq). In case the verification does not succeed, debugging a case-analysis-based proof is more challenging than debugging than a direct proof because each case could succeed by either proving its goal or demonstrating a contradiction among the hypotheses, and there is no general way to know ahead-of-time which outcome the proof would reach if it worked.

Instead, we notice that each comparison is just used to compute the carry bit from the previous addition: overflow occurred iff the output is smaller than the input.

```
  Lemma ltu_as_carry (a b : word) (s : word := word.add a b)
    (c : word := if word.ltu s a then word.of_Z 1 else word.of_Z 0)
    : word.unsigned c = (a + b) /  2 ^ 32.
  Proof. subst s c.
   rewrite word.unsigned_ltu. destr (word.add a b <? a); ZnWords. Qed.
```

Instantiating this lemma for each hypothesis using another ad-hoc `match` allows the goal to be proven using the general-purpose solver for linear integer arithmetic.

```
  repeat match goal with c := if word.ltu _ _ then _ else _ |- _ =>
    pose (ltu_as_carry _ _ : word.unsigned c = _); clearbody c end.
  unfold eval, List.fold_right; ZnWords.
Qed.
```

Appropriate tooling makes codeveloping programs and correctness proofs a breeze. The first prototype of this function was finished in less than two hours, including the correctness proof and the time taken to look up and understand how to handle carry chains of word-sized addition without hardware support. For this example, the Bedrock2 proof experience was smooth enough to support the prototyping itself: the easiest way to tell if an optimization was correct was to try to prove it. (The final proof script presented here is based on hindsight from that process and omits needless detours explored along the way, whether or not they resulted in successful proofs.)

I will return to considerations and remaining challenges in how to present verification conditions to program-correctness proofs in Section 3.5. But first, a deep dive into mechanics connecting the syntax trees of imperative Bedrock2 programs to proof contexts in Coq's purely functional language is in order.

3.2 Definition of Programmer-Facing Semantics

The examples so far enjoyed completely automatic symbolic execution, including proofs that the Bedrock2 programs cannot trigger undefined behavior when called as

specified. But what is the formal statement that is being proven when `straightline` steps forward through the commands of a function, and what can be concluded on the basis of it later? This section gives a thorough answer to this question, starting from proof-context-management building blocks and culminating with definitions of `spec_of` and (informally) the macros `fnspec!` and `fnspec_goal_for!`.

3.2.1 Variable Assignments

The simplest command we want to verify is a variable assignment, `x = e`. We want the symbolic execution of this command to add `x := e : word` to the proof context, and to have the context variable `x` be referenced in the symbolic execution of the following commands. If we were verifying a functional Coq program instead of a Bedrock2 program, the proof context would be updated in the described manner when running `intros x` on a goal of the form `let x : word := e in Goal'`. As Bedrock2 local variables refer to mutable locations, we cannot represent the as Coq variables directly, but updating a local-variable map can be expressed with a similar goal: `let l := map.put l x v in Goal'`, where `v` is the result of evaluating `e`.

To evaluate the expression `e`, the current local-variable map `l` and memory `m` are required. Evaluation of Bedrock2 expressions may also trigger undefined behavior (when accessing an uninitialized local variable or loading an unavailable address), so the appropriate model for it in Coq is a deterministic partial relation rather than a function. Returning to the verification condition of the assignment statement, we now have `∃ v, dexpr m l e v ∧ let l := map.put l x v in Goal'`. To achieve the desired proof context, `straightline` let-binds values whose existence it proves. In this case it generates the name of the new Coq binder for `v` from the string `x` that appears in the `set` command and, if necessary, renames the existing proof-context variable with the same name (from a previous assignment) using `'`.

Having figured out one case of the programming-facing definition of Bedrock2 semantics, it is high time to give it a type signature and a name. First attempt:

```
Definition cmd_ok (c : cmd) (m : mem) (l : locals) : Prop := (* bad *)
  match c with
  | cmd.set x e =>
    ∃ v, dexpr m l e v ∧
    let l := map.put l x v in
    Goal'
```

The command, memory, and local-variable map make sense as inputs to an execution, but `Goal'` is unbound. We can work out another case to find a replacement.

3.2.2 Sequencing and Conditionals

If this assignment is the last command of the function and assigns a value to the return variable, it would make sense to assert that `v` satisfies the function postcondition, but what if it is not? For the command `x1 = e1; x2 = e2`, we want a goal of the form

```
∃ v1, dexpr m l e1 v1 ∧ let l1 := map.put l x1 v1 in
  ∃ v2, dexpr m l e2 v2 ∧ let l2 := map.put l x2 v2 in Goal''
```

The second line in the goal above appears exactly in the place of `Goal'` in the attempted definition. Indeed, the right thing to prove about an intermediate command is that the remaining commands execute as desired! If we only wanted to handle lists of commands and no other nesting like loops and conditionals, we could have `cmd_ok` take this list as an argument and replace `Goal'` with a recursive call:

```
Fixpoint cmds_ok (cs : list cmd) (m : mem) (l : locals) : Prop :=
  match cs with
  | cons (cmd.set x e) cs' =>
    ∃ v, dexpr m l e v ∧
    let l := map.put l x v in
    cmds_ok cs' m l
```

The goal to be proven for a conditional expression `cond e ct cf` includes evaluation of the test expression and then the execution of one of the two commands depending on which branch is taken. One might want to write something like this:

```
  | cons (cmd.cond e ct cf) cs' => (* bad *)
    ∃ v, dexpr m l e v ∧
    cmds_ok ((if word.eqb v 0 then cf else ct) ++ cs') m l
```

The accumulation of later commands is increasingly inelegant, is not actually well-typed with the current `cmd` type, and would require a termination proof for `cmds_ok` in Coq. A key refactor is enabled by the observation that the commands `cs'` being appended to the branch case are a first-order representation of the *continuation* that describes where symbolic execution should resume after the branch is evaluated. Changing `cmd_ok` to a higher-order function that takes the continuation as an argument leads to the basic structure used in Bedrock2: a continuation-passing-style interpreter returning a single verification condition and using binders to represent variables that `straightline` should place in the proof context. With notations:

```
Definition cmd_ok (c : cmd) (m : mem) (l : locals)
                  (post : mem -> locals -> Prop) : Prop :=
  match c with
  | cmd.skip => post m l
  | cmd.set x e =>
```

```
    bind_ex v <- dexpr m l e;
    dlet! l := map.put l x v in
    post m l
| cmd.seq c1 c2 =>
    cmd_ok c1 m l (fun m l => cmd_ok c2 m l post)
| cmd.cond br ct cf =>
    bind_ex v <- dexpr m l br;
    (word.unsigned v <> 0 -> cmd_ok ct m l post) ∧
    (word.unsigned v  = 0 -> cmd_ok cf m l post)
...
```

3.2.3 Forward Reasoning Using Weakest Preconditions

From a logical perspective, the previous code snippet implements a predicate transformer that returns the weakest precondition that needs to hold before the execution of the command for the postcondition specified as the argument to hold after it. For example, the command that does nothing can only satisfy the postcondition if it is already satisfied, and the postcondition of the first command in a sequence is that the next command executes as desired.

During symbolic execution of a Bedrock2 program, the proof context contains the givens (variables and hypotheses), which together represent a postcondition of the commands processed so far. Amongst these variables are the local-variable map and the memory of the Bedrock2 program. The values of these can be defined in terms of variables universally quantified in the theorem statement, current and past values of individual local variables introduced by `straightline`, and constants in the Coq environment. Hypotheses include those generated from `cmd_ok` by `straightline`, for example posited outcomes of branch tests, and user-specified preconditions applied to the *initial* local variables and memory. Additionally, Section 3.3 describes how `straightline` propagates memory-related hypotheses across memory modifications.

The postcondition in the proof context is not necessarily the strongest for a number of reasons. Most plainly, the user-written proof script may have simply cleared some hypotheses. More fundamentally, symbolic execution of function calls relies on user-provided specifications, and leaving out details in specifications is useful to enable specification-preserving refactoring and optimization of function implementations.

The goal contains the postcondition-specific weakest precondition, `cmd_ok c m l P`. Each `straightline` step makes partial progress towards proving that the current proof context implies this weakest precondition by evaluating `cmd_ok` one `match`-step at a time and discharging the obligations it returns. In this sense, Bedrock2 uses a weakest-precondition definition to facilitate forward reasoning.

This approach has multiple advantages that may make it appealing for other lan-

guages as well. First, after the basic structure was established, extending `cmd_ok` for additional constructs has been straightforward: the question this definition answers is exactly "what should be proven about this construct?". Second, the definition of `cmd_ok` serves as a recipe for maintaining `straightline`: this tactic should specialize `cmd_ok` to the program at hand, recognize the generated state updates and proof obligations, and match them against the symbolic state in the proof context. Finally, we will see that this style allows for direct proofs of total correctness in presence of nondeterminism arising from abstraction and runtime input.

3.2.4 Function Calls and Functions

Recall that functions in Bedrock2 can take any number of machine words as arguments and can return the values of any number of local variables. A common use case of this facility is for functions to return a separate error code separately from the main output, for example `pktlen, err = recvEthernet(buf)`. The weakest-precondition definition handles this like an interpreter would: the arguments are evaluated in the caller's context, and the return values are assigned to the appropriate local variables. Note that only `args` and not all local variables `l` are passed into `call`, specifying lexical scoping; `rets` are similarly assigned left-to-right:

```
| cmd.call binds fname arges =>
  bind_ex args <- dexprs m l arges;
  call fname m args (fun m rets =>
    bind_ex_Some l <- map.putmany_of_list_zip binds rets l;
    post m l)
```

As Bedrock2 does not allow for recursion to allow for simple, sound, and predictable stack-usage analysis, we can define the verification conditions for a function call the same way an interpreter would be written and still get a total structurally recursive function that is easy to manipulate in proofs:

```
Fixpoint call (functions : list (string * func)) f m args post :=
  match functions with
  | nil => False
  | cons (declname, (argnames, retnames, c)) functions' =>
    if String.eqb declname f
    then bind_ex_Some l <- map.of_list_zip argnames args;
         cmd_ok (call functions') c m l (fun m l =>
             list_map (get l) retnames (fun rets : list word =>
             post m rets)).
    else call functions' f m args post
  end.
```

Unlike `cmd_ok`, `call` is not unfolded by `straightline`: doing so would lead to the

body of the callee being processed again, but we want modular verification! Instead, after the evaluation of argument expressions has been processed, `straightline_call` looks for an assumption that proves `call` for the concrete function name in question with any postcondition, likely under some universal quantifiers about arguments and preconditions about these values. Then a weakening lemma is used to connect the assumed `call` to the one required by the weakest-precondition goal: to prove `call` with postcondition Q given `call` with postcondition P, it suffices to prove that P implies Q. The proof of that implication will later proceed by symbolic execution of the remaining commands in the caller under the conditions that the callee's postcondition ensures. The appropriate assumption about the callee is exactly the function specification the programmer wrote for it. Indeed, `fnspec!` is just a notation for `call`! Notation by Clément Pit-Claudel, simplified here:

```
Notation "'fnspec!' f a0 .. an ,
          { 'requires' m := pre ; 'ensures' M := post }" :=
  (fun functions => (forall a0 .. an, forall m, pre ->
     call functions f m [a0 .. an] (fun M rets => rets = [] ∧ post))).
```

For example, an invocation of `swap` (Subsection 3.1.1) is proven automatically:

```
Definition swap_swap := func! (a, b) { swap(a, b); swap(a, b) }.

Instance spec_of_swap_swap : spec_of "swap_swap" :=
fnspec! "swap_swap" a_addr b_addr / a b R,
{ requires t m := m =* scalar a_addr a * scalar b_addr b * R;
  ensures  T M := M =* scalar a_addr a * scalar b_addr b * R ∧ T = t}.

Lemma swap_swap_ok : fnspec_goal_for! swap_swap.
Proof. repeat (straightline || straightline_call); eauto. Qed.
```

Removing `straightline_call` from the proof script has the symbolic execution stop at the call site, with goal `call functions "swap" t m [a_addr; b_addr] (...)`. Manually invoking `straightline_call` at this point would generate two subgoals: one for proving that the preconditions of `swap` are satisfied and one for symbolic execution of the next call to `swap`. This allows for straightforward invocation of custom proof automation or for interactive verification of preconditions for which an automated proof procedure is not available.

3.2.5 Linking Correctness Proofs of Functions

The correctness proofs of `swap` and `swap_swap` are independent and can be processed in any order or even in parallel. Further, the code and proof of `swap_swap` do not even depend on the *implementation* of `swap`, and the latter just depends on `spec_of_swap`. This is great for modularity, as changes in `swap` that can be proven against the same

specification do not require the proof of `swap_swap` to be revisited. The flip side of this indirection is that proving `swap_swap_ok` is not all that needs to be proven to ensure safe execution of the function, and another "linking" proof is required to show that all dependencies are present and appropriately verified:

```
Lemma link_swap_swap : spec_of_swap_swap [&,swap_swap; &,swap].
Proof. eauto using swap_swap_ok, swap_ok. Qed.
```

Instantiating `spec_of_swap_swap` with the explicit list of functions unfolds to the goal `call [&,swap_swap; &,swap] "swap_swap" t m [a_addr; b_addr] (...)` with the precondition and postcondition listed in `spec_of_swap`, and the proof confirms that the two functions indeed work together as expected. (The macro `&,` makes the Coq names of the definitions available to `call`). Individually, the statement proven as `spec_of_swap_swap` instead *assumes* that calling `swap` in the environment that `swap_swap` is in works as expected, and it concludes that `swap_swap` also works:

```
forall functions : list (string * func),
  spec_of_swap functions -> spec_of_swap_swap (&,swap_swap::functions)
```

This premise was added by the convenience macro `fnspec_goal_for!` used in `Lemma` declarations, which just looks up the specifications of the functions called by the current function and adds premises about their correctness to the lemma. This gathering of callees is not recursive: if a later change to `swap` added a dependency, its proof (and the linking lemma) would need to be revisited, but `swap_swap` and its proof would remain as-is.

An alternative to this design would be to fix the complete list of function specifications up-front and to prove both `swap_ok` and `swap_swap_ok` against that list. In VST without Verified Software Units, this list is customarily called `Gprog`, and it is used to support verification of mutually recursive functions in the sense of partial correctness (an infinite loop satisfies every specification). Bedrock2 `call` peeling off the current function from the list before descending into its body disallows recursion and thus rules out the possibility of nonterminating chains of function calls. More importantly, the same proof of a Bedrock2 function can be reused easily between usecases without rechecking it as they depend on only the individual specifications of the callees of the function, not the list of all functions that would be executed. (A final linking lemma ensuring that all functions were verified against the required specifications is needed either way.) It would be interesting to explore if Bedrock2's fine-grained approach could be extended to blocks of mutually recursive functions while maintaining the simple list-of-functions structure and the guarantee that the verified programs terminate (for modeling recursion, see Subsection 4.1.1 and [Cha+23, §2.1]).

3.2.6 Nondeterminism: Input and Stack Allocation

The appropriate verification conditions for run-time input and output are easily expressed in the language of weakest preconditions. This subsection will briefly review the key construction for completeness. Consider the following snippet from the SPI device driver of the Bedrock2-Kami "lightbulb" case study (Section 4.5):

```
io! busy = MMIOREAD($0x1002404c); (* poll receive FIFO *)
if !(busy >> $31) {
  b = busy & $0xff;
```

The highest bit of the value read from this MMIO register indicates whether the FIFO this register provides access to is empty, and if not, the lowest 8 bits contain an element popped from that FIFO. The most important risk in a FIFO-based driver is losing synchronization: accidentally reading one extra element too much or too little would make the meaning of the future inputs ambiguous. Thus it is critical that the specification of this program would cover both cases and constrain the behavior of the program differently depending on the value returned by `MMIOREAD`. The same should apply even if this value is not immediately branched on.

The solution is threefold. First, the weakest-precondition definition for MMIO will need to ensure that the address being read is usable for that purpose. Second, the continuation is invoked under a `forall` quantifying over the value returned by the MMIO read, and the rest of the program needs to be proven correct in all cases. Finally, both the address and the value are recorded in an event trace `t` that is threaded through the semantics just for the modeling of external calls:

```
| cmd.interact [x] "MMIOREAD" [e] =>
  bind_ex addr <- dexpr m l e;
  isMMIO addr ∧
  forall value : word,
    let l' := map.put l x value in
    let t' := cons ([addr], [value]) t in
    post t' m l')
```

The actual specification of `cmd.interact` is parametrized over an `ext_spec` of the same type signature as `call`, allowing for nondeterministc modeling of external calls with arbitrarily many arguments, return values, preconditions, postconditions, and even modifications to Bedrock2 memory.

Stack Allocation The contents of stack-allocated memory are unspecified. Formally, this is modeled as each `stackalloc` nondeterministically choosing the contents and address of the freshly allocated memory. Supporting internal nondeterminism is easier than supporting external nondeterminism: the choices are simply not included

in the event trace.

However, there is more to stack allocation than nondeterminism: the semantics need to specify that new memory is available after stack allocation, but only until the nested command finishes execution. This is required to allow the compiler to reuse the same memory for the control stack of a subsequent function call or to fulfill the next allocation request. This is accomplished by expressing the memory available to the nested command as the disjoint union of the stack-allocated region and the ambient memory *before and after* its execution, removing whatever bytes reside in the stack-allocated region in the latter case.

3.2.7 Loops, Termination, Invariants, and Specificatons

Writing the weakest-precondition definition as a function that recurses over the structure of the command is limiting for specifying loops, but the limitations are workable. What we cannot do is to unroll the loop and give `while e c` the same semantics as `if (e) { c; while e c }` – inlining the rules for conditionals and sequencing is possible, but calling `cmd_ok` on `while e c` again in the case for `while e c` is not allowed. Indeed, if the loop does not terminate, the very definition of `cmd_ok` for this loop would be circular and thus not acceptable in Coq's logic. Instead, the current design of Bedrock2 is to enforce termination: `cmd_ok` should be false (and provably so!) for a nonterminating program.

I will present a general solution for enforcing termination in a weakest-precondition-style specification of a programming language in Chapter 4, but for verifying concrete programs, it suffices to require a loop invariant and a boundedly decreasing measure. To maximize flexibility for program proofs, the programmer gets to choose any well-founded relation on any type and relate it to the state of the Bedrock2 program in a context-appropriate manner. For this to be sound (e.g., to rule out the "relation" that just allows any measure in any state), the proof obligation for continuing to another iteration of the loop has to require that *every* measure `v` that can be related to the starting state can decrease further:

```
forall v t m l, invariant v t m l ->
  bind_ex b <- dexpr m l e;
  (word.unsigned b <> 0 -> cmd_ok c t m l (fun t' m' l' =>
    ∃ v', invariant v' t' m' l' ∧ v' < v)
```

Advanced Loop Rules: Loops as Tail-Recursive Functions Following the example of Bedrock [Chl13], Bedrock2 provides the option to verify loops using a precondition and postcondition instead of an invariant. While an invariant answers the question "what always holds at the beginning of each loop iteration," the alternative loop specification answers "if this loop was refactored into a dedicated tail-recursive

function, what would its precondition and postcondition be?" As for normal functions, universally quantified initial variables and specification variables can appear in both the precondition and the postcondition, allowing the two to be related for a generic state.

The alternative perspective is particularly useful for loops whose later iterations consider a subset of the data needed by earlier iterations. For example, the following function checks whether two memory regions have equal contents with data-independent memory access and control flow:

```
Definition memequal := func! (x,y,n) ~> r {
  a = $0;
  while n {
    a = a | (load1(x) ^ load1(y));
    x = x + $1; y = y + $1; n = n - $1
  };
  r = a == $0
}.
```

A tail-recursion-style loop specification is possible and convenient even though the accumulator `a` depends on the data considered in past loop iterations: the postcondition can just relate the final accumulator (`A`) to the initial one (`a`).

```
fun (v : nat) xs ys => (* spec. variables; precondition: *)
fun   t m (x y n a : word) => (m =*> bytes x xs ∧ m =*> bytes y ys ∧
  v = n :> Z ∧ length xs = n :> Z ∧ length ys = n :> Z,
  fun T M (X Y N A : word) => (* postcondition: *)
    m = M ∧ t = T ∧ ∃ x, (x = 0 ↔ xs = ys) ∧ A = Z.lor a x :> Z)
```

The overall postcondition of `memequal` is then easily proven from the loop postcondition: `r = 1 ↔ xs = ys`. (The Bedrock2 loop lemma used here actually differs slightly from the one in Bedrock in that the Bedrock version has the verification of the post-loop code proceed from the loop precondition and the fact that the loop exited, whereas the Bedrock2 version uses the loop postcondition as the premise of that goal).

3.3 Separation Logic in Bedrock2

Bedrock2 semantics for (single-byte) memory access are just `map.get` and `map.get`, but verifying programs that access memory at non-constant addresses requires much more care than threading the local-variable map through `straightline`. The challenge is that `map.get` and `map.put` do not commute in general, just only when they are applied to different addresses, so correctness of even simple programs like the

second `swap` from Subsection 3.1.1 relies on administrative preconditions asserting disjointness between relevant memory regions.

The strategy is simple and standard: instead of reasoning directly about `map.put` and `map.get`, all places where the memory is mentioned will be reworded to treat it as a disjoint union of parts. This includes the symbolic state in the proof context, function preconditions and postconditions, and lemmas about memory access. For example, `swap` requires `m =* scalar a_addr a * scalar b_addr b * R`. Here `*` can be taken to mean the (partial) disjoint-union operation on maps, with `=*` asserting that the operation was successful and resulted in `m`. It is conventional to generalize the definition of `*` to predicates over memory fragments: the memory fragment `t` satisfies (`=*`) `scalar a_addr a` iff it contains exactly one machine word with value `a` starting at address `a_addr`. For the purposes of this section, either perspective works; for a more elaborate introduction to separation-logic predicates see [Cha20a].

3.3.1 Cancellation

The fundamental operation on separation-logic expressions is a cancellation step, that is identifiying and removing a given clause from a larger separation-logic formula. For example, symbolically executing `store(a_addr, t)` against a memory satisfying `m =* scalar a_addr a * scalar b_addr b * R` should result in a memory where parts disjoint from `scalar a a_addr a` are preserved but the clause itself is first removed (cancellation) and then replaced with `scalar a_addr t`. Generally, cancelling `X` from `A` means to find `Y` such that `X*Y` is equivalent to `A`; I will write `X*Y` $\Leftrightarrow$ `A`.

As cancellation will be performed for each memory access and for each `*` argument in the precondition of a function call, it has been a performance bottleneck in past verification efforts. This section will show how to implement the cancellation step on arbitrary separation-logic formulas in a single proof-engine operation plus a controlled partial evaluation. This implementation is used throughout Bedrock2 separation-logic proofs and accounts for a miniscule fraction of the total proof time. The implementation technique resembles the last "denotation" step of reflection-based rewriting procedures but does not require a deeply embedded language of all allowed separation-logic formulas; I call it shallow reflection.

Separation-logic formulas where `*` appears in a canonical associativity will be represented as lists of disjoint parts using the following wrapper.

```
Fixpoint seps xs :=
  match xs with
  | cons x xs => x * seps xs
  | cons x nil => x
  | nil => empty
  end%sep.
```

It will be an invariant of the cancellation procedure that each separation-logic formula inside the list argument to `seps` does not contain any `*` at the top level. Under this assumption, canceling any clause can be expressed as an application of the following lemma:

```
forall n xs, seps (remove_nth n xs) * nth n xs ⇔ seps xs
```

Three implementation tasks remain. For example, for `xs=[A; X; B; C]` and `n=1`:

1. Shallow reification: finding `[A;X;B;C]` s.t. `seps [A;X;B;C] = A*X*B*C`.
2. Matching: identifying `n` s.t. `nth n xs = X`.
3. Controlled denotation: turning `seps (remove_nth 1 [A;X;B;C])` into `A*B*C`.

It is important to keep efficiency in mind for each step. However, there is no need to shoot for optimality: the overall procedure will apply (refine with) the above lemma, which creates a new goal in the proof engine, so the goal is for each operation to be not much slower than that.

Shallow Reification A recursive proof script can walk the separation-logic formula, translating `*` to `cons`. As a premature optimization, typechecking of the list is deferred until the lemma application. Note again that the formulas themselves are left unchanged.

Matching A recursive proof script will walk the list, returning `0` if the current element matches and `1 +` the recursive call if not. The key choice is how an element `A` is matched against `X`: syntactic equality is fast to check but insufficient for calling `swap` because while `a_addr` is a function argument, `a` is not, and its value needs to be determined during cancellation of `scalar a_addr ?a` from the caller's symbolic state. Genaral unification modulo computation rules of Coq terms is undecidable, and the built-in heuristic implementation can be arbitrarily slow in practical cases. For Bedrock2, I implemented unification modulo unfolding of proof-context variables but no computation; it has served the needs of the project.

Controlled Denotation The main challenge in this step is keeping `A`, `B`, and `C` unchanged. This means that a generic evaluation strategy for Coq terms is unsuitable, as it might peform computation inside these clauses. For example, if `B` is `scalar b_addr (length [1;2;3;4])`, the definition of `scalar` should not be unfolded, and the `length` should not be reduced to `4` as that might break a user-written proof script that later searches the context for pointers to lengths. We can get a step closer by

using the configurable reduction tactics with an explicit list of definitions to unfold (here just `seps` and `remove_nth`). However, if `remove_nth` also appears inside `A`, `B`, or `C`, these occurrences would still be extraneously unfolded. This problem is bypassed by creating dedicated copies of the list operations (transitively) used by cancellation and using controlled reduction tactics unfold these new definitions only; for example `Definition myapp T := Eval cbv delta in @List.app T` and `cbv [myapp]`.

3.3.2 Cancellation During Symbolic Execution

In practice, preconditions do not appear as neat flat lists like `A*X*B*C`, but appear in arbitrary associativities such as `A*(X*B)*C`. As Bedrock2 provides a reliable implementation of cancellation, there is no good reason why even an ad-hoc proof should depend on the associativity of the separation-logic formulas. Thus, Bedrock2 separation-logic tactics eagerly flatten all separation-logic goals and hypotheses and keep them in `seps` form during separation-logic tactic execution. The flattening is implemented similarly to cancellation itself: shallow reification, a verified tree-to-list function operating on `seps`, and controlled denotation.

As mentioned briefly in the discussion of matching above, separation-logic formulas on which cancellation is performed can contain existential variables `?e` whose values can be determined during cancellation. This means that cancellation could succeed on the same goal with multiple different results, and thus cancelling multiple clauses from the same formula may depend on the order of these clauses. While scenarios where this flexibility occurs are often unintentional, they should not cause cancellation to fail when a solution is possible. For a degenerate example, cancelling `?X` and `A` from `A*B` should succeed (with `?X:=B`), but if `?X` were cancelled with `A` or `A*B` first, cancellation would fail needlessly.

To properly tackle this problem, cancellation as used in symbolic execution works on separation-logic equivalences with multiple clauses on each side, for example `A*B` $\Leftrightarrow$ `?X*A`. At each step, the algorithm picks one clause from each side and cancels them with each other. Three rules are used to ensure that a solution is found if one exists:

1. All cancellations of clauses without evars are performed before any cancellation of clauses with evars. It is safe to perform these cancellations greedily – if the goal is provable at all, it is provable after canceling two identical clauses from both sides.

2. A clause that *is* an evar (like `?X`) is only cancelled last, and it is then cancelled with the entire other side of the equivalence.

3. Other evar-instantiating cancellation steps can be backtracked if cancellation (or, optionally, a later proof step) fails after a tentative choice. This is imple-

mented using Coq's built-in support for multi-success tactics, so the scope of backtracking can be controlled by writing `once (ecancel; picker); next`.

Troubleshooting Cancellation Failures Proofs of programs whose separation-logic clauses include complex expressions can sometimes have a cancellation fail where the programmer expects it to succeed. The reason is often a minute difference, which may even be omitted from display by normal notations and might be resolvable through computation alone (but cancellation uses syntactic matching, not unification modulo computation rules). To assist with troubleshooting these situations, Bedrock2 provides the tactic `cancel_seps_at_indices` which takes two natural numbers as arguments and generates a subgoal asserting that the separation-logic clauses at these indices are equal. Interactive debugging can then proceed as for any equality proof, perhaps by using the tactic `f_equal` to compare the function symbols at the heads of both sides of the equality and generate subgoals for pairwise equalities of their arguments. Use of this tactic with literals as arguments is not recommended in finished proof scripts as, changes in the order of separation-logic clauses could break the proof, but as nothing in Bedrock2 actually shuffles the clauses around without a good reason, some users report success with this approach as well.

3.3.3 Separation-Logic Rewriting

A common task during proofs of memory-manipulating programs is replacing some subset of separation-logic clauses in the symbolic state with a logically equivalent but syntactically different description of the same memory. For example, the same memory region may be equivalently understood to contain a 4-byte array, two 2-byte arrays, or a 32-bit machine word. The first published iteration of the Bedrock system [Chl11] supported explicit annotations in program code to call out such equivalence transformations on the symbolic state; later [Chl13] similar instructions could be specified as hints for using particular preconditions and proving particular post-conditions.

These equivalences can be stated using $\Leftrightarrow$, proven as lemmas, and used to prove other lemmas, but manually instantiating lemmas does not lead to a satisfactory proof experience. Even lemmas stated with superfluous universal quantifiers and equational premises constraining their values cannot be always matched automatically. Consider trying to merge the two consecutive byte arrays in the following separation-logic formula: `(A * bytes p [a;b] * B) * C * (D * bytes (p+2) [c;d])`. No syntactic subexpression here would match the left-hand side of an array-concatenation lemma regardless of whether the arguments to the lemma are already known, so rewriting tactics provided with Coq are not applicable.

Instead, Bedrock2 implements rewriting using cancellation. Concretely, rewriting

the symbolic state `S` with a lemma establishing `X` ⇔ `Y` is performed by repeated cancellation on the goal `S` ⇔ `X*?R`, after which `Y*R` is derived using the congruence rule of `*` and ⇔.

To support lemmas with quantifiers, a recursive tactic script automatically instantiates all quantifiers with existential variables `?E`. Cancellation matching these variables against the symbolic state automatically determines appropriate values. A variant of the rewriting tactic is provided that attempts to solve the non-separation-logic premises of the lemma using another tactic that is passed in as an argument, backtracking cancellation decisions if unsuccessful. Left-to-right rewriting is supported by wrapping the lemma in the macro `symmetry!`.

Lifting Predicates and Logical Quantifiers If separation-logic formulas are allowed to contain existential quantifiers and pure predicates, they can be lifted to the top-level position by rewriting with an appropriate lemma. Further, Samuel Gruetter contributed a dedicated set of lemmas in the same style as the key lemma for cancellation among `seps`, which allow, for example, instantiating an existential variable in the nth clause of the current goal without reordering the clauses.

3.3.4 Cancellation Performance

The separation-logic automation described in this section is fast enough that it is not a bottleneck in any Bedrock2 proof despite ubiquitous use. To sanity-check that it also achieves the expected performance on large inputs, I implemented the cancellation microbenchmark from the paper on Bedrock with MirrorShard [Mal+13, Fig. 4] in Bedrock2 and measured performance of both implementations (with Coq 8.4 and 8.15 respectively). As this microbenchmark works on concrete linked-list segments, it does not evaluate performance of evar instantiation, side-condition solving, or backtracking (which is not supported in MirrorShard).

Canceling 256 linked-list elements (the largest reported input) takes about 0.5s in Bedrock2 (including reification and denotation) and about 0.5s in Bedrock with MirrorShard (excluding reification and denotation). For comparison, the baseline Ltac implementation in [Mal+13, Fig. 5] is *asymptotically* slower, reaching 1s on a slightly different benchmark before input size 16. Further, it appears that the MirrorShard implementation of the benchmark distinguishes individual linked-list elements using *unary* natural numbers, and changing it to use binary integers instead makes it take just 0.2s – more than twice as fast as Bedrock2. (Using opaque identifiers as in the Bedrock2 version does not appear practical given the architecture of MirrorShard, but I would not expect substantial additional speedup from it.)

`Qed`-time performance results are similar: the MirrorShard implementation which is based on (deep) reflection takes only 0.7s, while `Qed` after the Bedrock2 cancellation

microbenchmark takes 1.8s for the reification alone and an additional 1.7s for cancellation. (Setting `Strategy -100` for the identifiers unfolded during cancellation did not improve these times.) Thus I conclude that Bedrock2 separation-logic automation is a couple of times slower than MirrorShard while requiring two orders of magnitude less infrastructure code. In absolute terms, both are fast enough on realistic inputs.

3.4 Reasoning With Word Arithmetic

This section will elaborate on a collection of arithmetic-related challenges that continue to affect the program-proof experience in Bedrock2 and developments that build upon it. I report on substantial but not entirely satisfying progress in each case and briefly discuss potential future directions.

Proof obligations about word arithmetic are ubiquitous during verification of Bedrock2 programs. Automation for discharging these goals has improved drastically during (and to some degree because of) the development of the Bedrock2 ecosystem but does not reach the level of effectiveness as attributed to comparable subsystems in SMT solvers. This section will give an overview of the progress achieved so far and future directions that may be worth pursuing. Just to illustrate how pressing this issue is, below is the precondition and code for a simple example, the transmit path in the Ethernet driver, with a numbered annotation `(*1*)` for each location corresponding to a proof obligation requiring word-arithmetic reasoning:

```
fnspec! "lan9250_tx" p l / bs ~> err,
{ requires t m := m =*> bytes p bs ∧
    unsigned l = length bs ∧ unsigned l mod 4 = 0; ... }.

Definition lan9250_tx := func! (p, l) ~> err {
  err = lan9250_writeword((*1*)$TX_DATA_FIFO, $(2^13)|$(2^12)|l);
  require !err;
  err = lan9250_writeword((*2*)$TX_DATA_FIFO, l);
  require !err;
  while ($3 < l) {
    err = lan9250_writeword((*4*)$TX_DATA_FIFO, load4((*3*)p));
    if err { l = $0 (*5*) } else {
      p = p + $4;
      l = l - $4
      (*6*) (*7*) (* 8 *) (*9,10,11,12*)
    }
  } (*13* | *14*)
}.
```

Each call to a function that takes an address requires a proof that the address is

within the appropriate range (obligations 1-4). Proof obligation 5 requires that the buffer whose length is `0` mod `4` and no larger than `3` has no bytes remaining. Word-arithmetic proofs 6 and 7 establish the loop invariant for the address and length of the remainder of the packet, and instance 8 shows that the remaining length decreases. Obligations 9-14 are similar but arise from the postcondition about the MMIO trace generated by this function. Some of these goals are quick to prove by hand (1, 2), but others require the full conceptual strength of integer-linear arithmetic (5, 8). No easy workaround appears to apply.

3.4.1 Reducing Modular Arithmetic to Integer Arithmetic

When I started prototyping the components of what became the ecosystem I am now writing about, Coq had no support for automatically proving goals involving division or modulo operations. This limitation quickly became a bottleneck in proofs about arithmetic algorithms in Fiat Cryptography and inspired Jason Gross to implement a Coq proof script for reducing arithmetic goals involving division and modulo to goals involving multiplication and addition: For positive denominators, it is sound and complete to replace the hypotheses `n mod d = r` and `n/d = q` with `n = q*d + r` and `0 <= r < d`. The same technique generalizes to expressions, but care must be taken to avoid duplication of the arguments to division or modulo. The implementation of this transformation was later included in Coq[3]. However, automatic arithmetic-goal solving does not use it by default because some proofs that previously passed while treating division and modulo as uninterpreted functions appear computationally infeasible with the additional equations included.

Abstract Interpretation When setting up the first program-proof automation for Bedrock2, I considered but did not attempt a solver-based approach because the experience just described and additional preliminary experiments suggested that the linear-arithmetic solvers available in Coq would be too slow to be used automatically. Instead, Clark Wood and I tackled the arithmetic goals in the first examples using tedious ad-hoc rewriting-based proofs. To help with this effort, I built a data-flow-directed abstract-interpretation pass to infer constant upper and lower bounds on word expressions in the proof context. Even though the implementation is written in Ltac, the analysis makes efficient use of the proof context, handles shared subexpressions without inlining them (which would potentially increase in expression size exponentially), and achieves overall-unproblematic performance. However, the deductive power of this method is inherently limited. As is typical of Bedrock2 functions, the memory accesses in `lan9250_tx` require comparing the accessed address to the beginning of the packet, and the two are not separated by any constant.

[3] `https://github.com/coq/coq/pull/8062`

Improvements to Coq's Linear-Arithmetic Solver While not reliable enough for automatic use, Coq's linear-arithmetic tactic did still allow for quick discharging of a number of Bedrock2 proof obligations. Samuel Gruetter and I used it sporadically and reported a number of bugs and performance limitations to the Coq issue tracker. The persistent interest from Fiat Crypto and Bedrock2 developers inspired Coq developers to improve the proof support for arithmetic goals. In particular, Frédéric Besson who was working on a SMT solver for Coq [Bes21] stepped up to implement a number of substantial optimizations to linear-arithmetic solving. To name a few: the old implementation of Fourier elimination was replaced with Simplex and then further augmented with Fourier-elimination-based heuristics to find simple cutting planes early, case-splitting proofs were entirely replaced with cutting-plane proofs, and the infrastructure for translating linear arithmetic goals was refactored and generalized.

One can hope that some of these optimizations would have been implemented regardless of our involvement, but having Bedrock2's arithmetic-proof needs serve as a test suite for work-in-progress improvements appears to have worked out particularly well. The new linear-arithmetic solvers are still not fast enough to handle all instances of division and modulo in the Coq standard library, but they run plausibly tolerably fast on many examples in Bedrock2. A qualitative measure of the success of these algorithmic improvements is that performance issues relating to their use are now sometimes tracked down to other root causes. The on-by-default transformation of the propositional connectives around linear-arithmetic facts into conjunctive normal form is one example[4].

The `ZnWords` tactic shown in earlier examples, implemented by Samuel Gruetter, uses top-down rewriting to translate word-arithmetic goals into equivalent integer-arithmetic goals. An advantage of this strategy is that it captures the full behavior of word arithmetic, with or without overflow, in a single goal that can be handed off to a solver for propositional formulas over linear arithmetic. However, the integer-arithmetic goals generated in this manner are often more complicated than one might intuitively expect because intermediate operations that "obviously" do not overflow are given the full Euclidean-equation translation. This makes interpretation of verification failures or timeouts challenging, sometimes calling for a trial-and-error approach similar to how one might troubleshoot a failed SMT-solver query. The current implementation is also slow, often spending even more time on searching for and rewriting word expressions than on linear-arithmetic solving, and this makes iteration more difficult.

[4] `https://github.com/coq/coq/issues/12140`

3.4.2 Top-Down and Bottom-Up Translation Strategies

Naive manual rewriting-based proofs and integer-arithmetic goals generated by a complete-translation tactic have a commonality that explains their power and tediousness. Consider the expression `unsigned ((a ^+ b) ^+ c)` where `^+` stands for addition of 32-bit words. In the common case that this expression does not overflow, the straightforward approach to transforming this word expression into an integer expression would start by rewriting with a lemma of the following form:

`unsigned_add_nowrap x y: 0 <= x + y < 2`^{32} `-> unsigned (x ^+ y) = x + y`

Actually applying this rewrite rule to the given expression results in a duplication of work. Instantiating `x:=a ^+ b` gives the precondition `0 <= unsigned (a ^+ b) + c` and a new expression `(a ^+ b) + c`. These two expressions appear as separate Coq goals, with no supported way for sharing work between them. After rewriting the addition, its arguments must be translated to integer expressions twice: once to prove the precondition of the rewrite rule and once in the goal where the original expression appeared. Coq's `repeat rewrite` happily generates an exponential number of goals.

A translation that instead generates `x + y mod` 2^{32} does not exhibit the same issue syntactically during translation, avoiding exponential blowup in many cases. However, the solver still has to reason about both cases of whether `x + y` overflows or not. In the case of Coq's linear-arithmetic-proving algorithm, cutting planes are used to separate noninteger solutions from the feasible region or the rational relaxation of the problem, and choosing a small number of effective cutting planes can take exponential time (the problem is NP-complete, after all). Cases where the upper bound on the sum is a constant are handled using fast preprocessing, but in these cases the rewriting-based approach could have solved the side condition using abstract interpretation, without calling the linear-arithmetic solver at all.

Bottom-Up Translation Whether a word-arithmetic expression overflows can be determined based on its inputs alone, without considering the context in which the expression appears. Further, explicit overflow checking is relatively rare in low-level programming, so most expressions that do not overflow can be proven not to overflow based on the inputs alone, without considering outcomes of conditional tests. This observation suggests that most word expressions encountered during program verification can be translated to modular-arithmetic-free integer-arithmetic expressions eagerly or not at all.

I implemented a data-flow-directed tactic that proves assertions of the form `unsigned (f a b) = f' (unsigned a) (unsigned b)` based on similar assertions about `a` and `b`. When invoked on an expression, the tactic first recursively translates the subexpressions `a` and `b` and only then considers the word-to-integer translation rules for `f`. The proof obligation asserting that the result of `f` does not overflow a word is stated

in terms of the translations of `a` and `b` and is proven by a call to Coq's linear-integer-arithmetic solver. As a bonus, the simple recursion structure makes it easy to handle subexpressions shared using definitions like `d := a ^+ b` in the Coq context without duplicating them.

Preliminary performance experiments suggest that this method is fast enough to be used more widely than `ZnWords`. However, it is also intentionally less complete: expressions that do overflow are not translated to integer arithmetic at all. Hypothetically, a later execution of the bottom-up translation after additional hypotheses have been added to the context has the opportunity to complete this translation. Alternatively, `ZnWords` can be called for a complete translation to a hard-to-read integer-arithmetic goal to see whether it would be proven by the integer-arithmetic solver. A nontrivial practical downside of the particular incompleteness of this approach is that every change to the proof context could, in principle, make more information available to the integer-arithmetic solver and thus allow for additional progress to be made in the word-to-integer-arithmetic translation.

3.4.3 Bitwise Operations

The approaches based on translating word arithmetic to modular arithmetic and then integer arithmetic only apply to arithmetic operations on words. Bitwise operations such as logical and, or, and xor are satisfactorily handled by the perspective that machine words are sequences of bits. Concretely, the Coq function `Z.testbit : Z -> Z -> bool` and the lemma that two integers are equal if all of their bits are equal allow a word-equality goal to be traded for a bit-equality goal for a bit at a universally quantified index. Rewrite rules for individual operations are then used to translate bitwise operations to operations on individual bits and their indices, expressing the ith bit at the output in terms of bits of the input. For example, truncation is translated to a Boolean conjunction with an index comparison.

```
Z.testbit_mod_pow2: ∀ a n i, 0 <= n ->
  Z.testbit (a mod 2^n) i = ((i <? n) && Z.testbit a i)
```

A naive algorithm that exhaustively explores combinations of outcomes of integer comparisons and checks their consistency using an integer-arithmetic solver has been sufficient to prove the small bitwise-arithmetic goals that appear in Bedrock2 programs. However, note that the strategy presented here is distinct from bitblasting, which involves creating a Boolean variable for every bit of the machine word. A qualitative consequence of this difference is that the technique described here can be applied to words with a universally quantified number of bits, whereas bitblasting requires a concrete size.

The few goals that involve both bitwise and arithmetic operations (and where the bitwise operations do not fall into simple arithmetic patterns like implementing modulo

using a mask) have been solved by asserting the desired arithmetic property about the bitwise computations and proving each subgoal using a different method.

3.4.4 Evaluating Constant Expressions

Bedrock2 programs sometimes define values of compile-time constants using arbitrary Coq expressions. A minimal example is the usage of `$(2^13)` on the first line of `lan9250_tx` above: the `$` escapes the grammar of Bedrock2 to insert a Coq expression where the operator `^` stands for arbitrary-precision exponentiation. The desired proof strategy for programs involving complicated constant expressions is to keep these expressions in their original syntactic form for readability exactly as long as this does not impede the completeness of automatic proof procedures. Constant expressions can also appear in specifications of Bedrock2 programs and sometimes have non-integer types: for example, constant headers of network packets can be specified as lists of bytes. This seemingly trivial request is surprisingly challenging to satisfy generically in Coq, making for a spectacular case study of how practical usability of a proof assistant can get tangled up in its type-theoretic foundations.

First, there does not seem to be any reliable way to determine whether the evaluation of an arbitrary expression in the proof context would yield a value or result in a stuck term that contains a reference to some universally quantified variable or even an opaque proof. This practical limitation is especially surprising in the context of the metatheoretical strong-normalization guarantee of the Coq type system: every expression that is well-typed in the global context can be evaluated to a value. It is easy to check whether an expression is well-typed in the proof context, and it is possible (but slow) to clear the proof context and check whether the expression is well-typed in the section context, but there is no tactic-programming interface to check an expression against the global context.

Indeed, the section mechanism of Coq works very similarly to temporarily extending the global context with the section variables and assumptions as axioms. Interactive use of Coq has access to the `Print Assumptions` command that can be used to determine whether an expression (usually a proof) depends on any axioms, but it works by walking all references in its argument and takes several seconds on simple examples. Further, as `Print Assumptions` is intended for checking proofs, it does not complain about proofs appearing in its argument, and expressions that reference opaque proofs can get stuck during evaluation using every built-in evaluation mechanism of Coq even though they are considered computable from the perspective of the metatheory. Evaluating by trial and error is not an option: inlining functions operating on a stuck term can easily consume arbitrarily much memory.

Even if opaqueness was bypassed, trying to compute with proofs would be nonsensical from the perspective of a practical user and arbitrarily slow in any case. However, it is also not satisfactory to rule out all occurrences of proofs inside expressions.

As Coq's core logic must rule out nonterminating programs to remain consistent, nonstructurally recursive definitions must be justified using proofs of termination, and these proofs are considered a part of the definitions they justify. The machinery for defining nonstructurally recursive functions relies on an inductive definition of well-founded relations and defines well-founded recursion as recursion on the *proof* of well-founded accessibility of the function argument. The type-theoretic construct that allows for this is called subsingleton elimination.

Notably, the fast evaluation mechanisms (`cbv`, `vm_compute` and `native_compute`) in Coq do not treat proofs specially: whether or not used to justify termination, if an opaque proof appears in an expression, it remains as-is, potentially blocking full evaluation. This restriction is necessary exactly because these evaluation mechanisms can be used in arbitrary contexts, potentially accepting and returning expressions which contain universally quantified variables. However, the `Extraction` machinery for executing Coq code outside Coq strips proofs from all expressions, allowing computation to proceed even in the case of well-founded recursion justified by an opaque proof. This is possible because `Extraction` only operates in a global context, failing with "You can't do that" within a section. Further, while `Extraction` can translate almost all Coq constructs, the target languages do not support partial evaluation, so `Extraction` can be used only to compute expressions of first-order types such as integers and lists of integers but not functions and types. This restriction also ensures that the resulting value to will not need to reference the proofs that are erased.

Thus, seeking to evaluate an expression that is well-typed in the global context during a proof inside a section remains on our long wishlist of practical Coq improvements. An Ltac function that recursively inspects the expression and only allows an explicit list of integer-arithmetic operators and related functions before calling `vm_compute` is used as a stand-in. Note that both a global-context well-typedness check and a proof-erasing evaluator would be required to dispense with this ad-hoc hack within the current design of Coq.

3.4.5 Symbolic Execution With Address Arithmetic

Calling functions with computed addresses means that the memory footprint these functions operate on depends on arithmetic expressions, posing an obstacle to symbolic execution based on cancellation alone. In fact, the same consideration applies to stores and loads (including the one in `lan9250_tx`), but it is better to look at a richer example to understand the pattern. Consider a program that has access to a 10-byte array at address `b` and calls `memmove(b, b+1, 9)` to remove the first byte. The specification of `memmove` in Bedrock2 reads as follows:

```
fnspec! "memmove" (dst src n : word) / (d s : list byte) R,
{ requires t m := m =*> bytes src s * ∧ m =* bytes dst d * R ∧
    length s = n :> Z ∧ length d = n :> Z ∧ n <= 2^(width-1);
```

```
  ensures t' m := m =* bytes dst s * R ∧ t=t' }.
```

Naive cancellation would fail to solve the first precondition because the caller's symbolic state has `bytes b bs`, but the callee's precondition with `src := b^+1` requires `bytes (b+1) ?s`. Worse, the second precondition would be solved in a manner that needlessly leads to an unprovable goal. With `dst := b`, `bytes b bs` would be cancelled with `bytes b ?d` instantiating `?d := bs`. The problem is that the fourth precondition requires `length d = 9`, but `bs` has length 10. Indeed, only the first 9 elements of `bs` are to be overwritten by `memmove`, and the last byte would remain as-is, which can be proven by instantiating the specification with the frame `R := ptsto (b+9) (nth 9 bs)`.

The appropriate symbolic-execution rule in the presence of pointer arithmetic is to inspect the length preconditions first. Then, instead of looking for a syntactic match for `bytes (b+1) ?s`, the symbolic state is to be searched for a byte array spanning `b+1...b+1+9`. Checking this requires word-arithmetic queries about two inequalities, calculating an index and comparing it to the length: `(b+1)-b < 10` and `(b+1+9)-b <= 10`. Based on this, the symbolic state `bytes b bs` can be rewritten to `bytes b (firstn 1 bs) * bytes (b+1) (skipn 1 bs)` and have `bytes (b+1) ?s` cancelled from it, instantiating `?s := skipn 1 bs`. The same process is then repeated to determine `d`, starting from the symbolic state before the computation of `s` for simplicity.

3.5 Precondition-driven Quantifier Instantiation

Actually carrying out the above approach on top of the Coq proof engine is surprisingly tricky, even just for reasons orthogonal to the word-arithmetic-proving challenges discussed in the previous subsection. Two major limitations are at play: First, there is no intentional API for inspecting one of several goals to use in the proof of another. Further, even if this limitation is bypassed using evars or some other mutable-state gadget, the goals' contexts are completely independent as far as Coq is concerned, even if the goals correspond to the two sides of a ∧.

To understand how limiting this paradigm is, it is helpful to walk through the proof-engine operations that correspond to a straightforward attempt at implementing the symbolic-execution strategy. First, the specification of `memmove` is looked up and the quantifiers for runtime arguments `dst`, `src`, and `n` are filled in. The quantifiers for the specification arguments are instantiated with new existential variables `?d` and `?s`. Splitting the requirements stated using ∧ results in five subgoals for the preconditions, along with another goal for the execution of the code after the call to `memmove`. So far, so good.

For a clear-cut example of how these goals being unrelated is restrictive, consider trying to use a lemma `∀ bs, ∃ bs0 bs', bs = bs0 ++ bs'` to justify the cancellation

of a part of `bs` to cancel `bytes (b+1) ?s`. This lemma can be instantiated for `bs`, and the existential quantifier can be eliminated to yield a variable `bs'` in the proof context of the goal in which the lemma was instantiate, but cancellation will fail because `?s` cannot be instantiated with `bs'`. The reason is that the existential variable `?s` was created in a context that does not include `bs'`, so it cannot be instantiated with it.

A simple justification for this limitation is that `∃ n, ∀ m, n = m` must not be provable by creating an existential variable for `n` and instantiating it with `m`. As the requirements of `memmove` are to be proven in the same context, soundness is not at stake here. The same lemma can be used to instantiate `?s` if the lemma is used *before* instantiating the `memmove` specification's quantifiers with new existential variables. However, lifting the evar-context limitation would amount to treating the goals between which evar instantiations can be shared as appearing in the same context, not independent contexts as assumed by the Coq proof engine. Specifically, instantiating `?s` during the proof of the precondition would need to make the variable `bs'` appear in the context for symbolic execution of the code after the call to `memmove`.

If `bs'` is not the witness of an existentially quantified statement but rather a local definition `bs' := skipn 1 bs`, trying to unify `?s` with `bs'` instantiates `?s` with `skipn 1 bs` instead. This behavior is workable in principle, but it results in an unnecessarily verbose symbolic state for the verification of the code after the call to `memmove`. As an additional complication, properties proven about `bs'` during the proof of the precondition (for example, that it has length 9) cannot be carried over to the proof of the postcondition, so they will need to be rediscovered and reproved.

3.5.1 Artificial Example: Loading an Existentially Quantified Value

While calling `memmove` is a practically relevant and convenient example, the core challenge discussed in this subsection is not unique to address arithmetic: In fact, a less arbitrary-seeming form of the same problem would appear when passing a singly linked list to a function that simply dereferences a pointer. Specifically, it is conventional to specify a cell of nonempty singly linked list at address `p` and with value `v` in separation logic using an existentially quantified pointer to the next cell: `∃ pnext, scalar p pnext * scalar (p+4) v`. Independently, the simplest specification of a pointer-dereferencing function would be

```
fnspec! "deref" p / v ~> r,
{ requires t m := m =*> scalar p v;
  ensures T M  := T = t ∧ M = m ∧ r = v }.
```

Straightforward and modular symbolic execution of `deref(p)` is bound to get stuck.

Instantiating the quantifier for `v` with a new existential variable to reveal the precondition `m =*> scalar p v` fixes the context of that evar. After this, a simple proof script can easily inspect the precondition and understand that the existential quantifier inside the representation predicate for the singly linked list needs to be moved to the proof context. Cancellation would be able to prove `m =*> scalar p ?v` if the previous two steps were swapped, but it fails to instantiate `?v := pnext` because `pnext` was introduced after `?v`.

Again, `?v` appears in the postcondition of `deref`, so there is no hope of making progress with a proof of `scalar p pnext` in the context of the precondition alone. Further, consider what the specification of the function in which the call to `deref` resides would be. Positing the predicate for the singly linked list as the precondition and an implementation that simply returns the value received from `deref`, there does not appear to be a way to state a nontrivial postcondition:

```
fnspec! "callderef" p ~> r,
{ requires m := m =*> sll p ∧ p <> 0;
  ensures    := r = (* ?? What goes here? *) }.
```

Revealing the definition of the `fnspec!` notation, the problem can be simplified to `forall p (H : sll p ∧ p <> 0), call "callderef" (fun t m r => (*??*))`. A reader familiar with the type theory of Coq can observe that using an existentially quantified value from the definition of `H` to fill in `??` would constitute extracting information from the proof of a `Prop` to compute a word to use in an argument to `call`, a pattern that in general is inconsistent with classical logic. This connection yields insight that can be used to state the general form and root cause of the overall quantifier-instantiation challenge. The Bedrock2 program-proof design can be seen as centering around translating operations in imperative code to functional expressions in the proof context. For constructing a function-call node, its arguments must be translated first, and specification arguments are required in all cases where they influence the (specified) output of the resulting functional program.

3.5.2 Magic Wand as a Solution for Frame-Quantifier Instantiation

For the special case of the specification argument `R` that describes the rest of the memory to be preserved throughout the function call, the context-management challenge investigated in this subsection can be bypassed using an additional separation-logic connective, the magic wand (`-*`). Informally, `P -* Q` represents the memory fragment that would satisfy `Q` if only it were joined with a memory fragment that satisfies `P`; in symbols `m =* P -* Q` iff `∀ dm M, P dm -> M = dm ∪ m -> Q M`. (Here and onwards, equalities involving the disjoint-union operator ∪ are implicitly conjuncted with the fact that the parts are in fact disjoint). From the perspective of proof ma-

chinery, `P -* Q` can be thought of as the predicate that asserts that `P` can be cancelled from `Q`.

If `P` is the postcondition of the callee (instantiated with existential variables for other specification arguments as required) and `Q` is the predicate that the code after the function call executes as desired, it is sound and complete to instantiate the frame quantifier in the callee's specification with `P -* Q`. Doing so results in `P -* Q` appearing as one of the disjoint conjuncts in the precondition of callee. The other disjoint conjuncts are cancelled from the caller's symbolic state as usual, leaving behind only the ones not used by the function call, `R'`. At this point the goal is `R' ==> P -* Q`, which unfolds to `∀ m, R' m -> (P -* Q) m`, which unfolds to `∀ m, R' m -> ∀ dm M, P dm -> M = dm ∪ m -> Q M`, which is equivalent to `∀ M, (R' * P) M -> Q M`. Introducing `M` as the new memory and `R' * P` as the new symbolic state allows symbolic execution to continue to the post-function-call code described in `Q`.

This indirection means that the specification argument `R` does not have to be instantiated with `R'`, so `R'` can reference context variables introduced after the specification arguments were instantiated with existential variables. Instantiating `R` with `P -* Q` is unproblematic because `Q` is already present in the goal before symbolic execution of the function call and `P` can be extracted right after all other specification arguments are instantiated. Conveniently, the same instantiation is possible throughout cancellation, and specifically at the very end of it, so that goals displayed to the user do not need to include `-*` at all. However, using this trick for instantiating the frame `R` does not solve the context-management problem for other arguments of the callee's specification.

Samuel Gruetter and I arrived at this fine-grained characterization of the benefit provided by this use of the magic wand while investigating the issues discussed in this section; my notes tell me to credit him with the experimentation and initial insight. The trick for instantiating `R` with `P -* Q` is called the ramified frame rule, and a more optimistic description of its usefulness as a solution to evar-instantiation problems appears in [Cha20b]. The same concept is in turn credited to [KBA10], where a (different) ramified frame rule was used to modify the frame predicate itself.

3.5.3 Emulating Multiple Goals in the Same Context

It is possible to implement some limited tactic-programming functionality on top of the Coq proof engine without splitting the conjuncts of the callee's precondition into independent goals. For example, solving any one of the conjuncts using a tactic can be supported generically using a wrapper `solve_first tac`. Rewriting in the goal similarly works as expected because the other conjuncts are treated just like any other context in which a term is replaced with an equal one. However, any tactic that modifies the context needs to execute in the proof-engine goal for the entire conjunct.

A particularly inconvenient aspect of this strategy is that all lemmas that are applied to the goal need to be translated to a form where their hypotheses are joined using ∧ rather than `->`. For example:

```
Lemma uncurried_load_four_bytes_of_sep_at bs a R (m : mem)
  (H : m =*> bytes a bs ∧ length bs = 4) :
  load access_size.four m a = Some (word.of_Z (le_combine bs)).
```

Further, definitions that are unfolded to generate goals must take care not to introduce existential variables earlier than necessary. Calling the expression-evaluation predicate `dexpr m l e v` introduced with a fresh existential quantifier for `v` in `cmd_ok` of Section 3.2 is a great example of how violating this principle can lead to trouble. If the expression `e` includes `load` operations, the proofs that these loads succeed (by cancellation and arithmetic) must instantiate the existential variables for the result with expressions that are well-typed in the context of `v`, which does not contain any definitions or hypotheses introduced by the expression-evaluation proof. Specifying expression evaluation using a continuation-passing-style definition `expr_ok` similar to `cmd_ok` sidesteps this limitation:

```
Definition expr_ok (e : expr) (post : word -> Prop) : Prop :=
match e with
| expr.op op e1 e2 =>
  expr_ok e1 (fun v1 =>
    expr_ok e2 (fun v2 =>
      post (interp_binop op v1 v2)))
| expr.load s e => expr_ok e (fun a =>
    ∃ v, load s m a = Some v ∧ post v)
...
```

Relying on the fact that Bedrock2 expression evaluation is deterministic, it is possible to prove that the weakest-precondition generator for expressions matches `dexpr` exactly:

```
Lemma dexpr_expr_k e P :
  expr_ok m l e P -> ∃ v, dexpr m l e v ∧ P v.
```

Chapter 4

Omnisemantics

This chapter describes omnisemantics, specifically inductive omni-big-step and omni-small-step judgments, and their use in the integration proof of Bedrock2, its compiler to RISC-V, and a pipelined RISC-V processor. The presentation is intentionally more general when applicable: I am optimistic that this technique will be directly applicable and beneficial in other projects modeling nondeterminism and undefined behavior, perhaps even outside integration proofs and verified systems programming.

An omnisemantics judgement gives an operational description of the relation between starting states and *sets of outcomes* that may result from potentially nondeterministic execution or overapproximations thereof. To contrast, traditional operational semantics relate a starting state to every possible outcome using a separate derivation of the semantics judgement. Encoding sets and relations as predicates in Coq, the type signature of an omnisemantics definition becomes

```
state -> program -> (outcome -> Prop) -> Prop
```

whereas traditional big-step operational semantics have type

```
state -> program -> outcome -> Prop
```

In the context of axiomatic semantics, omnisemantics can be seen as predicate transformers defined (inductively, coinductively, or otherwise) by operational rules. The weakest-precondition definition presented in Section 3.2 also has the same type signature and interpretation but differs in using a nonoperational rule for loops (Subsection 3.2.7) as it is defined by structural recursion over program syntax.

Omnisemantics rules can directly assign each language construct undefined behavior (*no* outcome set), unreachable behavior (empty outcome set), deterministic behavior (one outcome), and unspecified behavior and nondeterminism (multiple outcomes),

all potentially defined in terms of behaviors of other constructs. One derivation of omnisemantics is enough to ascertain well-defined execution and to cover all possible (nondeterministic) executions from the given starting state; the name *omni*semantics was chosen to highlight this feature. Achieving this directly using straightforward operational rules in standard inductive or coinductive definitions allows for smooth reasoning about nondeterministic programs: induction over an omnisemantics derivation follows the control flow *and nondeterministic choices* throughout the execution of the program. With traditional small-step and big-step semantics where each derivation covers a single execution, just defining the notion that no execution goes wrong and all executions reach a desired outcome requires both an additional layer of semantics definitions and administrative changes to the core inductive judgement itself.

Omnisemantics and Compiler-Correctness Proofs The key motivation for developing omnisemantics for use in Bedrock2 was the need to support the extension of the Bedrock2 compiler (and its correctness proof) to programs that accept input during execution. As a component of an integrated system-correctness proof, the central requirement for the compiler-correctness proof is that the I/O behavior of the compiler-generated code must satisfy any specification proven against the source-language semantics. Omnisemantics allows an appropriately strengthened variant of this property (which accounts for memory and local variables) to be stated as an implication between source- and target-language omnisemantics judgements and proven directly by induction on the source-language omnisemantics judgement. Furthermore, while the inductive statement of compiler correctness includes a representation relation between the source-language state and target-language state, this relation only needs to be established when invoking the inductive hypothesis after each *source*-language step. The proof of a compiler that implements a source-language command using a sequence of several target-language commands can proceed by symbolic execution of the generated commands and allow the representation relation to be temporarily violated between intermediate target-language steps.

All this is in-line with intuitive expectations about compiler correctness but famously out-of-reach for compiler verification using traditional operational semantics for a nondeterministic target language. Naively using a direct adaptation of the compiler-correctness statement from the previous paragraph (forward simulation) is insufficient: the existence of a correct target-language execution does not rule out the possibility of other executions nondeterministically going wrong. Instead, the approach taken by CompCert [Ler09] and related work has been to model the semantics as deterministic so that a similar proof strategy can be applied soundly. However, not all features of C can be modeled deterministically (see Subsection 2.4.1), and even supporting the subset specified by CompCert requires an intricate abstract memory model [LB08] where memory blocks are assigned sequential identifiers. This technical construction must then be handled in the proofs, and the complexity of the required simulation relations (and of the proofs themselves) is a bottleneck for ex-

tending CompCert. Use of omnisemantics in Bedrock2 and RISC-V specifications and throughout the associated compiler-correctness proof allows for a simple (but nondeterministic) memory model, and the representation relation for Bedrock2 memory in a RISC-V machine is just disjoint union!

Historical Note An inductive definition in the style that my collaborators and I now call omnisemantics first appeared in [SSS16] where it was described as an axiomatic semantics that maps programs to predicate transformers. The paper demonstrates the power that big-step omnisemantics provide applied to the *source* language of a compiler. A case study shows the verification of a compiler pass that translates nondeterministic (underspecified) control flow expressed using guarded commands (unordered switch statements) to a deterministic language with normal conditionals and lexically scoped gotos, both formalized using omnisemantics. The authors recognize that their proofs are "straightforward," describe the theorem statement as "elegant," and note that proving the same using traditional semantics "seems complex and tedious." And then they move on to study the relationship between omnisemantics and earlier models of nondeterminism – fixed-point combinators and ω-iteration.

The big payoff of omnisemantics appears to have been missed: a nondeterminism-compatible compiler-proof methodology opens the door for direct modeling of underspecified intermediate *target* languages, removing the need for technical determinism! Even so, one might expect that resolving the challenge of source-language underspecification would be widely recognized, but no: As of this writing, the ACM Digital Library counts 0 citations of [SSS16], and traditional operational semantics remain the norm outside the Bedrock2 ecosystem. Despite substantial literature-searching by myself and collaborators, I only found the discussed work accidentally, after the integration-verification case study described in this chapter was completed.

I developed the definitions of big-step and small-step omnisemantics and the associated compiler-correctness definitions based on traditional operational semantics, the weakest-precondition definition (Section 3.2), and the safe-execution judgements used for Cito and Facade [WCC14] in the original Bedrock ecosystem. The definitions I formulated (in October 2018[1]) were then used to replace the deterministic stand-ins used in initial prototyping of the Bedrock2 compiler. Samuel Gruetter handled the vast majority of compiler implementation and proofs associated with translation of individual language constructs and optimizations before and after the switch to omnisemantics. When connecting the RISC-V semantics to the traditional operational semantics the Kami [Cho+17] processor was proven against, I designed an Omnisemantics-based variant of the RISC-V specification and the key correctness statement for modular integration verification. Then Joonwon Choi, Samuel Gruetter, and I worked together to reconcile the RISC-V environments assumed in the two projects and prove

[1] `https://github.com/mit-plv/bedrock2/issues/27#issuecomment-433987172`

a bit-precise equivalence between the two models of each instruction.

Arthur Charguéraud independently developed big-step omnisemantics in the context of functional languages, related them to traditional big-step and small-step semantics, and used them to prove type-soundness results [Cha20b]. Arthur Charguéraud, Samuel Gruetter, and I then joined our efforts and wrote a comprehensive description of omnisemantics and their uses [Cha+23], including some new results about partial-correctness modeling and compiling deterministic languages to nondeterministic languages.

4.1 Big-Step Omnisemantics for Bedrock2

The definition of weakest preconditions used for proving properties of Bedrock2 programs (Section 3.2) adequately specifies the validity and behavior of programs, but its structure does not follow execution in some cases. Two structural differences stand out in comparison to the interpreter used as the specification of the previous compiler prototype that supported a deterministic subset of Bedrock2. First, the weakest precondition for a loop asserts the existence of an invariant and a decreasing measure. Second, the environment of the program is described as an ordered list of functions, and calling a function further down the list removes that function and all preceding functions from the environment for the duration of the execution of the callee. These encoding choices are required to define weakest preconditions by recursion on the program structure, but they do not match the execution strategy of any plausible implementation of Bedrock2.

The encoding of weakest precondition also has a number of properties that an alternative semantics should preserve. Most importantly, writing the definition in continuation-passing style allows nondeterminism due to input and underspecified stack allocation to be encoded using a universal quantifier. A desirable consequence of this choice is that undefined behavior is properly accounted for: having to prove that the rest of the program satisfies the specification for every possible nondeterministic choice rules out undefined behavior across all possibilities. Ruling out recursive function calls by construction and using loop measures also ensures termination. (Or at least such is the intent – a similar definition but with the termination measure omitted from the loop case would have worked just as well for everything discussed so far.) It is also convenient that the weakest-precondition definition does not rely on any custom auxiliary state for keeping track of the progress of execution: control flow is encoded through choice of the continuation.

4.1.1 Defining Omnisemantics Inductively

These additional properties can be enforced elegantly through the structure of the inductive definition of omnisemantics. Some care is required to arrive at a well-formed inductive definition that Coq can soundly accept.

First, it makes sense to confirm that the type signature is permissible: while recursive definitions can return values of arbitrary types, the inductive closure of inference rules forms a new type or predicate. The continuation-passing type signature of the semantics (`state -> program -> (outcome -> Prop) -> Prop`) is acceptable as-is because the return type (of the continuation and the definition itself) is hardcoded to `Prop`. While a continuation-passing-style interpreter could be implemented in a way that supports arbitrary continuation return types, the weakest-precondition definition already relies on returning `Prop` as it models nondeterminism using universal quantification.

To understand the considerations for encoding the rules for nondeterminism and sequencing, it is useful to review how inductive definitions generalize tree types by allowing universally quantified inductive premises. For example, Coq accepts the following inductive definition with a base-case rule, a rule with a single inductive subtree, and an infinitary rule with one subtree for each value of type `A`:

```
Inductive allEven : ∀ {T : Type}, T -> Prop :=
   (* Variables:  *)        (* Premises: *)          (* Conclusion: *)
| eZ (n : nat)          (_ : n = 0)                     : allEven n
| e2 (n m : nat)        (_ : allEven m) (_ : n = 2+m)   : allEven n
| eF {A B} (f : A -> B) (_ : ∀ (a : A), allEven (f a)) : allEven f.
```

Occurrences of the predicate being defined are limited to the conclusions of premises. Specifically, `allEven (f a)` is allowed even though it appears under a universal quantifier, but `∀ (a : A), allEven a -> allEven (f a)` is not allowed because the first occurrence of `allEven` is not the conclusion of the premise. Arbitrary self-referential usage of the inductive predicate *cannot* be allowed in the rules that define it, lest we admit contradictory definitions such as "the proposition that is not true":

```
#[bypass_check(positivity)]
Inductive P {T} :=
| C                     (_ : P -> T)                    : P.
```

To see that the definition `P` is problematic, consider the instantiation `T := False`. The premise of rule `C` is then `P -> False` which is more commonly written `not P`. As `C` is the only rule, we have `P ↔ not P`, which is a contradiction regardless of how `P` is defined. Thus, some restrictions on the rules used in inductive definitions are required for logical consistency. The requirement that the predicate being defined must only appear in the conclusions of premises is conservative, but it is not redundant.

Stack Allocation Omnisemantics use universally quantified inductive premises to encode nondeterminism. The rule for the Bedrock2 command `stackalloc n as x in c` quantifiers over the `address` and `uninit`ialized contents of the memory that will be made available to the command `c`.

```
Inductive exec : cmd -> trace -> mem -> locals ->
                        (trace -> mem -> locals -> Prop) -> Prop :=
| stackalloc (x : string) (n : nat) (c : cmd) t m l post
  (_ : ∀ (addr : word) (uninit : list byte) M,
    length uninit = n ∧ M = m ∪ bytes addr uninit ->
      exec c t M (map.put l x addr) (fun t' M' l' =>
        ∃ m' dealloc, length dealloc = n ∧
        M' = m' ∪ bytes addr dealloc ∧
        post t' m' l'))
  : exec (cmd.stackalloc x n c) t m l post
...
```

Appropriate hypotheses describing the new memory `M` are provided so that the inductive application of `exec` to `c` can be proven even though the specific address and stack memory contents are unknown. The postcondition against which the inductive `exec` invocation needs to be proven is not just the postcondition of the scoped stack allocation itself; rather, the outer postcondition needs to hold after the stack-allocated memory has been successfully deallocated. And, once more, note that the rule for stack allocation does not involve any accounting of allocation identifiers. The only way the freshly allocated memory is distinguished in the state on which the inner command is executed is that `addr` is assigned to the local variable `x`.

Sequencing Following the example of the weakest-precondition generator, we would like to handle sequencing of commands by choosing the continuation of the first command to be that the second command executes correctly and satisfies the desired postcondition:

```
| disallowed_seq (c1 c2 : cmd) t m l post
  (_ : exec c1 t m l (fun t' m' l' => exec c2 t' m' l' post))
  : exec (cmd.seq c1 c2) t m l post
```

Trying to define the semantics inductively using this rule fails because the inner occurrence of `exec` is not strictly positive: it is an argument to `exec` itself (and further inside a lambda function), not the conclusion of a premise. This limitation of Coq is not spurious, at least not entirely. Specifically, it is not clear what doing induction over a derivation of `exec` defined using this rule would look like. Normally, an induction hypothesis is available with each inductive premise, but in this attempted definition the inner `exec` is applied to `t'`, `m'`, and `l'` which are bound by a lambda.

Intuitively, it seems desirable for the inductive hypothesis to be presented right next to the inner `exec` using $\wedge$, but this only makes sense because `exec` is intended to allow for weakening of the postcondition. However, as it is also possible to define a variant of `exec` that does not satisfy weakening [Cha+23, §2.3], a general mechanism for inductive definitions should not allow for this kind of wishful induction.

Instead, the desired meaning and a logically sufficient induction principle can be achieved by explicitly allowing for weakening of the postcondition of the outer inductive occurrence of `exec`:

```
| seq (c1 c2 : cmd) t m l post mid
  (_ : exec c1 t m l mid)
  (_ : forall t' m' l', mid t' m' l' -> exec c2 t' m' l' post)
  : exec (cmd.seq c1 c2) t m l post
```

The initial version of the sequencing rule is still valid and can be proven equivalent to the inductive-definition-compatible version after-the-fact. As might be expected based on the discussion of induction, the reverse direction of the equivalence relies on the postcondition-weakening lemma for `exec` (see Subsection 4.1.2):

```
Lemma seq_chained c1 c2 t m l post :
  exec c1 t m l (fun t' m' l' => exec c2 t' m' l' post)
  ↔ exec (cmd.seq c1 c2) t m l post.
Proof. split; try invert 1; eauto using seq, weaken. Qed.
```

Loops and Conditionals Having figured out how to express safe sequential execution of potentially nondeterministic commands using an inductive definition, applying the same pattern to loops straightforwardly leads to a satisfying formalization. Unlike a structurally recursive weakest-precondition definition, omnisemantics can specify the semantics of `while e c` by inlining the rules for `if (e) { c; while e c }`, including an inductive invocation of `exec` on the same loop in case the loop continues.

```
| while_false e c t m l post v
    (_ : eval_expr m l e = Some v) (_ : v = 0)
    (_ : post t m l)
  : exec (cmd.while e c) t m l post
| while_true e c t m l post v
    (_ : eval_expr m l e = Some v) (_ : v <> 0)
    mid (_ : exec c t m l mid)
    (_ : forall t' m' l', mid t' m' l' ->
         exec (cmd.while e c) t' m' l' post)
  : exec (cmd.while e c) t m l post
```

Separating the two cases corresponding to conditional behavior of the loop test into individual inductive rules is conventional in operational semantics, and it is also an expedient choice in Coq for omnisemantics in particular. A premise of the form `if v then exec … else …` would not be accepted as `exec` appears in it and is not directly the conclusion of the entire premise. While Coq does permit an extended form of inductive definitions where inductive premises can appear as parameters to *other* inductive definitions including logical connectives, its use is not well-supported by surrounding infrastructure. An inductive definition with a premise of the form `v <> 0` $\wedge$ `exec …` $\vee$ `…` would be accepted, but induction over it using the automatically generated induction principle is unnecessarily weak: the induction hypothesis for the nested inductive occurrence of `exec` is missing. It is possible to define induction principles manually to compensate for this issue, and a number of metaprogramming frameworks for Coq include examples that can generate satisfactory induction principles for a subset of the inductive definitions in Bedrock2, but staying within the well-supported fragment of basic inductive definitions appears to be the pragmatic choice still.

Function Calls The semantics of a complete Bedrock2 program are defined based on the entry point and a fixed environment of functions, `e`. The `exec` case for calling a function just looks up the definition of the callee by name and invokes `exec` on its body:

```
| call binds fname arges
    (_ : map.get e fname = Some (params, rets, fbody))
    ...
    (_ : exec fbody t m lf mid)
    (_ : forall t' m' l', mid t' m' l' -> ... )
  : exec (cmd.call binds fname arges) t m l mc post
...
```

Omitted from this definition is context management: argument expressions are evaluated in the caller's context, parameters of the callee are initialized to the corresponding values (`lf`), and return variables are transferred the other way around. Mutable memory and the static function environment are shared.

4.1.2 Input and Output, Basic Properties

I/O operations have the same mechanical aspects as function calls, but instead of `exec` on the callee's body the behavior of I/O operations is described by a *parameter* of the samentics named `ext_spec`. Additionally, the `exec` case for external calls updates the trace `t` with the name of the external call, its arguments, and return values (and memory modifications, but this aspect is not used in the case studies with integrated proofs so I am eliding it here). As input and output in different use cases

of Bedrock2 is modeled differently, the requirements and guarantees of an external call are specified as a part of the parameter `ext_spec` as well. It is expected that the definition of `ext_spec` may assert its postcondition under a universal quantifier. The continuation-passing-style structure leaves considerable flexibility to implementations of `ext_spec`. For example, the specification of an external call to check on the status of an asynchronous operation may quantify over a domain-specific representation of the observed status or simply assert a disjunction of two different applications of the postcondition for the "done" and "not done" cases.

The encoding considerations for the semantics are simply a combination of those from internal nondeterminism (discussed in the context of stack allocation) and state, but treating `ext_spec` as a parameter requires some restrictions on its behavior. To make this concept more concrete, here is a simplified `ext_spec` that allows interaction with specific MMIO peripherals accessible through concrete ranges of the address space. Here the `MMIOREAD` case just calls the postcondition under a universal quantifier but without any assumptions, but the type signature of `ext_spec` does not prevent it from calling `post` multiple times or even under a negation!

```
Definition isMMIO (addr : word) :=
  addr mod 4 = 0 ∧ (
  0x00020000 <= addr < 0x00022000 ∨
  0x10008000 <= addr < 0x10010000 ∨
  ... ).
Definition ext_spec t action args post : Prop :=
  if action =? "MMIOWRITE" then
    ∃ addr val, args = [addr; val] ∧ isMMIO addr ∧
      post []
  else if action =? "MMIOREAD" then
    ∃ addr, args = [addr] ∧ isMMIO addr ∧
      ∀ val, post [val]
  else False.
```

One could seek to specify exhaustively the properties that an `ext_spec` must have to make sense a as a specification of external calls. However, as `ext_spec` is itself a part of the specification of the execution environment of the language and must be vetted as a part of the statement of program-correctness proof, verifying it might be besides the point. Further, all `ext_spec` instances in the current Bedrock2 ecosystem are small, so a sophisticated property of them may be harder to audit than the concrete specifications themselves.

Instead, the Bedrock2 semantics only require basic properties of `ext_spec` when the corresponding property holds of the overall semantics and is used in the Bedrock2 ecosystem. The weakening property of omnisemantics is one example:

```
Lemma weaken : ∀ c t l m P Q, (∀ t m l, P t m l -> Q t m l) ->
```

```
  exec c t m l P -> exec c t m l Q.
Proof. (* induction on exec *) Qed.
```

As the `exec` case for I/O just calls `ext_spec`, this weakening lemma can only be proven under a corresponding assumption:

```
(∀ ret, P ret -> Q ret) -> ext_spec t a args P -> ext_spec t a args Q
```

The postcondition-weakening property satisfyingly rules out `ext_spec` instances that assert the negation of the postcondition.

The correctness proof of the Bedrock2-to-RISC-V compiler additionally relies on a postcondition-intersection property about `exec`. While the statement of compiler correctness includes just one `exec` hypothesis, the compiler also uses static-analysis passes to study the program. The static-analysis passes are in turn specified in terms of the semantics of Bedrock2. For example, computing the list of variables used in a well-defined bedrock2 command can be used to conclude that all other variables are left unchanged when the analyzed command is executed. When reasoning about use of static analysis, it is desirable to combine the derived postconditions:

```
Lemma intersect : ∀ c t l m P Q,
  exec c t m l P -> exec c t m l Q ->
  exec c t m l (fun c t m l => P c t m l ∧ Q c t m l).
Proof. (* induction on one exec, inversion on other *) Qed.
```

No Totality Whether or not to allow `ext_spec` instances that ignore the postcondition entirely is not clear. On one hand, an external call `exit()` that immediately terminates the program and thus does not execute the continuation seems sensible. However, a specification that allows terminating the program in this fashion does not say anything useful about what the program does: calling `exit()` immediately would satisfy any postcondition. A proper encoding of multiple termination mechanisms using omnisemantics would involve keeping the postcondition and conveying to it the specific outcome: `rets` of type `list word` might be replaced with an element of `Inductive outcome := Ret (rets : list word) | Exit (code : byte)`. The additional `Exit` outcome would then need to be threaded through `exec`. For more information on encoding alternative execution outcomes such as exceptions, see [Cha+23, Appendix C] and the associated code supplement.

The shortcut contemplated here could be ruled out using the following generic requirement about the total-correctness semantics: `totality := ∀ c t l m P, exec c t m l P -> ∃ T M L, P T M L`. It turns out that this property does not hold for Bedrock2 `exec` as currently specified. Even though any I/O specification currently used with Bedrock2 does not ignore its postcondition, the Bedrock2 semantics are not

total because the definition of `exec` itself may call the postcondition under subtly contradictory assumptions! The specifics of this are described along with the integration proof of the Bedrock2-RISC-V-integration case study in Subsection 4.5.3, showing that the design choice that inadvertently allows `exec` proofs for some programs to be wrapped up early does not compromise system correctness.

4.1.3 Terminating and Reactive Programs

The definition of semantics of loops and function calls is pleasantly free of termination-proof techniques. Choosing to define `exec` as the inductive closure of the omnisemantics rules limits proofs of `exec` to well-founded repetitions of applications of these rules, which is the appropriate generalization of finite proof trees to rules with universally quantified premises. In terms of program semantics, this corresponds to accepting programs that terminate in an input-dependent but a potentially arbitrarily large (unbounded) number of steps. Bedrock2 does not have a fundamental need for this flexibility: the programs have finite state and hence code receiving a fixed sequence of inputs must either enter an infinite loop or terminate in a bounded number of steps. However, while the state of and each input to a Bedrock2 program are finite, they are not small, and reasoning about their cardinality is not convenient. Having to manipulate explicit bounds on how long the program is allowed to run as required with the earlier interpreter-based specification of Bedrock2 and with popular step-indexed frameworks would be an unwelcome chore for all proofs affected by it. Defining the desired notion of correctness inductively conclusively bypasses this concern.

Using an inductive definition with omnisemantics rules specifies total correctness, which affects the meaning compiler-correctness theorems. Programs that would run forever under some input are ruled out and treated as undefined behavior just as if they violated memory-access rules. Thus a compiler-correctness theorem stated in terms of preservation of behavior of well-defined programs does not apply to compiling an infinite loop or a perpetually reactive echo server! The concern for real-life miscompilation is indirect but plausible: translating `while(1) ; f()` to `f(); while(1)` is a plausible emergent behavior for data-flow optimizations. A modular proof of integrated correctness needs to do better regardless: it can only assume the compiler upholds its mechanized specification.

A coinductive definition can be used to specify partial correctness using omnisemantics rules. Coauthors and I wrote about this option in [Cha+23, §2.4], showing basic properties and a proof of type soundness for a functional language with mutable references and nondeterminism. In the context of integration proofs, however, using a partial-correctness judgement seems even less satisfying than enforcing termination regardless of how the definition of partial correctness is encoded. For an extreme example, an infinite no-op loop satisfies every partial-correctness postcondition. Instead, I sought alternative techniques for deriving useful theorems about systems con-

taining Bedrock2 programs that can run arbitrarily long using only total-correctness semantics.

Well-Founded Total Correctness I would like to impress on the reader how general the notion total correctness is when application-specific assumptions about nondeterministic input are allowed. Not only is there no need to specify a bound on the number of execution steps before the program stops, but the execution may proceed without bounding its future extent at any point. In particular, it is permissible for the specification of the program to delegate the responsibility for deciding when the program should terminate to the outside world that the program accepts input from. To encode this, the `ext_spec` for an I/O operation would invoke its postcondition under a hypothesis which promises that either a well-founded measure has decreased or the input from the I/O operation is a request to shut down the program! If the program terminates in response to this request, it can be proven to satisfy a well-founded-total-correctness specification.

For a minimal example, consider analyzing a Bedrock2 program in the context of some unknown `deadline : nat`. Define the I/O specification in terms of this number and the length of the I/O trace `t` accumulated during the execution so far:

```
ext_spec t "keepGoing" [] post :=
  if deadline <? length t
  then post [0]
  else forall v, post [v]
```

Then the code snippet `go = 1; while go { doWork(); go = keepGoing() }` can be proven to terminate according to the weakest-precondition rules in Subsection 3.2.7 and in turn `exec`: The decreasing measure is `deadline - length t`, and the loop only continues if the second case of `ext_spec` is taken. Note that the source code of the program does not mention `deadline`, and the proof can be instantiated for any specific value of this variable. This universally quantified result guarantees that the program can be run for arbitrarily long *at the choice of the environment* but will not diverge on its own. Further, the postcondition of the program must be satisfied regardless of how soon it is terminated, enforcing that no amount of `doWork()` can get into a state where the program would crash when asked to shut down. Intuitively,

$$\textbf{toral correctness} \leftrightarrow \textbf{reactivity} \wedge \textbf{clean shutdown}$$

This termination-proof strategy, Bedrock2 and the RISC-V omnisemantics discussed in the next section were used for integrated verification of a memory-mapped cryptographic accelerator and its driver. A register-transfer-language model of the MMIO peripheral was related to a cycle-accurate model expressed in terms of abstract state, and this model was used as the `ext_spec` during the instantiation of Bedrock2 seman-

tics. The Bedrock2 driver was proven correct for any deadline, but the system-level theorem references the RTL of the MMIO peripheral and its concrete execution time in low double-digit cycles. It is my understanding that Jade Philipoom and Samuel Gruetter worked on the Bedrock2 side of this effort as a part of project Silver Oak, and that the MMIO interaction proof was completed successfully[2].

Theoretical Significance of Divergence Taking a step back from the concrete design considerations, there appears to be a deep relationship between whether interactive input and output are allowed and whether terminating or not-necessarily-terminating programs are of interest. A safely terminating program that is not allowed to accept any input is simply a specific way of specifying its output – it may be of great practical importance, but mathematically it is just a layer of indirection. Allowing for the possibility of divergence suddenly means that we can learn something by running the program. A program that searches for counterexamples to a conjecture may never terminate, but if it does, we know the conjecture does not hold. This is especially important if we wouldn't have been able to otherwise prove that this program would terminate. In this scenario, a semantics that identifies divergence with undefined behavior is not adequate. If the compilation-correctness definition *assumes* that the program terminates, a nonterminating input program may be compiled to a terminating program or a program that triggers undefined behavior.

This situation changes if runtime input and input-dependent well-founded termination are allowed. The program can have a vast range of behaviors under different inputs and run for arbitrarily long, as long as it terminates cleanly when requested externally. Only the pure form of the conjecture-checking example is ruled out; a variant that regularly checks for a termination request is permitted and satisfies the same curiosity.

Open Problem: Mixed Inductive-Coinductive Omnisemantics With that said, I would like to clarify that I do not see input-dependent termination arguments as a particularly elegant solution. Even though clean termination is an independently desirable property, and granting that proving it can highlight system-design oversights that otherwise might have been missed, a direct definition of semantics of programs that run arbitrarily long would likely be more satisfying.

A promising route around this bolted-on accounting mechanism is the *mixed inductive-coinductive* specification style [NU10] that can distinguish between *perpetually reactive* and *silently diverging* programs. However, the definitions are more involved than the single-inductive omnisemantics presented here, and the approach has not become popular or been tested in substantial experiments. I have confirmed that mixed

[2]Those interested in the development logs could start from `https://github.com/mit-plv/bedrock2/issues/186#issuecomment-839817142` or `https://github.com/project-oak/silveroak/blame/001e221a/firmware/IncrementWait/IncrementWaitToCava.v#L17`

inductive-coinductive definitions of big-step omnisemantics are possible and satisfy basic litmus-test properties. Pursuing a nontrivial case study with the goal of establishing and testing reasoning principles would be a reasonable next step, but the inelegance of externally guaranteed termination does not seem sufficient to motivate the effort. Nevertheless, a setting where nonterminating behaviors must be preserved by compilation might have a lot to gain from future work in this direction: factoring the terminating and nonterminating cases into one mixed inductive-coinductive omnisemantics definition appears promising as a means to deduplicate compiler proof effort associated with these cases. Can the duplication be eliminated entirely from proofs of compiler passes that are correct in either case?

4.2 Small-Step Omnisemantics for RISC-V

This section will cover the small-step analogue of the big-step omnisemantics discussed so far, but first, some background on the RISC-V specification to which this technique was applied is in order.

The formalization [Bou+21] of RISC-V used in the specification of the Bedrock-to-RISC-V compiler is factored into two qualitatively different parts: the instruction set and the execution environment. The goal of this factoring is to enable instruction-set description to be reused in substantially different execution environments that do not share the same notion of state, I/O, or execution outcomes.

To achieve this modularity, each instruction is specified in terms of primitive operations that have some resemblance to microoperations of reconfigurable processors but exist for specification purposes only. Each execution-environment specification then instantiates these primitives with models in terms of the concepts appropriate for that domain. The primitive RISC-V-semantics operations relevant to integrated verification with Bedrock2 fall into four categories:

1. Memory access: loads and stores of 1, 2, 4, and 8 bytes, for example `LoadWord context address`. The `context` tag is used to disambiguate between instruction and data fetches, whose use of different data paths and cache hierarchies in common implementations is observable through sequential-consistency violations even in single-core systems. Memory and MMIO addresses are treated uniformly from the perspective of the instruction set.

2. Register access: `SetRegister reg val` is implemented uniformly across all current environment specifications, but `SetCSRField` distinguishes between control and status registers implemented and not implemented by the execution environment.

3. Program-counter management: most instructions affect the process counter uniformly, with `EndCycleNormal` incrementing it by 4, but jumps and exceptions

use `SetPC`.

The key definition of the instruction-set specification is that of the execution of one instruction in terms of primitive operations:

```
Definition run1 : M unit :=
  pc <- getPC;
  inst <- loadWord Fetch pc;
  execute (decode iset (combine 4 inst));;
  endCycleNormal.
```

Instruction decoding is described using a boring functional program with many conditionals, a small snippet of which is reproduced here:

```
  let oimm12 := signExtend 12 (bitSlice inst 20 32) in
  let rs1 := bitSlice inst 15 20 in
  let opcode := bitSlice inst 0 7 in
  if (opcode =? opcode_LOAD) && (funct3 =? funct3_LB) then
    then Lb rd rs1 oimm12
    else if (opcode =? opcode_LOAD) && (funct3 =? funct3_LH)
      then Lh rd rs1 oimm12
```

Execution is more interesting, but still relatively uniform:

```
| And rd rs1 rs2 =>
    x <- getRegister rs1; y <- getRegister rs2;
    setRegister rd (x <&> y)
| Lui rd imm20 =>
    setRegister rd (ZToReg imm20)
| Beq rs1 rs2 sbimm12 =>
    x <- getRegister rs1; y <- getRegister rs2; pc <- getPC;
    when (x == y)
      (let newPC := pc + ZToReg sbimm12 in
       if remu newPC 2 /= 0
       then raiseExceptionWithInfo 0 0 newPC
       else setPC newPC)
```

Standard tricks for encoding state-manipulating syntax in a functional program are applicable to a basic deterministic instantiation of the primitive operations. The parameter `M unit` in the type signature of `run1` gets instantiated as `state -> unit * state`, and primitive operations are defined as functions returning `M R` where `R` is the type of the binder at the left of the `<-` above. This instantiation of the specification is automatically validated against the `riscv-tests` test suite after every change.

Compiler-Facing Challenges A deterministic instantiation of the primitives as functions in the error and state monad was also sufficient for early prototyping of the correctness proof of the Bedrock2-to-RISC-V compiler, omitting I/O support. With `run1` available as a function from state to state, the following definition and instruction-specific lemmas about it became the workhorse of the compiler-correctness proof. Here it is, slightly reformatted, but with a comment from the original:

```
(* alternative way of saying "∃ n, run1^n f m = m' ∧ P m'" *)
Inductive runsTo (m : @RiscvMachine w var) (P: _ -> Prop) : Prop :=
| runsToDone (_ : P m)                : runsTo m P
| runsToStep (_ : runsTo (run1 m) P) : runsTo m P.
```

The similarity to omnisemantics is striking in retrospect, but small-step omnisemantics were not known at the time. Instead, my enthusiasm for modeling input and output in Bedrock2 programs using nondeterminism inspired extensive experimentation with monad transformers and eventually stopped at an alternative direct definition of the monad `M`: `OStateND S A := S -> option (A * S) -> Prop`. The `None` constructor of `option` was used to indicate undefined behavior, and then `option (A * S) -> Prop` is a predicate that specifies which outcome are possible. This encoding is isomorphic the set of all possible outcomes, and `Return`ing a set of a single outcome can be encoded as follows:

```
Return A (a : A) : OStateND S A :=
  fun (s : S) (oas: option (A * S)) => oas = Some (a, s)
```

The `Bind` (`a <- m; f a`) operator for `OStateND` was then implemented by carefully defining the predicate that delineates the set of possible outcomes from running one command after another:

```
Bind A B (m : OStateND S A) (f : A -> OStateND S B) : 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);
```

A distinctive feature and limitation of `OStateND` is apparent from this definition: the outcome final state `obs` is bound *before* the inputs by which it is determined. As read directly in order, `Bind` for `OStateND` answers the question "what intermediate outcomes could have lead to this final outcome?" Either the `a <- m; f a` failed because `m` failed, or some intermediate result `a` and state `s` from the first primitive `m` allowed the second primitive `f` to result in `obs`.

This definition is a mouthful, but it was workable in the sense that after some puzzling over it we managed to wrap it in a more intuitive definition that generalizes the type signature of `runsTo` from earlier:

```
Definition mcomp_sat {S A} (m : OStateND S A) (s : S) P :=
  ∀ (o: option (A * S)), m s o -> ∃ a s', o = Some (a, s') ∧ P a s'.
```

The proofs of of `mcomp_sat` for `m` representing specific instructions still needed unfolding of `mcomp_sat`, and further unfolding the `OStateND` models of the underlying primitives frequently led to large formulas involving numerous existential quantifiers for intermediate values, calling for the use of elaborate ad-hoc proof-automation scripts. The RISC-V-specific lemmas required by the compiler proof were nonetheless recovered and extended to I/O, which I attribute to Samuel Gruetter's consistent efforts. This inglorious success signaled a green light to verification of drivers and construction of a demonstration system.

Processor-Facing Challenges Having `mcomp_sat` as a definition about `OStateND` and lemmas that establish it was only half of the story for integrated verification: it remained to show that `run1` instantiated with `OStateND` implies satisfactory instruction execution on the Kami [Cho+17] reference processor. As the specification and Kami reference processor use different factorings of the decoding and execution logic, the proof of the correspondence between them needs to consider an arbitrary instruction and match up outcomes of conditional tests and values of arithmetic expressions between the two sides. Despite my best efforts at proof-automation engineering and consultation with local experts, partial evaluation of the two specifications and equational reasoning between them remains a bottleneck: executing the proof script on 52 instruction-specific cases takes over 15 minutes.

The struggle to get the required time down to even this much motivated me to revisit `OStateND` and its `Bind` definition in particular because the existentially quantified intermediate variables would each need to be extracted from the `mcomp_sat` hypothesis and then substituted right away. Actually proving the conjectured relationship between RISC-V spec with I/O and Kami transition labels would have to wait until the proof-generation performance improved.

4.2.1 Weakest-Precondition Interpretations of Free Monads

Glancing at the specification of RISC-V instruction execution again, it is clear that the state of the processor is handled linearly, and thus creating new variables for intermediate versions of it is completely unnecessary. Similarly, while using named variables for intermediate variables can beneficially preserve some sharing when those variables are used multiple times, this benefit is thrown away immediately in the proof connecting this specification to Kami, to reconcile the fact that the two projects do not reliably use intermediate variables at the same granularity.

```
| Add rd rs1 rs2 =>
   x ← getRegister rs1; y ← getRegister rs2;
```

```
   setRegister rd (add x y)
```

The Bedrock2 weakest-precondition definition (see Section 3.2) intentionally preserves sharing, but its state is updated incrementally line-by-line. The reason it is challenging to follow the same pattern here is that the environment can only specify the meanings of `x <- ma; f x` and each operation such as `getRegister` and `setRegister`; it does not have access to the syntax tree as a whole. However, there is a standard technique for bypassing this limitation: `Q` can be instantiated with a freer monad [KI15], which defines each primitive operation as a *constructor* of an inductive datatype listing possible actions, not a function. Bind (`x <- ma; f x`) still has to flatten the associative structure of the binders to satisfy the monad laws, but no interpretation of the primitive operations is required.

```
Inductive free {action : Type} {result : action -> Type} {T : Type} :=
| act (a : action) (continuation : result a -> free)
| ret (x : T).
Arguments free : clear implicits.
Fixpoint bind {A B} (mx : free A) (fy : A -> free B) : free B :=
  match mx with
  | act a k => act a (fun x => bind (k x) fy)
  | ret x => fy x
  end.
```

It is then possible to write a structurally recursive interpreter that consumes these newly syntactic programs. The appropriate interpretation in the context of nondeterministic input with the possibility for undefined behavior is again a weakest-precondition transformer.

```
Fixpoint interp (a : free output) (s : state) post : Prop :=
  match a with
  | ret x => post x s
  | act a k => interp_action a s (fun r => interp (k r))
  end.
```

Following the pattern from the sequence case of the Bedrock2 weakest-precondition generator, the rule for an action uses the weakest-precondition interpretation of the rest of the program as the `Prop`-returning continuation of that action. This construction allows `mcomp_sat` to be defined directly as `interp`, bypassing the need for `OStateND`! At the same time, the specifications of individual actions are simplified, returning almost to the form that they were presented in before support for nondeterminism was added:

```
Definition interpret_action (a : riscv_primitive) (m : RiscvMachine) :
  (primitive_result a -> RiscvMachine -> Prop) -> Prop :=
  match a with
```

```
  | GetRegister reg => fun (post: word -> RiscvMachine -> Prop) =>
      let v := getReg m.(getRegs) reg in
      post v m
  | SetRegister reg v => fun post =>
      let regs := setReg reg v m.(getRegs) in
      post tt (withRegs regs m)
  | GetPC => fun post => post m.(getPc) m
  | SetPC newPC => fun post => post tt (withNextPc newPC m)
  ...
  | StartCycle => fun post =>
      post tt (withNextPc (word.add m.(getPc) (word.of_Z 4)) m)
  | EndCycleNormal => fun post => post tt (updatePc m)
  end.
```

Encoding the model of a nondeterministic primitive operation is as straightforward as in Bedrock2, but some care is required to figure out what the appropriate model for an MMIO action should be. In Bedrock2, we had the luxury of creating new syntax to distinguish MMIO loads and stores from normal memory operations (and I/O calls from internal function calls). In RISC-V, the instruction format is fixed, and it seems plausible that the same instruction could be used with a computed address to access both I/O addresses and normal memory. Further, an instantiation of the RISC-V semantics for DMA-based I/O should allow the distinction between I/O addresses and normal memory to change dynamically as ownership of memory regions is transferred between the peripherals and the processor.

The model I proposed for the RISC-V specification is that the memory is represented as a partial map from addresses to bytes, and access to any address not present in the memory is handled like an I/O action. That does not mean that loads and stores to arbitrary addresses are permitted: the first proof obligation for nonmemory address-space access is the `isMMIO` that also appears in Bedrock2 `ext_spec`. Deviating from Bedrock2 for historical reasons, the alternative to `ext_spec` for a RISC-V MMIO load is encoded using the precondition `canMMIORead` and the postcondition `MMIOReadOK`:

```
Definition nonmem_load (n : nat) (ctxid : SourceType) a m post :=
  isMMIO a ∧ canMMIORead n (getLog m) a ∧
   ∀ v, MMIOReadOK n (getLog m) a v -> post v (withLogItem (a,n,v) m).

Definition load (n : nat) (ctxid : SourceType) a m post :=
  (ctxid = Fetch -> isXAddr4 a m.(getXAddrs)) ∧
  match Memory.load_bytes n m.(getMem) a with
  | Some v => post v m
  | None => nonmem_load n ctxid a m post
  end.
```

Unfolding `interp_action` or `load` in a hypothesis reveals no spurious existential

quantifiers, and givens such as `isMMIO` in `nonmem_load` can be separated automatically into further hypotheses at the speed that the Coq proof engine allows for this operation. Further, unfolding `nonmem_load` reveals a universally quantified hypothesis, assuring that no matter which input the machine code receives from the external world, its execution will proceed soundly. The Kami-RISC-V connection proof immediately specializes this hypothesis to the input variable revealed through inverting the Kami step relation. But before looking into this proof in more detail, I would like to discuss how the `interp` construction can be extended to the execution of more than one instruction.

4.2.2 "Eventually" Combinator for Small-Step Omnisemantics

Machine code is executed one instruction at a time, with no designated end in sight; the basic RISC-V standard does not even include a halt instruction. Instead, the interesting property to ask of a piece of machine code is whether it will reach some application-specific final condition, such as jumping to a specified address or triggering a particular exception. Reaching a desirable state in `n` steps could be defined by `n`-times sequential composition of `step := interp run1`, but fixing `n` ahead of time would severely limit the flexibility provided to the machine-code implementation. Further, it may be impossible to know reliably ahead-of-time how many instructions a loop that performs I/O will execute, so the definition of `runsTo` for nondeterministic RISC-V programs cannot require a specific deadline to be specified anyway.

The definition of nondeterministic `runsTo` follows the same structure as the deterministic version, but allows for a set of possible states (`Q`) to be reached by the first step. Every state in this set must reach the specified termination condition `P`, but a separate `runsTo` derivation consisting of an arbitrary number of steps can be provided for each one:

```
Inductive runsTo (s : state) (P : state -> Prop) : Prop :=
| runsToDone   (_ : P s)                                  : runsTo s P
| runsToStep Q (_ : step s Q) (∀ s', Q s' -> runsTo s' P): runsTo s P.
```

Outside machine-readable definitions, I have found it convenient to pronounce `runsTo` "eventually" and write it as a superscript $\Diamond$ on the arrow that represents `step`:

$$\frac{P(s)}{s \to^{\Diamond} P} \qquad\qquad \frac{s \to Q \quad \forall s'.\, Q(s') \Rightarrow s' \to^{\Diamond} P(s')}{s \to^{\Diamond} P}$$

This presentation highlights three properties. First, transforming a `step` into a `runsTo` does not change its type signature. Second, the concept of `runsTo` is closely related to the eventually modality from temporal logic. Third, writing $\Diamond$ where `*` (for zero or more steps) would be written suggests that it is reflexive and transitive. The

complementary "always" combinator is defined in Subsection 4.4.3.

These properties hold, along with postcondition weakening and chained versions as shown for big-step omnisemantics. Further, the deterministic `runsTo` that only allows the first command to reach a single state is a subrelation of nondeterministic `runsTo`.

```
Lemma runsTo_weaken: forall (P Q : State -> Prop) s,
  (forall s', P s' -> Q final) -> runsTo s P -> runsTo s Q.
Lemma runsToStep_chained s P :
  step s (fun s' => runsTo s' P) -> runsTo s P.
Lemma runsTo_trans: forall P Q s,
  runsTo s P -> (forall s', P s' -> runsTo s' Q) -> runsTo s Q.
Lemma runsTo_trans_chained: forall (Q : State -> Prop) (s : State),
  runsTo s (fun s' => runsTo s' Q) -> runsTo s Q.
Lemma runsTo_det : forall s s1 P,
  step s (fun s' => s' = s1) -> runsTo s1 P -> runsTo s P.
```

Unlike Bedrock2 omnisemantics, the RISC-V semantics are total in the sense that if an action executes to a postcondition according to the semantics, then there actually exists an outcome for which the postcondition holds:

```
Lemma interpret_action_total {memOk: map.ok Mem} a s post :
  valid_machine_state s -> interpret_action a s post ->
  exists v s', valid_machine_state s' ∧ post v s'
```

4.3 Verifying Compilers Using Omnisemantics

The ability of omnisemantics directly to encode nondeterminism and undefined behavior in a single judgement simplifies some of the characteristic complexities of behavior-preservation proofs for compilers. This simplification does not come at the cost of precision: compiler-correctness theorems stated in terms of traditional semantics can be proven using omnisemantics. This section is based [Cha+23, §6].

4.3.1 Motivation: Avoiding Both Backward Simulations and Artificial Determinism

Following CompCert's terminology [Ler09], one particular evaluation of a program can admit one out of four possible behaviors: *terminate* (with a value, an exception, a fatal error, etc.), trigger *undefined behavior*, *diverge silently* after performing a finite number of I/O operations, or be *reactive* by performing an infinite sequence of I/O operations. Whether an error such as a division by zero is considered as a terminating behavior or as an undefined behavior is a design decision associated with

each programming language. Ideally, a general-purpose compiler should preserve behaviors, except that undefined behaviors can be replaced with anything. This section will focus on terminating behaviors; the reasoning behind this choice can be found in Subsection 4.1.3. (Nevertheless, lifting this restriction would be interesting future work.)

In the particular case of a deterministic programming language, compiler correctness for terminating programs can be established via a *forward-simulation* proof [3]. Such a proof consists of showing that each step from the source program *corresponds to* a number of steps in the compiled program. The correspondence involved is captured by a relation between source states and target states. Such forward-simulation proofs work well in practice. The main problem is that they do not generalize to nondeterministic languages.

Indeed, in the presence of nondeterminism, a source program may have several possible executions. The restriction to consider only terminating programs means that all executions of the source program terminate but possibly with different results. In this setting, a compiler is correct if (1) the compiled program always terminates, and (2) for any result that the compiled program may produce, the source program could have produced that result. This notion of correctness is called backward behavior inclusion. It may not be intuitive at first, but the inclusion is indeed *backwards*: the set of behaviors of the target program must be included in the set of behaviors of the source program.

To establish backward behavior inclusion, one may set up a *backward-simulation* proof. Such a proof consists of showing that each step from the target program corresponds to one or more steps in the source program[4]. Yet, backward simulations are much more unwieldy to set up than forward simulations. Indeed, in most cases one source-program step is implemented by multiple steps in the compiled program, thus a backward-simulation relation typically needs to relate many more pairs than a forward-simulation relation.

This observation motivated the CompCert project [Ler09] to exploit forward simulations as much as possible, at the cost of modeling features of the intermediate language as deterministic even when it is not natural to do so and ruling out programs whose behavior is not deterministic. For example, rather than revealing pointers as integers,

[3]The uses of "forward" and "backward" here to refer to the direction of compilation, "forward" meaning from source language to target language. This matches CompCert's usage and conflicts with other work [LV95] that uses "forward" and "backward" to refer to the direction of the state transitions.

[4]The number of corresponding steps in the source program should not be zero, otherwise the target program could diverge whereas the source program terminates. In practice, however, it is not always easy to find one source-program step that corresponds to a target-program step. In such situations, one may consider a generalized version of backward simulations that allow for zero source-program steps, provided that some well-founded measure decreases [Ler09].

CompCert semantics allocate pointers deterministically, taking care to trigger undefined behavior for any coding pattern that would be sensitive to the literal values of pointers (see Subsection 2.4.1 for discussion). Even so, runtime input does not fit the fully deterministic model, leading to the technical definitions of receptiveness and determinacy (roughly, capturing the idea of determinism modulo input) so that lemmas for flipping forward simulations into backwards simulations can be stated and proven. Omnisemantics remove the need for this machinery and extend the compiler-correctness proofs to cover programs whose behavior is not deterministically specified. The solutions is presented in three steps:

- Explain how omnisemantics can sidestep the need for backward simulations and artificially deterministic semantics, by carrying out forward-simulation proofs of compiler correctness, for nondeterministic terminating programs.
- Generalize to languages including I/O operations and to the case where the source language and target language are different.
- Analyze a case study, the compilation of Bedrock2 I/O and stack allocation to RISC-V. The latter case compiles nondeterministic commands to deterministic implementations, and both cases feature a big-step semantics for the source language and a small-step semantics for the target language, for free.

4.3.2 Establishing Correctness via Forward Simulations using Omnisemantics

Consider a compilation function written $\mathcal{C}(t)$. For simplicity, assume that the source and target language are identical, that compilation does not alter the result values, and that the language is state-free. These restrictions will be lifted in the next subsection. In this subsection, $t \Downarrow v$ denotes the standard big-step judgment that t evaluates to v, $t \Downarrow Q$ denotes the omni-big-step judgment and that t evaluates to a member of Q, and $\mathsf{terminates}(t)$ asserts that all executions of t terminate safely, without triggering undefined behavior. The compiler-correctness result for terminating programs captures preservation of termination and backward inclusion for results—points (1) and (2) stated earlier using traditional semantics:

BACKWARD-INCLUSION-FOR-TERMINATING-PROGRAMS:

$$\mathsf{terminates}(t) \quad \Rightarrow \quad \mathsf{terminates}(\mathcal{C}(t)) \quad \wedge \quad (\forall v.\ \ \mathcal{C}(t) \Downarrow v \quad \Rightarrow \quad t \Downarrow v)$$

The claim is that this correctness result can be derived from the following statement, which describes forward preservation of specifications.

$$\text{OMNI-FORWARD-PRESERVATION:} \qquad \forall Q. \qquad t \Downarrow Q \quad \Rightarrow \quad \mathcal{C}(t) \Downarrow Q$$

Assume that $\mathsf{terminates}(t)$ holds. Recall that omnisemantics enforces termination and constrains the outcome (this and other preliminaeies are proven in [Cha+23, §2.2]):

$$t \Downarrow Q \quad \iff \quad \mathsf{terminates}(t) \ \wedge \ (\forall v.\ (t \Downarrow v) \ \Rightarrow \ v \in Q)$$

Exploiting this equivalence, the OMNI-FORWARD-PRESERVATION assumption reformulates as follows.

$$\begin{array}{lll} \forall Q. & & (\mathsf{terminates}(t) \ \wedge \ (\forall v.\ (t \Downarrow v) \ \Rightarrow \ v \in Q)) \\ & \Rightarrow & (\mathsf{terminates}(\mathcal{C}(t)) \ \wedge \ (\forall v.\ (\mathcal{C}(t) \Downarrow v) \ \Rightarrow \ v \in Q)) \end{array}$$

The hypothesis $\mathsf{terminates}(t)$ holds by assumption. Now instantiate Q as the strongest postcondition for t, that is, as the set $\{v \,|\, (t \Downarrow v)\}$. This yields:

$$(\forall v.\ (t \Downarrow v) \ \Rightarrow \ (t \Downarrow v)) \quad \Rightarrow \quad \mathsf{terminates}(\mathcal{C}(t)) \ \wedge \ (\forall v.\ (\mathcal{C}(t) \Downarrow v) \ \Rightarrow \ (t \Downarrow v)).$$

The premise is a tautology, and the conclusion is exactly the result to be proven: BACKWARD-INCLUSION-FOR-TERMINATING-PROGRAMS.

4.3.3 Omnisemantics Simulations, I/O, and Cross-Language Compilation

More generally, the behavior of a terminating program consists of the final result and its interactions with the outside world (input and output). These interactions include, e.g., interaction with the standard input and output streams, system calls, etc. Each interaction is called an *event*, and the semantics judgment is extended to collect such events into a *trace* τ. Figure 4-1 shows three illustrative cases of how the rules are augmented with states and traces, making the choice to treat `rand` calls as observable events while reference-allocation nondeterminism remains internal. Choosing which nondeterministic choices are recorded in the trace determines which (external) interactions must be preserved by compilations and which (internal) nondeterministic choices the compiler may resolve as it sees fit. As a particularly fine-grained example, the trace might record that `malloc` was called and succeeded but omit the pointer it returned, to allow for optimizations that reduce the amount of allocation. This level of flexibility appears to be unique to omnisemantics. For a forward-simulation-based compiler-correctness proof, constructing a deterministic model of all internal nondeterminism can be arbitrarily complicated (the CompCert memory model is an example).

For languages of terms (that return values) rather than commands (that do not return values), a representation relation between source-level and target-level values would be needed, so this section will consider commands c from now on. In the current setting, *behavior inclusion* holds between a source-language program and a target-language program if all traces that the target-language program can produce

$$\textsc{omni-big-let-trace}\quad \frac{t_1/s/\tau \Downarrow Q_1 \qquad (\forall (v', s', \tau') \in Q_1.\ ([v'/x]\, t_2)/s'/\tau' \Downarrow Q)}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)/s/\tau \Downarrow Q}$$

$$\textsc{omni-big-rand-trace}\quad \frac{(\forall m.\ m \le n \implies (m, s, (n, m) :: \tau) \in Q)}{(\mathsf{rand}\, n)/s/\tau \Downarrow Q} \qquad \textsc{omni-big-ref}\quad \frac{\forall p \notin \mathsf{dom}\, s.\ (p,\ s[p := v], \tau) \in Q}{(\mathsf{ref}\, v)/s/\tau \Downarrow Q}$$

Figure 4-1: Omni-big-step semantics with traces, selected rules

(according to traditional small-step or big-step semantics) can also be produced by the source-language program. More formally, the traces that can be produced from a starting configuration $c/s/\tau$ are defined as

$$\mathsf{traces}(c, s, \tau) := \{\tau' \mid \exists s'.\ c/s/\tau \Downarrow s'/\tau'\}$$

and a compiler $\mathcal{C}$ satisfies behavior inclusion for a command starting from the initial source-level state s_{src} related to the initial target-level state s_{tgt} and initial trace τ if TraceInclusion as defined below holds.

$$\textsc{TraceInclusion}(c, s_{src}, s_{tgt}, \tau) \ :=\ \mathsf{traces}(\mathcal{C}(c), s_{tgt}, \tau) \subseteq \mathsf{traces}(c, s_{src}, \tau)$$

Assuming omni-big-step semantics $\Downarrow_{src}$ and $\Downarrow_{tgt}$ for the source and target languages, plus a representation relation R between source- and target-language states, let the *omnisemantics simulation*, a compiler-correctness criterion designed to be provable by induction on the $\Downarrow_{src}$ judgment, be defined as follows:

$$\begin{aligned} \textsc{OmnisemanticsSimulation}(c) :=\ & \forall s_{src}\ s_{tgt}\ \tau\ Q.\ R(s_{src}, s_{tgt}) \wedge c/s_{src}/\tau \Downarrow_{src} Q \\ & \implies \mathcal{C}(c)/s_{tgt}/\tau \Downarrow_{tgt} Q_R \\ \text{where } Q_R(s'_{tgt},\, \tau') :=\ & \exists s'_{src}.\ R(s'_{src}, s'_{tgt}) \wedge Q(s'_{src}, \tau') \end{aligned}$$

The goal for this section is to prove that an omnisemantics simulation implies trace inclusion if the source program terminates, i.e. to show

$$\begin{aligned} \forall c.\ & \textsc{OmnisemanticsSimulation}(c) \implies \\ & \forall\ s_{src}\ s_{tgt}\ \tau.\ \mathsf{terminates}(c, s_{src}, \tau) \wedge R(s_{src}, s_{tgt}) \implies \\ & \quad \textsc{TraceInclusion}(c, s_{src}, s_{tgt}, \tau) \end{aligned}$$

This proof relies on two properties: First, soundness of omni-big-step semantics with respect to traditional big-step semantics:

$$\forall c\ s\ s'\ \tau\ \tau'\ Q.\quad c/s/\tau \Downarrow s'/\tau' \wedge c/s \Downarrow Q \quad\implies\quad Q(s', \tau') \tag{4.1}$$

And conversely, that a program that terminates safely and whose traditional big-step executions all satisfy a postcondition also has an omnisemantics derivation:

$$\forall c\ s\ \tau\ Q.\ \mathsf{terminates}(c,s,\tau) \wedge (\forall s'\ \tau'.\ c/s/\tau \Downarrow s'/\tau' \implies Q(s',\tau')) \quad \implies \quad c/s/\tau \Downarrow Q \tag{4.2}$$

To show trace inclusion, i.e. $\mathsf{traces}(\mathcal{C}(c), s_{tgt}, \tau) \subseteq \mathsf{traces}(c, s_{src}, \tau)$, it suffices to assume a target-language execution $\mathcal{C}(c)/s_{tgt}/\tau \Downarrow s'_{tgt}/\tau'$ producing trace τ' and to show $\tau' \in \mathsf{traces}(c, s_{src}, \tau)$. By applying (4.2) to the source program (whose termination is assumed) and setting $Q(s'_{src}, \tau') := c/s_{src}/\tau \Downarrow s'_{src}/\tau'$ so that the second premise becomes a tautology, we obtain the source-level omnisemantics derivation $c/s_{src}/\tau \Downarrow (\lambda s'_{src}\ \tau'.\ c/s_{src}/\tau \Downarrow s'_{src}/\tau')$. Passing this fact into the omnisemantics simulation yields $\mathcal{C}(c)/s_{tgt}/\tau \Downarrow (\lambda s'_{tgt}\ \tau'.\exists s'_{src}.R(s'_{src}, s'_{tgt}) \wedge c/s_{src}/\tau \Downarrow s'_{src}/\tau')$. Now we can apply (4.1) to this fact and the originally assumed target-level execution and obtain an s'_{src} such that $R(s'_{src}, s'_{tgt})$ and $c/s_{src}/\tau \Downarrow s'_{src}/\tau'$, which by definition is exactly what needs to hold to have $\tau' \in \mathsf{traces}(c, s_{src}, \tau)$.

4.3.4 Case Study: Compiling Stack Allocation

This case study illustrates the case of a transformation that implements a nondeterministic source-language command using deterministic target-language commands. The transformation consists of adding a *stack-allocation* feature to the Bedrock2 compiler. Proving this transformation correct in Coq after I had sketched it informally took Samuel Gruetter only a few days of work, and most of the work was *not* concerned with dealing with nondeterminism.

This smooth outcome is in stark contrast to the outlook of using traditional evaluation judgments: verifying the same transformation would have required either more complex invariants, to set up a backward simulation; or completely rewriting the memory model so that pointers are represented by deterministically generated unobservable identifiers, to allow for a compiler-correctness proof by forward simulation. In fact, addressable stack allocation was initially omitted from Bedrock2 exactly to avoid these intricacies (based on the experience from CompCert), but switching to omnisemantics made its addition local and uncomplicated.

Recall that the stack-allocation feature consists of a command $\mathsf{let}\ x = \mathsf{stackalloc}\ n\ \mathsf{in}\ c$ that assigns an address to variable x at which n bytes of memory will be available during the execution of command c (but not after c finishes). The compiler implements this command by allocating the requested n bytes on the stack, computing the address at runtime based on the stack pointer.

The key challenge is that the source-language semantics does not feature a stack. The stack gets introduced further down the compilation chain. Thus, the simplest way to assign semantics to the stackalloc command in the source language is to say

that it allocates memory at a *nondeterministically chosen* memory location. This nondeterministic choice is described using a universal quantification in the omni-big-step rule shown below, reproduced from Section 4.1.1 with some presentational liberties:

$$\frac{\begin{array}{c}\forall m_{new}\ a.\ (\mathsf{dom}\ m_{new} \cap \mathsf{dom}\ m) = \emptyset\ \ \wedge\ \ \mathsf{dom}\ m_{new} = [a, a+n) \quad \Longrightarrow \\ c/(m \cup m_{new})/\ell[x := a]/\tau \Downarrow \lambda m'\ \ell'\ \tau'.\ P(m' - m_{new}, , \ell', \tau')\end{array}}{(\mathsf{let}\ x = \mathsf{stackalloc}\ n\ \mathsf{in}\ c)/m/\ell/\tau \Downarrow P}\ \textsc{stackalloc}$$

In the source language, the address returned by stackalloc is picked nondeterministically, whereas in the target language the address used for the allocation is deterministically computed, as the current stack pointer augmented with some offset.

Compiler-Correctness proof The compiler-correctness proof proceeds by induction on the omnisemantics derivation for the source language, producing a target-language execution with a related postcondition. The simulation relation R describes the target-language memory as a disjoint union of unallocated stack memory and the source-language memory state. Critically, the case for `stackalloc` has access to a universally quantified induction hypothesis (derived from the rule shown above) about *target-level executions of* $\mathcal{C}(c)$ *for any address* a.

As the address of the stack-allocated memory is not recorded in the trace, the compiler-correctness proof is free to instantiate it with the specific stack-space address, expressed in terms of compile-time stack-layout parameters and the runtime stack pointer. Reestablishing the simulation relation to satisfy the precondition of that induction hypothesis then involves carving out the freshly allocated memory from unused stack space and considering it a part of the source-level memory instead, matching the compiler-chosen memory layout and the preconditions of the stackalloc omnisemantics rule. It is this last part that made up the vast majority of the verification work in this case study; handling the nondeterminism itself is as straightforward as it gets.

Note that it would not be possible to complete the proof by instantiating a with a compiler-chosen offset from the stack pointer if the semantics recorded the value of a in the trace. The (unremarkable) proof for the interact command of Bedrock2 also has access to a universally quantified execution hypothesis, but it *must* directly instantiate its universally quantified induction hypothesis with the variable introduced when applying the target-level omnisemantics input rule to the goal, to match the target-language trace to the source-language trace. Either way, reasoning about the reduction of nondeterministim in an omni-forward-preservation proof boils down to instantiating a universal quantifier.

Design Decisions Around Proving Stack-Space Usage The first version of verified software-hardware based on Bedrock2 and the Kami RISC-V processor did not support stack allocation, but its compiler did support (non-recursive) function calls and computed the required amount of stack space. The `exec.stackalloc` rule of Bedrock was deliberately left to use a vacuous universal quantification in case the program runs out of memory, because the compiler-correctness theorem handles stack-usage accounting outside of the omnisemantics judgment, in an additional external judgment. In particular, this means that if `exec.stackalloc` is applied with a memory m whose domain already contains all (or almost all) addresses (which are 32-bit or 64-bit words), there might be no m_{new} and a such that the left-hand side of the implication above the line in `exec.stackalloc` holds, so any postcondition P can be derived (but the respective stack-usage judgement will be unprovable).

Effectively, this means that the source-language evaluation rules do not guarantee that the program never runs out of memory. This choice simplifies the program-logic proofs for concrete input programs but requires additional work in the compiler: the compiler performs a simple static-analysis pass over the call graph of the program to determine the maximum amount of stack space that the program needs. Since this analysis rejects recursive calls, the space upper bound is not hard to compute. The compiler-correctness proof contains an additional hypothesis requiring that at least that computed amount of memory is available in the state on which the target-language program begins its execution.

An alternative approach would be to introduce a notion of "amount of used stack space" in the source-language semantics and include an additional precondition in the `exec.stackalloc` rule that requires this amount to be bounded. This approach would put more complexity into the verification of source programs, while simplifying the compiler-correctness proof. In order to allow recursive calls and dynamically chosen stack-allocation sizes, reasoning about the amount of stack space in the program logic seems to become unavoidable, in which case this alternative approach would be preferrable.

4.3.5 Compilation from a Language in Omni-Big-Step to One in Omni-Small-Step

If the semantics of the source language of a compiler phase are most naturally expressed in omni-big-step, but the target language's semantics are best expressed in omni-small-step semantics, it is convenient to prove an omni-forward simulation directly from a big-step source execution to a small-step target execution. What follows is an attempt to give a flavor of the proof obligations that arise from switching from omni-big-step to omni-small-step during the correctness proof.

The RISC-V machine state is modeled in s_{tgt} as a quadruple of a memory m (that

contains both instructions and data), a register file *rf* mapping register names to machine words, a program counter *pc*, and a trace τ. One can prove an omni-forward simulation from big-step source semantics directly to small-step target semantics:

$$\forall s_{src}\, s_{tgt}\, P.\ R(s_{src}, s_{tgt}) \wedge s_{src} \Downarrow P \quad \Longrightarrow \quad s_{tgt} \longrightarrow^{\Diamond} (\lambda s'_{tgt}.\exists s'_{src}.\ R(s'_{src}, s'_{tgt}) \wedge P(s'_{src}))$$

where R asserts, among other conditions, that the memory of the target state s_{tgt} contains the compiled program.

The proof proceeds by symbolic execution of the target-language program by applying target-language rules and discharging their side conditions using the hypotheses obtained by inverting the source-language execution, with the only difference that instead of using the derived big-step rule `runsToStep_chained` for chaining, one now uses the following two rules: `runsToStep_chained` and `runsTo_trans`.

Applying `runsToStep_chained` turns the goal into an omni-single-small-step goal about RISC-V machine code with a given postcondition, which is suitable to discharged directly by unfolding `mcomp_sat` and `interp_action`. On the other hand, applying `runsTo_trans` creates two subgoals containing an uninstantiated unification variable for the intermediate postcondition. The unification variable appears as the postcondition in the first subgoal, so an induction hypothesis with the concrete postcondition from the theorem statement can be applied. In the second subgoal, this postcondition becomes the precondition for the remainder of the execution.

4.4 Omnisemantics and Kami

This section covers the proof connecting the RISC-V specification to the verified pipelined processor from Kami [Cho+17]. Originally, the Kami four-stage processor was proven against a single-cycle model described in the same hardware-description language in the sense of backward behavior inclusion. Both artifacts model input as nondeterministic, but Kami does not have a notion of undefined behavior and is specified using traditional operational semantics. The connection proof relies on undefined behavior in our RISC-V-specification instance to rule out cases where the Kami processor would not behave in a manner worth specifying more generically.

The Kami artifact also stands out from the rest of the Bedrock2 ecosystem in that it was designed, developed, and published independently prior to the overall integration-verification effort. Thus I could not ensure that otherwise-underconstrained design decisions would be resolved in ways maximally suitable for integration and combined proof, bringing about qualitatively different challenges during the integration proof. Rather than seeking a satisfying way to encode an interface specification which was already understood informally, reconciling the Kami processor with the Bedrock2-compiler's RISC-V specification started with known differences between the imple-

mented and specified behavior and progressed towards the least cumbersome compromise. Most significantly, the Kami processor started out without any support for runtime input and assumed that the program code is fixed and separate from the data memory. Joonwon Choi worked with us to implement relevant changes in Kami and to adapt the proofs to match.

4.4.1 Kami Language and Processor

The Kami framework centers around a Bluespec-style hardware description language with rules that can atomically `Read` and `Write` the `Register`s of multiple modules. If all rules follow the discipline to `Call` at most one state-modifying method on each module, the execution of each individual method is also atomic. The execution of the entire rule is predicated on all guards in `Assert` statements reached during its execution evaluating to nonzero values. Consider the following extended example, a Kami specification (`pdec`, for "decoupled memory") for issuing load instructions:

```
Let memReq := MethodSig ("rqFromProc" -- "enq")(RqFromProc) : Void.
Definition pdec := MODULE {
  Register "pc" : Pc addrSize <- pcInit
  with Register "rf" : Vector (Data dataBytes) rfIdx <- rfInit
  with Register "pgm" : Vector (Data instBytes) iaddrSize <- Default
  with Register "stall" : Bool <- false
  ...
  with Rule "reqLd" :=
    Read stall <- "stall";
    Assert !#stall;
    Read ppc : Pc addrSize <- "pc";
    Read rf <- "rf";
    Read pinit <- "pinit";
    Read pgm : Vector (Data instBytes) iaddrSize <- "pgm";
    Assert #pinit;
    LET rawInst <- #pgm@[toIAddr _ ppc];
    Assert (getOptype _ rawInst == $$opLd);
    LET addr <- getLdAddr _ rawInst;
    LET srcIdx <- getLdSrc _ rawInst;
    LET srcVal <- #rf@[#srcIdx];
    LET laddr <- calcLdAddr _ addr srcVal;
    Call memReq(STRUCT{"addr" ::= #laddr;      "op" ::= $$false;
                       "byteEn" ::= $$Default; "data" ::= $$Default});
    Write "stall" <- $$true;
    Retv
  ...
```

The declaration of the method `memReq` names the module where the implementation

of this method would be found after linking. Until then, the external method is assumed to have arbitrary (potentially nondeterministic) behavior but no undefined behavior. The `Call` to `memReq` at the end of the function passes arguments, setting `op` to `false` to indicate a read rather than a write. The one-bit `Register` called `stall` is used to ensure that the `reqLd` rule does not execute again (and emit another memory request) while the processor waits for a response from the memory: specifically, `Assert !#stall` on the second line of the rule prevents execution until the flag is cleared. Similarly, the implementation of the `enq` method of the FIFO `rqFromProc` (called as `memReq`) can cancel the execution of the entire rule if the FIFO is full.

Returning to the last `Assert` before the `Call`, we can see the instruction-decoding logic described using the pure (combinational) functions `getOptype`, `getldAddr`, and `getLdSrc`. The execution of the load instruction is described in terms of selecting the correct entry from the register file `rf` and combining it with the immediate offset operand `addr` using the pure function `calcLdAddr`. Setting the destination register to the loaded value is handled in a different rule that executes when the response from the memory is available; that rule also increments the process counter `pc`.

The specification and implementation of the Kami processor do not use `memReq` to fetch instructions (`rawInst`). Instead, the program memory is separated from the data memory and modeled using an (impractically large) `Register` file `pgm` in the specification. This design choice allowed the pipelined processor to be proven to match the memory operations of its single-cycle specification exactly, even though the fetch stage speculatively loads instructions that are not known to be needed for execution. In fact, the proof relies on the invariant that the program memory of the implementation matches that of the specification model while simultaneously treating requests to data memory as external calls with arbitrary behavior. The experiments described in the original Kami paper [Cho+17] instantiated the processor cores with concrete machine code for `pgm` in Coq before translation to Bluespec for simulation or synthesis.

To support integration with the Bedrock2 ecosystem while avoiding significant reengineering, the Kami processor (and its specification above) were augmented with machinery to load the program from external memory on reset. The register `pinit` is asserted in the rules for normal instructions to prevent execution until the internal program memory has been appropriately initialized. As writes to normal memory do not propagate to `pgm` during execution, the modified Kami processor still cannot be used for dynamic code loading. However, this modification as sufficient for a defensible generalization of the RISC-V semantics assumed by compiler proof and can be proven sound against the Kami processor's implementation of RISC-V. The full reconciliation of this limitation with the compiler proof is described in Subsection 4.4.4.

4.4.2 Refinement-Based Specifications

This subsection and the next answer the following questions:

1. How are the specifications of and theorems about Kami processors stated?
2. What would a specification of a Kami processor running a specific program look like?
3. Which generic statement relating a Kami processor to the Bedrock2 infrastructure could allow such specific specifications to be proven for new programs without revisiting the processor?

Both implementations and specifications of Kami processors are written as modules in the Kami language. The semantics of Kami modules are given by the relation `kstep m s1 s2` that models execution of one nondeterministically chosen rule from module `m` transitioning from state `s1` to `s2`. For synthesizable modules that contain definitions of all methods they call, the choice of which rule to execute is the main source of nondeterminism, but the definition of `kstep` also extends to modules that call external methods. Calls to methods not implemented by `m` are recorded in an event trace, and the return values of these methods are modeled as nondeterministic. For example, the `pdec` module from the previous subsection individually admits traces where the external calls to `memRq` and its complement `memRep` give inconsistent and nonsensical results.

As Kami does not have a notion of undefined behavior, the one-to-many relation `kstep m`, which is defined using traditional operational semantics, can be soundly extended to arbitrarily many steps (`^*`) and used for correctness specifications without additional bookkeeping. The "implements" relation (`<<==`) is defined in terms of (backward) trace inclusion, in a manner equivalent to the following transition-system-style definition:

```
Definition Trace (m: Module) (tr : list Label) :=
  ∃ s, (kstep m)^* m.(initial) s ∧ tr = s.(trace).
Notation "m1 <<== m2" := (∀ t, Trace m1 t -> Trace m2 t).
```

The main correctness lemma about the four-stage processor is stated using this relation: the trace of external interactions that the pipelined processor performs matches a possible trace of the `pdec` model, `p4st <<== pdec`. This property is proven without any assumptions about how the data memory operates. At the highest level, the proof consists of a sequence of refinement steps that justify individual optimizations in the four-stage processor such as pipelined instruction fetching.

```
p4st <<== p3st <<== ... <<== pdec
```

The `p4st` module is the main artifact that is extracted from Kami and used in the "lightbulb" case study at the end of this chapter. However, treating a processor separately from memory is a rather hardware-centric perspective, and a higher-level specification about the combined behavior of the processor is desirable both on its own merits and for connection to Bedrock2. This model is constructed in two steps: first, `p4st` is fitted with an adapter `iom` that answers asynchronous memory-access requests by calling similar methods that return the answer right away. The combination satisfies a new specification `pinst`: `p4st ++ iom <<== pinst`, where `++` is the Kami operator that concatenates modules and matches up method calls between them. Then, a mock memory implementation `mm` is concatenated to both sides; this module serves data-memory accesses using a huge register file but forwards MMIO operations to the only required external method `mmioExec`. Thus, the highest-level correctness theorem stated in terms of native definitions of the Kami framework reads as follows:

```
p4st ++ iom ++ mm <<== pinst ++ mm
```

The two sides of this refinement fact are models of the implementation and the specification; they will be referenced later as `p4mm := p4st ++ iom ++ mm` and `scmm := pinst ++ mm`. Further, the traces constrained by `<<==` here are specifically the traces of MMIO actions, each entry of which is either a store of a value to an address or a load of an address with a return value. As input is modeled using nondeterminism: if the system admits a trace with some load return value, it also admits traces with all syntactically valid return values for the same load.

Specifying The Kami-Bedrock2 Connection The integration-verification goal for this section is to bound the arguments the Kami processor passes to its MMIO methods using properties proven about Bedrock2 programs' I/O. More specifically, we would like a top-level correctness theorem of the form `∀ t, Trace p4mm t -> Bedrock2MMIOSpec t`. However, `Trace` is defined using `(kstep m)^*`, which allows an arbitrary (and not necessarily maximal) number of steps, potentially leading only to the middle of the execution of a Bedrock2 function. Thus, the above is not provable for any `Bedrock2MMIOSpec` that specifies that some I/O actions actually happened. This limitation is not purely technical; rather, any theorem that is applicable to observations of arbitrarily short or long executions of the Kami-Bedrock2 system would have this limitation. It makes sense to state a system-correctness theorem in this setting so that it directly asserts a trace property that is prefix-closed by construction:

```
∀ t, Trace p4mm t -> ∃ T, Bedrock2MMIOSpec (t ;++ T)
```

where `t ;++ T` represents the event trace where events `T` happen after `t`. (In the Coq development, further bookkeeping is required to mediate between Bedrock2, RISC-V, and Kami trace formats, but I am omitting it here.) The correctness proof of `p4mm` in terms of `<<==` guarantees that the traces of the implementation are a subset of those

of the specification, so it suffices to show

```
∀ t, Trace scmm t -> ∃ T, Bedrock2MMIOSpec (t ;++ T)
```

As `scmm` is the simplest model of the Kami processor, it does not make sense to look for a proof of this statement within the Kami framework alone. In fact, any proof of the above must rely on assumptions about the initial state of `scmm` and specifically about the program the processor is executing. Further, scenarios where the behavior of `scmm` (and `p4mm`) deviates from that specified for general RISC-V must be ruled out at all stages of execution. The most challenging example of this arises from the Kami processor executing instructions from an internal copy of the initial memory while our RISC-V specification demands sequential consistency between instruction fetches and data stores. Programs can store to computed addresses, so no mechanically checkable condition about the initial state can ensure compatible execution on `scmm`.

For formalizing this precondition, it is helpful to take a step back and notice a commonality between the requirement and the trace specification to be proven. The formalization of the precondition should guarantee that all execution steps use instructions that have not been overwritten in data memory, and the conclusion claims that all multi-step executions produce allowed I/O traces. We can abstract away the details about instruction fetching and MMIO actions and state a more general relationship between `scmm` and our omnisemantics for RISC-V: any invariant about the RISC-V specification can be transferred to `scmm` as long as the invariant rules out states where differences between the specifications can be observed. Indeed, `fun t => ∃ T, Bedrock2MMIOSpec (t ;++ T)` is a (weak, noninductive) invariant for RISC-V code compiled from Bedrock2 code that was proven to satisfy `Bedrock2MMIOSpec`, so an invariant-transfer lemma can yield the desired specification above.

4.4.3 "Always" Combinator for Small-Step Omnisemantics

For traditional small-step semantics, the notion of an invariant is ubiquitous: `P` is an invariant for Kami module `m` iff `∀ s, (kstep m)^* m.(initial) s -> P s`. Defining the analogous concept for omni-small-step semantics in Coq is subject to the same encoding challenges as big-step omnisemantics and the `runsTo` relation (see Subsection 4.2.2). Additionally, the very idea of invariants applies to arbitrarily long executions and to settings such as the current one, where there is no notion of termination. This difference means that an inductive definition in the style of `runsTo` cannot capture the right notion: at some point, the inductive derivation would have to end, but the invariant must continue. Intuitively, the appropriate definition should guarantee that no matter how many times the omnisemantics relation `step s Q` is chained in its own postcondition, that postcondition still implies `P`.

One option is to create a coinductive definition with rules similar to `runsTo`:

```
CoInductive always s (P : RiscvMachine -> Prop) : Prop :=
| alwaysStep Q (_: P s) (_: step s Q) (_: ∀ s', Q s' -> always s' P).
```

The lack of a base case is intentional: proofs of `always` would have to show that `alwaysStep` can be applied indefinitely, as many times as required by any proof that relies on an `always` assumption. The name `always` is inspired by the temporal-logic operator $\Box$, and the notation `s` $\rightarrow^{\Box}$ `P` seems appropriate.

However, unlike inductive types, Coq does not generate any reasoning principles for coinductive types automatically. To overcome this limitation, a coinduction principle can be proven by implementing a proof function that (syntactically) yields at least one application `alwaysStep` per recursive call. For this property, the coinduction principle is actually simpler than the definition itself and can be used directly:

```
Definition inductively s (P : RiscvMachine -> Prop) : Prop :=
  P s ∧ ∀ s', P s' -> step s' P.
Lemma always_inductively : ∀ s P, inductively s P -> always s P.
```

Equipped with either `always` or `inductively`, it is now possible to state the key lemma that connects the Kami processor `p4mm` (including a mock memory) to the RISC-V specification against which the Bedrock compiler is proven:

```
Lemma p4mm_implements_riscv : ∀ P rs ks ks', R ks rs ->
  always rs P -> (kstep p4mm)^* ks ks' -> ∃ rs', R ks' rs' ∧ P rs'.
```

This concise statement packs quite a lot of detail. First, variables `ks` and `ks'` refer to initial and final states of the Kami processor, whereas `rs` and `rs'` refer to the corresponding states of the RISC-V specification. Notice that the representation relation `R` is not a universally quantified variable; it encapsulates key invariants such as the correspondence between registers, traces, and memory between Kami and RISC-V states. This relationship is proven directly for the initial states but appears as a premise in the inductive lemma statement.

Unusually for a lemma lifting a property from one semantics to another, both `always` and `(kstep p4mm)^*` appear as premises. The syntactic similarity of the two occurrences in this lemma statement is explained away by the semantic difference between the predicates themselves. One derivation of `always` talks about arbitrarily many executions of RISC-V machine code and constrains all of them to satisfy `P` at every step. It also makes sense for `always` to appear as premise so that if the RISC-V specification assigns undefined behavior to any state reachable from `rs`, `always rs P` does not hold and the lemma is vacuous. On the other hand, `(kstep p4mm)^* ks ks'` merely asserts that it is possible that the processor implementation reaches state `ks'` during execution. This premise may hold for many `ks'` for the same `ks`, and quantifying over all of them appropriately constrains all possible executions modeled using traditional small-step semantics.

The proof of this lemma first replaces `p4mm` with `scmm` using `<<==` and then proceeds by induction over `(kstep p4mm)^* ks ks'`. Each `kstep` corresponds to zero or one steps according to our RISC-V semantics. In the latter case, the `always` hypothesis is inverted to reveal the corresponding `step` and a new `always` fact for the following steps. I attribute the vast majority of the substantial (human and computer) effort that was required for this proof to reconciling mundane differences between how the two frameworks model the same instruction decoding, arithmetic, and architectural state updates. A number of discrepancies between the Kami implementation of basic RISC-V instructions and our machine-code specification were found during integration and proof; they were reconciled and, once proven, the processor passed the appropriate RISC-V test suite. However, one more technically challenging piece remains to make this proof possible at all: `always rs P` and `R` need to rule out executions that would observe the separate-instruction-memory limitation of `p4mm` and `scmm`.

4.4.4 Reconciling Read-Only Instruction Memory

This subsection will describe how our RISC-V machine-code specification handles read-only instruction memory in the sense implemented by Kami. Directly specifying the exact behavior Kami implements would be feasible and possibly easier than the solution described here, but this solution would not be satisfying as a specification of RISC-V. Instead, I will present a nonobvious compromise that is compatible with RISC-V as specified by its steering committee and allows for integrated proofs with Kami.

While using read-only memory to store program code is common across embedded systems with all instruction sets, the restriction of the Kami processor goes beyond that pattern: in Kami, (writeable) data memory cannot be executed. The original Kami proofs treated instruction and data memory as separate address spaces, placing `p4mm` in the trusty company of Harvard-architecture processor designs such as AVR8 and PIC18. The appeal of this simplification from the perspective of processor implementation does not last when the system is considered as a whole. Use cases that benefit from the power-usage and processor-area reduction of read-only memory for instructions almost invariably rely on the same for storing constant data. If this memory is in a separate address space, then different load and store instructions are needed to access constant and nonconstant memory, which in turn requires duplication of library code for each combination of pointer-operand address spaces.

While Bedrock2 does provide a mechanism for address-space-independent access to read-only constants, there does not appear to be a satisfying way to model the same within the general design of RISC-V. Instead, the Kami-Bedrock2 integration takes the perspective that the Kami instruction memory is a cache between the processor and the data memory. Our modified version of `p4st` starts issuing memory requests to fill this cache upon reset, and it only executes the first instruction after the last

instruction has been loaded. Later execution can use normal RISC-V instructions to access and modify the same addresses that were loaded during processor initialization, but the processor will continue executing the (potentially stale) cached code. Perhaps surprisingly, this behavior is entirely valid according to the RISC-V specification ([WA17, §2.7]):

> RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on a RISC-V hart until that hart executes a FENCE.I instruction.

The Kami processor does not implement `FENCE.I`. There are no known obstacles that would prevent the same instruction-fetching logic that is used to initialize the internal instruction memory from being triggered by an instruction. However, fetching instructions generates memory load requests which are treated as externally visible behavior from the perspective of `p4st` and its immediate specification `pdec`, so the refetching logic of `FENCE.I` could not be verified as an internal detail of the pipeline. A modular verification of a processor that implements `FENCE.I` might involve a nondeterministic specification where instructions can be fetched arbitrarily early using external calls (to accommodate for the latency of the pipeline) and execution is allowed to use any instruction fetched for the process-counter address since the last `FENCE.I`. Specifying these details in a manner compatible with the component structure of an implementation is subtle, and a real effort for satisfying modular verification of a processor that does not implement sequential consistency between the instruction and data paths would likely uncover additional challenges.

As our RISC-V machine-code specification is formalized using omnisemantics, we have the luxury of being able to assign undefined behavior to scenarios where deviations from simple sequentially consistent semantics would be observed. (It is possible that the informal RISC-V specification intends to ascribe a more fine-grained nondeterministic behavior to execution of stale instructions, but our integration-verification case study does not need this flexibility.) Specifically, the environment model we use to instantiate our RISC-V specification for integration with Kami tracks a set of executable byte addresses `XAddrs` in addition to the contents of the data memory. Writing to an address removes this address from `XAddrs`, and a hypothetical implementation of `FENCE.I` would restore `XAddrs` to the entire domain of the memory. The `loadWord` specified by the environment model is called by the instruction-set specification (see Section 4.2) in `run1` as `loadWord Fetch pc`. The environment model checks the access-type flag and, in case of `Fetch`, requires that all four byte addresses spanned by the instruction designated by `pc` are in `XAddrs`.

The proof of any individual instruction-execution step in `p4mm_implements_riscv` relies on this restriction. Specifically, inverting the assumption `always rs P` yields `step rs Q`, and `step` unfolds to `interp run1`, which in turn unfolds to the following hypothesis:

```
H : isXAddr4 rpc riscvXAddrs ∧
     match Memory.load_bytes 4 riscvDataMemory rpc with
     | Some v => interp (execute (decode iset (combine 4 v));;
                         endCycleNormal)
     ...
```

The representation invariant `R` enforces that a fully initialized Kami processor's instruction memory `pgm` matches `riscvDataMemory` for all addresses in `riscvXAddrs` (and that the program counters match). Thus, the proof-automation script can replace the value returned by our RISC-V specification's `Memory.load_bytes` with appropriate reference to Kami instruction memory and proceed to consider instruction decoding.

The `XAddrs` assumption for the Kami-processor-compatibility proof does not come for free: the proof of the Bedrock2 compiler must discharge the `XAddrs` conjunct for every instruction the generated code executes. In practice, proving this comes down to threading the invariant that all program memory is covered by `XAddrs` through the compiler proof, but the statement is not a tautology. First, nothing else about the compiler's specification prevents it from emitting self-modifying code or even packaging a just-in-time compiler into the generated binary. Both of these compilation strategies would misbehave on the Kami processor, and on most processors if `FENCE.I` is not used appropriately. Additionally, Bedrock2 external calls for MMIO are compiled to load and store instructions which would modify `XAddrs` if they acted on data memory. The source-level requirement that addresses passed to the MMIO external calls lie within ranges dedicated to I/O (see Subsection 2.3.1) is required to show that `XAddrs` are preserved and execution of compiled code can resume after an MMIO store.

4.5 Verified Application-System Integration

To demonstrate the Bedrock2-Kami integration, a minimal Ethernet server that turns a light on and off in response to UDP packets was implemented and proven. This section will give an overview of this system, its use of the components described so far, and most importantly the overall correctness theorem proven about it.

The specification of this demonstration system is in terms of application-specific predicates about MMIO traces and the semantics of the Kami hardware-description language. Satisfyingly, the following intermediate specifications are not referenced by the statement of the top-level correctness theorem:

- Separation-logic specifications of the Bedrock2 application code and libraries
- Weakest-precondition verification conditions for Bedrock2 (Section 3.2)

- Omnisemantics for Bedrock2 (Section 4.1.1)
- Invariants and additional requirements of the Bedrock2 compiler
- Definitions of RISC-V instructions (Section 4.2)
- RISC-V environment model (Subsection 4.2.1)
- Kami processor specifications `pdec` and `sc` (Subsection 4.4.1)

Components associated with the above interfaces are proven correct, and these proofs are used to prove the top-level theorem, but the interfaces themselves cancel out during the composition of proofs. For example, the RISC-V specification is central to the proofs of the Bedrock2 compiler and its compatibility with the Kami processor, but the application-level trace specification and the Kami hardware-description language are independent of it. This is a great example of how verification of more components reduces the volume and complexity of the specifications, addressing the ages-old concern that the specifications themselves may have bugs.

Nevertheless, the chosen boundaries of the demonstration system still call for specifications of their own. While much simpler than the internal interfaces listed above, the outermost interfaces are not entirely trivial. In particular, the semantics of the Kami hardware-description language appear in the theorem statement to give meaning to the `p4mm` processor initialized with our application code. Further, the desired MMIO actions for driving an Ethernet controller, receiving Ethernet packets using it, and actuating a general-purpose output pin are described in what could reasonably be read as a program of its own. Still, both of these specifications require far fewer concepts than the internal interfaces and are free of common programming pitfalls such as the possibility of accidentally triggering undefined behavior.

4.5.1 Trace Predicates

The formal definitions of Bedrock2, RISC-V, and Kami all include I/O traces as a core specification technique. In these cases, the traces of one component are related to traces of another component in a general way, without writing down any concrete traces or describing their contents. For example, RISC-V machine code generated by the Bedrock2 compiler is proven to produce the same MMIO access traces as the source-language program. But how can we actually write down the traces we want to prove a concrete Bedrock2 program admits?

The trace of a program that only generates output and does not accept runtime input could be specified as a pure function of its arguments. A bare-minimum extension of functional programming with an input construct could be implemented similarly to the interpreter for primitive operations in the RISC-V specification (Subsection 4.2.1).

However, this model would only allow fully deterministic specifications, and underspecification through nondeterminism is important for keeping specifications concise and on-point.

Omnisemantics (or a similar interpreter) could be used to define a generic functional language with input and internal nondeterminism. In the future, this approach may be well and good, but using omnisemantics to state the top-level theorem of the system that pioneers their use in systems verification may be less than compelling for a skeptical reader. The relationships between omnisemantics and traditional semantics proven in Subsection 4.3.3 and Subsection 4.4.4 do a fair bit to establish omnisemantics as a satisfying alternative to traditional operational semantics. And yet a complete system proven against a simple non-omnisemantics specification using omnisemantics adds to this by demonstrating that no tricks relating to how the state of the system is modeled are required to make the semantics work as desired.

Based on this outlook, I chose to specify the lightbulb demonstration using a set of lightweight combinators for trace predicates inspired by regular expressions. The standard regular-expression combinators are easily defined:

```
Definition choice {T} (P1 P2 : list T -> Prop) : list T -> Prop :=
  fun l => P1 l ∨ P2 l.
Definition concat {T} (P1 P2 : list T -> Prop): list T -> Prop :=
  fun l => ∃ l1 l2, l = l1 ;++ l2 ∧ P1 l1 ∧ P2 l2.
Inductive kleene {T} (P : list T -> Prop) : list T -> Prop :=
| kleene_empty : kleene nil
| kleene_step l1 l2 ( _ : P l1) (_ : kleene l2) :  kleene (l1 ;++ l2).
```

As the trace specifications are proven in Coq, either interactively or using domain-specific proof automation, non-regular combinators can be handled as well. In particular, it is important for application specifications to describe relationships between earlier and later events. Popular matching engines that extend regular expressions allow for named (or numbered) subexpressions and backreferences that assert that a later part of the sequence must be equal to the sequence that matched the referenced expression. Departing from procedural matching, I allow a generalization of this feature where the named value may have a type different from the sequence itself, with a customizable relationship to trace contents. Semantically, this connective is just existential quantification lifted to trace predicates:

```
Definition existsl {A T} (P : A -> list T -> Prop) : list T -> Prop :=
  fun l => ∃ a, P a l.
```

These simple connectives are surprisingly powerful for defining practical specifications of sequential programs. For a small example, here is the specification for the Bedrock2 hardware-abstraction-layer function for setting the value of a general-purpose output pin on a microcontroller, presented without notations:

```
Definition gpio_set_bit (i:Z) (b : bool) : list OP -> Prop :=
existsl (fun v : word =>
  concat
    (eq [("ld", GPIO_DATA_ADDR, v)])
    (eq [("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 b)) (word.of_Z i)))]
)).
```

The predicate applies to lists of MMIO operations described as triples each with an access-type marker (`"ld"` or `"st"`), the address, and the value. A trace would satisfy this predicate if, for some value `v`, it consists exactly of a load returning `v` followed by a store of `v` with the `i`th bit set to `b`. Note that input and output are handled completely symmetrically in the specification, so specifications in this style do not inherently enforce determinism or causality. In fact, it is easy to come up with examples of specifications that cannot be reliably satisfied. Stating that a program will store a value and then read a related value back from an MMIO register can be achieved by swapping the arguments to `concat`, but no program would satisfy that specification in the model where memory-mapped input is modeled as nondeterministic.

4.5.2 Specifying the Ethernet-Connected Lightbulb Controller

The top-level specification of the Ethernet-connected lightbulb controller is stated using the same connectives to define nested abstractions for traces relevant to this application. There are three levels:

1. Driving basic input and output peripherals of the processor, GPIO and SPI
2. Driving the external Ethernet controller over SPI
3. Application actions described in terms of driver actions

I specified the first two layers by translating usage instructions of the devices from their programming manuals, occasionally relying on industry-standard register-naming conventions to fill in the gaps. These trace predicates are tedious but conceptually simple. It is not clear that the trace-predicate representation is any more or less readable than expressing the same actions in a conventional programming language. In fact, the very Bedrock2 programs that implement these specifications could make for a reasonable description of the desired behavior. The main benefit of specifying using trace predicates is that the semantics of Bedrock2, and in particular its absolute requirements for programs, do not need to be considered when auditing the system specification. For example, the proof of the receive function of the Ethernet driver against its specification caught a classic, exploitable buffer-overflow bug.

The top-level specification in terms of driver specifications is pleasantly simple and much easier to read than the corresponding Bedrock2 code. The primary reason for this is that the specification can be factored into conceptually separate behaviors without needing to describe how the program chooses which case is satisfiable based on the input it receives. This flexibility seems inherently linked to the ability to state unsatisfiable specifications: in fact, an early draft of the specification for the initialization sequence of the Ethernet card neglected the possibility of a timeout because the C prototype would just infinite-loop in case the Ethernet card does not respond. Describing the particular polling loop would needlessly complicate the specification, but just allowing failure after too many unsuccessful attempts can be concise.

The full trace-predicate specification of the Ethernet-connected lightbulb controller spans 144 lines, more than 100 lines of which are dedicated to driving the Ethernet controller itself over SPI. Using notations `|||`, `+++` and `^*` for `choice`, `concat`, and `kleene`, the top-level specification in terms of these actions appears below:

```
Definition BootSeq : list OP -> Prop :=
  iocfg +++ (lan9250_init_trace
               ||| lan9250_boot_timeout
               ||| (any+++spi_timeout)).

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_bit 23 cmd.

Definition goodHlTrace: list OP -> Prop :=
  BootSeq +++ ((EX b: bool, Recv b +++ LightbulbCmd b)
               ||| RecvInvalid ||| PollNone) ^*.
```

System-Level Correctness Theorem The top-level correctness property is stated directly in terms of this trace predicate:

```
Theorem end2end_lightbulb: ∀ mem0 t,
  bytes_at (instrencode lightbulb_insts) 0 mem0 ∧
  Trace (p4mm mem0) kt ->
    ∃ t: list (string * word * word),
      KamiRiscv.KamiLabelSeqR kt t ∧
      ∃ T, goodHlTrace (t ;++ T)
```

In words, running the pipelined processor `p4mm` with any memory `mem0` that contains the lightbulb-program machine code at address `0` only produces I/O traces that are related to (prefixes of) traces allowed by the application specification. The prefix closure is important because this theorem holds at *any point* during the execution, without reference to any notion of the software having "completed" a loop iteration. The relation `KamiRiscv.KamiLabelSeqR` simply maps Kami MMIO traces to triples with `"ld"` and `"st"`.

Another way to read this theorem is as system-bring-up recipe: compute the byte list `instrencode lightbulb_insts` in Coq, place it at address `0` in a memory, and arrange for this memory to be connected to a correctly synthesized copy of `p4mm`. Then, behavior described by `goodHlTrace` is to follow based on our proofs. 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. No semantics of instruction sets nor software programming languages need to be trusted in order to trust this theorem, and there is no unverified "host" device in our case study. There is also no linker and no bootloader: the processor starts execution from address 0, which is exactly where the system theorem says the program code should be placed.

On the other hand, according to this theorem alone, the system is not guaranteed ever to perform an I/O action. The correctness proofs of Bedrock2 programs and the compiler to RISC-V guarantee termination. However, the Kami processor is only specified in terms of what is true of all sequences of rule executions. There is no proof that some rule can always execute (lack of deadlocks) or that the available rule executions will complete each RISC-V instruction (lack of livelocks). This concern is not entirely hypothetical. During early prototyping of the Bedrock2-Kami integration, I observed a simple deadlock: a RISC-V load instruction with the destination-register field set to zero should ignore its result, but it was never executed. This issue was fixed along with other issues that were covered by the specification.

4.5.3 Physical Realization

I ran the system discussed in section this on an FPGA and confirmed that it works as expected when communicating with an unmodified Linux computer over Ethernet. This section will give a brief overview of the practical aspects of this experiment and review the formal correctness guarantees and additional considerations in the context of the physical artifact.

First, some steps are required to translate the in-Coq description of the system into a format understood by other tools. The Bedrock2 compiler is executed inside Coq and the resulting machine code (a list of bytes) is printed in hexadecimal using Coq notations. The Kami processor design is extracted using the Coq OCaml extrac-

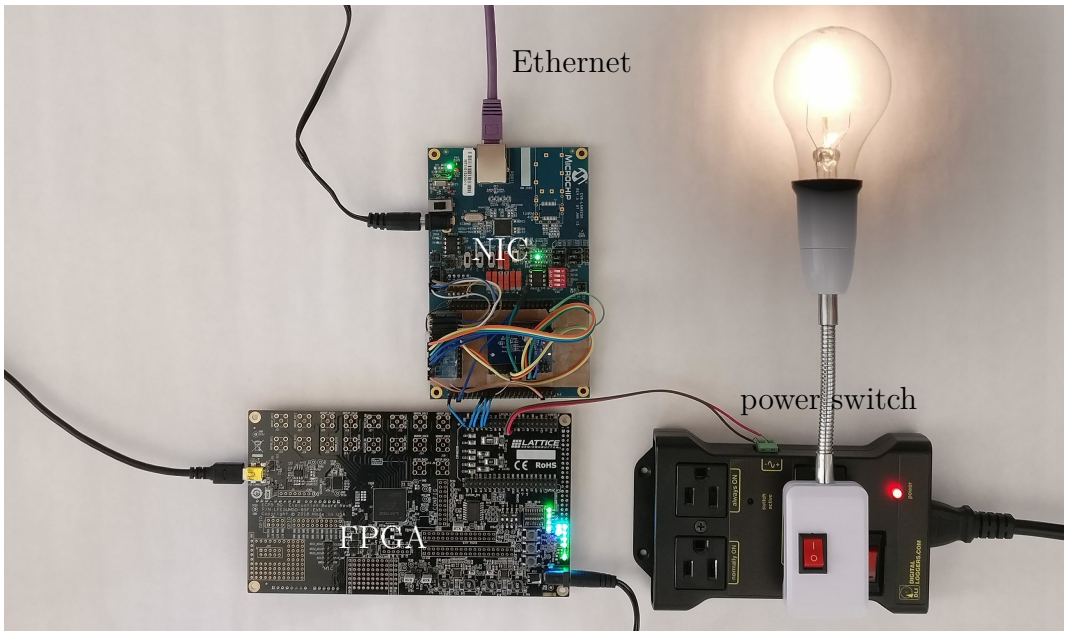

Figure 4-2: Demonstration of hardware-and-software integration using an FPGA

tion feature and exported as Bluespec HDL[5] using a dedicated tool from the Kami framework. The exported code relies on Bluespec implementations of primitive Kami modules for block RAM (for the instruction memory) and FIFOs (between pipeline stages). A small module written in Bluespec wraps the generated code and exposes methods to access the Bluespec FIFOs for memory requests and responses. The open-source Bluespec compiler analyzes this description to find opportunities for parallel execution of rules while preserving one-rule-at-a-time semantics and generates a Verilog RTL design.

The generated Verilog module is instantiated from the top-level Verilog module in this case study along with a memory module and a SPI peripheral. The memory-FIFO methods implemented in Bluespec are accessible from Verilog using a ready-enable interface: the implementation indicates when the method is ready to be called using an output wire, the user of the module can enable its execution using an input wire, and values of data wires are considered exchanged when both control signals are asserted. The memory module implements byte-addressable word-sized operations using four banks of block RAM and specifies the machine code extracted from Coq as the initial value for the memory. The SPI peripheral is triggered by requests to read or write its MMIO address and uses an 8-bit deserializer and serializer concurrently to input and output data; the input value can be retrieved through another MMIO address. Altogether, the top-level Verilog module presents the following interface:

```
module system(input clk,
  output spi_mosi, output spi_csn, output spi_clk, input spi_miso,
```

[5] `https://github.com/B-Lang-org/bsc`

```
  output lightbulb);
```

The five nonclock wires are mapped to FPGA balls and wired out to the Ethernet controller and a power switch controlling a real lightbulb (see Figure 4-2).

To go over the key components of the system concretely, the Ethernet controller is Microchip LAN9250, and the FPGA is Lattice ECP5-85k. Both are used on the manufacturer-recommended development boards. The Verilog code was synthesized for the FPGA using Yosys & Nextpnr [Sha+19]. I would like to emphasize that all software tools used are open-source and actively maintained. The hardware components have some inherent risk of becoming unavailable in the future, but they are simple enough to be simulated on an FPGA in case a direct replication is of interest.

Implications and Limitations of the System Theorem By instantiating all components with appropriate parameters and connecting their correctness proofs, the proof of the main theorem stands on its own. The Coq command `Print Assumptions` can be used to confirm that it only depends on standard Coq axioms: functional and propositional extensionality, Axiom K, and `JMeq_eq`. Thus, there is an unparalleled level of assurance that the system-correctness theorem is true as stated.

Proving any non-degenerate specification about a system also provides assurance of a number of harder-to-formalize but intuitively important properties. For example, the top-level theorem of the lightbulb controller rules out the possibility that an intentional attack through the network interface might take control of the processor, perhaps by exploiting a buffer-overflow bug. If an arbitrary-code-execution attack were possible, the injected code could perform an MMIO operation disallowed by `goodHlTrace`, violating the top-level theorem. Thus, one can rest assured that the verified lightbulb controller is not conscripted to a botnet. Further, the system-correctness theorem also rules out its unwitting participation in denial-of-service attacks by responding to misaddressed packets: the SPI interface can be used to send packets, but the specification only allows receiving them.

An obvious limitation of the top-level theorem is that it only covers a part of the aparatus whose operation is demonstrated. Tooling used to implement the hardware-description-language model of the system is not verified, and its output is executed by an FPGA whose desired operation is assumed. This last requirement is a bit more specific than just "correctness": reprogramming the same FPGA using the interface used to upload our design would obviously result in a different behavior. Finally, the theorem applies to the interface *between* the FPGA and the Ethernet controller shown in Figure 4-2, i.e. the network interface card is excluded from the verification. This last limitation is also a feature: even if the Ethernet controller behaves maliciously (perhaps because it is compromised), it cannot take control of the processor. In short, the system behaves as specified if executed as modeled.

Chapter 5

Connecting Fiat Cryptography

The chapters so far have discussed the integration and combined correctness proof of Bedrock2 programs and a software-and-hardware stack for executing them. A cross-cutting challenge for specifying interfaces and proving the components has been the desire to allow for direct and efficient implementations without locking in particular implementation choices through overspecification. This pursuit is motivated by extensibility and portability as much as by performance. While some standard optimizations such as register allocation are included to exercise the flexibility of the formal interfaces, most components are rather naive from an algorithmic standpoint.

This modest choice of scope was a conscious decision based on the expectation that seeking to verify low-level computer-systems components with many monotonous proof obligations would lead to proof-automation performance scaling issues. This concern was confirmed in multiples: the core proofs of each major component take tens of minutes to build in spite of substantial proof-performance-engineering efforts and occasional use of more manual, less computationally intensive proof methods. For proof-performance reasons alone, pursing verification of a substantial and sophisticated program using the Bedrock2 program would require substantial work on the obligation-proving procedures themselves. The Bedrock2-specific qualitative aspects of the program-proof experience could also stand being improved, but predictable and fast building blocks for solving proof obligations are a major bottleneck for that work as well. Yet it appears desirable to demonstrate that the integration-verification strategy behind the Bedrock2 ecosystem is compatible with code that performs non-trivial computations with good runtime performance. To do so, I will start from an artifact where proof-performance issues have already been addressed and make a connection to the Bedrock2 ecosystem.

This chapter covers the integration verification associated with Fiat Cryptography, a library of high-performance elliptic-curve cryptography implementations proven correct in Coq. There are two branches to this work. First, I will discuss optimizing com-

pilation of field-arithmetic implementations generated from Fiat-Cryptography templates to competitively fast assembly code for common x86 processors. The Coq proof of correctness of the generated code is derived from the proofs of arithmetic-algorithm templates, a compiler-correctness theorem about a fast partial evaluator and rewriting engine, and the correctness proof of a translation validator that checks low-level optimizations. Separately, both field-arithmetic implementations and elliptic-curve code from Fiat Cryptography are compiled to Bedrock2. A traditional certified compiler serves unrolled low-level code, while relational compilation using Rupicola is used for higher-level functions. Integrating these functions with a simple application written in Bedrock2 culminates in another Ethernet-connected demonstration system, this time running on a commercial RISC-V processor and serving as a public-key authenticated garage-door opener.

Historical Overview of Fiat Cryptography I started the Fiat Cryptography project in 2015 and demonstrated the first template that generates modulus-specific field-arithmetic code similar to what an expert would write by the beginning 2016. Specification work for higher-level elliptic-curve algorithms also started around the same time. The MEng theses of myself [Erb17] and Jade Philipoom [Phi18] document early work, including the overall architecture of the library, details of template implementation of field arithmetic, and Coq performance challenges tackled to get the effort off the ground at all. A compact pitch for the template-based strategy for generating proven-correct field-arithmetic code can be found in the subsequent conference paper [Erb+19]. The same publication also presents performance results for the algorithms and prime moduli for which straight-line C code could be generated using built-in Coq partial evaluation mechanisms without exhausting system memory or failing a one-hour timeout. Fiat Cryptography output appears distinctly faster than modulus-generic alternatives, comparable to optimized C code, but slower than expert-written assembly code.

Based on these results, field-arithmetic code generated by Fiat Cryptography was adopted by a number of cryptography libraries for popular software such as web browsers, Linux, OpenBSD, MirageOS, Google and Apple products. Working with the BoringSSL developers, I integrated some of the elliptic-curve code into that library as well. However, the first version of the Montgomery-ladder code transcribed from Coq had a simple (and fortunately inconsequential) pointer-passing bug, highlighting the practical importance of integration proofs for verified elliptic-curve-cryptography code. But the time was not right to tackle this challenge head-on: there was no verification infrastructure that could readily handle C-like code with the simple aliasing patterns that are common in code calling field-arithmetic routines, and iterating on Fiat Cryptography's compilation pipeline was inconvenient due to its slowness.

Seeking to extend Fiat Cryptography's field-arithmetic templates to larger parameters and more complicated arithmetic algorithms, Jason Gross set out to build a

faster, proven-correct partial evaluator. Collaborating closely on the design, we eventually arrived at a solution that supports evaluation-order-directed application of arbitrary rewrite rules, opening up the possibility for its application for constructing certified compilers outside Fiat Cryptography. The new reflection-based implementation achieved a 10x-1000x speedup on existing Fiat-Cryptography build targets, enabled development of new algorithm templates that would have been infeasible to instantiate without it, and substantially simplified the template code by removing the need for the most annoying performance hacks [Gro+22]. This years-long effort also resulted in a new (and faster) trick for feeding Coq goals into certified decision procedures [GEC18] and contributed to a thorough analysis of the performance-engineering landscape of Coq proofs [GE22; Gro21].

5.1 Optimized Assembly Code for Fiat Cryptography

This section describes the extension of Fiat Cryptography to generate x86 assembly code using a compiler based on combinatorial optimization. The code generated in this manner runs faster than that from the previous C backend and enjoys an integrated correctness proof against a mechanized semantics of the assembly language. The techniques used to establish this equivalence are pleasantly simple, and no changes to the previous lowest-level formal interface of Fiat Cryptography were required.

The Fiat-Cryptography toolchain starts from generic functional programs representing arithmetic algorithms. These templates are instantiate with parameters that specify the concrete modulus and key representation choices, for example how many machine words to use to represent each field element. The template instantiations are then partially evaluated, optimized using a database of rewrite rules, and analyzed to determine an appropriate finite-width integer type for each intermediate value. Conveniently, unrolling loops and inlining all functions below the field-arithmetic level is already common due to its performance benefits, so the code generated by Fiat Cryptography can be presented as a list of arithmetic operations. Both the templates and the compiler passes are proven correct for all compile-time and run-time inputs.

Before the improvements described in this chapter, the only supported usage of Fiat Cryptography was pretty-print the generated code as C (or Go, Zig, or Rust). This output would be augmented with documentation comments derived from the theorems proven about the arithmetic operations, but the translation itself is trusted. As the subset of required language features is relatively simple, there has not been a specific reason to be concerned about correctness, but an unverified translation would be an obstacle for integrated verification. Separately, conventional compilers are not all that good at compiling long sequences of arithmetic instructions that make heavy use

of one-bit registers to store carry bits. This limitation was highlighted as the biggest performance bottleneck during performance evaluation of Fiat Cryptography.

Joel Kuepper and collaborators designed a new code-generation backend for Fiat Cryptography that produces highly efficient code tailored to the architecture it runs on. CryptOpt treats the problem of finding efficient machine-code realizations of a straight-line program a combinatorial optimization problem. In other worse, choices about instruction scheduling, instruction selection, and register allocation are treated as parameters to a black-box function that maps the Fiat-Cryptography output to an assembly program. A randomized search algorithm is then used to optimize this function through incremental changes to the parameters and rapid experimental evaluation of the performance of the generated assembly code.

In short, the following mutation rules are used:

- Reordering the computation of an intermediate value to any point between when its arguments are computed and when its result is used.
- Switching to an alternative instruction or snippet to implement the same arithmetic operation, for example `add` vs. `lea` vs. `adox`.
- Assigning a variable to a register or a stack slot.

The optimization is initialized with a family of random variants of the same program. First, each candidate is optimized separately using random local search, iteratively switching to a mutated variant of the candidate unless the mutation makes it slower. Finally, the fastest candidate is run through additional optimization cycles. The performance comparisons are performed with a great deal of care to minimize noise and systematic errors [Kue+22, Appendix A].

Using these optimizations, CryptOpt compiles 9 representative Fiat-Cryptography-Generated functions to assembly code that is (on average) 20% faster than code generated by Clang's output and within the ballpark of assembly-language implementations. Further, a compiler backend that efficiently handles carry flags created the opportunity to usefully add new templates that heavily rely on this feature to Fiat Cryptography. With my guidance, Samuel Tian and Owen Conoly implemented new arithmetic-algorithm templates that capture and generalize state-of-the-art field-arithmetic techniques used for Curve25519 and secp256k1. Instantiating these templates using CryptOpt yields the fastest-known implementations for the relatively new 12th-generation Intel processors and top-tier performance across architectures. A slightly faster microarchitecture-specific implementation is likely to be possible, but a clear benefit of automated code generation is that fast code for new processor designs can be generated with minimal human effort.

5.1.1 Equivalence Checker for Data-Flow-Graphs

CryptOpt's random-mutation process is intended to only generate correct realizations of the input code. However, the standard of confidence for cryptographic implementations that are relied on Internet-wide is higher than that, at least aspirationally. Further, some bugs were observed when using development versions of the optimizer. Some of these bugs resulted in generation of invalid code that was easily detected during testing, whereas others led to generated code that seemed to pass tests but could not be judged correct or incorrect by human analysis.

I designed an equivalence checker to confirm that CryptOpt-generated assembly code is semantically equivalent to the programs produced by Fiat Cryptography's certified pipeline. Chuyue Sun and Jason Gross joined me to implement, prove, and extend this checker as development continued. The basic strategy is simple: both input programs are internalized to a data-flow-graph representation and canonicalized using rewrite rules, with special nodes for associative and commutative operations. Nodes that compute the same function of the same inputs are coalesced on-the-fly. Thus, it is sufficient to check that the outputs of the two programs being checked are represented using the same nodes in the data-flow graph. This checker is proven correct once-and-for-all and integrated with Fiat Cryptography compiler infrastructure.

The complete data model of the equivalence checker is remarkably simple. Inputs to the function are treated as opaque symbols, integer constants and standard word operations can be used, and each node in the data-flow graph can reference any previous nodes using a natural-number index.

```
Variant op :=
| input (bitwidth : N) (_ : symbol)
| const (_ : Z)
| iszero
| add (bitwidth : N)
| addcarry (bitwidth : N)
...
Definition dag : Type := list (op * list N).
```

The key operation for `dag` is merging a new expression, returning the index at which a node representing the root of the expression tree is stored. The expression to be merged is allowed to refer to nodes in the data-flow graph:

```
Inductive expr :=
| ExprRef (_ : N)
| ExprApp (_ : op) (args : list expr).
```

Rewrite rules that translate CryptoOpt's instruction-selection snippets to the arithmetic operation they implement can be implemented as functions that transform ex-

pressions, applied right before merging, and proven correct one-by-one. Specifically, the correctness condition for a rewrite rule is that it should not change the value the expression evaluates to regardless of the `dag` contents or function arguments `G`:

```
Definition rewrite_rule_ok r :=
  ∀ G d e v, dag_ok G d -> eval G d e v -> eval G d (r d e) v.
```

5.1.2 Checking Assembly-Level Optimizations

Symbolic execution of assembly code operating on registers (including flags) is implemented on top of the data-flow graph. Specifically, the symbolic state of an assembly program represents register contents using references to data-flow-graph nodes that evaluate to the appropriate values, or `None` for unknown values. The current value of the symbolic state is threaded through operations using the state monad, with getters and setters for registers, flags, and assembly-instruction operands defined to return data-flow-graph indices instead of values. For example, the `test` instruction that can be used to check whether a register is zero is symbolically executed as follows:

```
Definition Symex (instr : NormalInstruction) : M unit :=
  match instr.(mnemonic), instr.(args) with
  | test, [ea;eb] =>
    a <- GetOperand ea;
    b <- GetOperand eb;
    _ <- HavocFlags;;
    _ <- ZeroFlag CF;;
    _ <- ZeroFlag OF;;
    if Syntax.Equality.ARG_beq ea eb
    then zf <- Merge (rewriterules (iszero, [a])); SetFlag ZF zf
    else ret tt
  ...
```

It is acceptable for the symbolic-execution routine to be partial: the above snipped does leaves the output `ZF` havoced (`None`) if the t`test` instruction is used with two different registers as arguments. Symbolic execution is proven sound against a deterministic semantics of the relevant subset of x86 assembly, which is formalized similarly to RISC-V (see Section 4.2). For example, the reference behavior of `test` is to compute the bitwise and of its arguments:

```
Definition run1x86 (st: machine_state) instr : option machine_state :=
  match instr.(mnemonic), instr.(args) with
  | test, [src1; src2] =>
    v1 <- DenoteOperand sa s st src1;
    v2 <- DenoteOperand sa s st src2;
    let v := Z.land v1 v2 in
```

```
    let st := FlagsFromResult s st v in
    let st := SetFlag st CF false in
    let st := SetFlag st OF false in
    Some (HavocFlag st AF) (* leaves AF unspecified per refman *)
```

Unlike with RISC-V, it is important for efficient formalization that x86 is modeled at the level of the assembly language rather than machine code. There are many more instruction formats than there are assembly-language mnemonics, and instructions vary in what operands they allow. It is much easier to specify a generalization of x86 that, for example, allows arithmetic operations with two memory operands, than to try to exhaustively specify which combinations are allowed. (This approach means that an assembly-language program verified against this specification may fail to be assembled, but it is sound as long as all implemented constructs are modeled correctly). Verifying assembly code is also expedient because Fiat Cryptography's users accept assembly code (not binaries) as contributions to their codebases.

5.1.3 Symbolic Execution of Stack and Array Access

Field-arithmetic routines generated by Fiat Cryptography accept pointers to word arrays as arguments and store results to other designated arrays. Additionally, CryptOpt uses the stack for spilling word-sized temporaries. Thus, there is no pointer-chasing, and all addresses can be represented as offsets from arguments of the function. The symbolic state for memory is simply a list of (address, value) pairs, where both the address and the value are references to the data-flow graph (and the address node is usually an addition node). In this simple bag-of-arrays model, a store can be symbolically executed by updating the value associated with the right address:

```
Definition store a v (s : mem_state) : option mem_state :=
  n <- indexof (fun p => fst p =? a)%N s;
  Some (ListUtil.update_nth n (fun ptsto => (fst ptsto, v)) s).
```

Of course, it is only sound to overwrite just one address node's value if no other address nodes represent the same address. This invariant is enforced by specifying the relationship between the symbolic memory state and the x86 memory state using separation-logic:

```
Fixpoint Rmem (sm : Symbolic.mem_state) : mem_state -> Prop :=
  match sm with
  | nil => F
  | cons (ia, iv) sm' => Rcell64 ia iv * Rmem sm'
  end.
```

Given this definition, the correctness of `store` can be proven in the same sense as in Bedrock2:

```
Lemma store_Rmem : ∀ d s m (HR : R_mem d s m) a va i v v' s'
  eval d a va -> eval d i v ->
  Symbolic.store a i s = Some s' ->
  ∃ m', set_mem m va 8 v' = Some m' ∧ Rmem d s' m'.
```

The memory-representation relation is then used to refine the overall representation relation `R` that relates symbolic-execution states `s` to x86 machine-model states `m`. In terms of this relation, the correctness of the symbolic-execution engine can be stated as follows:

```
Lemma SymexLines_ok : ∀ F G s m asm _tt s',
  R F G s m ->
  Symbolic.SymexLines asm s = Success (_tt, s') ->
  ∃ m', Semantics.DenoteLines m asm = Some m' ∧ R F G s' m'.
```

5.1.4 Integration With Fiat Cryptography's Pipeline

The preexisting verified-compiler pipeline in Fiat Cryptography is implemented using Parametric Higher-Order Abstract Syntax (PHOAS) [Chl08] and specified using a structurally recursive interpreter. As the intermediate language of Fiat Cryptography has no undefined behavior and no nondeterminism, the interpreter can be easily defined in a type-safe manner. `Pipeline.Expr t` is the type of expressions whose evaluation yields values of the type encoded by `t`, which is an enumeration of simple types like integers, pairs, and lists. (The input language of the pipeline also supports higher-order functions, but these are eliminated during compilation through inlining.)

These expressions are translated to data-flow-graph nodes by a simple recursive function whose correctness can be proven by induction on the expression structure. Importantly, `op` symbolic execution of assembly code and symbolic evaluation of PHOAS expressions start with the same `symbol` values for corresponding function inputs. Specifically, a calling-convention model places the arguments of the functional program in arrays at freshly generated symbolic addresses and initializes the argument-passing registers with these addresses before symbolic execution. Fresh values are also used for other registers to track that callee-saved registers are restored to their original value by the end of the execution of the assembly function. Finally, the data-flow-graph indices associated with symbolic addresses of the designated output array are compared to the data-flow-graph indices returned by symbolic evaluation of the PHOAS expression. If the assembly code is correct and rewrite-rules appropriately match the instruction-selection flexibility afforded to CryptOpt, merging corresponding output expressions into the data-flow graph already mapped them to the same index!

The key theorem statement connecting the PHOAS pipeline to the x86 symbolic-execution engine is rather verbose due to calling-convention details and support for

functions with arbitrary numbers of arrays and scalars as input and output. A simplified snippet that shows the structure appears below:

```
Theorem equivalence_checker_correct {t} (F : mem_state -> Prop)
  (asm : Assembly.Syntax.Lines)
  (expr : Pipeline.Expr t) (Hwf : API.Wf expr)
  (args : list (word + list word))
  ...,
  check_equivalence asm expr ... = Success tt ->
  ∃ st' retvals,
  ∧ runx86Lines st asm = Some st'
  ∧ Pipeline.apply (Pipeline.Interp expr) args = Some retvals
  ∧ Rout F retvals stack_size base (to_asm args) savedregs ... st'
```

The verified equivalence-checker was successfully used to check all champion implementations produced by CryptOpt against the Fiat Cryptography code it started from. Most verification tasks completed within a second or two, but the slowest one took five minutes. Experience from Bedrock2 and earlier Fiat-Cryptography prototypes suggests that attempting to use the Coq proof engine to symbolically execute hundreds of lines of code whose compact representation inherently depends on sharing subexpressions would have taken much longer, if successful at all. Implementing a and verifying a dedicated functional program to check the equivalence was feasible because the set of optimizations whose results were to be reconciled with starting-point code was relatively small.

Intuitively, it seems that both the Bedrock2 symbolic-execution strategy and the x86 symbolic-execution strategy here could be implemented either as a functional program or as a proof script, but the engineering tradeoff between these options is steep. Despite being a relatively new addition to Fiat Cryptography, the x86 equivalence checker already has a backlog of feature wishes and annoyance-fixes that would require substantial refactoring and reproving to be implemented. In some cases, a less satisfying but easier-to-apply solution has been adopted instead, further complicating the code and hindering future extension.

5.2 Compiling Field Arithmetic to Bedrock2

Jade Philipoom implemented a compiler from the intermediate representation of Fiat Cryptography to Bedrock2 and proved it correct once-and-for-all. The translation is very simple and does not make any attempt to generate optimized code. Specifically, the generated code for each field-arithmetic operation works in three phases:

1. All inputs are loaded to local variables.

2. Arithmetic operations on local variables are used to compute the result.

3. The words representing the returned field element are stored to memory.

The interface of the Bedrock2 backend is similar to the x86 backend: pointers to input and output arrays are passed into the field-arithmetic function, avoiding the need for it to allocate memory. Further, as all inputs are read before any outputs are written, the generated code (and its proof) support usage where the input and output arrays overlap. The convenience lemma that establishes correctness of compiled code for an arbitrary binary field-arithmetic operation `op` reads as follows:

```
Definition binop_spec {name} (op : BinOp) :=
  fnspec! name (pout px py : word) / (out x y : felem) R,
  { requires t m :=
      bounded_arg1 op x ∧ bounded_arg2 op y ∧
      m =*> FElem px x ∧ m =*> FElem py y ∧
      m =*  FElem pout out * R;
    ensures t' m' :=  t = t' ∧
      ∃ out, eval out = interp_op op (eval x) (eval y) ∧
      m' =* FElem pout out * R ∧ bounded_ret op out}.
```

The definition `FElem` designates an array that is of appropriate length to store a field element and whose cells satisfy the representation predicate of the chosen field-element representation relation from Fiat Cryptography.

The Fiat-Crypto-to-Bedrock2 compiler is available both in the command-line interface of Fiat Cryptography and inside Coq. For example, Bedrock2 code to multiply integers modulo $2^{255} - 19$ using an efficient 10-position base-$2^{25.5}$ representation can be generated along with an appropriate instance of the compiler-correctness proof as follows:

```
Let n := 10. Let s := 2^255. Let c := [(1, 19)]%Z.
Derive fe25519_mul SuchThat (forall functions,
  binop_spec "fe25519_mul" bin_mul
    (field_representation:=field_representation n s c)
    (&,fe25519_mul :: functions))
As fe25519_mul_correct. Proof. derive_bedrock2_func mul_op. Qed.
```

Translating Arithmetic Operations The arithmetic operations supported by Bedrock2 are not an exact match for those that appear in field-arithmetic code generated by Fiat Cryptography. For example, Bedrock2 (and RISC-V) does not support add-with-carry or other word-arithmetic operations that return more than one register's worth of output. These discrepancies are resolved by running the Fiat-

Cryptography rewriting pipeline with Bedrock2-specific rewrite rules. While the design of an appropriate rewrite system can require careful thought, the rules can be proven correct one-by-one. This use of the Fiat-Cryptography rewriter to support a later compiler backend is a testament to its flexibility: indeed, new compiler features can be added and verified using new rewrite rules.

PHOAS-to-Omnisemantics Proofs The Fiat-Cryptography rewriting engine operates on a functional language specified using a total, deterministic interpreter from PHOAS expressions to Coq values. The verification of the compiler to Bedrock2 uses the `cmd_ok` definition (see Section 3.2) for output code. Similarly to other compilers from PHOAS, the proof of this translation proceeds by induction on the well-formedness assumption of the input expression. It is not surprising to me that this combination works, but this example is a first, so I am pointing out its success nonetheless: a semantics with the omnisemantics type signature can be used for compilation from PHOAS even when the generated code uses simple variable names.

5.2.1 If Fiat Cryptography Had Omnisemantics

Avoiding nondeterminism and undefined behavior in the compiler infrastructure of Fiat Cryptography was a conscious choice, but it is less clear-cut now that omnisemantics are available. Like with the CompCert C semantics (see 2.4), specifying the Fiat-Cryptography intermediate language as total and deterministic required nontrivial compromises and forced the design into arbitrary decisions that ended up being revisited multiple times. Specifically, a phase near the end of the pipeline analyzes straight-line code to infer possible ranges of values for variables and intermediate expressions. The syntax tree is then annotated with these ranges. But is is the semantics of a range annotation?

Considering this feature alone, I would like to argue that the appropriate definition for how to evaluate an expression with a range annotation would evaluate the expression, check that the resulting value is within the indicated range, and trigger undefined behavior otherwise. Notice that this particular use of undefined behavior sidesteps the most common criticism of it: the input language does not allow for range annotations on intermediate variables or expressions, so it would not be possible to accidentally trigger undefined behavior or have difficulty proving the lack of it. Instead, (the proof of) the range-analysis pass would have to prove that the annotations it generates are valid, which is already the case.

From the perspective of the subsequent compilation pass, undefined behavior allows for more compilation options. For example, a translator to a target architecture where 64-bit addition is preferable to 32-bit addition (e.g., `lea` on 64-bit x86) could map a 32-bit addition marked with a 32-bit range to a 64-bit operation. The same transformation is not possible if the range annotation has unspecified semantics, or

if it specified to truncate its argument to 32 bits: the latter explicitly invalidates the proposed optimization.

Currently, Fiat Cryptography includes a less-than-elegant function-level "bounds-relaxation-function" knob for controlling the sizes assigned to intermediate variables. Specifically, when compiling to 64-bit Bedrock2, the range-analysis pass itself is configured to relax the range annotations to `0 <= x < 2`^{64} for all intermediates that fit in this range. Even so, the proof of the Fiat-Cryptography-to-Bedrock2 compiler relies on the seemingly arbitrary choice that range annotations generated by the (sound) range-analysis pass truncate their argument. This seems inevitable: the same annotation could have been inserted by a different analysis: rewriting `x mod 2^k` to a range annotation on `x` would be silly but provably equivalent according to the current semantics.

Thus, I wonder whether defining the semantics of an otherwise purely functional language modeled using PHOAS would lead to as smooth of a compiler-proof experience as a definition based on a total interpreter. Some syntactic overhead in postconditions would be inevitable, but it seems like a low price to pay for being able to encode static-analysis results in the syntax tree in a meaningful way. Further, the same encoding would also allow unspecified behavior to be encoded smoothly. This feature could be used to handle operators such as bit-shifts whose fast implementations on different CPU architectures disagree on nonsensical inputs.

5.2.2 Arithmetic-Library Specification Issue Found During Integration Proof

The integration proof of an elliptic-curve library function and the Bedrock2 versions of appropriate Fiat-Cryptography arithmetic routines revealed an undocumented and dissatisfying limitation of an arithmetic-library function. Initializing a field element (which is represented by a number of machine words) with the value of a single machine word is often implemented by simply assigning the argument to the position with weight 1 and zeroing the other positions. However, many multiword representations of field elements also include an invariant about the range of possible values each individual position can take on. The simple implementation is only valid if the posited initial value is within the bounds of the weight-1 position.

All uses of this initialization function that were tracked in Fiat Cryptography passed in single-digit values and were compatible with invariants of all representations in Fiat Cryptography. It thus seems be appealing to keep the uses and the function itself as-is, but making the specification of the word-to-field-element function depend on the internal representation invariant of the field-arithmetic implementation does not allow for modular verification of callers. Specifying a global threshold, say single-digit values, is dissatisfying because it is arbitrary – it gives no guidance for picking the

right threshold that works with future field-arithmetic implementations and elliptic-curve code. Instead, I decided to change the implementation of initializing a field element from a machine word to propagate carries from the initialized position to higher-weight positions. The specification conundrum was thus bypassed at the cost of a handful of extra instructions in the non-performance-critical function.

5.3 Deriving Bedrock2 Code Using Rupicola

Rupicola [Pit+22] is a relational compilation toolchain for translating functional Coq code to Bedrock2 programs and generating proofs of correct translation on-the-fly. A translation from an input language as flexible as Coq's is inherently partial, and Rupicola embraces the partiality at compilation time to make room for sound extension of the compiler with domain-specific patterns. Specifically, Rupicola casts the problem of optimizing compilation as proof search. When asked to compile `f x`, Rupicola asserts the goal that there exists a Bedrock2 program whose `cmd_ok`-postcondition is related to `f x` and proceeds to prove that goal using a database of compilation hints, filling in the definition of the conjectured Bedrock2 program on the fly.

The expected usage of Rupicola involves adding custom lemmas to the compilation-hint database for each Coq function to be compiled. Arbitrarily complicated program fragments with application-domain-specific preconditions and postconditions can be used as compilation hints, but each hint must be triggered by a designated syntactic construct in the Coq code fed to Rupicola. A common pattern is mapping `let`-bound calls to Coq functions to calls of Bedrock2 functions that implement the same operation. For example `let x := y * z mod (2^255-19) in ...` is mapped to `fe25519_mul` from Section 5.2.

The key aspect to keep in mind is that the compilation rule is chosen based on the source-code function alone, but if it has preconditions, these preconditions must be solved by proof automation operating on the symbolic state of the Bedrock2 program being generated. In this case, the symbolic state must contain the field elements `y` and `z` at some addresses that Rupicola can use to call `fe25519_mul`. The proving of preconditions can determine values of arguments with which the compilation lemma is instantiated by filling in existential variables, so the symbolic state of the Bedrock2 correctness proof that Rupicola is building incrementally is also an input to each compilation step. The Bedrock2-level precondition needs to be explicitly specified when starting to compile a function. In this and a number of other ways, deriving a Bedrock2 program using Rupicola is closely related to using the Bedrock2 program logic to derive proof-context definitions (functional programs) that correspond to the computations performed by the Bedrock2 program.

For example, the implementation of IP-packet checksums used in the case study at the end of this chapter is translated from a functional program by using Rupicola

with the following specification:

```
fnspec! "ip_checksum" data_ptr wlen / (data : list byte) R ~> chk,
  { requires t m :=
      wlen = word.of_Z (Z.of_nat (length data)) ∧
      Z.of_nat (Datatypes.length data) < 2 ^ 32 ∧
      m =*> listarray_value AccessByte data_ptr data * R;
    ensures t' m' := t' = t ∧ m' = m ∧ chk = ip_checksum_impl data }.
```

The main benefit of using Rupicola is that the input code has purely functional semantics. While a number of marker notations are required for reliable compilation, these markers unfold to standard Coq primitives. For example `let/n` is just `let`, except it communicates to Rupicola that the location where the value will be stored in Bedrock2 should be chosen based on the name of the variable being defined. Similarly, special definitions to get Rupicola to produce Bedrock2 code with loops unfold to simple recursive functions when proving properties of the source code intended to be compiled with Rupicola. For example, IP checksums for even-length packets can be computed as follows:

```
Definition ip_checksum2_impl (bs: list byte) : word :=
  let/n chk16 := 0xffff in
  let/n chk16 := nd_ranged_for_all
      0 (Z.of_nat (List.length bs) / 2)
      (fun chk16 idx =>
         let/n b0 := ListArray.get bs (2 * idx) in
         let/n b1 := ListArray.get bs (2 * idx + 1) in
         let/n chk16 := ip_checksum_upd chk16 b0 b1 in
         chk16) chk16 in
  let/n chk16 := (~w chk16) &w 0xffff in
  chk16.
```

This approach offers great control over the generated Bedrock2 code:

```
func! (data, len) ~> chk16 {
  chk16 = $0xffff;
  _i = $0;
  while _i < len >> $1 {
    b0 = load1(data + $1 * ($2 * _i));
    b1 = load1(data + $1 * ($2 * _i + $1));
    w16 = b0 | b1 << $8;
    chk17 = chk16 + w16;
    chk16 = (chk17 & $0xffff) + chk17 >> $16;
    _i = _i + $1
  };
  chk16 = (chk16 ^ $(-1)) & $0xffff))
```

```
}
```

A similar derivation is used to implement the ChaCha20 stream cipher used to generate pseudo-random numbers in the case study at the end of this chapter.

Supporting programmatic derivation of Bedrock2 programs was one of my design goals for Bedrock2 and `cmd_ok` in particular; the existence of Rupicola indicates at least some level of success. The vast majority of the design and implementation of Rupicola, including the examples presented in this section and the next, was completed by Clément Pit-Claudel, Jade Philipoom, and Dustin Jamner. A full IP-checksum implementation originally written as a benchmark for Rupicola and an implementation of the ChaCha20 stream cipher are included

5.3.1 Compiling Elliptic-Curve Operations using Rupicola

In parallel with the work to integrate Fiat Cryptography with verified lower-level toolchains, collaborators and I extended the Coq library of proofs of elliptic-curve theory and algorithms. Building on optimized differential addition formulas I verified earlier, David Benjamin and I implemented and proved three variants of the projective x-coordinate ladder for Montgomery curves. I implemented and verified Jacobian coordinates for Weierstrass curves, including mixed addition between affine and Jacobian points that is commonly used with precomputed lookup tables. I implemented and verified precomputation-friendly formulas for XYZT coordinates for Edwards curves. Krit Boonsiriseth and I described and proved the isomorphism between Edwards curves and Montgomery curves, completing the set of connections between different affine coordinate systems for elliptic curves modeled in Fiat Cryptography. With some guidance from me, Krit Boonsiriseth decribed and proved a state-of-the-art lookup-table-based algorithm for computing multiplications between a constant point and variable but secret scalars [Ham12, §3.3].

Relatedly, Ashley Lin implemented an addition-chain generator for fast exponentiation with constant powers. This template is in turn used to implement field-element inversion as per Fermat's little theorem. Thus, the theorem that our implementation of the X25519 Diffie-Hellman function is correct relies on a computer-checked proof that $2^{255} - 19$ is prime, which is discharged through repeated use of a Coq proof of Pocklington's theorem (from Coqprime [TH07]).

All these algorithms are described as functional programs consisting of sequences of field-arithmetic-operation invocations and simple loops, making them excellent candidates for compilation with Rupicola. For the case study in the next section, the derivations for Montgomery ladder and addition-chain exponentiation were realized by Jade Philipoom, Dustin Jamner, and Ashley Lin.

Stack Allocation Rupicola strictly follows the structure of the input program during compilation. This includes not generating memory-management code unless requested. By default, assigning a value to a variable whose Bedrock2 representation is stored in memory means that the contents of the current memory location associated with that variable should be overwritten. This allows for manual control over variable lifetimes, but means that code needs to be annotated with the Rupicola-specific identity function `stack` for every new (nonoverwriting) assignment:

```
Definition ladderstep_gallina (m : positive) (a24 : F m)
           (X1 X2 Z2 X3 Z3: F m) : \<< F m, F m, F m, F m \>> :=
  let/n A := stack (X2+Z2) in
  let/n X2 := (X2-Z2) in
  let/n Z2 := (X3+Z3) in
  let/n Z3 := (X3-Z3) in
  ...
```

Read-Before-Write Aliasing The last subtraction in the above code snippet uses the same memory location for the right-hand-side input and the function output. The field-arithmetic implementations generated by Fiat Cryptography work correctly in spite of aliasing, and their specifications make this clear (see Section 5.2). Compiling Coq code that uses this flexibility using Rupicola does not require any additional effort from the developer. Specifically, Rupicola invokes Bedrock2's separation-logic cancellation (see Section 3.3) to discharge the three memory-related preconditions shown in `binop_spec`, and two of them just happen to be solved identically. This is a great example of how sufficiently predictable proof automation can be reused with ease.

5.4 Garage-Door-Opener Demonstration

I created another demonstration system to illustrate the integration of Fiat Cryptography and Bedrock2. The same drivers and Ethernet controller as in Subsection 4.5.3 are used, this time for both sending and receiving packets. Code generated by the field-arithmetic compiler and Rupicola derivations is linked with Bedrock2 code that was implemented and proven manually. The application is simple: an authorized user identified by an X25519 public key can remotely open and close a garage door controlled by the verified system (Figure 5-1).

Unlike the previous case study, the cryptography-enbaled software here is executed on a commercial RISC-V microcontroller. The Bedrock2-to-RISC-V compiler is still used, and the binary it generates is programmed directly to flash memory that the microcontroller boots from. Again, a system theorem is proven in terms of an invariant that holds throughout execution of RISC-V instructions. Unlike the Kami-based

Figure 5-1: Garage. Development board by Jessie Grosen, Adam Suhl, and myself.

system where hardware-description-language execution was defined using traditional semantics, the omnisemantics-based specification of RISC-V makes it easy to specify that the top-level loop of the demonstration program will keep executing:

```
invariant initial ∧
∀ st, invariant st ->
 mcomp_sat (run1 Decode.RV32IM) st invariant ∧
 ∃ suffix s0 s1, good_trace s0 (getLog (getMachine st) ;++ suffix) s1.
```

The trace predicate that makes up the substance of the top-level specification is stated in terms of Ethernet-driver actions, motor-driver actions, and a high-level specification of the X25519 Diffie-Hellman function, which is defined in terms of back-of-the-napkin elliptic-curve formulas. The complete specification is moderately verbose, but below is an illustrative sample from the key authentication check. First, the contents of the incoming packet are specified in terms of existentially quantified fields with implicit lengths:

```
let incoming : list byte :=
```

```
  (mac_local ++ mac_remote ++ be2 ethertype ++
   ih_const ++ be2 ip_length ++
   ip_idff ++ [ipproto] ++ le_split 2 ip_checksum ++
   ip_remote ++ ip_local ++
   udp_remote ++ UDP_LOCAL ++
   be2 udp_length ++ be2 udp_checksum ++
   garagedoor_header ++ garagedoor_payload) in
```

Then, the I/O trace is related to the packet just described as well as two I/O actions:

```
(∃ gpiostate action : word,
  (lan9250_recv _ incoming +++
  eq ("ld", GPIO_DATA_ADDR, gpiostate) +++
  eq ("st", GPIO_DATA_ADDR, action)) t ∧
```

Reading and writing the GPIO register is not conditional on the contents of the packet. However, the values the motor-control pins are set to are determined based on the Diffie-Hellman shared secret between the authorized public key and a one-time secret key `sk`. In case an invalid command is received, both motor-control signals are set to zero, commanding no action. A client in possession of the authorized key can indicate a request to open or close the garage door by revealing the appropriate 16-byte half of the ephemeral Diffie-Hellman shared secret:

```
  let m := firstn 16 garagedoor_payload in
  let v := le_split 32 (F.to_Z (x25519 sk GARAGEOWNER)) in
  ∃ set0 set1 : word,
  (set0 = 1 ↔ firstn 16 v = m) ∧ (set1 = 1 ↔ skipn 16 v = m) ∧
  action = drive_fwd_bwd gpiostate set0 set1)
```

Nondeterminism and Side Channels A weakness of this specification is that it does not enforce that the system behaves in a deterministic manner or otherwise ensure that seemingly arbitrary choices do not leak the secret key. Standard engineering practices were nonetheless used to keep memory accesses and accesses flow independent of secrets, but nothing was proven about this discipline. For example, the Montgomery ladder implementation compiled from Fiat Cryptography using Rupicola relies on a branch-free conditional-move operation for field elements, and the comparison of the shared secret is implemented using `memequal` from Section 3.2.7. The functional correctness of these countermeasures is covered by the proofs, but the formal model does not account for timing attacks or other side channels. (I did consider enriching the semantics of Bedrock2 and RISC-V with a leakage trace of memory-access addresses and branch decisions and proving that implementations of these interfaces transform it deterministically, but arguing this property for compilation passes such as spill-code generation seems challenging.)

5.4.1 Experimental Confirmation

The first execution of the binary artifact did not respond to the command immediately, leading to a surprise debugging session of supposedly verified code. After implementing support for exporting `gdb`-compatible function symbols from Bedrock2, it became clear that the verified code was in fact running the `x25519` computation, which even completed a handful of seconds later. The reason for the unexpected slowness was mundane: the microcontroller used for this experiment boots with the main clock frequency configured to 5% of the supported maximum. Fixing this setting reduced the delay from command to garage-door action to a fraction of a second.

Conclusion

Integration verification works.

Computer-systems components designed and implemented using different methodologies but specified and proven correct in the same proof assistant can be verified to work together based on specifications that capture requirements already informally associated with the roles of these components. Actually realizing the mechanized proofs for nontrivial systems requires pushing the limits of today's proof assistants, but the conceptual possibility does not appear precarious, and the payoff is immense. The integrated proof rules out violations of engineering disciplines associated with the interfaces between the components verified together. In particular, many of the nastiest bugs and security vulnerabilities fall into this category.

The practical impact of this approach hinges on a solution to the chicken-and-egg problem between creation of tooling-quality proof assistants and systems that rely on them. On one hand, the systems whose integrated correctness was proven as a part of the effort covered in this thesis are very small and simple, leaving room for reasonable skepticism about whether similarly satisfying and rigorous interface specifications can be found for larger systems. This uncertainty in turn leads to hesitation about investing into proof-assistant tooling. On the other hand, directly testing the hypothesis whether larger systems can be specified and proven in a modular manner already requires better proof-automation building blocks in a proof assistant suitable for soundly defining and reasoning about these specifications.

The state of the art today is just about at the threshold where select real-world applications can rely on system-level proofs to achieve assurance out-of-reach for other quality-assurance methods. Some celebration is warranted, and there is a lot to be optimistic about. But the path forward is unclear: who will build the `GCC` and `SQLite` of proof assistants and integration-proof libraries?

List of Figures

Bibliography

[04] *Defect Report #260: indeterminate values and identical representations*. Technical report. ISO TC1/SC22/WG14, Sept. 28, 2004. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

[06] *Defect Report #236: The interpretation of type based aliasing rule when applied to union objects or allocated objects*. Technical report. Version: 1.6. ISO TC1/SC22/WG14, May 8, 2006. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

[92] *Defect Report #028*. Technical report. WG14, Dec. 10, 1992. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_028.html.

[AKZ22] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying Hardware Security Modules with Information-Preserving Refinement". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pages 503–519. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/athalye.

[Alk+08] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. "The Verisoft Approach to Systems Verification". In: *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*. Edited by Natarajan Shankar and Jim Woodcock. Volume 5295. LNCS. Springer, 2008, pages 209–224.

[AN20] Andrew W. Appel and David A. Naumann. "Verified Sequential Malloc/Free". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2020. London, UK: Association for Computing Machinery, 2020, pages 48–59. ISBN: 9781450375665. DOI: 10.1145/3381898.3397211. URL: https://www.cs.princeton.edu/~appel/papers/memmgr.pdf.

[App+20] Andrew W. Appel, Lennart Beringer, Qinxiang Cao, and Josiah Dodds. *Verifiable C: Applying the Verified Software Toolchain to C programs*. Technical report. Version 2.5. May 29, 2020. URL: https://vst.cs.princeton.edu/download/VC.pdf.

[App22] Andrew W. Appel. "Coq's Vibrant Ecosystem for Verification Engineering (Invited Talk)". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2022. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pages 2–11. ISBN: 9781450391825. DOI: 10.1145/3497775.3503951. URL: https://doi.org/10.1145/3497775.3503951.

[Ath+19] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. "Notary: A Device for Secure Transaction Approval". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pages 97–113. ISBN: 9781450368735. DOI: 10.1145/3341301.3359661. URL: https://doi.org/10.1145/3341301.3359661.

[BBW14] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Precise and Abstract Memory Model for C Using Symbolic Values". In: *Programming Languages and Systems*. Edited by Jacques Garrigue. Cham: Springer International Publishing, 2014, pages 449–468. ISBN: 978-3-319-12736-1. URL: http://people.rennes.inria.fr/Frederic.Besson/C_memory_model.pdf.

[BBW15] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Concrete Memory Model for CompCert". In: *ITP 2015 : 6th International Conference on Interactive Theorem Proving*. Edited by Springer. Volume Lecture Notes in Computer Science (LNCS). Interactive Theorem Proving 9236. Nanjing, China, Aug. 2015, pages 67–83. DOI: 10.1007/978-3-319-22102-1_5. URL: http://cs.yale.edu/homes/wilke-pierre/itp-15.pdf.

[BBW17] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "CompCertS: A Memory-Aware Verified C Compiler using Pointer as Integer Semantics". In: *ITP 2017 - 8th International Conference on Interactive Theorem Proving*. Volume 10499. ITP 2017: Interactive Theorem Proving. Brasilia, Brazil: Springer, Sept. 2017, pages 81–97. DOI: 10.1007/978-3-319-66107-0_6. URL: http://cs.yale.edu/homes/wilke-pierre/itp-17.pdf.

[Bes21] Frédéric Besson. "Itauto: An Extensible Intuitionistic SAT Solver". In: *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Edited by Liron Cohen and Cezary Kaliszyk. Volume 193. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 9:1–9:18. ISBN: 978-3-95977-188-7. DOI: 10.4230/LIPIcs.ITP.2021.9. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13904.

[Bev+89] William R. Bevier, Warren A. Hunt, Jr., J. Strother Moore, and William D. Young. "An approach to systems verification". In: *J. Autom. Reasoning* (1989), pages 411–428. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.6467&rep=rep1&type=pdf.

[Bog13] Andreas Bogk. "Bug class genocide". In: *30C3*. 2013. URL: https://media.ccc.de/v/30C3_-_5412_-_en_-_saal_1_-_201312271830_-_bug_class_genocide_-_andreas_bogk/related.

[Bou+21] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andrew Wright, and Adam Chlipala. *Flexible Instruction-Set Semantics via Type Classes*. 2021. DOI: 10.48550/ARXIV.2104.00762. URL: https://arxiv.org/abs/2104.00762.

[Cha+23] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. "Omnisemantics: Smooth Handling of Nondeterminism". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2023). To appear. URL: https://www.charglueraud.org/research/2022/omnisemantics/omnisemantics.pdf (visited on 11/17/2022).

[Cha20a] Arthur Charguéraud. "Separation Logic for Sequential Programs (Functional Pearl)". In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3408998. URL: https://doi.org/10.1145/3408998.

[Cha20b] Arthur Charguéraud. *Separation Logic Foundations*. 2020. URL: https://softwarefoundations.cis.upenn.edu/slf-current/index.html.

[Chl08] Adam Chlipala. "Parametric Higher-Order Abstract Syntax for Mechanized Semantics". In: *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. Victoria, British Columbia, Canada, Sept. 2008. URL: http://adam.chlipala.net/papers/PhoasICFP08/.

[Chl11] Adam Chlipala. "Mostly-automated verification of low-level programs in computational separation l gic". In: *Proc. PLDI*. ACM, 2011, pages 234–245. URL: http://adam.chlipala.net/papers/BedrockPLDI11/BedrockPLDI11.ps.

[Chl13] Adam Chlipala. "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier". In: *18th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2013. 2013, pages 391–402. DOI: 10.1145/2500365.2500592.

[Chl15] Adam Chlipala. "From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification". In: *Proc. POPL*. ACM, 2015. URL: http://adam.chlipala.net/papers/BedrockICFP13/BedrockICFP13.pdf.

[Cho+17] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. "Kami: A Platform for High-level Parametric Hardware Specification and Its Modular Verification". In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017), 24:1–24:30. ISSN: 2475-1421. DOI: 10.1145/3110268. URL: http://doi.acm.org/10.1145/3110268.

[Chr+21] Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. "Wire Sorts: A Language Abstraction for Safe Hardware Composition". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pages 175–189. ISBN: 9781450383912. DOI: 10.1145/3453483.3454037. URL: https://plv.mpi-sws.org/refinedc/paper.pdf.

[Coh+09] Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. "A Precise Yet Efficient Memory Model For C". In: *Electron. Notes Theor. Comput. Sci.* 254 (Oct. 2009), pages 85–103. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2009.09.061. URL: https://sci-hub.se/10.1016/j.entcs.2009.09.061.

[Cor09] Jonathan Corbet. *Fun with NULL pointers, part 1*. July 20, 2009. URL: https://lwn.net/Articles/342330/.

[Doe+22] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. *Leapfrog: Certified Equivalence for Protocol Parsers*. 2022. DOI: 10.48550/ARXIV.2205.08762. URL: https://arxiv.org/abs/2205.08762.

[DSS10] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. "From operating-system correctness to pervasively verified applications". In: *Proc. IFM*. Springer-Verlag, 2010, pages 105–120. URL: https://hal.inria.fr/inria-00524575/document.

[Dun12] George Dunlap. *The Intel SYSRET privilege escalation*. June 13, 2012. URL: https://xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation/.

[Dzy14] Alex Dzyoba. *A tale about data corruption, stack and red zone*. Jan. 27, 2014. URL: https://alex.dzyoba.com/blog/redzone/.

[Erb+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. "Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pages 1202–1219. DOI: 10.1109/SP.2019.00005. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8835346.

[Erb+21] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. "Integration Verification Across Software and Hardware for a Simple Embedded System". In: *PLDI'21* (2021). DOI: 10.1145/3453483.3454065. URL: https://dl.acm.org/doi/pdf/10.1145/3453483.3454065.

[Erb17] Andres Erbsen. "Crafting Certified Elliptic Curve Cryptography Implementations in Coq". MEng Thesis. Massachusetts Institute of Technology, June 2017. URL: http://adam.chlipala.net/theses/andreser_meng.pdf.

[Ert15] M. Anton Ertl. *What every compiler writer should know about programmers or "Optimization" based on undefined behaviour hurts performance*. Technical report. 2015. URL: https://c9x.me/compile/bib/ubc.pdf.

[GE22] Jason Gross and Andres Erbsen. *10 Years of Superlinear Slowness in Coq*. Presented at The Coq Workshop 2022. Aug. 2022. URL: https://jasongross.github.io/papers/2022-superlinear-slowness-coq-workshop.pdf.

[GEC18] Jason Gross, Andres Erbsen, and Adam Chlipala. "Reification by Parametricity: Fast Setup for Proof by Reflection, in Two Lines of Ltac". In: *Proceedings of the 9th International Conference on Interactive Theorem Proving (ITP'18)*. Edited by Jeremy Avigad and Assia Mahboubi. Cham: Springer International Publishing, July 2018, pages 289–305. ISBN: 978-3-319-94821-8. DOI: 10.1007/978-3-319-94821-8_17. URL: http://adam.chlipala.net/papers/ReificationITP18/ReificationITP18.pdf.

[Gro+22] Jason Gross, Andres Erbsen, Jade Philipoom, Miraya Poddar-Agrawal, and Adam Chlipala. "Accelerating Verified-Compiler Development with a Verified Rewriting Engine". In: *Proceedings of the 13th International Conference on Interactive Theorem Proving (ITP 2022)*. Edited by June Andronick and Leonardo de Moura. Volume 237. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Aug. 2022, 17:1–17:18. ISBN: 978-3-95977-252-5. DOI: 10.4230/LIPIcs.ITP.2022.17. eprint: 2205.00862. URL: https://jasongross.github.io/papers/2022-rewriting-itp.pdf.

[Gro21] Jason S. Gross. "Performance Engineering of Proof-Based Software Systems at Scale". PhD Thesis. Massachusetts Institute of Technology, Feb. 2021. URL: https://jasongross.github.io/papers/2021-JGross-PhD-EECS-Feb2021.pdf.

[Gus+20] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, and Martin Uecker. *A Provenance-aware Memory Object Model for C*. Technical report. N2577. ISO TC1/SC22/WG14, Sept. 30, 2020. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2577.pdf.

[Ham12] Mike Hamburg. *Fast and compact elliptic-curve cryptography*. Cryptology ePrint Archive, Paper 2012/309. https://eprint.iacr.org/2012/309. 2012. URL: https://eprint.iacr.org/2012/309.

[Hun89] Warren A. Hunt. *Microprocessor Design Verification*. 1989. URL: http://www.cs.utexas.edu/users/boyer/ftp/cli-reports/048.pdf.

[JBK13] Jonas B. Jensen, Nick Benton, and Andrew Kennedy. "High-Level Separation Logic for Low-Level Code". In: *SIGPLAN Not.* 48.1 (Jan. 2013), pages 301–314. ISSN: 0362-1340. DOI: 10.1145/2480359.2429105. URL: https://doi.org/10.1145/2480359.2429105.

[Jun+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151. URL: https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf.

[Kan+15] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. "A Formal C Memory Model Supporting Integer-Pointer Casts". In: *SIGPLAN Not.* 50.6 (June 2015), pages 326–335. ISSN: 0362-1340. DOI: 10.1145/2813885.2738005. URL: https://www.cis.upenn.edu/~stevez/papers/KHM+15.pdf.

[KBA10] N. Krishnaswami, Lars Birkedal, and J. Aldrich. "Verifying event-driven programs using ramified frame properties". English. In: *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*. Edited by A. Kennedy and N. Benton. United States: Association for Computing Machinery, 2010, pages 63–76. DOI: https://www.cl.cam.ac.uk/~nk480/ramified-frames-tldi10.pdf.

[Keh19] Paul Kehrer. *Memory Unsafety in Apple's Operating Systems*. Technical report. July 23, 2019. URL: https://langui.sh/2019/07/23/apple-memory-safety/.

[KI15] Oleg Kiselyov and Hiromi Ishii. "Freer Monads, More Extensible Effects". In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell '15. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pages 94–105. ISBN: 9781450338080. DOI: 10.1145/2804302.2804319. URL: https://doi.org/10.1145/2804302.2804319.

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. "Formal C semantics: CompCert and the C standard". In: *ITP 2014: Fifth conference on Interactive Theorem Proving*. Volume 8558. Lecture Notes in Computer Science. Vienna, Austria: Springer, July 2014, pages 543–548. DOI: 10.1007/978-3-319-08970-6_36. URL: https://xavierleroy.org/publi/Formal-C-CompCert.pdf.

[Kre13] Robbert Krebbers. "Aliasing Restrictions of C11 Formalized in Coq". In: *Proceedings of the Third International Conference on Certified Programs and Proofs - Volume 8307*. Berlin, Heidelberg: Springer-Verlag, 2013, pages 50–65. ISBN: 9783319035444. DOI: 10.1007/978-3-319-03545-1_4. URL: https://robbertkrebbers.nl/research/articles/aliasing.pdf.

[Kre15] Robbert Krebbers. "The C standard formalized in Coq". PhD thesis. Radboud University Nijmegen, 2015. URL: https://robbertkrebbers.nl/research/thesis.pdf.

[Kue+22] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. *CryptOpt: Verified Compilation with Random Program Search for Cryptographic Primitives*. 2022. DOI: 10.48550/ARXIV.2211.10665. URL: https://arxiv.org/abs/2211.10665.

[Kum+14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. 10.1145/2535838.2535841. San Diego, California, USA: Association for Computing Machinery, 2014, pages 179–191. ISBN: 9781450325448. URL: https://ts.data61.csiro.au/publications/nicta_full_text/7494.pdf.

[KW12] Robbert Krebbers and Freek Wiedijk. *Subtleties of the ANSI/ISO C standard*. Document N1637. ISO/IEC JTC1/SC22/WG14, Sept. 2012. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1637.pdf.

[Lan16] Adam Langley. *memcpy (and friends) with NULL pointers*. June 26, 2016. URL: https://www.imperialviolet.org/2016/06/26/nonnull.html.

[LB08] Xavier Leroy and Sandrine Blazy. "Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations". In: *J. Autom. Reason.* 41.1 (July 2008), pages 1–31. ISSN: 0168-7433. DOI: 10.1007/s10817-008-9099-0. URL: https://xavierleroy.org/publi/memory-model-journal.pdf.

[Lep+22] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. "VIP: Verifying Real-World C Idioms with Integer-Pointer Casts". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498681. URL: https://iris-project.org/pdfs/2022-popl-vip.pdf.

[Ler06] Xavier Leroy. "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 2006, pages 42–54. DOI: 10.1145/1111037.1111042.

[Ler09] Xavier Leroy. "A Formally Verified Compiler Back-End". In: *J. Autom. Reason.* 43.4 (Dec. 2009), pages 363–446. ISSN: 0168-7433. DOI: 10.1007/s10817-009-9155-4. URL: https://xavierleroy.org/publi/compcert-backend.pdf.

[Löö+19] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. "Verified compilation on a verified processor". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pages 1041–1053. URL: https://cakeml.org/pldi19.pdf.

[LV95] Nancy Lynch and Frits Vaandrager. "Forward and Backward Simulations Part I: Untimed Systems". In: *Information and Computation* 121 (1995), pages 214–233.

[Mal+13] Gregory Malecha, Adam Chlipala, Thomas Braibant, Patrick Hulin, and Edward Z. Yang. "MirrorShard: Proof by Computational Reflection with Verified Hints". In: *CoRR* abs/1305.6543 (2013). arXiv: 1305.6543. URL: http://arxiv.org/abs/1305.6543.

[Mem+16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. "Into the Depths of C: Elaborating the de Facto Standards". In: *SIGPLAN Not.* 51.6 (June 2016). https://www.cl.cam.ac.uk/ pes20/cerberus/notes50-survey-discussion.html, pages 1–15. ISSN: 0362-1340. DOI: 10.1145/2980983.2908081. URL: https://www.cl.cam.ac.uk/~km569/into_the_depths_of_C.pdf.

[Mic13] James Mickens. "The Night Watch". In: *;login: logout* (Nov. 2013). URL: https://www.usenix.org/system/files/1311_05-08_mickens.pdf.

[Mil19] Matt Miller. *Memory Unsafety in Apple's Operating Systems*. Technical report. Feb. 7, 2019. URL: https://raw.githubusercontent.com/microsoft/MSRC-Security-Research/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[Moo88] J Strother Moore. *Piton: A Verified Assembly Level Language*. Technical report. Sept. 1988. URL: https://www.cs.utexas.edu/ftp/boyer/cli-reports/022.pdf.

[Myr08] Magnus O. Myreen. "Formal verification of machine-code programs". PhD thesis. University of Cambridge, Computer Laboratory, Trinity College, Dec. 2008. URL: https://www.cl.cam.ac.uk/~mom22/thesis.pdf.

[Nag+10] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "CETS: Compiler Enforced Temporal Safety for C". In: *Proceedings of the 2010 International Symposium on Memory Management*. ISMM '10. Toronto, Ontario, Canada: Association for Computing Machinery, 2010, pages 31–40. ISBN: 9781450300544. DOI: 10.1145/1806651.1806657. URL: https://repository.upenn.edu/cgi/viewcontent.cgi?referer=%5C&httpsredir=1%5C&article=1610%5C&context=cis_papers.

[Nel+17] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. "Hyperkernel: Push-Button Verification of an OS Kernel". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pages 252–269. ISBN: 9781450350853. DOI: 10.1145/3132747.3132748. URL: https://doi.org/10.1145/3132747.3132748.

[Nel+19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. "Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pages 225–242. ISBN: 9781450368735. DOI: 10.1145/3341301.3359641. URL: https://doi.org/10.1145/3341301.3359641.

[NU10] Keiko Nakata and Tarmo Uustalu. "Mixed Induction-Coinduction at Work for Coq". In: *2nd Workshop of Coq users, developers, and contributors* (2010). http://www.cs.ioc.ee/~keiko/papers/Coq2.pdf.

[Phi18] Jade Philipoom. "Correct-by-Construction Finite Field Arithmetic in Coq". MEng Thesis. Massachusetts Institute of Technology, Feb. 2018. URL: http://adam.chlipala.net/theses/jadep_meng.pdf.

[Pit+20] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. "Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs". In: *10th International Joint Conference on Automated Reasoning*. IJCAR 2020. Springer International Publishing, July 2020, pages 119–137. ISBN: 978-3-030-51054-1. DOI: 10.1007/978-3-030-51054-1_7. URL: https://pit-claudel.fr/clement/papers/fiat-to-facade-IJCAR20.pdf.

[Pit+22] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. "Relational Compilation for Performance-Critical Applications: Extensible Proof-Producing Translation of Functional Models into Low-Level Code". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pages 918–933. ISBN: 9781450392655. DOI: 10.1145/3519939.3523706. URL: https://doi.org/10.1145/3519939.3523706.

[Pro] The Chromium Projects. *Memory Safety*. Technical report. URL: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on 10/31/2022).

[Reg16] John Regehr. *The Strict Aliasing Situation is Pretty Bad*. Technical report. Mar. 15, 2016. URL: https://blog.regehr.org/archives/1307.

[Rit88] Dennis Ritchie. *noalias comments to X3J11*. Technical report. Murray Hill NJ: AT&T Bell Laboratories, Mar. 20, 1988. URL: https://www.lysator.liu.se/c/dmr-on-noalias.html.

[Šev+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency". In: *J. ACM* 60.3 (June 2013). ISSN: 0004-5411. DOI: 10.1145/2487241.2487248. URL: https://www.cl.cam.ac.uk/~pes20/CompCertTSO/doc/paper-long.pdf.

[Sha+19] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanović. *Yosys+nextpnr: an Open Source Framework from Verilog to Bitstream for Commercial FPGAs*. 4 page short paper at IEEE FCCM 2019. 2019. arXiv: 1903.10407 [cs.DC].

[Sig+18] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. "Nickel: A Framework for Design and Verification of Information Flow Control Systems". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pages 287–305. ISBN: 9781931971478. URL: https://unsat.cs.washington.edu/papers/sigurbjarnarson-nickel.pdf.

[SMK13] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "Translation Validation for a Verified OS Kernel". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pages 471–482. ISBN: 9781450320146. DOI: 10.1145/2491956.2462183. URL: https://www.cl.cam.ac.uk/~mom22/pldi13.pdf.

[SSS16] Steven Schäfer, Sigurd Schneider, and Gert Smolka. "Axiomatic Semantics for Compiler Verification". In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. St. Petersburg FL USA: ACM, Jan. 2016, pages 188–196. ISBN: 978-1-4503-4127-1. DOI: 10.1145/2854065.2854083.

[Str20] Zygimantas Straznickas. *Towards a Verified First-Stage Bootloader in Coq*. Technical report. MIT CSAIL, May 12, 2020. URL: http://adam.chlipala.net/theses/zygi.pdf.

[Sum+22] Alec Summers, Cathleen Zhang, Connor Mullaly, David Rothenberg, Jim Barry Jr., Kelly Todd, Luke Malinowski, Robert L. Heinemann, Jr., Rushi Purohit, Steve Christey Coley, Trent DeLor, Aleena Deen, Christopher Turner, David Jung, Robert Byers, Tanya Brewer, Tim Pinelli, and Vidya Ananthakrishna. *2022 CWE Top 25 Most Dangerous Software Weaknesses*. Technical report. The MITRE Corporation, Aug. 3, 2022. URL: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[TH07] Laurent Théry and Guillaume Hanrot. "Primality Proving with Elliptic Curves". In: *TPHOL 2007*. Edited by Klaus Schneider and Jens Brandt. Volume 4732. LNCS. Kaiserslautern, Germany: Springer-Verlag, Sept. 2007, pages 319–333. URL: https://hal.inria.fr/inria-00138382.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. "Types, Bytes, and Separation Logic". In: *SIGPLAN Not.* 42.1 (Jan. 2007), pages 97–108. ISSN: 0362-1340. DOI: 10.1145/1190215.1190234. URL: https://doi.org/10.1145/1190215.1190234.

[Tve09] Sergey Tverdyshev. "Formal Verification of Gate-Level Computer Systems". PhD thesis. Saarland University, May 2009. URL: http://www-wjp.cs.uni-saarland.de/publikationen/Tv09.pdf.

[WA17] Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual*. Version 2.2. May 7, 2017. URL: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[Wan16] Peng Wang. *The Facade Language*. Technical report. MIT CSAIL, 2016. URL: https://people.csail.mit.edu/wangpeng/facade-tr.pdf.

[WCC14] Peng Wang, Santiago Cuellar, and Adam Chlipala. "Compiler Verification Meets Cross-language Linking via Data Abstraction". In: *2014 ACM International Conference on Object Oriented Programming Systems L anguages & Applications*. OOPSLA 2014. 2014, pages 675–690. DOI: 10.1145/2660193.2660201. URL: http://adam.chlipala.net/papers/CitoOOPSLA14/CitoOOPSLA14.pdf.

[Wil16] Pierre Wilke. "Formally verified compilation of low-level C code". Theses. Université Rennes 1, Nov. 2016. URL: https://tel.archives-ouvertes.fr/tel-01483676/file/WILKE_Pierre.pdf.

[Wol15] Claire X. Wolf. *PicoRV32 – a size-optimized RISC-V CPU*. 2015. URL: https://github.com/YosysHQ/picorv32.

[Yod21] Victor Yodaiken. "How ISO C Became Unusable for Operating Systems Development". In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*. PLOS '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pages 84–90. ISBN: 9781450387071. DOI: 10.1145/3477113.3487274. URL: https://arxiv.org/pdf/2201.07845.pdf.

[Zao+19] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. "Verifying Software Network Functions with No Verification Expertise". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pages 275–290. ISBN: 9781450368735. DOI: 10.1145/3341301.3359647. URL: https://doi.org/10.1145/3341301.3359647.