Flexible Instruction-Set Semantics via Abstract Monads (Experience Report)

THOMAS BOURGEAT, MIT, USA
IAN CLESTER, Georgia Institute of Technology, USA
ANDRES ERBSEN, MIT, USA
SAMUEL GRUETTER, MIT, USA
PRATAP SINGH, CMU, USA
ANDY WRIGHT, MIT, USA
ADAM CHLIPALA, MIT, USA

Instruction sets, from families like x86 and ARM, are at the center of many ambitious formal-methods projects. Many verification, synthesis, programming, and debugging tools rely on formal semantics of instruction sets, but different tools can use semantics in rather different ways. The best-known work applying single semantics across diverse tools relies on domain-specific languages like Sail, where the language and its translation tools are specialized to the realm of instruction sets. In the context of the open RISC-V instruction-set family, we decided to explore a different approach, with semantics written in a carefully chosen subset of Haskell. This style does not depend on any new language translators, relying instead on parameterization of semantics over type-class instances. We have used a single core semantics to support testing, interactive proof, and model checking of both software and hardware, demonstrating that monads and the ability to abstract over them using type classes can support pleasant prototyping of ISA semantics.

CCS Concepts: • **Theory of computation → Program semantics**; • **Software and its engineering → Semantics**.

Additional Key Words and Phrases: instruction-set semantics, type classes, interactive proof assistants

ACM Reference Format:
Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andy Wright, and Adam Chlipala. 2023. Flexible Instruction-Set Semantics via Abstract Monads (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 192 (August 2023), 17 pages. https://doi.org/10.1145/3607833

1 INTRODUCTION

Machine-language instruction sets are at the center of many aspects of systems implementation and verification. Such an important interface deserves a formal semantics. One semantics should be usable as a specification, for simulation, testing, model checking, and interactive theorem proving. Furthermore, it should be usable for all the above applied to both software and hardware.

Many past projects demonstrated individual semantics usable for minorities of these cases. Leading approaches like Sail [Armstrong et al. 2019] involve domain-specific languages (DSLs) and ad-hoc translators from them into different languages appropriate to different use cases. Certainly, in many ways, it is hard to compete with languages purpose-built for a given style of program. It is not hard to stock a bestiary of potential domain-specific specification languages: high-level-language

Authors' addresses: Thomas Bourgeat, MIT, USA; Ian Clester, Georgia Institute of Technology, USA; Andres Erbsen, MIT, USA; Samuel Gruetter, MIT, USA; Pratap Singh, CMU, USA; Andy Wright, MIT, USA; Adam Chlipala, MIT, USA.

2475-1421/2023/8-ART192
https://doi.org/10.1145/3607833

semantics, distributed-systems invariants, temporal logic for hardware, even pure mathematics. All the above might be brought together in one end-to-end-verified system for e.g. a cryptographic protocol implementation, with different layers verified using different tools and styles. However, there is a lurking risk of a classic "n^2" compilers problem: a translator must be built (and maintained) between each relevant pair of specification language and input language of a formal-methods tool. We might also worry about learnability, for when one engineer encounters a specification in a new domain and must get to know a new language and not just a new library.

The two extremes of fully domain-specific and fully general-purpose specification languages present a challenging space of trade-offs, and it will take much more work to approach anything like a universal answer on which is "better." However, in the study we report in this paper, our goal was to demonstrate that an important specification category, namely instruction sets, can be handled pragmatically within a general-purpose specification language. We will discuss applications in simulation, model checking, and interactive proof of both software and hardware – without needing to write a single new translator between spec languages.

Our semantics is implemented in a relatively small subset of Haskell. Expressing the specification using a small set of core language features allows for straightforward translation to other languages. As Haskell is relatively popular, it has existing translators to Coq and Verilog, which we were able to rely on in our case studies. Achieving desirable code in three languages required some care but was possible through restriction rather than extension of our language usage and tooling. The specific language and set of translators are somewhat incidental to our larger message. The right common specification language of the future might very well be quite different, but at least it would still only need one translator per target language, avoiding the n^2 problem we warned about above. However, two Haskell features are central to how we make a single specification very flexible: monads and the ability to abstract over them using type classes.

Wadler [1992] introduced monads in functional programming as a way to write code that is abstracted over kinds of effects, and type classes [Wadler and Blott 1989] can be used to make that abstraction explicit. Indeed, abstract monads whose available operations are declared by a type class, and implemented differently by various instantiations of the type class, had already been used in projects like the Lava hardware framework [Bjesse et al. 1998]. We will show that essentially the same kind of abstraction is a good fit for the variation across uses of instruction-set specifications. All the work of adapting our RISC-V semantics for a new use case is in defining an instance of our type class that equips monads with a set of operations (henceforth called *primitives*) relevant to specifying RISC-V.

RISC-V interested us as a family of ISAs with an open standard and a rich ecosystem of implementations on both the hardware and software sides. It even already has an institutionally blessed semantics in the Sail DSL. While we would like to prove our semantics against that one eventually, we are held back for the moment by precisely the n^2 problem and specifically language-translation tooling generating unusably verbose code for some input-language features. That behavior is not surprising given that fast simulation via C code was more of a central design goal for Sail, and we may very well see this weakness corrected soon, at which point we would be glad to undertake the specification-reconciliation exercise (no longer needing to do significant Sail-specific tool hacking ourselves).

The remaining sections review important elements of the RISC-V ISA, explain our specification style, and discuss how to cover different use cases with different type-class instances. Our semantics is available on GitHub[1] under a permissive open-source license.

[1]The Haskell code is at https://github.com/mit-plv/riscv-semantics, and the Coq code is at https://github.com/mit-plv/riscv-coq. The artifact virtual machine associated with this paper is at https://zenodo.org/record/7992509.

By reviewing this experience in implementation and application, we aim to make the case that **a formal semantics for an industrial-strength instruction-set family can be implemented fruitfully in a general-purpose language and applied across a representative set of examples, in formal methods and elsewhere, without creating a domain-specific specification language and translators for it**. We make the case by showing how each example works merely by choosing the right type-class instances and perhaps also by calling a from-Haskell translator that already existed.

2 OVERVIEW

Our goal is to translate the RISC-V specification written in English [Waterman and Asanovic 2019a,b] into a broadly applicable, machine- and human-readable formal specification.

We want to support the many different use cases shown in Figure 1: For instance, some users want an executable specification that returns one single final machine state, while others might want to leave some inputs, parameters, or nondeterministic choices abstract and obtain a logic formula restricting the set of possible final states, or obtain a list of final states, or obtain a checker that simply answers whether a given execution trace is allowed by the specification.

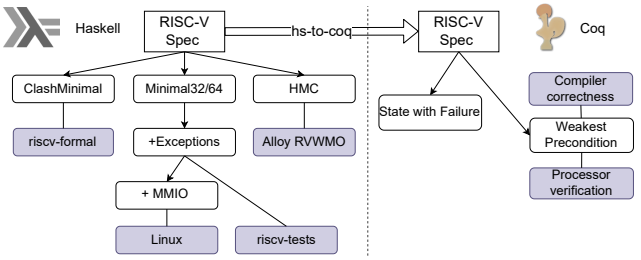

Fig. 1. Projects using our RISC-V specification. The boxes with rounded corners are type-class instantiations. Instances with names that start with "+" are derived from other instances (adding new features). The grey boxes show external projects that our specification interacts with.

These requirements lead us to the following dimensions of parameterization: Supported extensions (see Table 1), bitwidth (32-bit, 64-bit or left abstract)[2], platform-specific details, and use-case-specific details.

2.1 Choice of Language

Serious commitment to a multipurpose specification requires careful thought about the language it should be written in. One goal was to write readable functional programs that could be understood *intuitively* by hardware engineers or compiler hackers, even if they are not familiar with the underlying features (such as monads, type classes, etc.) enabling readability and parameterizability of the specification. Further goals were to create a specification that is practical to use in interactive theorem provers and to connect to other specifications, especially from the hardware world.

We found that Haskell was able to cover most of the constraints, thanks to the following:

- **do** notation provides syntactic sugar for readable imperative-looking code, particularly useful for this specification.
- The Clash compiler [QBayLogic 2020], compiling Haskell (with bounded recursion and finite datatypes) to Verilog/VHDL, is a bridge to hardware-model-checking tools.
- hs-to-coq [Breitner et al. 2018], a compiler that uses the GHC frontend to generate Haskell-like Coq code, is a good bridge to interactive theorem proving in Coq.

We restrict ourselves to concepts supported by these three Haskell compilers (hs-to-coq, Clash, GHC). Via hs-to-coq, we produced a semantics that was chosen as the reference machine model in several Coq projects [Erbsen 2022; Erbsen et al. 2021; Gruetter 2021; Gruetter et al. 2023]. Via

[2]The RISC-V specification also defines a 128-bit variant that we did not consider.

Table 1. Standard extensions of RISC-V 20191213 (excluded: E, D, Q, L, C, B, J, T, P V, N, Zifencei, Zam, Ztso). A checkmark indicates compatibility of a feature with a backend, and parentheses around a checkmark indicate that only a small fraction of that generated code has been exercised so far.

Description	Name	Haskell (GHC)?	Coq (hs-to-coq)?	Verilog (Clash)?
Integer	I	✓	✓	✓
Integer Multiply/Divide	M	✓	✓	✗
Atomics	A	✓	✗	✗
Single Floating-Point	F	✓	✗	✗
Control & Status Registers	Zicsr	✓	(✓)	✗

Clash, we produced a minimal "single-cycle" Verilog execution model for which authors of other specifications checked the agreement between our spec and theirs (see subsection 3.7). Finally, via GHC, we demonstrated the possibility to explore the basics of the RISC-V memory model (subsection 3.6) and to test our specification as a simulator (subsection 3.1 and subsection 3.3).

2.2 Structure of the Specification

Our RISC-V specification is composed of several *extensions* listed in Table 1, and an implementation can choose which subset of them to support. Our formalization of the specification only covers the most important of them.

The primitives. The key to supporting many different use cases is to specify the semantics of each instruction in terms of a small number of *primitives* listed in Figure 2, while leaving the implementations of these primitives to be filled in by the concrete use cases. The primitives include state-like constructs (for the registers, the memory, ...) plus control-flow-like constructs (`endCycle`) to capture the control-flow change in case of an exception (see subsection 3.2) raised in the middle of the semantics of a function (an early return).

The enumeration type `SourceType` helps distinguish three different modes in which a processor might access memory: memory accesses to page tables and so forth triggered implicitly (`VirtualMemory`), implicit reads to instruction memory via the program counter (`Fetch`), and reads or writes that a machine-code program explicitly requests (`Execute`). As for why this distinction is needed, one example is in our exploration of possible executions under weak memory (subsection 3.6), where explicit program memory accesses get sophisticated treatment, implicit accesses to instruction memory get simplified treatment (they are resolved against fixed program code), and implicit virtual-memory accesses are simply disallowed.

Missing axioms. One use of our semantics is verification of properties that hold for multiple (even infinitely many) different possible instantiations of this type class. In that way, we are able to establish *metatheorems* that hold across applications of the semantics, a goal hard to achieve when new applications are supported with new code produced by ad-hoc translators. However, for now, we do not support proving deep properties of our semantics for any possible instance of this type class, rather imposing additional restrictions on instances where needed. The reason is that we are not using *axiomatic type classes* [Wenzel 1997] that additionally assert logical axioms that any instance must validate. For instance, one might hope that, for any valid instance, getting the value of a register that we just wrote should return the value we have just written.

The trouble is that what counts as "reasonable" becomes much murkier very quickly, as we consider other aspects of the semantics. Foundational research questions remain around e.g. memory models in the presence of multisize accesses, virtual memory, self-modifying code, etc. These

```
-- Indicates which stage is the source of a memory access
data SourceType = VirtualMemory | Fetch | Execute
-- Type class providing the RISC-V primitives:
class (Monad p, MachineWidth t) => RiscvMachine p t | p -> t where
  getRegister :: Register -> p t
  setRegister :: Register -> t -> p ()
  getFPRegister :: FPRegister -> p Int32
  setFPRegister :: FPRegister -> Int32 -> p ()
  loadByte :: SourceType -> t -> p Int8
  loadHalf :: SourceType -> t -> p Int16
  loadWord :: SourceType -> t -> p Int32
  loadDouble :: SourceType -> t -> p Int64
  storeByte :: SourceType -> t -> Int8 -> p ()
  storeHalf :: SourceType -> t -> Int16 -> p ()
  storeWord :: SourceType -> t -> Int32 -> p ()
  storeDouble :: SourceType -> t -> Int64 -> p ()
  makeReservation :: t -> p ()
  checkReservation :: t -> p Bool
  clearReservation :: t -> p ()
  getCSRField :: CSRField -> p MachineInt
  unsafeSetCSRField :: (Integral s) => CSRField -> s -> p ()
  getPC :: p t
  setPC :: t -> p ()
  getPrivMode :: p PrivMode
  setPrivMode :: PrivMode -> p ()
  commit :: p ()
  endCycle :: forall z. p z
  flushTLB :: p ()
  fence :: MachineInt -> MachineInt -> p ()
  getPlatform :: p Platform
```

Fig. 2. The primitives of the abstract RISC-V monad

questions may come to be sufficiently settled to allow a complete-enough set of axioms to be included in our type class.

Instantiation. The abstract monad `p` (of kind `* -> *`) can be instantiated differently by each use case, which keeps our spec agnostic to the concrete state of the machine and to the kinds of effects that instructions can have. For instance, depending on the platform and the use case, an invocation of the `storeWord` primitive could update the memory of the machine state, or it could fail if the address is outside of the physical address range, or it could record constraints in a memory-model graph, or it could record an I/O event if the address is in a range that the platform uses for memory-mapped I/O, etc., and our specification is completely agnostic to these options.

The abstract type `t` is the type of the values stored in the integer registers. It can be instantiated with `Int32`, `Int64`, or left abstract for use cases where it makes sense to reason about all bitwidths at once. Requiring a `MachineWidth` type-class instance for `t` guarantees that there are arithmetic and logical operators for `t`. To distinguish `t` from helper integer values that do not live in registers,

we introduce an additional integer type `MachineInt`, which is an alias for `Int64`, and whose more-significant bits are sometimes ignored. In fact, whenever we need an n-bit integer (with $n \leq 64$) that does not live in a register, we use `MachineInt`, applying bitmasking where necessary. In this way, we dodge use of dependent bitvector types indexed by width, as provided by DSLs like Sail. The tradeoffs on this question may also be debated, but at least representation is eased in general-purpose languages with rich but not dependent types, and our case studies will demonstrate that a variety of tasks in programming and formal methods remain tractable with laxer help from typing.

Instructions. The above choice also shows up in our algebraic datatype of instructions:

```
data InstructionI =
  Sw { rs1 :: Register, rs2 :: Register, simm12 :: MachineInt } |
  Add { rd :: Register, rs1 :: Register, rs2 :: Register } |
  Beq { rs1 :: Register, rs2 :: Register, sbimm12 :: MachineInt } | ...
```

For instance, even though the offset field of the store-word instruction is only a signed 12-bit immediate, we represent it with a (64-bit) `MachineInt` for simplicity. Note that this simplification does not compromise correctness, because the specification only creates instructions in the decoder, which only ever writes 12-bit values into that field.

Decode. The decoder starts by defining symbolic names for notable bitfields of the instruction `inst` being decoded, using the function `bitSlice x i j` to extract bits $[$`i`, `j`$)$ of `x`:

```
opcode = bitSlice inst 0 7
rd = bitSlice inst 7 12
rs1 = bitSlice inst 15 20
rs2 = bitSlice inst 20 25
simm12 = signExtend 12 $ shift (bitSlice inst 25 32) 5 .|. bitSlice inst 7 12 ...
```

and then defines a decoder for each RISC-V extension:

```
decodeI
  | opcode==opcode_STORE, funct3==funct3_SW = Sw rs1 rs2 simm12
  | opcode==opcode_BRANCH, funct3==funct3_BEQ = Beq rs1 rs2 sbimm12 ...
```

and finally, checks if the decoded instruction is part of the supported extensions.

Execute. For each RISC-V extension supported, there is an `execute` function that expresses the effects of each instruction of the extension in terms of the primitives listed in Figure 2. For instance, here is the definition of the jump-and-link-register instruction, for which explanatory prose will be provided shortly thereafter:

```
execute (Jalr rd rs1 oimm12) = do
  x <- getRegister rs1
  pc <- getPC
  let newPC = (x + fromImm oimm12) .&. (complement 1)
  if (remu newPC 4 /= 0)
    then raiseExceptionWithInfo 0 0 (fromIntegral newPC)
    else (do
      setRegister rd (pc + 4)
      setPC newPC)
```

Readers who are familiar with Haskell might find it useful to know that the signature of this function is `execute :: forall p t. (RiscvMachine p t) =>InstructionI -> p ()`, but our spec is

intended also to be understandable for casual readers who just treat it as a funny form of almost natural language and run it on small litmus tests to resolve any doubts about corner cases.

The above definition was transcribed from the following English definition:

> The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register $rs1$, then setting the least-significant bit of the result to zero. The address of the instruction following the jump ($pc + 4$) is written to register rd. Register `x0` can be used as the destination if the result is not required.
> The JAL and JALR instructions will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary.

The mechanized specification uses Haskell's **`do`** notation to chain monadic operations, and it can also use standard Haskell constructs such as **`let`** or **`if`**. The binary operators (such as `+`, `/=` and `.&.` in this example) are provided through the `MachineWidth` type class, which takes a type parameter `t` that can be instantiated with 32-bit or 64-bit integers depending on the desired bitwidth. It inherits from Haskell's standard type classes `Integral` and `Bits`, allowing us to use the standard infix operators. `MachineWidth` also provides `fromImm` to convert *immediates* from instructions into register values `t`.

Note that **`let`** and **`if`** are Haskell's standard constructs rather than custom or abstract primitives, enabling readers familiar with these common constructs to study the specification with confidence. As a tradeoff, this use of standard Haskell syntax requires language translators to parse the Haskell code – just instantiating our `RiscvMachine` type class with another instance would not suffice.

We also note that `raiseExceptionWithInfo` is defined on top of the primitives, setting appropriate status registers and then initiating early exit.

Run. Finally, we define what it means to run one instruction. In Coq, we use a simplified[3] version that just fetches an instruction, decodes and executes it, and updates the program counter, whereas the Haskell version also considers interrupts, exceptions, and virtual memory.

Use-case- and platform-specific code. The components described so far form the specification and are grouped together in a directory called `Spec`. However, we have not yet defined how state is represented and how the primitives of Figure 2 are to be implemented. These use-case- and platform-specific definitions are in a separate directory called `Platform` and are the subject of the next section. The `Spec` and `Platform` directories each are about three thousand lines of code.

3 DIFFERENT MONADS FOR DIFFERENT USE CASES

Our RISC-V specification benefits from using a monad (the `p` in Figure 2) in the very same way as Wadler's interpreter in his classic "The Essence of Functional Programming" paper [Wadler 1992]. In this section, we tour the wide variety of applications that can be connected to this single semantics. The key claim we emphasize throughout is that **each application is supported simply by picking new type-class instance(s)** and perhaps also by calling general-purpose translators from Haskell – no new translators or even parsers specific to ISA descriptions were required.

We begin with a brief description of mundane use cases in interpretation (essentially the original application of monads that concerned Wadler). Next, we spend the bulk of our discussion on different flavors of interactive proofs with Coq, before wrapping up with case studies of automated model-checking of both machine-code programs and RISC-V processors using domain-specific tools.

[3]but still accurate with respect to real software and hardware, because both the processor and the compiler are proven to respect this same simplified specification

3.1 Simulation

Support **fast interpretation of machine-code programs** by choosing efficient but harder-to-reason-about data structures.

The most common way of modeling processors in the formal-methods world is to consider the machine to be deterministic, each cycle updating the state of the registers and the memory depending on the instruction present in memory at the location pointed to by the program counter.

Concretely, we wrote two instances of the `RiscvMachine` type class, named `Minimal32` and `Minimal64`, to obtain a 32- and a 64-bit *machine simulator*. We instantiated the type class in the I/O monad, using references and arrays to implement registers, program counter, and memory. In subsection 3.3, we summarize our experiments running specific binaries, where we can run about 100k instructions per second.

3.2 Supporting RISC-V Exceptions

Support **exceptions** by using a variant of the classic monad transformer for failure.

Many formal-methods-oriented projects do not want to deal with exceptions or interrupts, while others are interested in modeling and leveraging them. The formal-spec user should experience the costs of those further features incrementally as they are brought into play.

Raising an exception involves two components: modifying a bunch of state (namely numerous special state registers [CSRs]) and early exit from the main code path that computes a cycle's effect. In RISC-V, exceptions can be caused by virtual-memory translation failures, failed system privilege checks, and alignment problems, among other causes.

To be able to write monadic values carrying this early-exit information, we encode the early return in a layer of a `MaybeT` monad transformer. The crucial primitive already appeared in our definition of the `RiscvMachine` monad.

```
endCycle :: forall z. p z
```

Conspicuously, an implementation of `endCycle` is missing (meaning use triggers a Haskell-level "unimplemented" exception) from the base machine previously described as `Minimal64/32`, but there is one given in source file `Machine.hs`:

```
instance (RiscvMachine p t) => RiscvMachine (MaybeT p) t where
  getRegister r = lift (getRegister r)
  setRegister r v = lift (setRegister r v)
  ...
  endCycle = MaybeT (return Nothing) ...
```

This instance demonstrates something powerful about our spec: composability. It takes some existing instance of the same type class and builds on it, adding in the functionality of the `Maybe` monad. In that monad, a computation halts as soon as a step returns `Nothing`, precisely capturing the "early-return" behavior we want upon encountering an exception.

With this two-layer specification, we ran in simulation the `riscv-tests` test suite (`rv64mi`, `rv64si`, `rv64ui`, `rv64ua`), which is the standard community-maintained test suite.

3.3 Platform Modeling, MMIO, and Devices

Support **I/O features** by providing additional monad transformers.

We actually use this kind of instance augmentation repeatedly in our code, first to encode the semantics of core features (like RISC-V exceptions, as we have just seen) but also to add features like memory-mapped I/O devices to existing `RiscvMachine` instances.

For example, we enrich the platform with a concurrently running memory-mapped device: a UART connected to a terminal, generating interrupts received by the main loop of the simulator.

With this implementation, composed of three layers of specifications, we were able to run Linux in January 2019 at about 100k instructions per second. (We have not tracked newer Linux versions.)

While this strategy allows us to experiment, test, and vouch for our good coverage of the spec, this artifact is not especially competitive for running significant RISC-V programs. For instance, the popular QEMU runs at several hundred million instructions per second, but this performance gap is not surprising, because QEMU uses dynamic binary translation to map the RISC-V instructions to those of the host machine, whereas we run an unoptimized interpreter implemented in Haskell.

3.4 Interactive Theorem Proving

3.4.1 Translation from Haskell. Using hs-to-coq [Breitner et al. 2018], we can translate the Haskell specification to Coq, replacing designated Haskell library functions with corresponding Coq library functions. Since hs-to-coq was designed to model Haskell semantics in Coq as faithfully as possible, it ships with handwritten and auto-generated translations of Haskell's standard-library files, and by default they are referenced by the Coq files produced by hs-to-coq. However, for this project, we were not seeking a faithful reproduction of Haskell semantics in Coq but rather an idiomatic RISC-V specification in Coq. Therefore, we used hs-to-coq's *edit files* feature, which allows one to provide renaming and rewriting patterns to be applied during the translation, so that we could map all Haskell standard-library references to reasonably close Coq equivalents and obtain an idiomatic, Haskell-independent Coq specification. We used hs-to-coq to translate the files specifying instruction execution, the instruction decoder, as well as the CSR-file specification, while we manually wrote remaining files like utility definitions, the definition of the `RiscvMachine` type class, and proof-specific files.

3.4.2 Use as the Interface Between Software and Hardware. Our RISC-V Coq specification was used in a project [Erbsen et al. 2021] that combines a compiler-correctness proof with a processor-correctness proof. The combined theorem states that the I/O trace produced by the processor matches the one produced by the source program fed to the compiler, without referencing the RISC-V specification any more. Thus, auditors of the system can know the behavior of the system without having to audit whether both the compiler and the processor interpret the RISC-V specification in the same way, which greatly reduces the auditing burden.

3.4.3 Opting Out of Features and Opting Back In.

Support **starting formal-verification projects with non-RISC-V-compliant simplifications** and gradually becoming more realistic and compliant.

Our first version of the translation to Coq was driven by the requirements of the compiler-correctness project mentioned above, which required a very simple and manageable spec to get started, so it was decided that initially, CSRs should not be modeled. However, this also meant that we could not use the real `raiseException` function, nor the `translate` function (translating virtual to physical addresses), which starts by reading a CSR that indicates whether virtual memory is enabled. The solution was surprisingly simple: Since we had already chosen manual translation of the file containing the declaration of the `RiscvMachine` type class, we were free to abstract over `raiseException` and `translate` by adding them to the primitives of `RiscvMachine` (Figure 2). That is, we made our specification *more configurable* than RISC-V allows, and for the compiler, we instantiated `translate` to the identity function and `raiseException` to hard failure, because no instructions emitted by the compiler rely on exceptions, while at the same time, we kept open the possibility to instantiate these two functions with (more) real definitions.

Later, when we added CSRs to the Coq specification, we wrote a simplified `raiseException` function. Since the compiler does not use it, it was trivial to integrate this update with the proof.

3.4.4 Simulator in Coq.

Support **interpretation within a proof assistant** by developing similar instantiations to the earlier simulator case, using more naive purely functional data structures. A failure monad provides convenient opportunities to indicate unsupported language features, such that individual programs must then be proved not to exercise those features.

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

State monad with failure. An arguably simpler monad instantiation is `p := OState MachineState`, where `OState(S A: Type) := S → (option A) * S` uses a `None` answer to indicate that a failure occurred. For compiler-correctness proofs, the always-failing function `fail_hard` can be used to indicate that a situation occurred that the compiler is supposed to avoid, e.g. memory access at an invalid address, and a compiler-correctness proof then states that all valid source programs are translated to RISC-V programs that never fail.

Moreover, if the compiler has been designed to emit code that does not use certain features, the RISC-V specification can be simplified by implementing the primitives of Figure 2 used by these features as just `fail_hard`.

3.4.5 Nondeterminism.

Support **nondeterministic execution** by choosing a monad that associates executions with mathematical sets of results (a possibility not available directly in Haskell).

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

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

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

3.4.6 Runtime Input.

Support **input and output** by combining nondeterminism with extra state that records traces of interactions.

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

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

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

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

3.4.7 *Nondeterminism by Means of Weakest Preconditions.*

Support **smooth integration with Hoare-logic-style program verification** by first assigning programs meanings in the style of interaction trees and then applying recursive functions (like weakest-precondition computation) to those trees.

The Bedrock2 compiler [Erbsen et al. 2021] using our RISC-V specification requires RISC-V semantics that given an initial state `s`, a monadic computation `m` corresponding to the execution of a sequence of primitives from Figure 2, and a desired *postcondition*, returns the weakest precondition that must hold in order for the postcondition to hold. Therefore, it seems that we need the following bridge definition that tells when a monadic `OStateND` computation satisfies a postcondition:

```
mcomp_sat S A (m: OStateND S A) s post := ∀o, m s o → ∃a s', o = Some (a,s') ∧ post a s'
```

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

When instantiating `m` with a monadic computation involving many `Binds`, unfolding `mcomp_sat` and all `Binds` quickly leads to huge formulas with one existential for each intermediate state and answer. We found these formulas to be larger than what human brains can deal with productively. To fix the problem, it seems desirable to define `mcomp_sat` directly for each primitive. We can do so using a different instantiation of our `RiscvMachine` type class that materializes monadic computations into an **`Inductive`** with a constructor for each primitive from Figure 2, with an alternative definition of `mcomp_sat` that gives the weakest-precondition interpretation of this syntax. This monad is similar to freer monads [Kiselyov and Ishii 2015] and interaction trees [Xia et al. 2020].

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

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

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

3.5 Multiplication in Software: Reasoning About Multiple Instantiations of our Spec

Support **proving connections between semantics variants (e.g. standing for capabilities of different conformant processors)** by simply instantiating the semantics with different type-class instances and mentioning the different instantiations in single theorem statements and proofs.

Gruetter et al. [2023] use two instantiations of our RISC-V spec in the same proof, showing that a processor without support for the multiplication instruction, but with a trap handler that catches invalid-instruction exceptions and implements multiplication in software, behaves as if multiplication were implemented in hardware.

3.6 Model Checking with Weak Memory Models

Support **model-checking of all possible program executions under weak memory** by choosing a type-class instance to record information on alternative paths to try later.

In this section we sketch our approach to instantiate the type class to generate all outcomes of small multicore litmus tests with respect to the memory model. As far as we know, the state-of-the-art algorithm to compute all outcomes specified by the RISC-V weak memory model is the algorithm proposed by Kokologiannakis and Vafeiadis [2020]. It is a fairly sophisticated algorithm interleaving constraint-solving phases and phases that add and remove nodes in an execution graph. Its details are not relevant for the point we are trying to make in our paper, which is that our type-class structure is flexible enough to cover such a drastically different use case.

The algorithm revolves around 4 data structures:

- a control/data/addr dependency-bookkeeping data structure, to maintain a list of all the memory events that imply dependencies on the currently interpreted instruction
- a current partial execution graph, which is the graph of the memory events and their memory-model relations
- two bookkeeping data structures necessary for backtracking during search: a list of alternative partial execution graphs to explore later and a maintained set of all the read events that would be subject to revisiting, if a store to the same address would occur

Intuitively, our implementation goes as follow: we write an interpreter in charge of exploring an execution path depth-first. That interpreter also records all the alternative decisions that it could have taken on its way. The interpreter can either return successfully with a valid execution, or it can return that the execution that it explored ended up violating the memory model. In both cases, the interpreter updates the global bookkeeping of alternative executions.

We implement this model-checking interpreter by instantiating the `RiscvMachine` type class in the I/O monad. We use references to track state associated with the model-checking algorithm. We modify the `Minimal64` machine described in subsection 3.1 and add the partial-execution graph and bookkeeping data structures described above. The `RiscvMachine` instance implements dependency tracking in the `loadWord`, `storeWord`, and `fence` primitives; other primitives are implemented similarly to the `Minimal64` simulator. The complete implementation is 800 lines of Haskell.

Each generated graph is checked against the upstream official Alloy specification [Lustig 2018] for RISC-V's RVWMO memory model. We only support word load and store instructions plus a TSO fence, omitting the original work's features for atomics and release/acquire fences.

As is standard in memory-model work, we assume the instruction memory is separate from the data memory and not under the memory-model exploration. We also do not support mixed-size or misaligned accesses, virtual memory is deactivated, and there are no exceptions/interrupts.

We ran our model checker on the litmus tests provided by Flur et al. [2019], which were also used by the RISC-V Memory Model Task Group during development of the ISA specification. We are able to run all of the basic 2-thread litmus tests, where wall-clock times range from approximately 20 seconds for the smallest test cases to 3 minutes for the largest; we are able to run all 36 test cases in 50 minutes. Tests were run on a lightly loaded machine with a Haswell i7-5930K CPU and 64GB of DDR4 RAM running Arch Linux. The performance bottleneck here is calling Alloy by spawning a new Java Virtual Machine per query and probably also suboptimal Alloy parameters.

3.7 Model-Checking the Decode and Execute Functions

Support **compilation to hardware circuits** by carefully tweaking interpreter-style instances to avoid unbounded types.

The riscv-formal project [Wolf 2018] proposes a Verilog description of the state-update function for one cycle of RISC-V execution. They also have infrastructure to model-check their Verilog description against other descriptions. As our final case study, we decided to connect our specification to that ecosystem, validating it against the existing riscv-formal specification. Their tooling works directly on Verilog code, so it was convenient to translate our specification into that language. We used the Haskell-to-hardware converter Clash [QBayLogic 2020] to transform our specification, using a minimal state-monad instance, into a Verilog function, and the authors of riscv-formal model-checked that output against their reference riscv-formal to find discrepancies (as revealed by executing any single instruction from matching initial states).

Interestingly enough, the Clash instance of the specification is quite similar to the `Minimal` instances. However, the `Minimal` instances are not directly usable in Clash because they use a `Map` for the register file, and these potentially arbitrary-sized maps do not normalize well in Clash. Instead, we use the `Vector` datatypes in Clash. We also fought with instabilities of Clash's partial evaluation of programs, where sometimes it would fail to notice dead code or otherwise take advantage of predictability of some code spans. As a result, we did make a few lines of changes to our spec just to avoid some trouble with Clash.

However, there is one snippet consisting of 7 lines of code involving conditionals over the instruction set `iset`, which is statically known to be `RV32I`, and lists, statically known to be singletons, on which Clash runs into a normalization black hole before propagating these statically known constants enough to avoid it. It is probably possible to improve Clash's normalization so that this case is handled properly, but we currently shamelessly use a sed script to comment the offending lines.

This snippet highlights a classical example of a tradeoff: if one considers extensions and bitwidths to be statically fixed in one machine, using macros instead of **`if`** expressions would have sidestepped this problem altogether. On the other hand, having one spec able to handle bitwidths 32 and 64 simultaneously was a requirement for our users writing a generic RISC-V compiler and proving it generically. We decided having to replay all the proofs twice would have been worse.

4 RELATED WORK

Most recent work on multipurpose ISA specs has employed domain-specific languages toward ends similar to ours. The Sail [Armstrong et al. 2019; Mundkur et al. 2020] language is the highest-profile today for defining ISA semantics. Sail has a few features that make defining extensible ISAs more pleasant:

- Dependently typed bitvectors can help catch specification bugs.
- Open variant types allow to spread the cases of an algebraic data type (tagged union) over several files, and functions that consume them can also be spread over several files. This feature is very useful for the data type representing instructions: There is no need to say upfront what all the possible instructions are, and extensions can freely extend the data type. In Haskell, on the other hand, we need a two-layer algebraic data type: The first layer says to which extension an instruction belongs, and the second layer says which instruction it is. And in Haskell, the decode function needs to be a big monolithic function, whereas in Sail, each extension can define its own encode/decode mappings.
- `mapping` clauses can be used to specify bijections between the instruction AST and its encoding as 32-bit integers as well as its encoding as an assembly string, and from these declarative mappings, Sail automatically derives encoding and decoding functions.
- Specifying which extensions and/or custom instructions are used is done in the Makefile, by passing or not passing the files in question to the Sail translator. This approach leads to greater flexibility in composing a custom, non-RISC-V-compliant, simplified specification that only supports certain instructions. In contrast, our approach defines these choices within Haskell (or in Coq, respectively), leading to a little less flexibility.

A DSL in this category requires new tools to translate into specification languages required across use cases, and languages in this tradition had not previously allowed application-agnostic semantics to be first-class objects in logics.

Another DSL specific to the ARM ISA family, ASL [Reid 2016], received a lot of attention recently, thanks to systematic adoption for several of ARM's most important ISA variations. It can be translated automatically to Sail, and Sail can be translated to HOL4. However, the resulting HOL4 spec is too cumbersome to deal with directly in a compiler-correctness proof, so the CakeML compiler writers prefer proving their compiler against a simpler specification written in L3 [Fox 2012], Sail's predecessor. Kanabar et al. [2022] proved that the L3-based specification simulates the official Arm specification in ASL and combined this proof with the CakeML compiler's correctness proof, resulting in a compiler-correctness proof against an official formal specification of an industrial ISA.

These DSLs have been used for encoding several significant mainstream ISAs beyond RISC-V, and indeed it is possible that our approach would be less appropriate for legacy ISAs with complications and baggage beyond what we had to deal with in RISC-V.

Fox and Myreen [2010] defined and validated an ARM semantics in HOL4, using a fixed monad for state-threading. They also performed extensive validation across a few use cases, most notably in automated testing against processors.

Goel and Hunt [2013] developed a detailed model of x86 in ACL2. It supports two different use cases [Goel 2016, Section 5.2]: Efficient execution and formal reasoning in ACL2. To support both of them with the same model, they rely on an ACL2 feature called *Single-Threaded Objects*, or `stobj` for short [Boyer and Moore 2002]. The formal semantics of a `stobj` update are copy-on-write, but ACL2 syntactically enforces that a `stobj` is always used linearly, i.e. there is at most one reference to it, which allows the simulator to perform destructive updates without changing the semantics.

Several past projects have applied single ISA semantics to verify both hardware and software: the CLI stack [Bevier et al. 1989], VeriSoft [Alkassar et al. 2008], and CakeML [Lööw et al. 2019]. All three dealt only with bespoke, verification-motivated ISAs and only performed verification within the proof assistants their specifications were written in. In addition to its hardware-verification-motivated ISA called Silver, the CakeML compiler also supports compilation to ARMv6, ARMv8, MIPS-64, RISC-V, and x86-64, but no end-to-end proofs combining software and hardware were

made using these industrial ISAs. They have been specified using the L3 language [Fox 2012], Sail's predecessor. For each of these target ISAs, the L3 model defines (among other things) a `state` datatype, an `instruction` type, and a function **`next`** from `state` to `state` [Fox et al. 2017, Section 5].

Sail ISA semantics have also recently been connected to program proof with Islaris [Sammler et al. 2022], a framework that uses SMT-based symbolic execution of the semantics to produce more manageable verification conditions, to be discharged in Coq. Compared to our approach, at least one new trusted language translator is involved. The SMT solver is also trusted, rather than using it to generate proofs checkable by Coq. Islaris has been applied successfully to both RISC-V and more complex ARM ISAs.

5 CONCLUSION

We have presented a new approach to formal specification of hardware instruction sets, relying on type classes for easy instantiation to different use cases, thus avoiding any requirement for domain-specific language features. As a result, such a semantics can be written directly in popular general-purpose languages and translated to other forms using tools that are not domain-specific. Our example is for the up-and-coming RISC-V ISA family and has been applied across hardware and software and across different styles of formal methods, without requiring that a single new parser/translator be written to integrate with tools backed by interactive proof (Coq), relational model finding (Alloy), and SMT-based model checking (Yosys).

Some rough edges certainly remain, which should be considered in comparing our approach to that of domain-specific languages, motivating the search for new coding patterns to achieve better modularity in general-purpose languages. We would rather not have our `RiscvMachine` type class always contain the `getFPRegister` and `setFPRegister` primitives. It would be better if they were only present if the floating-point extension is supported, and similarly for `makeReservation`, `checkReservation`, and `clearReservation` required by the atomics extension. Such more fine-grained control over which primitives are needed could probably be obtained using a hierarchy of type classes, but we preferred to avoid this additional complexity.

The instruction-decode function is quite large, and every property about it that we proved (or tried to prove) in Coq leads to performance issues that require very careful, performance-aware proof engineering.

We did a few experiments trying to translate the specification to other languages, but it turned out to require more engineering effort than originally expected. Parsing Haskell is only the first step, and after that, unless the target language supports type classes the same way as Haskell does, the translator would have to perform semantic analysis to resolve the meanings of overloaded operators like `+` or `fromIntegral`: They sometimes refer to operations on the integer type `t` (that can be 32-bit or 64-bit), while in other locations, they refer to machine-width-independent operations.

Nonetheless, this approach allowed for a small arsenal of nontrivial tools to be constructed fairly quickly, and related mechanizations may have a place in future formal-methods ecosystems.

ACKNOWLEDGMENTS

We would like to thank Rishiyur Nikhil for feedback on this paper as well as on the development of our semantics over the years, alongside the other members of the ISA-semantics working group of the RISC-V Foundation. This research was supported by the National Science Foundation Expedition on the Science of Deep Specification (award CCF-1521584) and the Defense Advanced Research Projects Agency (award HR001118C0018). Part of this work was carried out while the non-MIT authors had appointments at MIT.

A DATA-AVAILABILITY STATEMENT

An artifact [Bourgeat et al. 2023] associated with this paper was evaluated and is freely available.

REFERENCES

Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. 2008. The Verisoft Approach to Systems Verification. In *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08) (LNCS, Vol. 5295)*, Natarajan Shankar and Jim Woodcock (Eds.). Springer, 209–224.

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–31. https://doi.org/10.1145/3290384

William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. 1989. An Approach to Systems Verification. *Journal of Automated Reasoning* (1989), 411–428. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.6467&rep=rep1&type=pdf

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 174–184. https://doi.org/10.1145/289423.289440

Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andy Wright, and Adam Chlipala. 2023. A RISC-V Formal Semantics in Haskell. https://doi.org/10.5281/zenodo.7992509

Robert S. Boyer and J Strother Moore. 2002. Single-Threaded Objects in ACL2. In *Practical Aspects of Declarative Languages*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Shriram Krishnamurthi, and C. R. Ramakrishnan (Eds.). Vol. 2257. Springer Berlin Heidelberg, Berlin, Heidelberg, 9–27. http://link.springer.com/10.1007/3-540-45587-6_3

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-Coq to Real-World Haskell Code (Experience Report). *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 89:1–89:16. https://doi.org/10.1145/3236784

Andres Erbsen. 2022. An End-to-End Verified Garage-Door Opener. https://github.com/mit-plv/fiat-crypto/blob/master/src/Bedrock/End2End/X25519/GarageDoor.v

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification Across Software and Hardware for a Simple Embedded System. *PLDI'21* (2021). https://doi.org/10.1145/3453483.3454065

Shaked Flur, Luc Maranget, and Peter Sewell. 2019. Litmus Test for the RISC-V Memory Model. https://github.com/litmus-tests/litmus-tests-riscv

Anthony Fox. 2012. Directions in ISA Specification. In *Interactive Theorem Proving (Lecture Notes in Computer Science)*, Lennart Beringer and Amy Felty (Eds.). Springer, Berlin, Heidelberg, 338–344. https://doi.org/10.1007/978-3-642-32347-8_23

Anthony Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. 2017. Verified Compilation of CakeML to Multiple Machine-Code Targets. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 125–137. https://doi.org/10.1145/3018610.3018621

Anthony C. J. Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Interactive Theorem Proving (ITP)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer, 243–258.

Shilpi Goel. 2016. *Formal Verification of Application and System Programs Based on a Validated X86 ISA Model.* Thesis. University of Texas at Austin. https://repositories.lib.utexas.edu/handle/2152/46437

Shilpi Goel and Warren A. Hunt. 2013. Automated Code Proofs on a Formal Model of the X86. In *Verified Software: Theories, Tools, Experiments (Lecture Notes in Computer Science)*, Ernie Cohen and Andrey Rybalchenko (Eds.). Springer, Berlin, Heidelberg, 222–241. https://doi.org/10.1007/978-3-642-54108-7_12

Samuel Gruetter. 2021. A Model of an OpenTitan Root-of-Trust System Running Hardware Accelerators and Their C/Bedrock2 Device Drivers. https://github.com/project-oak/silveroak/blob/main/firmware/RiscvMachineWithCavaDevice/Bedrock2ToCava.v

Samuel Gruetter, Thomas Bourgeat, and Adam Chlipala. 2023. *Proving That a System with Software Trap Handlers for Unimplemented Instructions Behaves as If They Were Implemented in Hardware.* Technical Report. https://samuelgruetter.net/assets/softmul.pdf

Hrutvik Kanabar, Anthony C. J. Fox, and Magnus O. Myreen. 2022. Taming an Authoritative Armv8 ISA Specification: L3 Validation and CakeML Compiler Verification. In *13th International Conference on Interactive Theorem Proving (ITP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 20:1–20:22. https://doi.org/10.4230/LIPIcs.ITP.2022.20

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. ACM, Vancouver BC Canada, 94–105. https://doi.org/10.1145/2804302.2804319

Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 1157–1171. https://doi.org/10.1145/3373376.3378480

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *In Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*.

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. *PLDI'19* (2019), 13.

Daniel Lustig. 2018. A Formalization of the RVWMO (RISC-V) Memory Model. https://github.com/daniellustig/riscv-memory-model

Prashanth Mundkur, Rishiyur Nikhil, Jon French, Brian Campbell, Robert Norton, Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher Pulte, and Peter Sewell. 2020. Sail RISC-V Model. (2020). https://github.com/rems-project/sail-riscv

QBayLogic. 2020. Clash. https://clash-lang.org/

Alastair Reid. 2016. Trustworthy Specifications of ARM® V8-A and v8-M System Level Architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 161–168. https://doi.org/10.1109/FMCAD.2016.7886675

Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: Verification of Machine Code against Authoritative ISA Semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 825–840. https://doi.org/10.1145/3519939.3523434

Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

P. Wadler and S. Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

Andrew Waterman and Krste Asanovic (Eds.). 2019a. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213. *RISC-V Foundation* (Dec. 2019). https://riscv.org/technical/specifications/

Andrew Waterman and Krste Asanovic (Eds.). 2019b. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified. *RISC-V Foundation* (June 2019). https://riscv.org/technical/specifications/

Markus Wenzel. 1997. Type Classes and Overloading in Higher-Order Logic. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '97)*. Springer-Verlag, Berlin, Heidelberg, 307–322.

Claire Wolf. 2018. RISC-V Formal Verification Framework. Symbiotic EDA. https://github.com/SymbioticEDA/riscv-formal

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

Received 2023-03-01; accepted 2023-06-27